Andrea De Lucia Filomena Ferrucci (Eds.)

# Software Engineering

International Summer Schools
ISSSE 2009-2011
Salerno, Italy
Revised Tutorial Lectures

Springer

Volume Editors

Andrea De Lucia
Filomena Ferrucci

Università di Salerno
Via Ponte don Melillo
84081 Fisciano (SA), Italy

E-mail:
adelucia@unisa.it
fferrucci@unisa.it

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36053-4 e-ISBN 978-3-642-36054-1
DOI 10.1007/978-3-642-36054-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955373

CR Subject Classification (1998): D.2, D.1, K.6.3, F.3, H.3-4, J.1, I.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

This volume collects chapters originating from some tutorial lectures given at the 2009, 2010, and 2011 editions of the International Summer School on Software Engineering (ISSSE). Beginning in 2003, ISSSE is an annual meeting point that aims to provide a contribution on some of the latest findings in the field of software engineering, which is an exciting, stimulating, and profitable research area with significant practical impacts on software industry. ISSSE contributes to training future researchers in this field and to bridging the gap between academia and industry, thereby facilitating knowledge exchange. Indeed, it is intended for PhD students, university researchers, and professionals from industry, and the format of the school aims at creating extensive discussion forums between lecturers and industrial and academic attendees.

Attracting about 60 participants each year, the program of the school includes state-of-the-art tutorials given by internationally recognized research leaders on very relevant topics for the scientific community. Each tutorial provides a general introduction to the chosen topic, while also covering the most important contributions in depth and identifying the main research challenges for software engineers. The focus is on methods, techniques, and tools; in some cases theory is required to provide a solid basis. Besides traditional tutorials, student talks and tool demos are also included in the program to further stimulate interaction.

This volume is organized in three parts, collecting chapters focused on Software Measurement and Empirical Software Engineering, Software Analysis, and Software Management.

Empirical software engineering has established itself as a research area aimed at building a body of knowledge supported by observation and empirical evidence. Thus, in the last few years there have been a great amount of empirical studies in software engineering meant to assess methods and techniques, identify important variables, and develop models. Measurement plays a key role in these observations and in empirical work as well as in a variety of practical goals in software engineering including production planning, monitoring, and control. Thus, to make the correct conclusions and to avoid wasting resources, it is crucial to use software measures that make sense for the specified goal. To this aim, a theoretical and empirical validation process should be carried out meant to prove that a measure really measures the attribute that it is supposed to and that it is practically useful. Starting from this observation, in the first chapter, Sandro Morasca illustrates the foundational aspects of software measurement needed for the theoretical validation process. He describes the notions of measurement theory for internal and external software attributes, the definition of properties for software measures through axiomatic approaches, and the proposal of a goal-oriented process that can be followed to carry out the definition and empirical validation of measures.

In Chap. 2, Martin Shepperd first examines the history of empirical software engineering and overviews different meta-analysis methods meant to combine multiple results. Then, he describes the process of systematic reviews as a means for systematically, objectively, and transparently locating, evaluating, and synthesizing evidence to answer a particular question. Finally, he identifies some future directions and challenges for researchers.

Chapter 3 focuses on software fault prediction, a field that has generated a great deal of research in the last few decades. The research aims to provide methods for identifying the components of a software system that most likely will contain faults and is motivated by the need to improve the efficiency of software testing, which is one of the most expensive phases of software development. Indeed, knowing in advance the potentially defective components, project managers can better decide how to allocate resources to test the system, concentrating their efforts on fault-prone components, thus improving the dependability, quality, and cost/effectiveness of the software product. In this chapter, Thomas J. Ostrand and Elaine J. Weyuker survey the research they have carried out in the last ten years in the context of software fault prediction.

In Chap. 4, Pollock et al. illustrate the main aspects of natural language program analysis (NLPA) that combines natural language processing techniques with program analysis to extract information for analysis of the source program. The research is motivated by the aim to improve the effectiveness of software tools that support program maintenance exploiting natural language clues from programmers' naming in literals, identifiers, and comments. The authors summarize the state of the art and illustrate how NLPA has been used to improve several applications, including search, query reformulation, navigation, and comment generation. An analysis of future directions in preprocessing, analysis, and applications of NLPA concludes the chapter.

In chap. 5, Andrian Marcus and Sonia Haiduc present and discuss the applications of text retrieval (TR) techniques to support concept location, in the context of software change. Concept location starts with a change request and results in the identification of the starting point in the source code for the desired change. In systems of medium or large size, developers need to be supported by suitable tools during the concept location task. To develop such tools, many different approaches exist that rely on different kinds of information. The most recent and most advanced techniques rely on TR methods. The authors provide an overview of the TR-based concept location techniques and their evolution, discuss their limitations, and identify future research directions.

In chap. 6, Andrea Zisman provides an overview of existing approaches for service discovery, an important activity in service-oriented computing meant to identify services based on functional, behavioral, quality, and contextual characteristics. Moreover, the author describes a framework that supports both static and dynamic identification of services. The static process can help the development of service-based systems at design-time by identifying the services that match specified characteristics. The dynamic process is used during runtime execution of a system to support the replacement of a service.

In Chap. 7, Frank Maurer and Theodore D. Hellmann give an overview of agile software development processes and techniques focusing the discussion on project management and quality assurance. They describe project planning with special attention to iteration planning and interaction design approaches, and illustrate agile quality assurance with a focus on test-driven development and the state space of testing.

In the last chapter, Frank van der Linden focuses on software product line engineering and describes some experiences of introducing software product line engineering in industry. He analyzes some problems originating from the distributed organization of companies and illustrates how practices from open source software development may be used to address the problems mentioned.

We wish to conclude by expressing our gratitude to the many people who supported the publication of this volume with their time and energy. First of all, we wish to thank the lecturers and all the authors for their valuable contribution. We also gratefully acknowledge the Scientific Committee members, for their work and for promoting the International Summer School on Software Engineering. We are also grateful to Gabriele Bavota, Vincenzo Deufemia, Sergio Di Martino, Fausto Fasano, Rita Francese, Carmine Gravino, Rocco Oliveto, Ignazio Passero, Abdallah Qusef, Michele Risi, Federica Sarro, and Giuseppe Scanniello, who were of great help in organizing the school. Finally, we want to thank Springer, for giving us the opportunity to publish this volume and all the staff involved.

We hope you will enjoy reading the chapters and find them relevant and fruitful for your work. We also hope that the tackled topics will encourage your research in the software engineering field and your participation in the International Summer School on Software Engineering.

January 2012

Andrea De Lucia
Filomena Ferrucci

# Table of Contents

# Fundamental Aspects of Software Measurement

Sandro Morasca

Università degli Studi dell'Insubria
Dipartimento di Scienze Biomediche, Informatiche e della Comunicazione
I-22100 Como, Italy
sandro.morasca@uninsubria.it

**Abstract.** Empirical studies are increasingly being used in Software Engineering research and practice. These studies rely on information obtained by measuring software artifacts and processes, and provide both measures and models based on measures as results. This paper illustrates a number of fundamental aspects of Software Measurement in the definition of measures that make sense, so they can be used appropriately. Specifically, we describe the foundations of measurement established by Measurement Theory and show how they can be used in Software Measurement for both internal and external software attributes. We also describe Axiomatic Approaches that have been defined in Software Measurement to capture the properties that measures for various software attributes are required to have. Finally, we show how Measurement Theory and Axiomatic Approaches can be used in an organized process for the definition and validation of measures used for building prediction models.

**Keywords:** Software Measurement, Measurement Theory, internal software attributes, external software attributes, Axiomatic Approaches, GQM.

## 1 Introduction

Measurement is an essential part in every scientific and engineering discipline and is a basic activity in everyday life. We use measurement for a variety of goals, by acquiring information that we can use for developing theories and models, devising, assessing, and using methods and techniques, and making informed and rational practical decisions. Researchers use measurement to provide evidence for supporting newly proposed techniques or for critically assessing existing ones. Industry practitioners use measurement for production planning, monitoring, and control, decision making, carrying out cost/benefit analyses, post-mortem analysis of production projects, learning from experience, improvement, etc. Final consumers use measurement to make sensible decision.

Thus, it is not surprising that measurement is at the core of many engineering disciplines, and the interest towards measurement has been steadily growing in Software Engineering too. However, Software Engineering differs from other engineering disciplines in a number of aspects that deeply affect Software Measurement.

- Software Engineering is a relatively young discipline, so its theories, methods, models, and techniques still need to be fully developed, assessed, consolidated, and improved. The body of knowledge in Software Engineering is still limited, if compared to the majority of engineering disciplines, which have been able to take advantage of scientific models and theories that have been elaborated over the centuries. These models and theories were built through a process that required 1) the identification of a number of fundamental concepts (e.g., length, mass, time, and electrical charge in Physics), 2) the study of their characteristics, 3) the investigation of how they are related to each other by means of theories and models, and 4) how they can be measured by collecting data from the field so theories and models can be validated and used.
- Software Engineering is a very human-intensive discipline, unlike the engineering branches that are based on the so-called hard sciences (e.g., Physics, Chemistry). One of the main tenets of these sciences is the repeatability experiments and their results. This is hardly ever the case in a number of interesting Software Engineering studies. For instance, it is clearly impossible to achieve repeatability when it comes to developing a software product. So, it is virtually impossible that given the same requirements and given two teams of developers with the same characteristics working in environments with the same characteristics, we obtain two identical values for the effort needed to develop a software application. Thus, Software Measurement models, theories, methods, and techniques will be different from those of the hard sciences and will probably not have the same nature, precision, and accuracy. As a matter of fact, some aspects of Software Measurement are more similar to measurement in the social sciences than measurement in the hard sciences.
- Software Engineering deals with artifacts that are mostly immaterial, like software code, test suites, requirements, etc., for which there is no real, direct experience. For instance, we are much more acquainted with the concept of length of a material object than the concept of "complexity" of a software program.

Given these specific aspects of Software Engineering, and the practical impact that Software Measurement can have on Software Engineering practices, we here investigate two basic questions, similar to those that are frequently quoted for Software Verification and Validation.

- Are we measuring the attributes right? When using or defining a measure, it is necessary to make sure that it truly quantifies the attribute it purports to measure. As mentioned above, Software Engineering has not yet reached the same level of maturity of other engineering disciplines, nor has Software Measurement, so this theoretical validation is a required activity for using or defining measures that make sense. Theoretical validation is also a necessary step not only for the empirical validation of software measures, but, even more importantly, for empirically validating Software Engineering techniques, methods, and theories. For instance, empirically validating the claim that the maintainability of a software system decreases if the coupling between its modules increases requires that one use sensible measures for coupling and maintainability. Given the lack of intuition about software product attributes, theoretical validation is not a trivial activity, as it involves formalizing intuitive ideas around which a widespread consensus still needs to be built.

– Are we measuring the right attributes? Measuring attributes that are irrelevant for our goals would clearly be useless. So, we need to select the right ones, given the available resources. The best way to provide evidence that an attribute is relevant for some specified goal is to use a sensible measure for that attribute and carry out an experimental or empirical study by showing that, for instance, it can be used for predicting some software product or process attribute of interest. This activity entails the empirical validation of the measure.

By not answering these two questions satisfactorily, we could end up with measures that are useless and waste the resources used for measurement. In an even worse scenario, we could actually make incorrect decisions based on inappropriate measures, which could even be detrimental to the achievement of our goals, in addition to wasting the resources used for measurement.

In the remainder of this paper, we describe the foundational aspects of Software Measurement, by using the two main approaches available in the related literature, namely Measurement Theory and Axiomatic Approaches. Beyond their mathematical appearance, both approaches are actually ways to formalize common sense, so as to reach a reasonable compromise between the principles of rigor and common sense that should rule in applied research. We describe these two approaches in the remainder of this paper to show their usefulness and potential, along with their strengths and weaknesses. In addition, we illustrate a goal-oriented process that can be followed to carry out the definition and empirical validation of sensible measures. This process also shows how the theoretical validation of measures can be used to prevent irrelevant measures or attributes from being used in prediction models.

We mostly address fundamental issues in the measurement of attributes of software artifacts in this paper. So, our primary focus here is on the theoretical validation of measures for attributes of software artifacts. The measurement of process attributes is somewhat more straightforward. Take design effort as an example of a software process attribute. The real challenge in this case is the prediction of the effort required to carry out software design, not its measurement. Software design effort prediction requires the identification of appropriate measures that can be quantified before design takes place and the availability of a quantitative model based on those measures. Software design effort measurement, instead, simply requires the identification of which activities belong to software design, the people who work on those activities, and the use of accurate collection tools for the effort used for software design. Thus, software design effort measurement is mostly concerned with a number of details which, albeit important, do not compare to the issues related to the building of a good prediction model.

The remainder of this paper is organized as follows. Section 2 introduces a few basic concepts and the terminology that is used in the paper. The use of Measurement Theory for the intrinsic attributes of software artifacts is described in Section 3, while Axiomatic Approaches are introduced in Section 4. A unified axiomatic approach for the description of a number of interesting intrinsic attributes of software artifacts is described in more detail in Section 5. Section 6 describes Probability Representations, which are a part of Measurement Theory that has received little attention in Software Measurement so far, and which can be used for providing firm foundations for all of those software artifact attributes that are of practical interest for software stakeholders. Section 7 illustrates GQM/MEasurement DEfinition Approach (GQM/MEDEA), a process for the definition of measures for prediction models that integrates the fundamental aspects of Software Measurement into a coherent, practical approach. Conclusions and an outline of future work on fundamental aspects of Software Measurement follow in Section 8.

## 2 A Few Basic Concepts

Here, we introduce a few basic concepts and terminology that will be used throughout the paper.

In Software Measurement, like in any kind of measurement, one measures the attributes of entities. It does not make sense to measure an "entity," without mentioning which specific attribute of that entity we would like to measure. It does not make much sense to "measure a car," for instance. It is necessary to specify whether we are measuring its length, width, height, weight, number of seats, maximum speed, etc. By the same token, it does not make much sense to measure a software program, unless we specify which particular attribute we have in mind. There is a plethora of different attributes that have been introduced in Software Measurement: size, complexity, cohesion, coupling, connectivity, usability, maintainability, readability, reliability, effort, development time, etc., to mention a few. Since there are so many of them, it is important to understand the nature of these attributes and identify their similarities and differences. In the long term, it would be useful to come to a generalized agreement about these fundamental concepts of Software Measurement so that everybody uses and understands the same concepts in the same way. For terminology consistency, we use the term attribute in this paper, instead of other terms that have been used in the past for the same concept, including quality, characteristic, subcharacteristic, factor, criterion.

Conversely, it does not make sense to measure an attribute without mentioning the entity on which it is measured. For instance, it does not make sense to measure the number of seats without referring to a specific car. In Software

Measurement, it would not make sense to measure "complexity" without specifying whose program it is. Software Measurement is used on a number of different entities that belong to the following two categories:

– software products and documents, e.g., source code, executable code, software design, test suites, requirements; we use the term "software artifact" to refer to any such entity;
– activities carried out during software production process, e.g., coding, deployment, testing, requirements analysis; we use the term "software process" to refer to any such entity.

A typical distinction in Software Measurement is made between internal and external attributes of entities.

– Internal attributes of software artifacts, such as size, structural complexity, coupling, cohesion, are usually said to be as those attributes of an entity that *can be measured* based only on the knowledge of the entity [15]. So, internal attributes are easy to measure: for instance, the size of a program is often measured by counting the number of its lines of code.
– External attributes, such as reliability, performance, usability, maintainability, portability, readability, testability, understandability, reusability, are characterized as those attributes that *cannot be measured* based only on the knowledge of the software artifact. Their measurement involves the artifact, its "environment," and the interactions between the artifact and the environment. For instance, the maintainability of a software program depends on the program itself, the team of people in charge of maintaining the program, the tools used, etc. The maintainability of a given program is likely to be higher if maintenance is carried out by the same people that developed the program than by other programmers. So, the knowledge of the program alone is not sufficient to quantify its maintainability.

From a practical point of view, external software attributes are the ones related to and of direct interest for the various categories of software "users," e.g.: the compiler/interpreter that translates a program; the computer on which it runs; the final user; the practitioners. These "users" may have different and possibly conflicting needs. The ensemble of the attributes that are relevant to the users completely describes what is known as the quality of a software product. Therefore, external attributes are the ones that have true industrial interest and relevance. However, because of their very nature, external attributes are in general more difficult to define and quantify than internal ones, as they require that a number of factors be taken into account, in addition to the software artifact [15,36]. On the other hand, internal attributes are much easier to quantify. However, they have no real interest or relevance *per se*. The measurement of an internal attribute of a software artifact (e.g., the size of a software design) is interesting only because it is believed or it is shown that the internal attribute is linked to: (1) some external attribute of the same artifact (e.g., the maintainability of the software design) or of some other artifact (e.g., the fault-proneness of the software code); (2) some attribute of the software process (e.g., the effort needed to develop the software design). So, one typically measures internal software attributes to assess or predict the value of external software attribute.

One final terminology clarification is in order before we start reviewing various fundamental aspects of Software Measurement. The term "metric" has been often used instead of "measure" in the Software Measurement and Software Engineering in the past. As it has been pointed out, "metric" has a more specialized meaning than "measure." The term "metric" is closely related to distance and it typically implies the presence of some unit of measurement. As we explain in the remainder of this paper, this is not necessarily the case, so the more general term "measure" is preferable. Therefore, we consistently use "measure" in the remainder of this paper.

## 3 Measurement Theory for Internal Software Attributes

The foundations of Measurement Theory were established by Stevens [38] in the 1940s, as a way to provide the mathematical underpinnings for measurement in the social and human sciences. It has been used in other disciplines and its concepts have been extended and consequences have been assessed since. Measurement Theory is now a quite well-established field. The interested reader can refer to [20,34] for more complete introductions to the subject. In Empirical Software Engineering, Measurement Theory has almost exclusively been used with reference to the measurement of the attributes of software artifacts, such as size, structural complexity, cohesion, coupling.

### 3.1 Basic Notions of Measurement Theory

The first and most important goal of Measurement Theory is to make sure that measures have properties that make them comply with intuition. So, Measurement Theory [20,34] makes clear that measuring is not just about numbers, i.e., assigning measurement values to entities for some attribute of interest. For instance, it would make little sense to have a software size measure that tells us that a program segment is longer than another program segment when we look at those two segments and conclude that it should actually be the other way round.

Beyond all the mathematics involved, Measurement Theory shows how to build a sensible, common sense bridge, i.e., a measure (Definition 3), between

- our "intuitive," empirical knowledge on a specified attribute of a specified set of entities, via the so-called Empirical Relational System (Definition 1), and
- the "quantitative," numerical knowledge about the attribute, via the so-called Numerical Relational System (Definition 2), so that
- the measure makes sense, i.e., it satisfies the so-called Representation Condition (Definition 4).

We now explain these concepts and we use the size of a set of program segments as an example to make these definitions more concrete.

**Definition 1 (Empirical Relational System).** *Given an attribute, let*

- *$E$ denote the set of entities for which we would like to measure the attribute*
- *$R_1$,..., $R_y$ denote $y$ empirical relations capturing our intuitive knowledge on the attribute: each $R_i$ has an arity $n_i$, so $R_i \subseteq E^{n_i}$; we write $(e_1, \ldots, e_{n_i}) \in R_i$ to denote that tuple $(e_1, \ldots, e_{n_i})$ is in relation $R_i$; if $R_i$ is a binary relation, we use the infix notation $e_1 R_i e_2$*
- *$o_1$, ..., $o_z$ denote $z$ empirical binary operations on the entities that describe how the combination of two entities yields another entity, i.e., $o_j : E \times E \rightarrow E$; we use an infix notation, e.g., $e_3 = e_1 o_j e_2$.*

*An Empirical Relational System is is an ordered tuple*

$$ERS = (E, R_1, \ldots, R_y, o_1, ..., o_z)$$

For instance, suppose we want to study the size of program segments. We typically have

- the set of entities $E$ is the set of program segments
- *longer_than* $\subseteq E \times E$, an empirical binary relation that represents our knowledge that, given any two program segments $e_1$ and $e_2$ for which $e_1$*longer_than*$e_2$, $e_1$ has a greater size than $e_2$
- a concatenation operation, i.e., $e_3 = e_1; e_2$.

Other attributes of the same set of entities, e.g., complexity, will have different kinds and sets of empirical relationships and operations than size has. This is due to the fact that we have different intuitions about different attributes of a set of entities.

No numbers or values are found in the Empirical Relational System, which only takes care of modeling our own empirical intuition. Measurement values are introduced by the Numerical Relational System, which we define next

**Definition 2 (Numerical Relational System).** *Given an attribute, let*

- *$V$ be the set of values that we use to measure the attribute*
- *$S_1$,..., $S_y$ denote $y$ relations on the values: each $S_i$ has the same arity $n_i$ of $R_i$*
- *$\bullet_1$, ..., $\bullet_z$ denote $z$ numerical binary operations on the values, so each $\bullet_j$ has the form $\bullet_j : V \times V \rightarrow V$; we use an infix notation, e.g., $v_3 = v_1 \bullet_j v_2$.*

*A Numerical Relational System is an ordered tuple*

$$NRS = (V, S_1, \ldots, S_y, \bullet_1, ..., \bullet_z)$$

Even though it is called Numerical Relational System, We have chosen to represent $V$ as a set of "values" and not necessarily numbers for greater generality and because in some cases numbers are not really needed and may even be misleading (e.g., for nominal or ordinal measures as described later in this section). In our segment size example, we can take

– $V = Re_{0+}$, the set of nonnegative real numbers, which means that the values of the size measures we use are nonegative real numbers
– a binary relation '>', which means that we want to use the natural ordering on those measurement values (so we can translate "longer_than" into '>' and back as the Representation Condition will mandate)
– a binary operation '+', which means want to be able to sum the sizes of segments (the Representation Condition will actually mandate that we sum the sizes of concatenated program segments).

The Numerical Relational System is purposefully defined to mirror the Empirical Relational System in the realm of values, even though the Numerical Relational System in itself does not predicate about the entities and the specific attribute investigated.

The connection between the Empirical Relational System and the Numerical Relational System, and thus, entities and values, is made via the concept of measure (Definition 3).

**Definition 3 (Measure).** *A function $m : E \rightarrow V$ is said to be a measure.*

However, there is more to the Empirical Relational System than just the set of entities on which it is based. The Empirical Relational System also gives information about what we know about an attribute of a set of entities. If that knowledge is not taken into account, any $m \in V^E$ is a measure, i.e., any assignment of values to program segments may be a measure, according to Definition 3. Given program segments $e_1, e_2, e_3$ such that $e_1 longer\_than e_2$ and $e_2 longer\_than e_3$, a measure $m$ according to Definition 3 may be very well provide values of $m(e_1)$, $m(e_2)$, and $m(e_3)$ such that $m(e_1) < m(e_2)$ and $m(e_3) < m(e_2)$, though this does not make sense to us. Measurement Theory introduces the Representation Condition (Definition 4) to discard all of those measures that contradict our intuition and keep only the fully sensible ones.

**Definition 4 (Representation Condition).** *A measure must satisfy the two conditions*

$$\forall i \in 1 \ldots n, \forall (e_1, \ldots, e_{n_i}) \in E^{n_i} (e_1, \ldots, e_{n_i}) \in R_i \Leftrightarrow (m(e_1), \ldots, m(e_{n_i})) \in S_i$$
$$\forall j \in 1 \ldots m, \forall (e_1, e_2) \in E \times E (m(e_1 o_j e_2) = m(e_1) \bullet_j m(e_2))$$

The Representation Condition translates into the following two conditions for our segment size example

– $e_1 longer\_than e_2 \Leftrightarrow m(e_1) > m(e_2)$, i.e., our intuition on the ordering of the program segments is mirrored by the ordering of the measurement values, and *vice versa*,

- $m(e_1;e_2) = m(e_1) + m(e_2)$, i.e., the size of a program segment obtained by concatenating two program segments is the sum of the sizes of the two program segments.

A sensible measure is defined as a scale (Definition 5) in Measurement Theory.

**Definition 5 (Scale).** *A scale is a triple* $(ERS, NRS, m)$*, where* $ERS$ *is an Empirical Relational System,* $NRS$ *is a Numerical Relational System, and* $m$ *is a measure that satisfies the Representation Condition.*

In what follows, we assume for simplicity that measures satisfy the Representation Condition, so we use the terms "scale" and "measure" interchangeably, unless explicitly stated.

So now, given an Empirical Relational System and a Numerical Relational System, we would need to find out if we can actually build a measure. However, the existence of a measure will depend on the specific Empirical Relational System and a Numerical Relational System, and we will not illustrate the issues related to the existence of a measure in detail. Rather, we investigate whether more than one legitimate measure may be built, given an Empirical Relational System and a Numerical Relational System. This should not come as a surprise, since it is well known from real life that we can quantify certain attributes of physical objects by using different equally legitimate measures. For instance, the length of a segment may be quantified equally well by meters, centimeters, yards, feet, inches, etc. We know that we can work equally well with one measure or another and that one measure can be translated into another by means of a multiplicative factor. For notational convenience, we denote by $M(ERS, NRS)$ the set of scales that can be defined based on an Empirical Relational System $ERS$ and a Numerical Relational System $NRS$.

However, the very existence of a set of equally good measures shows something a little bit more surprising: the bare value of a measure in itself does not provide a lot of information. For instance, saying that the length of a wooden board is 23 does not mean much, unless one specifies the unit of measurement. Clearly, talking about a 23 inch wooden board is not the same as talking about a 23 meter one. Introducing the concept of unit of measurement actually means taking one object as the reference one and then assigning a measurement value to all other objects as the times the other objects possess that specified attribute with respect to the reference object. So, if we say that a wooden board is 23 inches long, all we are saying is that it is 23 times longer than some wooden board that we took as the one that measures 1 inch. What is even more important is that this 23:1 ratio between the length of these two wooden boards is the same no matter the measure used to quantify their length, be it inches, feet, meters, etc. This is true for the ratios of the lengths of any pair of wooden boards, i.e., these ratios are invariant no matter the scale used. Invariant properties of scales are called meaningful statements and provide the real information content of a scale, as they do not depend on the conventional and arbitrary choice of one specific scale.

**Definition 6 (Meaningful Statement).** *A statement $S(m)$ that depends on a measure $m$ is* meaningful *if its truth value does not change across all scales, i.e.,* $\forall m \in M(ERS, NRS)(S(m)) \vee \forall m \in M(ERS, NRS)(\neg S(m))$.

So, a statement that is true with one scale is also true with all other scales, and one that is false with one scale is also false with all other scales. Choosing one scale instead of another basically means adopting a convention, because this choice does not affect the truth value of meaningful statements, like saying that a wooden board object is 23 times as long as another. Instead, suppose we can tell if a software failure is more critical than another on a 5-value criticality measure $cr'$. For instance, suppose that those five values are $\{1, 2, 3, 4, 5\}$, from 1 (least severe) to 5 (most severe). It is typically meaningless to say that criticality 2 failures are twice as severe as a criticality 1 failure, as the truth value of this statement depends on the specific choice of values. If we choose another scale $cr''$ with values $\{7, 35, 38, 981, 4365\}$, the truth value of the statement changes, as we would say that the failures in the second category are five times more severe than the ones in the first category. Still, the failures in the second category are given a value that is higher than the value for the first category. The ordering is preserved, and that is where the real information content of the scale lies. This piece of information is preserved by applying a monotonically increasing scale transformation, not just a proportional one. On the contrary, the length of wooden boards cannot undergo any monotonically increasing scale transformation, but only proportional transformations. So, different scales may be subject to different kinds of transformations without any loss of information, which are called admissible transformations.

**Definition 7 (Admissible Transformation).** *Given a scale $(ERS, NRS, m)$, the transformation of scale $f$ is admissible if $m' = f \circ m$ (i.e., $m'$ is the composition of $f$ and $m$) and $(ERS, NRS, m')$ is a scale.*

Actually, proportional transformations can be used for a number of different scales (e.g., the typical scales for weight), while monotonically increasing transformations can be used for all sorts of ranking. Measurement Theory identifies five different kinds of scales based on five different kinds of transformations scales can undergo while still preserving their meaningful statements. We now list these different kinds of scales in ascending order of the information they provide.

**Nominal Scales.** The values of these scales are labels–not necessarily numbers–for categories in which the entities are partitioned, with no notion of order among the categories. Their characterizing invariant property states that the actual labels used do not matter, as long as different labels are used for different categories. Formally, $\forall e_1, e_2 \in E$

$$\forall m \in M(ERS, NRS)(m(e_1) = m(e_2)) \vee \forall m \in M(ERS, NRS)(m(e_1) \neq m(e_2))$$

As the partitioning of the entities into the categories is the information that needs to be preserved, nominal scales can be transformed into other nominal scales via one-to-one transformations.

The programming language in which a program is written is an example of a nominal scale, i.e., we can associate the labels (i.e., values of the scale) $C$, $Java$, $COBOL$, etc. with each program. As long as programs written in the same language receive the same label and program written in different languages receive different labels, we can adopt programming language names like *alpha*, *beta*, *gamma*, etc.; or $Language1$, $Language2$, $Language3$, etc.; or 1, 2, 3, etc. Note that we do not need to use numbers as values of the measure. Actually, it would be meaningless to carry out even the simplest arithmetic operations. In the Numerical Relational System it is obviously true that "$1 + 2 = 3$," but that would become something nonsensical like $C + Java = COBOL$ just by choosing a different legitimate scale.

As for descriptive statistics, it is well known that the mode (i.e., the most frequent value) is the central tendency indicator that should be used with nominal measures, even though there may be more than one mode in a sample. The arithmetic average cannot be used, as it cannot even be computed, since arithmetic operations are barred. As a dispersion indicator, one may use the Information Content $H(f)$ computed by taking the frequencies of each value as their probabilities, i.e.,

$$H(f) = -\sum_{v \in V} f(v) \log_2 f(v)$$

where $f(v)$ is the frequency of value $v$.

Association statistical methods can be used too with nominal measures. For instance, suppose that we would like to find a software component that we may want to reuse in your software system and that there are a number of functionally equivalent candidate software components we can choose from and the only information we have is the programming language they are written in. Suppose also that we want to select the software component that has the lowest defect density, but we do not have that piece of information. If defect density data about components are available, association statistical methods like those based on chi-square tests can be used to find out how much we can rely on components written in different languages. So, even though information about defect density is not available and our measure (the programming language) does not involve any number, we can still make an informed and statistically sensible decision.

However, nominal measures only allow the classification of entities into different categories. A nominal measure for the size of program segments could only tell if two segments have the same size or not, but it would not provide any information on whether one program segment is larger or smaller than another one.

**Ordinal Scales.** In ordinal scales, the entities are partitioned into categories, and the values of these scales are *totally ordered* labels. Their characterizing invariant property states that the actual labels used do not matter, as long as the order of the values that label different categories is preserved. Formally, $\forall e_1, e_2 \in E$

$$\forall m \in M(ERS, NRS)(m(e_1) > m(e_2)) \vee$$
$$\forall m \in M(ERS, NRS)(m(e_1) = m(e_2)) \vee$$
$$\forall m \in M(ERS, NRS)(m(e_1) < m(e_2))$$

As the ordering across the categories is the piece of information that needs to be preserved, ordinal scales can be transformed into other scales via strictly monotonic transformations.

Roughly speaking, ordinal scales are like nominal scales for which, in addition, an ordering on the categories has been defined. Because of the existence of this additional property to be preserved when one scale is transformed into another scale, not all of the possible one-to-one transformations can be used. So, the set of admissible transformations for ordinal scales is a subset of the set of admissible transformations for nominal scales. This reduces the degree of arbitrariness in choosing a scale, and makes an ordinal scale more information-bearing than a nominal scale, because ordinal scales give information about the ordering of the entities and not just their belonging to classes. In our program segment size example, ordinal scales allow us to tell if a program segment is of greater length than another, not just that the two segments have different sizes.

A good example of an ordinal measure is failure criticality, as defined in many bug tracking systems, in which it is possible to associate a criticality value with each bug (for example, in SourceForge, bugs may be a ranked on a nine value scale). Like with nominal scales, it is possible to use numbers as values of ordinal scales, and it is even more tempting than with nominal scales to use arithmetic operations on these numbers. However, this would not be correct, as it would lead to meaningless results. Suppose here that we use an ordinal scale with five values 1, 2, 3, 4, and 5 under the usual numerical ordering. Alternatively, we could have used $A$, $B$, $C$, $D$, and $E$, with the usual alphabetical ordering, so $A$ is least severe and $E$ is most severe. While obviously $1+2=3$ in the mere realm of numbers, an operation like that would translate into something like $A+B=C$ if we adopt the alphabetical labels. This statement is meaningful if and only if we can actually say something like "the presence of a bug at criticality $A$ and one at criticality $B$ are equivalent to the presence of a bug at criticality $C$," under some notion of equivalence. However, this is an additional piece of information, that cannot be inferred from what we know about the mere ordering of the entities in any way. So, this is not a meaningful statement. To have an additional proof of this, let us transform the scale into another numerical scale with values 10, 15, 20, 25, 30. The corresponding statement would become $10+15=20$, which is clearly false. So, transforming a scale into another scale makes the truth value of the statement change, i.e., the statement is not meaningful. The real point here is that we know if one failure is more or less critical than another failure, but we have no idea by how much.

So, one may very well use numbers as values of an ordinal scale, but the kind of mathematical manipulations that can be made must be limited to using $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$. As a consequence it is not allowed to compute the arithmetic average or the standard deviation of a sample of ordinal values.

As for descriptive statistics, the median is the central tendency indicator of choice for ordinal measures. The median of a sample is defined as that value *med* in the sample such that less than half of the data points have values less than *med* and less than half of the data points have values greater than *med*. If the median is not unique, there may be at most two medians in a sample, and they have consecutive values. At any rate, since ordinal scales may be seen as specializations of nominal scales, the descriptive statistics of nominal scales can be applied to ordinal scales too.

Association statistical methods can be used too with ordinal measures. For instance, suppose that we need to find whether failure criticality is statistically related to the effort needed to solve a bug. For instance, suppose we would like to find out if it is true that bugs with higher criticality also take more time to be fixed. Suppose also that this bug fixing effort is measured on an ordinal scale, because the effort collection system allows software engineers to enter values in classes of values like “less than one hour,” “between one and four hours,” “between four hours and one work day,” “between one workday and one work week,” and “more than one work week.” Statistical indicators are available to investigate this association. For instance, one can use Spearman’s $\rho$ or Kendall’s $\tau$ [19], which provide a measure of the strength of the increasing or decreasing association between two ordinal variables (failure criticality and bug fixing effort, in our example). Also, statistical tests are available to check how statistically significant the associations are. These indicators do not assume that the two variables are linked by any specific functional form (e.g., linear). On the positive side, they can be used to investigate whether there is *any* increasing or decreasing association between two variables. On the negative side, it is not possible to build a specific estimation model, because an estimation model would be based on some functional form that links the two variables. Again, the association statistics for nominal scales can be used with ordinal scales as well.

Summarizing, by using a nominal or an ordinal scale, we can have information about an attribute of a set of entities, but we do not need to use any numbers. The following kinds of scales will require the use of numbers and will provide more refined information about an attribute of a set of entities.

**Interval Scales.** In interval scales, each entity is associated with a numerical value. Their characterizing invariant property states that the actual values used do not matter, as long as the ratios between all pairs of differences between values are preserved. Formally, by denoting the set of positive real numbers by $Re_+$, $\forall e_1, e_2, e_3, e_4 \in E$

$$\exists k_1, k_2 \in Re_+, \forall m \in M(ERS, NRS) k_1(m(e_1) - m(e_2)) = k_2(m(e_3) - m(e_4))$$

An interval scale $m'$ can be transformed into another interval scale $m''$ only via linear transformations $m'' = am' + b$, with $a > 0$, i.e., we can change the origin of the values (by changing $b$) and the unit of measurement (by changing $a$). Linear transformations are a subset of strictly monotonic transformations, which are the admissible transformations for ordinal measures. So, again, this reduces the number of possible measures into which an ordinal measure can be transformed, and, again, this makes interval scales even more information-bearing than ordinal scales.

Typical examples of interval scales are calendar time or temperature measured with the scales ordinarily used to this end. For instance, take the Celsius scale for temperatures. It is well known that the origin is conventionally established as the temperature at which water freezes under the pressure of one atmosphere. Also, the 100 Celsius degree mark is conventionally established as the temperature at which water boils under the pressure of one atmosphere. These conventional choices determine the (thereby conventional) extent of one Celsius degree. In addition, it is well known that Celsius degrees can be transformed into, say, Fahrenheit degrees, by means of the following linear transformation relationship

$$Fahrenheit = \frac{9}{5}Celsius + 32$$

It is easy to see that, in addition to the meaningful statements that can be made for ordinal scales, interval scales allow us to make statements in which the ratios of the differences between measurement values are preserved.

Not many software measures are defined at the interval level of measurement. The most important one is probably calendar time, which, for instance, is used during the planning or monitoring of a project. However, the importance of interval scales is that numbers are truly required as measurement values. For one thing, it would not be possible to carry out the linear transformations, otherwise. Some arithmetic manipulations are possible, as shown in the definition. For instance, subtraction between two values of an interval measure provides a result that makes sense. The difference between two dates, e.g., the end and the beginning of a software project, obviously provide the project's duration (which is actually a ratio scale, as we explain in later in this section). Nevertheless, not all possible arithmetic manipulations can be used. It would not make much sense to sum two dates for two events, e.g., May 28, 2005 and July 14, 2007, for instance. Also, if today's Celsius temperature is 20 Celsius degrees, and yesterday's was 10 Celsius degrees, it does not make any sense to say that today is twice as warm as yesterday, as can be easily shown by switching to Fahrenheit temperatures. So, taking the ratio of two interval measure values does not make sense.

Nevertheless, it is meaningful to compute the average value of an interval measure, even though averages are built by summing values if we are interested in comparing two average values. Suppose that the average of the values of a sample is greater than the average of the values of another sample when we use an interval measure. It can be shown that this relationship holds for any other interval scale chosen. So, the average is a good central tendency indicator for interval scales. At any rate, the same holds true for the medians, which can be clearly used for interval scales, which can be seen as a subset of ordinal scales. New dispersion statistics can be added to those "inherited" from ordinal scales, e.g., the standard deviation and the variance. As a matter of fact, they provide a metric evaluation of dispersion, unlike the dispersion indicators of nominal and ordinal scales.

As for association statistics, Pearson's correlation coefficient $r$ [19] can be used when interval scales are involved. However, when using statistical significance tests related to $r$, it is important to make sure that the assumptions underlying these tests are satisfied. Otherwise, there is a danger of obtaining results that are not statistically valid. At any rate, one can always resort to the association indicators that can be used with ordinal scales. It is true that some statistical power may be lost when using Spearman's $\rho$ or Kendall's $\tau$ instead of Pearson's $r$, but this loss may not be too high. For instance, it has been computed that the so-called Asymptotic Relative Efficiency of Kendall's $\tau$ with respect to Pearson's $r$ is 0.912. Roughly speaking, from a practical point of view, this means that 1,000 data points are needed to obtain enough evidence to reach acceptance or rejection of a statistical hypothesis on the association between two interval scales by using Kendall's $\tau$ when 912 data points are needed to obtain enough evidence to reach acceptance or rejection of a statistical hypothesis on their correlation. Thus, using Kendall's $\tau$ implies having to collect about 8.8% more data points than we would need with Pearson's $r$. The additional catch is that this value of Asymptotic Relative Efficiency is computed only if the underlying assumptions for using and statistically testing Pearson's $r$ are satisfied. These assumptions may not hold, in practice, so it is usually advisable to use Spearman's $\rho$ and/or Kendall's $\tau$ in addition to Pearson's $r$ when carrying out an analysis of the statistical dependence between two interval variables.

At any rate, with interval scales, we can use most of the traditional statistical indicators, because interval scales are truly numerical scales. The next kind of scales removes one of the degrees of arbitrariness intrinsic to interval scales: the origin is no longer conventional.

**Ratio Scales.** Each entity is associated with a numerical value by ratio scales. Their characterizing invariant property states that the actual values used do not matter, as long as the ratios between all the pairs of values are preserved. Formally, $\forall e_1, e_2 \in E$

$$\exists k_1, k_2 \in Re_+, \forall m \in M(ERS, NRS) k_1 m(e_1) = k_2 m(e_2)$$

So, this property implies that a ratio scale $m'$ can be transformed into another ratio scale $m''$ only via proportional transformations $m'' = am'$, with $a > 0$. This shows that it is possible to change the measurement unit by changing $a$, but not the origin as the value 0 in one scale correspond to the value 0 in all other scales, so it is invariant. The above formula shows that the set of admissible transformations for ratio scales is a subset of the admissible transformations for interval scales, the difference between ratio scales and interval scales basically being that for ratio scales have a natural origin, which is invariant, while a conventional origin can be chosen for interval scales. Ratio scales obviously can only take numerical values, like interval scales.

Size (e.g., volume or mass) is typically represented by ratio scales, for instance. Time durations or temperature intervals may be represented with ratio scales and, in general, the difference between two values of an interval measure is a ratio scale. In Software Engineering Measurement, software size is typically represented via ratio scales and so is development effort.

Legitimate operations involving ratio scales include differences, ratios, and sums. For instance, the size of a program segment composed of two program segments may be obtained as the sum of the sizes of those two program segments. As for descriptive and association statistics, there is basically the geometric mean that can be used with ratio scales, in addition to the other descriptive and association statistics that can also be used with interval scales. From a practical point of view, this shows that there is a real divide between ordinal and interval measures. The former are nonnumeric, while the latter are numerical ones.

**Absolute Scales.** Absolute scales are the most "extreme" kind of measures, in a sense. Each entity is associated with a numerical value in absolute scales, and their invariant property states that the actual values used *do* matter, since the only admissible transformation is identity, i.e., an absolute scale cannot be transformed into anything other than itself. Formally, $|M(ERS, NRS)| = 1$.

Again, this transformation is a subset of the possible transformations of ratio scales. The measurement unit is fixed and cannot be chosen conventionally. So, these scales are the most informative ones, since their values bear information themselves, and not only in relationship.

**Statistics of Scales.** The description of scale types is not simply a theoretical exercise, but it has important practical consequences, as we have already discussed. Some mathematical operations may not be applied to measures of certain measurement levels, e.g., summing may not be used for the numerical values of nominal, ordinal, or even interval measures. Based on the scale type of a measure, different indicators of central tendency can be used without resulting in meaningless statements, i.e., the mode for nominal scales, the median as well for ordinal scales, the arithmetic mean as well for interval scales, and the geometric mean as well for ratio and absolute scales. the same applies to dispersion indicators and statistical tests. Table 1 summarizes a few well-known indicators of central tendency, dispersion, and association that are appropriate for each scale type. In each cell of columns "Central Tendency," "Dispersion," and "Association," we report only the indicators that can be used for scales that are at least on that measurement level. So, these indicators can be used for scales at higher measurement levels. For instance, as already noted, the median can be used for ordinal, interval, ratio, and absolute scales, but not for nominal scales.

## 3.2 Additional Issues on Scales

Two additional issues on scales are often given some more attention, in practical use and in theoretical debates. We briefly discuss them here.

**Table 1.** Characteristics of different scale types

| Scale Type | Admissible Transformation | Examples | Central Tendency | Dispersion | Association |
|---|---|---|---|---|---|
| Nominal | Bijections | Gender, Progr. Language | Mode | Information Content | Chi-square |
| Ordinal | Monotonically increasing | Preference, Fail. Criticality | Median | Interquartile range | Spearman's $\rho$, Kendall's $\tau$ |
| Interval | Linear | Temperature, Milestone Date | Arithmetic Mean | Standard Deviation | Pearson's $r$ |
| Ratio | Proportional | Mass, Software Size | Geometric Mean | | |
| Absolute | Identity | Probability | | | |

**Subjective Scales.** An "objective" measure is one for which there is an unambiguous measurement procedure, so it is totally repeatable. A "subjective" measure is computed via a measurement procedure that leaves room for interpretation on how to measure it, so different people may come up with different measurement values for the same entity, the same attribute, and the same measure itself. It is usually believed that objective measures are always better than subjective measures, but this claim needs to be examined a bit further.

- For some attributes, no objective measure exists. The number of faults in a software program cannot be measured, so we may resort to subjective evaluations for it.
- Even when it is theoretically possible to use an objective measure, it may not be practically or economically viable to use that measure. For instance, the number of faults in a software application with a finite input domain may be measured by executing the application with all possible input values, but this would be impractical.
- Some measures look more objective than they actually are. Take this objective measure of reliability for a software application $a$: $objReliability(a) = 0$ if $a$ has at least one failure in the first month of operation, and otherwise $objReliability(a) = 1$ . Some failures may occur in the first month of operation, but they may go unnoticed, or their effect may surface several months later.
- An objective measure may be less useful than a subjective measure anyway. Take two measures (a subjective and an objective one) for two different attributes. Nothing guarantees that the objective measure is more useful than the subjective one to predict some variable of interest. Even for the same attribute, a subjective measure may be more useful than an objective one, if it captures that attribute more sensibly.

**Indirect Scales.** It is commonly said that measures built by combining other measures are indirect ones. For instance, fault density represented as the ratio between the number of uncovered faults and *LOC* (the number of lines of code of a program segment) would be an indirect measure. However, even among measurement theoreticians, there is no widespread consensus that it is actually necessary or useful to make the distinction between direct and indirect measures, or that it is even possible to make this distinction. One of the points is that even indirect scales should satisfy exactly the same requirements as direct scales, since indirect scales are scales anyway, so they should be built by using an Empirical Relational System, a Numerical Relational System, a function between them, and a Representation Condition. If all of these theoretical definition elements are in place, then there is no reason to distinguish between direct and indirect scales anyway [34].

### 3.3 Evaluation of Measurement Theory

Measurement Theory is the reference, ideal model to which one should tend in the definition of a measure. A measure defined in such a way as to comply with the Representation Condition is a legitimate measure for an attribute. However, Measurement Theory's constraints may be too strict. For instance, *LOC* does not comply with the Measurement Theory's requirements for size measures. Overall, Measurement Theory has been used to eliminate measures that have been proposed for the quantification of software attributes, but it has not been helpful or productive when it comes to defining new measures. Also, other than the modeling of size, no other use of Measurement Theory is known in Software Measurement. Thus, especially in this phase, in which Software Measurement has not reached a sufficient degree of maturity, it is useful to use other approaches like Axiomatic Approaches (see Sections 4 and 5), which have more relaxed requirements and so they do not eliminate a number of measures that Measurement Theory would reject. Measures may therefore get a chance to be better refined later on, and this contributes to having a better understanding of the characteristics of software attributes too.

## 4 Axiomatic Approaches

Other approaches have been used to represent the properties that can be expected of software attributes, e.g., [33,39,21,10,25,31,32,23]. The underlying idea has been long used in mathematics to define concepts via sets of axioms. The axioms for distance are a very well-known example. The distance $d$ between two elements $x$ and $y$ of any set $S$ is defined as a real-valued function $d : S \times S \rightarrow Re$ that satisfies the following three axioms.

**Distance Axiom 1** Nonnegativity. *The distance between any two elements is nonnegative, i.e.,* $\forall x, y \in S(d(x, y) \leq 0)$*, and it is zero if and only if the two elements coincide, i.e.,* $\forall x, y \in S(d(x, y) = 0 \Leftrightarrow x = y)$*.*

**Distance Axiom 2** Symmetry. *The distance between any two elements* $x$ *and* $y$ *is the same as the distance between* $y$ *and* $x$*, i.e.,* $\forall x, y \in S(d(x, y) = d(y, x))$*.*

**Distance Axiom 3** Triangular Inequality. *Given three elements, the sum of the distances between any two pairs is greater than the distance between the other pair of elements, i.e.,* $\forall x, y, z \in S(d(x, y) + d(y, z) \geq d(x, z))$.

These axioms have been applied to very concrete sets, such as sets of points in the physical world, and much more abstract ones, such as sets of functions in mathematics. Different functions that satisfy these axioms can be defined, even within the same application domain. The choice of a specific distance measure depends on a number of factors, including the measurement goals, tools, and resources. No matter the specific application, these three axioms are commonly accepted as the right axioms that capture what a distance measure should look like, so they are no longer a topic for debates, since a broad consensus has been reached about them. Other sets of axioms have been defined for other attributes (e.g., the Information Content $H(p)$ of a discrete probability distribution $p$, which is the basis of Information Theory).

The set of axioms for distance functions is certainly no longer controversial and its introduction is based on the properties of distances between physical points. The set of axioms for Information Content are quite recent and they address a more abstract attribute, but, there is now a widespread consensus about them. The introduction of Axiomatic Approaches in Software Measurement, is even more recent, due to the novelty of Software Engineering and, more specifically, of Software Measurement, which, as already noted, deals with somewhat abstract attributes of intangible entities. Therefore, it is natural that there has not been enough time to reach a broad consensus around specific sets of axioms for software attributes. Nevertheless, one of the main advantages of using an axiomatic approach over using an "operational" approach, i.e., providing a measure as if it was an "operational" definition for the attribute, is that the expected properties of the measures of the attribute are clearly spelled out. Thus, a common understanding of the properties can be reached and disagreements can focus on specific properties instead of more vaguely defined ideas. At any rate, Axiomatic Approaches have already been used to check if existing measures satisfy a specific set of axioms for the attribute that they are supposed to measure. Perhaps more importantly, these approaches have been used as guidelines during the definition of new measures.

Also, it must understood that these Axiomatic Approaches do not have the same "power" as Measurement Theory. Rather, the set of axioms associated with a specific attribute (e.g., software size) should be taken as sets of necessary properties that need to be satisfied by a measure for that software attribute, but not sufficient ones. Thus, those measures that do not satisfy the set of axioms for a software attribute cannot be taken as legitimate measures for that attribute. The measures that do satisfy the set of axioms are candidate measures for that software attribute, but they still need to be better examined. Finally, like with Measurement Theory, the measures that comply with the theoretical validation still need to undergo a thorough empirical validation that supports their practical usefulness. We address this issue in Section 7.

## 4.1 Weyuker's Complexity Axioms

Weyuker's approach [39] represents one of the first attempts to use axioms, to formalize the concept of program complexity. The approach introduces a set of nine axioms, which we number $W1, \ldots, W9$. Weyuker's approach was defined for the complexity of so called "program bodies," which we have called program segments so far. So, the approach was defined for the complexity of sequential programs or subroutines. The composition of program segments is concatenation and it is denoted by ';':$ps_1; ps_2$ denotes the concatenation of two program segments $ps_1$ and $ps_2$.

**W1.** A complexity measure must not be "too coarse" (part 1)

$$\exists ps_1, ps_2 (Complexity(ps_1) \neq Complexity(ps_2))$$

**W2.** A complexity measure must not be "too coarse" (part 2). Given the non-negative number $c$, there are only finitely many program segments of complexity $c$.

**W3.** A complexity measure must not be "too fine." There exist distinct program segments with different complexity

$$\exists ps_1, ps_2 (Complexity(ps_1) = Complexity(ps_2))$$

**W4.** Functionality and complexity have no one-to-one correspondence between them

$$\exists ps_1, ps_2 (ps_1 functionally\_equivalent\_tops_2) \wedge \\ (Complexity(ps_1) \neq Complexity(ps_2))$$

**W5.** Concatenating a program segment with another program segment may not decrease complexity

$$\forall ps_1, ps_2 (Complexity(ps_1) \leq Complexity(ps_1; ps_2)) \wedge \\ (Complexity(ps_2) \leq Complexity(ps_1; ps_2))$$

**W6.** The contribution of a program segment in terms of the overall program may depend on the rest of the program

$$\exists ps_1, ps_2, ps_3 (Complexity(ps_1) = Complexity(ps_2)) \wedge \\ (Complexity(ps_1; ps_3) \neq Complexity(ps_2; ps_3))$$
$$\exists ps_1, ps_2, ps_3 (Complexity(ps_1) = Complexity(ps_2)) \wedge \\ (Complexity(ps_3; ps_1) \neq Complexity(ps_3; ps_2))$$

**W7.** A complexity measure is sensitive to the permutation of statements. There exist $ps_1$ and $ps_2$, such that $ps_1$ is obtained via a permutation of the statements of $ps_2$ and $Complexity(ps_1) \neq Complexity(ps_1)$.

**W8.** A complexity measure is not sensitive to the specific identifiers used. If $ps_1$ is obtained by renaming the identifiers of $ps_2$, then

$$Complexity(ps_1) = Complexity(ps_2)$$

**W9.** There are program segments whose composition has a higher complexity than the sum of their complexities

$$\exists ps_1, ps_2 (Complexity(ps_1) + Complexity(ps_2) < Complexity(ps_1; ps_2))$$

The following analysis of Weyuker's axioms may shed some light on their characteristics and the kind of complexity that they are meant to describe.

– Axioms $W1$, $W2$, $W3$, $W4$, $W8$ do not characterize complexity alone, but they may be applied to all syntactically-based product measures, e.g., size measures. At any rate, they need to be made explicit in an axiomatic approach.
– Axiom $W5$ is a monotonicity axiom which shows that Weyuker's axioms are about "structural" complexity and not "psychological" complexity. Suppose that program segment $ps_1$ is an incomplete program, and the complete program is actually given by the concatenation $ps_1; ps_2$. It may very well be the case that the entire program is more understandable than $ps_1$ or $ps_2$ taken in isolation, as some coding decisions may be easier to understand if the entire code is available.
– Axiom $W7$ shows that the order of the statements does influence complexity. Without this axiom, it would be possible to define a control-flow complexity measure that is totally insensitive to the real control flow itself, as the statements in a program segment could be arbitrarily rearranged without affecting the value of a control-flow complexity measure.
– Axiom $W8$ too shows that Weyuker's axioms are about "structural" complexity, not "psychological" complexity. Renaming does not have any impact on Weyuker's concept of complexity, but it is obvious that, if a program segment's variables were renamed by using meaningless, absurd, or misleading names, the program segment's understandability would be certainly heavily affected, and, in turn, its "psychological" complexity.
– Axiom $W9$ is probably the one that most characterizes complexity, even if it does not come in a "strong" form, since it uses an existential quantification. The idea, however, is that there are cases in which the complexity of a program segment is higher than the sum of the complexities of its constituent program subsegments. This axiom, however, does not rule out the existence of two program segments whose composition has a *lower* complexity than the sum of their complexities

$$\exists ps_1, ps_2 Complexity(ps_1) + Complexity(ps_2) > Complexity(ps_1; ps_2)$$

## 5 A Unified Axiomatic Approach for Internal Software Attributes

We now illustrate the proposal initially defined by Briand, Morasca, and Basili [10,25] and its later refinements by Morasca [23]. This proposal addresses several different software product attributes, including size, complexity, cohesion, and coupling, which we discuss in this section. Based on an abstract graph-theoretic model of a software artifact description of a software artifact, each software attribute is associated with a set of axioms that its measures should satisfy. Thus, unlike in other approaches, a set of different software attributes are studied in a unified framework that makes it easier to identify the similarities and differences between software attributes. In addition, as it is based on an abstract graph-theoretic representation, this axiomatic approach can be applied for measures of many different artifacts that are encountered during the software life cycle, and not just software code.

### 5.1 Systems and Modules

The basic idea is that a system is a multigraph, where each arc is associated with a multiset of relationships, and each relationship has a type.

**Definition 8.** *System. A system $S$ is a pair $S =< E, R >$, where*

- *$E$ represents the set of elements of $S$*
- *$R \in N^{E \times E \times T}$*

*where $T$ is a finite set of types of relationships ($N$ is the set of natural numbers, including 0).*

The idea is that a software artifact contains a set basic elements, which are represented as the nodes of the multigraph. These elements are connected by possibly more than one relationship of possibly different types. The relationships between the elements are therefore represented by the multisets of typed arcs.

As an example, take the class diagram in Fig. 1, built by using a UML-like notation in which classes (like $C$ or $D$) may belong to two packages, so this notation is even more general than standard UML. The classes are the elements of the system. The arcs are annotated with different types, e.g., aggregations, inheritance, use, etc., and two classes may very well be connected by several relationships, of the same or of different types (see classes $K$ and $L$). In addition, a UML-like diagram may not even represent all of the relationships existing between classes. For instance, inheritance is a transitive relation, and transitive relationships are not explicitly represented. In Fig. 2, the aggregation between $M$ and $Q$ gets inherited by $N$, $O$, and $P$. So, the actual set of relationships may be greater than those that are explicitly mentioned in the graph.

To define axioms for internal software attributes defined for software artifacts, we first need to define an "algebra" whose operations are introduced next. In what follows, the same symbol (e.g., $\cup$ for union) may denote an operation between

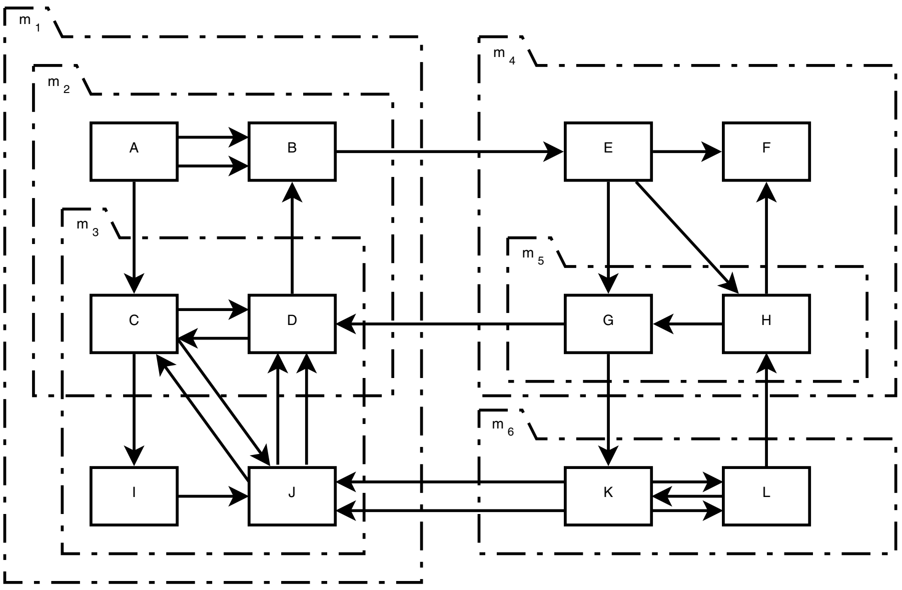


**Fig. 1.** Representation of a system and its modules in a UML-like language

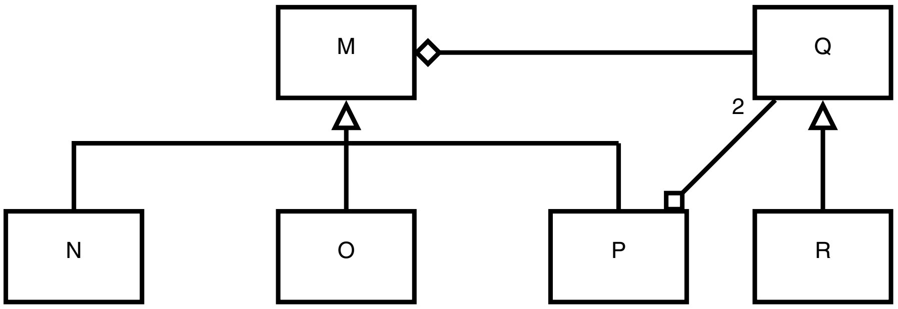


**Fig. 2.** A UML-like Class Diagram

– sets when sets of elements are involved
– multisets when multisets of typed relationships are involved
– modules (see Definition 9) when modules are involved.

These operations are different, but no confusion will arise because they never involve operands of different nature. For instance, no union will be defined between a multiset of typed relationships and a module.

For completeness, we here provide the meaning of these operations between two typed multisets of relationships $R_1$, $R_2$.

**Inclusion.** $R_1 \subseteq R_2 \Leftrightarrow \forall << a, b, t >, n_1 > \in R_1$,
$\exists << a, b, t >, n_2 > \in R_2 \wedge n_1 \leq n_2$, i.e., $R_2$ contains at least all the occurrences of the typed relationships in $R_1$.

**Union.** $R_3 = R_1 \cup R_2 \Leftrightarrow \forall << a, b, t >, n_3 > \in R_3$,
$\exists << a, b, t >, n_1 > \in R_1, << a, b, t >, n_2 > \in R_2, n_3 = n_1 + n_2$, i.e., $R_3$ gathers all the occurrences of the typed relationships in $R_1$ and $R_2$.

**Intersection.** $R_3 = R_1 \cap R_2 \Leftrightarrow \forall << a, b, t >, n_3 > \in R_3, \exists << a, b, t >, n_1 > \in R_1, << a, b, t >, n_2 > \in R_2, n_3 = min\{n_1, n_2\}$, i.e., $R_3$ contains all the occurrences of typed relationships in common to $R_1$ and $R_2$.

Using operations like the union implies that parts of a system be identifiable so they can be put together. Also, some internal software attributes naturally require that parts of a system be identifiable. For instance, coupling is typically defined as an attribute defined for the cooperating parts of a software system, or for the entire system. These parts of a system are actually subsystems, which we call *modules.*

**Definition 9.** *Module. Given a system* $S = < E, R >$*, a module* $m = < E_m, R_m >$ *is a system such that* $E_m \subseteq E \wedge R_m \subseteq R$*.*

For maximum generality and simplicity, a module is simply a subsystem, with no additional characteristics (e.g., an interface). At any rate, a module $m$ of a system will contain a multiset of relationships of its own, and there will be a (possibly empty) multiset of relationships that link $m$ to the rest of the system, which will be denoted as $OuterR(m)$. In Fig. 1, UML-like packages $m_1$, $m_2$, $m_3$, $m_4$, $m_5$, $m_6$, may be interpreted as modules. It will be our convention in the remainder of the paper that the set of elements and the multiset of relationships of a system or a module have the same subscript as the system or module, unless otherwise explicitly specified (e.g. $m_1 = < E_1, R_1 >$).

We can now introduce a few operations and definitions that compose the "algebra" of modules upon which the sets of axioms will be defined.

**Inclusion.** Module $m_1$ is said to be included in module $m_2$ (notation: $m_1 \subseteq m_2$) if $E_1 \subseteq E_2 \wedge R_1 \subseteq R_2$. In Fig. 1, $m_5 \subseteq m_4$.

**Union.** The union of modules $m_1$ and $m_2$ (notation: $m_1 \cup m_2$) is the module $< E_1 \cup E_2, R_1 \cup R_2 >$. In Fig. 1, $m_1 = m_2 \cup m_3$.

**Intersection.** The intersection of modules $m_1$ and $m_2$ (notation: $m_1 \cap m_2$) is the module $< E_1 \cap E_2, R_1 \cap R_2 >$. In Fig. 1, $m_2 \cap m_3$ is the module whose elements are classes $C$ and $D$ and whose relationships are $<< C, D, t >, 1 >$ and $<< D, C, u >, 1 >$ (assuming that they have type $t$ and $u$, respectively).

**Empty Module.** Module $< \emptyset, \emptyset >$ (denoted by $\emptyset$) is the empty module.

**Disjoint Modules.** Modules $m_1$ and $m_2$ are said to be disjoint if $m_1 \cap m_2 = \oslash$. In Fig. 1, $m_3$ and $m_6$ are disjoint.

**Unconnected Modules.** Two disjoint modules $m_1$ and $m_2$ of a system are said to be unconnected if $OuterR(m_1) \cap OuterR(m_2) = \oslash$. In Fig. 1, $m_4$ and $m_6$ are unconnected, while $m_3$ and $m_6$ are not unconnected.

## 5.2 Axiom Sets and Derived Properties

We here introduce a set of axioms for a few internal software attributes of interest. In addition, we show properties that can be derived as implications of those axioms, to further check whether the modeling of an internal software attributes is consistent with the intuition on it. As a matter of fact, the decision as to which properties are more basic and should be taken as axioms and which are derived properties is somewhat subjective. We mostly take properties satisfied by ratio measures as the axioms and, often, properties satisfied by ordinal measures are derived. (Each axiom and property is annotated by the level of measurement of the measures to which the axiom or property can be applied to.) The derived properties are "weaker" than the axioms base and are often satisfied by measures that are ordinal or nominal and not necessarily ratio ones. This is not just a theoretical exercise, but can guide the building of ordinal or nominal measures, instead of only ratio ones.

**Size.** The idea underlying the first axiom is that the size of a module composed of two possibly overlapping modules is not greater than the sum of the sizes of the two modules by themselves.

**Size Axiom 1** Union of Modules (ratio scales). *The size of a system $S$ is not greater than the sum of the sizes of two of its modules $m_1$ and $m_2$ such that each element of $S$ is an element of either $m_1$ or $m_2$ or both*

$$E = E_1 \cup E_2 \Rightarrow Size(S) \leq Size(m_1) + Size(m_2)$$

For instance, $Size(m_1) \leq Size(m_2) + Size(m_3)$ in Fig. 1.

However, when the two modules are disjoint, size is additive.

**Size Axiom 2** Module Additivity (ratio scales). *The size of a system $S$ is equal to the sum of the sizes of two of its modules $m_1$ and $m_2$ such that any element of $S$ is an element of either $m_1$ or $m_2$ but not both*

$$E = E_1 \cup E_2 \wedge E_1 \cap E_2 = \oslash \Rightarrow$$
$$Size(S) = Size(m_1) + Size(m_2)$$

Thus, $Size(m_1 \cup m_6) = Size(m_1) + Size(m_6)$ in Fig. 1.

A number of properties can be derived from these two base axioms, as follows:

– the size of the empty system is zero (ratio scales);

- the size of a system is nonnegative (ratio scales);
- the size of a system is not lower than the size of the empty system; though it can be clearly inferred from the first two derived properties, this is a property that can be used for ordinal scales too (ordinal scales);
- adding elements to a system cannot decrease its size (ordinal scales);
- relationships have no impact on size, i.e., two systems with the same elements will have the same size (nominal scales);
- a measure of size is computed as the sum of the "sizes" of its elements: if we take each element $e$ of a system and we build a module that only contains $e$, then compute the size of this newly defined module, and then sum the sizes of all these newly defined modules, we obtain the value of the size of the entire system (ratio scales).

The last two derived properties thus show that size is based on the elements of a software system and not on its relationships.

It turns out that this axiomatic definition of size is closely related to the axiomatic definition of what is known as "measure" in Measure Theory [30], an important branch of Mathematics that is a part of the basis of the theory of differentiation and integration in Calculus. So, this places these axioms on even firmer mathematical grounds.

Examples of size measures according to this axiomatic approach: *#Statements*, *LOC*, *#Modules*, *#Procedures*, Halstead's *Length* [17], *#Unique_Operators*, *#Unique_Operands*, *#Occurrences_of_Operators*, *#Occurrences_of_Operands*, *WMC* [13]. Instead, these are not size measures: Halstead's *Estimator_of_length* and *Volume* [17].

**Complexity.** We are dealing here with internal software attributes, so we here mean "structural" complexity, and not some kind "psychological" complexity, which would be an external software attribute. Complexity is based on the relationships among system elements, unlike size.

The idea underlying the first axiom, which characterizes complexity the most, is that the complexity of a system is never lower than the sum of the complexities of its modules taken in "isolation," i.e., when they have no relationships in common, even though they may have elements in common.

**Complexity Axiom 1.** Module Composition (ratio scales). *The complexity of a system $S$ is not lower than the sum of the complexities of any two of its modules $m_1$, $m_2$ with no relationships in common*

$$S \supseteq m_1 \cup m_2 \wedge R_1 \cap R_2 = \emptyset \Rightarrow$$
$$Complexity(S) \geq Complexity(m_1) + Complexity(m_2)$$

Suppose that the two modules $m_1$ and $m_2$ in Complexity Axiom 1 have elements in common. All of the transitive relationships that exist in $m_1$ and $m_2$ when they are taken in isolation still exist in $S$. In addition, $S$ may contain new transitive relationships between the elements of $m_1$ and $m_2$, which do not exist in either module in isolation. So, the complexity of $S$ is not lower than the sum of the complexities of the two modules in isolation. For instance, in Fig. 1, $Complexity(m_1) \geq Complexity(m_2) + Complexity(m_3)$.

When a system is made up of two unconnected modules, complexity is additive

**Complexity Axiom 2.** Unconnected Module Additivity (ratio scales). *The complexity of a system $S$ composed of two unconnected modules $m_1$, $m_2$ is equal to the sum of the complexities of the two modules*

$$\begin{aligned} S = m_1 \cup m_2 \wedge \\ m_1 \cap m_2 = \emptyset \wedge OuterR(m_1) \cap OuterR(m_2) = \emptyset \Rightarrow \\ Complexity(S) = Complexity(m_1) + Complexity(m_2) \end{aligned}$$

We now describe a few derived properties for complexity:

- a system with no relationships has zero complexity (ratio scales);
- the complexity of a system is nonnegative (ratio scales);
- the complexity of a system is not lower than the complexity of a system with no relationship, which can be clearly inferred from the first two derived properties; however, this is a property that can be used for ordinal scales too (ordinal scales);
- adding relationships to a system cannot decrease its complexity (ordinal scales);
- elements have no impact on complexity, i.e., two systems with the same relationships will have thee same complexity (nominal scales).

Summarizing, as opposed to size, complexity depends on relationships and not on elements.

These measures may be classified as complexity measures, according to the above axioms: Oviedo's data flow complexity measure $DF$ [29], $v(G) - p$, where $v(G)$ is McCabe's cyclomatic number and $p$ is the number of connected components in a control-flow graph [22]. These measures do not satisfy the above axioms: Henry and Kafura's information flow complexity measure [18], $RFC$ and $LCOM$ [13].

**Cohesion.** Cohesion is related to the *degree* and not the *extent* with which the elements of a module are tied to each other. Thus, cohesion measures are normalized.

**Cohesion Axiom 1.** Upper Bound (ordinal scales). *The cohesion of a module $m$ is not greater than a specified value $Max$, i.e., $Cohesion(m) \leq Max$.*

Elements are linked to each other via relationships, so adding relationships does not decrease cohesion.

**Cohesion Axiom 2.** Monotonicity (ordinal scales). *Let modules* $m_1 =< E, R_1 >$, $m_2 =< E, R_2 >$ *be two modules with the same set of elements* $E$*, and let* $R_1 \subseteq R_2$*. Then,* $Cohesion(m_1) \leq Cohesion(m_2)$*.*

A module has high cohesion if its elements are highly connected to each other. So, if we put together two modules haphazardly and these two modules are not connected to each other, we cannot hope that the cohesion of the new module will be greater than the cohesion of each the two original modules separately.

**Cohesion Axiom 3.** Unconnected Modules (ordinal scales). *Let* $m_1$ *and* $m_2$ *be two unconnected modules, then,*

$$max\{Cohesion(m_1), Cohesion(m_2)\} \geq Cohesion(m_1 \cup m_2)$$

Thus, in Fig. 1, we have

$$Cohesion(m_4 \cup m_6) \leq max\{Cohesion(m_4), Cohesion(m_6)\}$$

As a side note, these first three axioms may be safely applied to ordinal measures (e.g., a measure like Yourdon and Constantine's [40]).

The following axiom may be satisfied only by ratio measures.

**Cohesion Axiom 4.** Null Value (ratio scales). *The cohesion of a module with no relationships* $m =< E, \oslash >$ *is null, i.e.,* $Cohesion(m) = 0$*.*

The above axioms imply the following property:

- the cohesion of a module is not lower than the cohesion of a module with no relationships (ordinal scales).

Examples of cohesion measures according to the above axioms: $PRCI$, $NRCI$, $ORCI$ [11].

**Coupling.** As opposed to cohesion, the coupling of a module in a system is related to the *amount* of connection between the elements of a module and the elements of the rest of the system. The interconnections may be direct or transitive. So, adding a relationship, whether internal to the module or belonging to its set of outer relationships, can never decrease coupling.

**Coupling Axiom 1.** Monotonicity (ordinal scales).
*Adding a new relationship to a module* $m_1$ *or to its set of outer relationships* $OuterR(m_1)$ *does not decrease its coupling. So, if* $m_2$ *is a module such that* $E_2 = E_1$*, we have*

$$OuterR(m_2) \supseteq OuterR(m_1) \wedge R_2 \supseteq R_1 \Rightarrow \\ Coupling(m_2) \geq Coupling(m_1)$$

At any rate, if a module has no outer relationships, its elements are not connected with the rest of the system, so its coupling is zero.

**Coupling Axiom 2.** Null Value (ratio scales). *The coupling of a module with no outer relationships is null.*

Suppose now that we take two modules and put them together. The relationships from one to the other used to be outer ones, but become internal ones after the merging. Thus, we have lost some couplings of the two initial modules in the new module, whose coupling is not higher than the sum of the couplings of the two modules. In Fig. 1, when modules $m_2$ and $m_3$ are merged into module $m_1$ the relationships to and from $m_4$, $m_5$, and $m_6$ are still outer relationships for $m_1$, but the relationships between $m_2$ and $m_3$ have become internal relationships for $m_1$ (so, they may also contribute to the cohesion of $m_1$).

**Coupling Axiom 3.** Merging of Modules (ratio scales). *The coupling of the union of two modules $m_1$, $m_2$ is not greater than the sum of the couplings of the two modules*

$$Coupling(m_1 \cup m_2) \leq Coupling(m_1) + Coupling(m_2)$$

However, if the two original modules that got merged were not connected, no coupling has been lost and the new modules has exactly the same coupling as the two original modules.

**Coupling Axiom 4.** Unconnected Modules (ratio scales). *The coupling of the union of two unconnected modules is equal to the sum of their couplings*

$$m_1 \cap m_2 = \oslash \wedge OuterR(m_1) \cap OuterR(m_2) = \oslash \Rightarrow$$
$$Coupling(m_1 \cup m_2) = Coupling(m_1) + Coupling(m_2)$$

So, $Coupling(m_4 \cup m_6) = Coupling(m_4) + Coupling(m_6)$ in Fig. 1.

Like with the other internal attributes, derived properties can be found, as follows:

- the coupling of a module is nonnegative (ratio scales);
- the coupling of a module is not less than the coupling of a module with no outer relationships (ordinal scales).

Among the measures that may be classified as coupling measures according to the above axioms are: $TIC$ and $DIC$ [11], $CBO$ and $RFC$ [13]. Fenton's coupling measure [16] does not satisfy the above axioms.

### 5.3 Relationships between Software Attributes

Table 2 summarizes the main characteristics of software attributes for a module $m$ of a system according to the unified axiomatic approach described in this section. We report 1) the condition for the attribute to assume value zero in column "Null Value," 2) the variable with respect to which the attribute has a monotonic behavior in column "Monotonicity," and 3) the condition for additivity in column "Additivity," if any.

One of the goals of this axiomatic approach is to identify similarities, differences, and relationships between attributes, as we now concisely discuss.

**Table 2.** Characteristics of different software attributes

| Attribute | Null Value | Monotonicity | Additivity |
|---|---|---|---|
| Size | $E_m = \emptyset$ | $E_m$ | Separate modules |
| Complexity | $R_m = \emptyset$ | $R_m$ | Unconnected modules |
| Cohesion | $R_m = \emptyset$ | $R_m$ | NO |
| Coupling | $OuterR(m) = \emptyset$ | $OuterR(m) \cup R_m$ | Unconnected modules |

**Size vs. Complexity** These are the main differences in the properties of size and complexity

- size is based on elements, complexity is based on relationships
- the inequalities about the sums of sizes and complexities in Size Axiom 1 and Complexity Axiom 1 go in opposite directions
- complexity cannot be interpreted as the amount of relationships, as if it was the "size" of the set of relationships, while size is the sum of the "sizes" of the individual elements.

On the other hand, both size and complexity have additivity properties, though under different conditions (see Size Axiom 2 and Complexity Axiom 2).

**Complexity vs. Cohesion** Complexity and cohesion of a module share a number of similarities as both

- depend on the relationships within the module
- are null when there are no relationships in the module
- increase when a relationship is added to the relationships of the module.

It is possible to show that cohesion measures can actually be defined as absolute measures as follows. Given a complexity measure $cx$, for any given module $m$, suppose that there exists $cx_M(m)$ a maximum possible value for $cx$ when it is applied to the elements of module $m$. This may be reasonable, as there is a finite number of elements in $m$, and the elements may be linked by a limited number of relationships. Then, $ch(m) = cx(m)/cx_M(m)$ is a cohesion measure. This has two important consequences.

1. From a practical point of view, cohesion may increase when complexity increases. This might explain why sometimes cohesion measures are not very well related to fault-proneness [8], as the positive effect of the increase in cohesion on error-proneness is somewhat masked by the negative effect of an increase in complexity.
2. From a theoretical point of view, an equation like $ch(m) = cx(m)/cx_M(m)$ may used as a starting point to find quantitative relationships among attributes, as is usual in many scientific disciplines.

**Complexity vs. Coupling.** Both complexity and coupling of a module

- are null when there are no relationships in the module and outside it
- increase when a relationship is added to the relationships of the module.

One characterizing difference between complexity and coupling is that, when merging two disjoint modules are merged in a module, the complexity of the resulting module is not less than the sum of the complexities of the original modules, while the coupling of the resulting module is not greater than the sum of the couplings of the original modules.

## 6 External Software Attributes: Probability Representations

As explained in Section 2, a number of different external software attributes are of interest for several categories of software "users," depending on their specific goals and the type of application at hand. For instance, usability may be very important for the final users of web applications, while time efficiency may be a fundamental external software attribute for the users of a real-time system, which must deliver correct results within a specified time interval. As for practitioners, every decision made during software development is made, implicitly or explicitly, based on some external software attribute. For instance, when a decision is made between two alternative designs, a number of external software attributes are implicitly or explicitly taken into account, e.g., maintainability, portability, efficiency.

External software attributes may be conflicting. Increasing one may negatively affect others, so a satisfactory trade-off must be reached among them. Being able to assess these qualities may provide users and practitioners with a way to base decisions on firmer grounds and evaluate whether a software product's or component's quality is satisfactory according to a user's or practitioner's goals, and identify a product's or component's strengths and weaknesses.

A number of proposals have appeared in the literature to quantify these external software attributes (e.g., among several others, maintainability [28], usability [35]). In addition, standards have been defined to define the qualities (i.e., external software attributes) of software products, and, more generally, software artifacts. For instance, the ISO9126 standard [1,2] defines quality by means of 6 characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics, in turn, are defined in terms of subcharacteristics in a tree-like structure, and measures have been proposed for them too. (An additional characteristic is called quality in use, to summarize the quality as perceived by the user.)

Standards like ISO9126 are useful as reference frameworks, but they may turn out to be too general, as they are meant to address the development of many different kinds of software. So, they do not base the definition and quantification of software qualities on precise, formal, and unambiguous terms, which is what one would expect from measurement activities, which are among the most precise, formal, and unambiguous activities in engineering and scientific disciplines. It is probably impossible to remove all subjectivity and uncertainty in Empirical Software Engineering, due to the number of different factors that influence software production, and especially its being so heavily human-intensive. However, because of the nature of Software Engineering, it is important that the degree of subjectivity and uncertainty be reduced, and, most of all, formalized. Thus, external software attributes should be based on firm, mathematical grounds, to remove subjectivity and uncertainty to the extent possible, and highlight their possible sources and the factors that may influence them. Theoretically sound and sensible ways to measure external software attributes will help prevent the quantification of external software attributes via ill-defined measures or not fully justified approaches.

In this section, we describe a unified probability-based framework for measuring external software attributes [24], which shows that external software attributes should be quantified by means of probabilistic *estimation models* instead of *measures* as defined in Section 3.1.

We discuss the problems associated with using measures to quantify external software attributes (Section 6.1) and then describe of so-called "probability representations" along with their advantages (Section 6.2). Probability Representations are a part of Measurement Theory that is often neglected in Software Measurement, even though they have already been implicitly used in Software Measurement in the modeling and quantification of software reliability [27], for instance. Software reliability can be viewed as a "success story" in the modeling of external software attributes. We describe how it is possible to put another important external software attribute, i.e., software modifiability, on firm mathematical grounds (Section 6.3), to show another useful application of Probability Representations. However, it is not the goal of this paper to study any of these models in detail or propose or validate a specific model as the "right" estimation models for modifiability.

## 6.1 Issues in the Definition of External Attributes

While the distinction between internal and external software attributes may be useful to understand their nature, we would like to point out a few issues with this distinction.

**No such definition in Measurement Theory.** The distinction between internal and external attributes can only be found in the Software Measurement literature (e.g., [16,15], but not in the general, standard, authoritative literature on Measurement [20,34]

**Incompleteness of the Definition.** The definition of a measure given by Measurement Theory [20,34] is the one reported in Section 3.1: a measure is a function that associates a value with an entity. So, it is knowledge from that entity *alone* that must used in the definition of the measure, and not other entities that belong to the "environment" of the entity.

**Logical Problems in Defining Attributes by Means of their Measures.** The distinction between internal and external software attributes is based on whether their measures can be based on the entities alone or an "environment" as well, although attributes exist prior to and independent of how they can be measured. However, the definition of a measure logically *follows* the definition of the attribute it purports to measure: one defines a measure based on the attribute, not the attribute based on the measure. Also, suppose that two measures are defined for an attribute: one takes into account only information from the entity being measured, while the other also takes into account additional information about the "environment." According to the former measure, the attribute would be an internal one, but an external one according to the latter. So, the nature of the attribute would be uncertain, to say the least.

**Deterministic vs. Probabilistic Approaches.** An external software attribute (e.g., reliability or maintainability) may be affected by many variables (the "environment") in addition to the specific entity, so it would not be sensible to build a deterministic measure for it.

**Using Aggregate Indicators.** Aggregate indicators are often used to quantify external software attributes. For instance, the Mean Time Between Failures ($MTBF$) may be a quite useful piece of information about reliability, but it is not a measure of reliability in itself as we now explain.

- $MBTF$ is the expected value of the probability distribution of the time between failures, so quantifying $MBTF$ implies knowing this probability distribution. However, this is impossible, since probabilities cannot be measured in a frequentist approach, but they can only be estimated. This implies that $MBTF$ itself can only be estimated, but not measured.
- The probability distribution is a conditional one anyway, since it depends on the environment in which the program is used.

**Validating a Probabilistic Representation for an Attribute.** Probability Representations can be empirically validated in a probabilistic sense, while deterministic representations (like the ones shown in Section 3) should be validated in a totally different way. For instance, to check whether software size is additive with respect to some kind of concatenation operation, one should take all possible program segments, make all possible concatenations, and check if for all of these concatenations size is truly additive–which is totally unfeasible. Probability Representations can be validated through statistical inference procedures. It is true that these procedures can never provide absolute certainty, but this is acceptable because of the random nature of the modeling.

## 6.2 Probability Representations in Measurement Theory

Here, we introduce the basic concepts of Probability Representations defined in Measurement Theory [20] by slightly adapting them to our Software Measurement case. We first need to introduce the concept of algebra and of $\sigma$-algebra of sets on a set $X$.

**Definition 10 (Algebra on a Set).** *Suppose that $X$ is a nonempty set, and that $E$ is a nonempty family of subsets of $X$. $E$ is an algebra of sets on $X$ if and only if, for every $A, B \in E$*

*1.* $X - A \in E$
*2.* $A \bigcup B \in E$.

*The elements of E are called events and the individual elements of X are called outcomes, each of which is a possible results of a so-called random experiment [19]. So, an event is actually a set of outcomes, and X is the set of all possible outcomes.*

**Definition 11 (The Concept of $\sigma$-Algebra on a Set).** *If the conditions in Definition 10 hold and, in addition, E is closed under countable unions, i.e., whenever $A_i \in E$, with $i = 1, 2, \ldots$, it follows that $\bigcup_{i=1}^{\infty} A_i \in E$, then E is called a $\sigma$-algebra on X.*

Based on these definitions, the usual axiomatic definition of *unconditional* probability can be given [20].

However, we are here interested in *conditional* probability representations, because we are interested in conditional probabilities like the following ones.

**Continuous case.** $P(Eff \leq eff|art, env)$, i.e., the probability that a specified event occurs if one uses an amount of effort that is at most $eff$, provided that the environment *env* in which it happens and the artifact *art* on which it happens are specified, e.g., the probability that a specified artifact *art* is modified correctly with at most a specified amount of effort $eff$, in a specified modification environment *env*. In this case, effort $Eff$ is the random variable, once the environment and the artifact are known (i.e., conditioned on their knowledge). This probability can be used to quantify the external software attribute "modifiability," for which we provide a model in Section 6.3.

**Discrete case.** $P(N \leq n|art, env)$, i.e., the probability that a specified event occurs after at most $n$ trials, provided that the environment *env* in which it occurs and the artifact *art* on which it occurs are specified, e.g., the probability that a specified program *art* is covered (according to some specified notion of coverage) by executing it with at most $n$ inputs, in a specified environment *env*. The number of trials $N$ is the random variable, once the environment and the artifact are known (i.e., conditioned on their knowledge). This probability can be used to quantify the external software attribute "coverability." More details are provided in [24].

The set-theoretic notation of the theory of [20] can be interpreted as follows for our goals. The set $X$ is the set of all possible triple of the form $< op, art, env >$ where

– *op* is an "observable phenomenon," i.e., built via a predicate like $Eff \leq eff$ or $N \leq n$
– *art* is a specific artifact (e.g., a program)
– *env* is an environment in which *art* is used and *op* is observed.

For notational convenience, we denote conditional probabilities as $P(op|art, env)$. For instance, reliability can be quantified as a conditional probability, as follows

$$R(t) = P(t \leq T|art, env)$$

i.e., the probability that a failure occurs at time $T$ not less than a specified time $t$, in a specified program *art* and in a specified operational environment *env*.

Like with deterministic representations, we capture our intuitive knowledge on the ordering among conditional events via an order relation $\succsim$, whose meaning is "qualitatively at least as probable as" [20]. In general, suppose that $A$, $B$, $C$, and $D$ are events. By writing $A|B \succsim C|D$, we mean that event $A$, when event $B$ is known to occur, is "qualitatively at least as probable as" event $C$, when event $D$ is known to occur. In other words, instead of having a deterministic ordering among entities according to some attribute of interest like in deterministic representations, one has a probabilistic ordering. For instance, one may order software programs according to their modifiability in a probabilistic way (i.e., a program in an operational environment is *qualitatively at least as modifiable as* another program in another operational environment), instead of a deterministic way (i.e., a program is *certainly* more modifiable than another). For completeness, based on relation $\succsim$, one may also define relation $\sim$ as follows: $A|B \sim C|D$ if and only if $A|B \succsim C|D$ and $C|D \succsim A|B$.

The Representation Condition needed for conditional probability representations is

$$A|B \succsim C|D \Leftrightarrow P(A|B) \geq P(C|D)$$

At a first glance, it may appear that the order relation $\succsim$ is a binary relation that is a subset of $(E \times E) \times (E \times E)$, since $A|B \succsim C|D$ is simply a graphical convention for $< A, B > \succsim < C, D >$. However, some caution must be exercised. Conditional probabilities are defined as $P(A|B) = P(A \cap B)/P(B)$, so $P(A|B)$ is defined only if $P(B) > 0$. Thus, if $P(B) = 0$, writing $A|B \succsim C|D \Leftrightarrow P(A|B) \geq P(C|D)$ makes no sense. This means that any event $B$ such that $P(B) = 0$ cannot appear as the second element of $A|B$, i.e., as the conditioning event. Thus, by denoting with $NN$ (as in $NonNull$) the set of events $B$ such that $P(B) > 0$, the order relation $\succsim$ is actually a binary relation on $E \times NN$.

Here are necessary conditions (slightly adapted from [20]) for the Representation Condition. We simply list these axioms here for completeness. At any rate, more details on the general theoretical approach are provided in [20], and on their application in Software Measurement in [24].

**Definition 12 (Conditional Probability Axioms).** *Let $X$ be a nonempty set, $E$ an algebra of sets on $X$, $NN$ a subset of $E$, and $\succsim$ a binary relation on $E \times NN$. The quadruple $< X, E, NN, \succsim>$ is a structure of qualitative conditional probability if and only if for every $A$, $B$, $C$, $A'$, $B'$, and $C' \in E$ (or $\in NN$, whenever the name of the event appears to the right of $'|'$), the following axioms hold.*

1. *$< E \times NN, \succsim>$ is a weak order.*
2. *$X \in NN$ and $A \in E - NN$ if and only if $A|X \sim \emptyset|X$.*
3. *$X|X \sim A|A$ and $X|X \succsim A|B$.*
4. *$A|B \sim A \bigcap B|B$.*

5. *Suppose that* $A \bigcap B = A' \bigcap B' = \emptyset$. *If* $A|C \gtrsim A'|C$ *and* $B|C \succsim B'|C'$, *then* $A \bigcup B|C \succsim A' \bigcup B'|C'$; *also, if either hypothesis is* $\succ$, *then the conclusion is* $\succ$.
6. *Suppose that* $A \supset B \supset C$ *and* $A' \supset B' \supset C'$. *If* $B|A \succsim C'|B'$ *and* $C|B \succsim B'|A'$, *then* $C|A \succsim C'|A'$; *moreover, if either hypothesis is* $\succ$, *then the conclusion is* $\succ$.

The above axioms can be used as necessary conditions to find additional conditions under which an ordering relation on $E$ has an order-preserving function $P$ that satisfies the above axioms, i.e., the Representation Condition of Section 3.1. This means that different probability representations, i.e., different probability functions, may exist so that the Representation Condition of Section 3.1 is satisfied. Additional conditions may be provided to make the set of axioms sufficient [20]. However, we are not interested in these additional conditions here. In Section 6.3, we show how to build actual probability functions.

Note that it is not important here that our probabilistic intuitive knowledge is accurate. We only need to put the concept of using probabilities for external software attributes on solid bases. Empirical studies will show whether our intuitive knowledge is correct. If it is not, we need to modify our intuitive knowledge in such a way as to fit the empirical results. This is another value added of this approach, since it allows us to increase and refine our empirical knowledge about an attribute of interest.

## 6.3 Representing Modifiability

Based on the above Probability Representation approach, modifiability is here quantified as the probability that a given artifact, in a specified modification environment, is modified with a specified amount of effort, i.e., $Mod(eff) = Mod(Eff \leq eff|art, env) = P(Eff \leq eff|art, env)$.

Simply to show how a modifiability model can be built [24], rather than proposing it as the "right" or "preferred" modifiability model, suppose that the modifiability rate of an artifact (which is the counterpart of the hazard rate used for reliability [27] when studying modifiability) is a linear function of the probability $Mod(eff)$ that the artifact has been modified with $eff$ effort. The underlying idea is that, if an artifact needs to undergo one specific modification, As more and more effort is used to carry out that modification, (1) the higher is the probability $Mod(eff)$ that the modification is actually carried out, and (2) the higher is the instantaneous probability that the modification is going to be carried out if it has not been carried out so far (this instantaneous probability is actually the modifiability rate). Thus, we can write

$$\frac{Mod'(eff)}{1 - Mod(eff)} = a + bMod(eff)$$

Coefficient $a$ is the initial modifiability rate at $eff = 0$ and coefficient $b$ describes how well one uses the information contained in the modification activities up to effort $eff$. It can be shown that the function that describes the modification probability in closed form is

$$Mod(eff) = \frac{a(e^{(a+b)eff} - 1)}{b + ae^{(a+b)eff}}$$

Parameters $a$ and $b$ may be explicitly related to *env* and *art*. For instance, they may be a function of the number of people that modify the artifact and the size (e.g., the number of lines of code) of the artifact. For instance, we could have $a = \alpha \cdot \#people$ and $b = \beta \cdot 1/LOC$. Based on the past history of efforts needed to modify an artifact, parameters $a$ and $b$ or $\alpha$ and $\beta$ are estimated by using some statistical techniques [19]–provided that $a + b > 0$, and $a > 0$, since the modifiability rate is positive.

Once the probability distribution is known, a number of derived indices may be used to provide a concise idea for the probability distribution of an attribute, e.g., the expected values for the distributions obtained for modifiability (i.e., the average effort needed for modifying a software artifact). These derived indices may be used for instance to set process goals (e.g., the average effort needed for modifying a software artifact must be no greater than a specified value) or compare competing techniques (e.g., given two development techniques, one may choose the one that has the lower average average effort needed to modify a program).

One final note on modifiability. One may very well argue that modifiability depends on the specific modification that needs to be carried out. However, a similar remark applies to software reliability, which clearly depends on the specific inputs selected or the selection policy used. The fact that there are several different modifications that may be carried out and one of them is actually carried out in a specific way according to a random policy mirrors the random selection of inputs that is used in software reliability modeling. Actually, in software reliability modeling, one may argue that, once an input has been selected for a deterministic program, then there is only one possible result which is either correct or incorrect. Instead, when it comes to modifiability, when the need for a modification has been identified, many different random variables may influence the way the actual modification is carried out, and therefore the effort needed. Thus, the use of a probabilistic model may be even more justified for modifiability than for reliability.

## 7 GQM/MEDEA

As the definition of a measure needs to be carried out carefully, it is necessary to have a defined process in place. In this section, we describe the GQM/MEasure DEfinition Approach (GQM/MEDEA) [12], which takes advantage of the goal-oriented nature of the Goal/Question/Metric paradigm [6] to set the measurement goals of any measurement activity to guide the measure definition and validation process. GQM/MEDEA can be used for building so-called predictive models, i.e., models that use one or more internal software attributes to predict an external software attribute or process attribute of interest. We use a semi-formal notation, Data Flow Diagrams (DFDs) [14], to define and refine the steps used in GQM/MEDEA. In DFDs, bubbles denote activities, boxes external information sources/sinks, and arrows data flows. The arrows also provide an idea of the order in which the activities are executed, though, during the execution of an activity, any other activity may be resumed or started as long as its inputs are available. A bubble may be refined by a DFD, provided that the incoming and outgoing data flows of the bubble and its refining DFD are the same.

The topmost diagram in DFDs is called the Context Diagram (shown in Fig. 3, which represents the entire process as one bubble and shows the interactions of the measure definition process with information sources and sinks.

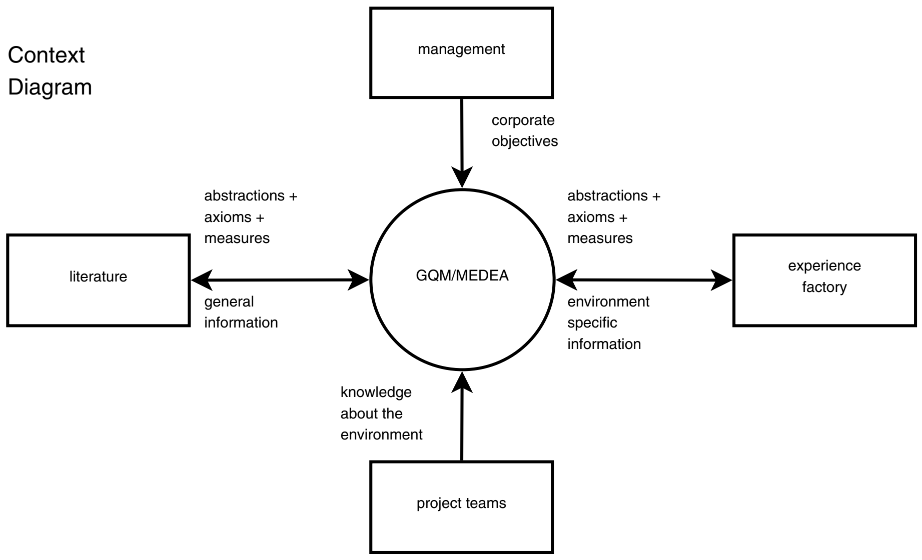

Context Diagram
management
corporate objectives
abstractions + axioms + measures
literature
general information
GQM/MEDEA
abstractions + axioms + measures
experience factory
environment specific information
knowledge about the environment
project teams

**Fig. 3.** Interactions of GQM/MEDEA with information sources and sinks

Several sources of information, as shown in Fig. 3, are used by the GQM/ MEDEA process:

- the management, to help define measures that are useful to achieve the corporate goals of a software organization (e.g.,"reduce maintenance effort");
- the personnel of the project(s) used to practically validate the measures; the people involved in the empirical study provide important information about the context of the study that cannot be found anywhere else;
- experience belonging to the software organization that has been previously gathered, distilled, and stored in the experience factory [4,7,5] (e.g., quantitative prediction models, lessons learned from past projects, measurement tools and procedures, or even raw project data);
- the scientific literature.

The measurement process itself should contribute its outputs to the experience factory with new artifacts, in the form of abstractions (i.e., models of software artifacts), measure properties, and measures. These outputs should be packaged and stored so that they can be efficiently and effectively reused later on, thus reducing the cost of measurement in an organization [6]. In a mature development environment, inputs for most of the steps should come from reused knowledge. Some of the steps that are made explicit in GQM/MEDEA are often left implicit during the definition of a measure. We have made them explicit to show all the logical steps that are carried out to identify all potential sources of problems. The main contribution of GQM/MEDEA to GQM is the definition of an organized process for the definition of software product measures based on GQM goals.

Fig. 4 7 shows the high-level structure of the approach.

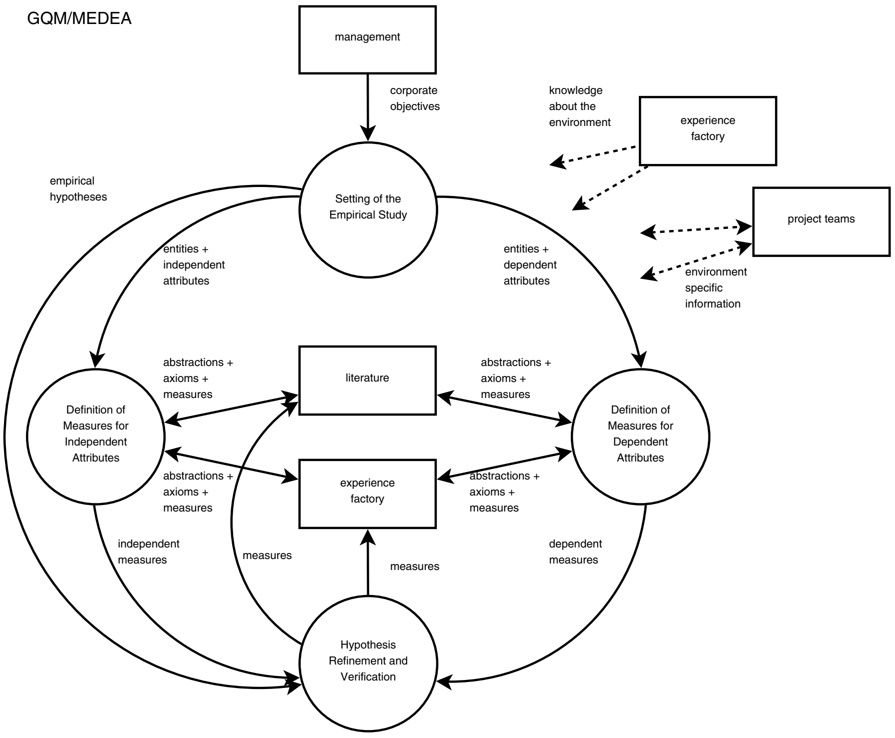

GQM/MEDEA
management
corporate objectives
knowledge about the environment
experience factory
project teams
environment specific information
empirical hypotheses
Setting of the Empirical Study
entities + independent attributes
entities + dependent attributes
abstractions + axioms + measures
literature
abstractions + axioms + measures
Definition of Measures for Independent Attributes
Definition of Measures for Dependent Attributes
abstractions + axioms + measures
experience factory
abstractions + axioms + measures
independent measures
measures
measures
dependent measures
Hypothesis Refinement and Verification

**Fig. 4.** GQM/MEDEA: high-level structure

Each high-level step of Fig. 4 is refined in the more detailed DFDs of Fig. 5. In Fig. 4 and in Fig. 5, we do not show explicitly the environment-specific information from the project teams and the experience factory, which permeates all activities represented in these figures, not to clutter the diagrams.

We now concisely illustrate the steps in Fig. 5. The interested reader may refer to [12] for more detailed information about GQM/MEDEA.

## 7.1 Setting of the Empirical Study

The steps and their connections are in Fig. 5(a). Corporate objectives (e.g., "reduce maintenance effort") are first refined into tactical goals (e.g., "improve the maintainability of the final product"), and then tactical goals are refined into measurement goals (e.g., "predict the maintainability of software code based on its design"). These refinements are based on knowledge about the environment provided by the project teams and the experience factory, which help identify processes and products that measurement should address. As the measurement goal should be made as precise as possible, goal-oriented techniques [6] can be used to detail the object of study, the specific quality to be investigated, the specific purpose for which the quality should be investigated, the immediate beneficiaries of the empirical investigation (e.g., the project managers), and the specific context in which the empirical investigation is carried out.

The measurement goals help establish a set of empirical hypotheses that relate (independent) attributes of some entities (e.g., the coupling of software components in design) to other (dependent) attributes of the same or different entities (e.g., the maintainability of the maintained software code). Dependent attributes are usually 1) external quality attributes of software systems or parts thereof, e.g., reliability, maintainability, effort, or 2) process attributes, e.g., development effort, development time, or number of faults. Independent attributes capture factors that are usually hypothesized to have a causal relationship with the dependent attribute. An empirical hypothesis describes how these two attributes are believed to be related, e.g., the coupling of the modules identified via a product's design is hypothesized to be negatively related to the final code maintainability. Empirical hypotheses cannot describe a specific functional form for this hypothesized dependency, because no measures have been yet defined for the independent and the dependent attributes. These definitions are carried out in the remainder of the measure definition process. So, empirical hypotheses are not statistical ones and cannot be tested. However, in the last phase of the GQM/MEDEA process (see Section 7.4), when specific measures have been defined for the independent and the dependent attributes, empirical hypotheses will be instantiated into statistical (and therefore testable) hypotheses.

## 7.2 Definition of Measures for the Independent Attributes

The process used to define measures for independent attributes is in Fig. 5(b). Independent attributes are formalized to characterize their measures, in ways like those in Sections 3 and 4. If an axiomatic approach is chosen, it is necessary to formalize entities via abstractions (e.g., graph models), which are built based on the entities, the independent attributes and their defining axioms. Once a correct abstraction is built, the axioms can be instantiated, i.e., a precise mapping of the specific characteristics of the model can be done onto the characteristics of

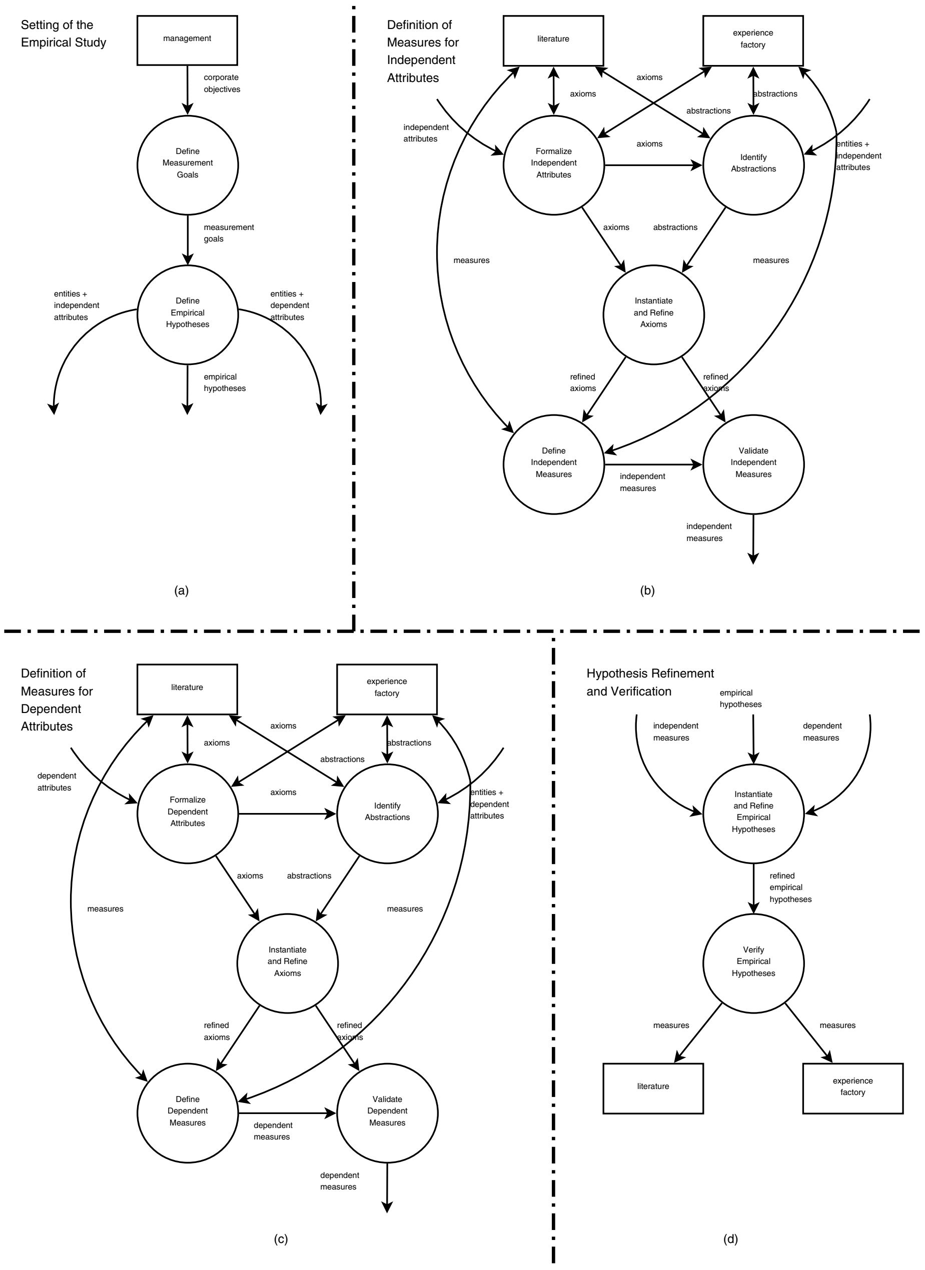

Setting of the Empirical Study
management
corporate objectives
Define Measurement Goals
measurement goals
Define Empirical Hypotheses
entities + independent attributes
entities + dependent attributes
empirical hypotheses
(a)
Definition of Measures for Independent Attributes
literature
experience factory
axioms
abstractions
independent attributes
Formalize Independent Attributes
Identify Abstractions
entities + independent attributes
measures
Instantiate and Refine Axioms
refined axioms
Define Independent Measures
Validate Independent Measures
independent measures
(b)
Definition of Measures for Dependent Attributes
dependent attributes
Formalize Dependent Attributes
entities + dependent attributes
Define Dependent Measures
Validate Dependent Measures
dependent measures
(c)
Hypothesis Refinement and Verification
empirical hypotheses
independent measures
dependent measures
Instantiate and Refine Empirical Hypotheses
refined empirical hypotheses
Verify Empirical Hypotheses
measures
literature
experience factory
(d)

**Fig. 5.** GQM/MEDEA: refined structure

the mathematical model upon which the formalization of an attribute is based. For instance, an abstraction of an object-oriented system can be obtained by mapping each class onto a different element of a graph-based model and each dependence between classes onto a relationship, like in Section 4. If a Measurement Theory-based approach is used, we need to identify the entities, the relationships we intuitively expect among entities, and the composition operations between entities first. In other words, we need to build the Empirical Relational System first, and the Numerical Relational System later on. Note that using an axiomatic approach may not provide all the information that is needed to build a measure. Different measures, which will give different orderings of entities, can be defined that satisfy a set of axioms. For instance, $LOC$ and $\#Statements$ both satisfy the axioms for size. However, given two program segments, it may very well be that a program segment has a value for $LOC$ greater than the other program segment, but a smaller value for $\#Statements$. So, an axiom set may be incomplete, and additional properties may need to be introduced to refine it and obtain a complete ordering of entities. These additional properties will depend on the specific application environment. Based on this refined set of axioms, new measures are defined or existing ones are selected for the attributes of entities. Additional checks may be required to verify whether the defined measures really comply with the refined set of axioms.

### 7.3 Definition of Measures for the Dependent Attributes

The GQM/MEDEA approach deals with independent and dependent attributes of entities in much the same way, as can be seen from Fig. 5(c). For instance, if our dependent attribute is a process attribute like maintenance effort, then effort can be modeled as a type of size. If our dependent attribute is maintainability, then we can use a Probability Representation approach like the one illustrated in Section 6.2. In the context of experimental design, the definition of measures for independent and dependent attributes via an organized and structured approach has the goal of reducing the threats to what is referred to as construct validity [37], i.e., the fact that a measure adequately captures the attribute it purports to measure. Although construct validity is key to the validity of an experiment, few guidelines exist to address that issue.

### 7.4 Hypothesis Refinement and Verification

Fig. 5(d) shows the steps carried out for hypothesis refinement and verification. The empirical hypotheses established as shown in Section 7.1 need to be refined and instantiated into statistical hypotheses to be verified, by using the measures defined for the independent and dependent attributes. One possibility is to provide a specific functional form for the relationship between independent and dependent measures, e.g., a linear relationship, so a correlation would be tested in a statistical way, based on actual development data. (Statistically testing associations would not be sufficient, because this does not lead to a prediction model, as we assume in this section for validating software measures.) Typically, additional data analysis problems have to be addressed such as outlier analysis [3] or the statistical power [19] of the study. The predictive model can be used to verify the plausibility of empirical hypotheses, in addition to being used as a prediction model in its own right.

## 8 Conclusions and Future Work

In this paper, we have shown a number of approaches for dealing with the fundamental aspects of Software Measurement, by describing the notions of Measurement Theory for both internal and external software attributes, the definition of properties for software measures via Axiomatic Approaches, and the proposal of an integrated process where the foundational aspects of Software Measurement can be coherently used.

A number of research and application direction should be pursued, including

– using Measurement Theory for modeling internal and external software attributes in a way that is consistent with intuition
– refining Axiomatic Approaches by building generalized consensus around the properties for software attributes
– extending Axiomatic Approaches to other software attributes of interest and understanding the relationships between different software attributes
– defining and refining processes for using the foundational aspects of Software Measurement in practice.

**Acknowledgments.** This work has been partially supported by project “Metodi e tecniche per la modellazione, lo sviluppo e la valutazione di sistemi software” funded by Università degli Studi dell’Insubria.

## References

1. ISO/IEC 9126-1:2001- Software Engineering - Product Quality Part 1: Quality Model. ISO/IEC (2001)
2. ISO/IEC 9126-2:2002- Software Engineering - Product Quality Part 1: External Metrics. ISO/IEC (2002)
3. Barnett, V., Lewis, T.: Outliers in statistical data, 3rd edn. John Wiley & Sons (1994)
4. Basili, V.R.: The Experience Factory and Its Relationship to Other Improvement Paradigms. In: Sommerville, I., Paul, M. (eds.) ESEC 1993. LNCS, vol. 717, pp. 68–83. Springer, Heidelberg (1993)
5. Basili, V.R., Caldiera, G., Rombach, H.D.: The Experience Factory. Encyclopedia of Software Engineering, vol. 2, pp. 511–519. John Wiley & Sons (2002), `http://books.google.es/books?id=CXpUAAAAMAAJ`
6. Basili, V.R., Rombach, H.D.: The tame project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering 14(6), 758–773 (1988)

7. Basili, V.R., Zelkowitz, M.V., McGarry, F.E., Page, G.T., Waligora, S., Pajerski, R.: Sel's software process improvement program. IEEE Software 12(6), 83–87 (1995)
8. Briand, L.C., Daly, J.W., Wüst, J.: A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering 3(1), 65–117 (1998)
9. Briand, L.C., Differding, C., Rombach, H.D.: Practical guidelines for measurement-based process improvement. Software Process: Improvement and Practice 2(4), 253–280 (1996), `http://www3.interscience.wiley.com/journal/24853/abstract`
10. Briand, L.C., Morasca, S., Basili, V.R.: Property-based software engineering measurement. IEEE Transactions on Software Engineering 22, 68–86 (1996), `http://portal.acm.org/citation.cfm?id=229713.229722`
11. Briand, L.C., Morasca, S., Basili, V.R.: Defining and validating measures for object-based high-level design. IEEE Transactions on Software Engineering 25, 722–743 (1999), `http://portal.acm.org/citation.cfm?id=325392.325404`
12. Briand, L.C., Morasca, S., Basili, V.R.: An operational process for goal-driven definition of measures. IEEE Transactions on Software Engineering 28, 1106–1125 (2002), `http://portal.acm.org/citation.cfm?id=630832.631301`
13. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
14. DeMarco, T.: Structured analysis and system specification. Yourdon computing series. Yourdon, Upper Saddle River (1979)
15. Fenton, N., Pfleeger, S.L.: Software metrics: a rigorous and practical approach, 2nd edn. PWS Publishing Co., Boston (1997)
16. Fenton, N.E.: Software metrics - a rigorous approach. Chapman and Hall (1991)
17. Halstead, M.H.: Elements of software science. Operating and programming systems series. Elsevier (1977), `http://books.google.com/books?id=zPcmAAAAMAAJ`
18. Henry, S.M., Kafura, D.G.: Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7(5), 510–518 (1981)
19. Kendall, M.G., Stuart, A.: The advanced theory of statistics, 4th edn. C. Griffin, London (1977)
20. Krantz, D.H., Luce, R.D., Suppes, P., Tversky, A.: Foundations of Measurement, vol. 1. Academic Press, San Diego (1971)
21. Lakshmanan, K.B., Jayaprakash, S., Sinha, P.K.: Properties of control-flow complexity measures. IEEE Transactions on Software Engineering 17(12), 1289–1295 (1991)
22. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering 2(4), 308–320 (1976)
23. Morasca, S.: Refining the axiomatic definition of internal software attributes. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, Kaiserslautern, Germany, October 9-10, pp. 188–197. ACM, New York (2008), `http://doi.acm.org/10.1145/1414004.1414035`
24. Morasca, S.: A probability-based approach for measuring external attributes of software artifacts. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, Lake Buena Vista, FL, USA, October 15-16, pp. 44–55. IEEE Computer Society, Washington, DC (2009), `http://dx.doi.org/10.1109/ESEM.2009.5316048`

25. Morasca, S., Briand, L.C.: Towards a theoretical framework for measuring software attributes. In: Proceedings of the 4th International Symposium on Software Metrics, IEEE METRICS 1997, Albuquerque, NM, USA, November 5-7, pp. 119–126. IEEE Computer Society, Washington, DC (1997), http://portal.acm.org/citation.cfm?id=823454.823906
26. Musa, J.D.: A theory of software reliability and its application. IEEE Transactions on Software Engineering 1(3), 312–327 (1975)
27. Musa, J.D.: Software Reliability Engineering. Osborne/McGraw-Hill (1998)
28. Oman, P., Hagemeister, J.R.: Metrics for assessing a software system's maintainability. In: Proceedings of ICSM 1992, Orlando, FL, USA, pp. 337–344 (1992)
29. Oviedo, E.I.: Control flow, data flow and program complexity. In: Proceedings of the 4th Computer Software and Applications Conference, COMPSAC 1980, Chicago, IL, USA, October 27-31, pp. 146–152. IEEE Press, Piscataway (1980)
30. Pap, E.: Some elements of the classical measure theory, pp. 27–82. Elsevier (2002), http://books.google.je/books?id=LylS9gsFEUEC
31. Poels, G., Dedene, G.: Comments on property-based software engineering measurement: Refining the additivity properties. IEEE Transactions on Software Engineering 23(3), 190–195 (1997)
32. Poels, G., Dedene, G.: Distance-based software measurement: necessary and sufficient properties for software measures. Information & Software Technology 42(1), 35–46 (2000)
33. Prather, R.E.: An axiomatic theory of software complexity measure. The Computer Journal 27(4), 340–347 (1984)
34. Roberts, F.: Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences, Encyclopedia of Mathematics and its Applications, vol. 7. Addison-Wesley (1979)
35. Sauro, J., Kindlund, E.: A method to standardize usability metrics into a single score. In: CHI 2005: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Portland, Oregon, USA, pp. 401–409 (2005)
36. Shepperd, M.J.: Foundations of software measurement. Prentice Hall (1995)
37. Spector, P.E.: Research designs. Quantitative applications in the social sciences. Sage Publications (1981), http://books.google.com/books?id=NQAJE_sh1qIC
38. Stevens, S.S.: On the theory of scales of measurement. Science 103(2684), 677–680 (1946), http://www.ncbi.nlm.nih.gov/pubmed/16085193
39. Weyuker, E.J.: Evaluating software complexity measures. IEEE Transactions on Software Engineering 14(9), 1357–1365 (1988)
40. Yourdon, E., Constantine, L.L.: Structured design: fundamentals of a discipline of computer program and systems design, 2nd edn. Yourdon Press, New York (1978)

# Combining Evidence and Meta-analysis in Software Engineering

Martin Shepperd

Brunel University,
London, UK
martin.shepperd@brunel.ac.uk

**Abstract.** Recently there has been a welcome move to realign software engineering as an evidence-based practice. Many research groups are actively conducting empirical research e.g. to compare different fault prediction models or the value of various architectural patterns. However, this brings some challenges. First, for a particular question, how can we locate *all* the relevant evidence (primary studies) and make sense of them in an unbiased way. Second, what if some of these primary studies are inconsistent? In which case how do we determine the 'true' answer? To address these challenges, software engineers are looking to other disciplines where the systematic review is normal practice (i.e. systematic, objective, transparent means of locating, evaluating and synthesising evidence to reach some evidence-based answer to a particular question). This chapter examines the history of empirical software engineering, overviews different meta-analysis methods and then describe the process of systematic reviews and conclude with some future directions and challenges for researchers.

## 1 Introduction – A Brief History of Empirical Software Engineering

Software engineering (SE) is a relatively new discipline with the term only having been coined just over 40 years ago at the now famous 1968 NATO Conference [38]. Its focus is the application of theory from disciplines such as computer science to the practice of developing non-trivial software systems for real users under real constraints. This was the consequence of a growing realisation that scale was the 'enemy' and that techniques and approaches suitable for the implementation of small algorithms did not necessarily scale up. Figure 1 shows a simplified timeline of developments in empirical software engineering.

Over the next two decades there was rapid growth in ideas for design, implementation and testing methods and tools, new representations and models, for good practice and pathological code structures, for effective project management and dialogue with clients, for software reuse and mathematical means of reasoning about the correctness of code [50]. Many claims and counter-claims were made for competing approaches. However, evidence was primarily anecdotal and based on the opinions of experts — often self-proclaimed. From this

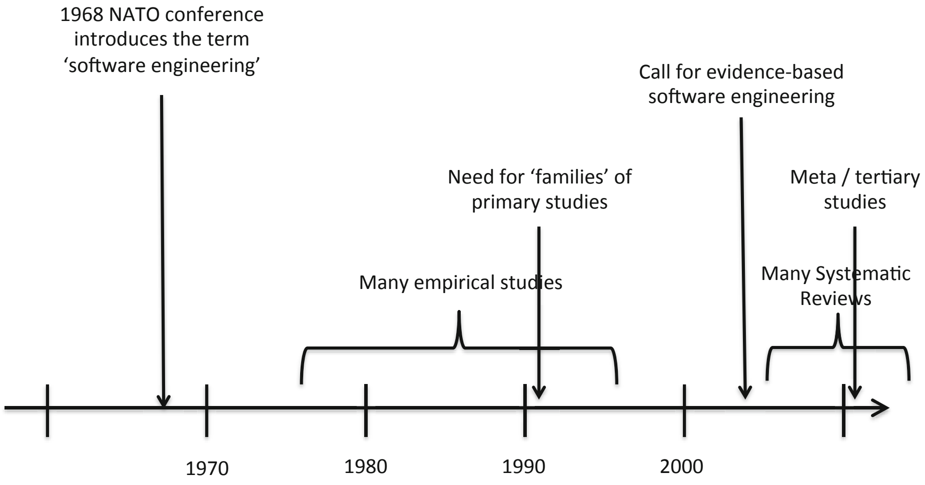

1968 NATO conference introduces the term 'software engineering'
Call for evidence-based software engineering
Need for 'families' of primary studies
Meta / tertiary studies
Many empirical studies
Many Systematic Reviews
1970
1980
1990
2000

**Fig. 1.** A Timeline for Empirical Software Engineering

period of great activity emerged an appreciation of the need to empirically evaluate these techniques and methods. From small beginnings in the 1980s arose a number of dedicated conferences and journals to the topic of empirical software engineering and a vast number of outputs[1]. Unsurprisingly, for an emerging discipline, some of the work was *ad hoc* and it was often difficult to discern an overall pattern.

By the 1990s the notion of explicitly constructing a body of empirical evidence began to gain momentum. At the forefront were the ideas of Basili and his co-workers with the proposal that individual primary studies should be seen as part of a "family of studies" rather than isolated activities [4]. Thus, studies could be replicated, context variables varied and results refined so that a framework for organising related studies could be built. Such an approach relies upon some narrative-based synthesis of the results and so did not fully solve the problem of how this might be accomplished in a rigorous fashion. It also relies upon having an appropriate set of initial studies for replication.

In parallel, other researchers such as Hayes [22], Pickard et al. [42] and Miller [37] started to consider the extent to which empirical results might be pooled for meta-analysis. The difficulty they all identified was that few primary studies provide access to raw data, or sufficient experimental details; consequently, pooling was not possible or extremely difficult. Indeed Pickard et al. concluded that without agreed sampling protocols for properly defined software engineering populations, and a set of standard measures recorded for all empirical studies,

[1] A simple search of the google scholar bibliographic database using the terms 'empirical' AND 'software engineering' retrieved more than 45000 hits and whilst I acknowledge this is not a sophisticated research tool it is indicative of substantial research effort in this area.

meta-analyses should not be conducted [42]. So for these reason meta-analysis of primary studies within SE has not been pursued until recently.

Nevertheless, in the following decade, there has been a move to explicitly position software engineering as an *evidence-driven* discipline. At the forefront was Kitchenham *et al.* [32] and Dybå *et al.* [16]). Their ideas were strongly influenced by clinical practice and the formation of the Cochrane Collaboration [10]. Kitchenham and Charters [28] then went on to formulate a set of guidelines for systematic reviews specifically for software engineers and these were principally derived from clinical practice. Their ideas quickly took hold and in the last six or seven years many systematic reviews have been conducted across a wide range of topics.

Kitchenham et al. [27,30] performed a systematic review of systematic literature reviews published between January 2004 and June 2008 (known as a meta-review or tertiary study) and found a total of 53 studies that satisfied their quality and inclusion criteria. This has recently been updated by da Silva et al. [49] to include the period up until the end of December 2009 who found an additional 67 new studies making a total of 120 in period of just six years with a dramatic increase in the rate of activity towards the end of that period. Both groups have reported that quality — essentially meaning adherence to some defined method for conducting the review — appeared also to be increasing. By contrast, in yet another tertiary study, Cruzes and Dybå [13] were somewhat more critical of the prevailing state of affairs. In particular they found that there was limited attention actually paid to the synthesis of results as opposed to merely cataloguing what existed. Less than half of all Reviews contained any synthesis, and of those that did, a narrative or relatively informal approach predominated. In other words, researchers are effective at locating primary studies but less so at combining their results to form some kind of conclusion. There are at least two contributory factors. First, the methods and reporting styles of many of the primary studies are so diverse as to make synthesis difficult, that is the problems noted by [42] persist. Second, many Reviews are not based on single answerable questions but instead seek to understand research activity in a particular field. These are known as Mapping Studies.

Presently, the range of Review topics is extremely diverse, ranging from mapping studies of distributed software development and agile software development, to a comparison of regression modelling with case-based reasoning for project prediction. Undoubtedly these reviews collectively provide a valuable resource and a basis for empirical software engineers to take stock of what has been accomplished to date and to plan effectively for the future. It is difficult to see how SE can proceed as a scientifically-based subject without careful cataloging and reappraisal of the many primary studies being conducted and published.

The remainder of this chapter is organised as follows. The next section examines what we mean by scientific, empirical evidence, how we can appraise its quality and how we can integrate more than one item of evidence (meta-analysis). Section 3 then looks at the complete Systematic Review process and how this incorporates elements of evidence appraisal and meta-analysis as well addressing basic questions such as what is goal of the review. The section concludes with a brief look at two international groups associated with the production, validation and publication of Systematic Reviews, namely the Cochrane and Campbell Collaborations. In Section4 we then examine how this specifically applies to empirical software engineering. The chapter concludes by considering the opportunities and challenges for Systematic Reviews in the future.

## 2 Using Evidence

In everyday language, evidence is an extremely broad notion simply meaning anything that might be used to show the truth of some assertion, however, within this chapter I concentrate upon scientific evidence and, specifically, *empirical* scientific evidence. By scientific I mean evidence that has been obtained in accordance to generally accepted principles, that is by adhering to some method and secondly that it is documented in a full and standard manner. By empirical I mean evidence that is derived from observation. For this reason one would not generally admit anecdote or opinion as scientific evidence. This is not a reason to deprecate qualitative data which when rigorously collected, analysed and documented can yield valuable, high quality scientific evidence.

### 2.1 Types of Evidence

Empirical scientific evidence may take many forms. It may be quantitative or qualitative or a mixture. It is also important to consider the method by which the evidence is obtained as this provides its meaning and significance. Classic methods include: controlled experiments and quasi-experiments; case studies; surveys; action research and ethnography. These are well documented elsewhere, see for instance [3,5,47,56,58]. One important point to stress is that despite such neat classification schemes many studies do not exactly fall into one category. Nor is there unanimity regarding definitions although the principles of objectivity through dealing with sources of bias, transparency and repeatability are widespread.

Nevertheless the concept of a hierarchy of evidence (in terms of desirability and trustworthiness) is quite widespread. Typically, evidence from systematic reviews and randomised controlled experiments (RCTs) are placed at the top of the hierarchy, whilst evidence from expert opinion is placed at the bottom. The underlying idea is that this will help us weigh or select evidence accordingly. However this can be a very simplistic approach. There are many reasons why formal experiments may not be available or appropriate such as:

– there may ethical reasons why the allocation of participants to a particular treatment may be considered inappropriate, and this problem is not limited to clinical research, for example, withholding an educational opportunity from a group of students in order to provide a control might well be considered unethical.

| | **Effectiveness** (how well does the intervention work?) | **Appropriateness** (how suitable is the intervention in the context?) | **Feasibility** (the wider organisational issues of introducing the intervention) |
|---|---|---|---|
| **Excellent** | Systematic review<br>Multi-centre study | Systematic review<br>Multi-centre study | Systematic review<br>Multi-centre study |
| **Good** | RCT<br>Observational study | RCT<br>Observational study<br>Interpretive study | RCT<br>Observational study<br>Interpretive study |
| **Fair** | Uncontrolled trial with dramatic results<br>Before/after study<br>Non-randomised controlled trial | Descriptive study<br>Focus group | Descriptive study<br>Action research<br>Before/after study<br>Focus group |
| **Poor** | Descriptive study<br>Case study<br>Expert opinion<br>Poor quality study | Case study<br>Expert opinion<br>Poor quality study | Case study<br>Expert opinion<br>Poor quality study |

**Fig. 2.** An Evidence Hierarchy (Adapted from [19])

- some interventions may be perceived as, or simply are, unpleasant and therefore hard to recruit volunteer participants.
- researchers may not have control over the allocation of the study units, due to reasons of cost or lack of influence. For example, a company will decide what software development method or programming language they intend to use for a particular project and this will usually be beyond the investigator's control.
- the effect may be very infrequent.
- dealing with a very new or emerging idea that appears significant so we wish to use the available evidence immediately.

So whilst there is some presumption that randomised controlled trials (RCTs) are the best quality empirical evidence, often referred to as the "gold standard", the above list indicates some reasons why they may not always be the most appropriate form of evidence. By extending the hierarchy to take account of the nature of research enquiry Evans [19] has produced a matrix (see Figure 2) which might be helpful if appraising the value of empirical evidence, though we might view well conducted case studies in a slightly more generous light than the matrix suggests.

In addition, many qualitative researchers strongly argue that other techniques such as ethnography enable a richer, fuller picture of the phenomena of interest and this may be of particular relevance where the basic constructs are not fully understood, e.g. quality of life, professional experience, etc. There has been some interest in using such empirical methods within software engineering for example [5,55] and more generally, case studies, surveys and interviews. Seaman [46] gives a useful overview of using methodologically sound qualitative methods within software engineering.

Examples of observational studies are (i) using data from an existing database, (ii) a cross-sectional study, (iii) a case series, (iv) a case-control design, (v) a design with historical controls, (vi) a cohort design and (vii) an ethnographic study. Such studies may also be more effective for assessing the effect of an intervention *in vivo* as opposed to *in vitro*.

To summarise, there is a huge diversity of differing empirical research methods. Their suitability for providing high quality evidence is largely dictated by relevance to the given research question and so the notion of some immutable hierarchy that some types of evidence are always preferable is rather simplistic.

### 2.2 Appraising Evidence Quality

What we have not yet considered is the strength of the evidence deriving from (i) the importance that we might view as being inherent in the type of empirical research and (ii) how well that research has been conducted including its relevance to the research question at hand. An example of the former might be, we would attach less weight to an anecdote from an expert than to a RCT. Examples of the latter might be RCTs with no, or ineffectual, blinding or observational studies with unbalanced treatments and a confounder. To ignore study quality seems particularly risky as it may result in highly misleading evidence becoming influential in our decision making. Unfortunately many independent studies have shown the prevalence of statistical and experimental design errors and / or poor reporting, see for example Yancey [57], Altman [1] and specifically within the context of empirical software engineering, Kitchenham et al. [31].

Most approaches to assessing the quality of a primary study are based on the use of checklists or scoring procedures where individual items are summed to produce an overall score. However, in a meta-analysis of 17 primary studies (all RCTs) using no less than 25 different quality instruments Jüni et al. [25] found that the results differed widely. By comparing the difference in effect between the 'low' and 'high' quality studies Jüni et al. investigated the extent to which the different quality instruments detected bias. For example, was it possible that that low quality studies were more likely to detect an effect, perhaps because of researcher bias or poor blinding strategies?. Unfortunately they found no clear pattern at all, with some studies reporting a negative difference in effect between 'low' and 'high' quality studies, others a positive difference and the majority no difference at all. The authors concluded that using overall summary scores to evaluate primary studies was 'problematic' and that researchers should focus only on those aspects of a study design that were relevant and important to their particular research question.

Determining the quality of qualitative or mixed data is even more challenging not least because of the many different philosophical and evaluative traditions that exist. One useful approach based upon a framework of 18 questions was produced for the UK government [51]. In order to make the instrument more generic the questions are aligned to high level aspects of a study which, in theory would be present irrespective of specific details of the methodology. These cover design, sampling, data collection, analysis, findings, reporting and neutrality. An example of a question on sampling is:

> How well defined is the sample design or target selection of cases or documents?

Suggested quality indicators are then provided for each question, so for the above, we have non-exhaustively:

> Description of study locations / areas and how and why chosen.
> Description of population of interest and how sample selection relates to it (e.g. typical, extreme case, diverse constituencies etc.)

Naturally there is something of a trade off between the generality of a quality instrument (i.e. how widely applicable it is) and its specificity. However, even the most detailed of schemes will still require appraisers to exercise their judgement. Interestingly, the Cochrane Collaboration in their handbook [23] do not recommend either a scale nor a checklist, but instead domain-based evaluation. This encourages the appraiser to consider a series of sources of bias under the following headings:

- selection bias (of participants)
- performance bias (e.g. due to lack of blinding)
- detection bias (e.g. due to lack of blind analysis)
- attrition bias (e.g. due to drop outs or missing data)
- reporting bias (e.g. due to selectivity of the researchers)
- other sources of bias (a catch all for other problems not accounted for above)

Of course the instrument is only a guideline and still requires a good deal of skill and judgement to apply it. In other words, there seems no simple mechanical shortcut to determining primary study quality, much as some meta-analysts might wish otherwise! This topic is re-visited in Section 3 where it is formally embedded as part of the Systematic Review process.

## 2.3 Meta-analysis

Suppose we have more than one item of evidence? Indeed, suppose we have a body of evidence which might be expected given science is — or at least usually is — a cooperative endeavour? And, suppose these items are not necessarily consistent? Results from different primary studies may vary for many reasons, such as differences in:

- research designs
- the sample characteristics leading to biases in how participants are sampled from underlying population

– the intervention e.g. if we're interested in the impact of using object-oriented architecture and staff on different projects having undertaken different training courses or even simply having read different books
– measures of the treatment effect or response variable e.g. an organisation where overtime is unpaid may record effort very differently to where overtime is remunerated
– context or setting e.g. safety critical systems versus games software.

So how do we combine multiple results, otherwise known as research synthesis? The idea of meta-analysis, i.e. pooling results from more than one primary study was first proposed by the statistician Karl Pearson in the early 20th century in which he combined a number of studies of typhoid vaccine effectiveness and then conducted a statistical analysis. Thus meta-analysis is a quantitative form of research synthesis. Although clinical subjects dominate, areas of application span from "astronomy to zoology" [41].

Essentially there are two purposes to meta-analysis. First is to assess the treatment effect (or effect size) that is common between the included primary studies. But, second is to assess the variability in effect between studies, and then to try to discover the reason for this variation. When there is variation between studies we say that there is heterogeneity. If this is not properly investigated this can threaten the validity of the meta-analysis. Plainly, some variation will arise through chance so typically it must be measured using a statistic such as $I^2$ [24] which shows the proportion of variation across studies that is due to heterogeneity rather than chance. Alternatively heterogeneity can be explored visually by means of a forest plot.

A basic approach is to compare effect size from the various primary studies in some standardised way. For a binary effect (e.g. has / does not have the disease) this is typically accomplished using odds ratios. For a continuous effect (e.g. productivity) this may be done using an effect size measure such as difference between means standardised by the pooled standard deviation or that of baseline study [18,21]. If there is high between-study variance then the primary study characteristics such as the population sampled, or type of study (if there are different types of design) can be coded. These characteristics are then used as independent variables to predict the differences in effect size. In this way it may be possible to correct for some methodological weaknesses in studies. Alternatively there may be evidence that there are distinct populations being sampled e.g. different countries or ages in which case partitioning the studies and re-analysing may be the best way forward. In this case the outcome will be answers to more specific or local research questions. Different techniques have been proposed for modelling and understanding the variation in effect (within and between studies) and these include meta-regression and Bayesian meta-analysis (see Sutton and Higgins for a more in depth discussion [54]).

Whilst one might have the impression that since randomised controlled trials (RCTs) are deemed to form the "gold standard" and there would be little purpose in trying to synthesise other weaker forms of evidence, there has recently been something of a sea change. For example, Dixon-Woods et al. state "[p]olicy-makers and practitioners are increasingly aware of the limitations of regarding randomised controlled trials as the sole source of evidence." [15]. As previously discussed there are many compelling reasons why some or all of the primary studies may not be experiments or RCTs.

There are are various techniques for meta-analysis of evidence from observational studies and in particular for dealing with qualitative data. Dixon-Woods et al. [15] provide a very useful summary and a table of the strengths and weaknesses of different meta-analytic techniques aimed at either qualitative or mixed primary data (e.g., content analysis, meta-ethnography, narrative analysis of qualitative research reports and even Bayesian meta-analysis).

Thematic analysis is one possible tool for such meta-analysis. Thematic analysis — a common technique used in the analysis of qualitative data within primary research — can be adapted to systematically identify the main, recurrent or most important (based on the review question) concepts across the multiple studies that are part of the meta-analysis. It is flexible and might be seen as a relatively 'lightweight' qualitative technique. However, this strength can also be a potential weakness due to lack of clear and concise guidelines leading to an "anything goes" approach. For a very brief overview see [2] and for more detailed account with an example see [7].

Related are the guidelines on producing a narrative research synthesis by Popay et al. [43]. The writers stress that they do not see narrative synthesis (NS) as "second best" nor the research synthesis of last resort but a technique that is not only complementary to more traditional quantitative meta-analysis but also valuable in increasing the likelihood of the findings being adopted by practitioners or policy-makers. The goal is "bringing together evidence in a way that tells a *convincing* story [my emphasis]" [43]. As the name implies the primary input is the use of words and text in order to summarise primary studies and explain the synthesis findings. The NS should focus on two aspects of a research question, namely the *effects* of the intervention and factors that influence the *implementation* of the intervention. The guidelines also suggest four main elements to the narrative or story. Quoting from the Guidelines [43] these are:

- Developing a theory of how the intervention works, why and for whom
- Developing a preliminary synthesis of findings of included studies
- Exploring relationships in the data
- Assessing the robustness of the synthesis

Combining them produces a $2 \times 4$ matrix which gives a purpose for each subcomponent of the overall narrative. A distinctive feature of NS is the emphasis upon theory building, something that can be neglected in meta-analysis. The Guidelines describe many different techniques that can be deployed including clustering, thematic and content analysis. However, it suggests a starting point is to summarise each primary study as a single paragraph description. Discovering patterns from these descriptions, particularly if there are a large number of studies is difficult so other techniques such as tabulation and clustering come into their own.

The challenge is that Narrative Synthesis is not based on any generally accepted deep theory and so is vulnerable to individual bias and consequently result in wrong conclusions. For this reason the Cochrane Collaboration has expressed some caution about its use, nevertheless, theNS Guidelines contain a great deal of useful advice, and might usefully be integrated into a formal Systematic Review.

In terms of what should be reported, Stroup et al. [53] give a detailed checklist under six main headings which given its breadth in many ways mirrors a full Systematic Review described in Section 3. Specific to the actual analysis part of meta-analysis are the following:

- an assessment of heterogeneity [of the primary study results]
- description of statistical methods (e.g., complete description of fixed or random effects models, justification of whether the chosen models account for predictors of study results ...)
- graphical summaries of individual study estimates and the overall estimate
- tables giving descriptive information for each included study
- results of sensitivity testing (e.g., subgroup analysis)
- an indication of the statistical uncertainty of the findings

However, some researchers remain very critical of the use of meta-analysis on observational studies, most notably Shapiro [48] who argues that particularly when dealing with small effects such an analysis is extremely vulnerable to the bias of each primary study (since participants are not randomly allocated to treatments) and other possible confounders. He concludes that meta-analyses should *only* performed on controlled experiments. Nevertheless, this is something of a minority position since it would require us to throw away much relevant and high quality evidence and would be particularly difficult for many open or complex research questions.

The general conclusion is that meta-analysis is a powerful tool but, unsurprisingly, can be abused when:

- if it is used when there is no underlying model or theory (for example, Egger et al. [17] flag up a meta-analysis that suggests causality between smoking and suicide rates which is biologically implausible, and in any case it is the social and mental states predisposing to suicide that are also co-vary with smoking. Nevertheless it can be easy to persuade oneself of the 'plausibility' of some set of results *post hoc*.
- the precision is over-emphasised. Again Egger et al. put it "we think that the statistical combination of studies should not generally be a prominent component of reviews of observational studies." [17].
- meta-analysis cannot turn poor quality studies into 'gold'. If a primary study is of very low quality then its conclusions are jeopardised irrespective of how it is merged with other studies [48].
- sources of heterogeneity are ignored e.g. compare variance between types of study, use of blinding, ... Address with sensitivity analysis, especially mixed or random effects model.

Finally, irrespective of how 'well' the meta-analysis is performed, the problem remains that if we are unsystematic in the way that we select evidence (from empirical primary studies) we are exposing ourselves to various sources of bias. In particular we need to consider the *quality* of evidence (this was addressed in the previous section) and that we select *all* relevant evidence which is the goal of a Systematic Review which will be discussed in the next section.

## 3 Systematic Reviews

It would be highly unusual for a single primary study to provide a definitive answer to any question of much significance to practitioners. Therefore we need to combine results from multiple primary studies. Traditionally researchers have sought to achieve this by means of narrative reviews. These are characterised by:

– convenience samples of relevant studies
– informal description of studies
– informal synthesis techniques such as subjective opinion
– reliance upon primary studies, which may be of low quality
– reliance upon primary studies that lack sufficient power to detect effects, hence they will be perceived as confirmation of the null hypothesis (i.e. no effect)
– opaque reasoning that is difficult to submit to independent scrutiny

Although the issues of informal synthesis and opaque reasoning are addressed by a formal meta-analysis, the other difficulties remain. Clearly such haphazard approaches can lead to a good deal of bias. This bias may arise from multiple sources including:

**Confirmation bias** is the tendency of researchers to confirm their prior belief or hypothesis [40].

**Publication bias** is the disproportionate likelihood of 'positive' results being published and the problem of 'negative' or 'neutral' results never being written up, otherwise known as the "File Drawer problem" [44].

**Preference bias** (Luborsky et al. [33] found in a meta-analysis of 29 primary studies in psychotherapy that almost 70% of the outcome variability could be predicted by the research team's 'allegiance' to a particular therapy or technique. Whilst some of this behaviour may be down to sheer prejudice it is also likely that 'allegiance' may also be a proxy for expertise. Many modern techniques are complex and require a great deal of expertise to apply effectively. This phenomenon has also been noted in machine learning where Michie et al. [36] comment that "in many cases authors have developed their own 'pet' algorithm in which they are expert but less so in other methods" which confounds comparison studies.

**Table 1.** A Comparison of Narrative Reviews, Meta-analysis and Systematic Reviews (adapted from [39])

| **Traditional Review** | **Meta-analysis** | **Systematic Review** |
|---|---|---|
| Method implicit | Research synthesis may be explicit | Method explicit in the Review protocol |
| No real question / lack of focus | Limited focus | Answers focused on specified research question |
| Biased / ad hoc selection of studies | Biased / ad hoc selection of studies | All relevant primary studies included |
| Unquestioning acceptance of primary study results | Possible appraisal of study quality | Quality of primary studies systematically appraised |
| No justification of reviewer conclusions | Conclusions based on meta-analysis | Conclusions based on the data derived (might involve a meta-analysis) |
| Results presented in black and white terms without indicating the uncertainty or variability | Results illustrate the hetero-geneity of data allowing quantification of uncertainty | Results illustrate the hetero-geneity of data allowing quantification of uncertainty |

Not only are narrative reviews[2] vulnerable to bias, but the societal cost deriving from a biased or incomplete or non-rigorous analysis may be substantial. In medicine this might, and indeed was shown to, lead to inappropriate clinical interventions resulting in harm to patients. In areas such as social policy and education, sub-optimal decisions might be made to the detriment of society. And likewise in software engineering, given the ubiquity of software-intensive systems in modern life, again the scope for harm (when such systems do not perform as intended) or wasted opportunities and resources (as a result of cost overruns and inefficient methods and procedures) is vast.

The growing appreciation of these problems, first in medicine and then in other areas, has led to the popularising of the Systematic Review. The late Archibald Cochrane, a leading epidemiologist, challenged the medical community back in the early 1970s concerning the absence of summaries of "all relevant randomised controlled trials [RCTs]" within a particular topic. Ultimately this resulted in the establishment of the Cochrane Collaboration [10] (named in honour of Cochrane) in 1993. This Collaboration was (and is) predicated upon the notion of performing Systematic Reviews to synthesise results from all relevant primary studies where typically RCTs were seen as the the highest quality of primary study due to their blinding and design to avoid bias. Table 1 contrasts the three approaches.

Whilst different groups and methodologists[3] propose varying numbers of steps and stages, overall there is a good deal of similarity and the actions needed can

[2] Narrative reviews, as opposed to narrative synthesis which was described in Section 2.3.

[3] See for example, The Cochrane Collaboration [23], Sackett et al. [45], Needleman [39] and Kitchenham et al. [28].

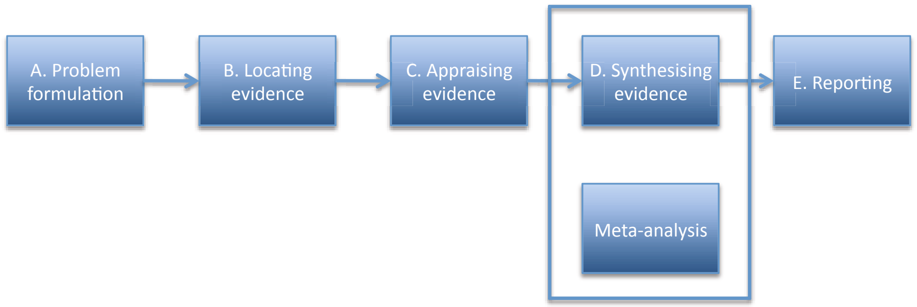


**Fig. 3.** The Five Stages of a Systematic Review

be mapped to five steps (as outlined by Cooper [11] and extended in [12] and shown diagrammatically in Figure 3. Note that the neat stages and absence of any backtracking or parallelism will in practice be illusory.)

The five basic steps therefore are:

**A. Problem formulation** This involves clarifying the purpose of the proposed Systematic Review and carefully specifying the research question(s) in a review protocol.

**B. Locating evidence** This involves a search of the literature and extraction of information from relevant primary studies. Systematic reviews very much emphasise the need to find, and use, *all* relevant evidence.

**C. Appraising evidence quality** A Systematic Review will have explicit quality inclusion criteria so that the overall synthesis is not contaminated by low quality studies.

**D. Evidence synthesis and interpretation** Relevant data must be extracted from the primary studies as a preliminary to the 'research synthesis' which involves some procedure — not necessarily quantitative — to make inferences with respect to the initial research questions based upon the totality of the evidence located. Note that, as indicated in Figure 3 meta-analysis is a form of research synthesis and might or might not be part of a Systematic Review.

**E. Reporting** A Review Report will contain some narrative, statistics (even if only simple descriptive statistics), graphs, charts and diagrams, tables, discussion of the findings and a list of threats to validity.

We now discuss each of these stages in more detail.

**A. Problem Formulation.** This involves clarifying the purpose of the proposed Systematic Review and carefully specifying the research question(s). Because the research question is so central to the whole conduct of a Survey many researchers find it helpful to adopt the PICO structure [39] where the question should state the **P**opulation, **I**ntervention, **C**omparison and **O**utcome. An example taken from a Systematic Review[35] on comparing the use of local and global data for software project effort prediction systems is given in Table 2.

**Table 2.** Example PICO Definition from MacDonell and Shepperd Review Comparing Local and Cross-Company Databases [35]

| | |
|---|---|
| Research question: | What evidence is there that cross-company estimation models are at least as good as within-company estimation models for predicting effort for software projects? |
| PICO definition: | |
| **P**opulation: | Local and global data sets relating to non-trivial, commercial software projects |
| **I**ntervention: | Effort estimation modelling – using global data |
| **C**omparison intervention: | Effort estimation modelling – using local data |
| **O**utcomes: | More accurate models, reduced bias in effort estimation |

The research question and purpose should be detailed in the review protocol which is a public statement of how the Review will be conducted. The very first thing must be to consider the significance of the proposed question(s) and how interesting the answer will be. Who are the stakeholders and how will they benefit from a rigorous answer to the research question? Are other research groups tackling the same question thereby rendering the proposed Review redundant?

A well conducted review is highly resource intensive so the opportunity cost will be considerable. In addition it is appropriate to try to estimate the likely size of the body of evidence. Obviously this will not be definitively known until after a search has been performed, however, if it is likely to be very small (perhaps less than five primary studies) a Mapping Study may be a more suitable approach. On the other hand if a large body of evidence is anticipated, the question of whether commensurate resources are available for Review arises. A half completed Review is of absolutely no value to anyone. A slapdash Review has negative value.

Another important topic that should be explicitly set out in the protocol is the inclusion / exclusion criteria for primary studies.The danger, if they are not clear, is that individual members of the Review team may make differing decisions or that there may be drift over the course of the study and lastly the context of the Review may be unclear to its consumers (both practitioners and consumers).

The protocol should also define the proposed Review process such as the databases to be searched and what validation procedures will be employed. More details of a typical structure, as mandated by the Campbell Collaboration, are given in Section 3.3. It is important since it firstly enables feedback and improvement prior to the commencement of the Review, secondly it provides a clear and detailed description for the scrutiny of other researchers and thirdly it serves as a reference statement for the Review team to prevent misunderstandings or drift.

**B. Locating Evidence.** This second phase involves the literature search and extraction of information from relevant primary studies. With the current widespread availability of bibliographic databases such as sciencedirect and specialised search engines such as google scholar this will almost certainly involve automated searching using one or more search queries. Systematic reviews very much emphasise the need to find, and use, *all* relevant evidence.

The challenge is to devise a search with high precision *and* recall. By this I mean a query that is efficient and does not retrieve large numbers of irrelevant studies that have to be hand checked, but at the same time has good coverage and so does not miss relevant studies. Usually there is something of a tradeoff between these two goals. Factors that contribute to the ease (or difficulty) of searching include the existence of an agreed terminology and bodies of evidence located within a single discipline.

Sometimes it may be effective to structure the search query along the lines of the research question particularly if a PICO structure has been employed. This might result in a query something like:

```
(<Pop term1> OR <Pop term2> ... ) AND (<Int term1> OR <Int term2> ...) AND
(<Comp term1> OR <Comp term2> ... ) AND (<Out term1> OR <Out term2> ...)
```

One technique for assessing the quality of a search is the capture-recapture method which whilst it may not give a precise estimate of the size of the missed literature may be useful in providing a 'ballpark' figure [52]. Even where this is not appropriate or possible, the underlying concept can be used to test the effectiveness of a search strategy by assessing its ability to retrieve previously known studies. If known studies are missed then the search should be augmented.

On occasions where Published reference lists should be scanned to determine if other relevant studies exist. Next authors of retrieved studies should be approach to discover if they have produced other relevant studies missed by the automated search, for example, because they are in press.

Having retrieved what is typically a large number of studies these then must be scrutinised for relevance according to the explicit inclusion criteria of the protocol. Often this can be accomplished by successively more detailed scrutiny so for instance the first pass scans the title and journal / conference, the next pass might read the abstract and the final shortlist of candidate papers will be read in their entirety. All decisions made should be documented so that a full audit trail is available. In addition since applying the inclusion criteria generally requires some degree of judgement it is a good idea to independently repeat some or all of the process to assess the degree of consistency. Formal measures of inter-rater reliability such as Cohen's $\kappa$ should be used.

**C. Appraising Evidence Quality.** A criticism that is often made of the narrative review is a tendency towards unquestioning acceptance of all located primary studies. In contrast, a Systematic Review will have explicit quality inclusion criteria so that the overall synthesis is not contaminated by low quality studies. As previously discussed in Section 2.2, many instruments have been proposed to help researchers assess candidate studies for inclusion. In a pleasingly circular way, [26] carry out a systematic review of different approaches for determining primary study quality. They found an astonishing 121 different instruments just in the field of medicine but disappointingly found little empirical evidence as to the validity of differing techniques nor that there was any "gold standard" critical appraisal tool for any study design, or any widely accepted "generic tool that can be applied equally well across study types". Therefore their advice was that the "interpretation of critical appraisal of research reports currently needs to be considered in light of the properties and intent of the critical appraisal tool chosen for the task" [26].

Typical quality instruments tend to take the form of additive checklists where a study receives some score out of $n$ where $n$ is the number of items within the list. Full compliance scores one, partial a half and no compliance zero. A good example in software engineering is the list used by Kitchenham et al. in their review of local versus cross-company data-sets for cost prediction [29]. The dangers are, as some commentators have pointed out, that some choices are highly arbitrary (e.g. why is a dataset of $n < 10$ judged to be poor? Why not 8 or 11? Another problem is the one-size fits all mentality particularly where there are mixed types of primary study. In addition many concerns have been raised to the approach of summary scores [25].

Rather than simply using the quality instrument as a threshold for inclusion, it is recommended that (i) researchers identify those aspects of primary study quality that are *relevant* and *important* to the research question and (ii) any analysis of effect be performed both with and without the lower quality studies. This at least highlights whether such studies do 'contaminate' the Review. What is not recommended is to weight results by quality and the use of compound scales and checklists [39].

**D. Evidence Synthesis and Interpretation.** Data extraction from the primary studies should involve the use of a form or data collection spreadsheet with appropriate headings and categories. Wherever subjective judgement is involved some sort of validation is required usually by comparing two independent checkers. This implies that two researchers are the minimum for any Systematic Review.

Next, interpreting the evidence that has been selected is central to the whole review process. Many writers refer to this as 'research synthesis' which involves some procedure — not necessarily quantitative — to make inferences with respect to the initial research questions based upon the totality of the evidence located.

**E. Reporting.** Some communities have detailed reporting protocols for Systematic Reviews. Whatever the minutiae it would be expected that a Review Report will contain some narrative, statistics (even if only simple descriptive statistics), graphs, charts and diagrams, tables and threats to validity.

## 3.1 Tertiary Reviews and Mapping Studies

Closely related to Systematic Reviews are Tertiary Reviews and Mapping Studies. A Tertiary Review or Meta-Review is a Systematic Review of Systematic Reviews and might be conducted to explore methodological or quality related questions, for example, what topics are being explored, what types of primary study are included, or what quality instruments are used to assess primary studies? Such reviews will follow the same methodology as a regular Systematic

Review. Appendix C of the Kitchenham and Charters Guidelines [28] contains an example of protocol specifically for a Tertiary Review. A recent example in software engineering is the Tertiary Review by Cruzes and Dybå[13] that investigated how Systematic Reviews the process of synthesis and concluded that this was largely neglected.

In contrast, a *Mapping Study* tends to be more general than a Systematic Review, focusing upon a research topic rather than a specific question (although clearly there is something of a continuum between the two). They are more appropriate when either the goal is some broad overview or understanding of research activity in a field or where little empirical evidence is expected to be found. These studies are therefore potentially useful in guiding future research to areas of need. They may also be relevant where important concepts are poorly understood, so for example in software engineering we still have little consensus as to what constitutes software productivity or what are the major factors influencing it. This means agreeing what might be a reasonable response variable and what other factors should be taken into account may well prove so challenging that a focused Systematic Review is unlikely to be a suitable research tool.

## 3.2 Cochrane Collaboration

The Cochrane Collaboration was established in 1993, and is an international non-profit and independent organisation, with the goal of making the most recent, accurate and informed evidence about healthcare available to clinicians, policy-makers and the public. It achieves this through systematic reviews which are made available in the Cochrane Library [9]. It also promotes good practice and hosts a wealth of educational material on the various methodological issues associated with systematic reviews and meta-analysis. This is now a major operation with the Library containing many thousands of reviews. It is fair to say that the Collaboration has had a transformative effect, not only upon medicine, but also may other unrelated disciplines. For more information visit http://www.cochrane.org.

## 3.3 Campbell Collaboration

This is loosely modelled on the better known Cochrane Collaboration and is concerned with systematic reviews in the areas of education, criminal justice and social welfare. Whilst there is much in common with medical systematic reviews, there are also substantive differences. Specifically these are the fact that primary studies are seldom randomised controlled trials (RCTs) and secondly, the outcomes of interest are typically a good deal more complex (e.g. reduction in the fear of crime) than survival rates or odds ratios. As a consequence, determining suitable rand comparable response variables can in itself be highly challenging. Presently the majority of Campbell reviews contain non-RCTs and determining how to integrate such evidence into a Review is a particular challenge for researchers in this field.

A protocol for a Campbell review should consist of the following sections:

1. Background for the review
2. Objectives of the review and the review question
3. Methods of the review
4. Timeframe
5. Plans for updating the review
6. Acknowledgements
7. Statement concerning conflict of interest
8. References and tables

Draft protocols are publicly available and subject to a formal review process prior to their approval as Campbell Collaboration (C2) authorised systematic reviews. Any deviation from the agreed protocol should be documented and justified in the final report. After the review has been completed it is again subject to review by subject and review methods experts. Once it is approved it is added to the Campbell Library with the option of additional publication elsewhere.

It is interesting to note that typical reviews lead to reports of the order of 85 pages, which is considerably in excess of the customary conference or journal paper in empirical software engineering (see Table 3.3). It may be that the importance of the review protocol is under-appreciated within software engineering.

**Table 3.** Campbell Collaboration Review Reports: (The 10 most downloaded reviews - accessed 17.5.2011)

| Review Topic | Pages |
|---|---|
| Bullying | 147 |
| Parent involvement | 49 |
| Cognitive behaviour | 27 |
| Delinquency | 88 |
| Cyber abuse | 54 |
| After school programmes | 53 |
| Mentoring | 112 |
| Kinship care | 171 |
| Parental imprisonment | 105 |
| Correctional boot camps | 42 |
| Mean | 85 |
| Median | 71 |

## 3.4 Reflections on Systematic Reviews

In this section I consider both Systematic Reviews and meta-analysis since it is somewhat artificial to separate them. A review is the source of primary studies for meta-analysis; a review without some form of synthesis lacks the means of forming conclusions or answering research questions.

According to Kitchenham and Charters [28] the major disadvantages of systematic reviews are (i) that they require considerably more effort than traditional literature reviews and (ii) the increased power of a meta-analysis might also be problematic, since it is possible to detect small biases as well as true effects. For this reason careful analysis of sources of heterogeneity and sensitivity analysis are strongly recommended [24]. Use of informal and graphical methods e.g. Forest Plots should be used [6] to augment formal statistical tests for heterogeneity. Furthermore, the systematic review provides little protection against publication bias other than contacting each author of every located primary study.

There are some quite vociferous critics including Eysenck [20] and Shapiro [48], although many of their comments relate more to the problems of doing Systematic Reviews badly or in inappropriate situations rather than identifying fundamental flaws with the methodology.

So how do we judge the quality of a particular Systematic Review[4]? One approach is the DARE Standard [14] which rates Reviews according to five factors (see Table 4 by applying a checklist. Criteria 1-3 are mandatory and overall minimum score of 4 out of 5 is required.

**Table 4.** DARE Inclusion Criteria for Systematic Reviews [14]

| | |
|---|---|
| 1. | Were inclusion / exclusion criteria reported? |
| 2. | Was the search adequate? |
| 3. | Were the included studies synthesised? |
| 4. | Was the validity of the included studies assessed? |
| 5. | Are sufficient details about the individual included studies presented? |

However, these potential pitfalls for conducting Systematic Reviews do *not* mean:

> "that researchers should return to writing highly subjective narrative reviews. Many of the principles of systematic reviews remain: a study protocol should be written in advance, complete literature searches carried out, and studies selected and data extracted in a reproducible and objective fashion." [17]

As a small encouragement concerning the reliability of systematic reviews an experiment by MacDonell et al. [34] based upon two independent systematic reviews of the same research question found that despite some differences in approach the results were almost identical.

## 4 Systematic Reviews and Empirical Software Engineering

As has already been noted,, there has been extremely rapid take up of the idea of the Systematic Review in empirical software engineering. Between the

[4] Note that that this is not the same as quality instruments for appraising individual primary studies, since a good Review may detect low quality primary studies.

three tertiary reviews of Systematic Reviews in empirical software engineering [27,30,49] a total of 120 studies were identified, that satisfied their quality and inclusion criteria. Given that this has been in the space of five or six years this is a remarkable effort. However, we need to mature and it is not simply a matter of conducting as many Reviews as possible. In particular Cruzes and Dybå[13] sound a cautionary note concerning lack of research synthesis (Step D in terms of the Five Step Model presented in Figure 3) that is prevalent in more than half the studies they examined.

It seems that many studies are simply listing or cataloging primary studies with little attempt made to combine results. Cruzes and Dybå[13] state that"there is limited attention paid to research synthesis in software engineering" [13] and this matters because "the potential of empirical research will not be realized if individual primary studies are merely listed". In many ways this is unsurprising since a substantial number of Reviews are actually mapping studies, if not in name certainly in practice. The consequence is research questions are far more general in nature and not really constructed around notions of effect and intervention. Second, the primary studies are frequently diverse and predominantly observational studies. And this leads to particular challenges for meta-analysis and challenges that other disciplines are still struggling with.

It therefore seems timely to explore how other subjects are addressing these problems. Despite, a study by Budgen et al. looking at similarities in research methods across disciplines, that reported strong dissimilarities between clinical medicine and software engineering ([8]; modified and summarised in [28], pp5-6) it would be fair to say that at least the spirit behind evidence-based medicine remains highly influential. Yet in many ways the Campbell Collaboration is far more closely aligned with the problems we seek to address.

The other issue, that arises from many Reviews in empirical software engineering is the paucity of high quality primary studies. Of course identifying areas of lack is major service that Systematic Reviews can perform, however, it is important we do not devote all our effort to Systematic Reviews and meta-analysis or we will run out of primary studies to combine!

The less attractive side of Systematic Reviews is excessive introspection. This is exemplified by unforeseen uses of Tertiary Reviews such as the source of promotional material for particular research centres[5]. The danger here is complex and context-sensitive phenomena are reduced to simple quality checklists which are in turn taken out of the intended situation, trivialised and then turned into dramatic conclusions.

## 4.1 Challenges for the Future

- We clearly need better and more relevant primary studies. Systematic Reviews can help identify areas of lack and for that matter areas of over-supply.

[5] See for example "LERO Researchers come out top in SLR quality ratings" available from http://www.lero.ie/news/leroresearcherscomeouttopslrqualityratings – accessed June 21, 2011.

– Next, we need better reporting protocols as this will simplify the process of extracting information from primary studies and also make meta-analysis more feasible.
– We should start to be far more selective about SRs and ensure that they target interesting questions (particularly to stakeholders other than researchers) and by implication there will be fewer new Mapping and Tertiary Studies.
– It may be that determining which discipline or community is the closest to empirical software engineering is slightly misleading. We should borrow from wherever is appropriate, but there seems a good deal we can learn from C2 and Cochrane in respect of (i) the public review of protocols (ii) more sophisticated and context-sensitive quality appraisal of primary studies (iii) richer and fuller synthesis of located studies and (iv) careful attention to the sources of variance in our results. There is much diversity within any discipline and so we should determine which methods are most suitable.

However, I would like to end this tutorial by listing the ten Cochrane Collaboration guiding principles[6]. The empirical software engineering community could do a lot worse than take them to heart.

1. Collaboration – by internally and externally fostering good communications, open decision-making and teamwork
2. Building on the enthusiasm of individuals – by involving and supporting people of different skills and backgrounds
3. Avoiding duplication – by good management and co-ordination to maximise economy of effort
4. Minimising bias – through a variety of approaches such as scientific rigour, ensuring broad participation, and avoiding conflicts of interest
5. Keeping up to date – by a commitment to ensure that Cochrane Reviews are maintained through identification and incorporation of new evidence
6. Striving for relevance – by promoting the assessment of healthcare interventions using outcomes that matter to people making choices in health care
7. Promoting access – by wide dissemination of the outputs of the Collaboration, taking advantage of strategic alliances, and by promoting appropriate prices, content and media to meet the needs of users worldwide
8. Ensuring quality – by being open and responsive to criticism, applying advances in methodology, and developing systems for quality improvement
9. Continuity – by ensuring that responsibility for reviews, editorial processes and key functions is maintained and renewed
10. Enabling wide participation – in the work of the Collaboration by reducing barriers to contributing and by encouraging diversity

## Glossary

**Experiment:** a study in which variables are intentionally manipulated. Usually an experiment will also involve the random allocation of experimental units

[6] http://www.cochrane.org/about-us/our-principles – Accessed June 21, 2011.

(participants, etc.) to treatments and these are distinguished from quasi-experiments where the allocation is non-random (for example dictated by circumstances).

**Mapping study:** sometimes known as a scoping study, is a form of systematic review that explores the general levels and forms of research activity within a particular topic, and as such contrasts with the narrower systematic that addresses a specific research question.

**Meta-analysis:** is some statistical procedure for pooling and analysing results from more than one primary study.

**Observational study:** is an empirical study where the variables of interest (context, treatment, response) are observed but not manipulated (unlike a formal experiment).

**Primary study:** this is an individual empirical study that is closest to the phenomena of interest. As such it might take the form of an experiment or an observation study.

**Secondary study:** is a study that does not directly collect data but (re-) analyses results from one or more primary studies. Some researchers use this term to specifically refer to the re-analysis of primary data for some different purpose to that originally intended, however I, in common with the majority of researchers, simply intend it to mean any analysis. Therefore a systematic review is an example of secondary analysis.

**Systematic review:** is a secondary study where the analysis is conducted in a disciplined, documented, inclusive and unbiased manner as possible.

**Tertiary study:** is a meta-level review, that is a review of other reviews or secondary studies.

## Other Resources

– The ESRC National Centre for Research Methods have a valuable website at http://www.ncrm.ac.uk/
– The Sage Encyclopaedia of Social Science Research Methods is available to subscribers online at http://www.sage-ereference.com/socialscience/
– Both the Cochrane and Campbell Collaborations maintain an excellent set of resources and educational material online at www.cochrane.org/ and www.campbellcollaboration.org/ respectively.
– The journal *Information and Software Technology* has a dedicated section to empirical software engineering Reviews. The journal website also hosts a copy of the Kitchenham and Charters Guidelines [28].

There are increasing numbers of useful software systems to support different aspects of systematic reviews:

– RevMan is freely available from http://ims.cochrane.org/revman/ and although primarily intended for Cochrane reviews can be run in non-Cochrane mode as well.
– The website of the Centre for Evidence-Based Medicine at Oxford University maintains a useful page of tools and online resources for conducting Reviews although with a clinical bias at http://www.cebm.net/index.aspx?o=1023.

## References

1. Altman, D.: Poor-quality medical research – what can journals do? JAMA 287(21), 2765–2767 (2002)
2. Aronson, J.: A pragmatic view of thematic analysis. The Qualitative Report 2(1) (1994)
3. Avison, D., Lau, F., Myers, M., Nielsen, P.A.: Action research. CACM 42(1), 94–97 (1999)
4. Basili, V.R., Shull, F., Lanubile, F.: Building knowledge through families of experiments. IEEE Transactions on Software Engineering 25(4), 456–473 (1999)
5. Beynon-Davies, P.: Ethnography and information systems development: Ethnography of, for and within is development. Information & Software Technology 39(8), 531–540 (1997)
6. Blettner, M., Sauerbrei, W., Schlehofer, B., Scheuchenpflug, T., Friedenreich, C.: Traditional reviews, meta-analyses and pooled analyses in epidemiology. Intl. J. of Epidemiology 28(1), 1–9 (1999)
7. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qualitative Research in Psychology 3(2), 77–101 (2006)
8. Budgen, D., Charters, S., Turner, M., Brereton, P., Kitchenham, B., Linkman, S.: Investigating the applicability of the evidence-based paradigm to software engineering. In: WISER 2006, pp. 7–13. ACM, Shanghai (2006)
9. Chalmers, I.: The Cochrane Collaboration: preparing, maintaining, and disseminating systematic reviews of the effects of health care. Annals N.Y. Acad. Sci. 703, 153–163 (1995)
10. Cochrane Collaboration: The Cochrane Collaboration. Tech. rep., `http://www.cochrane.org`
11. Cooper, H.: Scientific guidelines for conducting integrative research reviews. Review of Educational Research 52(summer), 291–302 (1982)
12. Cooper, H.: Research synthesis and meta-analysis: A step-by-step approach, 4th edn. Sage, Thousand Oaks (2010)
13. Cruzes, D., Dybå, T.: Research synthesis in software engineering: A tertiary study. Information and Software Technology 53(5), 440–455 (2011)
14. DARE: About DARE. Tech. rep., Centre for Reviews and Dissemination, York University, UK (2011), `http://www.crd.york.ac.uk/cms2web/AboutDare.asp`
15. Dixon-Woods, M., Agarwal, S., Jones, D., Young, B., Sutton, A.: Synthesising qualitative and quantitative evidence: a review of possible methods. Journal of Health Services Research & Policy 10(1), 45–55 (2005)
16. Dybå, T., Kitchenham, B., Jørgensen, M.: Evidence-based software engineering for practitioners. IEEE Software 22(1), 58–65 (2005)
17. Egger, M., Schneider, M., Davey-Smith, G.: Meta-analysis spurious precision? Meta-analysis of observational studies. BMJ 316, 140 (1998)
18. Ellis, P.: The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results. Cambridge University Press (2010)
19. Evans, D.: Hierarchy of evidence: a framework for ranking evidence evaluating healthcare interventions. Journal of Clinical Nursing 12, 77–84 (2003)
20. Eysenck, H.: Systematic reviews: Meta-analysis and its problems. BMJ 309, 789 (1994)
21. Glass, G., McGaw, B., Smith, M.: Meta-analysis in social research. Sage Publications, Beverly Hills (1981)

22. Hayes, W.: Research synthesis in software engineering: a case for meta-analysis. In: 6th IEEE International Softw. Metrics Symp., pp. 143–151. IEEE Computer Society, Boca Raton (1999)
23. Higgins, J., Green, S. (eds.): Cochrane Handbook for Systematic Reviews of Interventions, Version 5.1.0. The Cochrane Collaboration (2011)
24. Higgins, J., Thompson, S., Deeks, J., Altman, D.: Measuring inconsistency in meta-analysis. British Medical Journal 327, 557–560 (2003)
25. Jüni, P., Witschi, A., Bloch, R., Egger, M.: The hazards of scoring the quality of clinical trials for meta-analysis. JAMA 282(11), 1054–1060 (1999)
26. Katrak, P., Bialocerkowski, A.E., Massy-Westropp, N., Kumar, S., Grimmer, K.A.: A systematic review of the content of critical appraisal tools. BMC Medical Research Methodology (Electronic Resource) 4(1), 22 (2004)
27. Kitchenham, B., Brereton, P., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic literature reviews in software engineering a systematic literature review. Information & Software Technology 51(1), 7–15 (2009)
28. Kitchenham, B., Charters, S.: Guidelines for performing systematic literature reviews in software engineering, version 2.3. Tech. Rep. EBSE Technical Report EBSE-2007-01. Keele University, UK (2007)
29. Kitchenham, B., Mendes, E., Travassos, G.: A systematic review of cross- vs. within-company cost estimation studies. In: 10th Intl. Conf. Empirical Assessment in Soft. Eng, EASE (2006)
30. Kitchenham, B., Pretorius, R., Budgen, D., Brereton, P., Turner, M., Niazi, M., Linkman, S.: Systematic literature reviews in software engineering a tertiary study. Information and Software Technology 52(8), 792–805 (2010)
31. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Transactions on Software Engineering 28(8), 721–734 (2002)
32. Kitchenham, B., Dybå, T., Jørgensen, M.: Evidence-based software engineering. In: 27th IEEE Intl. Softw. Eng. Conf. (ICSE 2004), IEEE Computer Society, Edinburgh (2004)
33. Luborsky, L., Diguer, L., Seligman, D., Rosenthal, R., Krause, E., Johnson, S., Halperin, G., Bishop, M., Berman, J., Schweizer, E.: The researcher's own therapy allegiances: A wild card in comparisons of treatment efficacy. Clinical Psychology: Science and Practice 6(1), 95–106 (1999)
34. MacDonell, S., Shepperd, M., Kitchenham, B., Mendes, E.: How reliable are systematic reviews in empirical software engineering? IEEE Transactions on Software Engineering 36(5), 676–687 (2010)
35. MacDonell, S., Shepperd, M.: Comparing local and global software effort estimation models reflections on a systematic review. In: 1st Intl. Symp. on Empirical Softw. Eng. & Measurement, Madrid (2007)
36. Michie, D., Spiegelhalter, D.J., Taylor, C.C.: Machine learning, neural and statistical classification. Ellis Horwood Series in Artificial Intelligence. Ellis Horwood, Chichester (1994)
37. Miller, J.: Can results from software engineering experiments be safely combined? In: Briand, L. (ed.) IEEE 6th Intl. Metrics Symp. IEEE Computer Society, Boca Raton (1999)
38. Naur, P., Randall, B. (eds.): Software Engineering: Report on a Conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels (1968)
39. Needleman, I.: A guide to systematic reviews. Journal of Clinical Periodontology 29(suppl. s3), 6–9 (2002)

40. Oswald, M., Grosjean, S.: Confirmation bias. In: Pohl, R. (ed.) Cognitive Illusions: A Handbook on Fallacies and Biases in Thinking, Judgement and Memory, pp. 79–96. Psychology Press, Hove (2004)
41. Petticrew, M.: Systematic reviews from astronomy to zoology: Myths and misconceptions. British Medical Journal 322(7278), 98 (2001)
42. Pickard, L., Kitchenham, B., Jones, P.: Combining empirical results in software engineering. Information & Software Technology 40(14), 811–821 (1998)
43. Popay, J., Roberts, H., Sowden, A., Petticrew, M., Arai, L., Rodgers, M., Britten, N.: Guidance on the conduct of narrative synthesis in systematic reviews: A product from the ESRC methods programme. Tech. rep., Institute for Health Research, University of Lancaster (version 1, 2006)
44. Rosenthal, R.: The file drawer problem and tolerance for null results. Psychological Bulletin 86(3), 638–641 (1979)
45. Sackett, D., Straus, S., Richardson, W., Rosenberg, W., Haynes, R.B.: Evidence-Based Medicine: How to Practice and Teach EBM, 2nd edn. Churchill Livingstone, Edinburgh (2000)
46. Seaman, C.: Qualitative methods in empirical studies of software engineering. IEEE Transactions on Software Engineering 25(4), 557–572 (1999)
47. Shadish, W., Cook, T., Campbell, D.: Experimental and quasi-experimental designs for generalized causal inference. Houghton Mifflin, Boston (2002)
48. Shapiro, S.: Meta analysis/shmeta analysis. American J. of Epidemiology 140, 771–778 (1994)
49. da Silva, F., Santos, A., Soares, S., Franaa, A., Monteiro, C., Maciel, F.: Six years of systematic literature reviews in software engineering: An updated tertiary study. Information and Software Technology 53(9), 899–913 (2011)
50. Sommerville, I.: Software Engineering, 5th edn. Addison-Wesley, Wokingham (1996)
51. Spencer, L., Ritchie, J., Lewis, J., Dillon, L.: Quality in qualitative evaluation: A framework for assessing research evidence. Tech. rep., The Cabinet Office (2003)
52. Spoor, P., Airey, M., Bennett, C., Greensill, J., Williams, R.: Use of the capture-recapture technique to evaluate the completeness of systematic literature searches. BMJ 313(7053), 342–343 (1996)
53. Stroup, D., Berlin, J., Morton, S., Olkin, I., Williamson, G., Rennie, D., Moher, D., Becker, B., Sipe, T., Thacker, S.: Meta-analysis of observational studies in epidemiology: A proposal for reporting. JAMA 283(15), 2008–2012 (2000)
54. Sutton, A., Higgins, J.: Recent developments in meta-analysis. Statistics in Medicine 27, 625–650 (2008)
55. Viller, S., Sommerville, I.: Ethnographically informed analysis for software engineers. International J. of Human-Computer Studies 53(1), 169–196 (2000)
56. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M., Regnell, B., Wessln, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic, Norwell (2000)
57. Yancey, J.: Ten rules for reading clinical research reports. American J. of Surgery 159, 533–539 (1990)
58. Yin, R.: Case Study Research - Design and Methods, 3rd edn. SAGE Publications (2003)

# Predicting Bugs in Large Industrial Software Systems

Thomas J. Ostrand and Elaine J. Weyuker

AT&T Labs - Research, Florham Park, NJ
180 Park Avenue
Florham Park, NJ 07932
{ostrand,weyuker}@research.att.com

**Abstract.** This chapter is a survey of close to ten years of software fault prediction research performed by our group. We describe our initial motivation, the variables used to make predictions, provide a description of our standard model based on Negative Binomial Regression, and summarize the results of using this model to make predictions for nine large industrial software systems. The systems range in size from hundreds of thousands to millions of lines of code. All have been in the field for multiple years and many releases, and continue to be maintained and enhanced, usually at 3 month intervals.

Effectiveness of the fault predictions is assessed using two different metrics. We compare the effectiveness of the standard model to augmented models that include variables related to developer counts, to inter-file calling structure, and to information about specific developers who modified the code.

We also evaluate alternate prediction models based on different training algorithms, including Recursive Partitioning, Bayesian Additive Regression Trees, and Random Forests.

**Keywords:** software fault prediction, defect prediction, negative binomial model, fault-percentile average, buggy file ratio, calling structure, prediction tool.

## 1 Introduction

There has been a great deal of research describing different ways of identifying which parts of a software system are most likely to contain undiscovered defects. This is a potentially very important question because if managers knew where defects were likely to exist, then they could concentrate resources on those areas, and hopefully be able to identify them more quickly, using fewer resources, and ultimately wind up with more reliable software than would otherwise be produced.

In particular, test or quality assurance managers could decide which code units were likely to be most problematic and allocate software testing personnel and their time accordingly. Development managers could use the results of our research to help decide when it was particularly likely to be worthwhile allocating scarce resources to perform such expensive activities as design or code reviews, or to decide to redesign and re-implement repeatedly problematic parts of the system.

Our research has focused on making predictions for large long-lived industrial software systems. The predictions take the form of a list of those files predicted to have the largest number of defects in the next release, in descending order. In this chapter, we describe our approach to software fault prediction, which variables and models we have examined in order to accurately identify the most fault-prone files of a software system, and what we have observed when applying our models to different types of systems, with different levels of maturity.

We use the words *defect, fault* and *bug* interchangeably in this chapter, to mean some condition of the code that causes the system to behave differently from its specifications.

Each of the systems we studied uses a *software configuration management system* that combines both change management and version control functions in a single integrated system. The configuration management system tracks successive releases of the system under construction and all changes made to the code for any reason. Changes are initiated by writing a *modification request* or MR, which includes such information as the identity of the individual who is making the request, the individual who actually makes the change (if one is ultimately made), the job function of the requester, the stage of development when the change is being requested, any files changed as a result of the request, the dates of the original request and of the changes, the assigned severity of the MR, plus a number of other pieces of data. The MR writer has the option of providing a written natural language description of the reason for requesting the change, or suggestions for implementing the change, or any other information that might be useful for the person who eventually makes the change.

Despite the wealth of information contained in each modification request, one thing that is *not* generally specified in a uniform way is whether the change is being requested to add functionality, to modify some existing functionality as part of a planned or unplanned enhancement, or to fix a defect. This lack forced us to come up with an alternate way to determine whether a change was made because of a defect or for some other reason.

After experimenting with various ways of identifying defect MRs, we settled on using the phase of software development when the MR was originally created. Any MR that originates during system testing or customer acceptance testing is considered to define a defect.

The MR information is recorded in the database that is the heart of the configuration management system, which we mine to extract data needed for our prediction models. Although our goal is to identify the most fault-prone files, changes of all types, including bug fixes, planned or unplanned enhancements, and modifications of functionality, all play a role in helping to predict whether there will be faults in the next release of the software, and how many to expect.

Of the nine software systems for which we have made predictions, all but one had a regular quarterly release schedule. The configuration management system records data at the file level, and therefore that is the level at which our predictions are made. While all of the nine systems use the same (commercially-available) configuration management system or a slight variant of it, six of the

**Table 1.** System Information

| **System** | **Years in the field** | **Releases** | **Latest release** Files | **Latest release** LOC | **Average over releases** Faults | **Average over releases** Percent faulty files |
|---|---|---|---|---|---|---|
| Inventory1 | 4 | 17 | 1950 | 538,000 | 342 | 12.1% |
| Provisioning1 | 2 | 9 | 2241 | 438,000 | 34 | 1.3% |
| Voice Response | 2.25 | 9 | 1926 | 329,000 | 165 | 10.1% |
| Maintenance A | 9 | 35 | 668 | 442,000 | 44 | 4.8% |
| Maintenance B | 9 | 35 | 1413 | 384,000 | 44 | 2.4% |
| Maintenance C | 7 | 27 | 584 | 329,000 | 50 | 5.7% |
| Provisioning2 | 4 | 18 | 3920 | 1,520,000 | 388 | 5.1% |
| Utility | 4 | 18 | 802 | 281,000 | 90 | 5.9% |
| Inventory2 | 4 | 18 | 6693 | 2,116,000 | 358 | 2.9% |

systems we studied were developed and maintained by people at one company, and three were developed and maintained by people at a second company. All of the systems run continuously, all have been in the field for multiple years, and they range in size from hundreds of thousands up to millions of lines of code.

Information about the age, the size, and the detected faults for each system is summarized in Table 1. The two center columns of the table give the size of each system's latest release. The two rightmost columns show the number of detected faults and percent of faulty files per release, averaged over all the releases of the system. Often early releases have substantially more faults than later releases. The last column shows that faults are always concentrated in a fairly small percentage of the system's files.

The remainder of this chapter is organized as follows. Section 2 describes basic information about the negative binomial regression model we use to make predictions, and the variables included in the model. We describe two different ways of evaluating the effectiveness of predictions in Section 3. Section 4 looks at the impact of augmenting the standard model with additional variables.

In particular, in Section 4.1 we consider several different ways of adding developer count information to the model. Section 4.2 continues our exploration of the impact of augmenting the standard model, this time with information about which individuals have modified a file during the previous release. In Section 4.3 we study the impact of inter-file communication on our ability to accurately predict where defects will be in the next release. This is done by considering various ways of augmenting the standard model with calling structure information. In Section 5 we examine the use of different prediction models and compare the prediction accuracy of these models on Maintenance Systems A, B, and C. In particular, in Section 5.1 we study Recursive Partitioning, in Section 5.2 we examine the use of Random Forests, and in Section 5.3 we consider Bayesian Additive Regression Trees (BART). Section 6 describes our conclusions. Section 7 provides pointers to some of our research papers where more details can be found.

## 2 Our Approach to Defect Prediction Modeling

We have used a negative binomial regression model [4] to make predictions for all the systems we considered. Negative binomial regression has much in common with standard linear regression, but they differ in two fundamental ways. The first is that while linear regression models the expected value of the fault counts themselves, negative binomial regression uses a linear function of the predictor variables to model the *logarithm* of the expected fault count values. The second difference is that linear regression assumes that the counts come from a normal distribution while negative binomial regression requires that fault counts be nonnegative integers and so assumes that each count comes from a negative binomial distribution.

More specifically, if $y_i$ is the observed fault count for a given file and release pair, and $x_i$ is the corresponding vector of predictor variables, then negative binomial regression models $y_i$ as a negative binomial distribution with mean $\lambda_i = e^{\beta' x_i}$ and variance $\lambda_i + k{\lambda_i}^2$ for unknown $k \geq 0$. $k{\lambda_i}^2$ is included to accommodate possible over-dispersion that we often see for fault counts relative to that implied by Poisson regression.

The negative binomial model that makes fault predictions for a release of a system is based on the fault and change history of the system, as well as characteristics such as file type and file size, as far back as data can be obtained. For each Release N, a new model is created to make its fault predictions, using training data from Releases 1 through N-1. Once the model is created, values of input variables from the previous two releases are used to make predictions for Release N.

We developed a *standard model* for predictions on the basis of initial studies of faults and system characteristics of systems Inventory1, Provisioning1, and Voice Response, listed in Table 1. Our subsequent work with six additional systems validated the effectiveness of the standard model.

The input variables included in the standard model are

- lines of code (LOC)
- faults in Release N-1
- changes in Releases N-1 and N-2
- files status (new, changed, unchanged)
- file age in terms of number of previous releases (0, 1, 2-4, 5 or greater)
- file type (programming language identified by the file extension, e.g., C, C++, java, sql)

Because the standard model requires change data from two previous releases to predict the current release, it is applied starting at release 3 of each of the systems.

All the variables in the standard model showed some correlation with fault occurrence in the first three studied systems. Some variables, like programming language and file age, have a fixed set of values over all releases of a given system, while others, like size, faults and changes, can vary over the lifetime of the system.

We have often found that the best single indicator of faults was size of a file at the beginning of the release. This was typically measured as the logarithm of the number of lines of code, log(LOC).

In many cases we observed that newer files tend to exhibit more problems than ones that have been in the system for many releases, and we therefore categorize the number of releases a file has been in the system. We have found that using four categories worked well: files that were introduced in the current release, files that have been in just one prior release, files in two to four prior releases, and those that have been in five or more earlier releases.

We have also observed that if a file has been changed in recent releases or had defects in recent releases, it often has problems in the current release. Therefore, the standard model includes counts of defects in the most recent previous release and changes in each of the two previous releases. To reduce skewness of the predictor variables, for each of these counts we use the square root rather than the count itself.

Because we have observed that different programming language have different fault rates, and so may be an important predictor, we use a series of dummy variables for all but one language.

The last thing we account for in the standard model is the maturity of the system, measured in terms of the system release number. This is done because the number of defects in the system often varies substantially from release to release, even after controlling for the ages of files in the system. Again we use a series of dummy variables.

We have evaluated several different modifications of our standard model and found either small improvements over the standard model results or no improvement at all. Variations that involve the use of additional variables beyond those in the standard model are described in Section 4 of this chapter.

We have also examined the effectiveness of using different prediction models and found that none of those considered performed better than our standard negative binomial regression model. These alternate models are discussed in Section 5.

For each file of a release, our prediction models compute an estimated number of faults that the file is predicted to have in the next release. We have built an automated tool which extracts the necessary data from the software configuration management system's database, builds the prediction model, and does the calculation of the number of faults predicted for each file in the next release. The tool then outputs a sorted list of the files in decreasing order of the predicted number of faults. Our goal is to help the development and test teams prioritize their efforts and resource usage, based on these predictions.

We and other researchers have repeatedly observed that the distribution of faults among files of large systems is highly non-uniform, with a relatively small percentage of files typically containing a large percentage of the defects. This means that if our model correctly identifies the most fault-prone files, collectively they will account for a very large percentage of the defects.

Our prediction model was designed to be applied just prior to the beginning of system testing for the most recent version of a system. At that point, the code should be stabilized, with unit and integration testing complete. The systems we have studied typically have most defects identified during pre-release system testing with few defects identified in the field. Our goal is to catch *all* defects during pre-release testing so that the customer never sees a problem. Our tool should help in this process.

## 3 Evaluating the Prediction Results

In this section we describe two different ways we have used to evaluate how successful our predictions have been. In all of our research, we have considered the percentage of bugs contained in the $X\%$ of the files predicted to contain the largest numbers of defects. Often we set $X$ to 20 and compute what percentage of the actual bugs turned out to be contained in the predicted worst 20% of the files. That is the value shown in the next to the last column of Table 2. In each case the number was averaged over all releases for which we made predictions for a given system. The table shows that for the nine systems studied, the predictions were generally very successful with the top 20% of the files predicted to contain the largest numbers of defects containing from 75% to 93% of the defects actually detected in the next release.

Of course 20% is an arbitrary value, and it is possible that the results can vary substantially using a slightly larger or smaller value of $X$, and so we proposed an alternate measure called the *fault-percentile average*. Essentially, this measure summarizes the percentage of actual defects over all possible values of the cutoff percent.

Let $K$ be the number of files in a release. The fault-percentile average is computed for a given set of predictions by sorting the files based on the predicted fault count. The ordering is from smallest to largest number of predicted faults: $f_1, f_2, ..., f_K$. The $k^{th}$ file is assigned the percentile value $100k/K$. If $n_k$ is the actual number of faults in file $f_k$, then the fault-percentile average for the given prediction ordering is the mean percentile value, weighted by $n_k$.

As predictions improve, files with high numbers of actual faults move closer to the high end of the listing, and their contributions to the percentile average increase. The overall fault-percentile average is equivalent to the average, over all values of $m$, of the percent of actual faults contained in the $m$ files with the highest predicted numbers of faults. The fault-percentile averages for six of the systems we studied are shown in the last column of Table 1.

Both of these measures of prediction success have their uses, and provide users of the prediction model with confidence in the model's accuracy. We have observed that practitioners tend to find the use of a concrete value of $X$ more useful, while the fault-percentile average is less sensitive to the choice of the cutoff value and therefore is particularly well-suited for comparing alternative models.

**Table 2.** Fault-percentile average and actual faults in top 20% of predicted files for nine systems

| System | Years in the field | Releases | LOC | Actual faults found in top 20% of predicted files | Fault-percentile average |
|---|---|---|---|---|---|
| Inventory1 | 4 | 17 | 538,000 | 83% | not available |
| Provisioning1 | 2 | 9 | 438,000 | 83% | not available |
| Voice Response | 2.25 | 9 | 329,000 | 75% | not available |
| Maintenance A | 9 | 35 | 442,000 | 81% | 88% |
| Maintenance B | 9 | 35 | 384,000 | 93% | 93% |
| Maintenance C | 7 | 27 | 329,000 | 76% | 88% |
| Provisioning2 | 4 | 18 | 1,520,000 | 91% | 93% |
| Utility | 4 | 18 | 281,000 | 87% | 92% |
| Inventory2 | 4 | 18 | 2,116,000 | 93% | 95% |

## 4 Alternate Predictor Variables

In this section, we discuss ways that we have tried to modify our standard model in the hope of improving predictions.

### 4.1 Adding Developer Count Information

Many software engineers believe that code is most reliable when it has been written and maintained by only one programmer. The more general belief is that the more programmers who are involved with the code, the more likely it is that errors will be made and faults introduced. This is the *too many cooks spoil the broth* hypothesis. It certainly seems reasonable to expect more problems if different people write separate parts of a single file, or if one person writes the original code and others later make maintenance updates or fixes. There is an obvious potential for miscommunication and misunderstanding of the algorithm being implemented, as well as of the coding techniques being used. On the other hand, if two people are collaborating on a single piece of code, they might be able to check each other's work for mistakes, resulting in fewer faults.

To evaluate the influence that multiple developers might have on faults in a file, we created models using three additional variables relating to the number of separate developers who interacted with the file in previous releases. We refer to these variables as the *developer count variables.*

– *previous developers*: the number of developers who modified the file during the prior release.
– *new developers*: The number of developers who modified the file during the prior release, but did not work on the file in any earlier release.
– *cumulative developers*: The number of *distinct* developers who modified the file during all releases through the prior release.

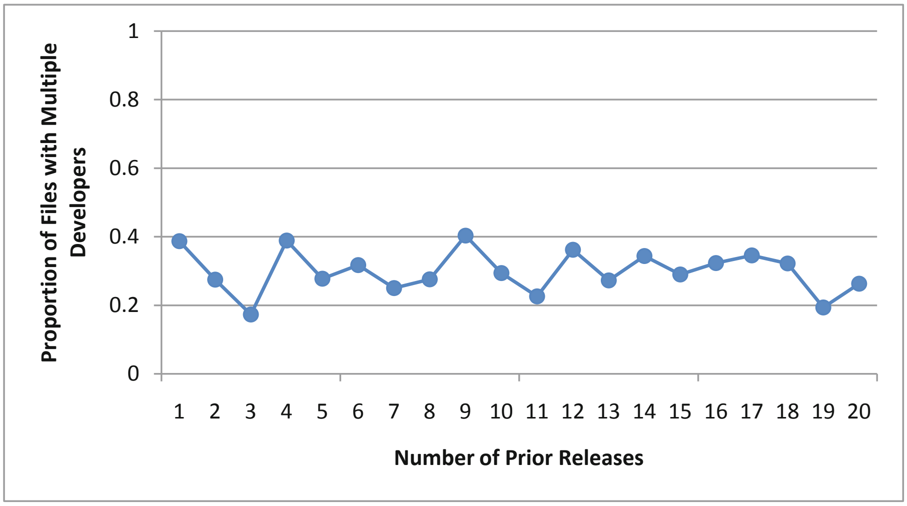


**Fig. 1.** Proportion of Changed Files with Multiple Developers, Maintenance A

The values of these variables are obtained from the MR history of the project. The initial creation of the file is not considered a change, and does not contribute to any of the three developer variables.

The first two variables are zero for any file that was not changed in the previous release. The new developers variable is zero even for files that were changed in the previous release if all the changers had also made changes in earlier releases. The cumulative developers variable can be non-zero for files that are unchanged in the previous release because it measures how many distinct people have changed the file since its original creation. The value of this variable is monotonically non-decreasing from release to release.

We investigated the three developer count variables for Maintenance systems A, B, and C. Maintenance A revealed some surprising facts about the distribution of these counts. Figures 1, 2 and 3 give some information about the variables for the first 21 releases of Maintenance A.

How common is it for a file to be changed by more than one developer? Figure 1 shows that 20 to 40% of the changed files of System A have been edited by multiple developers, and the proportion doesn't vary much as files age. Even more files have at least one new developer at each release, and aging of files does not diminish the new developers. Figure 2 shows that the proportion of changed files with at least one new developer tends to stay above 40% independent of the age of the file. The ratio is 1 for Release 1, because every developer is new for every file.

Over a file's lifetime, it may be changed not at all after its initial creation, or changed many times, and by varying numbers of developers. Figure 3 shows the distribution of the number of developers that changed files in System A for the first 20 releases of their lifetime. Files that have been in the system for fewer than

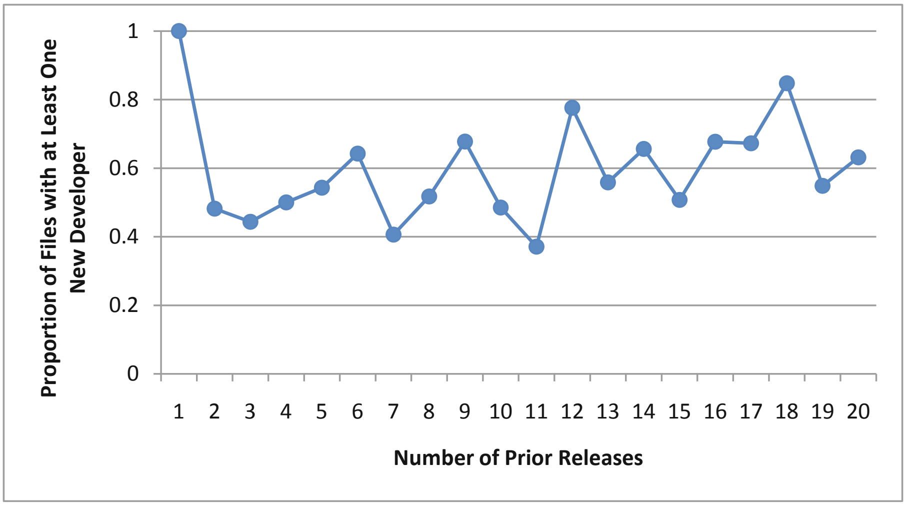


**Fig. 2.** Proportion of Changed Files with New Developers, Maintenance A

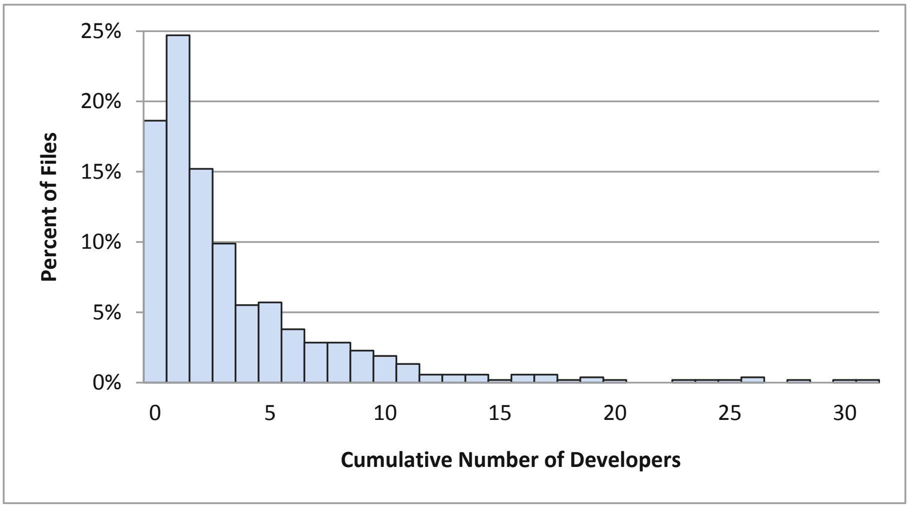


**Fig. 3.** Cumulative Number of Developers after 20 Releases, Maintenance A

20 releases are not included in this figure. Notice that 18% of the files were never changed after their initial creation, 25% were under control of a single developer throughout, about half the files were changed by three or fewer developers, and 60% were changed by five or fewer developers.

Although the proportion drops quickly after five developers, there do exist files that have been changed by 10 or more, up to 31 different individuals, during the

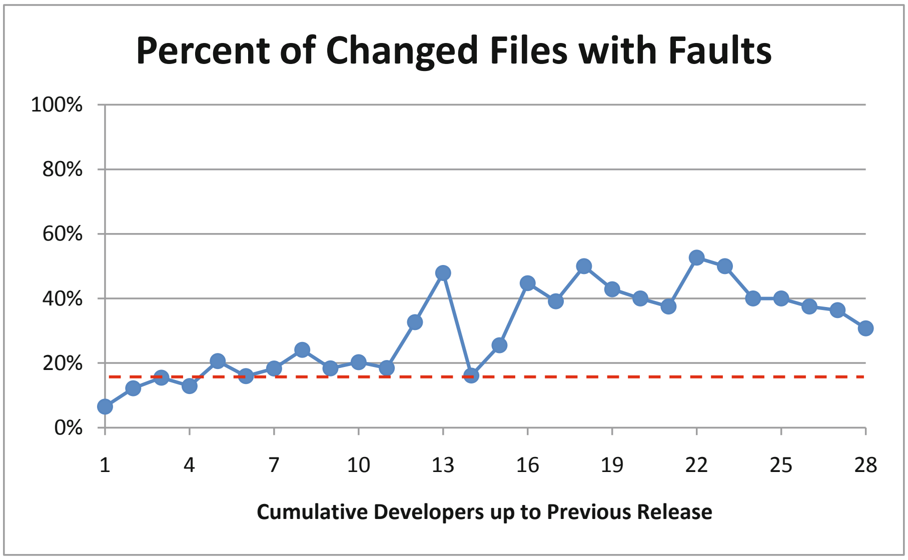


**Fig. 4.** Faulty Files per Cumulative Developers, Maintenance A

first 20 releases. Software engineering folklore would target those files as being highly fault-prone after so many different people have edited them.

Figure 4 shows there is indeed an increase in fault-proneness as the total number of developers increases over a file's lifetime. This figure shows the percent of changed files that are faulty, as a function of the cumulative number of developers that have touched the file over its lifetime. Although there are a few files in the system that have been touched by as many as 50 developers, above 28 the data is too sparse to be meaningful. The average percent of changed files that are faulty, regardless of the number of developers, is 17.7%, shown by the dashed red line across the figure.

Similar charts in Figures 5 and 6 show that the fault-proneness also tends to be higher with higher numbers of developers and new developers, both counted only in the previous release. However, these graphs are not as convincing as Figure 4, mainly because almost all the files are concentrated in the first two or three points in each graph. For example, of the 3110 files that were changed over the course of 35 releases, 2636 had either no or one new developer in the previous release, 404 had either two or three new developers, and only 70 had more than three new developers at the previous release.

The relations that appear in these graphs are confirmed by the results of prediction models that use the three developer count metrics. When the two metrics that consider only the previous release are added to the standard model, the prediction results barely change. Augmenting the standard model with cumulative developers yielded a small, but statistically significant, increase in the accuracy of the predictions for systems Maintenance A and B, but no increase for Maintenance C. Averaged over releases 3-35, the percent of faults captured

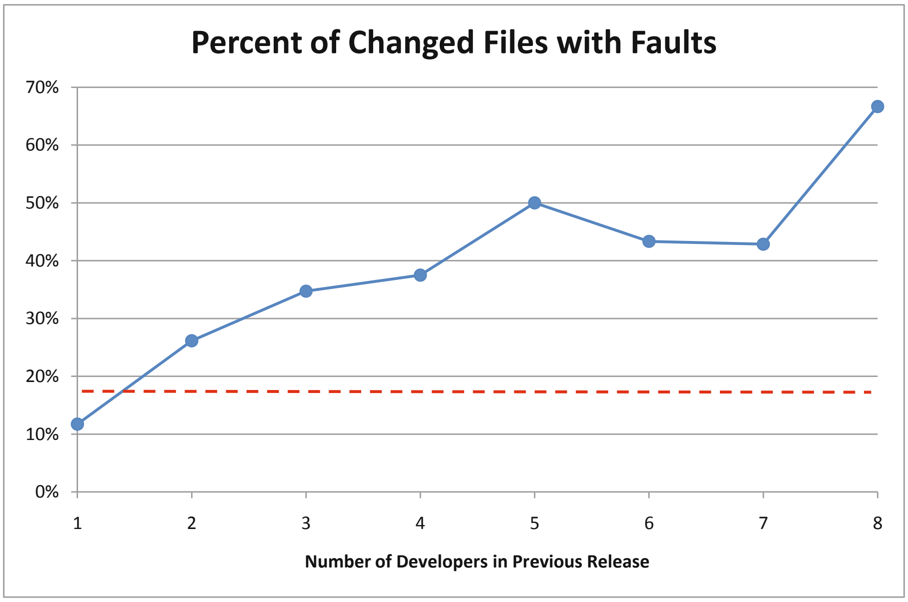


**Fig. 5.** Faulty Files per Developers in Previous Release, Maintenance A

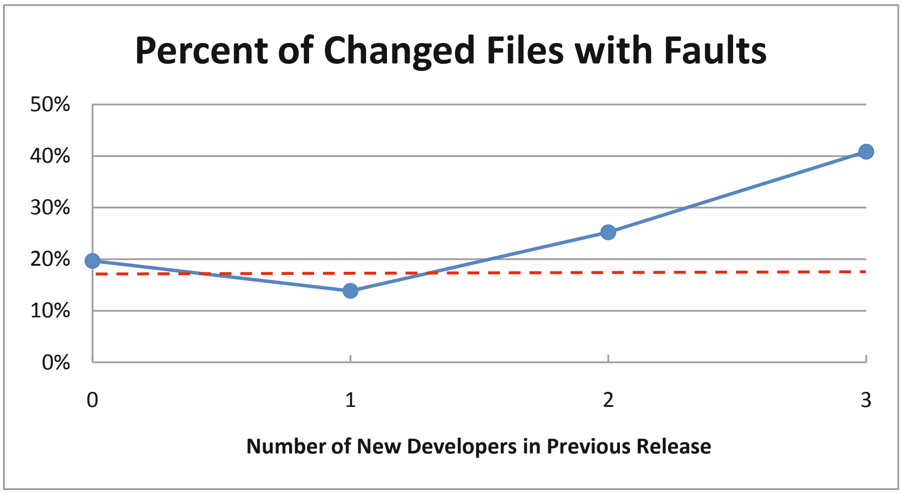


**Fig. 6.** Faulty Files per New Developers in Previous Release, Maintenance A

in the top 20% of the files predicted by the cumulative developers augmented model exceeds the standard model results by 0.2% for System A and by 1.0% for System B.

These models were also applied to the Provisioning2 system, and here the advantage of the cumulative developer augmented model is stronger. Figure 7

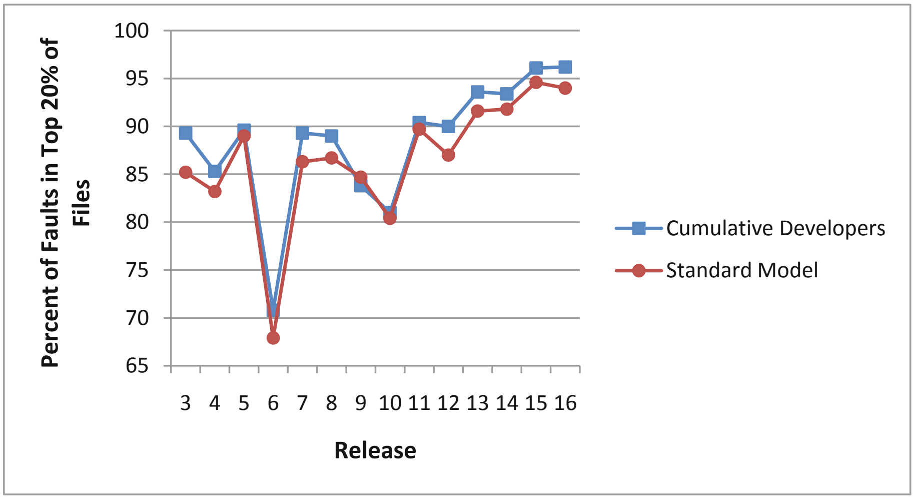


**Fig. 7.** Prediction Accuracy with and without Cumulative Developers, Provisioning2

shows the release-by-release difference between the two models. The augmented model is ahead of the standard model for all releases except Release 9, where it falls only slightly behind. The lower value on the y-axis is set at 65 to facilitate visualizing the difference between the values for the two models. The average accuracy over releases 3-16 is 86.6% for the standard model, and 88.4% for the cumulative developer augmented model.

## 4.2 Adding Individual Developer Information

In the previous subsection, we considered the impact of augmenting the standard model with three different ways of counting the number of different developers who modified a file in past releases. In this section, we investigate whether knowing which particular developer modified a file can be helpful in predicting whether the file is likely to be faulty or not in the next release.

The common intuition is that knowing who changed a file should be a very potent predictor of fault-proneness since we believe we can identify who are the "good" programmers, and who are not. Specifically we ask whether files changed by particular developers are systematically either more or less likely to be faulty than the average. If this is true, then we might be able to augment the standard model to use this information to improve predictions.

There are several issues that make the use of individual developer data much more difficult than using other code or history characteristics that contribute to the standard model. In particular, while every file has a size, an age, and a programming language, developers come and go from a project, and therefore there is frequently insufficient data about a developer to make any meaningful use of it. Furthermore, often developers change few files, even over a substantial number of releases.

For example, during the first 17 releases of system Provisioning2, approximately 70% of the 177 developers who worked on this system during that time period modified fewer than 50 files, even when we included in this count, every time Developer $X$ changed File $Y$ during multiple releases. That is, if Developer $X$ changed File $Y$ six different times, that added six to the count associated with Developer $X$.

We typically focus on files that were changed in the previous release because, although in most cases 85% - 97% of all files are either new to the release or remain unchanged from the previous release, the small fraction of changed files accounted for 67% of the files that had defects for Releases 2 through 18.

We define the notion of the *buggy file ratio* as a way of determining the possible relationship between defects in files and the developers who changed the files in the previous release. The buggy file ratio of developer $D$ at Release $R-1$ is the fraction of files changed by developer D at Release R-1 that subsequently have one or more faults at Release R.

Of the 55 developers who did change at least 50 files during these 17 releases, we do see some variation. Figure 8 shows the buggy file ratio for nine of these developers. In addition to showing the ratio, the graph also shows the number of files that a given developer changed during the previous release. Each point is labeled with a number indicating the number of files a particular developer changed during the previous release. To clarify, Developer 14 changed 35 files during Release 1. This can be determined by looking at the first point in the upper left panel of Figure 8. The vertical height of the point reflects the buggy file ratio, indicating the proportion of files changed by Developer 14 that contained one or more defect in Release 2. For this case, the ratio is 0.31 because 11 out of the 35 files changed by Developer 14 contained bugs.

The leftmost graph in the second row presents information for Developer 18 The two files changed during Release 1 were both fault-free in Release 2. In Release 10, 20 of the 50 files changed by this developer in Release 9 had bugs.

In each of the graphs we provide a solid line that shows the mean proportion of buggy files across developers for each release. This is useful for comparison purposes. If a file is changed by multiple developers during a given release, it is included in the plots for each developer who modified it.

We selected the nine developers shown in Figure 8 to provide some examples of developers who are always, or almost always, above average, and also some that are always, or almost always, below average. But most developers, we observed, are like the center row of the figure and sometimes do very well leading to few faults in the next release, and other times do very poorly, meaning there are many defects in the files they changed in the next release. But even being regularly above or below average is not really good enough to be of real use for predictions. For the buggy file ratio to be a useful fault predictor, the ideal situation would be for "good" developers to be at or near 0, and for "poor" developers to have buggy file ratios at or near 1.

We analyzed the buggy file ratio for three different systems (Provisioning2, Utility, and Inventory2), and found weak evidence of some persistence over time

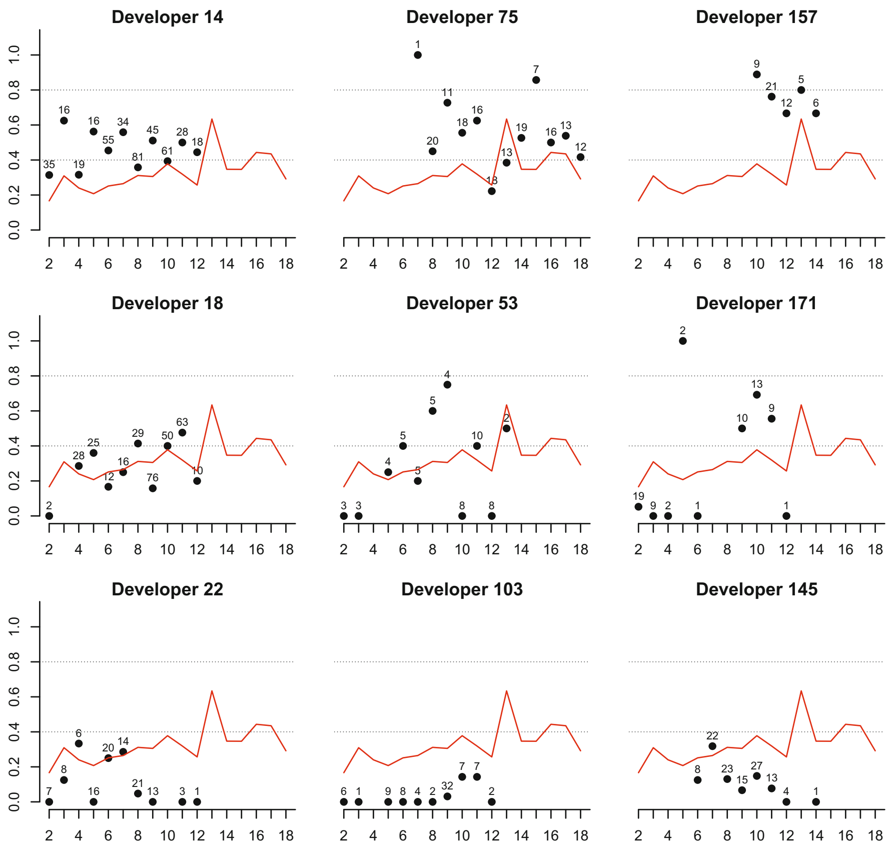


**Fig. 8.** Buggy file ratios for selected developers in Provisioning2

of fault-proneness associated with individuals. However, incorporating this information into the prediction model yielded no statistically significant improvement to the prediction accuracy when the 20% metric was used for assessment. In fact, adding this factor sometimes even decreased the accuracy of the predictions. When accuracy was assesed using the fault-percentile average metric, there was a very small, but statistically significant, improvement to the prediction accuracy for all three systems.

## 4.3 Adding Calling Structure Information

Modern programming techniques argue that high cohesion and low coupling between the files or modules of a system lead to higher quality code that is less prone to errors. The rationale for these beliefs is that increased communication between program units raises the chances of miscommunication and can lead to mismatches between parameters and misunderstanding of input and output values.

If these beliefs are correct, then variables that measure the extent of inter-file communication could be useful predictors of faults. We examined a set of variables that relate to the calling structure of C and C++ programs, and built prediction models that included those variables.

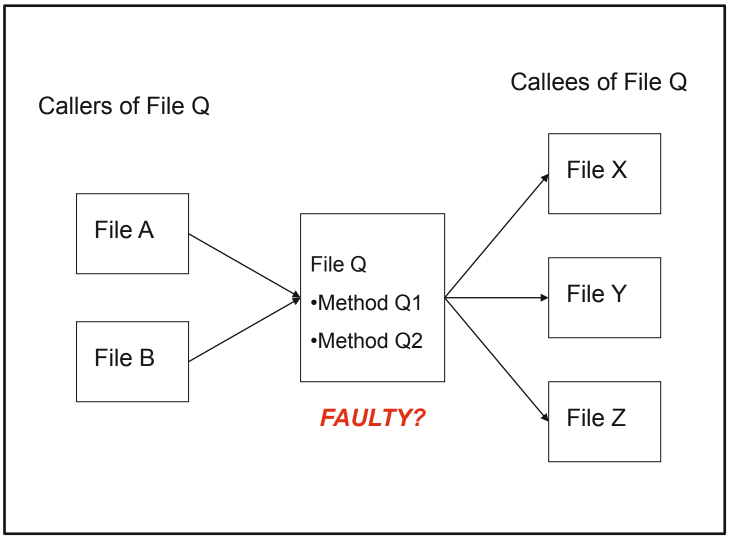


**Fig. 9.** Calling Structure

Figure 9 shows the basic calling structure relations that we use. Because our fault and change count data are at the file level, we take the caller-callee relations of methods and apply them to the files that contain the methods. Files A and B contain code that calls Method Q1 or Q2 in file Q, making A and B *callers* of Q. Files X, Y and Z contain methods that are called or invoked by Method Q1 or Q2 in file Q. X, Y and Z are *callees* of Q. Q is a caller of X, Y and Z, and a callee of A and B. Since the general expectation is that fault-proneness is related to the degree of inter-file communication, we considered the following attributes of a file in the release being predicted as variables to augment the standard prediction model.

– number of callers and callees
– number of new callers and callees
– number of prior new callers and callees
– number of prior changed callers and callees
– number of prior faulty callers and callees
– ratio of internal calls to total calls

A new caller or callee is a file that is new in the current release (the release being predicted). A prior {new, changed, faulty} caller or callee is a file that was {new, changed, or faulty} in the previous release. For example, the number of *prior faulty callers of file F* is the number of files with faults in the previous release that contain calls to any method in F in the current release.

Predictor variables come from two broad classes of system attributes, which we call *Code attributes* and *History attributes.* Code attributes are obtainable directly from the code of the system; they include file size, file type, code complexity, and dependency relations. History attributes include counts of faults and changes to the code in previous releases, and counts of developers who touched the code in the past. The standard prediction model described in Section 2 contains both code and history attributes, but no calling structure variables.

To determine the effectiveness for fault prediction of the calling structure attributes, we built and evaluated single variable models for each of the attributes used in the standard model, as well as the calling structure attributes. Each model started with a set of dummy variables for the release numbers, to avoid the possibility that values of a particular variable that are unique to a particular release would skew the resulting final model. Then, starting from the best single variable predictor, we tried adding each unused variable to the latest best model, measured the accuracy of the augmented model, and chose the single variable that most improved the accuracy. The process of adding a new predictor variable was continued as long as the improvement of the newest model over the previous one had significance with P-value $\leq .001$. Figure 10 is a pseudo-code description of the process.

The model construction procedure was carried out for systems Maintenance A and B. The code of Maintenance A is written in many different languages, but over 70% of its files and 90% of its lines of code is either C, C++ or C++ with embedded SQL (C-Sql). Maintenance B is almost entirely C++. The code structure tool we used was only able to analyze C and C++ code, so we restricted attention to those file types. To make a fair comparison between the original standard model and models that include calling structure variables, we derived new models using only the C, C++ and C-Sql files for both systems. Models were constructed using three different sets of attributes:

– code and history attributes, including calling structure attributes
– code and history attributes, without any calling structure attributes
– code attributes only, including calling structure attributes

Contrary to expectations, adding calling structure information to the standard model did not result in any improvement to the prediction results, and in fact yielded slightly poorer fault identification. When all attributes were considered, the best uni-variate model for Maintenance A is based on Prior Changes, and identifies 68% of the faults in the top 20% of the predicted files. The final Maintenance A model contains Prior Changes, KLOC, NewFile, CSql-file, PriorFaultyCallees, Callers, C-file, and PriorFaultyCallers. This model identifies 77.1% of the faults in the top 20% of files.

For Maintenance B, the best uni-variate model is based on KLOC, and it identifies 88% of the faults in the top 20% of files. The final model contains KLOC,

```
Initialize:
    Model M consists solely of RelNum
    A = a subset of the attributes

Add Variable:
    for each attribute a in A:
        construct a model consisting of $M \cup \{a\}$
        create predictions for the system
        evaluate the prediction improvement
    determine the single attribute $a'$ that maximizes the improvement
    if *P-value* $\geq 0.001$, STOP. M is the final model.
    else
        add $a'$ to M
        remove $a'$ from A
        if A is empty, STOP. M is the final model.
        else repeat the Add Variable step
```

**Fig. 10.** Pseudo-code for Model Construction

PriorDevelopers, Cohesion, Newfile, CumulativeDeveloper, Priorchanges, PriorNewFile, and PriorNewCallees. This model identifies 91.3% of the faults in the top 20% of files. The corresponding best models without calling structure attributes identify 77.7% of faults for Maintenance A and 91.4% for Maintenance B.

Because some projects might not have tools to collect and maintain fault and change history, we also evaluated models that used only code and calling structure attributes that can be obtained directly from current and past code versions. These code-only models yield slightly lower results than the code & history models. For Maintenance A, the final model contains KLOC, PriorChangedCallees, NewFile, PriorNewFile, CSql-file, Callers, PriorChangedCallers, and C-file. The model identifies 75.2% of the faults in the top 20% of files. For Maintenance B, the final model contains KLOC, Cohesion, PriorChangedCallees, NewFile, PriorNewFile, and PriorNewCallees. The model identifies 90.6% of the faults.

## 5 Alternate Prediction Models

Negative binomial regression is not the only method for creating a prediction model. In this section we evaluate three alternative models, *recursive partitioning, Bayesian additive regression trees*, and *random forests*, and compare their results to the NBR results. The first two alternatives produced results that were noticeably inferior to the NBR results, while the third was approximately equal, but computationally more expensive.

All prediction algorithms are developed by analyzing a set of data with known outcomes, and finding an algorithm that fits that data. The data with known outcomes are the *observations*, also known as the *training data.* The algorithm is then applied to new data to produce predictions. In the case of fault prediction for release N of a system, the known data is all the information available about files in releases 1 through N-1, including their fault counts. Negative binomial regression uses the observations to develop a linear formula with predicted fault count as its output.

## 5.1 Recursive Partitioning

Recursive partitioning [2] builds a binary decision tree where each leaf node represents a group of observations whose output values are all close. Each interior node of the tree is a decision based on the value of a single predictor variable. When the tree based on all the training data has been constructed, a new observation can be sent from the root to a leaf node depending on the values of its predictor variables. The predicted output value for the new observation is the average output value of all the training observations in the leaf node's group.

Figure 11 shows a simple example of a decision tree. The interior decision nodes are in rectangles, and the leaf nodes in circles. The decision nodes are all true/false decisions, with a true outcome taking the left branch from the node. The first decision node sends an input to the left if the observation has fewer than four faults in the previous release, and to the right if there were four or more faults. A file with four or more faults in the previous release, whose language is one of b, g, or j, and which has been in the system for fewer than four releases is predicted to have four faults in the next release.

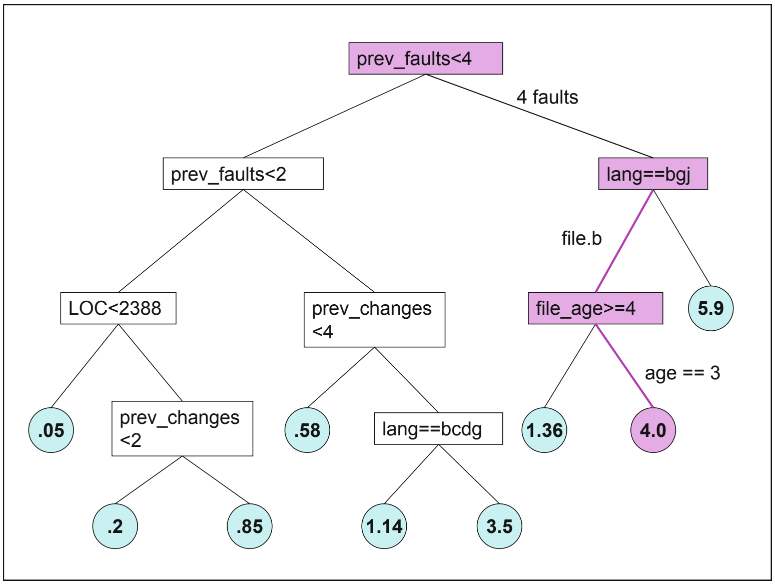


**Fig. 11.** Recursive Partitioning tree

The recursive partitioning tree is built by starting with all the training observations at the root node. The observations are split by using the single predictor variable that minimizes the total squared error of all observations on each side of the split. A single observation's error is the difference between that observation's output value and the average of all output values in the group. The process continues on the two child nodes, and recursively until it terminates according to three control parameters: *minsplit, minbucket*, and *cp*. If a node has fewer than *minsplit* observations, it will not be split further, and becomes a leaf node. If the best split of a node would result in either of its child nodes containing fewer than *minbucket* observations, then the node will not be split, and becomes a leaf node. Finally, a node will not be split unless the squared error of each of its child nodes is at least *cp* better than the error of the parent node.

For our study, we used minsplit = 7 and minbucket = 20 (the default values in the rpart package of R). We experimented with values of cp from .01 to .00001, using the data from the Maintenance A system. Naturally, the smaller the value of cp, the larger the resulting decision tree. For cp = .01, .005, .001, .0005, and .0001, the trees had, respectively, 9, 15, 45, 74, and 171 leaf nodes. Too large a value of cp results in a small tree, with the vast majority of training observations together in a single leaf node. Too small a value can result in overfitting to the training data, and poor results for predicting outcomes on new data. For Maintenance A and B, *cp* = .0005 produced trees that put the highest percent of faulty files in the the top 20% of predicted files. For Maintenance C, *cp* = .001 yielded the best tree, with *cp* = .0005 a close second.

## 5.2 Random Forests

A drawback of recursive partitioning is that the predictions from the decision tree are highly dependent on the first few splits made at and near the top of the tree. The random forests technique [1] avoids this by creating many randomized trees, and basing predictions on the average of their results. Randomizing is accomplished in two ways, first by basing each split on a subset of the training data, and second by choosing the variable on which to make each split from a small, randomly chosen subset of the predictor variables. The training data subset for each split is a random choice of observations, with replacement, from the entire training set.

Our random forest predictions were done using the RandomForest package from R. At each node split, two randomly chosen predictor variables were evaluated to create the split. We considered forests of 1, 20, 100, and 500 trees, and obtained the most accurate predictions with 500 trees.

The accuracy of prediction results from random forests was better than that from the single tree recursive partitioning approach, and was close to the results obtained from the standard NBR model.

### 5.3 Bayesian Additive Regression Trees

While the predictions from the random forests model is the average of predictions from many trees, Bayesian additive regression trees (BART) makes predictions by summing the output of many trees, each of which models a part of the output variable's predicted behavior. The BART method is described in detail by Chipman et al. [3]. Our BART models were built using the BayesTrees package in R, which allows varying the key parameters of the method to improve the results. Our best results were obtained with models that created 100 trees, but they were not as good as either the NBR or the random forest results.

### 5.4 Comparing Prediction Results

Predictions from the four modeling methods were compared on the Maintenance A, B, and C systems.

Figure 12 shows the fault-percentile averages of the four different modeling methods discussed above, applied to releases 3-35 of Maintenance A. The NBR and Random Forest models, shown with solid lines, track each other quite closely, and are relatively consistent over all the releases. The other two models, shown with dotted lines, have noticeably lower results on some releases, and also are much more erratic.

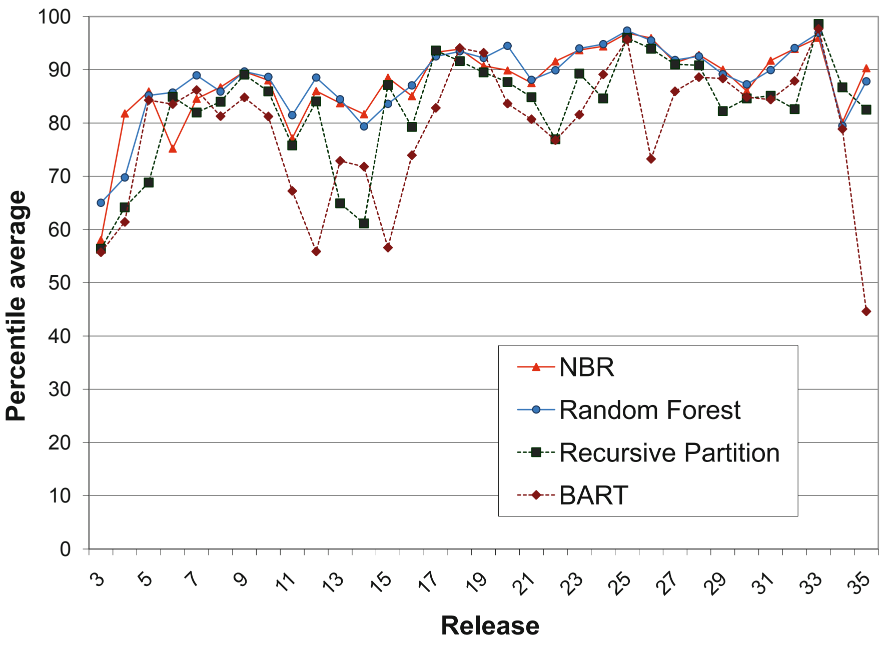


**Fig. 12.** Comparison of Alternate Prediction Models

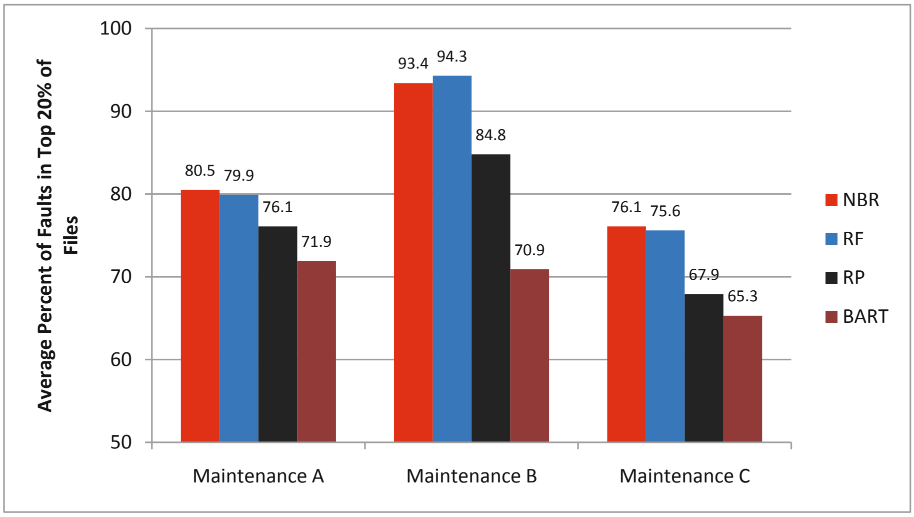


**Fig. 13.** Average Percent of Faults for 3 Maintenance Systems

In Figure 13, the average performance of each model is shown for the three maintenance systems. NBR and Random Forests clearly are superior to the other two models.

## 6 Conclusions

This chapter has provided a description of our research on software fault prediction models. We have described our standard model, and shown the results of its use on nine different long-lived industrial software systems. Collectively we have made predictions for almost 180 different releases of these systems. In spite of the differing functionality of the systems, development and testing personnel, corporation that wrote and maintained them, development methodologies, and level of maturity, our standard model always behaved very well.

Using the 20% metric, we were able to correctly identify files that accounted for between 75% and 93% of the actual defects occurring in the system. For the six systems for which we also used the fault-percentile average metric for assessment, the percentages ranged from 88% to 95% of the actual defects occurring in the system.

We also studied the impact of augmenting our models with a variety of different intuitively appropriate variables including three different ways of incorporating the number of different developers who modified a file, and the past behavior of individual developers who modified a file. In all cases, there was little or no improvement to the predictive accuracy above the standard model. Similarly, when adding calling structure information to the standard model, the improvement was at best minor.

While we believed that our standard model which used Negative Binomial Regression, provided excellent results, we wanted to make sure that a different model using the same variables did not behave even better. We therefore made predictions for several of the systems using three different models: Recursive Partitioning, Random Forests, and Bayesian Additive Regression Trees (BART). We observed that our standard model behaved better than Recursive Partitioning and BART, and while Random Forests behaved comparably, it did so at significantly greater computational cost.

We now have an automated tool that automatically extracts necessary data from the configuration management system, builds the prediction model and reports the results in terms of an ordered list of the files most likely to contain the largest numbers of defects.

## 7 Literature

Much of the material presented in this chapter is discussed in greater detail in a series of publications by the authors. An initial study of faults in large systems and their association with various characteristics of the software was presented in [5]. This study was the initial foundation for identifying attributes that might be useful independent variables for a fault prediction model.

The standard negative binomial regression model first described in [6], and applied to the Inventory 1 and Provisioning 1 systems. The developer count attributes were studied and applied to systems Maintenance A, B and C in [10]. Information about individual developers and its possible use to produce more accurate predictions was evaluated in [7].

Different algorithms for building prediction models were examined in [11]. Calling structure relations and their application to prediction models were studied in [8].

**Acknowledgments.** The research described in this chapter was performed by the authors, in partnership with our colleague Robert M. Bell from AT&T Labs - Research. Yonghee Shin contributed to the research described in the Calling Structure section, while Andy Nocera and Andrew Gauld of AT&T contributed to the construction of the automated prediction tool.

## References

1. Breiman, L.: Random Forests. Machine Learning 45, 5–32 (2001)
2. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth, Belmont (1984)
3. Chipman, H.A., George, E.I., McCulloch, R.E.: BART: Bayesian Additive Regression Trees (2008), `http://arxiv.org/abs/0806.3286v1`
4. McCullagh, P., Nelder, J.A.: Generalized Linear Models, 2nd edn. Chapman and Hall, London (1989)

5. Ostrand, T.J., Weyuker, E.J.: The Distribution of Faults in a Large Industrial Software System. In: International Symposium on Software Testing and Analysis (ISSTA 2002), pp. 55–64. ACM Press, New York (2002)
6. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans. on Software Engineering 31(4), 340–355 (2005)
7. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Programmer-based Fault Prediction. In: Predictive Models for Software Engineering (PROMISE 2010). ACM Press, New York (2010)
8. Shin, Y., Bell, R.M., Ostrand, T.J., Weyuker, E.J.: On the use of calling structure information to improve fault prediction. Empirical Software Eng. (July 2011), http://www.springerlink.com/content/r4q76v4317148451/
9. Weyuker, E.J., Ostrand, T.J., Bell, R.M.: We're Finding Most of the Bugs, but What are we Missing? In: 3rd International Conference on Software Testing. IEEE Press, New York (2010)
10. Weyuker, E.J., Ostrand, T.J., Bell, R.M.: Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. Empirical Software Eng. 13(5), 539–559 (2008)
11. Weyuker, E.J., Ostrand, T.J., Bell, R.M.: Comparing the Effectiveness of Several Modeling Methods for Fault Prediction. Empirical Software Eng. 15(3), 277–295 (2010)

# Natural Language-Based Software Analyses and Tools for Software Maintenance

Lori Pollock[1], K. Vijay-Shanker[1], Emily Hill[2], Giriprasad Sridhara[1], and David Shepherd[3,⋆]

[1] Computer and Information Sciences, University of Delaware, Newark, DE 19716
{pollock,vijay,gsridhar}@cis.udel.edu
[2] Computer Science, Montclair State University, Montclair, NJ 07043
hillem@mail.montclair.edu
[3] ABB Inc., US Corporate Research
davidshepherd@gmail.com

**Abstract.** Significant portions of software life cycle resources are devoted to program maintenance, which motivates the development of automated techniques and tools to support the tedious, error-prone tasks. Natural language clues from programmers' naming in literals, identifiers, and comments can be leveraged to improve the effectiveness of many software tools. For example, they can be used to increase the accuracy of software search tools, improve the ability of program navigation tools to recommend related methods, and raise the accuracy of other program analyses by providing access to natural language information. This chapter focuses on how to capture, model, and apply the programmers' conceptual knowledge expressed in both linguistic information as well as programming language structure and semantics. We call this kind of analysis Natural Language Program Analysis (NLPA) since it combines natural language processing techniques with program analysis to extract information for analysis of the source program.

**Keywords:** software maintenance, natural language program analysis, software engineering tools.

## 1 Introduction

Despite decades of knowledge that software engineering techniques can reduce software maintenance costs, focusing on fast initial product releases and leveraging existing legacy systems means that as much as 90% of software life cycle resources are spent on maintenance [22]. Software engineers continually identify and remove bugs, add or modify features, and change code to improve properties such as performance or security. Often, the maintainer is a newcomer to the whole system or the parts of the system relevant to the maintenance task. Before

⋆ The authors' work in this area has been supported by the National Science Foundation Grants No. CCF-0702401 and CCF-0915803.

they can safely make changes, they need to perform tasks that sometimes involve tedious, error-prone collection and analysis of information over these large, complex systems to understand the code adequately for making good decisions.

By automating error-prone tasks that do not require human involvement and presenting that information to maintainers in a useful way, software tools and environments can reduce high maintenance costs. Software development environments now include tools for searching for relevant code segments, navigating the code, providing contextual information at a given program point, and many other kinds of information and predictions that help determine where to make changes, the kinds of changes to make and their potential impact.

Automated program analyses historically build models of the program using the static programming language syntax and semantics and dynamic information from executing the program, and then perform analysis over the model to gather and infer information that is useful for the software maintainer. While the syntax of the program and the associated programming language semantics convey the intended computations to the computer system, programmers often convey the domain concepts through identifier names and comments. This natural language information can be very useful to a wide variety of software development and maintenance tools, including software search, navigation, exploration, debugging, and documentation.

This chapter focuses on how to capture, model, and apply the programmer's conceptual knowledge expressed in both linguistic information rooted in words as well as programming language structure and semantics. We call this kind of analysis, Natural Language Program Analysis (NLPA), since it combines natural language processing techniques with program analysis to extract natural language information from the source program. The underlying premise is that many developers attempt to make their code readable and follow similar patterns in their naming conventions. NLPA leverages these common patterns and naming conventions to automatically extract useful natural language information from the source code.

A number of researchers have studied what these naming conventions are and how they can be leveraged in software engineering tools. Researchers have gained insight into naming convention habits by studying how developers write identifiers and name methods [12, 50, 53] and use domain terms in source code [31], as well as by studying how source code vocabulary in identifiers evolves across multiple versions [2, 3]. These studies have led to guidelines for proper naming [18, 39, 41, 48] and using these conventions to debug poorly named methods [40], as well as identifying synonyms for words used in source code [38] and building dictionaries of verb-object relations [32]. Natural language information has been used by software engineering tools to search source code for concern location [66, 54, 74, 78], find starting points for bug localization [57, 75], automatically recover traceability links between software artifacts [1, 4, 17, 19, 64, 68, 76, 103], assemble and assess software libraries [58, 69], assess code quality [52, 65], and mine topics from source code [46, 55, 67].

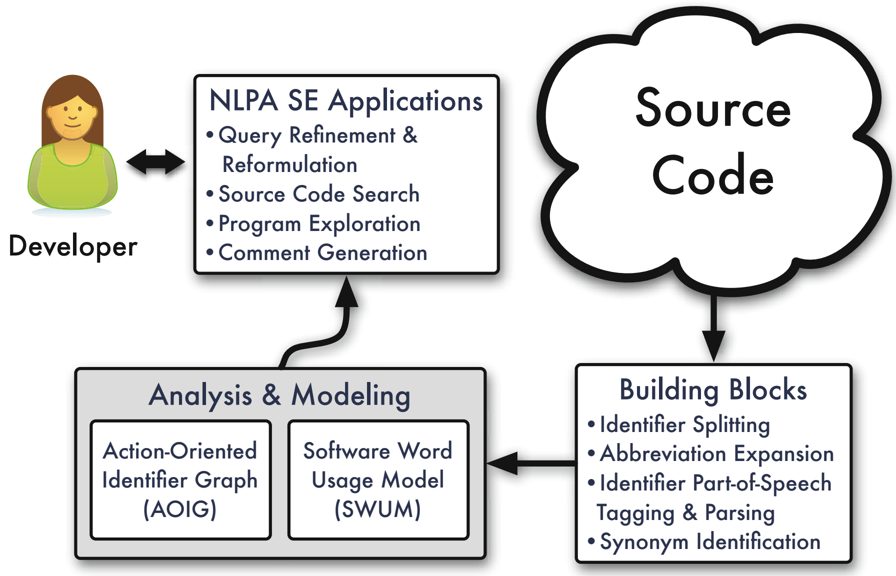


**Fig. 1.** Chapter Overview

Most existing tools that leverage words in identifiers treat a program as a "bag of words" [63], i.e., words are viewed as independent occurrences with no relationships. Not taking advantage of the natural language semantics captured by the relationships between words can lead to reduced accuracy. For example, consider searching for the query "add item" in a shopping cart application. The presence of "add" and "item" in two separate statements of the same method does not necessarily indicate that the method is performing an "add item" action—the method may be *adding an action* to the system's queue and then *getting the item field* of another object in the system. Ignoring the relationships between words causes irrelevant results to be returned by the search, distracting the user from the relevant results. This suggests that richer semantic representations of natural language information in source code may lead to more accurate software engineering tools.

In this chapter, we provide an overview of NLPA that accounts for *how words occur together in code*, rather than just counting frequencies. NLPA can be used to (a) increase the accuracy of software search tools by providing a natural language description of program artifacts to search, (b) improve the ability of program navigation tools to recommend related procedures through natural language clues, (c) increase the accuracy of other program analyses by providing access to natural language information, and (d) enable automatic comment generation from source code.

Figure 1 depicts an overview of the relationship of the following sections. We begin with the building blocks of NLPA—preprocessing source code identifiers to enable analysis of the natural language for software engineering tools. We then motivate the focus on analyzing the natural language of source code from the perspective of verbs and actions, and present an overview of a model of software word usage in source code to serve as the underlying model for NLPA-based applications. We describe briefly how NLPA has been used to improve several applications, including search, query reformulation, navigation, and comment generation. We conclude by summarizing the state of NLPA and future directions in preprocessing, analysis, and applications of NLPA.

## 2 Building Blocks

In this section, we discuss the problems of automatically splitting individual identifiers into component words, automatically expanding abbreviations which are so common in identifiers, automatically tagging the part-of-speech of individual words as used in different identifier contexts, and automatically learning synonyms used in programs to connect similar programmer intent.

### 2.1 Identifier Splitting

A key first step in analyzing the words that programmers use is to accurately split each identifier into its component words and abbreviations. Programmers often compose identifiers from multiple words and abbreviations, but unlike English writing, program identifiers cannot contain spaces (e.g., ASTVisitorTree, newValidatingXMLInputStream, jLabel6, buildXMLforComposite). Automatic splitting of multi-word identifiers is straightforward when programmers follow conventions such as using non-alphabetic characters (e.g., "_" and numbers) to separate words and abbreviations, or camel-casing (where the first letter of each word is upper case) [12, 18, 49, 53]. However, camel casing is not followed in certain situations, and may be modified to improve readability (e.g., ConvertASCIItoUTF, sizeof, SIMPLETYPENAME).

An identifier-splitting algorithm takes a given set of identifiers as input and outputs the set of substrings partitioning the identifier. An identifier $t = (s_0, s_1, s_3, ...s_n)$, where $s_i$ is a letter, digit, or special character. Most algorithms first separate the id before and after each sequence of special characters and digits, and each substring is then considered as a candidate to be further split. For these alphabetic terms, there are four possible cases to consider in deciding whether to split at a given point between $s_i$ and $s_{i+1}$:

1. $s_i$ is lower case and $s_j$ is upper case (e.g., getString, setPoint)
2. $s_i$ is upper case and $s_j$ is lower case (e.g., getMAXstring, ASTVisitor)
3. both $s_i$ and $s_j$ are lower case (e.g., notype, databasefield, actionparameters)
4. both $s_i$ and $s_j$ are upper case (e.g., NONNEGATIVEDECIMALTYPE)

Case (1) is the natural place to split for straightforward camel case without abbreviations. Case (2) demonstrates how following strict camel casing can provide incorrect splitting (e.g., get MA Xstring). We call the problem of deciding where to split when there is alternating lower and upper case present, the *mixed-case* *id splitting problem.* We refer to cases (3) and (4) as the *same-case id splitting problem.*

Currently, there exists a small set of identifier-splitting algorithms. Some depend on dictionaries, some exploit the occurrence of component words in other parts of the source code and use frequencies, and some combine this information. The greedy approach [24] is based on a predefined dictionary of words and abbreviations, and splits are determined based on whether the word is found in the dictionary, with longer words preferred. The Samurai [20] approach is based on the premise that strings composing multi-word identifiers in a given program are most likely used elsewhere in the same program, or in other programs. Thus, Samurai mines string frequencies from source code, and builds a program-specific frequency table and a global frequency table from mining a large corpus of programs. The frequency tables are used in the scoring function applied during both mixed-case splitting and same-case splitting.

GenTest [51] focuses on the same-case splitting problem. Given a same-case term, GenTest first generates all possible splittings. Each potential split is scored (i.e., tested) and the highest scoring split is selected. The scoring function uses a set of metrics ranging over term characteristics, dictionaries and information from non-source code artifacts, and information derived from the program itself or corpus of programs. The Dynamic Time Warping approach [59] is based on the observation that programmers build new identifiers by applying a set of transformation rules to words, such as dropping all vowels. Using a dictionary containing words and terms belonging to the application domain or synonymous, the goal is to identify a near optimal matching between substrings of the identifier and words in the dictionary, using an approach inspired by speech recognition.

Enslen, et al.'s [20] empirical study showed that Samurai misses same-case splits identified by the Greedy algorithm but outperforms Greedy overall by making significantly fewer oversplits. Lawrie, et al. [51] results from comparing Greedy, Samurai, and GenTest on same-case identifier splitting for Java showed that both GenTest and Samurai achieve at least 84% accuracy in identifier splitting, with GenTest achieving slight higher accuracy than Samurai. The Dynamic Time Warping approach has not been evaluated against the other techniques yet.

## 2.2 Abbreviation Expansion

When writing software, developers often use abbreviations in identifier names, especially for identifiers that must be typed often and for domain-specific words used in comments. Most existing software tools that use the natural language information in comments and identifiers do nothing to address abbreviations, and therefore may miss meaningful pieces of code or relationships between software artifacts. For example, if a developer is searching for context handling code, she might enter the query 'context'. If the abbreviation 'ctx' is used in the code instead of 'context', the search tool will miss relevant code.

Abbreviations used in program identifiers generally fall into two categories: single-word and multi-word. Single-word abbreviations are short forms whose long form (full word expansion) consists of a single word, such as 'attr' (attribute) and ‘src’ (source). Single letter abbreviations are also commonly used, predominantly for local variables with very little scope outside a class or method [53], such as ‘i’ (integer). Multi-word abbreviations are short forms that when expanded into long form consist of more than one word, such as acronyms. In fact, acronyms can be so widely used that the long form is rarely seen, such as ‘ftp’ or ‘gif’. Some uses of acronyms are very localized, such as type acronyms. When creating local variables or naming method parameters, a common naming scheme is to use the type’s abbreviation. For example, a variable of the type ArrayIndexOutOfBoundsException may be abbreviated ‘aiobe’. Some multi-word abbreviations combine single-word abbreviations, acronyms, or dictionary words. Examples include ‘oid’ (object identifier) and ‘doctype’ (document type).

**Expansion Techniques.** Automatically expanding abbreviations requires the following steps: (1) identifying whether a token is a non-dictionary word, and therefore a short form candidate; (2) searching for potential long forms for the given short form; and (3) selecting the most appropriate long form from among the set of potential long form candidates.

One simple way to expand short forms in code is to manually create a dictionary of common short forms [85]. Although most developers understand that ‘str’ is a short form for ‘string’, not all abbreviations are as easy to resolve. Consider the abbreviation ‘comp’. Depending on the context in which the word appears, ‘comp’ could mean either ‘compare’ or ‘component’. Thus, a simple dictionary of common short forms will not suffice. In addition, manually created dictionaries are limited to abbreviations known to the dictionary builders.

More intelligent abbreviation expansion mechanisms have been developed for software. Lawrie, Feild, and Binkley (LFB) [35] extract lists of potential expansions as words and phrases, and perform a two-stage expansion for each abbreviation occurrence in the code. For each function f in the program, they create a list of words contained in the comments before or within the function f or in identifiers with word boundaries (e.g., camel casing) occurring in f, and a phrase dictionary created by running the comments and multi-word-identifiers through a phrase finder [25]. They create a stop word list containing programming language keywords. After stemming and filtering by the stop word list, expansion of a given non-dictionary word occurrence in a function f involves first looking in f’s word list and phrase dictionary, and then in a natural language dictionary. A word is a potential expansion of an abbreviation when the abbreviation starts with the same letter and every letter of the abbreviation occurs in the word in order. This technique returns a potential expansion only if there is a single possible expansion.

When they manually checked a random sample of 64 identifiers requiring expansion (from a set of C, C++, and Java codes), only approximately 20% of the identifiers were expanded correctly. In another quantitative study of all identifiers in their 158-program suite of over 8 million unique terms, only 7% of the total number of identifier terms were expanded by their technique; these expansions were not checked for correctness.

The AMAP abbreviation expansion approach [36] was developed with the goal of improving on this technique and including heuristics to choose between multiple possible expansions. AMAP utilizes a scoping approach similar to variable scoping and automatically mines potential long forms for a given short form from the source code. AMAP creates a regular expression from the short form to search for potential long forms. When looking for long forms, AMAP starts at the closest scope to the short form, such as type names and statements, and gradually broadens its scope to include the method, its comments, and the class comments. If the technique is still unsuccessful in finding a long form, it attempts to find the most likely long form found within the program and then in Java SE 1.5. With each successive scope, AMAP includes more general, i.e., less domain-specific, information in its long form search.

In an evaluation of 227 non-dictionary words randomly selected from 5 open source programs, AMAP automatically expanded 63% of the words to the correct long form. On the same set, LFB expanded 40% correctly.

## 2.3 Part-of-Speech Tagging and Identifier Parsing

To facilitate extraction of relations between words appearing in identifiers, the identifier splitting and abbreviation expansion are followed by identifying (i.e., tagging) the parts of speech of each word in the identifier and then the lexical components of the identifier. Each word in an identifier is tagged with a part-of-speech such as noun, noun modifier, verb, verb modifier, or preposition, and identifiers are then chunked into phrases such as noun phrases, verb groups, and prepositional phrases. In the example below, words in identifier `findFileInPaths` are tagged as verb - noun - preposition - noun, and chunked as follows:

```
File findFileInPaths(): [find]: VG [file]: NP [in [paths]: NP]: PP
```

Part-of-speech tagging and identifier parsing in software is complicated by several programmer behaviors. Programmers invent new non-dictionary words and their own grammatical structures when naming program elements. For instance, they form new adjectives by adding "able" to a verb (e.g., "give" becomes "givable"). Thus, traditional parsers for natural language fail to accurately capture the lexical structure of program identifiers.

Liblit, et al. identified common morphological patterns and naming conventions [53], which can serve as a starting point for parsing rule development. Høst and Østvold created a phrase book of commonly occurring method name patterns [41]. Although the contents of the phrase book can be used to generate more accurate semantic representations, the rules cannot be applied to parse arbitrary method signatures.

To represent each method name as a verb and direct object, Shepherd, et al. [88] used WordNet [70] to approximate possible parts of speech for words in method names, favoring the verb tag for words in the first position because methods typically encapsulate actions. Through manual analysis of function identifiers, Caprile and Tonella developed a grammar for function identifiers [12], and applied it to an identifier restructuring tool [13]. Hill [34] developed a set of identifier grammar rules for Java, as part of the construction of the Software Word Usage Model (SWUM) defined in Section 3. Caprile and Tonella's grammar shares similarities with the SWUM grammar rules; however, SWUM's rules cover a broader set of identifiers and were developed using a much larger code base (18 million LOC in Java versus 230 KLOC in C). Because Caprile and Tonella's grammar was developed exclusively on C code, the similarities between their grammar and SWUM's provides further evidence that the construction rules built for Java can translate to other languages.

Parsing rules can be developed by identifying common word and grammar patterns from a corpus of programs, developing a generalized set of rules based on the word and grammar patterns, evaluating their effectiveness, and then iterating to expand the set of parsing rules to capture more identifier categories accurately. Using this approach, Hill, et al. [34] developed an algorithm to automatically parse identifiers according to the grammar. Malik [60] improves the accuracy of the SWUM grammar by extensive POS tagging based on suffixes, discovering more kinds of method signatures due to programmer conventions, and using context frequencies in determining POS tags.

## 2.4 Synonyms in Programs

A human developer skimming for code related to "removing an item from a shopping cart" understands that the method deleteCartItem(Item) is relevant, even though it uses the synonym *delete* rather than *remove*. Similarly, automated tools such as code search and query reformulation need to automatically recognize these synonym relations between words to be able to successfully help humans find related code in large-scale software systems. In fact, knowledge of word relations such as synonyms, antonyms, hypernyms, and hyponyms can all aid in improving the effectiveness of software tools supporting software maintenance activities.

Broadly defined, search tools use queries and similarity measures on software artifacts (source code, documentation, maintenance requests, version control logs, etc.) to facilitate a particular software engineering or program comprehension task. Tools which use natural language or keyword queries and matching can benefit by expanding queries and adding related words to textual artifact representations. For example, synonyms are especially useful in overcoming vocabulary mismatches between the query and software artifacts, especially with regard to the concept assignment problem [8].

Several software maintenance tools have been developed that use some notion of synonyms. FindConcept [87] expands search queries with synonyms to locate concerns more accurately in code. FindConcept obtains synonyms from WordNet [70], a lexical database of word relations that was manually constructed for English text. iComment [98] automatically expands queries with similar topic words to resolve inconsistencies between comments and code and therefore helps to automatically locate bugs. Their lexical database of word relations was automatically mined from the comments of two large programs.

One potential technique for finding synonyms and other word relations in software is to use Latent Semantic Indexing, an information retrieval technique that uses the co-occurrences of words in documents to discover hidden semantic relations between words [64, 66]. However, since the technique is based on co-occurrences of words, the resulting word relations are not guaranteed to be semantically similar. Another approach is to use synonyms found in English text, such as the synonyms found in WordNet, for finding the synonyms used in software.

Sridhara, et al. [92] performed a comparative study of six state of the art, English-based semantic similarity techniques (used for finding various word relations) to evaluate their effectiveness on words from the comments and identifiers in software. Their results suggest that applying English-based semantic similarity techniques to software without any customization could be detrimental to the performance of the client software tools. The analysis indicated that none of the techniques appear to perform well at recall levels above 25%, where recall is the percentage of true positives in related word pairs returned. In general, the number of returned possible synonyms can be 10 times greater than the number of desired results, and much more for high levels of recall. Sridhara, et al. propose two promising strategies to customize the existing semantic similarity techniques to software: (1) augment WordNet with relations specific to software, possibly by mining word relations in software, or (2) improve the estimation of word probabilities used by the information content-based techniques, which currently use a probability distribution of words based on English text.

Falleri et al. [23] use English-based POS tagging to automatically extract and organize concepts from program identifiers in a WordNet-like structure, focusing on hyperonym and hyponym relations between the extracted concepts. They share similar insights into the challenges of applying these techniques to software with [92]. Host and Ostvold [38] propose identifying synonymous verbs by associating a verb with each method, characterizing each method by a set of attributes, and measuring different forms of entropy over the corpus of methods with their attributes and associated verbs. Specifically, they investigate the effect on entropy in the corpus when they eliminate one of the verbs as a possible synonymous verb pair. If the effects are beneficial, they say that a possible synonym has been identified. Their results showed that they could identify reasonable synonym candidates for many verbs, but choosing genuine synonyms among the candidates will require a more sophisticated model of the abstract semantics of methods.

## 3 Analysis and Modeling

For software maintenance tasks, action words are central because most maintenance tasks involve modifying, perfecting, or adapting existing actions [88]. Actions tend to be scattered in object-oriented programs, because the organization of actions is secondary to the organization of objects [100]. Like English, typically actions (or operations) in source are represented by verbs, and nouns correspond to objects [10]. In a programming language, verbs usually appear in the identifiers of method names, possibly in part because the Java Language Specification recommends that "method names should be verbs or verb phrases and class types should be descriptive nouns or noun phrases" [30]. Therefore, initial extraction efforts focused on method names and the surrounding information (i.e., method signatures and comments).

In English or software, identifying the verb in a phrase does not always fully describe the phrase's action. To fully describe a specific action, it is important to consider the *theme*. A *theme* is the object that the action (implied by the verb) acts upon, and usually appears as a direct object (DO). There is an especially strong relationship between verbs and their themes in English [14]. An example is (parked, car) in the sentence "The person parked the *car*." Similarly, in the context of a single program code, often verbs, such as "remove," act on many different objects, such as "remove attribute", "remove screen", "remove entry", and "remove template". Therefore, to identify specific actions in a program, it is important to examine the direct objects of each verb (e.g., the direct object of the phrase "remove the attribute" is "attribute").

Initial NLPA [28, 88, 87] analyzed the source code to extract action words, in the form of verb-DO pairs. A verb-DO pair is defined to be two terms in which the first term is an action or verb, and the second term is a direct object for the first term's action. The program is modeled by an action-oriented identifier graph (AOIG) that explicitly represents the occurrences of verbs and direct objects of a program, mapping each verb-DO pair with their occurrences in the code.

The analysis focuses on occurrences of verbs and DOs in method declarations and comments, string literals, and local variable names within or referring to method declarations. After identifier splitting, POS tagging and chunking, a set of verbs is extracted from the method signature. POS tagging typically finds verbs in either the leftmost position or rightmost position or not in the signature. Special verbs such as "run" and "convert", which do not occur explicitly in names, are implicitly identified through common word patterns in identifiers. The direct object is identified based on the position of the verb. If a set of extraction rules fits the same identifier, a set of verb-DO pairs is returned with the client application ultimately determining the most appropriate pair. Table 1 shows examples of the locations where verbs are identified and how the corresponding DO is identified.

The AOIG is a first step in representing the actions and themes occurring in source code as verb-DO pairs. The next step in NLPA, the Software Word Usage Model (SWUM), was developed to address two of AOIG's shortcomings: (1) any

**Table 1.** Locating Verb-DO Pairs in Method Signatures

| **Class** | **Verb** | **DO** | **Example** |
|---|---|---|---|
| Left-Verb | Leftmost word | Rest of method name | public URL *parse***Url**() |
| Right-Verb | Rightmost word | Rest of method name | public void **mouse***Dragged*() |
| Special-Verb | Leftmost word | Specific to Verb | public void **HostList**.on*Save*() |
| Unidentifiable-Verb | "no verb" | Method name | public void **message**() |

verb in a methods name is assumed to be its action, and (2) verb-DO pairs were the only NLPA information extracted from source code.

First, the AOIG's rules to extract verb-DO pairs seek to capture the method's intended action and theme as verb and DO only. This works well for methods like parseURL(String url), where the verb-DO "parse URL" accurately represents the method's action and theme. However, not all identifiable verbs and objects in a method's name capture intended actions. Consider mouseDragged(), where the method's implementation is reacting to the mouse dragged event. AOIG's rules greedily identify the verb-DO as "dragged mouse." Instead, when the action is difficult to determine from the signature alone, SWUM marks such methods as general or event handlers to indicate that the model may not accurately capture the method's action. In this example, SWUM extracts the generic "handle mouse dragged" as an event handler.

Aside from these cases, the AOIG generally identifies the correct verb and DO as the action and theme. There are some situations where the action and theme are insufficient to accurately represent the source code's intent. For example, consider searching an online store application for code related to adding an item to the shopping cart. The action-theme "add item" will occur in many different contexts, such as adding an item to a shopping cart, a wish list, or an inventory list. Although the AOIG can model "add item," it cannot model the more specific "add item to cart," "add item to wish list," or "add item to inventory" concepts that would differentiate between the relevant and irrelevant pieces of source code.

SWUM moves beyond verb-DO pairs by capturing arbitrary *phrasal concepts*, such as full verb phrases (VPs) with verbs, direct objects, indirect objects and prepositions, and noun phrases (NPs) with no identifiable verb. For example, consider the method addEntry(ItemEntry ie) in the ShoppingCart class. SWUM's phrasal concepts can capture the verb, DO, and the indirect object: "add entry to shopping cart". SWUM also associates the DO in the method's name, entry, with the equivalent parameter, ItemEntry. Thus, SWUM's phrasal concepts can capture deeper semantic relationships found anywhere in the source code. Table 2 includes additional examples of how SWUM extracts the action (e.g., verb), theme (e.g., direct object), and secondary argument (e.g., indirect object) semantic roles for different categories of method signatures, overcoming many AOIG limitations.

SWUM uses a number of heuristics to determine whether the method name should be parsed as a NP, VP, or some special class of method name. For example, boolean checkers like isVisible or containsKey are a special class of method name. Another special class is the set of general names, which include event-driven methods like actionPerformed(), keyPressed(), or Thread.run(). If a method name has not been classified as a checker, general, or beginning with a preposition, SWUM assumes the name starts with a verb in base form and moves on to identifying the verb's arguments. The direct and indirect objects are inferred by looking in the name, parameters, declaring class, and return type of the method signature.

**Table 2.** Example SWUM Representation for Method Signatures

| **Class** | **Example** | **SWUM action-theme [secondary argument]** |
|---|---|---|
| Base verb | URLHandler.parseURL(String url) | parse-URL |
| Modal verb | FIFO.canUnmount(Device device) | can unmount-device |
| Contains Preposition | DropDownButton.addToToolBar( JToolBar toolbar) | add-drop down button [*to* tool bar] |
| Event Handler | MotionListener.mouseDragged( MouseEvent e) | handle-mouse dragged |
| Begins with Preposition | HostList.onSave() | handle-on save |
| Noun Phrase | BaseList.newIterator() | get-new iterator |
| Void Noun Phrase | JoinAction.message() | handle-message |

## 4 Applications of Natural Language Program Analysis

This section surveys several client tools that have been built to demonstrate how leveraging NLPA can improve a broad range of tools useful during software maintenance.

### 4.1 Targeted Software Maintenance Tools

To modify an application, developers must identify the high-level idea, or concept, to be changed and then locate the concept's concern, or implementation, in the code. This is called the concern location problem. In object-oriented languages where the code is organized around objects or classes, action-oriented concerns, such as "play track", are scattered across different classes and files, i.e., cross-cutting.

**Source Code Search for Concern Location.** To identify code relevant to a concern, developers typically use an *iterative refinement* process [26, 33] as shown in Figure 2. In this process, the developer enters a query into a source code search tool. Depending on the relevance of the results, the user will reformulate the query and search again. This process continues until the user is satisfied with the results (or gives up). In this process, the user has two important tasks: (1) query formulation and (2) determining whether the search results are relevant.

Studies show that formulating effective natural language queries can be as important as the search algorithm itself [33]. During query formulation, the developer must guess what words were used by the original developer to implement the targeted feature. Unfortunately, the likelihood of two people choosing the same keyword for a familiar concept is only between 10-15% [29]. Specifically, query formulation is complicated by the vocabulary mismatch problem [33] (multiple words for the same topic), polysemy (one word with multiple meanings), and the fact that queries with words that frequently occur in the software system will return many irrelevant results [63].

It is very difficult to overcome these challenges by automatically expanding a query on the user's behalf. For polysemy and word frequency, the user needs to

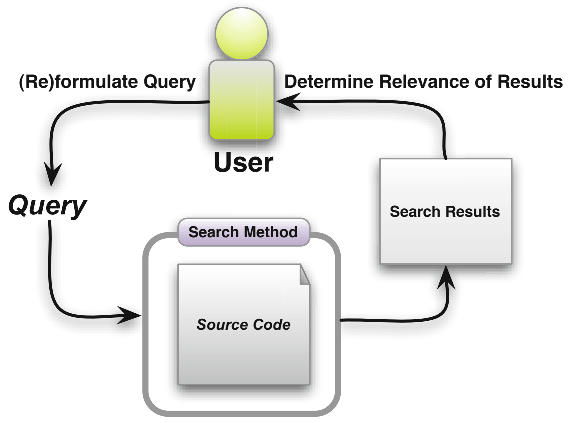


**Fig. 2.** Iterative Query Refinement and Search Process

add additional query words about the feature to restrict the search results. Such detailed knowledge about the feature exists only in the developer's mind. Further, automatically expanding a query with inappropriate synonyms can return worse results than using no expansion [92]. Thus, the role of automation is not to automatically expand the query, but to provide support that will enable the human user to quickly formulate an effective query.

Few systems recommend alternative words to help developers reformulate poor queries. One approach automatically suggests close matches for misspelled query terms [73], but does not address the larger vocabulary mismatch problem. Sections 4.2 and 4.3 present two complementary approaches that help the developer to formulate effective queries. These sections, along with Section 4.4, also present innovative ways of using NLPA to improve source code search for concern location.

**Exploring and Understanding Concerns.** Navigation and exploration tools help developers explore and understand the program structure from a starting point in the code. In general, these fall into two main categories: semi-automated approaches, which provide automatically gathered information to the user but require the developer to initiate every navigation step (stepwise), and approaches that automatically traverse the program structure and return many related elements without user intervention (recursive).

Stepwise navigation tools begin from a relevant starting element and allow developers to explore structurally related program elements such as methods, fields, or classes. Some navigation tools allow developers to query structurally connected components one edge away [15, 82, 89] or recommend structurally related elements 1-2 edges away [80, 86]. Stepwise navigation tools suggest manageable numbers of elements to be investigated, but provide limited contextual information since the developer is only presented a small neighborhood of program elements at each step. Each successive structural element to be explored must be manually selected. For example, if a developer were to use a stepwise navigation tool for an "add auction" concern consisting of 24 methods and 6 fields, the developer would have to initiate as many as 19 exploration steps.

In contrast, recursive exploration tools provide more structural context by automatically exploring many structural edges away from the starting element [95, 102, 106, 107] (e.g., by including callers 5 edges up the call chain). For instance, program slicing identifies which elements of a program may affect the data values computed at some point of interest, usually by following edges in a program dependence graph [102]. Because the number of structurally connected components can grow very quickly as new program elements are added to the result set, some recursive navigation tools (e.g., thin slicing) employ filtering techniques to eliminate unnecessary results [95]. In addition, a textual similarity metric has been used as a stopping criteria in slicing [43], which is another way of filtering.

Rather than explore data dependences, some recursive navigation techniques reduce expense by exploring the call graph [36, 107]. One approach is to filter based purely on call graph information such as the number of edges away from the starting element or the number of callees [107]. These filters can be further refined by using textual information [36]. Section 4.5 presents a recursive program exploration technique that takes advantage of NLPA.

Once the developer has located the elements of the concern, the next step is to understand the related code. Several studies have demonstrated the utility of comments for understanding software [97, 101, 105]. However, few software projects adequately document the code to reduce future maintenance costs [44, 90]. To alleviate this problem, Section 4.6 presents an NLPA-based technique to automatically generate comments directly from the source code.

## 4.2 FindConcept: A Concern Location Tool Based on the Action-Oriented Identifier Graph

FindConcept [87] is a concern location tool that leverages both traditional structural program analysis and natural language processing of source code. FindConcept improved upon the state-of-the-art by expanding the user's initial query and by searching over a natural language representation of source code (i.e., the AOIG).

When using FindConcept, developers formulate their query as a verb-DO pair (e.g., "draw circle"). FindConcept then helps the user expand their query by suggesting additional terms. FindConcept generates these suggestions by comparing the initial query to existing terms in the AOIG and by analyzing the usage patterns of these relevant words. When the user is satisfied with their expanded query, they trigger a search over the AOIG program representation, which returns search results displayed as a program graph. Displaying results as a graph of methods connected by structural edges (e.g., call graph edges) allows developers to quickly understand the concern.

Shepherd, et al. [87] evaluated FindConcept against two other concern location tools. During this evaluation, eighteen programmers completed a set of nine

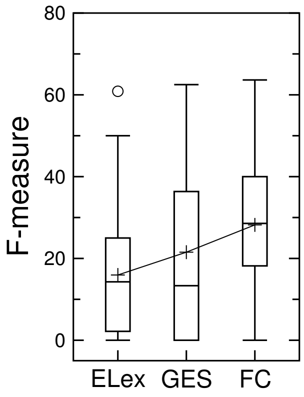


**Fig. 3.** Overall effectiveness results by search tool; FC = FindConcept

search tasks, where each search task consisted of an application and a concept to be found. Programmers were asked to use one of the three tools to complete each task (see [87] for detailed experimental setup).

FindConcept was compared with Eclipse's built-in lexical search (ELex [42]) and a modified Google Eclipse search (GES [74]). Similar to grep's functionality, ELex allows users to search using a regular expression query over source code, returning an unranked list of files that match the query. GES integrates Google Desktop Search into the Eclipse workbench, allowing users to search Java files with information-retrieval-style queries and return a set of ranked files. GES was modified to return individual methods instead of files, for comparison.

Shepherd, et al. [87] used F measure, which combines precision and recall, to measure the effectiveness of FindConcept, GES, and Elex. They measured user effort by tracking the time that users spent formulating a final query for each task. Figure 3 shows the F measure results. The box represents the inner 50% of the data, the middle line represents the median, the plus represents the mean, and outliers are represented by an '∘'. According to these measures, their study showed that FindConcept was more consistently effective than either Elex or GES, without requiring additional user effort. Analysis of the cases in which FindConcept's performance was worse or similar to GES or Elex indicated that straightforward improvements to the AOIG creation process would improve FindConcept's effectiveness. These observations have informed subsequent work in extracting natural language information from source code [34].

Based on this initial success, Hill, et al.'s [34] work has generalized FindConcept's approach, including significant contributions to the query expansion process and the creation of a more general natural language representation of source code. Their work not only extracts verb-DO pairs but entire verb phrases from source code, which avoids the issues FindConcept encountered during its evaluation.

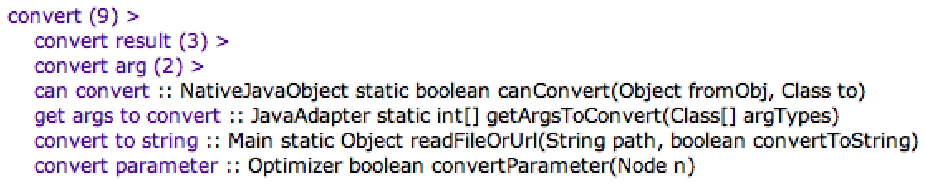
## 4.3 Contextual Query Reformulation

In addition to providing automated support to the developer in formulating queries in a different way than FindConcept, the contextual query reformulation technique [37], called *contextual search*, also helps the user discriminate between relevant and irrelevant search results. The key insight is that the *context* of words surrounding the query terms in the code is important to quickly determine result relevance and reformulate queries. For example, online search engines such as Google display the context of words when searching natural language text. The contextual search approach automatically captures the context of query words by extracting and generating natural language *phrases*, or word sequences, from the underlying source code. By associating and displaying these phrases with the program elements they describe, the user can see the context of the matches to the query words, and determine the relevance of each program element to the search.

Consider the search results for the query "convert" in Figure 4. The method signatures matching the query are to the right, with the corresponding phrases to the left. By skimming the list of words occurring with "convert" in these phrases, we notice that convert can behave as a verb which acts on objects such as "result,", "arg", or "parameter"; or convert can itself be acted upon or modified by words such as "can" and "get args to." If the user were searching for code related to "converting arguments," they could quickly scan the list of phrases and identify "convert arg" as relevant. Thus, understanding this context allows the user to quickly discard irrelevant results without having to investigate the code, and focus on groups of related signatures that are more likely to be relevant.

**Going beyond Verb-DO Queries.** The experimental study described in Section 4.2 showed that capturing specific word relations in identifiers, such as verb-DO pairs, enabled users to produce more effective queries more consistently than with two competing search tools. However, strict verb-DO queries cannot be used to search for every feature. For example, verb-DO pairs cannot be used to search for features expressed as noun phrases without a verb, such as "reserved keyword" or "mp3 player."

One potential approach to go beyond verb-DO pairs is to capture all word relation pairs in software by using co-occurrences [62]. The key problem with



```
convert (9) >
   convert result (3) >
   convert arg (2) >
   can convert :: NativeJavaObject static boolean canConvert(Object fromObj, Class to)
   get args to convert :: JavaAdapter static int[] getArgsToConvert(Class[] argTypes)
   convert to string :: Main static Object readFileOrUrl(String path, boolean convertToString)
   convert parameter :: Optimizer boolean convertParameter(Node n)
```

**Fig. 4.** Example results for "convert" query. Phrases are to the left, followed by the number of matching signatures, and signatures follow '::'.

co-occurring word pairs is that *word order matters*. For example, knowing that "item" and "add" co-occur more often than due to chance is less useful than simply knowing that the phrase "add item" frequently occurs. This observation prompted the use of phrases, based on SWUM's phrasal concepts, to develop the contextual query reformulation technique.

Contextual query reformulation relies on SWUM's phrasal concepts to extract phrases from source code because existing techniques for extracting phrases did not meet the needs of the concern location problem. There is work on automatically extracting topic words and phrases from source code [67, 71], displaying search results in a concept lattice of keywords [72], and clustering program elements that share similar phrases [46]. Although useful for exploring the overall word usage of an unfamiliar software system, these techniques are not sufficient for exploring all usage. In contrast to the contextual approach, these approaches either filter the topics based on perceived importance to the system [46, 71, 72], or do not produce human understandable topic labels [67]. Since it is impossible to predict a priori what will be of interest to the developer, the contextual approach lets the developer filter the results with a natural language query, and uses human-readable extracted phrases.

**The Contextual Query Reformulation Approach.** After using SWUM to automatically extract phrases for method signatures, the contextual query reformulation technique searches the resulting phrases for instances of the query words. Related phrases, along with the methods they were generated from, are grouped into a hierarchy based on partial phrase matching. As illustrated in Figure 4, phrases at the top of the hierarchy are more general and contain fewer words, whereas phrases more deeply nested in the hierarchy are more specific and contain more words.

An empirical evaluation with 22 developers was conducted to compare contextual search (*context*) with verb-DO (*V-DO*) recommendations without synonym suggestions from WordNet. Synonyms were not used in order to explore whether natural language phrases beyond *V-DO* improve searching capabilities, without studying effects caused by synonym recommendations or other minor algorithmic differences.

The results show that contextual search significantly outperforms V-DO recommendations in terms of effort and effectiveness. Figure 5 presents the results of the comparison in a box and whisker plot. The box represents the inner 50% of the data, the middle line represents the median, the plus represents the mean, and outliers are represented by an '×'.

In terms of effort, shown on the left, developers entered 5 more queries on average for *V-DO* than for *context*. In most cases, this was due to the fact that users found it difficult to formulate strict V-DO queries for all the concerns. One subject said,"I really liked the verb-direct object search add-on, but had trouble formulating some of the mandatory verbs, for example with the `sqrt2` query." In situations where *V-DO* could not extract a verb, users had trouble formulating successful queries and therefore expended more effort than with *context*.

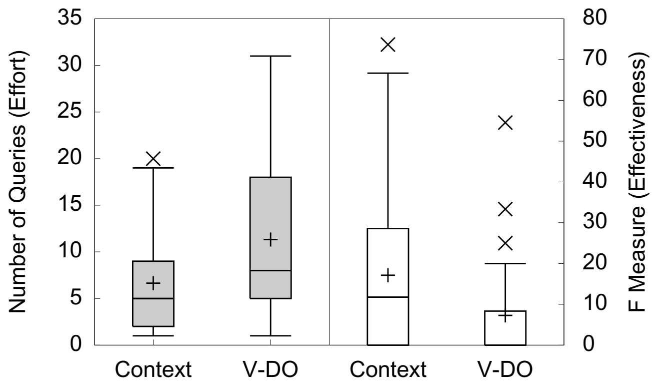


**Fig. 5.** Effort and Effectiveness Results for *context* and *V-DO*. Effort is measured in terms of the number of queries entered, shown on the left. Effectiveness is measured in terms of the F Measure, shown on the right.

*V-DO*'s inability to extract verbs in all situations also led to poor effectiveness, shown on the right in Figure 5. Although the developers found *V-DO*'s query recommendations to be helpful, the recommendations did not provide significantly improved results. For example, another subject said, "In the V-DO part especially, it was difficult to find an accurate list [of signatures] for each concern by specifying complete V-DO combinations." Thus, the more flexible phrase extraction process of *context* allowed for higher F measure values.

## 4.4 SWUM-Based Search

As described in Section 4.2, experimental results showed that AOIG-based Find-Concept is more consistently effective than two existing search techniques. Source code search effectiveness can be even further increased by taking advantage of SWUM's richer representation of natural language in a program [34]. The core of SWUM-based search is the SWUM-based scoring function, *swum*, which scores the relevance of program elements based on where the query words occur in the code by integrating location, semantic role, head distance, and usage information:

- *Location.* When a method is well-named, its signature summarizes its intent, while the body implements it using a variety of words that may be unrelated. A query word in the signature is a stronger indicator of relevance than the body.
- *Semantic role.* Prior research has shown that using semantic roles such as action and theme can improve search effectiveness [87]. That intuition is taken further by distinguishing where query words occur in terms of additional semantic roles.
- *Head distance.* The closer a query word occurs to the head, or right-most, position of a phrase, the more strongly the phrase relates to the query word.

For example, the phrase "image file" is more relevant to the concept of "saving a file" than "file server manager".
- *Usage.* If a query word frequently occurs throughout the rest of the program, it is not as good at discriminating between relevant and irrelevant results. This idea is commonly used in information retrieval techniques [63].

Individual query words are first scored based on their usage pattern in the code as well as their head distance within a phrase. This score for a phrase is then scaled based on its semantic role, where actions and themes are assigned the highest coefficient multiplier. If a method is difficult for SWUM to split or parse, purely lexical regular expressions are used to calculate the score, scaled by a low coefficient. The score from a method's signature is combined with lexical body information in the final *swum* score.

This *swum* score was compared with the existing FindConcept search results [87], except for 1 concern which was used in *swum*'s training set. The queries for *swum* were formulated by the subjects using the contextual query reformulation technique, as presented in Section 4.3. Two variants of SWUM were compared: *SWUM10*, which uses the top 10 ranked results, and *SWUMT*, which uses a more sophisticated threshold that takes the average of the top 20 results to determine relevance. Depending on how the distribution of scores is skewed, the threshold for SWUMT can be more or less than 10.

Based on the F measure shown in Figure 6, ELex appears inferior to the other search techniques. The SWUM-based techniques, SWUM10 and SWUMT, appear to be more consistently effective than FindConcept or GES. These results are confirmed by the precision and recall results. In terms of precision, SWUMT is a clear front-runner closely followed by FindConcept. For recall, SWUM10, SWUMT, and GES appear to have similar results.

It is not surprising that ELex, the technique with the worst precision, also has the best recall. For most queries in this study, ELex typically returns too many results. This ensures ELex finds many relevant results, but too many irrelevant ones. These observations independently confirm earlier results that used tools similar to ELex for feature location [5]. SWUM10 and SWUMT begin to approach ELex's high recall, without sacrificing precision.

Overall, SWUMT is a very competitive search technique when the query words match relevant signatures. However, when body information is important to locating the concern, GES is the best state of the art technique in this study. Although GES outperformed SWUMT, SWUM10, and FindConcept for some of the concerns, its performance in general seems to be unpredictable. When GES did not have the best performance, it tended to be a little better, and sometimes even worse, than ELex. In contrast, even though SWUMT did not always have the best results, it was usually competitive.

To investigate this observation, the approaches were ranked from 1–5 based on their maximum F measure score for each concern, giving ties the same rank. Using this measure, SWUMT is the most highly ranked technique with an average rank of 2.38 and a standard deviation (std) of 1.18. GES has an average of 2.75 (std 1.19), SWUM10 an average of 2.88 (std 1.64), FindConcept an average

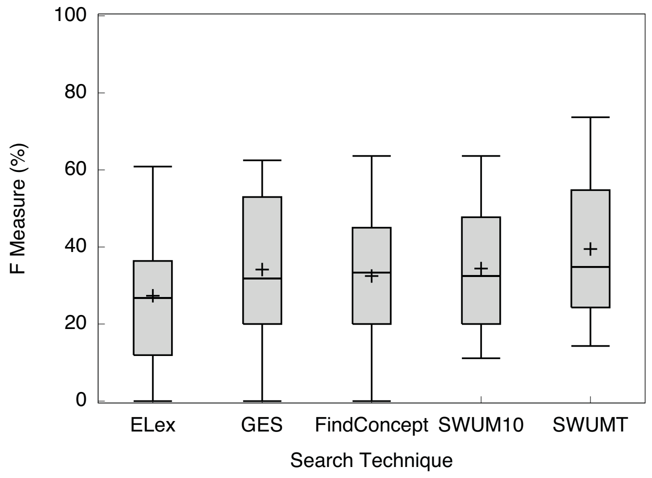


**Fig. 6.** f-measure results for state of the art search techniques

of 3.00 (std 0.93), and ELex and average of 3.50 (std 1.41). From these results, we can see that SWUMT and GES are the best overall techniques in this study, but that SWUMT is consistently ranked more highly overall.

## 4.5 Program Exploration

Despite evidence that successful programmers use program structure *as well as* identifier names to explore software [81], most existing program exploration techniques use either structural *or* textual information. Using only one type of information, current automated tools ignore valuable clues about a developer's intentions [7]—clues critical to the human program comprehension process.

By utilizing textual *as well as* structural program information, automatic program exploration tools can potentially mirror how humans attempt to understand code [47]. Combining information enables exploration tools to automatically prune irrelevant structural edges. By eliminating irrelevant edges, exploration tools can recursively search a structural program representation to provide the maintainer with a broad, high level view of the code relevant to a maintenance task—without including the entire program.

Dora the Program Explorer, or Dora, is an automatic exploration technique that takes as input a natural language query related to the maintenance task and a program structure representation to be explored [36]. Dora then outputs a subset of the program structure relevant to the query, called a *relevant neighborhood*. Dora currently uses the call graph for program structure, and takes a seed method as a starting point. By recursively traversing call edges, Dora identifies the relevant neighborhood for this seed.

Dora uses structural information by traversing structural call edges to find the set of callers and callees for the seed method. These methods become candidates for the relevant neighborhood. Dora uses textual information to score each candidates' relevance to the query. Candidates scored higher than a given threshold, $t_1 = 0.5$, are added to the relevant neighborhood. Candidates scored less than $t_1$ but more than a threshold $t_2 = 0.3$ are further explored to ensure they are not connected to more relevant methods. The exploration process is recursively repeated for each method added to the relevant neighborhood.

To determine a method's relevance to the query, Dora uses a unique similarity measure that takes into account how frequently the query words occur in the method versus the remainder of the program, as well as where the query words appear. Dora captures word frequency based on the tf-idf score commonly used in information retrieval (IR) [63]. In addition, Dora more highly weights the tf-idf of query words occurring in the method name versus the body. The weights were automatically trained using a logistic regression model on a set of 9 concerns [36].

Dora's sophisticated relevance score (*Dora*) was evaluated against two simpler relevance scores: boolean-AND (*AND*) and boolean-OR (*OR*). These techniques output either 0 or 1: *AND* outputs 1 if *all* query terms appear in the method; *OR* outputs 1 if *any* query term appears in the method. In addition, Dora was compared to a purely structural technique, *Suade* [80, 104]. These 4 techniques were compared using 8 concerns from 4 open source Java programs [83]. The 8 concerns contain a total of 160 seed methods and 1885 call edges (with overlap). For each method $m$ in the set of evaluation concerns, each scoring technique was applied to all the callers and callees of $m$, and the precision and recall for $m$ were calculated.

The results of this study are summarized in Figure 7. Each bar shows the distribution of F measures calculated for each seed method across all the concerns. The shaded box represents 50% of the data, from the 25th to 75th percentiles. The horizontal bar represents the median, and the plus represents the mean.

Since each shaded box extends from 0, at least 25% of the 160 methods considered by each technique have 0% recall and precision. However, *Dora* achieves 100% precision and recall for 25% of the data—more than any other technique. *Suade* and *OR* appear to perform similarly to one another, although *OR* has a slightly higher mean F measure. These trends were verified with a Bonferroni mean separation test. *Dora* performs significantly better than structural-based *Suade*, although neither *Dora* nor *Suade* are significantly different from *OR*. All the approaches outperform *AND* with statistical significance.

Overall, *Dora* appears to be the most successful technique, and structural-based *Suade* to be competitive with the naive textual- and structural-based *OR*. Of all the techniques, naive *AND* had the worst performance. *AND*'s poor performance indicates that simply combining textual and structural information alone does not guarantee success. The success of a textual- and structural-based (NLPA) technique is highly dependent on the performance of the textual scoring technique.

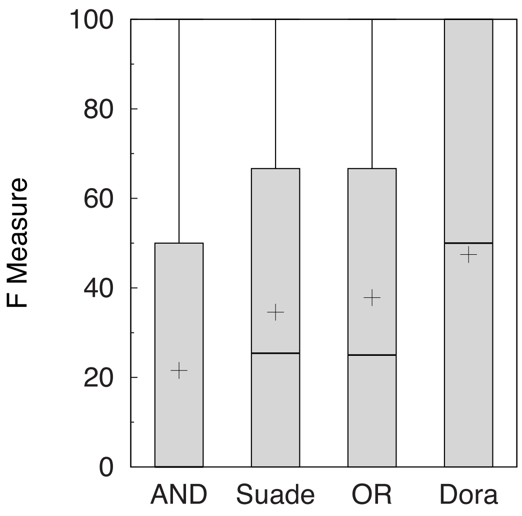


**Fig. 7.** f-measure across exploration techniques

### 4.6 Comment Generation

In spite of numerous studies demonstrating the utility of comments for understanding and analyzing software [97–99, 101, 105], few software projects adequately document the code to reduce future maintenance costs [44, 90]. Lack of comments may be fine when programmers use descriptive identifier names [27]; however, precise identifiers that accurately describe an entity lead to very long identifier names [9, 53], which can actually *reduce* code readability, Another way is to encourage the developer to write comments (1) by automatically prompting the developer to enter them [21, 79], or, (2) by using a top-down design paradigm and generating comments directly from the specification [84], or, (3) by using a documentation-first approach to development [45]. Although these solutions can be used to comment newly created systems, they are not suitable for existing legacy systems.

An alternative to developer-written comments is to automatically generate comments directly from the source code [11, 56]. These approaches are limited to inferring documentation for exceptions [11] and generating API function cross-references [56], and are not intended for generating descriptive summary comments.

This section describes how Sridhara, et al. [91, 93, 94] leveraged NLPA, specifically SWUM, to automatically generate method summary comments [91], detect high level actions in method bodies for improved summaries [93], and generate parameter comments and summaries with integrated parameter usage information [94]. The method summaries are leading comments that describe a method's intent, called *descriptive comments.* Descriptive comments summarize the major algorithmic actions of the method, similar to how an abstract provides a summary for a natural language document [61]. Descriptive parameter comments describe the high-level role of a parameter in achieving the computational intent of a method. Figure 8 shows example output from the automatic summary and parameter comment generator.

```
public static void main(String[] args) {
 int port = -1;
 try {
    port = Integer.parseInt(args[0]);
  } catch (ArrayIndexOutOfBoundsException e) {
    println("Usage: java MetaServer PORT_NUMBER");
    System.exit(-1);
  } catch (NumberFormatException e) {
    println("Usage: java MetaServer PORT_NUMBER");
     System.exit(-1);
   }
   MetaServer metaServer = null;
   try {
      metaServer = new MetaServer(port);
   } catch (IOException e) {
      logger.warning("Could not create MetaServer!");
      System.exit(-1);
   }
   metaServer.start();
}
```

**Fig. 8.** Example of a generated summary and parameter comment for a Java Method @summary : /** Start meta server */. @param args: create meta server, using args.

The key insight in automatic summary comment generation is to model the process after natural language generation, dividing the problem into sub-problems of content selection and text generation [77]. *Content selection* involves choosing the important or central code statements within a method that must be included in the summary comment. For a selected code statement, *text generation* determines how to express the content in natural language phrases in a concise yet precise manner. Figure 9 depicts the summary comment generation process with the NLPA preprocessing to build the linguistic and traditional program representations.

**Generating Method Summary Comments.** The first phase selects s_units as content for the summary, where an *s_unit* is a Java statement, except when the statement is a control flow statement; then, the s_unit is the control flow expression with one of the *if, while, for* or *switch* keywords. Heuristics for s_unit selection are based on characteristics similar to beacons [16] for summary comments, where a beacon is a surface feature which facilitates comprehension. Ending s_units are statements that lie at the control exit of a method, as methods often perform a set of actions to accomplish a final action, which is often the main purpose of the method. Same-action s_units are method calls where the callee's name indicates the same action as the method being analyzed for comment generation. A void-return s_unit is a method call that does not return a value or whose return value is not assigned to a variable; these methods often supply useful content for a summary because they are invoked purely for its side effects. These three kinds of s_units are first identified, and then their

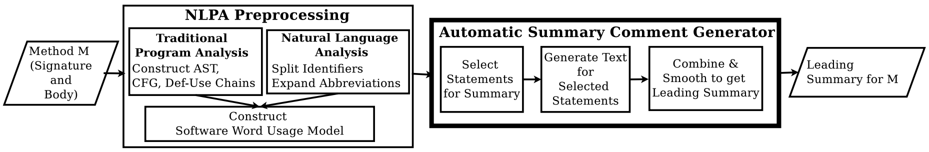

Method M (Signature and Body)
NLPA Preprocessing
Traditional Program Analysis
Construct AST, CFG, Def-Use Chains
Natural Language Analysis
Split Identifiers
Expand Abbreviations
Construct Software Word Usage Model
Automatic Summary Comment Generator
Select Statements for Summary
Generate Text for Selected Statements
Combine & Smooth to get Leading Summary
Leading Summary for M

**Fig. 9.** The Summary Comment Generation Process

data-facilitating s_units are identified, which are those s_units that assign data to variables used in these s_units. Any s_units controlling the execution of any of these s_units are included. Finally, ubiquitous operations such as logging or exception handling are filtered out.

The text generation phase determines how to express the selected s_units as English phrases and how to integrate the phrases to mitigate redundancy. For example, for the s_unit:

```
f.getContentPane().add(view.getComponent(), CENTER)
```

The output phrase is:

```
/* Add component of drawing view to content pane of frame*/
```

A naive approach to text generation is to generate a phrase based only on the statement. For example, given `print(current)`; one can generate the phrase "print current". The problem with this approach is that the name of the variable `current` alone is insufficient; the reader is left with no concept of what is being printed. The missing contextual information is `current`'s type, which is `Document`. Instead, a process called *lexicalization* is used, in which the type information of a variable is incorporated such that more descriptive noun phrases are generated for a variable.

Text generation is achieved through a set of templates. Consider a method call `M(...)`. In Java, a method implements an operation and typically begins with a verb phrase [96]. Thus, a verb phrase for `M` is generated. The template for the verb phrase is:

*action theme secondary-args*
*and get return-type* [if M returns a value]

where *action, theme* and *secondary arguments* of `M` are identified by SWUM and correspond to the verb, noun phrase and prepositional phrases of the verb phrase.

---

Selected s_unit: `os.print(msg)`
Generated Phrase: `/* Print message to output stream */`

---

There are additional templates for different constructs such as *nested method calls, composed method calls, assignments, returns, conditional* and *loop expressions.* The goal in template creation is to be precise while not being too verbose.

```
9     for (int x = 0;  x < vAttacks.size(); x++) {
10   WeaponAttackAction waa=vAttacks.elementAt(x);
11   float fDanger = getExpectedDamage(g, waa);
12   if (fDanger > fHighest) {
13     fHighest = fDanger;
14     waaHighest = waa;
15   }
16 }
17 return waaHighest;
```

**Listing 1.1.** Lines 9-16 implement a high level action. Synthesized description: "Get weapon attack action object (in vectorAttacks) with highest expected damage."

**Improving Generated Comments.** More concise and higher level summary comments are achievable if groupings of related statements can be recognized as implementing a higher level action. These same identified high level actions could also be used in *ExtractMethod* refactoring and other applications like traceability recovery and concern location. For example, the code fragment from lines 9 to 16 in Listing 1.1 implements a high level action. Sridhara, et al. [93] leverage SWUM and traditional program analysis to both identify statement groupings that form high level actions, and generate the English phrases to express them.

Similarly, leading comments are improved by parameter comments and/or integrating parameter usage information into the summary comment itself. The challenges are distinguishing the main role the parameter plays among its potentially many uses within the body, expressing that role in English, and then integrating that information into the existing summary comment. Sridhara et al. [94] developed heuristics for identification of the main parameter role using both SWUM and other information such as static estimation of execution frequency [6]. Phrases are generated such that the parameter comment is linked with the summary (i.e., there are overlapping words between the parameter comment and summary). In Figure 8, in addition to using line 4 to generate the parameter comment, line 14 is used to ensure that the parameter comment is *connected* to the summary (via "meta server").

**Evaluating Automatically Generated Comments.** Sridhara, et al. evaluated their work [91, 93, 94] by obtaining judgements of the generated comments from expert programmers. For summary comments, a majority of the developers believed that the generated summary was accurate in 87% of the evaluated methods. A majority stated that the summary comments did not miss important information in 75% of the evaluated methods. Finally, the majority noted that the synthesized summary was not too verbose in 87.5% of the evaluated methods. Similarly, the evaluation of the high level action identification and description also yielded positive results [93]. In an evaluation of 75 code fragments with identified high level actions, 192 of the 225 developer responses agreed that the synthesized description represented the high level action in the code fragments. In the evaluation of the generated comments for 33 parameters [94], 89 of the 99 developer responses agreed that the comments were accurate. 89 of the 99 responses also agreed that the comments were useful in understanding the parameter's role.

## 5 Summary

Results from various empirical evaluations described in this chapter demonstrate that Natural Language Program Analysis can significantly improve the effectiveness of tools to aid in software maintenance. By extracting phrases to represent method signatures, and using the phrases to search for query word instances in the code, developers can gain help in both reformulating search queries and discriminating between relevant and irrelevant search results. Using the Software Word Usage Model (SWUM) can improve scoring functions at the core of software search tools. SWUM has enabled automatic generation of method summary comments with parameter role information.

Experiments have also shown that the usefulness of NLPA in client applications is affected by its accuracy in extracting information from the source code. The software maintenance tools should be improved even further as identifier splitting, abbreviation expansion, part-of-speech tagging, and word relation determiners are improved for the software domain. While much progress in automation has been achieved, the empirical results thus far indicate that there is considerable room for improving all of these building blocks as well as analysis and modeling.

**Acknowledgments.** The authors greatly benefitted from work together with Zachary Fry, Eric Enslen, Sana Malik, Michelle Allen, and Divya Muppaneni.

## References

1. Abadi, A., Nisenson, M., Simionovici, Y.: A Traceability Technique for Specifications. In: ICPC 2008: Proceedings of the 16th IEEE International Conference on Program Comprehension, pp. 103–112 (2008)
2. Abebe, S., Haiduc, S., Marcus, A., Tonella, P., Antoniol, G.: Analyzing the Evolution of the Source Code Vocabulary, pp. 189–198 (2009)
3. Antoniol, G., Gueheneuc, Y.G., Merlo, E., Tonella, P.: Mining the Lexicon Used by Programmers during Sofware Evolution. In: IEEE International Conference on Software Maintenance, pp. 14–23 (2007)
4. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering Traceability Links between Code and Documentation. IEEE Transactions on Software Engineering 28(10), 970–983 (2002)
5. Antoniol, G., Gueheneuc, Y.: Feature Identification: An Epidemiological Metaphor. IEEE Transactions on Software Engineering 32(9), 627–641 (2006)
6. Ball, T., Larus, J.R.: Branch Prediction for Free. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 300–313. ACM Press, New York (1993)
7. Biggerstaff, T.J.: Design Recovery for Maintenance and Reuse. Computer 22(7), 36–49 (1989)
8. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: The Concept Assignment Problem in Program Understanding. In: ICSE 1993: Proceedings of the 15th International Conference on Software Engineering, pp. 482–498 (1993)

9. Binkley, D., Lawrie, D., Maex, S., Morrell, C.: Impact of Limited Memory Resources. In: Proceedings of the 16th IEEE International Conference on Program Comprehension (2008)
10. Booch, G.: Object-oriented Design. Ada Lett. I(3), 64–76 (1982)
11. Buse, R.P., Weimer, W.R.: Automatic Documentation Inference for Exceptions. In: International Symp. on Software Testing and Analysis, 2008, pp. 273–282. ACM (2008)
12. Caprile, B., Tonella, P.: Nomen Est Omen: Analyzing the Language of Function Identifiers. In: WCRE 1999: Proceedings of the 6th Working Conference on Reverse Engineering, pp. 112–122 (1999)
13. Caprile, B., Tonella, P.: Restructuring Program Identifier Names. In: ICSM 2000: Proceedings of the International Conference on Software Maintenance (ICSM 2000), p. 97. IEEE Computer Society, Washington, DC (2000)
14. Carroll, J., Briscoe, T.: High Precision Extraction of Grammatical Relations. In: 7th International Workshop on Parsing Technologies (2001), `citeseer.ist.psu.edu/article/carroll01high.html`
15. Chen, K., Rajlich, V.: Case Study of Feature Location Using Dependence Graph. In: IWPC 2000: Proceedings of the 8th International Workshop on Program Comprehension, pp. 241–249 (2000)
16. Crosby, M.E., Scholtz, J., Wiedenbeck, S.: The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In: 14th Workshop of the Psychology of Programming Interest Group, pp. 18–21. Brunel University (2002)
17. De Lucia, A., Oliveto, R., Tortora, G.: Assessing IR-based Traceability Recovery Tools through Controlled Experiments. Empirical Softw. Engg. 14(1), 57–92 (2009)
18. Deissenboeck, F., Pizka, M.: Concise and Consistent Naming. Software Quality Control 14(3), 261–282 (2006)
19. Eaddy, M., Aho, A.V., Antoniol, G., Gueheneuc, Y.: Cerberus: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In: ICPC 2008: Proceedings of the 16th IEEE International Conference on Program Comprehension. IEEE Computer Society, Washington, DC (2008)
20. Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining Source Code to Automatically Split Identifiers for Software Analysis. In: Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009, pp. 71–80 (2009)
21. Erickson, T.E.: An Automated FORTRAN Documenter. In: Proceedings of the 1st Annual International Conference on Systems Documentation, pp. 40–45. ACM, New York (1982)
22. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. IT Professional 2(3), 17–23 (2000)
23. Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C., Prince, V., Dao, M.: Automatic Extraction of a WordNet-Like Identifier Network from Software. In: 18th Int'l Conf. on Program Comprehension, pp. 4–13. IEEE (2010)
24. Feild, H., Binkley, D., Lawrie, D.: An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers. In: Proceedings of IASTED International Conference on Software Engineering and Applications, SEA 2006 (2006)
25. Feng, F., Croft, W.B.: Probabilistic Techniques for Phrase Extraction. Information Processing and Management 37(2), 199–220 (2001)

26. Fischer, G., Nieper-Lemke, H.: Helgon: Extending the Retrieval by Reformulation Paradigm. In: CHI 1989: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 357–362 (1989)
27. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
28. Fry, Z.P., Shepherd, D., Hill, E., Pollock, L., Vijay-Shanker, K.: Analysing Source Code: Looking for Useful Verb-Direct Object Pairs in all the Right Places. Software, IET 2(1), 27–36 (2008)
29. Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T.: The Vocabulary Problem in Human-System Communication. Communications of the ACM 30(11), 964–971 (1987)
30. Gosling, J., Joy, B., Steele, G.: Java Language Specification, (September 2006), `http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html`
31. Haiduc, S., Marcus, A.: On the Use of Domain Terms in Source Code. In: ICPC 2008: Proceedings of the 16th IEEE International Conference on Program Comprehension, pp. 113–122 (2008)
32. Hayase, Y., Kashima, Y., Manabe, Y., Inoue, K.: Building Domain Specific Dictionaries of Verb-Object Relation from Source Code. In: 15th European Conference on Software Maintenance and Reengineering (CSMR 2011), pp. 93–100. IEEE Computer Society (2011)
33. Henninger, S.: Using Iterative Refinement to Find Reusable Software. IEEE Software 11(5), 48–59 (1994)
34. Hill, E.: Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration. Ph.D. thesis, University of Delaware (2010)
35. Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., Vijay-Shanker, K.: AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. In: MSR 2008: Proceedings of the 5th International Working Conference on Mining Software Repositories, IEEE Computer Society, Washington, DC (2008)
36. Hill, E., Pollock, L., Vijay-Shanker, K.: Exploring the Neighborhood with Dora to Expedite Software Maintenance. In: ASE 2007: Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE 2007), pp. 14–23. IEEE Computer Society, Washington, DC (2007)
37. Hill, E., Pollock, L., Vijay-Shanker, K.: Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse. In: ICSE 2009: Proceedings of the 31st International Conference on Software Engineering (2009)
38. Høst, E., Østvold, B.: Canonical Method Names for Java. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 226–245. Springer, Heidelberg (2011)
39. Høst, E.W., Østvold, B.M.: The Programmer's Lexicon, Volume I: The Verbs. In: SCAM 2007: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 193–202. IEEE Computer Society, Washington, DC (2007)
40. Høst, E.W., Østvold, B.M.: Debugging Method Names. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 294–317. Springer, Heidelberg (2009)
41. Høst, E.W., Østvold, B.M.: The Java Programmer's Phrase Book. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 322–341. Springer, Heidelberg (2009)
42. IBM: Eclipse IDE (2010), `http://www.eclipse.org`

43. Ishio, T., Niitani, R., Murphy, G.C., Inoue, K.: A Program Slicing Approach for Locating Functional Concerns. Tech. rep., Graduate School of Information Science and Technology, Osaka University (March 2007), `http://sel.ist.osaka-u.ac.jp/~ishio/TR-slicing2007.pdf`
44. Kajko-Mattsson, M.: A Survey of Documentation Practice within Corrective Maintenance. Empirical Software Engineering 10(1), 31–55 (2005)
45. Knuth, D.E.: Literate Programming. The Computer Journal 27(2), 97–111 (1984)
46. Kuhn, A., Ducasse, S., Gírba, T.: Semantic Clustering: Identifying Topics in Source Code. Information Systems and Technologies 49(3), 230–243 (2007)
47. Lawrance, J., Bellamy, R., Burnett, M., Rector, K.: Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks. In: CHI 2008: Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems, pp. 1323–1332. ACM, New York (2008)
48. Lawrie, D., Feild, H., Binkley, D.: An Empirical Study of Rules for well-formed Identifiers. Journal of Software Maintenance and Evolution 19(4), 205–229 (2007)
49. Lawrie, D., Feild, H., Binkley, D.: Extracting Meaning from Abbreviated Identifiers. In: SCAM 2007: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), pp. 213–222 (2007)
50. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a Name? A Study of Identifiers. In: ICPC 2006: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006), pp. 3–12. IEEE Computer Society, Washington, DC (2006)
51. Lawrie, D., Binkley, D., Morrell, C.: Normalizing Source Code Vocabulary. In: Working Conference on Reverse Engineering (WCRE), pp. 3–12 (2010)
52. Lawrie, D., Feild, H., Binkley, D.: Leveraged Quality Assessment using Information Retrieval Techniques. In: ICPC 2006: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006), pp. 149–158. IEEE Computer Society, Washington, DC (2006)
53. Liblit, B., Begel, A., Sweetser, E.: Cognitive Perspectives on the Role of Naming in Computer Programs. In: Proceedings of the 18th Annual Psychology of Programming Workshop (2006)
54. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: Mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18(2), 300–336 (2009)
55. Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., Baldi, P.: Mining Concepts from Code with Probabilistic Topic Models. In: ASE 2007: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 461–464. ACM, New York (2007)
56. Long, F., Wang, X., Cai, Y.: API Hyperlinking via Structural Overlap. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM (2009)
57. Lukins, S., Kraft, N., Etzkorn, L.: Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In: WCRE 2008: Proceedings of the 15th Working Conference on Reverse Engineering, pp. 155–164 (2008)
58. Maarek, Y.S., Berry, D.M., Kaiser, G.E.: An Information Retrieval Approach for Automatically Constructing Software Libraries. IEEE Transactions on Software Engineering 17(8), 800–813 (1991)
59. Madani, N., Guerrouj, L., Penta, M.D., Gueheneuc, Y.G., Antoniol, G.: Recognizing Words from Source Code Identifiers using Speech Recognition Techniques. In: European Conference on Software Maintenance and Reengineering, CSMR (2010)

60. Malik, S.: Parsing Java Method Names for Improved Software Analysis. Tech. rep., University of Delaware (Senior Thesis) (2011)
61. Mani, I.: Automatic Summarization. John Benjamins (2001)
62. Manning, C., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT Press, Cambridge (1999)
63. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York (2008)
64. Marcus, A., Maletic, J.I.: Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, pp. 125–135 (2003)
65. Marcus, A., Poshyvanyk, D.: The Conceptual Cohesion of Classes. In: ICSM 2005: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), pp. 133–142. IEEE Computer Society, Washington, DC (2005)
66. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An Information Retrieval Approach to Concept Location in Source Code. In: WCRE 2004: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004), pp. 214–223 (2004)
67. Maskeri, G., Sarkar, S., Heafield, K.: Mining Business Topics in Source Code using Latent Dirichlet Allocation. In: ISEC 2008: Proceedings of the 1st India Software Engineering Conference, pp. 113–120 (2008)
68. McMillan, C., Poshyvanyk, D., Revelle, M.: Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery. In: TEFSE 2009: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp. 41–48. IEEE Computer Society, Washington, DC (2009)
69. Michail, A., Notkin, D.: Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. In: ICSE 1999: Proceedings of the 21st International Conference on Software Engineering, pp. 463–472. IEEE Computer Society Press, Los Alamitos (1999)
70. Miller, G.: WordNet: a lexical database for English. Communications of the ACM, pp. 39–41 (1995)
71. Ohba, M., Gondow, K.: Toward Mining "Concept Keywords" from Identifiers in Large Software Projects. In: MSR 2005: Proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–5 (2005)
72. Poshyvanyk, D., Marcus, A.: Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In: ICPC 2007: Proceedings of the 15th IEEE International Conference on Program Comprehension, pp. 37–48. IEEE Computer Society, Washington, DC (2007)
73. Poshyvanyk, D., Marcus, A., Dong, Y.: JIRiSS – an Eclipse Plug-in for Source Code Exploration. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), pp. 252–255 (2006)
74. Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., Liu, D.: Source Code Exploration with Google. In: ICSM 2006: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006), pp. 334–338 (2006)
75. Rao, S., Kak, A.: Retrieval from Software Libraries for Bug Localization: a Comparative Study of Generic and Composite Text Models. In: Proceeding of the 8th Working Conference on Mining Software Repositories, MSR 2011, pp. 43–52. ACM, New York (2011), http://doi.acm.org/10.1145/1985441.1985451
76. Ratanotayanon, S., Sim, S.E., Raycraft, D.J.: Cross-artifact Traceability using Lightweight Links. In: TEFSE 2009: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp. 57–64. IEEE Computer Society, Washington, DC (2009)

77. Reiter, E., Dale, R.: Building Natural Language Generation Systems. Cambridge University Press (2000)
78. Revelle, M., Dit, B., Poshyvanyk, D.: Using Data Fusion and Web Mining to Support Feature Location in Software. In: IEEE 18th International Conference on Program Comprehension, ICPC 2010 (2010)
79. Roach, D., Berghel, H., Talburt, J.R.: An Interactive Source Commenter for Prolog Programs. SIGDOC Asterisk J. Comput. Doc. 14(4), 141–145 (1990)
80. Robillard, M.P.: Automatic Generation of Suggestions for Program Investigation. In: ESEC/FSE-13: 10th European Software Engineering Conference Held Jointernationaly with 13th ACM SIGSOFT International Symp on Foundations of Software Engineering, pp. 11–20 (2005)
81. Robillard, M.P., Coelho, W.: How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Transactions on Software Engineering 30(12), 889–903 (2004)
82. Robillard, M.P., Murphy, G.C.: Concern Graphs: Finding and Describing Concerns using Structural Program Dependencies. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 406–416 (2002)
83. Robillard, M.P., Shepherd, D., Hill, E., Vijay-Shanker, K., Pollock, L.: An Empirical Study of the Concept Assignment Problem. Tech. Rep. SOCS-TR-2007.3, School of Computer Science, McGill University (2007), `http://www.cs.mcgill.ca/~martin/concerns/`
84. Robillard, P.N.: Schematic pseudocode for program constructs and its computer automation by SCHEMACODE. Commun. ACM 29(11), 1072–1089 (1986)
85. Runeson, P., Alexandersson, M., Nyholm, O.: Detection of Duplicate Defect Reports Using Natural Language Processing. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 499–510. IEEE Computer Society, Washington, DC (2007)
86. Saul, Z.M., Filkov, V., Devanbu, P., Bird, C.: Recommending Random Walks. In: ESEC-FSE 2007: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 15–24. ACM Press, New York (2007)
87. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In: AOSD 2007: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (2007)
88. Shepherd, D., Pollock, L., Vijay-Shanker, K.: Towards Supporting on-demand Virtual Remodularization using Program Graphs. In: AOSD 2006: Proceedings of the 5th International Conference on Aspect-Oriented Software Development, pp. 3–14 (2006)
89. Sinha, V., Karger, D., Miller, R.: Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In: Visual Languages and Human-Centric Computing, VL/HCC 2006 (2006)
90. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A Study of the Documentation Essential to Software Maintenance. In: 23rd Annual International Conference on Design of Communication, pp. 68–75. ACM (2005)
91. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards Automatically Generating Summary Comments for Java Methods. In: ASE 2010: Proceedings of the 25th IEEE International Conference on Automated Software Engineering (ASE 2010) (2010)

92. Sridhara, G., Hill, E., Pollock, L., Vijay-Shanker, K.: Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In: Proceedings of the 16th IEEE International Conference on Program Comprehension. IEEE (2008)
93. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Automatically Detecting and Describing High Level Actions within Methods. In: ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 101–110. ACM, New York (2011)
94. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Generating Parameter Comments and Integrating with Method Summaries. In: International Conference on Program Comprehension, ICPC 2011 (2011)
95. Sridharan, M., Fink, S., Bodik, R.: Thin Slicing. In: PLDI 2007: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (2007)
96. SUN: How to Write Doc Comments for the Javadoc Tool, `http://java.sun.com/j2se/javadoc/writingdoccomments/`
97. Takang, A.A., Grubb, P.A., Macredie, R.D.: The effects of comments and identifier names on program comprehensibility: an experimental investigation. J. Prog. Lang. 4(3), 143–167 (1996)
98. Tan, L., Yuan, D., Krishna, G., Zhou, Y.: /*iComment: Bugs or Bad Comments?*/. In: SOSP 2007: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 145–158. ACM, New York (2007)
99. Tan, L., Zhou, Y., Padioleau, Y.: aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 11–20. ACM, New York (2011)
100. Tarr, P., Ossher, H., Harrison, W., Stanley, M., Sutton, J.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: ICSE 1999: Proceedings of the 21st International Conference on Software Engineering, pp. 107–119. IEEE Computer Society Press, Los Alamitos (1999)
101. Tenny, T.: Program Readability: Procedures Versus Comments. IEEE Trans. Softw. Eng. 14(9), 1271–1279 (1988)
102. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages 3(3), 121–189 (1995)
103. Wang, X., Lai, G., Liu, C.: Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering. Electron. Notes Theor. Comput. Sci. 243, 121–137 (2009)
104. Warr, F.W., Robillard, M.P.: Suade: Topology-Based Searches for Software Investigation. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 780–783 (2007)
105. Woodfield, S.N., Dunsmore, H.E., Shen, V.Y.: The Effect of Modularization and Comments on Program Comprehension. In: Proceedings of the 5th International Conference on Software Engineering. IEEE Press (1981)
106. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A Brief Survey of Program Slicing. SIGSOFT Software Engineering Notes 30(2), 1–36 (2005)
107. Ying, A.T.T., Tarr, P.L.: Filtering out methods you wish you hadn't navigated. In: Eclipse 2007: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange, pp. 11–15. ACM, New York (2007)

# Text Retrieval Approaches for Concept Location in Source Code

Andrian Marcus and Sonia Haiduc

Department of Computer Science, Wayne State University, Detroit, MI, USA
{amarcus,sonja}@wayne.edu

**Abstract.** Concept location in source code is an essential activity during software change. It starts with a change request and results in a place in the source code where the change is to be implemented. As a program comprehension activity, it is also part of other software evolution tasks, such as, bug localization, recovery of traceability links between software artifacts, retrieving software components for reuse, etc. While concept location is primarily a human activity, tool support is necessary given the large amount of information encoded in source code. Many such tools rely on text retrieval techniques and help developers perform concept location much like document retrieval on web. This paper presents and discusses the applications of text retrieval to support concept location, in the context of software change.

**Keywords:** Concept location, concern location, feature location, information retrieval, software maintenance.

## 1 Introduction

Program comprehension is one of the most pervasive activities performed by developers and is particularly important and frequent during software maintenance and evolution, where it has been estimated to account for more than half of the time. *Recognition* and *location* are two fundamental comprehension processes [1] employed very often, when developers need to match their understanding of the problem domain to its representation in the source code and vice-versa. This problem is known in the literature as the *concept assignment* problem. Biggerstaff et al. [2] defined the concept assignment problem as "… discovering human oriented concepts and assigning them to their implementation instances within a program …". Concept assignment and its comprehension processes are performed often during software maintenance and evolution, in the context of various activities. Depending on the context, various instances of this problem have been defined and addressed.

*Concept location* in source code [3] is one such instance and it is the focus of this paper. Its name and definition originates from the *concept assignment* problem [2] and *feature location* in code [4]. It is related to other areas of research, such as, *fault localization*, *traceability link recovery between software artifacts*, etc., all of which are different instances of the *concept assignment* problem. What sets it apart from the related problems is that concept location is defined in the context of software change. Concept location starts with a change request and results in the starting point (in the source code) for the desired change [3]. Thus, when the first code location where changes need to be implemented is identified, concept location ends. Some researchers have adopted a more relaxed definition of concept location, considering it the task of determining all the locations in the code where changes need to be implemented. However, in the context of software change, we consider this as two different tasks, i.e., concept location, responsible for identifying the first change location, and impact analysis, which is responsible for finding the rest of the change locations starting from the one determined by concept location.

One particular instance of concept location is *feature location*, which deals with identifying the source code corresponding to a specific functionality of the software system that is accessible and observable by the user (i.e., a *feature*). In other words, the difference between concept and feature location is that feature location is focused on special concepts (i.e., features). All features are concepts, but not all concepts are features. For example, a *linked list* is a concept from the solution domain which may be implemented in the source code, yet it is not a specific feature of the system. Since the features of a software system are associated with its behavior, most feature location techniques rely on the execution of the software.

*Concern location* is a task very similar to concept and feature location. However, it deals with locating *concerns* in the code, i.e., anything that stakeholders of the software consider to be a conceptual unit, such as features, requirements, design idioms, or implementation mechanisms [5]. The difference between concept and concern location is in the context and the scope. Concept location is performed during software change (hence it has a specific input and output), whereas concern location is a context agnostic view of the activity. Also, concern location usually involves finding all the code elements participating in the implementation of a concern, rather than locating just one of them. In this paper we are interested in concept location and its variant feature location, as well as concern location.

While concept location might be performed manually in small systems, the task can become daunting in systems of medium or large size. The software engineering research community has recognized this problem and proposed a series of approaches and tools to help developers during this task. Most approaches (manual or otherwise) rely on searching and software analysis techniques. Different categories of approaches making use of different types of information have been defined, including those using static information about the structure of the system, those using dynamic information obtained by executing the program, etc. One specific type of information leveraged by many techniques is encoded in the textual data present in the source code. In fact, the textual data is used by many instances of the concept assignment problem, especially by those involving different types of software artifacts, such as, the traceability link recovery problem. Text is the common type of data between different software artifacts and also used to capture communications between stakeholders. The most recent and most advanced techniques used to extract and analyze text to support concept location (and its related activities) rely on Text Retrieval (TR) methods [6]. In most cases, concept location is redefined as a text retrieval problem.

In this paper we offer an overview of the TR-based concept location techniques and their evolution, discuss their limitations, and identify directions for future research in this area. We will not discuss other instances of the concept assignment problem that use TR a part of their solution, such as, the traceability link recovery problem. The focus of the paper is on concept location, feature location and concern location.

The rest of the paper is organized as follows. Section 2 presents a generic approach to concept location as a text retrieval problem. Section 3 presents the existing work in text retrieval-based concept location organized according to the steps of the generic approach, whereas Section 4 presents an overview of the tools which make use of text retrieval techniques to aid developers during concept location. Section 5 overviews the evaluation methods used in text retrieval-based concept location, and Section 6 discusses the current state of the research in this field and identifies directions for future research.

## 2 Concept Location as a Text Retrieval Problem

We define concept location in the context of software change, which occurs in the presence of a source code modification request. The software change process [7] starts with the modification request and ends with a set of correct changes to the existing code and addition of new code (see Figure 1). The software maintainer undertakes a set of activities to determine the parts of the software that need to be changed: concept location, impact analysis, change propagation, and refactoring. Concept location starts with the change request (input) and ends when the developer finds the first location in the source code where a change must be implemented (output). The code location can be a class definition, method, file, etc., depending on the needs of the developer. The next activity in the software change process, i.e., impact analysis starts from the result of concept location and identifies the rest of the source code locations that are affected by the change. Some researchers have considered impact analysis a part of concept location. In this paper, however, we make the distinction between the two and our focus is on concept location. We present the approaches which unify concept location and impact analysis as well, but specify in these cases the deviation from the definition we adopt.

During concept location, developers have an information need, which is finding one starting point in the code where changes need to be implemented. In order to satisfy this need, they search and navigate the source code. In this process, developers can use as a starting point the textual description of the change they need to perform, which often provides information that helps formulate a query for a search, choose a starting point for the navigation of the code, or choose a scenario for executing the program. In any case, the task of the developers is identifying the right fragment of code from the large amount of possibilities available in the source code of a system.

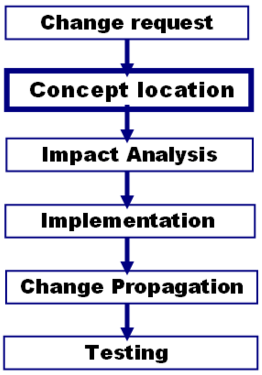


**Fig. 1.** Simplified view of the software change process (adapted from [7]). Concept location starts with the change requests and produces the input for impact analysis.

This situation is somewhat similar to a user searching and navigating the web in order to identify among the vast amount of web pages available the ones that satisfy her current information need. Both concept location and web search can be considered instances of a classic retrieval problem: given a large collection of available documents and an information need (usually formulated as a search query), determine those documents from the collection that satisfy the information need [8]. Moreover, considering that source code is mostly text, concept location can be seen as a specific text retrieval problem, where the documents represent fragments of source code from a software system.

For the reminder of the paper, we will consider concept location as a text retrieval problem and will discuss the existing techniques to address it from this perspective.

In the next two subsections we define the text retrieval terminology that will be used for the rest of the paper (not necessarily familiar for the software engineering readers) and present the generic concept location approach based on text retrieval, adapted from the text retrieval for natural language documents.

## 2.1 Terminology

This section introduces some of the concepts specific to the field of text retrieval, which we will be using in the reminder of the paper. Each of the definitions includes examples for concept location.

**Document**

A *document* represents the specific unit of retrieval, i.e., what a user would get as a result to a search. In the case of written natural language or the web, a document can be a news article, a book, a chapter, a web page, etc. In the case of concept location, the documents refer to source code elements and can be source code files, classes, methods, functions, lines of code, etc.

**Term**
A *term* is a unit of discourse considered by the text retrieval technique, used to express meaning in a document. In natural language, this is usually equivalent to a word or its lexical root. In the case of concept location in source code, however, a term need not be a word, as often identifiers contain units of discourse that are not dictionary words. In source code, the terms in the documents are extracted from the identifiers and comments present in the source code. For example, the identifier "getName" represents the name of a method, and contains the term "getName" or the terms "get" and "name", in case the identifiers are split according to common naming conventions. The terms will be associated with the document from which they were extracted, in this case the method getName.

**Corpus**
A collection of documents is called a *corpus*. In the case of concept location in source code it can be, for example, the set of all classes in the system, or the set of all methods, etc.

**Indexing**
The text retrieval techniques transform the documents into an intermediary, mathematical representation which is then used for fast accessing and retrieval. The process of transforming the documents to this internal representation (i.e., *index*) is called *indexing*. This is the core action performed by the TR-based search engines. Different search engines use different TR models to define the indices.

**Corpus normalization**
Usually, before a corpus is indexed using a TR technique, it has to be transformed, that is, normalized in order to improve the efficacy of the retrieval. This normalization is comprised of several steps, each of them optional. In the case of concept location, the steps are *identifier splitting*, where identifiers are split into their constituent terms based on common identifier naming rules (i.e., "setValue" and "set_value" will be split to the terms "set" and "value"), *filtering*, where terms that do not contain meaningful information are removed, and *stemming*, where the terms are reduced to their lexical root (e.g., "name", "names", "named", "naming" are all reduced to "name").

**Query**
A *query* is a word or set of words usually written by a user as input for a retrieval operation and which represent an expression of the information need of the user. The text retrieval technique analyzes the index for documents that are relevant to the input query and retrieves them. If the corpus was normalized before indexing, the query will be subjected to the same normalization steps as the corpus before being run by the text retrieval technique. A query for concept location in source code has the same form, i.e., a natural language phrase of one or more words, as a query used to search the internet using online search engines like Google.

**Relevance**
A document is *relevant* to the information need of a user if its contents satisfy completely or partially the information need. It is often referred to as *relevance to the query*, as it is presumed that the user expresses her information need well in the input query. However, this might not always be the case, as sometimes users can have a hard time formulating queries due to lack of information. This is a problem that the Information Retrieval field recognizes and tries to address and is known as *the search paradigm.* In this paper, we assume that the user expresses the information need well in the query, thus we use the terms *relevance to query* and *relevance to the information need of the user* interchangeably.

**Target method/class**
A class or a method is referred to as *target* when, during a software change task, it is one of the classes or methods affected by the change. For example, if a class C contains a bug and fixing the bug is the current change task, in order for the change to be implemented class C needs to be modified. Thus, class C is a target class.

### 2.2 The Generic Approach for Text Retrieval-Based Concept Location

The generic text retrieval approach to concept location in source code is based on the approach defined by the Information Retrieval field for retrieving natural language documents (a.k.a. text retrieval). All text retrieval-based approaches to concept location are based on this generic model and they differ in how the various steps are instantiated. The generic approach is composed of five major steps, described below (see Figure 2). Program comprehension is an activity that involves the developer and tools that support her. As a comprehension activity, concept location also relies on the human. In consequence, some of these steps of the generic concept location process are done by specialized tools, while others need to be performed by the developer, who is responsible for final decisions and judgments. In the description of each step, we mention whether the step is performed automatically or not.

```
1. (A)Create and normalize corpus
   a. (U/A)Choose granularity level
   b. (A)Extract corpus from source code
   c. (A)Split identifiers (optional)
   d. (A)Filter stop words (optional)
   e. (A)Stem terms (optional)
2. (A)Index corpus
3. (U/A)Formulate and normalize query
   a. (U/A)Formulate query
   b. (A)Split identifiers (optional)
   c. (A)Filter stop words (optional)
   d. (A)Stem query (optional)
4. (A)Retrieve results and present them
5. (U)Examine results
```

**Fig. 2.** The generic approach for text retrieval-based concept location. (A) indicates that the step is performed automatically, whereas (U) indicates that it requires the involvement of a user.

1. *Corpus creation and normalization*

Text retrieval techniques are designed to work with a collection of documents written in natural language. While source code contains natural language, it contains much more information besides text, which is of no use to text retrieval techniques. The first step of the approach deals with transforming the source code into a format appropriate for text retrieval techniques. This step is performed using specialized tools for extracting and normalizing the corpus of a software system.

First of all, the source code of a system needs to be divided into documents. In the case of natural language this division is often straight-forward and is usually done without any input from the user, e.g., each web page on the web is a separate document, each book in a library is a separate document, etc. This, however, is not true for source code. The developer needs to make a conscious decision about what should be considered a document, i.e., what will be the granularity of the retrieval (e.g., a line of code, a function, a file). Also, the range of choices can depend on the programming paradigm and on the structure of each software system. For example, in the case of object-oriented code, two obvious choices are classes and methods. However, these are not applicable in the case of procedural software systems, where a file or a function would be more appropriate.

Once the granularity level has been determined, the documents are identified and extracted from the source code. In particular, the identifiers and comments are extracted from each source code document, as they represent the meaning-baring elements of the code, as well as additional strings, such as, constant values. This step usually requires programming language-specific parsers.

After the extraction, a few optional steps (i.e., corpus normalization) can be performed before the documents are indexed by the text retrieval techniques. First, the identifiers can be split into their constituent terms according to common naming conventions. For example, "setValue", "set_value", "SETvalue", etc. would be all split to "set" and "value". This requires also a decision about keeping the original form of the identifiers or not. Keeping the original identifiers along with the words resulted after splitting can advantage any identifiers that the developers might include in the queries. On the other hand, when no identifiers are included in the queries keeping the original form might negatively impact the results.

The extracted documents contain some source code-specific terms, like programming keywords, which do not contribute meaning to the documents. Also, the documents can contain common English terms like conjunctions, prepositions, common adverbs, etc., which are too widespread to be specific to any document. The documents can be subjected to *filtering* algorithms in order to eliminate these types of terms (known as stop words).

Sometimes different variations of the same term are used in the source code and in the query, while all referring to the same concept. For example, "open", "opened", "opening", "opens" all refer to the concept "open", even though the exact words used for expressing it are different. However, from a semantic point of view, the meanings of the different variants are very similar. In order to account for this when retrieving source code documents, *stemming* can be applied, which reduces all the lexical variations of a word to the root form. How this operation is performed and what the end result is depends on the chosen stemmer. For example, using the Porter stemmer [9], all the words in the example above would be stemmed to "open".

The result after applying all the steps described above is the *normalized corpus* of the software system, i.e., a collection of source code documents ready for indexing. This corpus represents the input for the text retrieval engine.

2. *Corpus indexing*

In this step, a mathematical representation of the corpus is built, which is stored by the text retrieval engine in a quickly accessible format called *index*. Each document in the source code corpus (i.e., each method, class, etc.) has a corresponding entry in the index. This step is different for every text retrieval technique and is often what sets the various text retrieval techniques apart. It is performed by specialized tools.

The terms in a document can be assigned a higher or lower importance, or *weight*, based on two criteria: how well they describe the current document (*local weight*) and how well they represent the entire corpus (*global weight*).

In the case of highly dynamic document collections, like the web, indexing is done continuously in the background, and is transparent to the user. In the case when the documents in the collection are rarely added, deleted, or modified, which is often the case with source code, this step can be performed only once, before the first retrieval task is performed.

3. *Query formulation and normalization*

This step is usually performed by the developer, but it can be also performed automatically in certain instances. It involves defining a set of words that describe the information need of the developer (i.e., the concepts to be located), which constitute the query. The developers can use the information contained in the description of the task at hand (e.g., a bug report, a new feature request, etc.) as a starting point for formulating the query. At the same time, developers can use previous knowledge, the system documentation or any other sources of information that can help them formulate queries. After the query is formulated, it is subjected to the same normalization applied to the corpus (i.e., filtering, stemming, etc.). Once normalized, the query is run by the text retrieval technique.

The query formulation is a very important part of the text-retrieval based concept location, as the success of this type of approaches is highly dependent on the given query. If the initial query did not lead to satisfactory results, developers can always return to this step and reformulate the query, a process which can benefit from tool support.

4. *Retrieval and ranked list presentation*

Once the query formulated by the developer is run, the text retrieval technique computes semantic similarities between the query and every document present in the corpus. Then it retrieves a ranked list of results, which contains all the documents in the corpus in descending order of their similarity to the query and displays the list to the developer. Thus, the documents that match the query the best will be placed first in the result list. By showing a ranked list of results, text retrieval approaches overcome one of the major limitations of the string-matching techniques previously used for textual search (i.e., *grep*), which presents the results of a search in no particular order to the user. When using text retrieval approaches for concept location, the developer can start investigating the results at the top of the ranked list and move down, as the likelihood that documents in the result list are relevant to the query decreases in that direction.

There are several similarity measures that can be used when comparing the contents of the query with documents in the corpus. However, the similarity measures that can be used in a particular case depend on the type of text retrieval technique used. The choice of similarity must be done with care, as it can have an impact on the results. All the actions described in this step are performed entirely automatic.

5. *Results examination*

After the list of documents has been retrieved, it is presented to the developer, who can start examining the ranked list of source code documents. Usually, the recommended order of examination is from the top of the list to the bottom, as the documents that are most likely to be relevant to the query according to the text retrieval technique are located at the top of the list. The examination of the results is performed entirely by the developer.

For every source code document examined, a decision is required whether the document will be changed or not. If it will be changed, then concept location succeeded and it ends. Concept location is usually followed by impact analysis, in order to determine what other parts of the code have to change. Some approaches merge these two activities conceptually. If the document is not going to be changed and the developer accumulated new knowledge from the investigated documents, she can formulate a better query (e.g., narrow down the search criteria) by repeating step 3. Otherwise, the next document in the list should be examined.

## 3 Text Retrieval-Based Approaches for Concept Location

The text retrieval-based approaches proposed in the concept location field have instantiated the above steps in various ways. Most research investigated how changes to particular steps in the process described above impact the results of concept location. This section presents an overview of the research in the field of text retrieval-based concept location, organized by the steps of the basic approach and discusses how and why the approaches instantiated these steps. As our focus is on describing the text retrieval approaches, we focus only on the steps which are performed at least in part automatically by the approach. Research in concept location is also concerned with the activities that the developers undertake directly and the underlying cognitive processes, but such research is outside the scope of this paper.

## 3.1 Corpus Creation and Normalization

The way the corpus is created can influence the results of a text retrieval – based concept location approach independent of the actual text retrieval technique used. Some of the research efforts in the field have been concerned with investigating the impact of the corpus creation on the results of text retrieval-based concept location.

The granularity of the documents can influence greatly the results of concept location, as the frequency and term co-occurrence information change depending on how the documents are chosen. Researchers have used various document granularities for concept location and its variants. However, there seems to be a preference for smaller granularity levels, such as, methods [10-29] in the case of object oriented systems and functions [30-33] for other programming paradigms. This preference can be explained by several factors. First, methods and functions locate the concepts and features in more detail than classes and files. Second, most of the approaches make use of structural or dynamic information in addition to the text retrieval techniques, which often use information that is at the level of granularity of methods or functions. For example, call graphs and their variants are often used for acquiring structural information about the system, and execution traces are used for the dynamic component. The approaches using the combination of text retrieval with other types of information about the software system are explained in detail in Section 3.5. Third, as more and more approaches made use of the method level granularity, researchers had little choice in the cases when a comparison to previous approaches was desired.

Even though less widespread, file level granularity [34, 35] and class level granularity [36] were used in some work in the field making use of text retrieval.

The success of retrieval depends on many factors, but one of the basic conditions that need to be satisfied is that the vocabulary of the source code corpus needs to be more or less the same as the vocabulary available to developers for formulating queries. As developers usually express their information need using dictionary words, it is important that the source code corpus contains also such words. Identifiers, however, are often composed of several concatenated dictionary words. It is important therefore that the corpus normalization includes a step where the identifiers are decomposed into their constituent words. This step is usually performed automatically, as manual identifier splitting is unfeasible due to the high number of identifiers in a software system.

Identifier splitting is usually employed when using text retrieval on source code and the effect of different identifier splitting approaches on text retrieval-based feature location has been studied in [11]. The paper investigates the benefits of using accurate techniques compared to more primitive splitting techniques, with the goal of determining if using accurate splitting can improve the results of text retrieval-based feature location significantly. In order to achieve this, the results of text retrieval-based feature location when using camel case, the Samurai [37] and manual splitting are compared in two open source systems. Two approaches making use of text retrieval are investigated, one implementing the generic approach described in Section 2.2, and one combining text retrieval with dynamic information. The results suggest that feature location techniques using text retrieval can benefit from better splitting algorithms in some cases, and that the improvement while using manual splitting over the automatic approaches is statistically significant in these cases. The results for feature location using the combination of text retrieval and dynamic analysis, however, do not show any improvement while using manual splitting, indicating that any splitting technique will suffice if execution data is available.

Filtering [10, 11, 13, 16-18, 20, 22-24, 38, 39] and stemming [10, 11, 16-18, 20, 22-24, 28] are the last two steps in the normalization step and they are now standard steps in corpus normalization.

## 3.2 Corpus Indexing

The internal representation of documents during text retrieval-based concept location depends on the particular model used. Different text retrieval models perform in different ways and researchers have employed various such models in the attempt to find the one that performs the best. Remember that text retrieval techniques were developed with natural language in mind. While a lot of the text in the source code is natural language, its structure is quite unique; hence the performance of the text retrieval techniques may not be the same as on natural language corpora. This section focuses on the choices of text retrieval models and engines used in supporting concept location. For more details about each of the text retrieval techniques mentioned in this section, please refer to [6, 8, 40].

Cubranic et al. [34, 41] applied the *Vector Space Model (VSM)* [42] for concern location. VSM represents documents in an n-dimensional vector space, where n represents the number of unique terms in the source code corpus. Each document has associated an n-dimensional vector, which contains information about the relevance of each of the terms in the corpus for the document. Several term weights can be used to determine this importance, and they are usually a combination of local weight, which conveys information about the frequency of a term in the document, and a global weight, which indicates how relevant the term is to all the documents in the corpus.

The approach proposed in [41] makes use of VSM with log-entropy term weights to find concepts not only in artifacts found in the source code of the system, but also in online documentation, bug reports, and communications between developers. The implemented tool, Hipikat forms a group memory from the source code repositories, issue trackers, communication channels, and web documents of a project and then uses VSM to suggest related artifacts to one selected artifact or a user query by using the cosine similarity between their term vectors. Case studies have been performed on AVID24 [41] and on Eclipse [34, 41], to evaluate if Hipikat helps developers perform maintenance tasks on unfamiliar systems and the results were promising. While Hipikat was not presented in the context of software change specifically, it can be used to support concept location and other similar tasks.

VSM was also used by Zhao et al. [30, 32] in SNIAFL, a static non-interactive approach to feature location which uses text retrieval in conjunction with a branch-reserving call graph. [31] introduces another approach based on combining VSM with static information, as well as [17]. Details about the combination between text retrieval and other sources of information are described in Section 3.5.

In [25] the lexical similarity based on using the cosine similarity between two documents in the VSM is combined with a technique based on using already available mappings between features and program elements in order to determine the structural similarity between a new feature and the program elements.

An approach based on VSM, but using a modified cosine similarity measure, meant to perform better than the original cosine, was implemented in the open source search library Lucene [43]. This approach was used in the implementation of the FLAT^3 tool [44], where it can be combined also with dynamic information. The same approach was used and modified in [10], combining it with a query expansion mechanism, and in [26], where it has been combined with historical and static information.

VSM, even though relatively successful, has some limitations due to the fact that the similarity between a query and a document is correlated with the number of terms they share. While logical, this approach requires that different documents referring to the same concept use the same words in order to be recognized as similar. However, this is not always the case, as a document and a user generated query could use different terms when referring to the same concept or could use the same term, but referring to different concepts. These problems are termed synonymy and polysemy respectively. This is of potentially great detriment to novice software developers or developers encountering an unfamiliar system for the first time, as they may not possess a vocabulary in sync with the one used in the source code.

To deal with part of these limitations, *Latent Semantic Indexing* (*LSI*) [45], a more advanced text retrieval technique, was proposed as a substitute for VSM in the context of concept location. In LSI, the initial vector space formed from the corpus is projected to a lower dimensionality, in which dimensions do not represent terms, but rather latent concepts. LSI, therefore, does not match words exactly, but rather by comparing the vectors in the reduced semantic space it is able to capture their latent meaning and match words that are similar in terms of their usage.

LSI was first applied to concept location by Marcus et al. [33, 36], and was used to map feature descriptions expressed in natural language to the methods in the code that implement them. In [33] the LSI-based approach was applied on a change request in Mosaic, a medium sized software system written in C. A set of developer-formulated queries were used, as well as queries semi-automatically generated using LSI. The results indicated that LSI performs better than string matching (i.e., grep) and dependency graph navigation for concept location in Mosaic. However, the paper also underlined the fact that none of the concept location approaches leads to perfect results and suggested that a combination of approaches might be a good direction to follow. These conclusions were strengthened by the results obtained in [36], where the approach was used for locating concepts in two object-oriented systems.

Cubranic et al. updated the VSM-based approach they proposed in [41] to one using LSI in [35]. LSI was also used in combination with other techniques and sources of information for performing feature location. In [12, 13], it was used in combination with probabilistic ranking and dynamic information for feature location, in [15] is used to order the methods executed in a single execution scenario, and in [14] is used in combination with Formal Concept Analysis in order to present to the user a clustered list of results, formed by clustering the set of documents most similar to a query. In [20], the approach in [13] was extended and complemented with static information and compared to each of the combined techniques considered individually. LSI was also used in combination with static call graph navigation for feature location in [29] and with dynamic information and a search-based optimization approach in [18]. Other papers where one of the presented approaches using LSI was applied are [24], [46], and [17].

Even though capable of dealing with synonymy successfully, LSI is not able to model and deal with polysemy. Moreover, the non-probabilistic nature of the method raises issues with respect to the principles on which the approach is based [47].

One approach which overcomes these issues is *Language Models (LM)* [48]. This is a probabilistic approach that builds a model of the language used in each of the documents in the corpus, and then computes the probability of the query to be generated by each of these models and orders the documents in the result list based on this probability.

Cleary and Exton introduced the use of LM for concept location in their approach called *cognitive assignment* [16, 19]. The same authors had previously used a Bayes classifier [49] in order to categorize the source code documents in pertaining to a feature or not. In [16, 19], the authors extended the original LM approach by considering indirect correspondences between query and document terms so that relevant source code can be retrieved even if it does not contain any of the query terms. To do that, queries are expanded by analyzing term relationships from both source code and non-source code artifacts like as bug reports, mailing lists, external documentation, etc. A case study was conducted on Eclipse in which cognitive assignment was compared to other text retrieval techniques, such as language modeling, dependency language model, vector space model, and LSI. The results indicate that cognitive assignment matches the performance of the other text retrieval techniques and in some cases it outperforms it.

Language models, even though capable of dealing with polysemy, are not able of dealing efficiently with synonymy, as they are not designed to capture semantic relations between words based on term co-occurrences. The semantic relations between words are learned from (synthetically created) query-document pairs and are not directly based on co-occurrences within the document collection [47].

*Latent Dirichlet Allocation (LDA)* [50] overcomes these limitations, as it is able to represent both synonymy and polysemy. LDA is a probabilistic and fully generative topic model that is used to extract the latent, or hidden, topics present in a collection of documents and to model each document as a finite mixture over the set of topics. Each topic in this set is a probability distribution over the set of terms that make up the vocabulary of the document collection. In LDA, similarity between a document and a query Q is computed as the conditional probability of the query given the document.

LDA has also another advantage, as it is able to produce immediately interpretable results, which is not true in the case of LSI and LM, where it is difficult to interpret why a document is similar to the query. In the results returned by LDA, however, the most likely terms for each topic can be examined to determine the likely meaning of the topic. Though LSI has been used to extract and label topics [51], it cannot do so directly or in isolation.

LDA has first been applied to concept location by Lukins et al. [22, 23]. A topic model of the source code corpus is first built, which can be then queried by the user. The new approach was applied for bug location in three software systems, i.e., Rhino, Mozilla, and Eclipse, and compared to the LSI basic approach for concept location. The results indicated that LDA can be successfully applied to concept location in software and that LDA is more effective in locating bugs than LSI in the three systems analyzed.

In order to take advantage of the relationships between documents and their strength, Revelle et al. [28] introduced the use of web mining techniques like PageRank and the HITS algorithms, and their combination with information retrieval and dynamic information for feature locationAs of today, there is insufficient empirical or theoretical work to clearly show which text retrieval model or document representation is best suited for concept location in source code. Benchmarking efforts [52] will hopefully provide more insight into this issue.

## 3.3 Query Formulation and Normalization

One of the major limitations of text-retrieval based approaches to concept location is the fact that they are highly sensitive to the input query, and studies have shown that effective natural language queries can be as important as the retrieval algorithm itself [53]. In order for the text retrieval techniques to succeed, the query needs to contain terms that describe the information need of the developer clearly and precisely, using a vocabulary in concordance with the one used in the source code. This is a difficult problem, as developers performing concept location are often not familiar with the source code nor with the vocabulary used in its identifiers and comments. Also, there are no guidelines on what a query should contain. Moreover, different developers formulate queries quite differently [33] and the likelihood of two people choosing the same keyword for a familiar concept is only 10-15% [54]. Some developers are able to express the problem better than others and, in fact, some might have troubles expressing the information need in a way that leads them to the right code location even after several reformulations of the query [55]. Under these circumstances, researchers have focused on helping developers formulate queries offering suggestions that developers can follow, or introducing methods for automatic query formulation that do not need human intervention. At the same time, it is important to know when a query, formulated either manually or automatically, does not lead to the wanted results and needs to be reformulated. This might be difficult in some cases for developers, so approaches have been proposed to address this problem automatically.

A series of papers have addressed the problem of query formulation and refinement in the context of text retrieval-based concept location. Some efforts have focused on offering developers alternatives and suggestions when formulating queries, as opposed to reformulating the query automatically for them. Poshyvanyk et al.

introduce such an approach in their tool JIRiSS [39], based on using LSI to search the source code of a software system for classes and methods relevant to a query. The query term recommendation component of JIRiSS generates the vocabulary of the software system and the frequency of occurrences of every word in the vocabulary. The goal is to help the developers get familiar with the terms used in the system so they can choose words to include in the query. This feature is meant to be especially useful for users unfamiliar with the system and its naming conventions.

The approaches based on suggestions still require the developers to go through the suggestions and make their best guess as to which of the terms should be included in the query. The following approaches, however, not only suggest words for query reformulation, but actually refine the query without human intervention.

Marcus et al. [33] used LSI to automatically expand and reformulate queries by extracting words and identifiers from the source code of the system that are related to a given query word or phrase. The automatic queries used in the presented case study were created starting from the initial one-word query by adding the top n most similar terms to the query, and keeping or removing the initial word. The results indicated that the automatic technique can generate queries that are comparable in performance to the user-generated queries, while at the same time requiring no domain knowledge to formulate them.

An automatic query reformulation approach for concept location based on using relevance feedback in combination with LSI was introduced in [10]. In this approach, the developer does not need to formulate or reformulate the query, but rather just needs to provide feedback about the top n results provided by LSI, indicating their relevancy to the task at hand. In this context, relevance refers to the fact that a document is related conceptually to the task, but does not change. The text retrieval begins with the description of the task as the initial query. Then, after the developer provides feedback about the results of the retrieval, this feedback is used to automatically reformulate the query. More specifically, terms from the documents indicated as relevant are added to the query and terms from the documents marked as irrelevant are deleted from the query. This iterative process continues until the developer finds the documents to be changed, or she decides to use other techniques to locate them. The main advantage of this approach is the fact that the developer feedback is taken into account, while having the developer focus on analyzing what she knows best, i.e., source code. Also, the results of the case study performed on three software systems indicates that the text retrieval technique incorporating relevance feedback is more efficient than using text retrieval alone.

Hayashi et al. [21] build on the idea introduced in [10] and propose an interactive environment for comprehending feature implementations. In this approach, developers understand feature implementations by performing feature location several times. Each time feature location is performed with a different query, obtained automatically by expanding the query using synonym lists and by using relevance feedback provided in the previous steps.

A query expansion technique based on leveraging term relationships in a text retrieval approach based on LM was proposed by Cleary et al. [16, 19]. The term relationships from both source code and non-source code artifacts are captured in a model based on the notion of information flow between terms and meant to produce information-based inferences which correlate with inferences made by humans [56]. Using the measure of the degree of information flow between concepts and terms and given a concept or set of concepts in the form of a query, one can compute information flow values for each term in the vocabulary. Then, by imposing a threshold or by selecting a set of the top ranked terms a set of terms related by information flow to the terms in the query can be defined. These terms are then added to the query.

Revelle et al. [27] introduced a way by which queries are automatically reformulated from the identifiers in a method known to be relevant to a feature. The results show that this type of queries is just as effective as the queries formulated by a human.

Determining when a query should be reformulated because it does not lead to the wanted results is a hard problem, and often requires time and analyzing the results of the retrieval in detail. This is even more difficult for developers not familiar with the source code of a software system, as they do not know beforehand what parts of the code they should find to satisfy their information need. This is often referred to as the *search paradox*.

In order to ease the task of determining when a query can lead to good results or not and speed up the reformulation of poor queries, Haiduc and Marcus [57] proposed using a series of approaches from the field of Information Retrieval. The approaches are divided into *pre-retrieval* and *post-retrieval*. Pre-retrieval approaches predict the performance of a query before the retrieval stage is reached and are, thus, independent of the ranked list of results. They base their predictions only on query terms, collection statistics and external sources such as dictionaries. In contrast, post-retrieval methods can additionally analyze the retrieval results and make a prediction based on the properties of the documents found in the result set.

Finding good word suggestions and automatically reformulating the query are one approach to deal with the dependence of the text retrieval techniques on the query. However, this problem can be also approached by combining the text retrieval techniques with other sources of information, like structural or dynamic information about the software. By making use of more sources of information and combining them in order to obtain the final set of results, text retrieval-based approaches are less sensible to the natural language query. Researchers have applied this approach in various forms, presented in detail in section 3.5.

## 3.4 Retrieval and Ranked List Presentation

The results of concept location using text retrieval techniques are usually presented to the user as a ranked list where documents are ordered according to their similarity to the query, with the most similar documents located at the top of the list. However, sometimes the target documents might not be located at the top of the ranked list, in which case the list needs to be manipulated in order to improve the initial results returned by the text retrieval technique. Research efforts have been made in this direction and this section describes them.

Poshyvanyk and Marcus [14] applied Formal Concept Analysis (FCA) on the ranked list of results in order to organize the top returned documents in hierarchical clusters based on their semantics. FCA takes as input a matrix specifying objects and their associated attributes and produces as output clusters, referred to as concepts, of the given objects based on their shared attributes. These concepts can be organized hierarchically in a concept lattice. In this case, the objects considered were methods and the attributes were the words that appear in the source code of the methods. The top k words appearing in the first n methods in the ranked result list returned by LSI were used to construct the input matrix required by FCA and to create the concept lattice. Nodes in the lattice have associated attributes (terms) and objects (methods), and programmers can focus on the nodes with attributes similar to their query to find feature-relevant methods. The new concept location technique based on FCA was compared against using LSI alone on two maintenance tasks in Eclipse. The results indicated that using the concept lattice-based approach developers could locate a concept in code by analyzing fewer methods.

In [17], the authors use clustering in order to group the source code documents before retrieval, such that the text retrieval technique returns clusters instead of individual documents. This approach allows relevant documents that might have been placed on a high position in the original, unclustered result list to be placed on the top of the list of clusters, as long as the other documents in the cluster are similar to the query. By modifying the ranking of the documents this way, users can get to relevant documents in fewer steps. The new technique was compared with the baseline TR-based approach using data associated with changes in response to 198 bug reports in the three software systems. The results indicate that the new technique outperforms the baseline in average.

In order to take advantage of the relationships between documents and their strength, Revelle et al. [28] introduced the use of web mining techniques like Page-Rank and the As observed, the main thread of research on ranking the list of retrieved documents is focused on filtering the list of results or identifying relationships between documents that help group them. One aspect ignored largely by current research is the presentation of the results. Exception is the work in [14], where the results are presented as a lattice rather than a list of ranked documents. More research is needed in this area, maybe based on software visualization.

## 3.5 Combining TR with Other Sources of Information

Text retrieval-based concept location techniques take advantage of only one type of information found in the source code of a system, i.e., the text. Source code contains, however, much more information than just text, which can reveal code relationships that text cannot capture. Structural information reveals relationships between software components based on the information flow and dependencies between them, whereas dynamic information reveals relations based on the behavior of the software. At the same time, repositories like source code version control systems, issue trackers, forums, etc. also contain information about a software system that can prove relevant to current change tasks performed by developers. Using these sources of information in addition to text retrieval can improve concept location.

At the same time, using other sources of information can overcome some of the limitations that text retrieval techniques exhibit. For example, the sensibility to the natural language query formulated by the user can be overcome by combining the expertise of the text retrieval with that of other approaches.

Researchers have recognized the benefits of using additional source of information which can complement the textual information leveraged by text retrieval techniques and have proposed approaches that make use of these sources of information. This section describes these approaches.

**Text retrieval combined with structural static analysis**

Combining structural and textual information for concept location was an obvious step that researchers have made, as either textual analysis can be used to reduce the set of results that the static analysis produces or static analysis can be used to find additional relevant documents starting from the set of top results returned by the textual analysis. Several researchers have proposed approaches based on these ideas.

Zhao et al. [32] introduced a static, non-interactive approach to feature location, which uses text retrieval in combination with a branch-reserving call graph (BRCG), which is an expanded version of a call graph with branch information. Text retrieval, i.e., VSM is used to retrieve an initial set of methods specific to the feature. Then, a gap threshold technique is used to find the largest difference between the similarities of consecutive methods in the ranked list of results. The methods above this gap are considered to be the initial elements specific to the feature. Starting from these, additional relevant methods are found by pruning the BRCG to remove branches that are not in the initial set. Note that the authors of this paper considered concept location and impact analysis as one single step, thus determining all the target methods during the analysis. The relevance of branches that are included in the initial set is propagated through the dependence relations in the graph, generating a static pseudo-execution trace. In the two case studies performed, the new technique achieved better precision and recall than both a pure text retrieval approach and a purely dynamic approach.

Shao et al. [29] combines text retrieval and static control flow information for feature location. First, LSI is used to rank all the methods in a software system by their relevance to a query. Then, for each method in the ranked list, a call graph is constructed and inspected to assign a call graph score to each method based on the number of direct neighbors of the method that also appear in the ranked list returned by LSI. The cosine similarity of the method, as returned by LSI, and its call graph score are then combined using an affine transformation, and a new ranked list is produced.

Ahn et al. [31] use a new approach which builds and analyzes a weighted call graph using the similarity values obtained from VSM in order to locate features specified in the manual of a software system. First, the approach uses VSM to recover textual relationships between the description of the feature and the source code elements. Further, the approach uses a weighted call graph to select core functions among the retrieved functions. Then, by analyzing the weighted call graph, the approach determines the final set of relevant functions.

Ratanotayanon et al. [26] investigated the effect of integrating more data sources with text retrieval for concept location, and one of the data sources considered was a static dependency graph. The findings of the case studies performed revealed that it is not always better to have more diverse data, in particular adding the static dependency graph data to changesets increased recall, but drove down precision.

Peng et al. [25] measures the relevance between a feature and a program element based on the textual and structural similarity of the program element to the feature. The new approach uses an iterative process to propagate the knowledge of the already established mappings between a feature and a program element to the neighboring features and program elements in the call graph. The underlying intuition is that the more feature-element mappings are recovered, the more likely it becomes that the approach may recover further related feature-element mappings.

Scaniello and Marcus [17] used the cosine similarity as returned by VSM and structural dependencies between the methods in the corpus in order to cluster the corpus before retrieval. In consequence, the user retrieves clusters instead of individual documents. The evaluation showed that the new approach performs better than the baseline.

**Text retrieval combined with dynamic analysis**

Dynamic analysis and textual analysis can be combined in different ways for concept location. One way to combine them is to use dynamic analysis to filter the program elements for textual analysis instead of ranking all the program elements in a software system. Also, both analyses can rank program elements by their relevance to a feature, so another direction is to combine the rankings produced by the two techniques.

Poshyvanyk et al. [13] introduced the Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) approach, which combines two existent concept location techniques, one based on LSI text retrieval [33] and the other on dynamic analysis, i.e., Scenario based Probabilistic Ranking (SPR), introduced first by Antoniol et al. [58]. Both approaches rank program elements according to their relevance to a given feature. Their rankings are combined through an affine transformation to produce the final results. The weight given to SPR and LSI can be varied to reflect the amount of confidence that should be assigned to each. The case study performed on two large software systems indicate that the new, combined approach outperforms the two techniques on which it is based.

The Single Trace and Information Retrieval approach, introduced in [15] is a feature location technique that applies text retrieval on the execution information collected from exercising a single scenario relevant to the feature. The technique first requires the execution of a scenario relevant to the feature by the developer, and captures the trace of that execution. Then it uses as input a query from the developer and applies LSI to produce a list of methods from the execution trace which are ranked based on the similarity with the query. This approach ranks only the methods that appear in the execution trace, as opposed to ranking all the methods from the system, as done by previous approaches. The results of the case studies performed on jEdit and Eclipse, which compared the new approach to LSI, SPR and PROMESIR indicate that the technique not only reduces dramatically the search space, but also leads to better results in general. [11] also used this approach when studying the impact of different identifier splitting algorithms on the results of concept location. An Eclipse plug-in that supports this approach, FLAT3 [44], was also developed.

Asadi et al. [Asadi'10] introduce an approach which identifies concepts in execution traces by determining cohesive and decoupled trace fragments. The approach relies on text retrieval, dynamic information, trace compression techniques, and a genetic algorithm. First, scenarios which exercise the features of interest are run and their execution traces are captured. The traces are then modified such that the functions not relevant to the features are removed (i.e., utility functions, cross-cutting concerns) and repeating method sequences are compressed. LSI is then used to compute the Conceptual Cohesion of classes, according to the metric defined by Marcus et al. [59]. Using the conceptual cohesion as a fitness function, a genetic algorithm is then applied in order to segment the execution traces into conceptually-cohesive segments related to the feature being exercised. The empirical study performed on two Java systems indicates that the approach is able to locate concepts with high precision.

**Text retrieval combined with structural and dynamic analysis**
Researchers have studied also the combination of text retrieval with both static and dynamic techniques, as they all offer a different perspective on the concept location problem.

Eaddy et al. [20] were the first to propose an approach that combines all three sources of information. The approach uses a technique called prune dependency analysis, which functions based on the principle that a relationship between a program element and a feature exists if the program element should be removed or modified if the feature were to be pruned from the software system. The new approach uses PROMESIR to combine rankings of program elements from execution traces with rankings from text retrieval to produce seeds for the pruning process. Given an initial set of relevant elements to be pruned, the approach determines additional relevant elements. Using a large benchmark of mappings between code and features for over 400 features, the new approach was compared to techniques using each of the individual approaches it is based on independently and in combinations. The results revealed that combining the three types of analysis was the most effective approach.

Revelle et al. [28] proposed a feature location technique that combines text retrieval with the results produced by applying advanced link analysis algorithms on a call graph containing methods obtained using execution information. The approach takes as an input a query and the trace resulted after the execution of a scenario that exercises the wanted feature. A program dependence graph is generated for the methods present in the execution trace by using the caller and callees methods as nodes and the relations between them as edges. Two link analysis algorithms, i.e., PageRank [Brin'98] and HITS [Kleinberg'99] are applied on the program dependence graph and assign a score to each method in the graph based on their importance in the graph. The proposed technique for concept location filters out from the execution trace those methods that obtained either a very high or very low score from the link analysis algorithms (they are either at the top or at the bottom of the ranked list returned by the link analysis algorithms). After these methods are eliminated, the remaining methods in the trace are ranked using the text retrieval technique, based on their similarity to the input query. The evaluation of the new approach was performed on two systems and it was compared to using link analysis techniques alone, LSI, and the SITIR [15] approach. The results showed that the new approach combining text retrieval techniques with web mining, static and dynamic information outperformed all the other approaches.

Hayashi et al. [21] proposed an iterative approach to feature location which combines LSI with dynamic analysis based on the execution of a test case, and static analysis, which is used to navigate the dependencies of the methods found in the execution trace. The approach also makes use of relevance feedback. The paper claims that the iterative approach leads to improved query formulation by end users and the feedback provided by users during the iterative process enhances the understanding of features implemented in a given software system. The proposed approach requires as input source code, a test case (used to derive dynamic dependencies), a query, and hints (relevance feedback) and returns to the user source code entities ordered by their respective evaluation scores. Evaluation of the tool, which compares the interactive and non-interactive versions of the approach, indicate that the iterative technique is capable of reducing overhead, although does not always perform well.

**Text retrieval combined with historical information**
The approach introduced by Cubranic et al. [35] makes use of text retrieval and archival information for feature location. The approach forms a group memory from the history of a project as recorded in source code repositories, issue trackers, communication channels, and web documents, based on which it recommends artifacts, such as, online documentation, file versions, bug reports, or communications. Text retrieval is used to determine links between these artifacts, as well as artifacts relevant to a user query. The approach has been evaluated in two case studies, with promising results.

[24] used information found in past bug reports to improve concept location using text retrieval, and presented an extended model of LSI which takes advantage of this information as well. The new approach mined data from past bug reports where the bug has been linked to its location in the source code. The three stages followed in the approach are extracting semantic data from source code, adding additional information from previous bugs, and querying the LSI model. The two case studies performed in order to evaluate this model revealed that the new model taking into consideration also historical data performs significantly better than LSI alone.

Ratanotayanon et al. [60] introduced the notion of Transitive Changeset, which is a changeset extended using information extracted from the revision history. A changeset temporally associates changes and conceptual descriptions provided in a commit transaction. Transitive Changesets are created from information that is recorded by revision control systems and other common software tools, such as issue trackers, and extend the available information using transitive relationships. Transitive Changesets contain conceptual-level information that is difficult to find in the source code and relates this information to a list of program elements in the code. Using information retrieval techniques, the Transitive Changesets are indexed to create a searchable repository with which programmers can locate features through searching changesets.

The program elements included in the group of selected changesets are expanded using a static dependency graph ranked using relevance metrics. A prototype of this approach was developed as an Eclipse plug-in, named Kayley.

Kadgi et al. [61] have recently introduced a technique making use of LSI and historical information captured in version control repositories in order to recommend a ranked list of expert developers to assist in the implementation of software change requests (e.g., bug reports and feature requests). LSI is first used to perform concept location on the source code and identify source code entities, e.g., files and classes, relevant to a given textual description of a change request. The previous commits from version control repositories of these entities are then mined for expert developers. The role of the IR method in selectively reducing the mining space is different from previous approaches that textually index past change requests and commits. The approach is evaluated on change requests from three open-source systems and the results show that the presented approach outperforms two previous recommendation alternatives substantially.

## 4 Tools for Text Retrieval-Based Concept Location

IRiSS [38] and JIRiSS [39] are both tools for text retrieval-based concept location, and are based on LSI. IRiSS implements text retrieval-based concept location as an add-on to MS Visual Studio .NET, while JIRiSS implements it in Eclipse, as a plug-in. Both tools work like the built-in search functionality in the two development environments, with the difference that they use LSI instead of keyword-based retrieval, they present the results in a ranked list, and the classes and methods that match the query are listed, instead of just lines of code. JIRiSS is an extension to IRiSS that also includes fragment-based retrieval, software vocabulary extraction, query spell checking, and word suggestions to improve queries.

Xie et al. [Xie'06] introduced a tool that supports textual analysis through visualization, by combining IRiSS [Poshyvanyk'05] and sv3D [Marcus'03]. IRiSS performs concept location via LSI and sv3D creates a 3D visualization of the results, showing polycylinders that represent classes and methods in the system. The colors of the polycylinders correspond to the similarity to the query and the height of the polycylinders represent the number of times the program element was visited in the past. The combination of these two tools allows a developer to have a visual representation of the results, as opposed to examine a ranked list of results.

Hipikat, introduced by Cubranic et al. [35] is a tool that makes use of text retrieval and historical information for concept location. Along with source code, the tool makes use of documents from source code repositories, issue trackers, communication channels, and web documents in order to locate artifacts that are related to a user query. The query can be itself an artifact, and Hipikat uses text retrieval to locate the artifacts that are similar to it.

Cleary and Exton [Cleary'06] implemented an Eclipse plug-in based on language models that supports the cognitive assignment technique proposed by the authors. The tool allows a developer that is unfamiliar with a system to generate and store a set of links between the problem domain concepts stored in a cognitive map and the relevant parts of the source code. The developer can select a particular concept as a query and based on it the tool uses the language model representation to identify program elements that the developer should investigated.

Google Eclipse Search (GES) [62] is an Eclipse plug-in that integrates Google Desktop Search (GDS) and Eclipse for source code retrieval. GDS is an off-the-shelf component that uses a proprietary information retrieval algorithm for its retrieval. It allows users to search for files on their desktops similar to the way they would search for information on the Internet. By integrating GDS with Eclipse, programmers can search source code in a similar way. In an evaluation on a Java system, GES was shown to produce accurate results and when compared against Eclipse file search functionality, GES is considerably faster in producing the results.

TopicXP, developed by Savage et al. [Savage'10a] is an Eclipse plug-in that extracts a set of topics (which can be considered as concepts) from the source code using LDA. The topics generated are mapped to the source code, and the relationship between the topics is determined by examining the static dependencies from the code. The developer is able to navigate through these topics or to access the source code associated with them.

The Feature Location and Textual Tracing Tool (FLAT3) [Savage'10b] implements support for text retrieval-based and dynamic feature location, based on the SITIR technique introduced in [Liu'07]. Programmers can define and name features and then associate entire or partial classes, methods, and fields with them, based on the concept location they perform using the tool. Once the features and their program elements are defined and linked, they can be saved and retrieved at a later time.

## 5 Evaluation of Text Retrieval-Based Concept Location

Evaluation plays an important part in text retrieval-based concept location, as it determines if new approaches are found useful by developers or if they improve the state of the art and come closer to solving the problem. The evaluation of text retrieval-based concept location techniques has been approached from different perspectives by researchers in the field: qualitative or quantitative. The qualitative evaluation aims at investigating *how well* the text retrieval-based techniques satisfied the information need of the user, and what was learned in the process. Quantitative evaluation, on the other hand, investigates the "*how much*" aspects of the evaluation, i.e., how much effort is needed for locating the concepts, how much time does a user spend to locate the concept using the text retrieval-based approach, how much better did one technique perform over the previous work, how much of the user information need was satisfied, etc.

### 5.1 Measures and Metrics

Quantitative evaluation usually involves a systematic empirical investigation of a research approach via statistical, mathematical or computational techniques. Metrics and measures are central to quantitative evaluation. Using quantitative evaluation, new research approaches can be easily compared to previous efforts in order to get an estimation of *how much* the new techniques achieve.

For concept location based on text retrieval, there are a series of measures that researchers have used in order to evaluate the performance of the techniques quantitatively. Some of them are measures used to evaluate text retrieval techniques in the field of information retrieval, while others are specific to concept location. We briefly introduce the most widely used metrics before discussing how researchers have addressed the quantitative evaluation in text retrieval-based concept location. As some works use more than one measure for the quantitative evaluation, we will discuss the details of the works after defining all the measures, instead of mentioning them right after each definition.

*Jaccard index*

The most basic measure used for the evaluation of text retrieval-based concept location techniques, the *Jaccard index* is a measure for comparing the similarity and diversity of sample sets. It is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Researchers have used the Jaccard measure in order to determine the similarity between the set of results returned by a text retrieval approach to the set of target methods.

*Precision*

*Precision* is one of the metrics used in text retrieval to measure the performance of an approach. It is measured by the fraction of retrieved documents that are also relevant to the user query.

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Precision takes by default all retrieved documents into account, but it can also be evaluated at a given cut-off rank, considering only the topmost results returned by the approach. This measure is called precision *at n*. A high precision means that many of the retrieved documents are relevant to the query. This does not mean, however, that all relevant documents were retrieved.

Note that the meaning and usage of “precision” in the field of Information Retrieval differs from the definition of accuracy and precision within other branches of science and technology.

*Recall*

*Recall* is a measure often used alongside precision to evaluate text retrieval techniques, and it is also from the field of Information Retrieval. Recall is equal to the ratio of relevant documents that are retrieved, divided by the total number of documents retrieved.

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

High recall signifies that many of the relevant documents were retrieved. However, unrelated documents might have also been retrieved. It is trivial to achieve recall of 100% by returning all documents in response to any query. Therefore, recall alone is not enough as one needs to measure also the number of non-relevant documents, for example by computing the precision. Recall and precision are, thus, often computed together.

*Average precision*

*Average precision* is defined as the mean of the precision scores obtained after each relevant document is retrieved, and measures how accurately a text retrieval technique ranks a set of known relevant documents. An approach that returns more relevant documents with higher rankings will perform better under the average precision measure when compared with an approach which does not. For example, considering a set of 10 relevant documents out of a corpus of 100, an approach that returns the 10 relevant documents in the top 10 ranking positions has have an average precision of 1 whereas an approach which does not rank all the relevant documents in the top 10 positions would get an average precision score of < 1. The average precision v for a text retrieval approach is defined over the set of document rankings produced by the approach as follows:

$$v = \frac{1}{R}\sum_{i=1}^{n}\frac{x_i}{i}\sum_{k=1}^{i}x_k$$

where R is the total number of relevant documents in the collection, n is the number of documents included in the ranked document list, and $x_i$ and $x_k$ are 1 if the $i^{th}$ and $k^{th}$ documents, respectively, are relevant or 0 otherwise.

*Effectiveness*

*Effectiveness* is a measure specific for evaluating concept location approaches, and was first proposed by Poshyvanyk et al. [13]. The measure was introduced in order to allow the comparison between concept location approaches using text retrieval and those using other methods, like dynamic analysis. Previously, the measures used for evaluating text retrieval-based concept location techniques were precision and recall, which may not be adequate for evaluating other approaches. Moreover, when using text retrieval, unless a threshold is used the recall is always 100%, as all the documents in the system are retrieved and ranked. Also, if the approach applies text retrieval on an execution trace, precision is always 1/n, where n is the number of executed methods to a given scenario.

To deal with these issues, *effectiveness* was defined as the rank of the first changed method related to the concept or feature of interest. This allows also for a measurement of the effort of a developer during the location process, which can be defined as the number of methods which appear in the final ranked list that the developer needs to investigate. A lower value effectiveness value indicates less effort, hence a more effective technique.

## 5.2 Quantitative Evaluations

Asadi et al. [18] used quantitative evaluation in two case studies to investigate the accuracy and completeness of their new approach, based on LSI, dynamic information, and a search-based optimization algorithm. More specifically, they used the Jaccard measure in order to determine how stable the approach was in determining concepts in an execution trace and how much the concepts identified by the approach were similar to the oracle. They also used precision in order to determine the accuracy of the determined concepts. The new approach was not compared to previous ones.

Marcus et al. [33] performed the quantitative evaluation of the LSI-based concept location approach by using three measures, i.e., precision, recall, and the position of the last target method in the list of results returned by LSI. A series of user queries and automatically generated queries were run the results indicated that the text retrieval-based approach returns the target methods in most cases within the first 22 results.

Zhao et al. [30, 32] performed a quantitative evaluation of the SNIAFL approach using a large set of features from two software systems and precision and recall. They compared the results of their approach to the approach using only text retrieval and to the one using only dynamic information. The results obtained revealed that the SNIAFL approach outperforms the other two approaches.

In order to be able to compare their results with those obtained by Zhao et al. [30], Ahn et al. [31] followed closely the evaluation approach used in [30]. For that, they used the same data set as [30], performed a quantitative evaluation of their newly proposed approach using precision and recall , and compared their results to the ones obtained using the approach introduced by Zhao et al. [30]. They used cut values of 3, 6, 9, 12, and 15 in order to compute the recall and precision at different thresholds, and performed the concept location using the two techniques for all the requirements in the system to get the average of precision and recall.

Cubranic et al. performed in [35] a quantitative evaluation of their approach, Hipikat. A set of Eclipse changes were extracted from the issue tracking system and a set of 20 bugs were randomly selected from these to perform the evaluation. The measures used for the quantitative evaluation were recall and precision, as well as the rank of the first useful recommendation in the list of results. Even though Hipikat was not compared to any other approaches to concept location, the results indicated that it was able to determine all the affected files in the majority of the cases.

Recall, precision and f-measure were also used also in [20] in order to compare CERBERUS with previous approaches to concept location. Recall and precision were used also as measures of performance in [25, 26].

Cleary et al. [19] performed a quantitative assessment of the newly introduced cognitive assignment tool on a software system. The accuracy of the new approach was determined by computing the average precision based on a set of expert mappings between the description of four concerns and their implementation in the source code. At the same time, the new approach was compared using average precision to three other approaches, i.e., a classic language models approach, a dependency based language modeling approach, and LSI and the results indicated that the new approach significantly outperformed the previous approaches.

The same authors performed another evaluation of the cognitive assignment approach in [16], where they used the same measure of average precision in order to compare their approach to several other existing approaches, i.e., VSM, LSI, the classical language model approach, the dependency-based language model approach, and KL-divergence. The results showed that overall the cognitive assignment approach performs better than any of the other approaches.

Poshyvanyk et al. [13] performed three case studies to quantitatively evaluate the newly introduced approach based on combining LSI with Probabilistic Ranking. The metric used for the evaluation was the effectiveness. The new approach, PROMESIR performed the best when compared with LSI and Scenario-based Probabilistic Ranking (SPR). [15] used the same measure, i.e., effectiveness, to further compare PROMESIR to LSI and SITIR, which is a new approach based on using LSI to rank the methods in a single execution trace. SITIR proved to be comparable in results to PROMESIR, while requiring less user effort.

Rao and Kak [63] used both average precision and effectiveness in order to compare nine different text retrieval techniques, both generic and composite, for bug location in source code. Effectiveness was also as the quantitative measure of choice in [10, 11, 17, 22-24, 28, 29].

Poshyvanyk and Marcus use a specific set of measures in [14], i.e., lattice distillation factor and lattice browsing complexity in order to examine the benefits of their newly introduced approach, based on using Formal Concept Analysis with LSI. These measures are, however, specific for approaches using lattices to organize results.

Revelle et al. [27] used a new metric, i.e., the percentage of relevant methods in the top ten results retrieved, for comparing the performance of ten feature location techniques, all making use of text retrieval, alone or in combination with other sources of data.

## 5.3 Qualitative Evaluations

In some cases, only a quantitative evaluation might not be enough to fully understand the implication of the results or some characteristics of the approach being evaluated. Under these circumstances, researchers have made use of *qualitative evaluations*, which discuss in detail aspects of the technique used or results obtained.

Cleary et al. [16] performed a qualitative analysis of their cognitive assignment approach in order to identify reasons for the failure of the approach in some cases. In order to do this, they chose two concepts in Eclipse JDT Core for which they performed a manual identification of the methods involved in the implementation of the concepts. For these two concepts, they analyzed in detail the terms used in the source code of the involved methods, as well as the properties of the methods like size, quantity of comments included in the method, the quality of the identifiers, etc. They found that most of the terms in the query did not appear in the source code of one of the concepts, for which the concept assignment technique failed, as opposed to the second concept, where the terms included in the query were relevant terms existing in the source code, thus explaining the success of the cognitive assignment approach.

In [41] the authors performed an initial qualitative evaluation of the Hipikat approach, which focused on whether the recommendations of relevant artifacts given by the system were of help to developers working on a change task. The authors also investigated if there were recommendations that the developers would have found useful but Hipikat did not recommend. In a first study, using an oracle group memory for a medium-sized system as a database for Hipikat, the authors had pairs of graduate students analyze the recommendations given by Hipikat while they were performing two change tasks on the system. At the end of the study, seven pairs of graduate students submitted a report and six students were interviewed. Overall, the subjects reported that Hipikat helped them to start the change tasks they were assigned, as well as helped them identify the classes and methods that they need to understand and change. The study revealed also that the usefulness of the suggestions depends on the context of the suggestion and on the experience of the developer receiving the suggestion. In a second study, Hipikat was evaluated on a completed enhancement for Eclipse that was logged in BugZilla. The subject of the study was one of the authors of the paper and he was observed while using Hipikat for the change task in Eclipse.

In a second evaluation of Hipikat [34], the authors study the usefulness of the recommendations of the approach on two previously completed enhancements (one easy one difficult) in Eclipse, extracted from the issue tracking database of Eclipse. The twelve paid participants in the study were mostly a mix of graduate students and professional developers. In order to capture the data needed for the qualitative evaluation, the authors used recorded interviews with the participants, performed first after they had a mental plan of the change before the implementation, and then after the implementation of the change was completed. A screen capture software was also used to record the actions performed by the participants, and Hipikat was instrumented to capture the queries written by developers in a file. The results of the qualitative analysis of all the materials recorded during the study revealed that newcomers can use the information presented by Hipikat to achieve results comparable in quality and correctness to those of more experienced members of the team.

Marcus et al. performed also a qualitative evaluation of the LSI-based approach to concept location in [33], by analyzing in detail the results obtained by the new approach to those obtained by grep and dependency graph navigation in [64]. The LSI-based approach was found to be almost as easy and flexible to use as grep.

Additionally, LSI led to better results as it was able to identify certain parts of a concept that were missed by the dependence graph search and grep.

The same approach introduced in [33] was qualitatively evaluated on object oriented software systems in [36]. The process of finding the relevant methods using LSI, grep and dependency graph search was analyzed and compared on one Java and one C++ system. The strengths and weaknesses of each approach were identified and discussed and the advantages of a combination of the approaches are highlighted.

Zhao et al. [30, 32] performed also a qualitative evaluation of their SNIAFL approach, along with the quantitative evaluation. Along with SNIAFL, they also analyzed the approaches using only text retrieval and only dynamic information, respectively. They conclude that while their approach works well on average, for the recovered pseudo execution traces, it leads to too many irrelevant traces. However, the approach was able to find some unusual traces and the overall effectiveness of the approach was underlined.

Dit et al. [11] also performed a qualitative evaluation of their results and described the problems and advantages of each of the identifier splitting techniques studied. They also noticed that in some cases the queries presented problems due to the vocabulary mismatch problem, and that this mismatch was more severe for bugs than features.

Hayashi et al. [21] performed a brief qualitative evaluation of their approach and determined two reasons for the poor performance of their technique in some cases, i.e., inappropriate queries and events related to more than one feature.

## 6 Discussion and Directions for Future Research

While the paper did not present a timeline of the research on using text retrieval for concept location, there is an observable evolution of the work in the field. Early efforts focused more on determining whether particular text retrieval models (e.g., VSM, LSI, probabilistic models, etc.) are suitable for this problem and on defining methodologies of using such techniques. Evaluation of the early work was based on small case studies. More recent work focused on combining text retrieval with structural and dynamic analysis. Meanwhile the empirical evaluation matured and the newer work is evaluated using sizeable data extracted from software repositories and also includes comparing specific techniques. Newer retrieval models (e.g., based on topic modeling) were also investigated recently and the attention of researchers also shifted to issues such as query formulation, corpus normalization, etc.

All in all, text retrieval proved to be an essential technique to support concept location in source code. So much so that all state of the art techniques incorporate today a text retrieval engine, one way or another. More than that, related approaches (e.g., traceability link recovery, source code search in repositories, bug localization, etc.) all use today text retrieval techniques.

While the combination of text retrieval with additional software analyses (i.e., structural and dynamic) seems to be the best suited for concept location, the use of text retrieval in support of this important software engineering problem deserves more attention and needs further research.

Most text retrieval techniques are used in black-box fashion when applied to source code based corpora. Such techniques come with a variety of parameters that need to be tuned for individual applications. Parameter tuning and customization of text retrieval techniques for source code corpora should be investigated further.

The empirical validation of this work matured in the past years, but it needs to grow further. There is a need of annotated corpora and evaluation data, which hopefully will eventually result in a community defined and accepted benchmark. With such a benchmark in place, more subtle work can be undertaken. For example, one could investigate the combined use of various text retrieval techniques or more interesting ways to create the source code corpus could be defined (e.g., by employing selective weighting schemes on source code terms).

## References

1. Rajlich, V.: Intensions are a Key to Program Comprehension. In: International Conference on Program Comprehension, pp. 1–9 (2009)
2. Biggerstaff, T.J., Mitbander, B.G., Webster, D.E.: The Concept Assignment Problem in Program Understanding. In: 15th IEEE/ACM International Conference on Software Engineering, pp. 482–498 (1994)
3. Rajlich, V., Wilde, N.: The Role of Concepts in Program Comprehension. In: IEEE International Workshop on Program Comprehension, pp. 271–278. IEEE Computer Society Press (2002)
4. Wilde, N., et al.: Locating User Functionality in Old Code. In: IEEE International Conference on Software Maintenance, pp. 200–205 (1992)
5. Robillard, M.P., Murphy, G.C.: Representing concerns in source code. ACM Transactions on Software Engineering and Methodology 16(1) (2007)
6. Salton, G., McGill, M.: Introduction to Modern Information Retrieval. McGraw-Hill (1983)
7. Rajlich, V., Gosavi, P.: Incremental Change in Object-Oriented Programming. IEEE Software 21(4), 62–69 (2004)
8. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
9. Porter, M.: An Algorithm for Suffix Stripping. Program 14(3), 130–137 (1980)
10. Gay, G., et al.: On the Use of Relevance Feedback in IR-Based Concept Location. In: IEEE International Conference on Software Maintenance, pp. 351–360 (2009)
11. Dit, B., et al.: Can Better Identifier Splitting Techniques Help Feature Location? In: 19th IEEE International Conference on Program Comprehension, pp. 11–20 (2011)
12. Poshyvanyk, D., et al.: Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. In: 14th IEEE International Conference on Program Comprehension, pp. 137–146 (2006)
13. Poshyvanyk, D., et al.: Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval. IEEE Transactions on Software Engineering 33(6), 420–432 (2007)

14. Poshyvanyk, D., Marcus, A.: Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In: 15th IEEE International Conference on Program Comprehension, pp. 37–46. IEEE Computer Society (2007)
15. Liu, D., et al.: Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace. In: 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 234–243 (2007)
16. Cleary, B., et al.: An empirical analysis of information retrieval based concept location techniques in software comprehension. Empirical Software Engineering 14(1), 93–130 (2009)
17. Scanniello, G., Marcus, A.: Clustering Support for Static Concept Location in Source Code. In: 19th IEEE International Conference on Program Comprehension, pp. 1–10 (2011)
18. Asadi, F., et al.: A Heuristic-based Approach to Identify Concepts in Execution Traces. In: 14th European Conference on Software Maintenance and Reengineering, pp. 31–40 (2010)
19. Cleary, B., Exton, C.: Assisting Concept Location in Software Comprehension. In: 19th Psychology of Programming Workshop, pp. 42–55 (2007)
20. Eaddy, M., et al.: CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In: 17th IEEE International Conference on Program Comprehension, pp. 53–62 (2008)
21. Hayashi, S., Sekine, K., Saeki, M.: iFL: An Interactive Environment for Understanding Feature Implementations. In: 26th IEEE International Conference on Software Maintenance, pp. 1–5 (2010)
22. Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In: 15th Working Conference on Reverse Engineering, pp. 155–164 (2008)
23. Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using Latent Dirichlet Allocation. Information and Software Technology 52, 972–990 (2010)
24. Nichols, B.D.: Augmented bug localization using past bug information. In: 48th ACM Annual Southeast Regional Conference, pp. 1–6 (2010)
25. Peng, X., et al.: Iterative Context-Aware Feature Location. In: 33rd International Conference on Software Engineering, NIER Track, pp. 900–903 (2011)
26. Ratanotayanon, S., Choi, H.J., Sim, S.E.: My Repository Runneth Over: An Empirical Study on Diversifying Data Sources to Improve Feature Search. In: 18th IEEE International Conference on Program Comprehension, pp. 206–305 (2010)
27. Revelle, M., Poshyvanyk, D.: An Exploratory Study on Assessing Feature Location Techniques. In: 17th IEEE International Conference on Program Comprehension, pp. 218–222 (2009)
28. Revelle, M., Dit, B., Poshyvanyk, D.: Using Data Fusion and Web Mining to Support Feature Location in Software. In: 18th IEEE International Conference on Program Comprehension, pp. 14–23 (2010)
29. Shao, P., Smith, R.K.: Feature location by IR modules and call graph. In: 47th ACM Annual Southeast Regional Conference (2009)
30. Zhao, W., et al.: SNIAFL: towards a static non-interactive approach to feature location. In: 26th International Conference on Software Engineering, pp. 293–303 (2004)
31. Ahn, S.-Y., et al.: A Weighted Call Graph Approach for Finding Relevant Components in Source Code. In: 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, pp. 539–544 (2009)
32. Zhao, W., et al.: SNIAFL: Towards a Static Non-interactive Approach to Feature Location. ACM Transactions on Software Engineering and Methodologies 15(2), 195–226 (2006)

33. Marcus, A., et al.: An Information Retrieval Approach to Concept Location in Source Code. In: 11th IEEE Working Conference on Reverse Engineering, pp. 214–223 (2004)
34. Cubranic, D., et al.: Learning from project history: a case study for software development. In: ACM Conference on Computer Supported Cooperative Work, pp. 82–91 (2004)
35. Cubranic, D., et al.: Hipikat: A Project Memory for Software Development. IEEE Transactions on Software Engineering 31(6), 446–465 (2005)
36. Marcus, A., et al.: Static Techniques for Concept Location in Object-Oriented Code. In: 13th IEEE International Workshop on Program Comprehension, pp. 33–42 (2005)
37. Enslen, E., et al.: Mining Source Code to Automatically Split Identifiers for Software Analysis. In: 6th IEEE Working Conference on Mining Software Repositories, pp. 71–80 (2009)
38. Poshyvanyk, D., et al.: IRiSS - A Source Code Exploration Tool. In: 21st IEEE International Conference on Software Maintenance, pp. 69–72 (2005)
39. Poshyvanyk, D., Marcus, A., Dong, Y.: JIRiSS - an Eclipse plug-in for Source Code Exploration. In: 14th IEEE International Conference on Program Comprehension, pp. 252–255 (2006)
40. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley (1999)
41. Cubranic, D., Murphy, G.C.: Hipikat: Recommending pertinent software development artifacts. In: 25th International Conference on Software Engineering, pp. 408–418 (2003)
42. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Communications of the ACM 18(11), 613–620 (1975)
43. Hatcher, E., Gospodnetić, O.: Lucene in Action. Manning Publications (2004)
44. Savage, T., Revelle, M., Poshyvanyk, D.: FLAT^3: Feature Location and Textual Tracing Tool. In: 32nd ACM/IEEE International Conference on Software Engineering, Tool Demo, pp. 255–258 (2010)
45. Deerwester, S., et al.: Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science 41, 391–407 (1990)
46. Dit, B.: Monitoring the Searching and Browsing Behavior of Developers in Eclipse during Concept Location. Department of Computer Science, Wayne State University, Detroit (2009)
47. Hofmann, T.: From Latent Semantic Indexing to Language Models and Back. In: Workshop on Language Modeling and Information Retrieval (2001)
48. Ponte, J.M., Croft, W.B.: A Language Modeling Approach to Information Retrieval. In: 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 275–281 (1998)
49. Cleary, B., Exton, C.: The Cognitive Assignment Eclipse Plug-in. In: 14th IEEE International Conference on Program Comprehension, pp. 241–244 (2006)
50. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet Allocation. Journal of Machine Learning Research 3, 993–1022 (2003)
51. Kuhn, A., Ducasse, S., Girba, T.: Semantic Clustering: Identifying Topics in Source Code. Information and Software Technology 49(3), 230–243 (2007)
52. Ohlemacher, S., Marcus, A.: Towards a Benchmark and Automatic Calibration for IR-based Concept Location. In: 19th IEEE International Conference on Program Comprehension, pp. 246–249 (2011)
53. Henninger, S.: Using iterative refinement to find reusable software. IEEE Software 11(5), 48–59 (1994)
54. Furnas, G.W., et al.: The Vocabulary Problem in Human-System Communication. Communications of the ACM 30(11), 964–971 (1987)

55. Starke, J., Luce, C., Sillito, J.: Searching and Skimming: An Exploratory Study. In: International Conference on Software Maintenance, pp. 157–166 (2009)
56. Song, D., Bruza, P.: Towards Context-sensitive Information Inference. Journal of the American Soceity for Information Science and Technology 4, 321–334 (2003)
57. Haiduc, S., Marcus, A.: On the Effect of the Query in IR-based Concept Location. In: 19th IEEE International Conference on Program Comprehension, pp. 234–237 (2011)
58. Antoniol, G., Gueheneuc, Y.G.: Feature Identification: An Epidemiological Metaphor. IEEE Transactions on Software Engineering 32(9), 627–641 (2006)
59. Marcus, A., Poshyvanyk, D.: The Conceptual Cohesion of Classes. In: 21st IEEE International Conference on Software Maintenance, pp. 133–142 (2005)
60. Ratanotayanon, S., Choi, H.J., Elliott Sim, S.: Using transitive changesets to support feature location. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 341–344 (2010)
61. Kagdi, H., et al.: Assigning change requests to software developers. Journal of Software Maintenance and Evolution: Research and Practice (2011) (to appear)
62. Poshyvanyk, D., Petrenko, M., Marcus, A.: Integrating COTS Search Engines into Eclipse: Google Desktop Case Study. In: Proceedings of the 2nd International ICSE 2007 Workshop on Incorporating COTS Software Into Software Systems: Tools and Techniques, pp. 6–10 (2007)
63. Rao, S., Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: 8th Working Conference on Mining Software Repositories, pp. 43–52 (2011)
64. Chen, K., Vaclav, R.: RIPPLES: Tool for Change in Legacy Software. In: International Conference on Software Maintenance, pp. 230–239 (2001)

# Discovering Services

Andrea Zisman

School of Informatics, City University London,
London, EC1V 0HB, United Kingdom
a.zisman@soi.city.ac.uk

**Abstract.** This tutorial paper presents an overview of existing approaches for service discovery and describes a service discovery framework that can support both static and dynamic service discovery. The framework and its extensions have been developed within the EU 6th Framework projects SeCSE and Gredia and the EU 7th Framework Network of Excellence S-Cube.

## 1 Introduction

Current software systems need to be more flexible, adaptable, and versatile. Service-oriented computing (SOC), a paradigm that envisages software as a temporary service rather than permanent property, aims to provide a more flexible approach to software development. In this context, services are loosely-coupled autonomous computer-based entities owned by third parties with different functionality, which can be combined to realise applications and create dynamic business processes.

Service-oriented computing has attracted great interest from industry and research communities all over the world. Service integrators, developers, and providers are collaborating to address the various challenges in the area. Various approaches and tools have been proposed to support different areas of SOC such as (a) languages to describe services, (b) service design and development, (c) service discovery, (d) service composition and adaptation, (e) service management and monitoring, and (f) service governance.

An important activity in SOC is concerned with service discovery; i.e., the identification of services based on one, or a combination of, functional, behavioural, quality, and contextual aspects. Several approaches have been proposed to support service discovery. These approaches can be classified as (a) *static* [23][32][41][62] or (b) *dynamic* [16][19][79][80] service discovery approaches. The static approaches are characterized by the identification of services during development (design-time) of service-based systems to assist with the development of such systems. The identified services are bound to the service-based systems prior to the execution of the systems. The dynamic approaches are characterised by the identification of services during execution time of service-based systems in order to replace participating services that may become malfunctioning or unavailable, or are changed by service providers or contextual characteristics (e.g., performance or cost changes). In this case, the identified services are bound to the service-based systems during the execution of the systems.

In this tutorial paper, we present an overview of existing approaches for service discovery and describe a service discovery framework that supports both static and dynamic service discovery. The work presented in this paper was developed with the collaboration of various colleagues and contains the results of several years of research in the topic as already described in various papers [18][37][38][44][45] [63][64][65][66][77][78][79][80][81]. In this paper, we provide a summary of the work; where details can be found in the list of references above. The work has been developed as part of three European-funded research projects, namely (a) the Service-centric System Engineering (SeCSE) EU-ICT-FP6 project [60], (b) the Grid Enable access to rich meDIA content (GREDIA) EU-ICT-FP6 project [22], and (c) the Software Services and Systems (S-CUBE) EU-ICT-FP7 project [59].

The remaining of this paper is organized as follows. In Section 2 we present an account of existing approaches for service discovery. In Section 3 we discuss our static and dynamic service discovery framework. In Section 4 we present some extensions of the framework. Finally, in Section 5 we summarise the work and discuss some final remarks.

## 2 Overview of Existing Approaches

Several approaches have been proposed in the literature to support service discovery. In [21], the authors introduce the topic of service discovery and describe some initial work in this area. In [28], the discovery of services is addressed as a problem of matching queries specified using a variant of Description Logic (DL) with service profiles specified in OWL-S [50]. The matching process is based on the computation of subsumption relations between service profiles and supports different types of matching (exact, plug-in, subsume, intersection, and disjoint matching).

The work in [26] proposes the use of graph transformation rules for specifying services and service discovery queries. These rules represent each service operation by two "source" and "target" object graphs whose nodes and edges correspond to data entities and relationships between them, respectively. Matching in this approach is based on the use of RDQL by testing sub-graph relations and establishing if a specification matching relation holds between the query and the service description. In [34], the authors have also proposed the use of graph-matching for service discovery but very few details of the matching algorithm are available.

The approach in [72] proposes four similarity assessment methods to support service matching, namely lexical, attribute, interface, and quality-of-service (QoS) similarity. These forms of similarity assessment can be used either separately or jointly. Their lexical similarity method calculates the distance between two words in concept hierarchies based on the lexicons WordNet and HowNet. The attribute similarity uses hyperonymy/hyponymy relations to construct hierarchical structures and calculates the similarity of two attributes based on the distance of the nodes that represent them in the hierarchy. The interface similarity is based on the comparison of the names and types of the parameters of the operations of two services. The parameter name similarity is assessed by using the lexical similarity while the similarity of the parameter data types is computed by using the data type mapping table proposed in [13]. An evaluation of the approach in [72] has shown precision between 42% and 62% for interface similarities whilst the preliminary evaluation of our approach showed precision figures in the range of 55% to 100%.

METEOR-S [2] is a system that adopts a constraint driven approach to service discovery in which queries are integrated into the composition process of a service based system and are represented as a collection of tuples of features, weight, and constraints. Details of the query matching process, which is used in this system, were not available.

The approach in [29] uses service descriptions constructed as collections of annotated multi-purpose extensible data containers, called "tuples", which can be queried via XQuery. The service specifications assumed in [34] include only specifications of service operation signatures and do not incorporate QoS properties or behavioural descriptions of services. Thus, this approach is primarily focused on interface queries where operation signatures are matched using string matching. This form of matching is very limited, as it cannot account for even small variations in operation signature specifications such as the use of different parameter names or alternative orderings of parameters.

In [24][25], the authors advocate the use of (abstract) behavioural models of service specifications in order to increase the precision of service discovery. Their approach locates services, which satisfy task requirement properties expressed formally in temporal logic, by using a lightweight automated reasoning tool. Another approach that suggests behavioral matching for service discovery based on similarity measures has been proposed in [23].

In [62], the authors also argue that the use of behavior signatures of web services such as conversations of web services, events and activities of services, and semantic descriptions of services can improve service discovery. Their approach proposes a behavioral model for services, which associates messages exchanged between services with activities performed within the services. A query language based on first-order logic that focuses on properties of behavior signatures is used to support the discovery process. These properties include temporal features of sequences of service messages or activities and semantic descriptions of activities. The discovery process is based on the use of evaluation algorithms for the query language. Another approach that advocates the use of behavioral specifications represented as BPEL [9] for service discovery has been proposed in [48]. In this approach the authors suggest the use of BPEL specifications as a way of resolving ambiguities between requests and services and use a tree-alignment algorithm to identify matching between request and services.

There have been proposals for specific query languages to support web services discovery [51][52][55][74]. The query language proposed in [55] is used to support composition of services based on user’s goals. NaLIX [74], which is a language that was developed to allow querying XML databases based on natural language, has also been adapted to cater for service discovery. In [51], the authors propose USQL (Unified Service Query language), an XML-based language to represent syntactic, semantic, and quality of service search criteria. The query language used in [81] is more complete, since it accounts for the representation of behavioral aspects of the system being developed. Moreover, in [81], structural and behavioral criteria are represented by complete UML class and interaction models including the specification of complex types. In addition, the constraint query language in [81] allows for the specification of not only quality aspects of the system, but also extra conditions concerned with structural and behavioral criteria. An extension of USQL that incorporates the behavioral part of our query language has been proposed in [52].

Semantic web matchmaking approaches have been proposed to support service discovery based on logic reasoning of terminological concept relations represented on ontologies [1][2][28][32][35][36][41][68][69]. The METEOR-S [2] system adopts a constraint driven service discovery approach in which queries are integrated into the composition process of a service base system and are represented as a collection of tuples of features, weight, and constraints. In [1], semantic, temporal, and security constraints are considered during service discovery. In our framework, extra constraints concerned with structural, behavioral, and quality aspects of the system being constructed are also considered. In [28] the discovery of services is addressed as a problem of matching queries specified as a variant of Description Logic (DL). The work in [35] extends existing approaches by supporting explicit and implicit semantic by using logic based, approximate matching, and IR techniques. The work in [69] proposes QoS-based selection of services. In [32], the authors present a goal-based model for service discovery that considers re-use of pre-defined goals, discovery of relevant abstract services described in terms of capabilities that do not depend on dynamic factors (state), and contracting of concrete services to fulfill requesting goals. These best matches provide the designer an opportunity to choose the most adequate service and to become more familiar with the available services and, therefore, design the system based on this availability. Matching based on the structure of data types is important during the design phase of (hybrid) service base systems since they specify the functionality and constraints of the system being constructed during design phase.

The WSDL-M2 approach [36], uses lexical matching to calculate linguistic similarities between concepts, and structural matching to evaluate the overall similarity between composite concepts. Moreover, this approach combines vector-space model techniques with synonyms based on WordNet and semantic relations of two concepts using WordNet. The structural matching is based on maximum weight bipartite matching problem in which the weights in the edges are denoted by the lexical similarities between the two elements associated with the edge. Several experiments have been conducted to analyze the work in terms of the use of vector-space model technique, use of vector-space model technique combined with WordNet, and use of semantics. The results achieved precision values between 46% to 100% for one set of services and 15% to 40% for another set of services. An extension of this work that achieved better precision results proposes the use of customizable hybrid approach in which a matching can be composed of different techniques with distinct weights depending on the needs of the organization, domain, or context. This approach suggests the use of compositions that mix several techniques (mixed), use techniques in different stages as a refinement for the next stage (cascade), or switch among techniques based on pre-defined criteria (switching).

Another approach that combines WordNet-based techniques and structure matching for service discovery has been proposed in [68]. Initial experiments of this approach in a collection of 19 services have achieved an average (a) precision of 48% and recall of 100% when using WorNet with vector-space model techniques, (b) precision of 20% and recall of 72% when using structure matching technique that takes into consideration the data types of the parameters of the operations, and (c) precision of 35.2% and recall of 81.8% when combining structure matching with semantic distance of the names of the operations and data types. Although the structural matching used in our framework is similar to the technique used in [68].

Other approaches have been proposed to support quality-of-services aware composition in which services are composed to contribute to achieve quality of service characteristics and support service level agreements [12][49][53]. Although existing approaches have contributed to assist service composition an approach that uses these compositions as part of the development of service-based systems has not been proposed.

Several approaches have also been proposed to support context awareness in service discovery [8][9][14][16][19][33][56]. In [19], context information is represented by key-value pairs attached to the edges of a graph representing service classifications. This approach does not integrate context information with behavioural and quality matching and, context information is stored explicitly in a service repository that must be updated following context changes. In [9], queries, services, and context information are expressed in ontologies. Context information in this approach can also be used as an implicit input to a service that is not explicitly provided by the user (e.g. user location). The approach in [8] focuses on user context information (e.g. location and time) and uses it to discover the most appropriate network operator before making phone calls. Other approaches focusing on discovery protocols in mobile computing also support context [14][56].

The work in [73] locates components based on context-aware browsing. In this approach, the interaction of software developers with the development environment is monitored and candidate components that match the development context based on signature matching are identified and presented to developers for browsing. The above context-aware approaches support simple conditions regarding context information in service discovery, do not fully integrate context with behavioral criteria in service discovery, and have limited applicability since they depend on the use of specific ontologies for the expression of context conditions.

Approaches for dynamic service composition, in which services are identified and aggregated during runtime in support of certain functional and quality characteristics of the desired systems have been proposed in [3][7][12][15][20][46][58].

Approaches for reactive adaptation of service composition were proposed in [3][6][31][42][70]. These approaches support changes in service composition based on pre-defined policies [6], self-healing of compositions based on detection of exceptions and repair using handlers [70], context-based adaptation of compositions using negotiation and repair actions [3]; and key performance indicator (KPI) analysis and the use of adaptation strategies related to the KPI fulfilment [31].

Exceptions to the reactive approaches are found in the works in [17][27][30][40][47][67]. The work in [17] is based on prediction of performance failures to support self-healing of compositions. The work uses semi-Markov models for performance predictions, service reliability model, and minimization in the number of service re-selection in case of changes. The decision to adapt is based on the performance of a single service, while our framework considers a group of related service operations in a composition, avoiding unnecessary changes to the composition. Moreover, the work in [17] does not support unavailability and malfunctioning of operations, services, and providers, as well as spatial correlations between these elements in a composition.

In [40] the PREvent approach is described to support prediction and prevention of SLA violations in service compositions based on event monitoring and machine learning techniques. The prediction of violations is calculated only at defined checkpoints in a composition based on regression classifiers prediction models.

The works in [27][47][67] advocate the use of testing to anticipate problems in service compositions and trigger adaptation requests. The approach in [67] supports identification of nine types of mismatches between services to be used in a composition and their requests based on pre-defined test cases. In [27][47] test cases are created during the deployment of service compositions and used to identify violations after a service is invoked for the first time. However, the creation of test cases is not an easy task and the work does not specify how to generate new test cases for a modified composition.

Although the above approaches have contributed to the problem of service discovery, none of them supports service discovery as part of the design process of service-base systems, as well as during execution time of these systems. There are no other approaches that focus on service discovery based on structural, behavioural, quality, and contextual descriptions of services at the same time, and approaches that support service requests based on structural, behavioural, quality, and contextual constraints of the system. In addition, no approach has advocated a proactive service discovery in which services are identified in parallel to the execution of service-based systems. In the following, we describe our service discovery framework with the above characteristics.

## 3 Service Discovery Framework

The static and dynamic service discovery framework supports both (i) design of service-based systems based on existing available services in service repositories and (ii) adaptation of service-based systems by replacing a participating service by another available service when necessary. More specifically, the framework supports the identification of services that provide functional and non-functional properties, as well as some extra constraints of service-based systems as specified by their requirements, architecture, design models, and workflow.

In the static phase, the framework assumes an iterative process in which queries are derived from service-based system models and discovered services are used to amend and re-formulate the models. In the dynamic phase, the framework supports a proactive push mode of query execution in which candidate services are identified in parallel to system execution based on subscribed services and queries. In both cases, the discovery process is based on similarity analysis and distance measures of service requests against service specifications. In the framework, service discovery queries include structural, behavioural, quality, and contextual aspects of services specified in a Service Discovery Query Language (SerDiQueL) [81] that we have developed. The work assumes service specifications represented as facets describing different aspects of the services, namely structural (represented in WSDL [71]), behavioural (represented in BPEL4WS [9]), quality (represented in XML format), and context (represented in XML format).

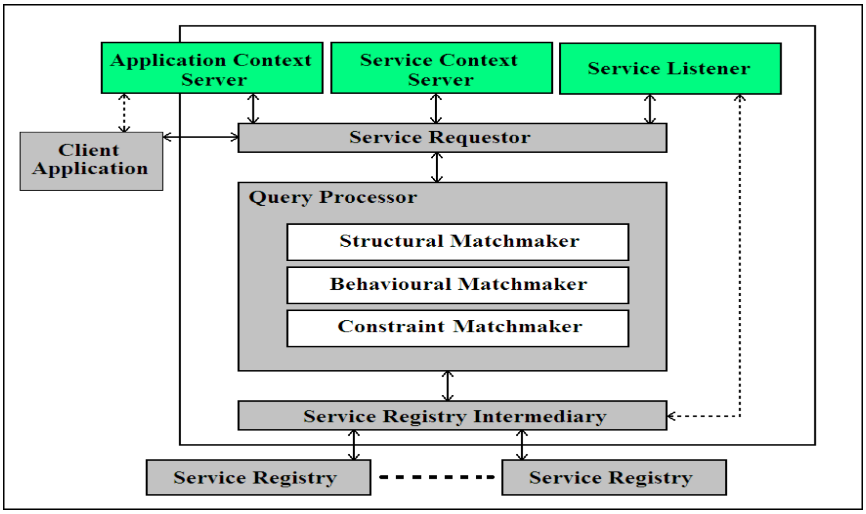


**Fig. 1.** Architecture Overview of the Framework

The architectural overview of the framework is shown in Figure 1. As shown in the figure, the main components of the framework are: (a) *client application*, that supports creation of service requests; (b) *service requestor*, that prepares service queries to be evaluated, organises the results of a query, returns the results to a client application, manages query subscriptions, and receives information from listeners; (c) *query processor*, that parses a query and evaluates the parts of a query against service specifications; (d) *service registry intermediary*, that provides an interface to access services from various registries; (e) *context servers*, that support acquisition of context information about services and application environment; (f) *service listener*, that sends to service requestor notifications of new available services or changes in services; and (g) *service registry*, that contains services specified by a set of facets (e.g., textual, structural, behavioural, quality, and contextual facets).

A more detailed view of the query processor component is shown in Figure 2. As shown in the figure, a query is received by the query processor and executed in terms of its *hard* and *soft* constraints. Hard constraints need to be satisfied by all candidate services and are used to filter services in the service repository that match the hard constraints. Soft constraints do not need to be satisfied by all candidate services and are used to rank services in relation to the queries together with the structural and behavioural aspects in a query. The matching of structural, behavioural, and soft constraint aspects in a query provide structural, behavioural, and soft constraint distances between service specifications and the query that are used in the computation of an overall distance between a service and query. The overall distance is used to rank candidate services with respect to a query.

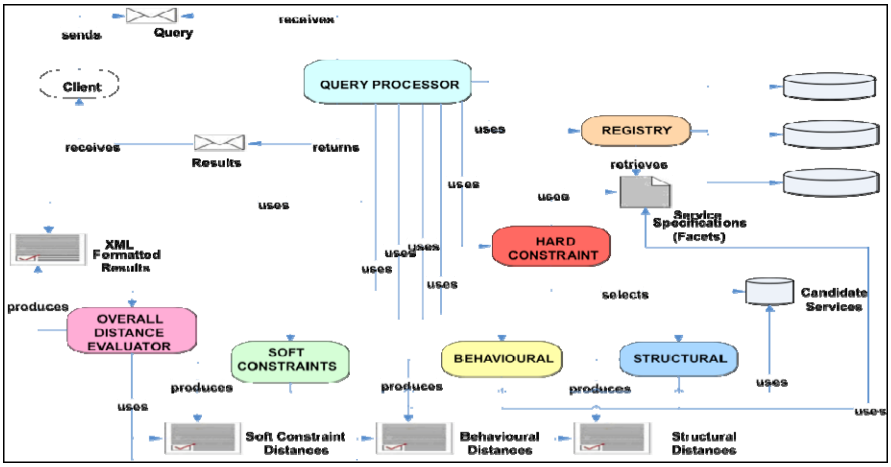


**Fig. 2.** Overview of the Query Processor Component

## 3.1 Service Discovery Query Language

In the framework, we use an XML-based service discovery query language called SerDiQueL [81] that allows for the specification of structural, behavioural, quality, and contextual characteristics of services to be discovered or service-based systems being developed. More specifically, in SerDiQueL a query may contain different criteria, namely: (i) structural, specified by models describing the interface of a required service; (iii) behavioural, specified by models describing the behavioural of a required service; and (iii) constraints, specifying extra conditions for the service to be discovered. These extra conditions may be concerned with structural, functional, quality, or contextual aspects of a service to be discovered that cannot be represented by interface or behavioural model descriptions used in the framework. For example, specification of the time or cost to execute a certain operation in a service, the receiver of a message, or the provider of a service.

A *contextual* constraint is concerned with information that changes dynamically during the operation of the service-based application and/or the services that the system deploys, while a *non-contextual* constraint is concerned with static information. The non-contextual constraints can be *hard* or *soft.* As mentioned above, a hard constraint must be satisfied by all discovered services for a query and are used to *filter* services that do not comply with them. The soft constraints do not need to be satisfied by all discovered services, but are used to *rank* candidate services.

Figure 3 presents the overall XML schema of SerDiQueL. As shown in the figure, a query specified in the language (ServiceQuery) has three elements representing structural, behavioural, and constraint sub-queries. The division of a query into these three sub-queries is to (i) allow the representation of these three types of information and (ii) support the representation of queries with arbitrary combinations of these types of information.

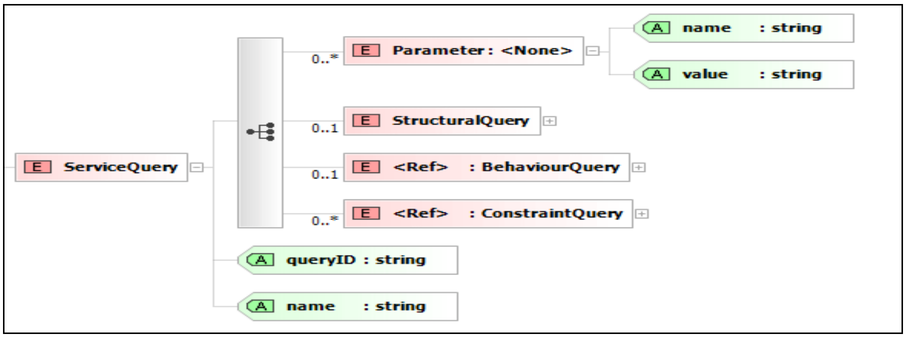


**Fig. 3.** Overview XML Schema for SerDiQueL

**Structural Sub-query**

The structural sub-query describes structural aspects of (i) a service-based system being developed (for static service discovery) or (ii) a service that needs to be replaced and is being used by a service-based system (for dynamic service discovery).

The description of structural aspects for case (i) is based on design models of this system. Our service discovery framework assumes design models expressed in UML class and sequence diagrams represented as XMI documents, due to the popularity of using UML for designing software systems in general, and service-based system in particular. However, the structural sub-query could be based on other types of design models representing the functionality of a system. In order to support the definition of structural aspects of a system under development based on UML models, we have developed a UML 2.0 profile [37]. The profile defines a set of stereotypes for different types of UML elements such as messages in sequence diagrams, or operations and classes defining the types of arguments in the messages.

For example, messages in a sequence diagram may be stereotyped as: (i) *query messages*, representing service operations needed in identified services; (ii) *context messages*, representing additional constraints for the query messages (e.g. if a context message has a parameter p1 with the same name as a parameter p2 of a query message, then the type of p1 should be taken as the type of p2); (iii) *bound messages*, representing concrete service operations that have been discovered in previous query executions.

In our framework, structural sub-queries for a service-based system being developed are automatically generated from the class and sequence diagrams of a system based on the selection of messages from the designer of the system. The description of structural aspects of a service-based system based on design models of these systems supports the representation of operations being searched in different services together with the representation of the input and output parameters of these operations and their respective data types. This is important to assist with the matching of structural aspects of the systems with structural aspects (interface descriptions) of available services.

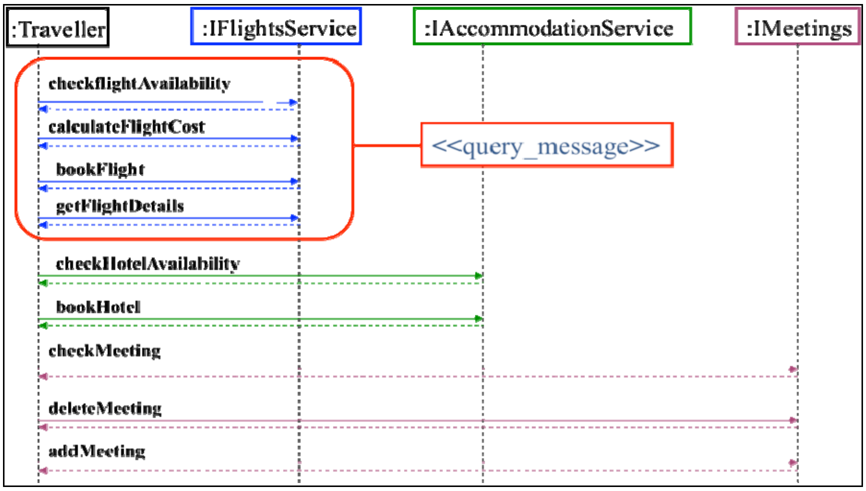


**Fig. 4.** Behavioural Model for *ConferenceTravel* SBS

As an example, consider a conference travel service-based system (*ConferenceTravel SBS*) being developed with the UML design models shown in Figures 4 and 5. Suppose the developer of this system interested in services that can support booking of flights for someone to attend a conference. In this case, a designer wants to find service operations that can provide implementation of messages as specified in the diagram in Figures 4 and 5 (<<query_message>>), with the respective classes representing the data types of the parameters of the query messages, namely:

*checkFlightAvailability (flight:FlightInfo): Boolean,*
*calculateFlightCost(flight:FlightInfo):Price,*
*bookFlight(flight:FlightInfo):String,*
*getFlightDetails(flightReference:String):FlightInfo*

The description of structural aspects for dynamic service discovery (case (ii)) is represented as the WSDL [71] specification of a service to be replaced. SerDiQueL supports a complete representation of the structural aspects of a service to be identified as interface descriptions. In the framework, the structural sub-queries for a service that needs to be replaced during execution time are automatic generated based on the notification that a service became malfunctioning, unavailable, or there have been changes in the characteristics of the service or in the context of the application environment.

In both static and dynamic service discovery, structural sub-queries are matched against interface descriptions of services specified as WSDL considering the names of the operations and the data types of the parameters of the operations.

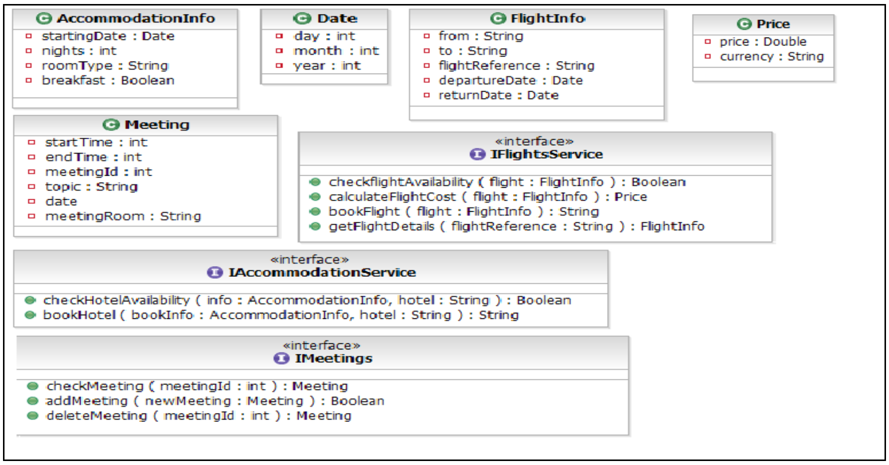

AccommodationInfo
startingDate : Date
nights : int
roomType : String
breakfast : Boolean
Date
day : int
month : int
year : int
FlightInfo
from : String
to : String
flightReference : String
departureDate : Date
returnDate : Date
Price
price : Double
currency : String
Meeting
startTime : int
endTime : int
meetingId : int
topic : String
date
meetingRoom : String
«interface»
IFlightsService
checkflightAvailability ( flight : FlightInfo ) : Boolean
calculateFlightCost ( flight : FlightInfo ) : Price
bookFlight ( flight : FlightInfo ) : String
getFlightDetails ( flightReference : String ) : FlightInfo
«interface»
IAccommodationService
checkHotelAvailability ( info : AccommodationInfo, hotel : String ) : Boolean
bookHotel ( bookInfo : AccommodationInfo, hotel : String ) : String
«interface»
IMeetings
checkMeeting ( meetingId : int ) : Meeting
addMeeting ( newMeeting : Meeting ) : Boolean
deleteMeeting ( meetingId : int ) : Meeting

**Fig. 5.** Structural Model for *ConferenceTravel* SBS

**Behavioural Sub-query**

The behavioural sub-query is based on temporal logic supporting the representation of behavioural aspects of required services. In particular, it supports the description of queries that verify (a) the existence of a certain functionality, or a sequence of functionalities, in a service specification; (b) the order in which certain functionalities should be executed by a service; (c) dependencies between functionalities; (d) pre-conditions; and (e) loops. Figure 6 shows a graphical representation of the SerDiQueL's XML schema for behavioural sub-queries. As shown in the figure, a behavioural sub-query is defined as (a) a single condition, a negated condition, or a conjunction of conditions, or (b) a sequence of expressions separated by logical operators.

In SerDiQueL, a behavioural sub-query is specified in terms of predicates, namely: (a) *Requires*, that describes service operations that need to exist in service specifications; (b) *GuaranteedMember*, that represents an element that needs to occur in all traces of execution; (c) *OccursBefore/OccursAfter*, that represents the order of occurrence of two member elements; (d) *Sequence*, that represents two or more members that must occur in a certain order; and (e) *Loop*, that represents a sequence of elements that are executed several times if a condition is satisfied.

As an example, consider the *ConferenceTravel* SBS shown in Figures 4 and 5. Figure 7 shows the description of the behaviour subquery in SerDiQueL. As shown in Figure 7, the *Requires* elements specify the requirement for the existence of operations checkFlightAvailability, calculateFlightCost, bookFlight, and getFlightDetails. The *Sequence* element specifies the order of these operations.

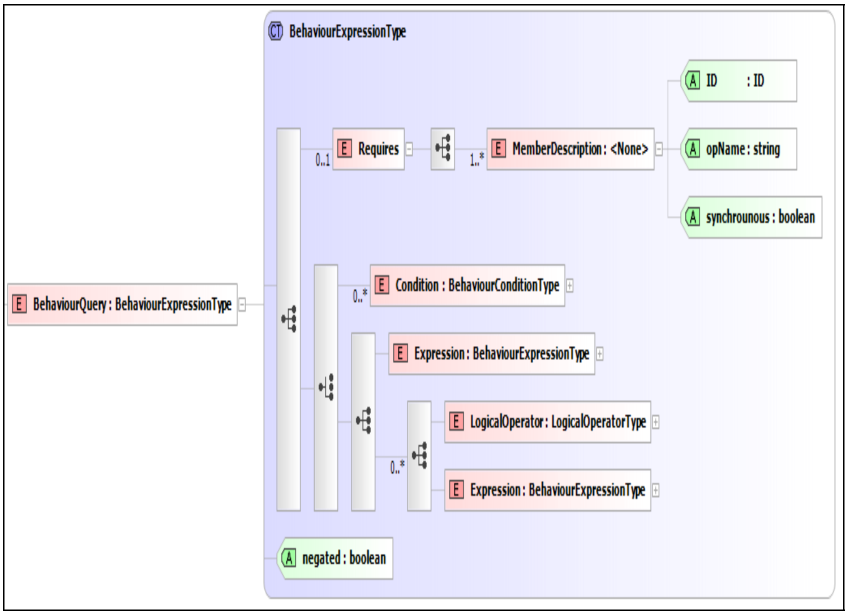

BehaviourExpressionType
BehaviourQuery : BehaviourExpressionType
Requires
0..1
1..*
MemberDescription : <None>
ID : ID
opName : string
synchrounous : boolean
Condition : BehaviourConditionType
0..*
Expression : BehaviourExpressionType
LogicalOperator : LogicalOperatorType
0..*
Expression : BehaviourExpressionType
negated : boolean

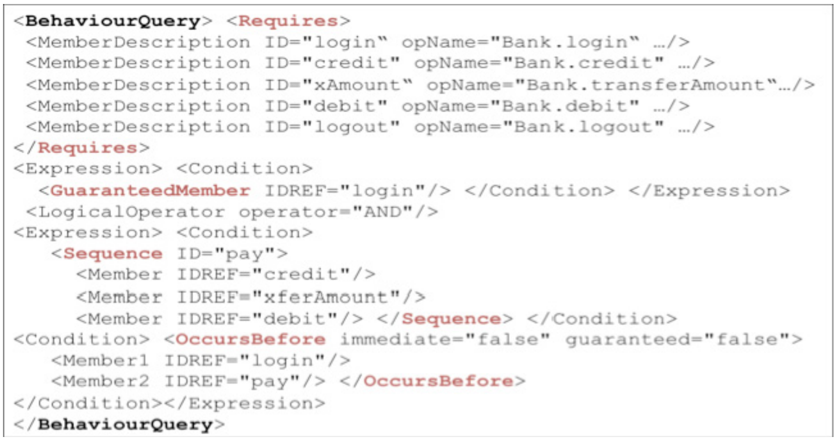
**Fig. 6.** XML Schema for Bheavioural Sub-query in SerDiQueL

```
<BehaviourQuery>

<Requires>
 <MemberDescription ID="avail" opName="checkflightavailability"…/>
 <MemberDescription ID="cost" opName="calculateflightcost" …/>
 <MemberDescription ID="book" opName="bookflight"…/>
 <MemberDescription ID="details" opName="getflightdetails" …/>
</Requires>

<Expression>
 <Condition>
   <Sequence ID="flight">
     <Member IDREF="avail"/>
     <Member IDREF="cost"/>
     <Member IDREF="book"/>
     <Member IDREF="details"/>
   </Sequence> </Condition>
 </Condition>
</Expression>
</BehaviourQuery>
```

**Fig. 7.** Example of Behavioural Sub-query in SerDiQueL

For another example of a behavioural sub-query, consider a dynamic service discovery situation in which a service that allows payments to be made by money transfer from a client's bank account in a service-based system becomes malfunctioning and needs to be replaced. Suppose that the operations below are executed in the current service that needs to be replaced, and that a user needs to be authenticated before accessing any functionality of the service. In this case, a service that supports the operations of the current services in the order below needs to be identified. Figure 8 shows the description of the behavioural sub-query in SerDiQueL for this example.

*login(userID:string, password:string):boolean*
*credit(accountId:string, amount:double):balance*
*transferAmount(fromAccId:int, toAccID:int, amount:double):boolean*
*debit(accountId:string, amount:double):balance*
*logout(userID:string):boolean*



```
<BehaviourQuery> <Requires>
 <MemberDescription ID="login" opName="Bank.login" .../>
 <MemberDescription ID="credit" opName="Bank.credit" .../>
 <MemberDescription ID="xAmount" opName="Bank.transferAmount".../>
 <MemberDescription ID="debit" opName="Bank.debit" .../>
 <MemberDescription ID="logout" opName="Bank.logout" .../>
</Requires>
<Expression> <Condition>
  <GuaranteedMember IDREF="login"/> </Condition> </Expression>
 <LogicalOperator operator="AND"/>
<Expression> <Condition>
   <Sequence ID="pay">
     <Member IDREF="credit"/>
     <Member IDREF="xferAmount"/>
     <Member IDREF="debit"/> </Sequence> </Condition>
<Condition> <OccursBefore immediate="false" guaranteed="false">
   <Member1 IDREF="login"/>
   <Member2 IDREF="pay"/> </OccursBefore>
</Condition></Expression>
</BehaviourQuery>
```

**Fig. 8.** Example of Behavioural Sub-query in SerDiQueL

As shown in Figure 8, the *Requires* elements specify the requirement for the existence of operations login, credit, transferAmount, debit, and logout. The *GuanranteeMember* element specifies that operation login needs to appear in all traces of execution in the service. The *Sequence* element specifies the order of operations credit, transferAmount, and debit. The *OccursBefore* element specifies that operation login needs to be executed before the operations in the sequence.

## Constraint Sub-query

The constraint sub-query describes different types of extra conditions that need to be fulfilled by a service. A constraint can be classified as contextual or non-contextual. The non-contextual constraints in a sub-query can be evaluated against any type of service specification (facet) in the service registries. The contextual constraints are evaluated against *context facets.* These context facets are associated with services and describe context information of the operations in these services. Context information is specified as context operations that are executed at run-time. The framework assumes the existence of context services that provide context information. Details of the context constraint matching are described in [64].

Figure 9 shows a graphical representation of SerDiQueL's XML schema for specifying constraints. As shown in the figure, a constraint sub-query is defined as a single logical expression, a negated logical expression, or a conjunction or disjunction of two or more logical expressions, combined by logical operators.

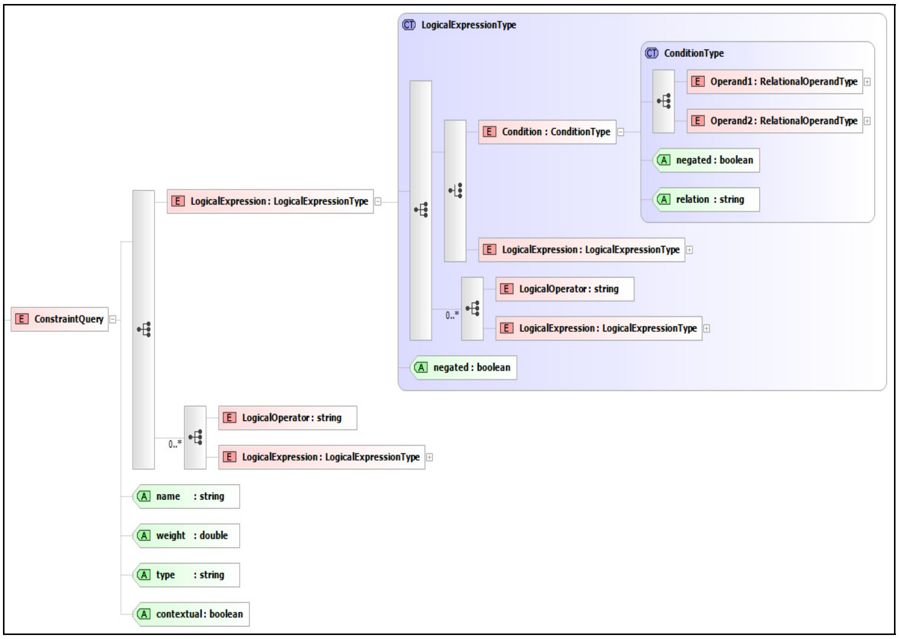
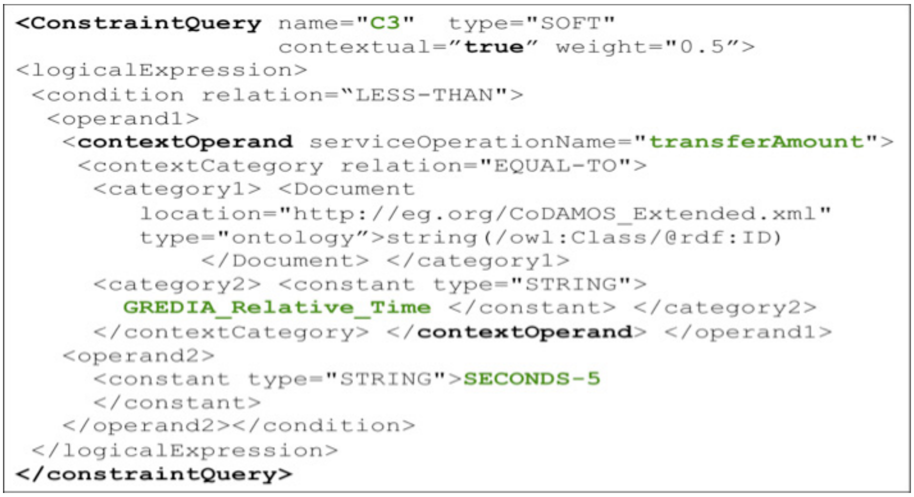

LogicalExpressionType
ConditionType
Operand1 : RelationalOperandType
Operand2 : RelationalOperandType
Condition : ConditionType
negated : boolean
relation : string
LogicalExpression : LogicalExpressionType
LogicalExpression : LogicalExpressionType
LogicalOperator : string
0..*
LogicalExpression : LogicalExpressionType
negated : boolean
ConstraintQuery
LogicalOperator : string
0..*
LogicalExpression : LogicalExpressionType
name : string
weight : double
type : string
contextual : boolean

**Fig. 9.** XML Schema for Constraint Sub-query in SerDiQueL

In order to illustrate, consider the example of the service that allows payments to be made by money transfer from a client's bank account in a service-based system that needs to be replaced. Assume two constraints for this service, namely (a) non-contextual constraint concerned with the fact that the service needs to be available 24 hours a day, and (b) contextual constraint specifying that the time required

```
<ConstraintQuery name="C2" type="SOFT"
                 contextual="false" weight="0.5">
 <logicalExpression> <condition relation="EQUAL-TO">
  <operand1>
   <nonContextOperand facetName="QoS" facetType="QoS">
     //QoSCharacteristic[Name="Availability"]/Metrics/
       Metric[Name="OpenTime"][Unit="Hours"]/MinValue
   </nonContextOperand> </operand1>
  <operand2>
   <constant type="STRING">00:00</constant>
  </operand2> </condition> </logicalExpression>
  <logicalOperator>and</logicalOperator>
  <logicalExpression> <condition relation="EQUAL-TO">
   <operand1>
    <nonContextOperand facetName="QoS" facetType="QoS">
      //QoSCharacteristic[Name="Availability"]/Metrics/
        Metric[Name="OpenTime"][Unit="Hours"]/MaxValue
    </nonContextOperand> </operand1>
   <operand2><constant type="STRING">24:00</constant> …
</ConstraintQuery>
```

**Fig. 10.** Example of Non-Contextual Constraint in SerDiQueL



```
<ConstraintQuery name="C3"  type="SOFT"
                 contextual="true" weight="0.5">
<logicalExpression>
 <condition relation="LESS-THAN">
  <operand1>
   <contextOperand serviceOperationName="transferAmount">
    <contextCategory relation="EQUAL-TO">
     <category1> <Document
        location="http://eg.org/CoDAMOS_Extended.xml"
        type="ontology">string(/owl:Class/@rdf:ID)
            </Document> </category1>
     <category2> <constant type="STRING">
       GREDIA_Relative_Time </constant> </category2>
     </contextCategory> </contextOperand> </operand1>
   <operand2>
     <constant type="STRING">SECONDS-5
     </constant>
   </operand2></condition>
 </logicalExpression>
</constraintQuery>
```

**Fig. 11.** Example of Contextual Constraint in SerDiQueL

to transfer money from a user's bank account should not be more than 5 seconds. Figure 10 shows the non-contextual constraint in SerDiQueL (case (a)) and Figure 11 shows the contextual constraint in SerDiQueL (case (b)).

Figure 10 shows that in facet QoS the minimal and maximum availability values should be between 00:00 and 24:00, respectively. Figure 11 specifies that any candidate service that can support transfer of money from a user's bank account (i.e., services that match operation *transferMoney*) needs to have a context operation classified in the category GREDIA_RELATIVE_TIME in ontology http://eg.org/CoDAMos_Extended.xml, and the result of executing this operation has to be less than SECONDS-5 for this service to be accepted.

## 3.2 Service Discovery Execution

As explained above, for both static and dynamic service discovery, matchings between queries and service specifications are executed in a two-phase process. In the first phase, the query processor searches service registries in order to identify services that satisfy the hard constraints of a query based on exact matchings (*filtering* phase). In the second phase, candidate services identified in the filtering phase are matched against the structural, behavioural, and constraints sub-queries, and the best candidate services for the query are identified (*ranking* phase). The ranking phase is executed based on the computation of partial distances, namely *structural*, *behavioural*, soft *non-contextual*, and *contextual* distances when applicable. The partial distances computed between services and a query are aggregated into an overall distance which is then used to select the best services for a query. The best services for a query are selected based on an instance of the assignment problem [54].

There may be some differences in the execution process of a query. These differences are due to the lack of hard, behavioural, and soft contextual and non-contextual constraints in a query, or any combinations of these constraints. In cases where there are no hard constraints in a query, the filtering phase is not executed and partial distances are calculated for all the services in the registries. Also if there are no behavioral or soft constraints in a query, the computation of the relevant partial distances is bypassed and the overall distance is computed by using only the partial distances of the types of constraints specified in a query. Note that structural constraints are always present in a query and, therefore, distances based on these constraints are always calculated.

Other differences in the execution process of a query exist in the case when a query is to be performed to support static or dynamic service discovery. During static service discovery, the structural and behavioural matching processes are flexible allowing the identification of services whose structure and behaviour characteristics have different degrees of similarity to those of a required service, and behaviour matchings with alternative or missing mappings between a required service and an existing service. The flexibility and alternative/missing mappings contribute to the reformulation of the design models of the service-based system under development and to the design of service-based systems based on characteristics of existing services. During dynamic service discovery, the structural and behavioural matching process requires matches with services that can be used to substitute services in an already deployed system. Therefore, in this case, it is necessary to guarantee that the input information for invoking the service that needs to be replaced in the system covers the input information needed by a candidate service, and that the information produced by the candidate service covers the information expected from the service to be replaced. It is also necessary to preserve the order of the different functionalities to be executed by a service.

The *structural* matching between a query and a service is performed by comparing (i) the signatures of query messages in the structural model of a service-based system against the signatures of the operations of WSDL specifications of candidate services, during the design of service-based systems; or (ii) the signature of the operations in the WSDL specification of a service that needs to be replaced in a service-based system against the signature of the operations of WSDL specifications of candidate services, during dynamic service discovery. In both cases, the structural matching is based on the comparison of graphs representing the data types of the parameters of the operations and the linguistic distances of the names of operations and parameters. Details of the comparison of graphs and structural distances can be found in [37][38][63][65][77][79][80].

The *behavioural* matching between a query and a service is performed by comparing the behavioural specification of the services and the behavioural sub-query. In this case, the behavioural specifications of the service and the behavioural sub-query are converted into state machine models and distances between these state machines are calculated based on similarities of these state machines. Details about the construction and comparison of the state machines can be found in [37][38][63][65][77][79][80].

The soft constraint matching (*contextual* and *non-contextual*) between a query and a service is performed by analysing the conditions in the constraint part of a query against service specifications. Details of this matching can be found in [64][80].

**Static Service Discovery.** In the case of static service discovery, the set of candidate services with their respective distances are presented to the designer of the system. The designer selects a service that is used to reformulate the design model of the system being constructed and trigger new service discovery execution. As an example, consider the *ConferenceTravel* scenario and the query shown in Figures 4 and 5. Table 1 shows the result of this query for the best three candidate services in a service registry with 70 services and 212 operations, including services to support travel arrangements. As shown in Table 1, the best match is for service AAirline1, with its respective operations. Figure 12 shows the design model reformulated with service AAirline1 (instead of placeholder *:IFlightService* as in the case of Figure 4), and its respective operations that are now <<bound_messages>>.

**Table 1.** Results of *ConferenceTravel* Query

| Operation: **checkFlightAvailability** | | | |
|---|---|---|---|
| **Service** | **Operation** | **Distance (struct)** | **Distance (overall)** |
| AAirline1 | checkFlightAvailability | 0.0644 | 0.3548 |
| DeltaAirline2 | checkFlightAvailability | 0.0733 | 0.3578 |
| DeltaAirline | FlightAvailability | 0.1918 | 0.3973 |
| AAirline1 | getFlightDetails | 0.2199 | 0.4067 |
| | | | |
| Operation: **calculateFlightCost** | | | |
| **Service** | **Operation** | **Distance (struct)** | **Distance (overall)** |
| AAirline1 | calculateFlightPrice | 0.0882 | 0.3627 |
| DeltaAirline2 | calculateTicketCost | 0.1418 | 0.3806 |
| AAirline2 | FlightAvailability | 0.1916 | 0.3972 |
| AAirline1 | checkFlightAvailability | 0.2019 | 0.4006 |
| | | | |
| Operation: **bookFlight** | | | |
| **Service** | **Operation** | **Distance (struct)** | **Distance (overall)** |
| AAirline1 | bookFlight | 0.0672 | 0.3557 |
| AAirline1 | calculateFlightPrice | 0.1759 | 0.3919 |
| AAirline1 | checkFlightAvailability | 0.1805 | 0.3935 |
| DeltaAirline2 | FlightAvailability | 0.1848 | 0.3949 |
| | | | |
| Operation: **getFlightDetails** | | | |
| AAirline1 | getFlightDetails | 0.0816 | 0.3605 |
| AAirline1 | bookFlight | 0.1869 | 0.3956 |
| AAgenda | getAddress | 0.1919 | 0.3973 |
| AAirline1 | calculateFlightPrice | 0.1925 | 0.3975 |

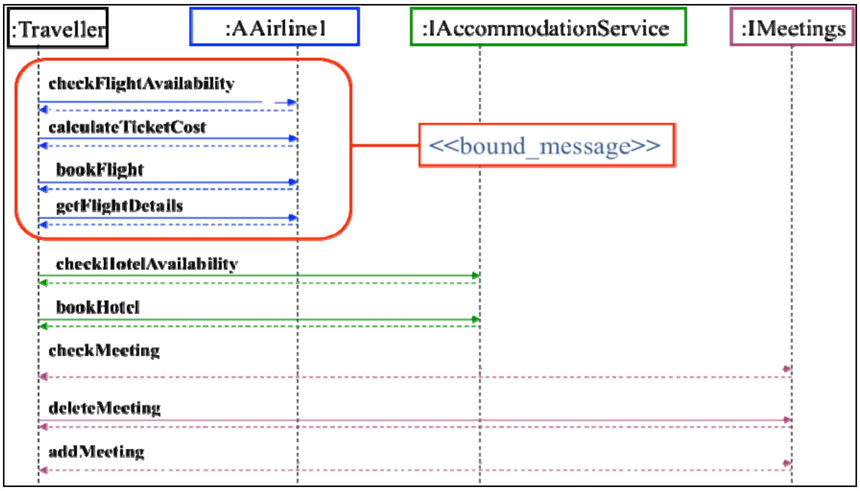


**Fig. 12.** Reformulated Behavioural Model for *ConferenceTravel* SBS

**Dynamic Service Discovery.** For dynamic service discovery, the framework supports the identification of services in both pull and proactive push modes of query execution due to (a) unavailability or malfunctioning of services, (b) changes in the context of services or the service-based system environment, (c) changes in the structural or behavioural characteristics of services, and (d) emergency of a new services that are better than the services already deployed in the service-based system.

In the *pull mode* of query execution, the query processor executes a query and maintains services with distances from the query that does not exceed a certain threshold. The *proactive push mode* of query execution consists of identifying a set of replacement services for services already deployed in the service-based system, based on subscribed services and queries during the execution time of service-based system, and using these replacement services when necessary due to circumstances (a) to (d) above. For each subscribed service, the framework identifies an up-to-date set of candidate services. This set is maintained in parallel to the execution of a service-based system and includes only services whose overall distance from the query subscribed for a certain service does not exceed a given threshold, in ascending order. When necessary, a replacement service for a deployed service in the service-based system is used from the set of up-to-date candidate services. Details about the matching process to create the up-to-date set of candidate services and how the approach deals with each case (a) to (d) above is described in [18][79][80].

In the framework, the replacement of a service in a service-based system may not take place right after modifications occur in the set of candidate services for the service. This is due to the fact that an immediate replacement might be inappropriate. For example, consider the situation in which a service *S* is executing some transactions on behalf of the application, at the time when a new better service is found. The decision to stop the execution of the application in order to replace a service for which an alternative service has been found is based on *replacement policies*.

The replacement policies consider the position of a service S that may need to be replaced with respect to the current execution point of the service-based system and the situations that trigger the need for changes in the system. It tries to avoid making changes that can be executed in the future. More specifically, there are three different positions that needs to be considered with respect to a service S that needs to be replaced, namely: (a) *not_in_path*, when service S in not in the current execution path of the system, i.e., S appears in a different branch of the system's execution path or before the current point in the execution path; (b) *current*, when service S is in the current execution point of the system; and (c) *next_in_path*, when service S is in the current execution path of the system, and will be invoked some time in the future. Depending on the positions of service S, the replacement policies verify if (i) changes are required to be performed so that the system can continue its operations; (ii) changes can wait to be performed after the current execution of the system; and (ii) no changes are required. Details about the replacement policies used in the framework can be found in [44].

## 4 Extensions

The framework assumes that services will be described in service repositories by different facets such as structural, behavioral, quality, or contextual characteristics. However, it is not possible to assume that services will always be described in terms of all the above characteristics. Current service registries guarantee the existence of structural descriptions of services, typically in the form of WSDL specifications [29][48][51], even though it is necessary to identify services in terms of its other characteristics to allow a more precise service selection.

In order to support the above need and the lack of behavioural service descriptions in service repositories, we have extended the framework with a monitor component that verifies the satisfiability of behavioural and contextual properties of the services against messages exchanged between a service-based system and the services deployed by the system. Details about this extension are described in [45]. The monitor is based on the previous work presented in [43][45] supporting monitoring of behavioural and contextual characteristics of service-based systems.

Figure 13 shows the architecture of the framework with the extension. As shown in the figure, the monitor component is responsible for identifying services that become unavailable, changes in the behavioural or contextual characteristics of the services deployed in a service-based system and their replacement candidate services, and changes in the context of the service-based system. The monitor is also responsible for verifying whether the behavioural and contextual conditions specified in service discovery queries are satisfied by services.

In the framework, the behavioural properties to be monitored are derived from translation of SerDiQueL queries into event calculus (EC) [61] in terms of events and fluents. The satisfiability of properties by the services is verified by the analyzer component of the monitor based on invocations of the services by the service client component and the events collected for these services by the event collector component. The monitor deploys a service client for each service that needs to be monitored. The service client component is responsible for the invocation of services and the generation of runtime events intercepted by the event collector. The event collector component is responsible to gather runtime information during the execution of services and to make this information available during the verification of the different properties. The services that do not satisfy the behavioural properties are not considered as possible candidate services during the computation of the ranking stage.

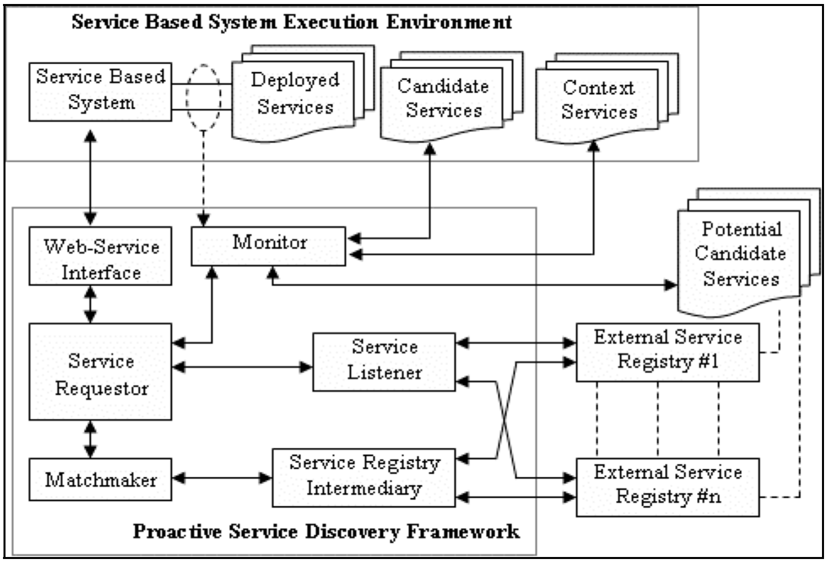

Service Based System Execution Environment
Service Based System
Deployed Services
Candidate Services
Context Services
Web-Service Interface
Monitor
Potential Candidate Services
Service Requestor
Service Listener
External Service Registry #1
Matchmaker
Service Registry Intermediary
External Service Registry #n
Proactive Service Discovery Framework

**Fig. 13.** Extension of the Service Discovery Framework Architecture

## 5 Conclusion and Final Remarks

In this tutorial paper we have provided an overview of a service discovery framework that supports both static and dynamic identification of services represented by structural, behavioural, quality, and contextual characteristics. The framework supports a service discovery query language that can represent complex queries to be matched against different types of service specifications. The static service discovery process of the framework can be used to assist with the development of service-based systems in which services matching some characteristics of the system being developed are identified and used to amend or reformulate the design models of the systems being developed. The dynamic service discovery process of the framework is used to support the replacement of a service in a service-based system during runtime execution of the system due to several situations. The dynamic discovery process is based on a proactive approach in which candidate services for services deployed in service-based systems are identified in parallel to the execution of the systems.

The framework has been evaluated in several scenarios for different queries, considering different objectives such as precision, recall, and performance measurements of the matching process. The framework has also been evaluated to verify the advantages of using the proactive push mode of query execution during dynamic service discovery with respect to a reactive approach, and using the monitor component when it is not possible to guarantee the existence of behavioural service specifications. The results of these evaluations have been positive and are described in [37][45][65][79][80].

Currently we are extending the framework to support several points: (a) service discovery based on service reputation and trust aspects; (b) service-based system adaptation triggered by changes in business activities, user requirements, and quality of services; (c) service discovery considering behavioural composition of candidate services; and (d) verification of service-based system design models after amending these models with the models of discovered services.

## References

[1] Agarwal, S., Studer, R.: Automatic Matchmaking of Web Services. In: International Conference on Web Services, pp. 45–54. IEEE Press (2006)

[2] Aggarwal, R., Verma, K., Miller, J., Milnor, W.: Constraint Driven Web Service Composition in METEOR-S. In: 2004 IEEE International Conference on Services Computing, pp. 23–30. IEEE Press, New York (2004)

[3] Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software 24, 39–46 (2007)

[4] Albert, P., Henocque, L., Kleiner, M.: Configuration-Based Workflow Composition. In: 2005 IEEE International Conference on Web Services, pp. 285–292. IEEE Computer Society (2005)

[5] Baresi, L., Di Nitto, E., Ghezzi, C.: Inconsistency and Ephemerality in a World of e-Services. In: Workshop on Requirements Engineering for Open Systems, Requirements Engineering Conference (2003)

[6] Baresi, L., Di Nitto, E., Ghezzi, C., Guinea, S.: A Framework for the Deployment of Adaptable Web Service Compositions. Service Oriented Computing and Applications Journal 1, 75–91 (2007)

[7] Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware Web Service Composition. In: 2006 IEEE International Conference on Web Services, pp. 72–82. IEEE Computer Society (2006)

[8] Bormann, F., Flake, S.: Towards Context-Aware Service Discovery: A Case Study for a new Advice of Charge Service. In: 14th IST Mobile and Wireless Communications Summit, Dresden (2005)

[9] BPEL4WS. Business process Execution language for Web Services 1.1, `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`

[10] Broens, T., Pokraev, S., van Sinderen, M., Koolwaaij, J., Dockhorn Costa, P.: Context-Aware, Ontology-Based Service Discovery. In: Markopoulos, P., Eggen, B., Aarts, E., Crowley, J.L. (eds.) EUSAI 2004. LNCS, vol. 3295, pp. 72–83. Springer, Heidelberg (2004)

[11] Canfora, G., Di Penta, M., Esposito, R., Perfetto, F., Villani, M.L.: Service Composition (re)Binding Driven by Application–Specific QoS. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 141–152. Springer, Heidelberg (2006)
[12] Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: QoS-Aware Replanning of Composite Web Services. In: 2005 IEEE International Conference on Web Services, pp. 121–129. IEEE Computer Society (2005)
[13] Cardoso, J., Sheth, A.: Semantic e-Workflow Composition. Journal of Intelligent Information Systems 21, 191–225 (2003)
[14] Lee, C., Helal, S.: Context Attributes: An Approach to Enable Context-awareness for Service Discovery. In: 2003 Symposium on Applications and the Internet, pp. 22–30. IEEE Computer Society (2003)
[15] Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
[16] Cuddy, S., Katchabaw, M., Lutfiyya, H.: Context-Aware Service Selection Based on Dynamic and Static Service Attributes. In: IEEE International Conference on Wireless And Mobile Computing, Networking and Communications, pp. 13–20. IEEE Press, New York (2005)
[17] Dai, Y., Yang, L., Zhang, B.: QoS-Driven Self-Healing Web Service Composition Based on Performance Prediction. Journal of Computer Science and Technology 24, 250–261 (2009)
[18] Dolley, J., Zisman, A., Spanoudakis, G.: Runtime Service Discovery for Grid Applications. In: Bessis, N. (ed.) Grid Technology for Maximizing Collaborative Decision Management and Support: Advancing Effective Virtual Organizations, pp. 212–234. IGI Global (2009)
[19] Doulkeridis, C., Loutas, N., Vazirgiannis, M.: A System Architecture for Context-Aware Service Discovery. Electr. Notes Theoretical Computer Science 146, 101–116 (2006)
[20] Fujii, K., Suda, T.: Semantics-based Dynamic Web Service Composition. Int. Journal of Cooperative Inf. Systems 15, 293–324 (2006)
[21] Garofalakis, J., Panagis, Y., Sakkopoulos, E., Tsakalidis, A.: Web Service Discovery Mechanisms: Looking for a Needle in a Haystack. In: International Workshop on Web Engineering, Hypermedia Development and Web Engineering Principles and Techniques: Put Them in Use, in Conjunction with ACM Hypertext 2004, Santa Cruz (August 2004)
[22] GREDIA, `http://www.gredia.eu`
[23] Grirori, D., Corrales, J.C., Bouzeghoube, M.: Behavioral Matching for Service Retrieval, International Conference on Web Services. In: International Conference on Web Services, 2006, pp. 145–152. IEEE Computer Society (2006)
[24] Hall, R.J., Zisman, A.: Behavioral Models as Service Descriptions. In: 2nd Int. Conference on Service Oriented Computing, pp. 163–172. ACM (2004)
[25] Hall, R.J., Zisman, A.: Validating Personal Requirements by Assisted Symbolic Behavior Browsing. In: 19th IEEE International Conference on Automated Software Engineering, pp. 56–66. IEEE Press, New York (2004)
[26] Hausmann, J.H., Heckel, R., Lohmann, M.: Model-based Discovery of Web Services. In: IEEE International Conference on Web Services, pp. 324–331. IEEE Press, New York (2004)

[27] Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: A Framework for Proactive Self-adaptation of Service-Based Applications Based on Online Testing. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 122–133. Springer, Heidelberg (2008)

[28] Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: The Making of a Web Ontology Language. Journal of Web Semantics 1, 7–26 (2003)

[29] Hoschek, W.: The Web Service Discovery Architecture. In: 2002 ACM/IEEE Conference on Supercomputing. ACM (2002)

[30] Jun, N., Bin, Z., Xiamgyu, Z., Zhiliang, Z., Dancheng, L.: Two-Stage Adaptation for Dependable Service-Oriented System. In: 2010 International Conference on Service Sciences, pp. 143–147. IEEE Computer Society (2010)

[31] Kazhamiakin, R., Wetzstein, B., Karastoyanova, D., Pistore, M., Leymann, F.: Adaptation of Service-Based Applications Based on Process Quality Factor Analysis. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 395–404. Springer, Heidelberg (2010)

[32] Keller, U., Lara, R., Lausen, H., Polleres, A., Fensel, D.: Automatic Location of Services. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 1–16. Springer, Heidelberg (2005)

[33] Khedr, M., Karmouch, A.: Enhancing Service Discovery with Context Information. In: ITS 2002 (2002)

[34] Klein, M., Bernstein, A.: Toward High-Precision Service Retrieval. IEEE Internet Computing 8, 30–36 (2004)

[35] Klusch, M., Fries, B., Sycara, K.: Automated Semantic Web Service Discovery with OWLS-MX. In: 5th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS), pp. 915–922. ACM (2006)

[36] Kokash, N., van den Heuvel, W.-J., D'Andrea, V.: Leveraging Web Services Discovery with Customizable Hybrid Matching. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 522–528. Springer, Heidelberg (2006)

[37] Kozlenkov, A., Spanoudakis, G., Zisman, A., Fasoulas, F., Sanchez, F.: Architecture-driven Service Discovery for Service Centric Systems. International Journal of Web Services Research (Special Issue on Service Engineering) 4, 81–112 (2007)

[38] Kozlenkov, A., Spanoudakis, G., Zisman, A., Fasoulas, F., Sanchez, F.: A Framework for Architecture Driven Service Discovery. In: 2006 International Workshop on Service Oriented Software Engineering (IW-SOSE 2006), in Conjunction with ICSE 2006, pp. 67–73. ACM (2006)

[39] Kramler, G., Kapsammer, E., Kappel, G., Retschitzegger, W.: Towards Using UML 2 for Modelling Web Service Collaboration Protocols. In: 1st Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA 2005), pp. 227–238. Springer-Verlag London Limited (2006)

[40] Leitner, P., Michlmayr, A., Rosenber, F., Dustdar, S.: Monitoring, Prediction and Prevention of SLA Violations in Composite Services. In: 2010 IEEE International Conference on Web Services, pp. 369–376. IEEE Computer Society, Washington (2010)

[41] Li, L., Horrock, I.: A Software Framework for Matchmaking based on Semantic Web Technology. In: 12th Int. World Wide Web Conference - Workshop on E-Services and the Semantic Web, pp. 331–339. ACM (2003)

[42] Lin, K.J., Zhang, J., Zhai, Y., Xu, B.: The Design and Implementation of Service Process Reconfiguration with End-to-end QoS Constraints in SOA. Journal of Service Oriented Computing and Applications 4, 157–168 (2010)

[43] Mahbub, K., Spanoudakis, G.: Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In: 2005 IEEE International Conference on Web Services, pp. 257–265. IEEE Computer Society (2005)

[44] Mahbub, K., Zisman, A.: Replacement Policies for Service-Based Systems. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 345–357. Springer, Heidelberg (2010)

[45] Mahbub, K., Spanoudakis, G., Zisman, A.: A Monitoring Approach for Runtime Service Discovery. Automated Software Engineering Journal 18, 117–161 (2011)

[46] Pernici, B. (ed.): MAIS Project. Mobile Information Systems – Infrastructure and Design for Flexibility and Adaptability. Springer (2006)

[47] Metzer, A., Sammodi, O., Pohl, K., Rzepka, M.: Towards Pro-active Adaptation with Confidence Augumenting Service Monitoring with Online Testing. In: 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 20–28. ACM (2010)

[48] Mikhaiel, R., Stroulia, E.: Examining Usage Protocols for Service Discovery. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 496–502. Springer, Heidelberg (2006)

[49] Nguyen, X.T., Kowalczyk, R., Han, J.: Using Dynamic Asynchronous Aggregate Search for Quality Guarantees of Multiple Web Services Compositions. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 129–140. Springer, Heidelberg (2006)

[50] OWL-S (2003), http://www.daml.org/services/owl-s/1.0

[51] Pantazoglou, M., Tsalgatidou, A., Athanasopoulos, G.: Discovering Web Services and JXTA Peer-to-Peer Services in a Unified Manner. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 104–115. Springer, Heidelberg (2006)

[52] Pantazoglou, M., Tsalgatidou, A., Spanoudakis, G.: Behavior-aware, Unified Service Discovery. In: Service-Oriented Computing: a Look at the Inside Workshop, SOC@Inside 2007, Co-located with ICSOC 2007 (2007)

[53] De Paoli, F., Lulli, G., Maurino, A.: Design of Quality-Based Composite Web Services. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 153–164. Springer, Heidelberg (2006)

[54] Papadimitriou, C., Steiglitz, K.: Combinatorial Optimisation: Algorithms and Complexity. Dover Publications (1998)

[55] Papazoglou, M., Aiello, M., Pistore, M., Yang, J.: XSRL: A Request Language for web services. In: IEEE Internet Computing(2002), `http://infolab.uvt.nl/pub/papazogloump-2002-61.pdf`

[56] Pawar, P., Tokmakoff, A.: Ontology-based Context-aware service discovery for pervasive environments. In: 1st IEEE International Workshop on Services Integration in Pervasive Environments (SIPE 2006), Co-located with IEEE ICPS 2006, pp. 1–7. IEEE Computer Society (2006)

[57] Di Penta, M., Esposito, R., Villani, M.L., Codato, R., Colombo, M., Di Nitto, E.: WS Binder: a Framework to enable Dynamic Binding of Composite Web Services. In: 2006 International Workshop on Service-Oriented Software Engineering (SOSE 2006), pp. 74–80. ACM, New York (2006)

[58] Pistore, M., Marconi, A., Bertolini, P., Traverso, P.: Automated Composition of Web Services by Planning at the Knowledge Level. In: Kaelbling, L.P., Saffiotti, A. (eds.) Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 1252–1259. Professional Book Center (2005)

[59] S-CUBE, the European Network of Excellence in Software Services and Systems, `http://www.s-cube-network.eu/`

[60] SeCSE, Service Centric System Engineering, `http://www.secse-project.eu/`

[61] Shanahan, M.: The Event Calculus Explained. In: Wooldridge, M.J., Veloso, M.M. (eds.) Artificial Intelligence Today. LNCS (LNAI), vol. 1600, pp. 409–430. Springer, Heidelberg (1999)

[62] Shen, Z., Su, J.: Web Service Discovery Based on Behavior Signature. In: IEEE International Conference on Services Computing (SCC 2005), pp. 279–286. IEEE Computer Society (2005)

[63] Spanoudakis, G., Zisman, A., Kozlenkov, A.: A Service Discovery Framework for Service Centric Systems. In: IEEE International Conference on Services Computing (SCC 2005), pp. 251–259. IEEE Computer Society (2005)

[64] Spanoudakis, G., Mahbub, K., Zisman, A.: A Platform for Context-Aware Runtime Service Discovery. In: 2007 IEEE International Conference on Web Services, pp. 233–240. IEEE Computer Society (2007)

[65] Spanoudakis, G., Zisman, A.: Discovering Services during Service-based System Design using UML. IEEE Transactions of Software Engineering 36, 371–389 (2010)

[66] Spanoudakis, G., Zisman, A.: Designing and Adapting Service-based Systems: A Service Discovery Framework. In: Dustdar, S., Li, F. (eds.) Service Engineering: European Research Results, pp. 261–298. Springer (2010) ISBN 978-3-7091-0414-9

[67] Tosi, D., Denaro, G., Pezzè, M.: Towards Autonomic Service-Oriented Applications. International Journal of Autonomic Computing (IJAC) 1, 58–80 (2009)

[68] Wang, Y., Stroulia, E.: Semantic Structure Matching for Assessing Web-Service Similarity. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 194–207. Springer, Heidelberg (2003)

[69] Wang, X., Vitvar, T., Kerrigan, M., Toma, I.: A QoS-Aware Selection Model for Semantic Web Services. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 390–401. Springer, Heidelberg (2006)

[70] WSDiamond, Web Services - DIAgnosability, Monitoring and Diagnosis, `http://wsdiamond.di.unito.it`

[71] WSDL, Web Services Description Language, `http://www.w3.org/TR/wsdl`

[72] Wu, J., Wu, Z.: Similarity-based Web Service Matchmaking. In: IEEE International Conference on Services Computing, pp. 287–294. IEEE Computer Society (2005)

[73] Ye, Y., Fischer, G.: Context-Aware Browsing of Large Component Repositories. In: IEEE 16th Int. Conference on Automated Software Engineering (ASE), pp. 99–106. IEEE Computer Society (2001)

[74] Yunyao, V., Yanh, H., Jagadish, H.: NaLIX: an Interactive Natural Language Interface for Querying XML. In: Özcan, F. (ed.) ACM SIGMOD International Conference on Management of Data, pp. 900–902. ACM (2005)

[75] Zachos, K., Zhu, X., Maiden, N., Jones, S.: Seamlessly Integrating Service Discovery into UML Requirements Processes. In: 2006 International Workshop of Service Oriented Software Engineering (IW-SOSE), in Conjunction with ICSE 2006, pp. 60–66. ACM, New York (2006)

[76] Zaremski, A.M., Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology 4, 146–170 (1995)

[77] Zisman, A., Spanoudakis, G.: UML-Based Service Discovery Framework. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 402–414. Springer, Heidelberg (2006)

[78] Zisman, A., Mahbub, K., Spanoudakis, G.: A Service Discovery Framework based on Linear Composition. In: 2007 IEEE International Conference on Services Computing, pp. 536–543. IEEE Computer Society (2007)
[79] Zisman, A., Spanoudakis, G., Dooley, J.: Proactive Runtime Service Discovery. In: 2008 IEEE International Conference on Services Computing, pp. 237–245. IEEE Computer Society (2008)
[80] Zisman, A., Spanoudakis, G., Dooley, J.: A Framework for Dynamic Service Discovery. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 158–167. ACM (2008)
[81] Zisman, A., Spanoudakis, G., Dooley, J.: A Query Language for Service Discovery. In: 4th International Conference on Software and Data Technologies (2009)

# People-Centered Software Development: An Overview of Agile Methodologies

Frank Maurer and Theodore D. Hellmann

The University of Calgary, Department of Computer Science,
2500 University Drive NW, Calgary, Alberta, Canada
{frank.maurer,tdhellma}@ucalgary.ca

**Abstract.** This chapter gives an overview of agile software development processes and techniques. The first part of the chapter covers the major agile project management techniques with a focus on project planning. Iteration planning and interaction design approaches are given special focus. The second part of the chapter covers agile quality assurance with a focus on test-driven development and the state space of testing. Current problems in agile testing, including measuring test quality and testing applications with large state spaces, are discussed.

**Keywords:** Agile Methods, Agile Project Management, Agile Interaction Design, Test-Driven Development, State Space Testing.

## 1 Introduction

Software development is a complex undertaking that poses substantial challenges to teams in industry. In the 1980ies, companies tried to use Computer-Aided Software Engineering (CASE) tools to increase the efficiency of software development processes. The core idea was to use graphical notations to describe the functionality of a software system on an abstract level and then generate (most of) the code from it. However, the success of these approaches was limited and software teams nowadays still write code manually.

In the 1990ies, software developers were a scarce resource and companies focused on improving the development process to optimize their development efforts. Software process improvement (SPI) initiatives following CMMI, ISO 900x or SPICE ideas were commonplace. SPI approaches basically require an organization to define the steps and outcomes of each development step and then ensure that all teams are following these best practices: document what you do and do what is documented. As a side effect of the process definition, organizations often adopted Tayloristic[1] waterfall processes where steps in the process corresponded to roles in the organization and handoffs between steps happened in the form of documents. Unfortunately, many SPI

[1] In his seminal 1911 book "The Principles of Scientific Management", Frederick Taylor discussed repeatable manufacturing processes with a strong division of labor and a separation between manufacturing and engineering work.

implementations resulted in heavyweight, document-centric processes that created a substantial overhead for software development teams.

Agile processes are trying to swing the pendulum back. Proponents of agile methods ask the questions: how can we refocus projects on the bare minimum required to make software development effective and efficient? What does a software development team really have to do to create business value?

Agile methods came to the forefront of the discussion in the software development community in the late 1990ies and have been widely adopted since then. Initially, in the late 1990ies and early 2000s, teams started their journey by using ideas from extreme programming (XP) to improve their engineering processes. Test-driven development, pair programing, continuous integration, short release cycles, refactoring, simple design and on-site customer are techniques that were included in Kent Beck's XP book [1]. XP introductions often happened bottom-up: software developers pushed the ideas into their development projects and hoped to streamline the delivery of value to their customers.

By the mid 2000s, agile methods moved from the development cubicle to the front-line management level. At that time, many teams started their agile adoption with ideas from Scrum for improving the management of software projects. Ken Schwaber's Scrum [2] emphasizes iterative and incremental development, self-organizing teams and continuous process improvement in small steps. The methodology provides a set of tools to help with coordinating software development efforts while ensuring that value is delivered to customers frequently and reliably. This focus on project management issues made Scrum a favorite for front-line and middle management – which resulted in a middle-out strategy for agile method adoption where middle managers pushed agile ideas downwards into their teams as well as upwards into senior management.

More recently, in the late 2000s/early 2010s, agile adoptions often seem to be pushed from senior management to the whole enterprise. Mary & Tom Poppendieck's Lean Software Development [3] is based on ideas from the Toyota Production System and translates them into software development processes. Lean software development provides guidelines for enterprise-level agile adoptions and includes techniques like value stream mapping, flow, reducing cycle time and kanban.

While we highlighted XP, Scrum and Lean above, other methodologies fall into the agile space and had substantial impact on the area. Feature-driven development [4], DSDM [5], Crystal Clear [6], and adaptive software development [7] are some of the approaches that had a substantial impact on the thinking and progress in the agile community. However, our own – subjective – observations with industrial partners clearly indicate that XP, Scrum and Lean are the ones that are more widely adopted and discussed.

When we interact with teams that want to adopt agile approaches, we usually suggest they initially focus on two aspects: agile project management and agile quality assurance. The remainder of this chapter discusses these in more detail.

In Section 2, we provide an overview of agile project management approaches. We discuss user stories, user story mapping, and low-fidelity prototyping as well as release and iteration planning. Section 3 presents an overview of agile quality assurance focusing on test-driven development and acceptance test-driven development, and also discusses the increasingly-important topic of graphical user interface (GUI) testing. The concept of the state space of an application as it relates to testing is also described in Section 3, as well as the implications of this concept in relation to GUI testing. The final section summarizes our findings.

## 2 Agile Project Management

Agile project management is based on four values:

- Communication,
- Simplicity,
- Feedback, and
- Courage.

Communication is key for any software development project. Business representatives understand their problems and can develop ideas about how they can be overcome with software. However, they usually do not have the technical skills to develop the software system. Thus, communication is an essential bridge between the business domain and the development domain. Communication is needed between all stakeholders in a project – from senior management to future users, IT operations, software development, user experience, project management.

Simplicity is about asking the question: what is the simplest thing that could possibly work? The question needs to be raised when designing software to avoid gold-plating and over-engineering – YAGNI (you ain't gonna need it) is the agile battle cry. But it also needs to be raised in regard to project planning and progress tracking: what does a team have to do to get an accurate picture of the future development effort?

Feedback is fast and frequent in agile teams. Essential feedback comes from putting the system (or updates) into production as quickly as possible. Feedback from real use allows the development team to find bugs early and fix them. It helps the team to steer the project back onto the right path when needed and provides necessary confirmation of success when users *do not* find problems with newly deployed features. Feedback from successful regression testing provides validation that existing features have not been broken by new development results – ensuring the effort estimates remain valid and the project stays on track.

Courage is needed when developers point out unrealistic expectations to customers: not everything can be delivered by a few weeks of work. Courage is also essential when the development team has to explain to the customer why delivering new features must be postponed for a major redesign of the existing platform.

A core agile strategy that embodies the four agile values is the creation of holistic teams.

### 2.1 Whole Team

A primary goal in agile project management is to create a "whole team" that has all skills required to successfully create a software system. The team usually includes business stakeholders, analysts, software architects and designers, developers, testers, as well as any other stakeholder that needs to be involved in the discussions. Some agile methods, for example XP, argue that teams usually require multi-skilled personnel: generalists that can fulfill multiple roles for the team. In such teams, role rotation is common. However, teams –specifically larger teams – often include specialists that focus on certain aspects of the project. Depending on workload, specialists are shared between multiple teams, e.g. database administrators or usability experts often serve in their respective roles in multiple teams. The whole team is involved in collaboratively planning the next steps in the development effort. If possible, project planning is conducted by bringing all team members into the same room for a face-to-face conversation.

### 2.2 Project Management

Project management deals with four variables: cost, scope, schedule and quality.

Cost in software development is highly correlated to the number and quality of team members. Cost overruns were – and still are – a major problem for software development projects. The scope of a project is defined by the set of all features that need to be delivered to the customer. The schedule determines when a feature is or should be delivered. The customer perception of quality is based on fitness for purpose as well as the number of bugs that are found after delivery. Project management needs to determine the appropriate balance between these dimensions. Improvements in one dimension often impact other dimensions; for example, reducing the time to delivery can to a certain extent be accomplished by hiring additional developers for the duration of the project – which makes the project more expensive. It is a fallacy that project management can optimize each dimension individually.

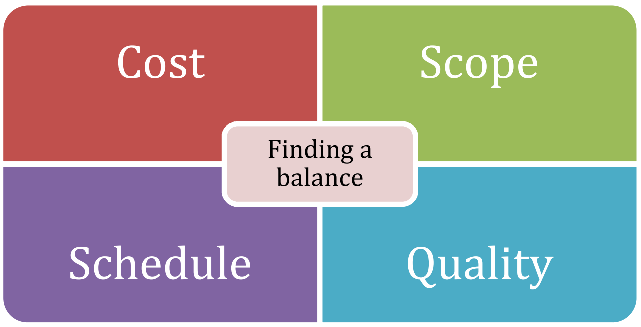


**Fig. 1.** Project variables

Agile methods recommend against spending much effort on upfront work. After acquiring a basic understanding of the project's goals and high-level requirements, teams are expected to quickly start development iterations that deliver potentially shippable product functionality. Usually, upfront work is limited to days or a few weeks of effort. This approach is quite the opposite of more traditional software development processes that front-load the development process and emphasize a thorough and detailed analysis of software requirements followed by substantial architectural and design work. The benefits of the agile approach are:

- As business environments and processes change quickly in today's competitive environment, a large delay between determining a requirement and delivering it might make this requirement obsolete. In this sense, the agile approach minimizes the risk that effort is spent on analyzing, designing and implementing features that will be unnecessary by the time they are delivered.
- The limited amount of development effort available in short iterations naturally forces business stakeholders to prioritize their feature requests. Reasonable businesspeople understand that a team will not be able to deliver all their requirements in the next few weeks and will determine what features are most urgently needed. As a result, requirements tend to be fulfilled in decreasing levels of importance or urgency. This in turn allows management to cut off a project when it determines that the business value of future iterations does not justify the costs incurred by them.
- As requirements are quickly turned into implemented features, feedback from actual use helps to determine if these features are what is actually needed or need to be revised.
- Effort spent on upfront work is actually wasted if the system is never delivered to production. Limiting work before delivering a first feature set to production reduces this risk.
- Source code is where the rubber hits the road in software development. Detailed analysis and design models that are unrealistic exist – but the first attempt to build the system often finds their issues quickly.

However, proponents of more upfront-centric approaches have arguments that can be seen as a criticism of the agile style:

- Empirical studies have shown that fixing a bug after the software is delivered is 60-100 times more expensive than fixing the same issue in the analysis phase [8]. Thus, a thorough process that emphasizes analysis and design will save expenses, as it does not allow bugs to slip through.
- Assuming the software designers get a set of current as well as future requirements, they can develop code structures that make future changes easy and cost effective compared to refactoring as needed. Designing with models is less expensive than designing in code.
- Starting development without a basic understanding of the project's vision and goals will likely lead to wasted effort as initial implementation will likely become useless over time.

Development teams should weight these arguments before deciding which approach they want to follow. In the following section, we will discuss techniques used by agile teams that are trying to strike a balance between these conflicting approaches.

## 2.3 Agile Project Planning

Agile teams usually plan on three levels of abstraction:

- Project vision
- Release plan
- Iteration plan

The project vision captures the really big picture: Why is the project run? What are the expected benefits? What are the budgetary and other constraints? How will the organization function after the project is successfully completed? A project vision is often used to establish a project budget or, at least, a budget that allows the organization to refine the vision enough so that a go/no-go decision can be made. Agile teams try to minimize this upfront work to avoid getting stuck in analysis without getting feedback about delivered product functionality.

A project vision needs to clearly describe the anticipated benefits for the business as well as assessment criteria that management can use to evaluate progress towards realizing the vision. Agile teams need management oversight to ensure that the next iteration/release still delivers enough business value to justify the development costs.

Release planning creates a strategic picture on the project. The team looks a few months ahead and determines the high-level features/user stories that need to be realized in that time frame. In practice, we observed teams creating release plans for the next three to six months, with a few exceptions looking approximately one year ahead. The release plan determines release dates and iteration length. Scope is captured on a high level but may be changed in the future based on new insights gained during development. User stories from release planning form the initial product backlog.

Iteration planning determines the work for the next development iteration. The whole team gets together to review what was delivered in the last iteration and then collaboratively determines what should be delivered by the end of the next iteration.

### 2.3.1 Planning a Release

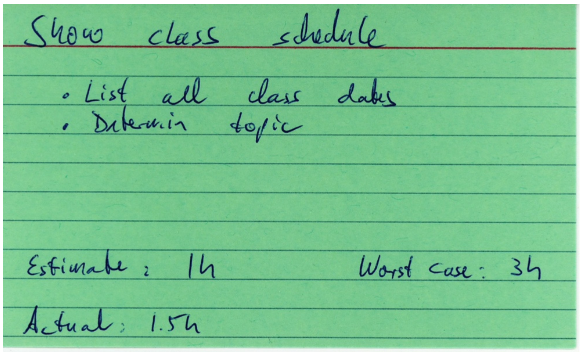
For release planning, we recommend that the whole team gets together to

- collect and discuss high-level user stories that should go into the next software release,
- build a user story map, and
- create low fidelity prototypes.

Release planning is often conducted in a 1-3 day workshop involving the whole team and, if possible, external stakeholders. It starts with collecting user stories on different levels of abstraction: epics, themes, and implementation-ready stories.

*User Stories***.** A user story (also called: backlog entry or feature request) briefly describes a requirement that has business value. It serves as boundary object that enables communication between different stakeholder groups. According to Wikipedia, “A boundary object is a concept in sociology to describe information used in different ways by different communities. They are plastic, interpreted differently across communities but with enough immutable content to maintain integrity”[2].

A user story is captured on an index card (see Fig. 2). As a bare minimum, the user story has a name and a short description of the requirement. Descriptions need to be in customer language and avoid IT terminology. They need to be understandable by all team members. A user story also often includes effort estimates and is used to note actual effort during development.



Show class schedule

- List all class dates
- Determin topic

Estimate: 1h Worst case: 3h

Actual: 1.5h

**Fig. 2.** Example story card

Mike Cohn, a prominent author focusing on agile project management, recommends a more structured approach for user stories:

- As a *[type of user]*
- I want to *[perform some task]*
- so that I can *[reach some goal]*

This structure helps business stakeholders prioritize user stories.

Index cards are small and will not be able to capture all details about the user story. They act as reminders to the developers to discuss these details with business representatives as soon as they start working on the story implementation.

Often, the back of the story is used to capture acceptance criteria for a user story. However, a more recent recommendation is to capture these in form of executable acceptance tests using frameworks such as Fit [9] GreenPepper[3] or BDD[4]. This approach is described in more detail in Section 3.

[2] `http://en.wikipedia.org/wiki/Boundary_object` (last visited 29 July 2011).
[3] `http://www.greenpeppersoftware.com/` (last visited 29 July 2011).
[4] `http://dannorth.net/introducing-bdd/` (last visited 29 July 2011).

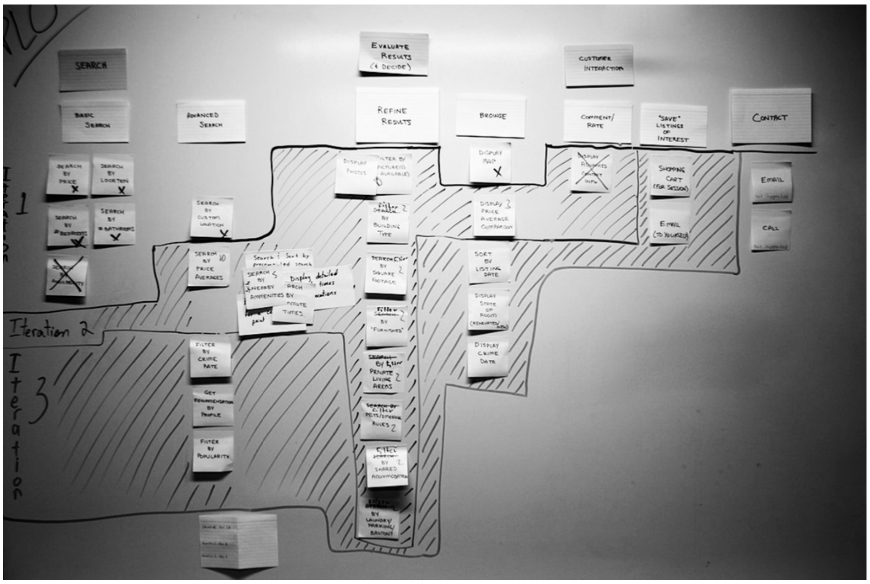


**Fig. 3.** Example user story map

### 2.3.2 Mapping User Stories

A user story map[5] organizes and prioritizes user stories for a release. It makes the workflow of the system visible to the whole team and shows the relationship between large user stories and their parts. Fig. 3 shows an example user story map developed by a student team in its release planning workshop.

A user story map shows the sequence of activities of the system's workflow horizontally at the top of the board, left to right. The team then organizes (sub)tasks under their activity in the order of the workflow. This shows the relationship between activities and (sub)tasks while maintaining the time dimension of work steps. Concurrent or alternative tasks are added vertically by priority. Fig. 4 shows a conceptual example. Subtasks for tasks can be added as needed (not shown in Fig. 4).

During release planning, the user story map is created, discussed, refined, extended, updated, and changed until it is complete. The team checks completeness by validating that the "story" of the system can be told by connecting the activity cards. It then evaluates if the tasks and subtasks provide enough information for answering more detailed questions.

After the user story map is complete, the team determines coherent sets of tasks for the iterations that make up the release. Fig. 3 shows this with an example.

[5] Developed by Jeff Patton `http://www.agileproductdesign.com/presentations/user_story_mapping/index.html` (last visited 29 July 2011).

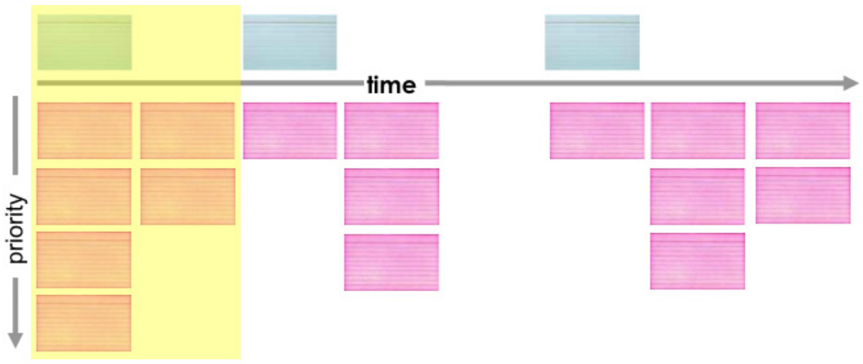


**Fig. 4.** Conceptual user story map

The creation of a user story map is a – limited – upfront planning exercise. However, the time spent on it is substantially shorter than the time used for requirements analysis in more traditional processes.

When properly conducted, release planning is a collaborative exercise involving the whole team that creates a shared understanding of the release goals and system workflow.

### 2.3.3 Low-Fidelity Prototyping

User story maps allow a team to get a high-level overview of a system's feature set. A second technique that is used by agile teams to supplement the map is low-fidelity prototyping. Low-fidelity prototypes are particularly useful for development efforts with a strong focus on usability and interaction design issues. A low fidelity prototype is a sketch – a hand-drawn representation – of a user interface. A sequence of sketches that illustrates a workflow of a system is called a storyboard.

Teams use sketches and/or storyboards to capture the conceptual structure of the system's user interface. Fig. 5 shows an example sketch that illustrates the user interface of an agile project planning tool.

Bill Buxton's book [10] discusses the benefits of sketching for software development. Sketches are quick and easy to make. Thus, they fit well into short, iterative development cycles as exhibited by agile development teams. They can be provided when needed. As they are cheap to make, they are also disposable and their creator usually does not have a strong stake in them. As a result, teams often develop multiple alternative sketches for a user interface and discuss them with end users as well as other stakeholders. The series of ideas that is illustrated by alternative sketches often helps teams to clarify the design intent and the concept underlying the user interactions. The final system is often a combination of different ideas (expressed by different sketches).

Sketches also elicit feedback on the "right" level. Their rendering and style makes it clear that they are conceptual in nature and that the look of the system is not

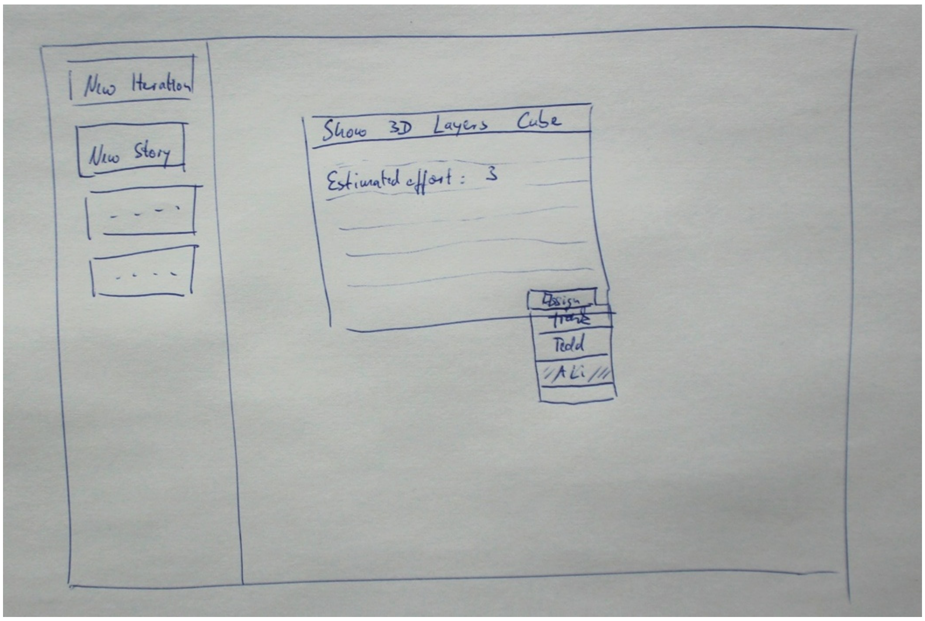


**Fig. 5.** User interface sketch

finalized yet. Users usually comment on conceptual structures instead of the choice of color or fonts. These conceptual level comments are exactly what interface designers need in the early stage of a development project.

Sketches help to start conversations between users and designers in the same way as user stories trigger discussions between developers and business representatives. Their value lies in provoking interactions between all stakeholders and helping teams derive better solutions for their customers.

Where a sketch illustrates the layout (or wireframe) of a single screen, a story board captures a workflow supported by a user interface. Story boarding is a technique borrowed from the movie industry: "Storyboards are graphic organizers such as a series of illustrations or images displayed in sequence for the purpose of pre-visualizing a motion picture, animation, motion graphic or interactive media sequence, including website interactivity." [6]

Fig. 6 shows an example story board for an agile planning tool. The sequence of sketches illustrates how a user can create a story card, give it a name and effort estimate and, lastly, select the developer responsible for it.

Interaction designers can use sequences of sketches, as available from storyboards, to simulate the workflow with the user. These Wizard-of-Oz [11] experiments allow gathering feedback on the usability of a user interface before the implementation exists and are often used to ensure that even the first version of a UI creates a positive user experience.

[6] `http://en.wikipedia.org/wiki/Storyboard`

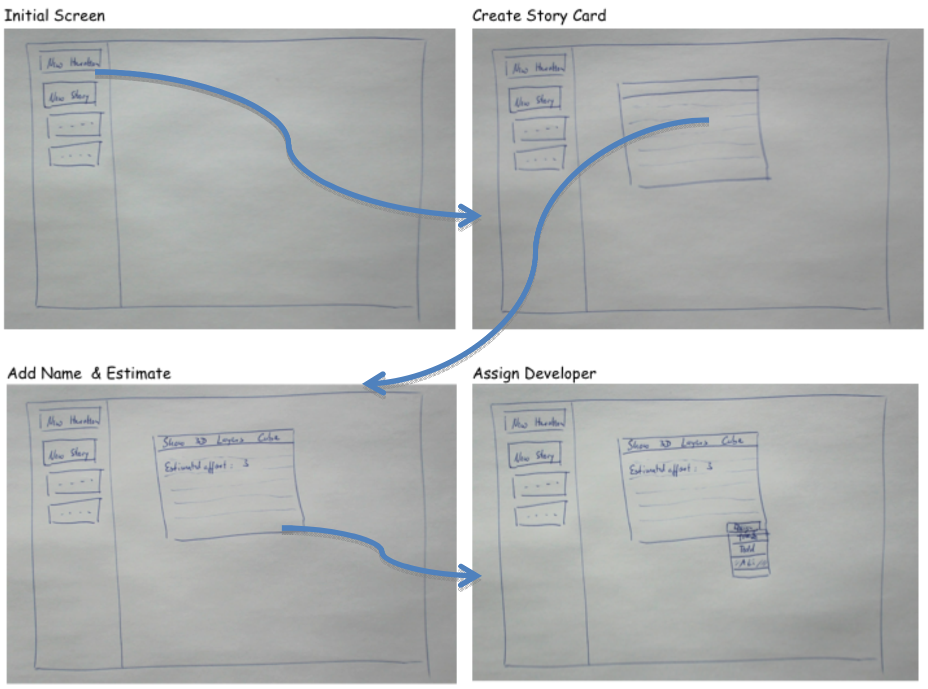


**Fig. 6.** Example story board

In Section 3.3, we will discuss how automated Wizard-of-Oz tests based on storyboards can also be used for test-driven development of graphical user interfaces.

### 2.3.4 Iteration Planning

Iteration planning allows a team to get a tactical picture of the development effort: teams look ahead for a few weeks and create a realistic plan on what should be accomplished by the end of the iteration. The length of iterations is usually fixed in an agile project and the team will deliver on time (maybe with a reduced scope; see iteration planning). Most teams we work with run iterations for 2-4 weeks. Lately, there seems to be a tendency towards shorter iterations, i.e. we see more and more teams moving towards a two week cycle.

Following an iterative development process results in fixed delivery dates. Customers can expect to get a new set of production-ready features at the end of each iteration. Fixed delivery dates (with a slightly variable scope) have advantages:

- Sometimes external deadlines are hard. If a system needs to be demonstrated at a trade show to have a benefit for the developing company, the date is fixed externally and it doesn't help to deliver a more complete system after the show has ended.
- Reliable delivery of new features increases the trust of customers in the development team. Unfortunately in the past, customers were often

burned by late deliveries and low quality. Thus, a constant and reliable delivery cycle increases the customer's confidence in the team and usually results in a more collaborative work environment.

- Developer motivation increases when they constantly deliver new features to their customers. Everybody likes to be successful – and delivering an increment is seen as a success.
- Putting new features into production allows the team to get feedback from actual use of the new functionality. While teams try their best to get everything right, the chances are that some details are wrong. Getting systems into actual use will quickly discover such issues and allow development teams to fix them quickly. Instead of accumulating technical debt over a long time, fast delivery will allow teams to deal with it in more manageable chunks.
- Reoccurring short-term delivery dates create some pressure on the team to focus their efforts on concrete steps. Parkinson's law states that work fills the time available for its completion. Short deadlines encourage teams to work on relevant tasks.

Iteration planning meetings normally run for a few hours and are attended by the whole team. The team selects the highest priority user stories from the user story map and discusses them. When needed, additional user stories are brought forward and the user story map is augmented accordingly.

The goal of team discussion is to enable developers to come up with a realistic estimate of the development effort for the story. These estimates together with the time available in the iteration allow the team to select a realistic set of stories that should be implemented in the upcoming iteration.

Cohn [12], p. 83+85, suggests that teams consider two dimensions when prioritizing user stories: business value and development risks. He suggests (see Fig. 7) to start with high risk, high value stories. Addressing high risk stories first allows a team to determine if the system is technically as well as economically feasible at all (and if not: cancel the project quickly before incurring the majority of the project costs).

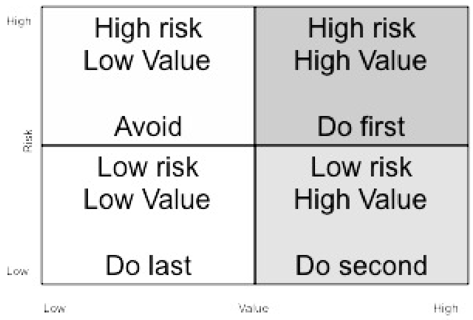

High
Risk
Low
High risk Low Value Avoid
High risk High Value Do first
Low risk Low Value Do last
Low risk High Value Do second
Low
Value
High

**Fig. 7.** Business value and development risk

To determine how many user stories fit into the upcoming iteration, the team estimates user stories and determines its velocity.

*Effort estimation*: Effort estimates try to determine the size or complexity of a story by comparing it with others of similar complexity. Teams use different metrics for their estimates: (ideal) hours, story points or even gummy bears[7]. The goal of the estimates is to cluster stories that require similar efforts into the same bin – not to determine the amount of work hours needed for completing the user story (velocity is used for this). Typically, developers use their experience to determine an estimate. They remember similar tasks from the past and derive their estimate by remembering the effort of the past tasks. This means that estimates are mainly based on expert opinion and analogical reasoning.

Some teams use planning poker to derive estimates collaboratively. Each team member estimates for herself and then places a card with her best estimate on a table. If the set of cards shows different numbers from different developers, the team discusses these discrepancies and then estimates again until the estimates converge. Big discrepancies in estimates are treated as opportunities to refine the understanding of the story in the team as the differences are usually a result of an inconsistent understanding of what the story entails.

We recommend that developers provide two estimates for each story:

- Most likely estimate: the estimate that she thinks is really needed if no unexpected events happen while developing the story.
- Worst case estimate: the developer is asked to come up with a number that she is willing to guarantee

We treat the most likely estimate as a 50:50 chance that the actual effort needed to complete the story is at or below the estimate. On the other hand, we see the worst case estimate as a 95% chance that the actual effort is below the estimate.

Managers need to be careful in *not* treating most-likely estimates as commitments. The goal of estimation is to get the most realistic picture possible of what will happen in the next iteration. When estimates are treated as commitments or promises, developers will start over-estimating their effort to be on the safe side.

Estimates are not 100% accurate. A team will only know how much effort a task is after it finishes working on it. Thus, planning is not about getting *the* correct picture but is about getting a perspective on the development project that allows a team to move forward while providing customers a good idea of what will be delivered at the end of the iteration.

For any iteration, estimates should stay within one order of magnitude. This prevents an effort overrun in one task from dominating the results of the iteration. When user story efforts are too far apart, small tasks can be combined or large tasks can be split. Splitting a task can be based on [12], p 121ff:

[7] The "gummy bear" metric attempts to make it clear that the number that is derived by the developers can not directly be mapped to calendar time.

- the data supported by the story (e.g. Loan summary → List of individual loans → List of loans with error handling)
- operations performed within a story (e.g. separate create, read, update, delete (CRUD) operations)
- removing cross-cutting concerns (e.g. a story without and with security)
- separating functional from non-functional requirement (make it work, then make it fast)

When all user stories that might go into the next iteration are estimated, a team uses its velocity to determine how many of these are likely to be accomplished in the upcoming iteration.

*Team velocity*: A team's velocity determines how many story points are likely to be completed in the next iteration. Teams use a simple heuristic to determine this number: yesterday's weather. The assumption is that a team will be able to complete as many story points in the next iteration as it finished in the last iteration. The number is then slightly modified based on the number of person days in the upcoming iteration compared to the number of person days in the last one.

Combining story point estimates with velocity creates a simple approach for project planning. In our experience, it works rather well assuming that

- there are no major changes in the team and
- the team doesn't dramatically change its approach to estimating from one iteration to the next.

The approach is self-adaptive and corrects for developer optimism. A team that takes on too many user stories in one iteration will see its velocity reduced in the next iteration as they did not finish all their tasks. When a manager realizes that a team runs out of tasks in the current iteration, she can always go back to the business representatives and ask for more user stories. When they are also completed, the team's velocity will go up for the next iteration.

As estimates come from developers, some managers argue that they now have the power to slack off. However, this is counter-balanced by the customer's ability to cancel a project if progress is too small to accomplish its vision within a given budget.

While a team's velocity determines how many story points the business representatives can select for an iteration, one question remains: which of the two estimates should be used? The answer is: both. We usually recommend that teams first select a number of must-have stories for the next iteration based on the worst-case estimates. Business representatives can be quite sure that the developers will complete these tasks as the worst-case estimates will likely be met. However, the expectation is that not all tasks will require the effort as determined by the worst-case estimate. Thus, the team selects as second set of optional user stories while keeping the sum of the most-likely case estimates of all selected stories below the velocity:

- $\sum_{i \in US_1} worst\_case\_estimate(user\ story_i) < velocity$
- $\sum_{i \in US} most\_likely\_case\_estimate(user\ story_i) < velocity$
  where $US_1$is the set of must have stories, $US_2$ is the set of optional user stories and $US = US_1 \cup US_2$

These constraints on the one hand ensure that the customers know at the beginning of the iteration which user stories will definitely be delivered while any remaining time is filled with optional user stories based on the customer's priorities.

## 2.4 Progress Tracking

Agile teams track their progress on three levels of abstraction:

- Daily: Are we in trouble at the moment?
- Iteration: Will we make our tactical goals?
- Project: Will we reach our vision?

For tracking daily progress, most agile teams use a short stand-up meeting at a regular time. During the daily stand-up, each team member reports on three questions:

- What have you done since the last meeting?
- What will you do before the next meeting?
- What is in your way?

The meeting is limited to at most 15 minutes and held at the same time and place every workday. The meeting is not meant for problem solving but for bringing issues to the attention of the whole team so that an appropriate group of people can be identified that can get together after the stand-up and find a solution.

Nobody sits during a stand-up. This encourages people to keep everything short.

Daily stand-ups force people to think about their short term goals and report on their short term accomplishments. The latter creates some benevolent peer-pressure as developers not making any progress on their tasks for several days in a row become very visible. The last question addressed by each team member helps to discover roadblocks quickly. The earlier a team knows about an issue, the earlier it can find a solution.

Tracking progress within an iteration is done with task boards. A task board shows the stages through which each user story/task goes and where it currently is. Fig. 8 shows an example task board from Mike Cohn's web site. In this example, user stories are split into individual tasks. These tasks go through four stages: to do, in progress, to verify and done. Each row in the task board shows the tasks for a certain user story. Task boards are widely used by agile teams. They act as widely visible information radiators that help all team members to understand how much progress is being made in the current iteration.

For a more detailed tracking of progress against the iteration goals, teams sometimes use burn-down charts. These chart the amount of not-yet completed tasks on a daily basis[2].

At the end of each iteration, an iteration review is conducted to show to product owners and customers/users how much progress was made during the iteration. During the review, the team demonstrates all features that were completed in the current iteration. An iteration review should not impose extra overhead on the development team. Thus, it is conducted using the development equipment. Features shown during the review must represent potentially shippable product functionality: i.e. it is then a business decision if the feature goes live or not.

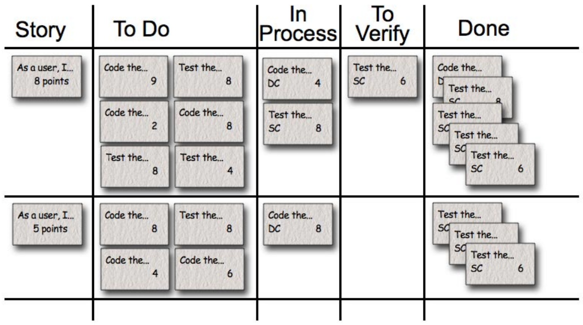
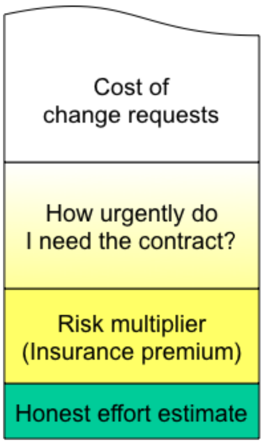


**Fig. 8.** Task board example[8]

Iteration reviews give senior management an opportunity to see if the progress that was made is sufficient to make the project vision reachable within the given budgetary and scheduling constraints.

## 2.5 Business Contracts

Historically, software development contracts had a time-and-expenses structure, i.e. customers paid a fixed amount of money per developer hour plus reimbursed the team for all expenses incurred by the project. However, given the amount of cost overruns in the past, customer organizations switched to a fixed price/fixed scope structure: customers pay a fixed amount of money to the development team that in turn must deliver a set of features laid out in a detailed requirements specification. This approach tries to move the risk of incorrect estimates from the customer side to the developer side. Unfortunately, the success of this approach is quite limited:


Cost of change requests
How urgently do I need the contract?
Risk multiplier (Insurance premium)
Honest effort estimate

- Development organizations realize that a fixed price/fixed scope contract makes them vulnerable for incorrect effort estimates. After they determine the honest effort estimate for a contract, they will use a risk multiplier when they submit a bid. This multiplier de-facto serves as an insurance premium that a client has to pay to the development organization to assume the technical risk of the contract. If the initial effort estimate is correct, the client organization pays more than needed to avoid the risk of becoming burned by incorrect estimates.
- The size of the risk multiplier is determined primarily by how urgently the development organization wants the contract. When business is booming, customers will pay a high premium.

[8] From `http://www.mountaingoatsoftware.com/scrum/task-boards`

> When it is slow, they will pay less. However, even when a development organization lowballs the project bid to get the contract, the customer still will pay too much as developers know that over the course of a project customers always change the requirements.[9] Developers can overcharge for changes as switching the development organization mid-project is usually not economically feasible for the customer organization due to penalties written into the contract. Fig. 9. illustrates these issues.

Agile organizations can replace fixed price/fixed scope contract with a time-and-expenses contract with an early termination clause. The later limits the risk of the customer organization as it can cancel the project quickly when it realizes that the project will not be able to deliver on its vision given the current budget constraints.

While project planning and progress tracking are important aspects of agile software development processes, bad software delivered on time is still bad software. Thus, we are now discussing how agile teams assure that they delivered high quality.

## 3 Agile Quality Assurance

Have you ever worried that the feature you've been developing doesn't match the expectations of your customer? Have you ever been reluctant to change code because you might break something? Have you ever been unsure about whether or not you've finished a feature? Have you ever been terrified that one of the other developers might go on vacation, and that no one else will be able to understand what his code does?

Agile quality assurance is a set of testing methodologies that have evolved over time to minimize these risks on software development projects. The overall goal of these methodologies is to increase understanding of and communication about the system that's being developed. These practices can be divided into two classes: developer-facing and customer-facing tests. Tests written by developers ultimately help them design and understand the system, while tests written under the auspices of customers help developers understand what customers want and help customers understand what developers can offer.

Developer-facing tests require in-depth knowledge of the way in which the system works, and require technical proficiency in a testing language to understand. These tests are usually glass-box (or white-box) tests in which parts of the source code of the system are tested. This name derives from the fact that the system under development is being treated as something we can look into and inspect the intermediate results of our actions. For example, in glass-box testing, we can set or inspect the state of specific objects or call only specific methods within the application rather – as opposed to triggering high-level functionality, which would result in changes to the states of many objects and many method calls. This allows much more fine-grained understanding of the way in which a system works, and will help developers ensure that the feature they are coding matches their goals for its behavior – in other words, that

[9] We've been involved in software development for about 30 years now and haven't seen a single project where requirements stayed fixed for its whole duration.

developers are building the system right. Developer-facing tests are almost always automated through a testing framework like JUnit[10] or the Visual Studio Unit Testing Framework[11]. Examples of developer-facing tests include unit tests, integration tests, and system tests.

Customer-facing tests, on the other hand, are intended to be understandable by domain experts without requiring programming knowledge. These tests tend to be black-box tests in which the internals of the system are not considered. An input is provided, and the expectations of the business experts are compared against the output the system produces. This sort of test ensures that the code created by developers fulfills customer expectations – in other words, that developers are building the right system. Customer-facing tests can be automated (through a system like FitNesse[12] or Green-Pepper[13]) or manual (through live demos on the actual system).

Customer- and developer-facing tests can be envisioned as two partially-overlapping squares, as shown in Fig. 10. Developer-facing tests can show that individual parts of an application are working in detail on a programmatic level, but not that defined features are missing. Customer-facing tests, on the other hand, can show that features are present, but not that they are working in detail on a programmatic level.

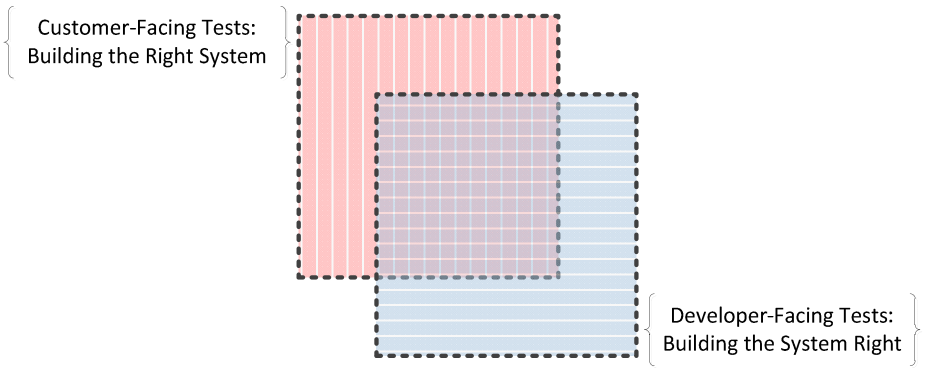


**Fig. 10.** Both genres of tests are necessary in agile quality assurance

Finally, agile quality assurance tends to make heavy use of automated developer- and customer-facing tests. This is due to the fact that the same tests will tend to get run a large number of times on an agile project. For example, refactoring is a key concept in agile software development. However, there is a risk that developers may introduce errors into the program while performing this task. A suite of automated tests can catch these errors quickly, which both emboldens developers to aggressively refactor their code while at the same time making the refactoring process much safer. In this light, automated tests are definitely worthwhile.

[10] `http://junit.org`
[11] `http://msdn.microsoft.com/en-us/library/ms243147.aspx`
[12] `http://fitnesse.org/`
[13] `http://www.greenpeppersoftware.com`

This isn't always the case though – it's a fallacy to think that you have you automate every single test on your project. This is because some automated tests actually cost more to create and maintain over the course of the project than a manual equivalent. Brian Marick addressed this point eloquently in 1998:

"It took me a long time, but I finally realized that I was over-automating, that only some of the tests I created should be automated. Some of the tests I was automating not only did not find bugs when they were rerun, they had no significant prospect of doing so. Automating them was not a rational decision" [13].

Remember, the point of test automation is to save effort in the long term. If a test is difficult to automate or doesn't have a reasonable chance of catching bugs, it may be more cost-effective to run this test manually. The best tests to automate are those that have a good chance to find bugs over a long life expectancy.

## 3.1 Test-Driven Development

Test-driven development (TDD) is a software development paradigm in which tests are written before the code they are referencing actually exists. The tests used in TDD are assumed to be automated, developer-facing unit tests unless otherwise specified. This activity is more about software design and communication than it is about testing per se, though it does build up a suite of regression tests that are useful for detecting errors introduced by changes made later on. The goal of TDD is to increase the confidence that developers have in their code, decrease the occurrence of bugs that make it through to the customer, prevent the re-introduction of bugs, and increase communication between developers and customers.

The first step in TDD is to write a new test. This test should be confined to a single new part of the system – a new method, a new class, or a new feature depending on the scope of the testing. This causes the system to enter a red state: at least one test is failing. In other words, there is something wrong with the system - it's missing the part specified by the new test. This defines a goal for the developer: get the system back to a working state as quickly as possible. From this perspective, tests are driving the development of the system.

Initially, this new test should be the only failing test for the system, so the next step is to verify that this test is failing. If this new test passes immediately upon creation, either:

1) there's something wrong with the test; or
2) the "new" part of the system already exists, meaning no new code is necessary; or
3) the developer misunderstood the current design of the system as a test that is expected to fail in fact passes, meaning the developer will have to increase her knowledge about the system.

Once we've watched our test fail, code should be written with the specific goal of getting the new test to pass - no code should be written unless it directly relates to making this test pass! Additionally, it's alright if our code is not perfect at this point, because we'll improve it in the next step.

Once the test is passing, the system is back in a green state (all tests are passing), and we can focus on the crucial last step: refactoring. In the previous paragraphs, the emphasis was on speed. This means that it's crucial for us to go back to the new code to make it efficient, secure, robust, maintainable, or any one of a number of software quality concerns. However, this process is a safe one now because of the new test. If our refactoring causes this test – or any other test – to fail, our first priority again becomes getting the system back to a working state. Because of the suite of regression tests built up through TDD, developers can aggressively refactor the code base of an application.

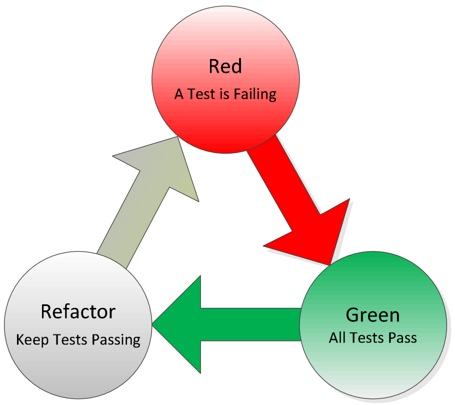


**Fig. 11.** The test-driven development cycle

Evaluations of TDD have had mixed results. In general, it would seem that TDD has a negative effect on productivity and a positive effect on quality [14]. However, as was mentioned in Section 2.2, bugs found after release of an application are significantly more expensive to address, so any decrease in productivity needs to be viewed with this in mind as the studies in the above mentioned publication did usually not include data about post-deployment productivity comparisons.

## 3.2 Acceptance Test-Driven Development

In Acceptance Test-Driven Development (ATDD), instead of creating automated, developer-facing unit tests, we create a suite of customer-facing system tests. These tests are created before the features they test are implemented, as in TDD. However, in ATDD, these tests should actually be created by customer representatives as descriptions of what the application should behave like when it is working correctly. Because of this, many acceptance testing frameworks, like FitNesse and GreenPepper, include an interface that is friendlier to non-technical test writers. In practice, business representatives may still need assistance in writing tests, in which case they should be paired with testers who can help them write tests (not write tests for them!). An example of a FitNesse test (showing a hypothetical business expert’s expectations for the result (right) of division given a specific numerator (left) and denominator (middle)) is shown in Fig. 12.

```
|eg.Division|
|numerator|denominator|quotient?|
|10       |2          |5        |
|12.6     |3          |4.2      |
|100      |4          |33       |
```

**Fig. 12.** Example FitNesse acceptance test[14]

It’s important to note that most tools which are advertised as acceptance testing tools interact with an application below the level of its user interface by directly making calls into business methods of the application and verifying the results. This is useful in that avoiding the GUI simplifies the creation of an automated test drastically. However, if parts of the GUI are important to a customer’s acceptance criteria, it will be difficult to automate that part of the test for ATDD using such tools. It’s possible to write acceptance tests that involve interactions with a GUI, but few methods exist that make it easy to perform ATDD of a GUI.

As with TDD, tests created for ATDD should be automated wherever possible. Acceptance tests written using many acceptance testing tools can be run alongside other automated tests as part of the suite of regression tests. This means that changes to the application under test that cause violations of the customer’s acceptance criteria can be detected quickly and easily.

However, the functionality of modern applications is becoming increasingly difficult to test automatically. This is due in part to the fact that modern applications are heavily dependent on GUI-based interactions. While it is easy to automate tests of the functionality of a standard webpage or desktop application using FitNesse tests, it’s difficult to automate tests of applications with complex GUIs. In these instances, it may be preferable to specify manual tests using a tool like Microsoft Test Manager[15], which integrates manual tests with other software development tools. When using manual tests as part of ATDD, however, developers will need to execute these tests manually numerous times during development, which can be a tedious and expensive process.

## 3.3 Test-Driven Development of Graphical User Interfaces

User interfaces are an important part of almost every modern application. Simply put, they allow users to interact with applications. Traditionally, this was done with

[14] Source: `http://fitnesse.org/FitNesse.UserGuide.TwoMinuteExample`

[15] `http://msdn.microsoft.com/en-us/VS2010TrainingCourse_AuthoringAndRunningManualTests`

keyboard and mouse, but this is now possible using touch input in mobile phones (like the iPhone and Windows Phone 7), tablet computers (like the iPad and Asus EEE Slate), and digital surfaces (like the Microsoft Surface, SMART Board, and SMART Table). Further, up and coming technologies like the Microsoft Kinect are making it possible to interact with a computer without even touching it. Clearly, user interfaces are an important and complex concern in software development.

Further, user interfaces can be either event-driven or loop-driven. Event-driven interfaces primarily respond to input from the user. Examples of event-driven interfaces include traditional desktop applications and web pages. Loop-driven interfaces are primarily driven by the passage of time, but will also take user input into account. Many computer games are excellent examples of loop-driven interfaces. The difference is that in an event-driven interface a sequence of interactions will produce the same result regardless of timing, but in a loop-driven interface this is unpredictable.

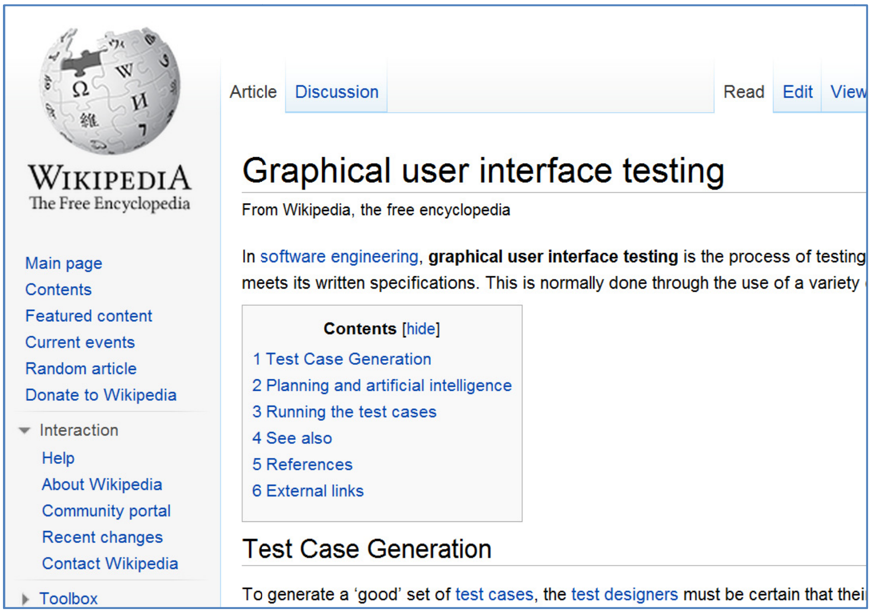
For the purposes of this chapter, let us consider only event-driven graphical user interfaces (GUIs) based on mouse, keyboard, or touch interaction. While powerful patterns for dealing with the complexity of GUIs exist (e.g. the Model-View-Controller pattern), there is still a significant amount of code present in a GUI – in fact, 45-60% of an application's code can be dedicated to its GUI [15]. In line with this, one case study found that 60% of software defects found after release relate to GUI code, and of these defects, 65% result in a loss of functionality [16]. Taken together, these studies suggest that GUI testing is an area of significant concern.

However, automated GUI testing is far from straightforward. In order to better understand what makes GUI testing a daunting task, let us consider four fundamental concerns of automated software testing made especially clear in this context:

- Complexity,
- Verification,
- Change, and
- Cross-Process Testing

The complexity of an application refers to the number of alternative actions that are possible. GUIs allow a great amount of freedom to user interaction, making them very complex. When testing the functionality of a GUI-based application using automated tests, two factors are of prime importance: the number of steps in the test; and the number of times each action in the GUI is taken within a test [17]. In Section 3.4, the implications of the complexity of modern GUIs will be explored in more detail. In order to notice that a bug has been triggered, a test must also contain verifications that will be able to notice that bug [18]. This is especially tricky when considering that many aspects of GUIs are subjective. For example, it can be difficult to create a test for determining whether a web page was rendered correctly. Third, GUIs tend to change drastically over the course of development. A GUI test can show up as failing although the underlying code is actually working [19] [20]. This is especially important since a large number of false alarms from the GUI testing suite will cause developers to lose confidence in their regression suite [21]. Finally, these difficulties are compounded by the fact that GUI tests generally interact with a GUI from a different process. This means that the test will not have access to the internals of the GUI it is testing. Instead of simply calling a method on an object, it's necessary to first locate that object within the GUI. This is generally done by traversing the tree of graphical elements from the root window object until a widget matching details of the desired widget – as it appeared when the test was created – is found. Additionally, because of this cross-process testing, it's rare for all information about a widget to be exposed. For example, the Button class as implemented in Windows Presentation Foundation[16] can be tested through the InvokePattern interface in the Windows Automation API[17]. The Button itself has 136 properties, but InvokePattern exposes only 20 of these for use by test code. It can be difficult to create strong tests in the (common) case that one of the properties that isn't exposed is important to the functionality of a feature.

With these concerns in mind, we need to consider what the purpose of our GUI testing actually is. There are two distinct forms of GUI testing: testing the look of the GUI; and performing system testing of the application through its GUI. Take for example the Wikipedia entry for GUI testing, Fig. 13. If we want to verify that on this page that the Wikipedia logo appears as the upper-leftmost widget, that directly below it is a "Main page" hyperlink, and so on, we are testing the look of the GUI. If instead we want to verify that clicking on the link to software engineering in the first paragraph takes us to a page titled "Software engineering – Wikipedia, the free encyclopedia," then we are testing the functionality of the system as a whole. Again,



Graphical user interface testing

From Wikipedia, the free encyclopedia

In software engineering, **graphical user interface testing** is the process of testing
meets its written specifications. This is normally done through the use of a variety

**Contents** [hide]
1 Test Case Generation
2 Planning and artificial intelligence
3 Running the test cases
4 See also
5 References
6 External links

Test Case Generation

To generate a 'good' set of test cases, the test designers must be certain that thei

**Fig. 13.** A sample GUI[18]

[16] `http://msdn.microsoft.com/en-us/library/ms754130.aspx`
[17] `http://msdn.microsoft.com/en-us/library/dd561932(v=VS.85).aspx`
[18] From `http://en.wikipedia.org/wiki/Graphical_user_interface_testing`

for the present, let us consider the second of these approaches. Essentially, we are performing ATDD through the application's GUI instead of below it.

It is possible to write GUI tests for use in test-driven development of a GUI manually using available GUI testing tools. For example, it is entirely possible to write a Selenium[19] test by hand before a GUI exists even though Selenium is primarily a capture/replay tool (CRT) – a testing application that records a series of interactions with a system and records them in a format that can be replayed later as a test. This approach has been supported in the past by tool like TestNG-Abbot [22] and FEST [23], but has not received widespread uptake. This could be due to the fact that test authors need to know a large amount of detailed information about the GUI to be created in order to write a test.

A simpler approach to UITDD involves the creation of an automated low-fidelity prototype using a program like ActiveStory Enhanced [24] or SketchFlow[20]. These prototypes are event-based GUIs that respond to user input in the same way in which actual GUIs do. This means that they generate events when a user interacts with them. These events can be captured using a CRT, like white[21] or LEET [25], in the same way in which they can be used to record events from an actual GUI. These events can then be replayed on the actual GUI, with one caveat: the elements in the prototype that are generating events need to have the same identifying information as the equivalent elements in the actual GUI.

Consider for example the prototype shown in Fig. 14. It was created in SketchFlow, which means that each widget will raise recordable events when interacted with. We can use this prototype both for testing the actual GUI and testing the actual application through its GUI. From the prototype, we can use information about, for example, the arrangement of widgets to create tests of the GUI, or we could use the functionality demonstrated through the prototype to create acceptance tests of the actual application. For example, we can fill in the fields as shown in Fig. 14, then click the "Clear Report" button and verify that the fields have been cleared. We can then use this test for verification of both the form and functionality of the actual application, Fig. 15.

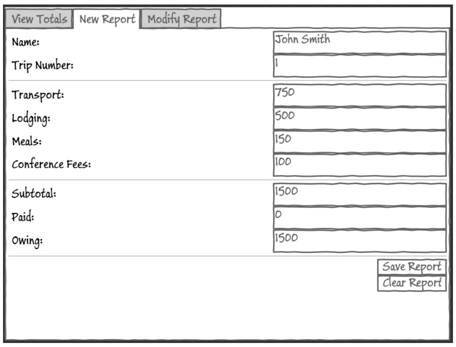
There are several advantages to this approach. First, this approach to UITDD has the advantage of being able to make use of CRTs, which makes it much easier to create tests than it would be to create them by hand. Second, by integrating medium-fidelity prototyping into the TDD process, we are creating another opportunity for testing – usability testing, as described in Section 2.3.3. This means that it is possible to detect usability errors early in the development process, which not only makes them cheaper to fix, but also reduces the number of changes to the GUI that will be necessary later in the software development process. This reduces the risk that changes to the GUI will break GUI tests since there will be fewer of them. Third, this

[19] `http://www.seleniumhq.org`

[20] `http://www.microsoft.com/expression/products/sketchflow_overview.aspx`

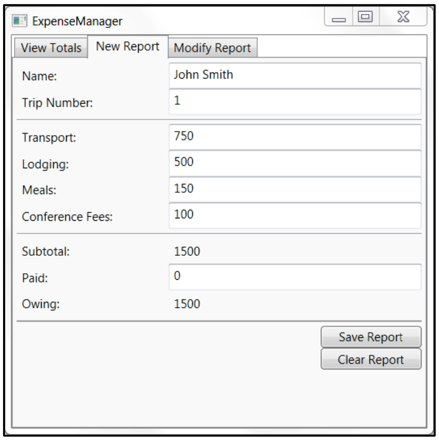
[21] `http://white.codeplex.com`

approach reduces the apparent complexity of the application we're testing by helping us specify which parts of it are going to be important early on. Only the important flows of each feature of the application will be shown in a low-fidelity prototype, so we will automatically know which parts of the application (and which sequences of events) we need to focus on when we are recording tests.



Name: John Smith
Trip Number: 1
Transport: 750
Lodging: 500
Meals: 150
Conference Fees: 100
Subtotal: 1500
Paid: 0
Owing: 1500

**Fig. 14.** Prototype of Expense-Manager's GUI



Name: John Smith
Trip Number: 1
Transport: 750
Lodging: 500
Meals: 150
Conference Fees: 100
Subtotal: 1500
Paid: 0
Owing: 1500

**Fig. 15.** Implementation of the GUI of ExpenseManager

## 3.4 State Space of Testing

The state of a program is the set of values of all variables defined and instantiated by that program. The state of the system grows when new objects are instantiated and shrinks when garbage collection takes place. In an object-oriented application, the state should be represented as a graph with each node containing a set of objects, each of which also contains the state of each of its fields. The state space of an application, then, is the graph of all possible states that the application can enter. Method calls cause the application to transition from one state to another by changing the values of variables. In a completely deterministic program, the state space would be a linear sequence of states leading to a single terminal state. Whenever a program can be influenced by outside factors – such as interaction with the file system, input from a user, or any number of other events – then the state space will become a graph with multiple edges between many of its state nodes. The more possible states there are, the larger the state space becomes.

For example, consider Fig. 16. In this example, the states of the system are shown within square brackets with highlight boxes surrounding new information and method calls that cause transitions are described within callouts to the numbered arrows. Note that each method call has a discrete – and testable – effect on the overall state of the system.

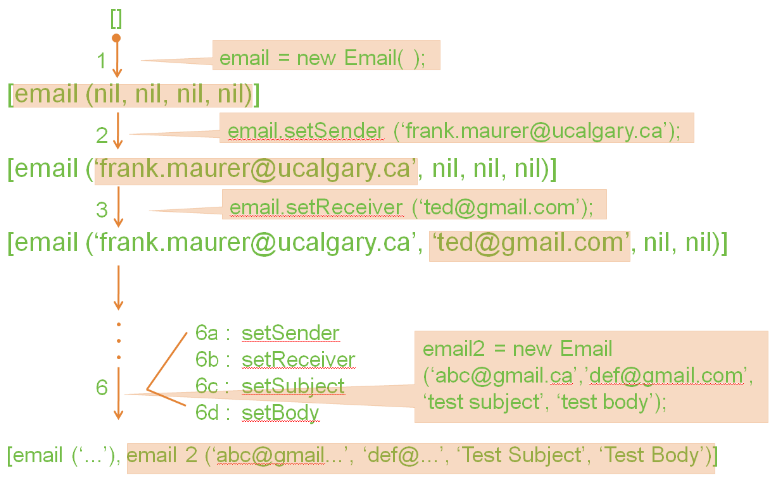

[]
1
email = new Email( );
[email (nil, nil, nil, nil)]
2
email.setSender ('frank.maurer@ucalgary.ca');
[email ('frank.maurer@ucalgary.ca', nil, nil, nil)]
3
email.setReceiver ('ted@gmail.com');
[email ('frank.maurer@ucalgary.ca', 'ted@gmail.com', nil, nil)]
6
6a : setSender
6b : setReceiver
6c : setSubject
6d : setBody
email2 = new Email ('abc@gmail.ca','def@gmail.com', 'test subject', 'test body');
[email ('…'), email 2 ('abc@gmail…', 'def@…', 'Test Subject', 'Test Body')]

**Fig. 16.** A subset of the state space of a sample email application

When viewing a system as a state space to be explored, the goal of a software tester is threefold:

- To map out and understand the system
- To set up automated tests that will detect new bugs introduced by changes
- To search for new bugs in unexplored regions of the state space

The difficult with GUI testing from this perspective is: how can our tests cover the important parts of the state space effectively when we can only create so many tests? This concern will be addressed in more detail in the following section.

There are many issues to consider when viewing an application under test as a state space to explore. First, what starting state should be used? Many applications, for example web sites or document viewers, can be accessed initially in many different ways. Second, the state of an application may not be entirely visible to test code. For example, with black-box GUI tests, it's possible to make verifications regarding the state of the GUI, but not about the state of the underlying application. This makes it difficult to actually determine whether features are entirely working, or even if a test was able to navigate through the state space to the correct state. Third, as an extension of this last point, an application's state space should be viewed as a subset of the state space of the computer as a whole. Interactions with the file system, network, or a database, processing and threading timings controlled by the operating system, and even time can be an important part of the state of an application. Finally, once we've chosen a starting point, how do we move the system into a state in which we are interested? For some applications, like websites, it's possible to navigate directly to a desired state. For other GUI-based applications, there may be only one starting state, and there may be many intervening states between the starting state and a state that we want to test. Changes to these intervening states can cause a test to fail when the functionality the test was initially intended to test is still working. This is because the test isn't able to navigate to the correct state. In such a case, either the test will generate an exception by trying to perform impossible actions on the GUI, or verifications will fail because they are being run on a different state then they were intended to. Both of these failures can occur when the system is actually working correctly. An illustration of this can be found in Fig. 17.

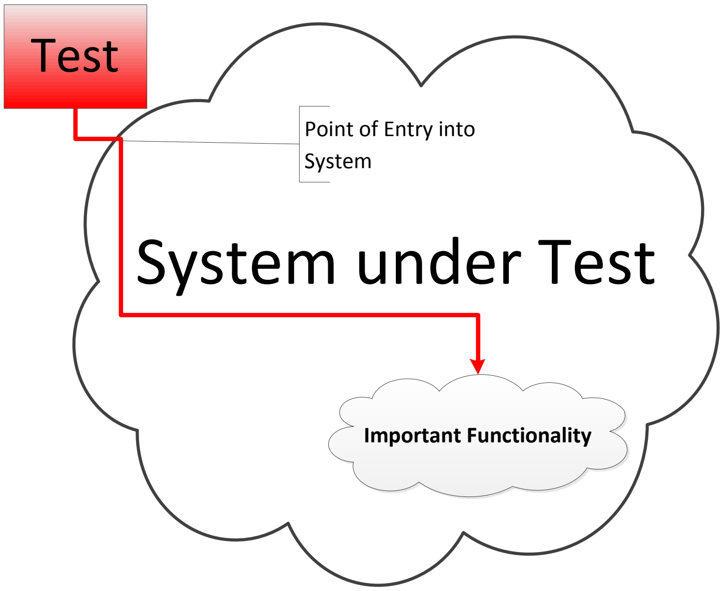


**Fig. 17.** Tests may need to traverse significant portions of the state space to reach and test interesting functionality

## 3.5 Test Quality

There are a variety of methods available to determine how good our testing is. Two of the most popular are code coverage and mutation testing.

Code coverage is a measure of how much of the system a test suite actually enters during testing. However, there are many different types of code coverage. The most lenient definition is line coverage (also known as: statement coverage). When code coverage is referred to in an agile environment without specifying what kind of coverage metric is being used, this is the type that is meant. Line coverage is a measure of the number of lines of the application that were executed during a test run, but doesn't account for the quality of that execution. For example, consider an "if" statement that can resolve in two distinct ways. From the perspective of line coverage, it doesn't matter which way the condition is resolved – the if statement itself will be considered covered either way. Close to the other extreme, we have multiple condition / decision coverage (MC/DC). In MC/DC,

- Every result of every decision (set of conditions) must be evaluated,
- Every result of every condition must be evaluated,

- Each condition in a decision must be shown to independently affect the result, and
- Each entry and exit point of the program must be used.

This method of evaluating code coverage is very exact, and is used in instances where a software failure would have catastrophic consequences – such as software used in guidance and control of aircraft. As with all code coverage metrics, the goal is to get as many states within the application's state space as possible visited by test code, if not verified in detail.

Mutation testing, on the other hand, is a very different approach to checking the quality of a test suite. In mutation testing, we actually modify parts of the system, then run our tests against this "mutant." If our tests are not strong enough to realize that the system has changed, then the tests need to be modified. By combining mutation testing and code coverage, we can get information not only about which parts of our system haven't been tested yet, but also when our verifications about a state aren't strong enough.

## 3.6 Testing Graphical User Interfaces – A State Space Explosion

As was alluded to in the previous section, state spaces tend to be very large. This is especially true when we consider the state space of a GUI-based application. Consider, for example, the primitive calculator application shown in Fig. 18. How many visible widgets does it contain? How many properties do you think are contained in all of those widgets, and how many methods are there to call?

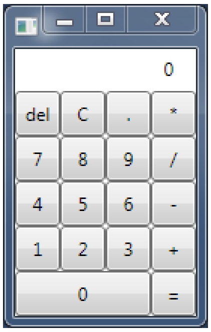


**Fig. 18.** A simple calculator application

In this application, there are:

- 19 visible widgets,
- 2577 properties of these widgets, and
- 4007 methods that can be called on these widgets.

In this supposedly-simple application, there are thousands of properties that go into the definition of **each** state that the system can enter. This means that it would not be possible attempt to visit every state of the application and verify every property. Considering the fact that GUI errors do impact customers, and that GUIs for applications that are used in the real world are significantly more complicated than this example, it's extremely important to think about GUI testing in terms of exploring the state space of an application in a way that will provide a good return on investment for our effort.

So, how do we deal with the fact that it's easy to create systems which will be impossible to conclusively test? We have to prioritize which parts of the state space are more likely to contain bugs or are susceptible to the future introduction of bugs. These parts of the state space are more important than others – for example, division by zero is a significantly important concept in our calculator application, and every attempt to divide by zero from any state should result in a transition to the same state (a state in which "cannot divide by zero" or the equivalent is displayed).

The corollary to this is that, where we can identify parts of a system that introduce *unnecessary* complication into the state space, we can encapsulate this complexity using mocking (sometimes referred to as isolation). That is, we can encapsulate complexity that is not essential to the purpose of a test so that our tests are more reliable, simpler, and have fewer dependencies. Mocking frameworks, such as Moles[22] or jMock[23], allow us to detect when a complex object would be created and instead replace it with a mock object. Mock objects can be interacted with by other objects in the same way as the object they are replacing. However, they will return a predetermined value instead of interacting with other parts of the system. Additionally, mock objects can record the parameters used in method calls into their methods so that we can verify later on that other parts of the system are interacting with the object we are mocking in an appropriate manner. In essence, this allows us to reduce the size of in individual state or of the state space as a whole by replacing a set of complicated, difficult-to-verify objects or chains of method calls with a single, predictable object or state. An illustration of the latter is provided in Fig. 18.

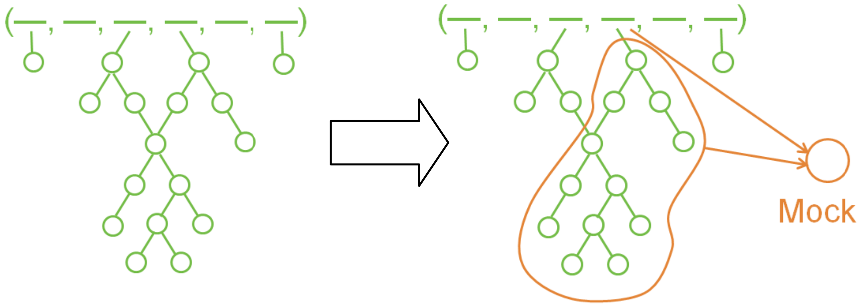


**Fig. 19.** Replacing part of the state space with a simple mock

[22] http://msdn.microsoft.com/en-us/library/ff798506.aspx
[23] http://jmock.org/

For example, when a feature interacts with a database, the state space of the database becomes part of the state space that we are testing. This can slow down test execution and cause confusing test failures for a host of reasons that are completely unrelated to the purpose of the test: make sure the feature is working. Testing the feature inclusive of the database additionally tests the network connection to the database, the database itself, the database contents etc. Rather than putting up with this additional complexity, we can simply mock out the portion of the system that relies directly on this database and instead work with predefined, predictable data. Mocking can be used to avoid a range of complications that regularly complicate testing, from database and networking issues to file access to testing features that depend on a specific date to testing multi-threaded applications. This allows us to focus on the real question: given that all its dependencies are working, does our feature work correctly?

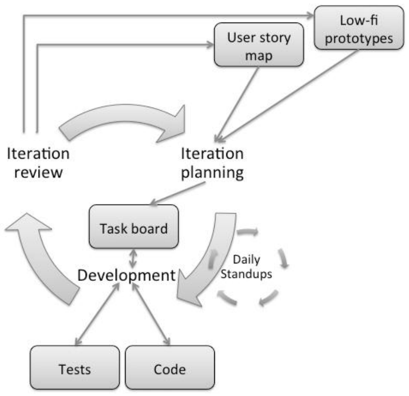


**Fig. 20.** Development process

## 4 Summary and Conclusions

This chapter gave an overview of core strategies used by agile software development teams.

We focused our discussion on two aspects: project management and quality assurance. Fig. 19 summarizes our discussion. It shows that agile teams use an iterative development processes with a daily feedback loop consisting of standup meetings. User story maps and low fidelity prototypes are used as cost-effective means of capturing the strategic view of the projects. These are updated based on new knowledge gained in the current iteration. The development team primarily delivers source code and tests. The team demonstrates the iteration results at the end of each development cycle to all stakeholders. This in turn is the basis for the next iteration planning meeting, which determines the goals for the upcoming cycle.

Agile processes are mainstream software development methodologies used by many teams within the software development industry. While they are no silver bullet and require substantial rigor and commitment from teams, they seem to be delivering results.

## References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
2. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press (2004)

3. Poppendieck, M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley (2003)
4. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall (2002)
5. Stapleton, J. (ed.): DSDM Consortium: DSDM: Business Focused Development. Pearson Education (2003)
6. Cockburn, A.: Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley (2004)
7. Highsmith, J.A.: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House (1999)
8. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, Boston (2001)
9. Mugridge, R., Cunningham, W.: Fit for Developing Software: Framework for Integrated Tests. Prentice Hall (2005)
10. Buxton, B.: Sketching User Experiences: Getting the Design Right and the Right Design. Morgan Kaufmann (2007)
11. Kelley, J.F.: An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications. ACM Transactions on Office Information Systems 2(1), 26–41 (1984)
12. Cohn, M.: Agile Estimating and Planning. Prentice Hall (2005)
13. Marick, B.: When Should a Test Be Automated? In: Proceedings of the 11th International Software Quality Week, San Francisco, vol. 11 (1998)
14. Jeffries, R., Melnik, G.: Guest Editors' Introduction: TDD - The Art of Fearless Programming. IEEE Software, 24–30 (2007)
15. Memon, A.M.: A Comprehensive Framework for Testing Graphical User Interfaces. PhD thesis, University of Pittsburgh (2001)
16. Robinson, B., Brooks, P.: An Initial Study of Customer-Reported GUI Defects. In: IEEE International Conference on Software Testing, Verification, and Validation Workshops, pp. 267–274. IEEE (2009)
17. Xie, Q., Memon, A.M.: Using a Pilot Study to Derive a GUI Model for Automated Testing. ACM Transactions on Software Engineering and Methodology 18(2), 1–35 (2008)
18. Memon, A., Banerjee, I., Nagarajan, A.: What Test Oracle Should I Use for Effective GUI Testing? In: 18th IEEE International Conference on Automated Software Engineering, pp. 164–173. IEEE (2003)
19. Memon, A.M., Soffa, M.L.: Regression Testing of GUIs. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 118–127. ACM (2003)
20. Memon, A.M.: Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. ACM Transactions on Software Engineering and Methodology 18(2), 1–36 (2008)
21. Holmes, A., Kellogg, M.: Automating Functional Tests Using Selenium. In: AGILE 2006, pp. 270–275. IEEE (2006)
22. Ruiz, A., Price, Y.W.: Test-Driven GUI Development with TestNG and Abbot. IEEE Software, 51–57 (2007)
23. Ruiz, A., Price, Y.W.: GUI Testing Made Easy. In: Testing: Academic and Industrial Conference - Practice and Research Techniques, pp. 99–103. IEEE (2008)
24. Hosseini-Khayat, A., Hellmann, T.D., Maurer, F.: Distributed and Automated Usability Testing of Low-Fidelity Prototypes. In: International Conference on Agile Methods in Software Development, pp. 59–66. IEEE (2010)
25. Hellmann, T.D., Maurer, F.: Rule-Based Exploratory Testing of Graphical User Interfaces. In: International Conference on Agile Methods in Software Development, pp. 107–116. IEEE (2011)

# Open Source Practices in Software Product Line Engineering

Frank van der Linden

Philips Healthcare, Veenpluis 6, 5684 PC Best, The Netherlands
frank.van.der.linden@philips.com

**Abstract.** This chapter presents a short introduction to software product line engineering. It describes experiences of introducing software product line engineering in industry followed by a discussion on some problems in product line engineering originating from the distributed organisation that is involved in many cases. It addresses how solutions originating from open source software development may be used to solve the mentioned problems, and it describes some cases where open source practices have shown to be very useful.

**Keywords:** software product line engineering, open source, inner source, distributed software development.

## 1 Introduction

This chapter is based on experiences of Philips Healthcare within a series of ITEA projects on software product lines – Esaps, Café, Families [13,14] – and one on collaborative development – COSI [3]. Results of these projects are applied to continuously improve the software product line development capabilities iof the company.

Originally the products of Philips Healthcare are imaging systems supporting medical diagnosis. Most of these systems need extensive image processing, storage, exchange and viewing. Images can be small – several kB – to very large – several GB, and they are still increasing in size. They may be 2D, 3D or even 4D images. Increasingly the products are meant to be used during intervention, meaning that massive image processing has to be performed with low latency.

Philips Healthcare has a large software development organisation – there are more than 1500 software developers. Development is done world-wide and is structured around product groups. However, cross product group reuse is necessary, since image processing and handling is important for most product groups. Philips Healthcare has set up a product line platform to serve all business groups; [17]. Most of the different business groups have set up a product line based on this platform.

The product line was started in 1998, based on a strategic decision by the management. Most of the well known reasons for introducing a product line also applied here: Reduce development effort and time to market, improve product offering and enable new system combinations.

The first step was to get enough management support. When this was settled, an architecture was needed. This architecture was defined by the architects of a small set of product groups. The initial platform was based on existing assets of these product groups. Before that time, each product group had its own architectures and effort was duplicated in the multiple development of very similar software functionality. The product line technology was aimed at the development of a common platform to be used by many product groups, and the management of reuse of components that are built for this platform.

The evolutionary introduction of the product-line was essential to be able to manage it. Evolution proceeds in two dimensions:

1. Provide more functionality
2. Involve more product groups

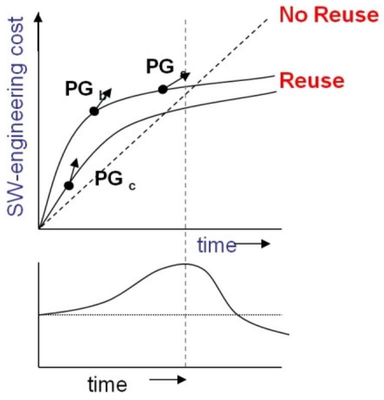


**Fig. 1.** Organisational tension rises, then drops steep once the breakeven point is passed

Evolution is necessary, as the introduction of the product line initially takes up additional effort of the involved development groups. The top of Fig. 1 gives schematic figure of the development cost at the introduction of a product line. The “no reuse” line shows that the cost for new functionality grows increasingly over time when reuse is not applied; old functionality has to be redeveloped again as well. The two curved lines show the costs when adopting the product line. As different groups have different investments and reuse possibilities, the curves will vary from the different groups. In any case initially there is an investment to integrate the platform in the own development. Only after successful integration the reuse will save cost. Typically, the tension in the interaction between the platform development group and the adopting group grows when investments are done, but there is still no pay back. The platform is seen to hold up the business. However after the breakeven point is reached, and it is recognised by the adopting group, tension will drop very fast. This is shown at the bottom of Fig. 1. In order to manage the tension well, it is best to spread the tension of the different groups over time. This is depicted by the dots with arrows at the top of Fig. 1. Different product groups are at different places in the adoption, and consequently the associated level of tension is also different.

After more than 10 years most product groups are willing to fund the platform groups. As newly acquired companies have been integrated, there are still groups in the course of adopting the platform. Based on measurements, it is found that the development of a reusable component takes about 1.6 times the effort with respect to the development of a component of a single product. However, the reusable components are used by many groups. So in case of reuse by 10 groups this leads for each of the product groups only 16% of the cost of making a component on your own. This knowledge supports the willingness to fund the platform by the product groups.

In the forthcoming sections give our experience on the main ingredients and benefits of software product line development. Specific attention is placed on variability management, distributed development and interaction with open source projects.

## 2 Software Product Line Engineering

*Software product line engineering* [11,17,19,21] is an answer to the increasing demand for individualised software-intensive systems. High quality individualised systems need to be developed at low cost, with short time to market. Software product line engineering provides an answer through a pro-active reuse of all development artefacts. In particular, this involves a platform and the architecture. However proactive reuse implies strategic, planned and consistent reuse of all artefacts, including requirements, features, components, code, test cases, etc. This has to be supported by commonality and variability management that links specific variants of the system to the artefacts. In Experienced advantages of product lines report large improvements in product, maintenance and training cost, productivity, lead-time and quality

Software product line engineering is supported by a development process set-up that distinguishes two interacting development processes: domain engineering and application engineering. The former is meant to systematically develop variability and artefacts to be reused (by others); the latter is to develop applications, applying variability and reusing these artefacts.

### 2.1 Commonality and Variability

*Commonality* is defined as “a list of assumptions that are true for all product line applications” [5,25]. *Product line variability* is defined as “Variation (differences) between the systems that belong to a product line in terms of properties and qualities” [18]. Commonality will be developed during domain engineering, and reused unchanged during application engineering. This constitutes an important aspect of reuse, as common artefacts are to be used by all applications. The architecture and platform are artefacts that are part of the commonality of a product line. The variability expresses the variation between the applications of the product line. As variability has to be dealt with compositional, different applications may have the same common part of the variability. This also amounts to reuse, but to a lesser extent that the commonality. It is explicitly defined through distinguishing between *variation points* – what does vary – and *variants* – how does it vary. In several aspects a variation point can be seen as a type and a variant as an instance of the corresponding type[1].

Many stakeholders in the development process explicit decide upon what amounts to commonality and variability in product lines. Each stakeholder has his/her own view on what should be common and what should vary. For instance, the product management will decide based on market trends, user needs, business and technology strategy. The software architects and developers have insight in technical trends and possibilities and use that for selecting commonality and variability. In any case product line variability is a decision and not an inherent property of a development artefact.

Within variability management a distinction is being made between external and internal variability. *External variability* is visible to the customer. Their choice of adopting the system will be supported by this external variability. Consequently, it is related to the business strategy and external variability is mainly determined by the product management. *Internal variability* comes from different development stakeholders, and is not of importance to the customer. Keeping it hidden will reduce the complexity of the communicated variability to the customer. They are usually introduced due to technical reasons during the development process. They may involve choices of underlying hardware, system software, middleware and tools. Sometimes it is influenced by the need to use legacy software for certain systems. They will also be related to the choices made for dependability requirements like performance, security, etc.

Variability is refined in several stages. Variability is initially defined during the requirements phase. This variability needs to be mapped to variability in architecture. This can further be refined to variability in the components, and the code. Finally variability needs also be mapped to the test artefacts. Each of these mappings is usually *n-m*, where *n* and *m* are small numbers. Keeping these numbers small is important to facilitate variability management. Note that during the mapping variability will be refined and new *internal* variability will be added.

In order to keep a consistent view on variability it needs explicitly be modelled in a compositional way. This model has to be related through traceability to the development artefacts, in order to support variant the selection process. We have proposed an Orthogonal Variability Model (OVM) as a language for defining variability models [21]. The orthogonal variability model is independent of the variation mechanisms in other development models. This is crucial, since variability mechanisms in different models may not easily be mapped on each other. The OVM is *not* compositional – it only groups variation points with variants, but compositional mechanisms can be added to it[2].

[1] This is still to be investigated. Variation points and variants are concepts usable in all stages of development. Consequently variation points should not be mixed with types in a programming language.

[2] This has to be investigated. The most important decision is on how the variability interfaces between the components should be described.

## 2.2 Double Lifecycle

Software product line development distinguishes responsibilities for *domain engineering* – introducing reuse and variability – and *application engineering* – exploiting reuse and variability. These responsibilities are supported by two interrelated, but distinct, processes; see Fig. 2. As domain and application engineering have different paces and requirements, they may use different methods and tools.

Application engineering usually has several instances active at the same moment, each of them developing for different products. Application Engineering deals with short term concerns of delivering applications to the market. It reuses the domain artefacts following to the domain engineering architecture and it defines the binding of variability for individual applications. Application engineering consist of a multitude of are fast processes that are aimed to deliver a single product.

Domain engineering usually has only one instance, although sometimes some subsystems have their own domain engineering. Domain engineering is, generally, dealing with long-term concerns of the product line. One important task is to define the scope of the product line. Commonality and variability is based on the scope. Domain engineering develops reusable domain artefacts incorporating commonality and variability. Important artefacts are the architecture and a product line platform. Domain engineering is a continuous process increasingly introducing new common software in the platform.

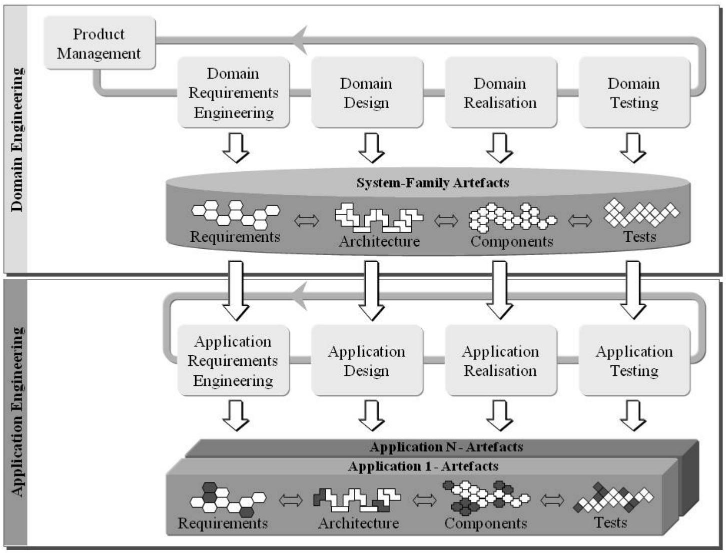


**Fig. 2.** Two development processes; copy from [21]

Special care is needed for the collaboration between domain and application engineering. Domain engineering delivers reusable artefacts, commonality and variability. These have to be adopted by the application engineering. Conversely, application engineering provides feedback on the quality of the delivered domain artefacts. In addition application artefacts may be candidate to become reusable. Only when the collaboration between domain and application engineering is managed well these interactions can work well.

## 2.3 BAPO

There are four main aspects of software engineering, that all need to be addressed consistently to ensure effective, efficient and correct development. These aspects are: Business, Architecture, Process and Organisation (BAPO); for more details, cf. [1,17]. It has to be considered that decisions in one aspect usually have consequences to the other aspects as well. Stakeholders representing these four aspects all need to be involved in software engineering decisions. In particular, good software product lines engineering needs to consider the BAPO aspects as well. In [17] we have described an extensive description of all BAPO aspect in software product line engineering. In particular, it describes the change of concerns for these aspects during the evolution of a product line.

The *business aspect* deals with costs, profits, strategy and planning. Product line engineering connects many projects and departments in the company. This means a serious investment, and it will influence the strategy, marketing and the financial aspects of the company. In particular, not only the software developers are influenced, but eventually almost everybody working for the company will be influenced by software product line engineering.

Domain engineering is not profitable in itself. It only becomes profitable when it is applied by application engineering. The business needs to provide funding for domain engineering using profits of application engineering. This often leads to decisions on making reusable on what pays itself back. It is a strategic business decision what will be reused (scope of the product line) and what will be left to application engineering. It is dependent on functionalities that will go to the market, in short and medium term. If this is not done right either too much effort will be put in the platform, but the level of reuse is too low, or the platform is too small, and double development still takes place in application engineering. As it is important to know the effectiveness of the development is. The business aspect may set up metrics, and measures the performance of the product line development.

The *architecture aspect* deals with the technical means to build the software. One of the most important assets of the product line is the product line reference architecture that determines the organisation of components and their interaction and the development rules for the complete product line. Architecture decision influences system's quality attributes for all applications, the commonality and variability. In many cases the architecture determines fixed interfaces within the family, which eases exchange of components, and thus supporting ease of reuse. To facilitate application engineering, domain engineering architects may decide on the use of specific tools to support e.g., domain specific languages, or model based development. Important in variability management is the use of configuration methodology and/or tools to simplify binding of variants to variation points, and do this in a consistent manner.

Third party software plays a role in most product lines. Presently it is unrealistic to expect that all software is built by the company. The architecture makes important decision on which third party software to use, for which purpose and how to use it. These are far reaching decisions influencing all application building groups.

The *process aspect* deals with roles, responsibilities and relationships in the development of the product line. For both domain and application engineering, development roles (requirements, architecture, development, testing) are determined, but also roles for the collaboration processes between application and domain engineering need to be determined. Important collaboration roles are:

- that domain engineering needs to ensure that the platform used and the architecture rules applied well
- Roadmaps of domain and application engineering need to be aligned. Application groups have new feature requests, and it is important to know when these features will be delivered in the platform
- Many issues and problems with domain artefacts originate from application engineering, where these artefacts are deployed in real systems. Consequently the management of change request is a collaborative issue.
- Application engineering develops application specific artefacts. Sometimes these artefacts of interest for other applications. In that case it would be best to promote this software to become a domain artefact. Domain and application engineering need to collaborate to promote artefacts from application to the domain. To reduce risk, a promising technology will sometimes first be introduced in a specific application only. When the introduction is successful, the related aspects need to be promoted in the domain as well.

The process aspect has to provide for processes in which these collaboration roles can be active.

The *organisation aspect* deals with people and organisational structures to support the product line development. It distributes responsibilities and roles over people and organisations. Product line engineering introduces a matrix structure with disciplinary roles in one dimension, and products in the other dimension; see Fig. 3. The organisation needs to map the matrix on the hierarchical company structure, taking into account that people that are placed together will collaborate better that those that are located far away of each other.

The placement of people influences the behaviour of the development groups with respect to responsibility for profit and loss, accountability and funding. As reuse does not provide direct income, the responsibility for reuse needs to be captured on an organisational level. It can be solved in several ways. In a small organisation all can be done in a single group. For larger organisations this is not possible. In many cases domain engineering is done a single group or department. In that case the cost of domain engineering is clear equal of the cost of that department, and funding can directed to this department. However, this has as disadvantage that the application group his direct influence on the domain engineering group, and therefore the flow of requirements and funding towards domain engineering, and the flow of domain artefacts towards the application group has to be carefully managed. Another solution may be that domain engineering is part of a single application engineering group. This has as advantage that domain engineering is closer to the market, and the flow of information, at least with its own application group, is good. It has ad disadvantage that the own group will be served first, leading to problems serving the other groups.

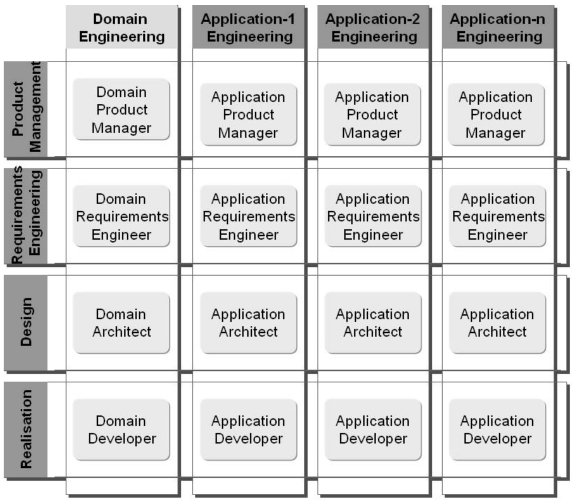


**Fig. 3.** Product line matrix organisation; copy from [17]

In many cases it is important to counter the disadvantages by introduction of a virtual organisation in addition to the actual organisation. Teams involving people form different organisation units form a "virtual" team in the other dimension. For instance in an application organisation, virtual teams may be set up for functional issues, such as a virtual architecture or testing group.

## 3 Distributed Development

Product line development is often *distributed* development. This has several causes. First, product lines are usually large – dealing with a large part of the portfolio of a company. Hence, there are many people involved in its development. For many reasons companies cannot place all these people to a single room, and thus the product line needs to be developed within distributed sites. Secondly, software shifts to commodity: Software that was originally differentiating gets to be obtained as commodity – [10,16], consequently 3rd party software is increasingly used within the product line. Consequently large heterogeneous groups are working on the same software. This leads to a complex situation like the one shown in Fig. 4. The dark ellipses, inside the dotted box, denote the different development groups inside the company. The light ellipses denote external development groups. Interaction – including sharing of software – is denoted by arrows.

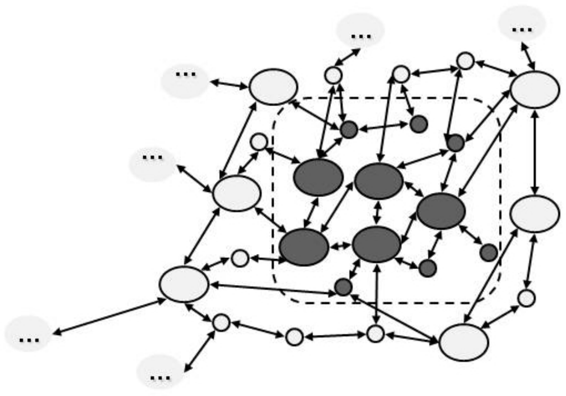


**Fig. 4.** Distributed development

## 3.1 Collaboration

An important aspect of distributed development is collaboration. Software is made together with external parties that are more or less trusted. To deal with the different levels of trust, it is important to know how much trust is needed for different parts of the software. Fig. 5 provides a means to manage this. It shows the landscape of technology commodification and collaboration; cf. [16]. The vertical axis deals with commodity and differentiation. In all software only a part is *differentiating*. This part is used by the company to distinguish itself from the competition, and this is where the profit of the product originates from. This kind of software is situated at the top of the diagram. Another part, at the vertical middle part of the diagram, is only *basic to the business*. It is domain related software that each company needs to have in the products, otherwise it will not be accepted by the clients. However, since the competition also provides this software, not much profit can be generated from this. Finally, at the bottom of the diagram, there is an amount of *commodity* software. This is software that can be found in almost all software products. Examples are system, communication and data storage software. All this software is necessary to let the product work, but also here no profit can be obtained from its presence.

The horizontal axis in Fig. 5 shows the different ways to *collaborate* on software. To the left hand side, there is no collaboration. Software is developed in the company itself. In the horizontal middle part the collaboration of software is done between companies that made agreements on working together. Finally, at the right hand side software is situated that is developed in open communities.

In total we have 9 regions in the landscape of collaborating of different kinds of software. In this landscape there are two regions that are better to be avoided. As the borders of these regions are not very sharp, their borders do coinciding exactly with borders of the 9 regions discussed above. The first region is located at the lower left-hand side of Fig. 5. It describes the region where the company itself is producing commodity software. The disadvantage of this is that the company is using its own personnel and resources to produce commodity, which can be obtained elsewhere and often cheaper, and in shorter times. Using the personnel to produce differentiating software generates much more added value. The top right hand region is also to be avoided. In that case differentiating software is produced in an open community. This means that the competition can get hold easily of the software, and the own competitive advantage, and related IP value, will be diminished.

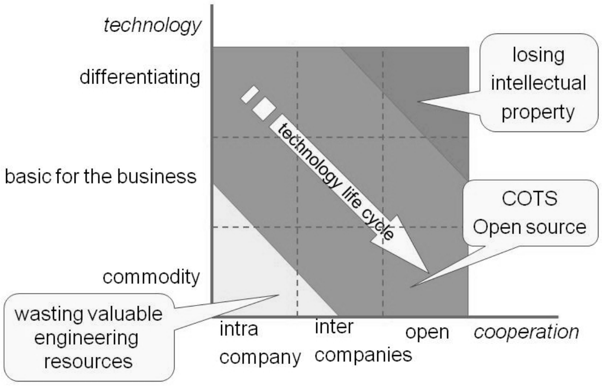


**Fig. 5.** Commodification of software

However, it is observed that over time any software is moving from top to bottom in the landscape of Fig. 5; see also [16]. Software that was originally differentiating is moving to become just basic to the business, and later even commodity. Consequently, there is a growing part of the software that is commodity. To deal with the situation, the business has to decide at which moment it is best to move from left to right – involving more collaboration – on certain pieces of software, during its move towards commodity.

## 3.2 Business Aspect

Companies doing product line engineering are faced with the challenge to manage complexity of heterogeneous distributed development. Although this initially seems to be an organisational issue, also the other BAPO aspects will be involved to deal with this issue. In particular the business aspect needs special attention. Decisions on this aspect will have an influence on the tree other aspects: architecture choices, distribution of processes and distributed responsibilities. When considering the leverage of open source in a company, the business has to deal with relevant aspect of what to share and what to keep private. An issue is how much trust can be put in open source communities. It is important to determine which software is placed in the collaboration landscape of Fig. 5.

## 3.3 Open Collaboration

Within the COSI project [6] we have concluded that leveraging open source advantages is beneficial in the management of distributed development. This is not a surprise, since open source developments show that large and distributed groups are able to produce high quality software.

In open collaboration, the organisation is inherently distributed. Even more distributed than what is normal in companies. In addition, there is no formal reporting structure. However, this usually works very well. People are contributing because they get respect and acknowledgement in these communities and the role in development determined by meritocracy [20]. Collaboration in these communities is supported with several web-based community tools that support open exchange of information.

When the choice is made to collaborate, the business has to decide in which manner, and how to select the parties to collaborated with. In cases of open collaboration, it has to be clear what can be open and what should be kept proprietary. The business needs to decide on costs and profits, not only in the own organisation, but also with more or less trusted collaborators. In the case of open development, it has to become clear how to collaborate, how much effort to put in the collaboration, and what to donate to the community, to get best results.

There are 5 different ways, to leverage the advantages of open source development in distributed development for software product lines; cf. Fig. 6. These are:

1. Developing with OSS practices
2. Using OSS tools in developing products
3. Developing products containing OSS
4. Developing OSS products
5. Engaging and leverage the community

In the next section we elaborate these different ways, and give some examples from Philips Practice.

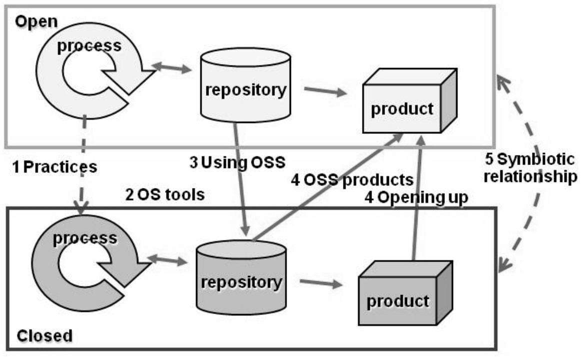


**Fig. 6.** Leveraging open source opportunities for product lines

## 4 Leveraging Open Source Opportunities

### 4.1 Developing with OSS Practices

A first way to exploit OSS is not to use open source software, but to use the practices instead. Open source development is intrinsically distributed, thus the open source development processes may also be made applicable for distributed development within companies. This cannot be done unchanged, since companies do have deadlines for products, accounting and decision hierarchies, and job appraisal mechanisms.

Many good elements of open source development can be taken over, and these can be translated within company borders. For instance, easy access to all information of the software, helps users to understand the software better, and is crucial for improvements and validating. Another aspect is the early and often release of software. This supports collaboration on the software, nut it also helps users to build on new software – although it may not be completely tested. Within open development, the accessibility of information supports the distributed ownership and control of assets. Those that are the best, and are available, will be contributing most. Applying open source methods within company border is called *Inner Source* [10,15,22,24] within Philips. It takes over the advantages mentioned above and uses normal organisation mechanisms for internal company issues, such as escalation of conflicts and the development and use of roadmaps.

The most important reason for Philips to move to inner source development was to resolve the organisational issue that domain engineering was becoming a bottleneck in product line development. Increasingly more business units are using the platform developed by the domain engineering group. Consequently they issued more feature and change request, but it was difficult to enlarge the domain engineering group. Moreover, the bi-yearly release of the platform was too slow for the application groups, since they could only build on the new features after they were released. This led to roadmap mismatches.

By taking the best elements of open source development, it was expected that the software engineers in the development departments will be working closer together, supported by direct communication. This supports sharing the platform knowledge and it eases the possibilities for the application groups to contribute to domain assets that are crucial to them. This reduces the dependency of the application groups on the domain engineering group. Components are to be developed in the department that has the best expertise.

To support this inner open collaboration, the platform documentation should be open. This improves trust in, and quality of, the platform. Consequently source code and relevant documentation was published on an internal website, which was easily accessible for all development departments. Note that also test suites and test results are part of the documentation, and published. This facilitates the use of regression tests for the platform and the applications built on it.

In order to involve users early, the platform software, which has biyearly thoroughly tested releases, also gives early access to earlier versions of the software. A label indicates the status of each version:

- *Works-In-Progress* are tested versions of the software updated bimonthly.
- *Snapshots* are low tested versions of the software, updated biweekly.
- *Bleeding edge* are untested version of the software, instantaneous accessible.

These early versions, including all relevant development information, are visible to the complete community. This facilitates early use of the software. Departments can already start developing upon new features of the platform before it is completely tested. This improves the time to market drastically. If errors in early platform releases are repaired, the new features can even be used before the official release date. Early feedback leads to faster updates and earlier high quality and good functionality. Application groups are encouraged to provide necessary changes in the components they use. A simple add-on script supports the patching of components. In general, priorities on what to adapt and by whom, is prepared by the discussion departments and mailing lists, but it is decided by the management of the involved departments.

In order to ensure clarity on the status of components developed in the distributed setting a set of rules regulates the *ownership and control* of software components. The owner of a component has the right to accept new versions of the components, but then he/she also accept the maintenance of it. Therefore acceptance should not be done lightly. Any user is allowed to change components. However, after changing a component, the ownership moves and the adapter get maintenance obligations for the changed component. The adapter may offer the adapted component back to the domain engineering group. The domain engineering group owns and develops components for the official releases of the platform. An adapted component may improve the platform, thus acceptance of adaptations should be considered. It is taken over when it is decided that the adaptations are beneficial to the other users as well. When the adaptation is only of interest for a single user, this user is allowed to keep the adapted version, but the maintenance burden stays by them.

In certain situations, e.g. when many change requests are dealing with the same set of components, the domain engineering group may decide that the platform needs refactoring, and a collection of related components needs to be adapted. In that case the development cannot be delegated to a single developer. A virtual team may be set up that has the assignment to make the change. Such a team will usually consist of people from the domain engineering group and some from application groups. This virtual team performs the adaptation in several stages, and each document and components change is visible to all the developers. An issue here is that several parts of the company hierarchy are involved. The higher management layers need to approve and facilitate the virtual team.

An important issue in inner source development is the proper distribution of costs of the development. The domain engineering group does not develop products, and does not have direct income. It needs to be paid by the application groups, e.g. through a licensing scheme. Application groups that contribute are initially responsible for the maintenance cost and effort for the adapted components. However, if they convince the domain engineering group of the wider usability of the adapted component, the domain engineering group may take the component back. This reduces the maintenance cost for the original contributor. In addition, a reduction of the license fee may be in place when an application department provides many good contributions to the platform.

The results of the use of inner source within Philips are that there is a steady growth of the number of users that are involved. In general about 50% of the involved users are active, this means that they are involved in discussion groups, provide documentation and/or provide improvements to components. From the start of inner source up to now, three times more application department served with a domain engineering group of the same size. The quality of the platform is improved largely; shown in a much lower defect ratio. The environment improves the feedback from application development departments, and they find defects early, leading to significant time to market gains. The new collaborative development environment boosted collaboration enormously, and we see many collaborations running at any time on many aspects of the platform. The formal help desk, which was in use before the inner source development was introduced, could be removed, as its role was taken over by mailing lists and discussion groups.

## 4.2 Using OSS Tools in Developing Products

A second important way to leverage open source within companies is the use of open source development tools. Using open source software in this way is often easy, since the used software is usually not part of the final product, making it easy to comply with licences. However, for maintenance reasons tools increasingly become part of product. However for maintenance reasons open source tools are shipped with the product, the licensing issue of the next section 4.3 have to be addressed as well.

Presently many good open source development tools already available, and these certainly will be used within in-company developments. Some good examples are: the eclipse tool [8], which may be extended with plug-in tools for many aspects of the development. Philips uses several open source tools for inner source development. For instance Subversion® [3] for version management and the Semantic MediaWiKi [22] for document exchange.

To support the inner source development within Philips, a complete environment of open and proprietary tools is in use. Access to the code is via the proprietary CollabNet® tool [4]. CollabNet is based on open source tools. It provides role based access control, mailing lists and discussion groups. Version control on code is supported via Subversion. Other documents and files are shared through Semantic MediaWiKi enabling the sharing of all kinds of document formats.

To reduce introduction risks, this environment was introduced evolutionary over Philips Healthcare. It led to a much simplified development process for the involved departments, and it was easy to learn, and had a minimal impact on other processes. It provides a scalable solution for global and inter-organizational collaboration, and it increases the collaboration agility in the company, and reduces administration overhead.

## 4.3 Developing Products Containing OSS

The use of open source components is similar to the use of other 3rd party software, such as COTS. As can be expected, open source and other 3rd party components are usually commodity software; cf. Fig. 5. As the user of these components is not the driver of the development, and often not involved at all, the user also does not have much control over the functionality and quality. In particular, the compliance of 3rd party solutions to the own architecture standards and interfaces is a continuous issue. To ensure compliance to the latter, wrappers can be used.

In order to prepare the future use of any 3rd party components, it is important to have advanced knowledge on future releases. In the case of COTS components, this is usually done by having good contacts with the supplier, such as a direct contact person, visits to user conferences, etc. This will provide information on future features and quality issues, although guarantees will only come when the next version is delivered, sometimes as a pre-release or beta-test version. A serious issue is the disruptive nature of COTS releases. Architecture rules may have been changed, and backwards compatibility is only guaranteed for a few earlier releases. It often takes several moths till the new version can easily be integrated in new products.

In the case of open source components, advanced knowledge can be obtained via good contacts as well. However, in this case, good contacts are different; they will be maintained through involvement in the community. Although mailing lists and discussion groups will give already lots of information. Active involvement in these will lead to more precise information. Change requests and contributions may even lead to adaptations that are useful, but these will not always be followed; see also section 4.5. In any case, in this way the user has a limited control on the future of the components. An important advantage is the continuous evolution of open source. Disruptive changes are seldom, and when they come they are discussed intensively. This means that new features may be incorporated early, and even not completely tested software can already be used to build the own product. Issuing bug reports and corrections will keep open source at quality.

In all cases, licensing of open source is an issue. Licenses may have requirements on the status of the software it is connected to. Open source licenses may be in conflict with each other, or with proprietary licenses. Consequently, it is necessary to know which open source, under which licence is present in a product. In a distributed development case this means that open source needs to be communicated to everybody that uses it. However, it even is necessary for an integrator to inform all providers of the use of open source software, as his may be in conflict with other pieces of software that will be integrated.

In product line development it is an issue to introduce open source in domain or application developments. As open source is commodity software, it is likely that it can be used by many application development departments. Therefore it may be wise to integrate it in domain software. However, to reduce risks, new software in product lines is first only applied in a “trial” application. If it works well for this application, the open source can be integrated in the platform.

## 4.4 Developing OSS Products

Opening up parts of software of the product line is another way to leverage open source results. Especially, when software is becoming commodity, this is an option; cf. Fig. 5. This way, the involved software is given away, but in return there are many possible business advantages connected to it [2].

An important benefit is that others will be involved in the maintenance. When the software is becoming commodity, it is expensive to keep the maintenance of the software within the company. Successful open source software is maintained by the community. The company effort on maintenance can be reduced.

Opening up software may be useful if the software is supporting an (open) standard. When opening up is successful, it will lead to improve the interoperability with products of the competition. This is required by the users of the product that do not want to be dependent on a single supplier. Success is dependent on the need for this software, when there is already a competing open source solution for the same standard, the chance of success may be lower, it then depends on other qualities and the willingness of the competition to support this initiative.

In cases that there is not yet a standard, but a standard is needed, opening up the own solution may make it a de-facto standard, when it is opened up. As own products comply already with this software, it is easy to keep compliance to the proposed de-facto standard. In that case, again, the interoperability with products from the competition may increase.

The competition may have similar software on the market for a seemingly too high price. Opening up the own solution may be a way to devalue this propriety solution, as the own solution is open, and thus it is freely obtainable. In that case it also may lead to the interoperability benefits of above.

A more internal business reason is that opening the source of the software may drive acceptance of the software. The visibility of the source leads to possibilities of the user to inspect it, and improve it when needed. This will lead to an improvement of the trust in the product as well.

Similarly opening software may increase its quality, such as security or safety. Others in the community may have experience, and need for these qualities and may incorporate it in the open source. In addition, software that has a larger diversity of users will have more extensive tests, leading to better recognition and repair of the errors.

It may be the case that relevant 3rd party software is (in danger of) not being supported anymore by the supplier. Users of the software may decide that the software still needs to be maintained, and integrated with new developments. Together with the supplier it may be decided to open up the software and move maintenance from the originating company.

Opening up software may also support the sales of something else. For instance the software may be used for services that are the main source of income of the company. In that case the company can still focus on its own added value and still get the required quality software.

For example, consider the Philips DICOM[3] Validation Tool kit (DVTk) [7]. DICOM was introduced as an interoperability standard for professional medical imaging equipment in hospitals. It supported medical image exchange for many kinds of medical images. In 1995 the DICOM interface was considered a quality interface, it was only available a system option for the high-end systems. Providing DICOM support was differentiating for the companies providing medical imaging systems; cf. Fig. 7. In 1999 DICOM was no longer differentiating. The hospitals all needed interoperability and they just expected DICOM support in the equipment. Philips decided that it profitable that the competition follows the standard, and it provided the DVTk application binary freely downloadable via their own website. Soon it became clear that development and maintenance costs for the software became a burden, as the software did not provide direct added value. In 2001, a joint development started with another company (AGFA) to share development costs and increase adoption of DVTk. In 2005 it was decided to create an open platform to ensure uniformity. The software is still domain specific, it is regarded as commodity.

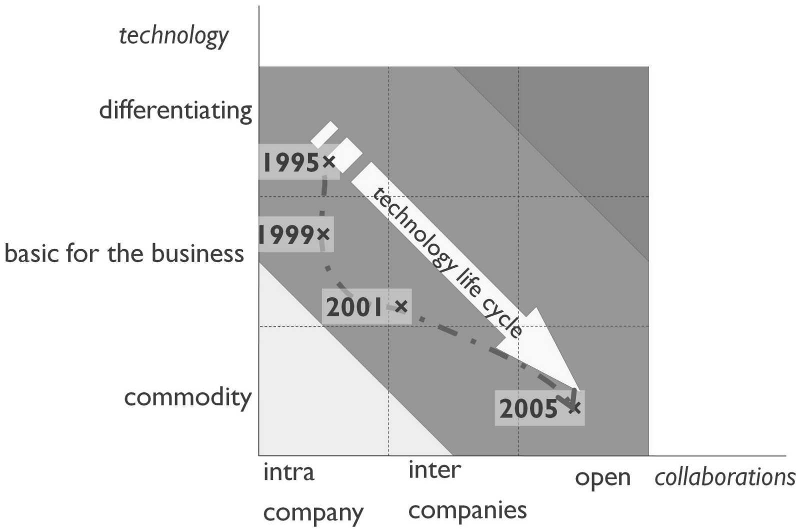


**Fig. 7.** Commodification of DVTk

This open source community is presently dealing with state of the art tools for interoperability in the medical imaging domain. To broaden its scope, it successfully applied to a commercial SW tool development tender for the IHE-Radiation Oncology Test SW and the IHE Gazelle Open Source tooling project.

[3] DICOM$^{SM}$ – Digital Imaging and Communications in Medicine `http://medical.nema.org/`

### 4.5 Engaging and Leverage the Community

A final way to leverage advantages of open source development is to have a symbiotic relationship with an open source community. This means that the company is involved in the open source community, and the open source software is important for the company. This asks for an active involvement in the community to ensure that the issues that are important for the company are addressed by the community. This not only means to use the open source software, and issue some bug requests, but also to donate software and patches solving issues. This means that the company provides people and effort to maintain the software at a high level.

An example of this is the involvement of Philips in the improvement of Subversion® [3]. Subversion is a version management system that is strong in distributed development. This makes it a good candidate for using it for the inner source development. However, the software also has some weak points that need to be repaired before it can be used. Luckily, Subversion is an open source development, and thus Philips people could join and address, and improve these observed weak points. The archive is globally accessible, giving single view on archive. The tool is optimised for low connectivity conditions, enabling access from almost anywhere. Another feature it is based on change sets, which enables departments not to accept certain changes, supporting the inner source way of working. It is easy to link archives from several origins together.

The weak points observed, originate from the fact that groups that were originally developing independent at distributed sites, are moving to work together on a single virtual distributed repository. Philips products may be used for many years, and similarly products must be supported for many years. Products will be upgraded during the lifespan, to keep the functionality and quality at high levels. Consequently, the version management has to deal with many active versions of the product line platform. Due to the high quality requirements on medical systems, regulatory bodies enforce strict regulations on how products are created. In particular traceability is the key to achieve quality.

The weak spot of Subversion (v.1.4) was that merging and rename handling was not working properly for the Philips traceability requirements. There was no built-in merge tracking, and renames were merged incorrectly. Merging between different archives was not possible. The involvement of Philips Healthcare in the Subversion community was mainly to improve these aspects of the tool. This lead to an improvement of the tool in version 1.5 that solved all issues on merge tracking and rename handing. This lead to a version management tool that is usable for the inner source community of Philips. It was adapted to requirements from Philips, but performed with joint effort of the Subversion community.

## 5 Conclusions

This paper describes product lines, and what are the advantages for companies to do so. It also shows important aspects to address during product line development: Business, Architecture, Process and Organisation. One of the organisational consequences of product line engineering is the issue of distributed development. Open source communities provide examples of well working distributed development. This gives ideas to introduce the benefits of open source within product line development. Several of the options of using open source for product line are addressed, in particular inner source that mimics the open source distributed development within company setting.

Presently inner source development is established within Philips Healthcare. In addition, other aspects of open source are also addressed, although on smaller scale. The adoption of inner source is performed stage-wise to reduce company risks. Groups that start involvement find it easy to take over the inner source way of working. Inner Source helped break the platform bottleneck, since using departments are able to create patches.

## References

1. America, P., Obbink, H., van Ommering, R., van der Linden, F.: CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering. In: Donohue, P. (ed.) 1st International Conference on Software Product Lines; Experiences and Research Directions, SPLC, pp. 167–180. Kluwer (2000)
2. Babar, M.A., Fitzgerald, B., Ågerfalk, P., Lundell, B., Thiel, S.: On the Importance of Sound Architectural Practices in the Use of OSS in Software Product Lines. In: van der Linden, F., Lundell, B. (eds.) 2nd International Workshop on Open Source Software and Product Lines (OSSPL 2007), Kyoto, Japan (2007) http://itea-cosi.org/modules/wikimod/index.php?page=OssPlas07
3. Apache™ - Subversion®, http://subversion.apache.org
4. CollabNet in.c, http://www.collab.net
5. Coplien, J., Hoffmann, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Software 15, 37–45 (1998)
6. COSI – Co-development using inner & Open source in Software Intensive systems, ITEA project 2005-2008, http://www.esi.es/index.php?hl=&op=14.4#cosi
7. DVTk – Dicom Validation toolkit, open source software supporting medical image exchange compliance, http://www.dvtk.org
8. The Eclipse Foundation, http://www.eclipse.org
9. Engelfriet, A.: Open Source and Open Innovation. Koninklijke Philips Electronics, NV handout: LinuxWorld Open Summit (2007), http://www.idc.com/nordic/downloads/events/linuxworld07/9%20-Arnoud%20Engelfriet.pdf
10. Jilderda, A., Rötschke, T.: Architecture Analysis Needs an Open Source Process. In: Ebert, J., Kullbach, B., Lehner, F. (eds.) Workshop Software Reenginering, Universität Koblenz-Landau, Koblenz (2001)
11. Käkölä, T., Dueñas, J.C.: Research Issues in Software Product Lines—Engineering and Management. Springer, Heidelberg (2006)
12. van der Linden, F.J., Wijnstra, J.G.: Platform Engineering for the Medical Domain. In: van der Linden, F. (ed.) PFE 2002. LNCS, vol. 2290, pp. 224–237. Springer, Heidelberg (2002)
13. van der Linden, F.: Software Product Families in Europe: The Esaps and Café Projects. IEEE Software 19, 41–49 (2002)

14. van der Linden, F.: Engineering Software Architectures, Processes and Platforms for System Families - ESAPS Overview. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 383–397. Springer, Heidelberg (2002)
15. van der Linden, F.: Applying Open Source Principles in Product Lines. Upgrade X, 32–40 (2009)
16. van der Linden, F., Lundell, B., Marttiin, P.: Commodification of Industrial Software: A Case for Open Source. IEEE Software 26, 77–83 (2009)
17. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer (2007)
18. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., Saval, G.: Disambiguating the Documentation of Variability in Software Product Lines: a Separation of Concerns, Formalization and Automated Analysis. In: Sutcliffe, A. (ed.) 15th IEEE International Conference on Requirements, pp. 243–253 (2007)
19. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering 5, 128–138 (1979)
20. Perens, B.: The Emerging Economic Paradigm of Open Source. First Monday 10(10) (2005), `http://www.firstmonday.org/issues/special10_10/perens/index.html`
21. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer (2005)
22. Sematic Mediawiki, `http://semantic-mediawiki.org/wiki/Semantic_MediaWiki`
23. Stellman, A., Greene, J.: Beautiful Teams. Inner Source, an interview with Auke Jilderda, ch. 8, pp. 103–111. O'Reily (2009)
24. Wesselius, J.: The Bazaar inside the Cathedral: Business Models for Internal Markets. IEEE Software 25, 60–66 (2008)
25. Weiss, D.M.: Commonality Analysis: A Systematic Process for Defining Families. In: van der Linden, F.J. (ed.) ARES 1998. LNCS, vol. 1429, pp. 214–222. Springer, Heidelberg (1998)
26. Wijnstra, J.G.: Critical Factors for a Successful Platform-Based Product Family Approach. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 68–89. Springer, Heidelberg (2002)

# Author Index