# Software Reuse and Product Line Engineering


**Eduardo Santana de Almeida**

**Abstract** Systematic Software Reuse is one of the most effective software engineering approaches for obtaining benefits related to productivity, quality, and cost reduction. In this chapter, we discuss its origins and motivations, obstacles, its success and failure aspects, and future directions. In addition, we present the main ideas and important directions related to Software Product Lines, a key reuse approach.

## 1 Introduction

Hardware engineers have succeeded in developing increasingly complex and powerful systems. On the other hand, it is well-known that hardware engineering cannot be compared to software engineering because of software characteristics such as no mass, no color, and so on (Cox 1990). However, software engineers are faced with a growing demand for complex and powerful software systems, where new products have to be developed more rapidly and product cycles seem to decrease, at times, to almost nothing. Some advances in software engineering have contributed to increased productivity, such as Object-Oriented Programming (OOP), Component-Based Development (CBD), Domain Engineering (DE), Software Product Lines (SPL), and Software Ecosystems, among others. These advances are known ways to achieve software reuse.

In the software reuse and software engineering literature, there are different published rates about reuse (Poulin 2006), however, some studies have shown that 40–60% of code is reusable from one application to another, 60% of design and code are reusable in business applications, 75% of program functions are common to more than one program, and only 15% of the code found in most systems is unique and new to a specific application (Ezran et al. 2002). According to Mili

E. S. de Almeida
Federal University of Bahia, Salvador, Bahia, Brazil
e-mail: esa@dcc.ufba.br; esa@rise.com.br



S. Cha et al. (eds.), *Handbook of Software Engineering*,
https://doi.org/10.1007/978-3-030-00262-6_8

et al. (1995), rates of actual and potential reuse range from 15% to 85%. With the maximization of the reuse of tested, certified, and organized assets, organizations can obtain improvements in cost, time, and quality as will be explained in the next sections.

This remainder of this chapter is organized as follows. In Sect. 2, we present the main concepts and principles related to software reuse. Section 3 presents an organized tour with the seminal papers in the field. Section 4 introduces Software Product Lines (SPL) an effective approach for software reuse, and, finally, Sect. 5 presents the conclusions.

## 2 Concepts and Principles

Many different viewpoints exist about the definitions involving software reuse. For Frakes and Isoda (1994), software reuse is defined as the use of engineering knowledge or artifacts from existing systems to build new ones. Tracz (1995) considers reuse as the use of software that was designed for reuse. Basili et al. (1996) define software reuse as the use of everything associated with a software project, including knowledge. According to Ezran et al. (2002), software reuse is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality, and business performance.

In this chapter, Krueger's general view of software reuse will be adopted (Krueger 1992).

**Definition** Software reuse is the process of creating software systems from existing software rather than building them from scratch.

The most common form of reusable asset is, of course, source code in some programming language, but it is not the only one. We can reuse several assets from different phases in the software development life cycle, such as (Mili et al. 2002):

- *Requirements*: Whereas code assets are executable, requirements specifications are not; they are the products of eliciting user requirements and recording them in some notation. These specifications can be reused to build either compound specifications or variations of the original product.
- *Designs*: Designs are generic representation of design decisions and their essence is the design/problem-solving knowledge that they capture. In contrast to source code, designs are not executable. On the other hand, in contrast to requirements, they capture structural information rather than functional information. They are represented by patterns or styles that can be instantiated in different ways to produce concrete designs.
- *Tests*: Test cases and test data can be reused on similar projects. The first one is harder and the idea is to design them to be explicitly reused, for example, in a

product family. Test data can be used to test a product with a similar set of inputs but different output scenarios.
- *Documentation*: Natural-language documentation that accompanies a reusable asset can be considered as a reusable asset itself. For example, documentation to reuse a software component or test case.

Once the different types of reusable assets are presented, it is important to differentiate between systematic × nonsystematic reuse, which can present different benefits and problems.

**Definition** Systematic reuse is the reuse of assets within a structured plan with well-defined processes and life cycles and commitments for funding, staffing, and incentives for production and use of reusable assets (Lim 1998). On the other hand, nonsystematic reuse is, by contrast, ad hoc, dependent on individual knowledge and initiative, not deployed consistently throughout the organization, and subject to little if any management planning and control. In general, if the organization is reasonably mature and well managed, it is not impossible for nonsystematic reuse to achieve some good results. However, the more problematic outcome is that nonsystematic reuse is chaotic, based on high risk of individual heroic employees, and amplifies problems and defects rather than damping them (Ezran et al. 2002).

Besides the kind of reuse that an organization can adopt, an asset can be reused in different ways, such as: black box, white box, and gray box reuse.

**Definition** If an asset is reused without the need for any adaptation, it is known as *black box reuse*. If necessary to change the internal body of an asset in order to obtain the required properties, it is known as *white box reuse*. The intermediate situation, where adaptation is achieved by setting parameters, is known as *gray box reuse*.

A key aspect for organizations willing to adopt systematic reuse is the domain.

**Definition** A domain is an area of knowledge or activity characterized by a family of related systems. It is characterized by a set of concepts and terminology understood by practitioners in that specific area of knowledge. A domain can also be defined by the common managed features that satisfy a specific market or mission (Mili et al. 2002).

**Definition** A domain can be considered *vertical* or *horizontal*. The first one refers to reuse that exploits functional similarities in a single application domain. It is contrasted with horizontal domain, which exploits similarities across two or more application domains. We can consider vertical domains areas such as avionics, social networks, medical systems, and so on. On the other hand, security, logging, and network communication can be considered horizontal domains (Ezran et al. 2002). It is important to highlight that this differentiation is subjective. An asset can be considered as vertical within a domain; however, if this domain is split into finer domains, the assets may be shared, and thus become horizontal.

In the systematic reuse process, it is possible to identify three stakeholders: the management, which initiates the reuse initiative and monitors the costs and benefits;

the development *for* reuse team (aka domain engineering), which is responsible for producing, classifying, and maintaining reusable assets; and development *with* reuse team (aka application engineering), which is responsible for producing applications using reusable assets.

In order to create reusable software, the development for/with reuse teams makes use of two concepts strongly related: *feature* and *binding time.*

**Definition** A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems (Kang et al. 1990). According to Kang et al. (1990), features can be classified as mandatory, optional, and alternative. Common features among different products are modeled as *mandatory features*, while different features among them can be optional or alternative. *Optional features* represent selectable features for products of a given domain and *alternative features* indicate that no more than one feature can be selected for a product.

Figure 1 shows an example of a feature model with mandatory, optional, and alternative features (Kang et al. 1990). Based on this feature model, we can derive different combination of products. `Transmission` and `Horsepower` are mandatory features. On the other hand, `Air conditioning` is an optional feature, thus, we can have products with this feature or not. `Transmission` has two alternative features, which means that we have to choose between `manual` or `automatic`. It is not possible to have both in the same product. Composition rules supplement the feature model with mutual dependency (requires) and mutual exclusion (excludes) relationships, which are used to constrain the selection from optional or alternative features. That is, it is possible to specify which features should be selected along with a designated one and which features should not. In the figure, there is mutual dependency between `Air conditioning` and `Horsepower`.

Other classifications can be found in the literature such as that of Czarnecki and Eisenecker (2000). However, in this chapter, we will use the original one defined by Kang et al. (1990).

**Definition** When we derive a product, we make decisions and decide which features will be included in the product. Thus, we bind a decision. Different

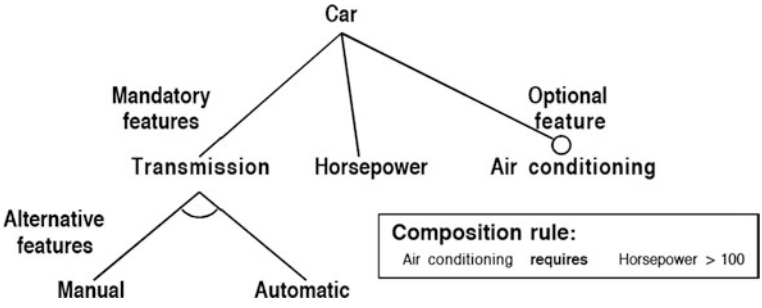


**Fig. 1** Features from the Car domain

implementation techniques (parameterization, configuration properties, conditional compilation, inheritance, and so on) allow binding decisions at different times, that is, they allow different *binding times*.

In their book, Apel et al. (2013) distinguish among compile-time binding, load-time binding, and runtime binding. Using an implementation technique that supports compile-binding time, software engineers make decisions of which features to include at or before compile time. Code of deselected features is then not even compiled into the product. Examples include the use of preprocessors (conditional compilation directives) and feature-oriented programming. With an implementation technique that enables load-time binding, software engineers can defer feature selection until the program is actually started, that is, during compilation all variations are still available; they are decided after deployment, based on command-line parameters or configuration files. Finally, some techniques even support runtime binding, where decisions are deferred to runtime and can even change during program execution. Examples of techniques that support load-time and runtime variability include reflection and context-oriented programming. There is not a best binding time and each one has advantages and disadvantages.

Once reusable assets are created, they should be made available somewhere.

**Definition** A repository is the place where reusable software assets are stored, along with the catalog of assets (Ezran et al. 2002). All the stakeholders should be able to access and use it easily.

A repository to store reusable assets offers the following advantages:

- The definition and common recognition of a place for assets, therefore, a known and unique place to look for and deposit assets
- A homogeneous way of documenting, searching, and accounting for assets
- A defined way of managing changes and enhancements to assets, including configuration management procedures

Mili et al. (1998) and Burégio et al. (2008) present important considerations on repository systems.

## *2.1 Software Reuse Benefits*

Software reuse presents positive impact on software quality, as well as on cost and productivity (Lim 1994; Basili et al. 1996; Sametinger 1997; Frakes and Succi 2001).

**Quality Improvements** Software reuse results in improvements in quality, productivity, and reliability.

- **Quality**. Error fixes accumulate from reuse to reuse. This yields higher quality for a reused component than would be the case for a component that is developed and used only once.

- **Productivity**. A productivity gain is achieved due to less code that has to be developed. This results in less testing efforts and also saves analysis and design labor, yielding overall savings in cost.
- **Reliability**. Using well-tested components increases the reliability of a software system. Moreover, the use of a component in several systems increases the chance of errors being detected and strengthens confidence in that component.

**Effort Reduction** Software reuse provides a reduction in redundant work and development time, which yields a shorter time to market.

- **Redundant work and development time**. Developing every system from scratch means redundant development of many parts such as requirement specifications, use cases, architecture, and so on. This can be avoided when these parts are available as reusable assets and can be shared, resulting in less development and less associated time and costs.
- **Time to market**. The success or failure of a software product is often determined by its time to market. Using reusable assets can result in a reduction of that time.
- **Documentation**. Although documentation is very important for the maintenance of a system, it is often neglected. Reusing software components reduces the amount of documentation to be written but compounds the importance of what is written. Thus, only the overall structure of the system, and newly developed assets have to be documented.
- **Maintenance costs**. Fewer defects can be expected when proven quality components have been used and less maintainability of the system.
- **Team size**. Some large development teams suffer from a communication overload. Doubling the size of a development team does not result in doubled productivity. If many components can be reused, then software systems can be developed with smaller teams, leading to better communications and increased productivity.

Ezran et al. (2002) presented some important estimates[1] of actual improvements due to reuse in organizations using programming languages ranging from Ada to Cobol and C++:

- **DEC**
  - Cycle time: 67–80% lower (reuse levels 50–80%)
- **First National Bank of Chicago**
  - Cycle time: 67–80% lower (reuse levels 50–80%)
- **Fujitsu**
  - Proportion of projects on schedule: increased from 20% to 70%
  - Effort to customize package: reduced from 30 person-months to 4 person-days

[1]However, how these data were measured and which languages were used in each company are not discussed.

- **GTE**
  - Cost: $14M[2] lower (reuse level 14%)
- **Hewlett-Packard**
  - Defects: 24% and 76% lower (two projects)
  - Productivity: 40% and 57% higher (same two projects)
  - Time-to-market: 42% lower (one of the above two projects)
- **NEC**
  - Productivity: 6.7 times higher
  - Quality: 2.8 times better
- **Raytheon**
  - Productivity: 50% higher (reuse level 60%)
- **Toshiba**
  - Defects: 20–30% lower (reuse level 60%)

## *2.2 The Obstacles*

Despite the benefits of software reuse, there are some factors that directly or indirectly influence its adoption. These factors can be managerial, organizational, economical, conceptual, or technical (Sametinger 1997).

**Managerial and Organizational Obstacles** Reuse is not just a technical problem that has to be solved by software engineers. Thus, management support and adequate organizational structures are equally important. The most common reuse obstacles are:

- **Lack of management support**. Since software reuse causes upfront costs, it cannot be widely achieved in an organization without support of top-level management. Managers have to be informed about initial costs and have to be convinced about expected savings.
- **Project management**. Managing traditional projects is not an easy task, mainly, projects related to software reuse. Making the step to large-scale software reuse has an impact on the whole software life cycle.
- **Inadequate organizational structures**. Organizational structures must consider different needs that arise when explicit, large-scale reuse is being adopted. For example, a separate team can be defined to develop, maintain, and certify software components.

[2]All the values presented are in US dollars.

- **Management incentives**. A lack of incentives prohibits managers from letting their developers spend time in making components of a system reusable. Their success is often measured only in the time needed to complete a project. Doing any work beyond that, although beneficial for the company as a whole, diminishes their success.

**Economic Obstacles** Reuse can save money in the long run, but that is not for free. Costs associated with reuse can be (Poulin 1997; Sametinger 1997): costs of making something reusable, costs of reusing it, and costs of defining and implementing a reuse process. Moreover, reuse requires upfront investments in infrastructure, methodology, training, tools (among others things), and archives, with payoffs being realized only years later. Developing assets for reuse is more expensive than developing them for single use only (Poulin 1997). Higher levels of quality, reliability, portability, maintainability, generality, and more extensive documentation are necessary, thus such increased costs are not justified when a component is used only once.

**Conceptual and Technical Obstacles** The technical obstacles for software reuse include issues related to search and retrieval of components, legacy components, and aspects involving adaptation (Sametinger 1997):

- **Difficulty of finding reusable software**. In order to reuse software components there should exist efficient ways to search and retrieve them. Moreover, it is important to have a well-organized repository containing components with some means of accessing it (Mili et al. 1998; Lucrédio et al. 2004).
- **Non-reusability of found software**. Easy access to existing software does not necessarily increase software reuse. Reusable assets should be carefully specified, designed, implemented, and documented; thus, sometimes, modifying and adapting software can be more expensive than programming the needed functionality from scratch.
- **Legacy components not suitable for reuse**. One known approach for software reuse is to use legacy software. However, simply recovering existing assets from legacy system and trying to reuse them for new developments is not sufficient for systematic reuse. Reengineering can help in extracting reusable components from legacy systems, but the efforts needed for understanding and extraction should be considered.
- **Modification**. It is very difficult to find a component that works exactly in the same way that the developer wants. In this way, modifications are necessary and there should exist ways to determine their effects on the component and its previous verification results.

### *2.3 The Basic Features*

The software reuse area has three key features (Ezran et al. 2002):

1. **Reuse is a systematic software development practice**. Systematic software reuse means:
   (a) Understanding how reuse can contribute toward the goals of the whole business
   (b) Defining a technical and managerial strategy to achieve maximum value from reuse
   (c) Integrating reuse into the whole software process, and into the software process improvement program
   (d) Ensuring all software staff have the necessary competence and motivation
   (e) Establishing appropriate organizational, technical, and budgetary support
   (f) Using appropriate measurements to control reuse performance
2. **Reuse exploits similarities in requirements and/or architecture between applications**. Opportunities for reuse from one application to another originate in their having similar requirements, or similar architectures, or both. The search for similarities should begin as close as possible to those points of origin—that is, when requirements are identified and architectural decisions are made. The possibilities for exploiting similarities should be maximized by having a development process that is designed and managed to give full visibility to the flow, from requirements and architecture to all subsequent work products.
3. **Reuse offers substantial benefits in productivity, quality, and business performance**. Systematic software reuse is a technique that is employed to address the need for improvement of software development quality and efficiency (Krueger 1992). Quality and productivity could be improved by reusing all forms of proven experience, including products and processes, as well as quality and productivity models. Productivity could be increased by using existing experience, rather than creating everything from the beginning (Basili et al. 1996). Business performance improvements include lower costs, shorter time to market, and higher customer satisfaction, which have already been noted under the headings of productivity and quality improvements. These benefits can initiate a virtuous circle of higher profitability, growth, competitiveness, increased market share, and entry to new markets.

## 3 Organized Tour: Genealogy and Seminal Papers

In the previous section, we discussed important concepts and principles related to software reuse. This section presents a deep diving in the area describing the main work in the field. It is not too simple to define a final list and some important work can be left behind. However, we believe that the work presented represents solid

contributions in the field. The reader can check also important surveys in the area such as Krueger (1992), Mili et al. (1995), Kim and Stohr (1998), and Almeida et al. (2007).

The seminal papers were divided in seven areas, which we believe cover the area properly.

## *3.1 The Roots*

In 1968, during the NATO Software Engineering Conference, generally considered the birthplace of the field, the focus was the software crisis—the problem of building large, reliable software systems in a controlled, cost-effective way. From the beginning, software reuse was considered as a way for overcoming the software crisis. An invited paper at the conference: "*Mass Produced Software Components*" by McIlroy (1968), ended up being the seminal paper on software reuse. In McIlroy's words: "*the software industry is weakly founded and one aspect of this weakness is the absence of a software component sub-industry*" (p. 80), a starting point to investigate mass-production techniques in software. In the "mass production techniques," his emphasis is on "techniques" and not in "mass production."

McIlroy argued for standard catalogs of routines, classified by precision, robustness, time–space performance, size limits, and binding time of parameters; to apply routines in the catalogs to any one of a larger class of often quite different machines; and, to have confidence in the quality of the routines.

McIlroy had a great vision to propose those ideas almost 50 years ago. His ideas were the foundations to start the work on repository systems and software reuse processes such as component-based development, domain engineering, and software product lines. ComponentSource,[3] a large industrial component market, is the closest solution to McIlroy's ideas for a component industry.

Based on a set of important questions such as: what are the different approaches to reusing software? How effective are the different approaches? What is required to implement a software reuse technology?, Krueger (1992) presented one of the first surveys in the software reuse area.

He classified the approaches in eight categories: high-level languages, design and code scavenging, source code components, software schemas, application generators, very high-level languages, transformational systems, and software architectures. Next, he used a taxonomy to describe and compare the different approaches and make generalizations about the field of software reuse. The taxonomy characterizes each reuse approach in terms of its reusable artifacts and the way these artifacts are *abstracted* (What type of software artifacts are reused and what abstractions are used to describe the artifacts?), *selected* (How are reusable artifacts

[3]ComponentSource—https://www.componentsource.com/help-support/publisher/faqs-open-market

selected for reuse?), *specialized* (How are generalized artifacts specialized for reuse?), and *integrated* (How are reusable artifacts integrated to create a complete software system?).

Many of the ideas discussed in the survey are still relevant and valid for today. We had other advances in the field as new paradigms such as aspect-oriented programming, service-oriented computing, and so on, but the conceptual framework could be reused and updated for the current days. We think that some improvements could be made with the use of evidence (Kitchenham et al. 2004) to state that some approach is more robust than another, but in no way does it diminish the relevance of the work.

Tracz's (1995) book presents an overview on software reuse covering the motivations, inhibitors, benefits, myths, and future directions in the field. The book is based on a collection of short essays and updated from various columns and papers published over the years. Another important aspect of the book is the easy language and humor introduced in each chapter.

## *3.2 Libraries and Repository Systems*

As we defined in Sect. 2, a repository is the place where reusable software assets are stored, along with the catalog of assets. Substantial work in the software reuse area was conducted to define mechanisms to store, search, and retrieve software assets.

Based on the premises that a fundamental problem in software reuse was the lack of tools to locate potential code for reuse, Frakes and Nejmeh (1986) presented an approach using the CATALOG information retrieval system to create, maintain, search, and retrieve code in the `C` language. The solution was used within AT&T.

CATALOG featured a database generator that assisted users in setting up databases, an interactive tool for creating, modifying, adding, and deleting records, and a search interface with a menu driven for novice users, and a command-driven mode for expert users. Techniques such as inverted files used in current search engines as well as automatic stemming and phonetic matching were used in the solution. CATALOG databases were built using B-Trees in order to optimize search and retrieval features.

Frakes and Nejmeh also identified that the extent to which information retrieval technology could promote software reuse was directly related to the quality and accuracy of the information in its software database. That is, poor descriptions of code and functionality could decrease the probability that the code could be located for potential reuse during the search process. Thus, they defined a software template design to promote reuse. The goal was to keep this documentation for each module and function stored in the tool to increase the ease with which it could be reused. It was an important contribution for future work in the area of component documentation.

After the initial work from Frakes and Nejmeh (1986), Prieto-Diaz and Freeman (1987) presented a work which formed the foundations for the component search

research. They proposed a facet-based scheme to classify software components. The facet scheme was based on the assumptions that collections of reusable components are very large and growing continuously, and that there are large groups of similar components.

In this approach, a limited number of characteristics (facets) that a component may have are defined. According to them, facets are sometimes considered as perspectives, viewpoints, or dimensions of a particular domain. Then, a set of possible keywords are associated to each facet. In order to describe a component, one or more keywords are chosen for each facet. Thus, it is possible to describe components according to their different characteristics. Unlike the traditional hierarchical classifications, where a single node from a tree-based scheme is chosen, facet-based classification allows multiple keywords to be associated to a single facet, reducing the chances of ambiguity and duplication.

Nowadays, facets are used in electronic commerce websites such as e-Bay.

In 2000, while the researchers were defining new methods and techniques to search and retrieve software components based on "conventional directions," Ye and Fischer (2000) came up with a great idea inverting the search scenario. According to them, software component-based reuse is difficult for developers to adopt because they must know what components exist in a reuse repository and then they must know how to retrieve them.

Their solution, called active reuse repository systems, used active information delivery mechanisms to deliver potentially reusable components that are relevant to the current development task. The idea was that they could help software developers reuse components they did not even know existed. It could also reduce the cost of component search because software developers need neither to specify reuse queries explicitly, nor to switch working contexts back and forth between development environments and repository systems. The idea of active reuse can be found in some software development tools such as Eclipse (based on additional plugins).

The reader can find additional discussion on libraries and repository systems in Mili et al. (1998), Lucrédio et al. (2004), Burégio et al. (2008), and Sim and Gallardo-Valencia (2013).

## *3.3 Generative Reuse*

Generative reuse is based on the reuse of a generation process rather than the reuse of components (compositional approach). Examples of this kind of reuse are generators for lexical analyzers, parsers, and compilers, application generators, language-based generators, and transformation systems (Sametinger 1997).

Application generators reuse complete software systems design and are appropriate in application domains where many similar systems are written, one system is modified or rewritten many times during its lifetime, or many prototypes of a system are necessary to converge on a usable product. Language-based generators

provide a specification language that represents the problem domain and simultaneously hides implementation details from the developer. Specification languages allow developers to create systems using constructs that are considered high-level relative to programming languages. Finally, with transformation systems, software is developed in two phases: in the first one, describing the semantic behavior of a software system and applying transformations to the high-level specifications (Krueger 1992).

Neighbors (1984) presented the pioneer work on generative reuse based on his PhD thesis. The Draco approach organized reusable components based on problem domains. Source-to-source program transformations, module interconnection languages, software components, and domain-specific languages worked together in the construction of similar systems from reusable parts.

In general, Draco performed three activities: (1) It accepted a definition of a problem domain as a high-level, domain-specific language called a domain language. Both the syntax and semantics of the domain language had to be described. (2) Once a domain language had been described, Draco could accept a description of a software system to be constructed as a statement or program in the domain language. (3) Finally, once a complete domain language program had been given, then Draco could refine the statement into an executable program under human guidance. It means that a developer wrote a program in a domain language and it was compiled into successively lower-level domains until it eventually ended up as executable code in a language such as C, C++, or Java.

Neighbors' work was very important because he introduced ideas such as domain, domain-specific languages, and generators. His influence can be found in commercial product lines tools such as Gears[4] and pure::variants.[5]

Batory et al. (1994) introduced the GenVoca strategy that supported the generation of software components by composing based on layers in a Layer-of-Abstraction (LOA) model of program construction. Each layer provides a cohesive set of services needed by the overall component. They defined also realms to represent types, and components to implement or realize those types. The components need not be fully concrete, and may be parameterized by other realms or types. The parameterization relationship defines a dependency between the realms, which in some cases implies a layered structure of the components and realms.

The organization of GenVoca is similar in many ways to Draco; a key difference is that GenVoca relies on larger grained refinements (layers). Thus, whereas a Draco refinement might affect a single function invocation, a GenVoca transformation will perform a coordinated refinement of all instance variables and all methods within in several related classes in a single refinement step.

A nice overview on generative reuse can be found in Biggerstaff (1998). In this paper, the author discusses 15 years of research in the field, addressing the key

[4]BigLever—http://www.biglever.com/

[5]pure-systems—https://www.pure-systems.com/

elements of generative success, main problems, evidence from the solutions, and a guide to generative reuse technologies.

## *3.4 Metrics and Economic Models*

Metrics and economic models play an important role in software engineering in general, and in software reuse in particular. They enable managers to quantify, justify, and document their decisions providing a sound basis for their decision-making process.

Favaro (1991) analyzed the economics of reuse based on a model from Barnes et al. (1998). He estimated the quantities `R` (proportion of reused code in the product) and `b` (cost relative of incorporating the reused code into the new product) for an Ada development project. According to him, it was difficult to estimate `R` because it was unclear whether to measure source code or relative size of the load modules. Regarding `b`, it was even more difficult to estimate because it was unclear whether cost should be measured as the amount of real-time necessary to install the component in the application and whether the cost of learning should be included.

Favaro identified that the cost of reusability increased with the complexity of the component. In addition, he found that some components must be used approximately 5–13 times before their costs are recovered.

Frakes and Terry (1996) surveyed metrics and models of software reuse and provided a classification structure to help users select them. They reviewed six types of metrics and models: cost–benefit models, maturity assessment models, reuse metrics, failure modes models, reusability assessment models, and reuse library metrics. The work can be considered the main survey in the area of software reuse metrics and models.

Based on his experience at IBM and Lockheed Martin, Poulin (1997) surveyed the software reuse metrics area and presented directions to implement a metric program considering the software development processes for/with reuse and the roles involved. Poulin recommends the following metrics for a reuse program:

1. **Reuse % for measuring reuse levels**

   ```
   Reuse% = RSI/Total Statements x 100%
   ```

   RSI means Reused Source Instruction. It is important to guarantee uniformity of results and equity across organizations. Deciding what to count and what not to count can sound easy, but in real-life projects, it is difficult. A manager and software engineer cannot agree about count, for example, product maintenance as reuse (code from new versions), use of COTS as reuse, code libraries as reuse, and so on. All of these cases are not counted as reuse by Poulin and we agree with him.

2. **Reuse Cost Avoidance (RCA)** for quantifying the benefits of reusing software to an organization:

```
RCA = Development Cost Avoidance + Service Cost Avoidance
```

Where Development Cost Avoidance (DCA):

```
DCA = RSI x (1 - RCR) x (New code cost)
```

RCR (Relative Cost of Reuse) has the default value = 0.2
And Service Cost Avoidance (SCA):

```
SCA = RSI x (Your error rate) x (Your error cost)
```

The previous RCA metrics considers that an organization only consumes reusable software. If the organization also produces reusable software, we can subtract the Additional Development Cost (ADC) from RCA to obtain the organization's ROI:

```
ADC = (RCWR - 1) x (Code written for reuse by others) x
                    (New code cost)
```

RCWR has the default value = 1.5

It is important to highlight that if an organization has previous data that better represent RCR and RCWR, these values should be used.

The software reuse metrics area presents solid references with metrics related to reuse processes and repository systems, which can be used in current software development projects. However, in order to have success with it, organizations should define clearly its goals and based on them define what to measure. Metrics themselves do not make sense if used alone just as numbers. Thus, thinking in terms of the Goal Question Metric (GQM) approach is essential to success.

Lim (1998), besides discussing several aspects of software reuse, presents a nice survey on software reuse metrics and economic models.

## *3.5 Reuse Models*

Reuse maturity models support an assessment of how advanced reuse programs are in implementing systematic reuse, using in general an ordinal scale of reuse phases (Frakes and Terry 1996). It is a set of stages through which an organization progresses inspired by Capability Maturity Model Integration (CMMI). Each stage has a certain set of characteristics and brings the organization to a higher level of quality and productivity.

The reuse capability model developed by the Software Productivity Consortium (SPC) was composed of two elements: an assessment model and an implementation model (Davis 1993). The first one consisted of a set of categorized critical success factors that an organization can use to assess the current state of its reuse practices. The factors are organized into four primary groups: management, application development, asset development, and process and technology factors.

The implementation model aids prioritize goals and build four successive stages of reuse implementation: opportunistic, integrated, leveraged, and anticipating.

While several researchers were creating new reuse models, another direction was explored by Frakes and Fox (1996). According to them, failure models analysis provides an approach to measure and improve a reuse process based on a model of the ways a reuse process can fail. The model could be used to evaluate the quality of a systematic reuse program, to determine reuse obstacles in an organization, and to define an improvement strategy for a systematic reuse program.

The reuse failure model has seven failure modes corresponding to steps a software developer will need to complete in order to reuse a component. The failure modes are: no attempt to reuse; part does not exist; part is not available; part is not found; part is not understood; part is not valid; and part cannot be integrated. In order to use the model, an organization gathers data on reuse failure modes and causes, and then uses this information to prioritize its reuse improvement activities.

Garcia (2010) defined the RiSE Reference Model (RiSE-RM) whose purpose was to determine which process areas, goals, and key practices should be considered by companies interested in adopting a systematic reuse approach. It includes a set of process areas, guidelines, practices, and process outcomes.

RiSE-RM has two primary goals: (1) to help in the assessment of an organization's current situation (maturity level) in terms of software reuse practices; and (2) to aid the organization in the improvement of their productivity, quality, and competitiveness though the adoption of software reuse practices.

RiSE-RM model was evolved by RiSE Labs through discussions with industry practitioners and software reuse researchers and the state of the art in the area of reuse adoption model. It has seven maturity levels that reflect a degree of the reuse process maturity. The levels are: informal reuse, basic reuse, planned reuse, managed reuse, family-oriented products reuse, measured reuse, and proactive reuse. The model was validated by software reuse experts using a survey and experimented in Brazilian companies. The results were considered interesting and the model can be adopted for software development organizations considering starting a reuse program.

Several reuse maturity models have been developed along the years and the reader can check more information in Frakes and Terry (1996) and Lim (1998).

## *3.6 Software Reuse Methods and Processes*

Software applications are complex products that are difficult to develop and test and, often, present unexpected and undesired behaviors that may even cause severe problems and damage. For these reasons, researchers and practitioners have been paying increasing attention to understanding and improving the quality of the software being developed. It is accomplished through a number of approaches, techniques, and tools, and one of the main directions investigated is centered on the study and improvement of the process through which software is developed.

According to Sommerville (2006), a software process is a set of activities that leads to the production of a software product. Processes are important and necessary to define how an organization performs its activities, and how people work and interact in order to achieve their goals.

The adoption of either a new, well-defined, managed software process or a customized one is a possible facilitator for success in reuse programs (Morisio et al. 2002). However, the choice of a specific software reuse process is not a trivial task, because there are technical (management, measurements, tools, etc.) and nontechnical (education, culture, organizational aspects, etc.) aspects that must be considered.

In 1976, Parnas (1976) introduced the ideas of program families. According to him, we can consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members. This work made extensive contributions for the software engineering area in general, such as stepwise refinement, commonality analysis, design decisions, and sure, software reuse. Parnas' ideas were essentials for the field of software product lines.

Based on motivation that the research involving software reuse processes was too general and did not present concrete techniques to perform tasks such as architecture and component modeling and implementation, three software development experts—Jacobson, Griss, and Jonsson—created the Reuse-driven Software Engineering Business (RSEB) (Jacobson et al. 1997). RSEB is a use-case-driven systematic reuse process based on the UML notation. The method was designed to facilitate both the development of reusable object-oriented software and software reuse.

Key ideas in RSEB are: the explicit focus on modeling variability and maintaining traceability links connecting representation of variability throughout the models, that is, variability in use cases can be traced to variability in the analysis, design, and implementation object models.

RSEB has separated processes for Domain Engineering and Application Engineering. Domain Engineering in RSEB consists of two processes: Application Family Engineering, concerned with the development and maintenance of the overall layered system architecture and Component System Engineering, concerned with the development of components for the different parts of the application system with a focus on building and packaging robust, extendible, and flexible components.

Despite the RSEB focus on variability, the process components of Application Family Engineering and Component System Engineering do not include essential domain analysis techniques such as domain scoping and modeling. Moreover, the process does not describe a systematic way to perform the asset development as proposed. Another shortcoming of RSEB is the lack of feature models to perform domain modeling, considered a key aspect by the reuse community (Kang et al. 1990). In RSEB, variability is expressed at the highest level in the form of variation points, which are then implemented in other models using variability mechanisms.

Kang et al. (1998) presented the thesis that there were many attempts to support software reuse, but most of these efforts had focused on two directions: exploratory research to understand issues in Domain-Specific Software Architectures (DSSA), component integration and application generation mechanisms; and theoretical research on software architecture and architecture specification languages, development of reusable patterns, and design recovery from existing code. Kang et al. considered that there were few efforts to develop systematic methods for discovering commonality and using this information to engineer software for reuse. It was their motivation to develop the Feature-Oriented Reuse Method (FORM) (Kang et al. 1998), an extension of their previous work (Kang et al. 1990).

FORM is a systematic method that focuses on capturing commonalities and differences of applications in a domain in terms of features and using the analysis results to develop domain architectures and components. In FORM, the use of features is motivated by the fact that customers and engineers often speak of product characteristics in terms of features the product has and/or delivers.

FORM method consists of two major engineering processes: Domain Engineering and Application Engineering. The domain engineering process consists of activities for analyzing systems in a domain and creating reference architectures and reusable components based on the analysis results. The application engineering process consists of activities for developing applications using the artifacts created during domain engineering.

There are three phases in the domain engineering process: context analysis, domain modeling, and architecture (and component) modeling. FORM does not discuss the context analysis phase; however, the domain modeling phase is very well explored with regard to features. The core of FORM lies in the analysis of domain features and use of these features to develop reusable domain artifacts. The domain architecture, which is used as a reference model for creating architectures for different systems, is defined in terms of a set of models, each one representing the architecture at a different level of abstraction. Nevertheless, aspects such as component identification, specification, design, implementation, and packaging are under-investigated.

After the domain engineering, the application engineering process is performed. Once again, the emphasis is on the analysis phase with the use of the developed features. However, few directions are defined to select the architectural model and develop the applications using the existing components.

Bayer et al. (1999) proposed the Product Line Software Engineering (PuLSE) methodology. The methodology was developed with the purpose of enabling the conception and deployment of software product lines within a large variety of enterprise contexts. One important feature of PuLSE is that it is the result of a bottom-up effort: the methodology captures and leverages the results (the lessons learned) from technology transfer activities with industrial customers.

PuLSE is composed of three main elements: the Deployment phases, the Technical components, and the Support components. The deployment phases are a set of stages that describe activities for initialization, infrastructure construction, infrastructure usage, and evolution and management of product lines. The technical

components provide the technical know-how needed to make the product line development operational. For this task, PuLSE has components for Customization (BC), Scoping (Eco), Modeling (CDA), Architecting (DSSA), Instantiating (I), and Evolution and Management (EM). At the end, the support components are packages of information, or guidelines, which enable a better adaptation, evolution, and deployment of the product line.

The PuLSE methodology presents an initial direction to develop software product lines. However, some points are not well discussed. For example, the component PuLSE-DSSA supports the definition of a domain-specific software architecture, which covers current and future applications of the product line. Nevertheless, aspects such as specification, design, and implementation of the architecture's components are not presented. Bayer et al. consider it an advantage, because PuLSE-DSSA does not require a specific design methodology or a specific Architecture Description Language (ADL). We do not agree with this vision because the lack of details is the biggest problem related to software reuse processes. The same problem can be seen in the Usage phase, in which product line members are specified, derived, and validated without explicit details on how this can be done.

Based on industrial experiences in software development, especially at Lucent Technologies, David Weiss and Chi Lai presented the Family-Oriented Abstraction, Specification, and Translation (FAST) process (Weiss and Lai 1999). Their goal was to provide a systematic approach to analyze potential families and to develop facilities and processes for generating family members. FAST defines a pattern for software production processes that strives to resolve the tension between rapid production and careful engineering. A primary characteristic of the pattern is that all FAST processes are organized into three subprocesses: Domain Qualification (DQ), Domain Engineering, and Application Engineering.

Domain Qualification consists of an economic analysis of the family and requires estimating the number and value of family members and the cost to produce them. Domain Engineering makes it possible to generate members of a family and is primarily an investment process; it represents a capital investment in both an environment and the processes for rapidly and easily producing family members using the environment. Application Engineering uses the environment and processes to generate family members in response to customer requirements (Weiss and Lai 1999).

The key aspects of FAST is that the process is derived from practical experiences in industrial environments, the systematic definition of inputs, outputs, steps, roles, and the utilization of a process model that describes the process. However, some activities in the process such as in Domain Engineering are not as simple to perform, for example, the specification of an Application Modeling Language (AML)—language for modeling a member of a domain—or to design the compiler to generate the family members.

van Ommering et al. (2000) created at Philips, Koala, a component model for consumer electronics. The main motivation for the development of Koala was to handle the diversity and complexity of embedded software and its increasing production speed. In Koala, a component is a unit of design composed of a

specification and an implementation. Semantically, Koala components are units of computation and control connected in an architecture.

The components are defined in an ADL consisting of an IDL for defining component interfaces, a CDL for defining components, and a Data Definition Language (DDL) for specifying local data in components. Koala component definitions are compiled by the Koala compiler to their implementation in a programming language, for example, C (Lau and Wang 2007). The Koala component model was used successfully to build a product population for consumer electronics from repositories of preexisting components.

Roshandel et al. (2004) presented Mae, an architecture evolution environment. It combines Software Configuration Management (SCM) principles and architectural concepts in a single model so that changes made to an architectural model are semantically interpreted prior to the application of SCM processes. Mae uses a type system to model architectural primitives and version control is performed over type revisions and variants of a given type.

Mae enables modeling, analysis, and management of different versions of architectural artifacts, and supports domain-specific extensions to capture additional system properties. The authors applied Mae to manage the specification and evolution of three different systems: an audio/video entertainment system, a troop deployment and battle simulation system, and a mobile robot system built in cooperation with NASA Jet Propulsion Laboratory (JPL). The experience showed that Mae is usable, scalable, and applicable to real-world problems (Roshandel et al. 2004).

More information about software reuse methods and processes can be seen in Lim (1998) and Almeida et al. (2005).

## *3.7 Software Reuse: The Past Future*

With the maturity of the area, several researchers have discussed future directions in software reuse. Kim and Stohr (1998) discussed exhaustively seven crucial aspects related to reuse: definitions, economic issues, processes, technologies, behavioral issues, organizational issues, and, finally, legal and contractual issues. Based on this analysis, they highlighted the following directions for research and development: *measurements*, *methodologies* (development for and with reuse), *tools*, and *nontechnical aspects*.

In 1999, during a panel (Zand et al. 1999) in the Symposium on Software Reusability (SSR), software reuse specialists such as Victor Basili, Ira Baxter, and Martin Griss presented several issues related to software reuse adoption in large scale. Among the considerations discussed, the following points were highlighted: (1) *education in software reuse area still is a weak point*; (2) *the necessity of the academia and industry to work together*; and, (3) *the necessity of experimental studies to validate new work*.

Frakes and Kang (2005) presented a summary on the software reuse research discussing unsolved problems based on the Eighth International Conference on Software Reuse (ICSR), in Madrid, Spain. According to them, open problems in reuse included: *reuse programs and strategies for organizations, organizational issues, measurements, methodologies, libraries, reliability, safety,* and *scalability*.

As these work were published more than 10 years ago, to conduct an analysis based on their prediction is not too hard. Among the future directions defined, we consider that many advances were achieved in the field. In this sense, we believe that the following areas need more investigation: measurement for software product lines and safety for reuse in general.

## 4 Software Product Lines (SPL): An Effective Reuse Approach

The way that goods are produced has changed significantly over time. While goods were previously handcrafted for individual customers (Pohl et al., 2005), the number of people who could afford to buy several kinds of products have increased.

In the domain of vehicles, this led to Henry Ford's invention of the mass production (product line), which enabled production for a mass market cheaper than individual product creation on a handcrafted basis. The same idea was made also by Boeing, Dell, and even McDonald's (Clements and Northrop 2001).

Customers were satisfied with standardized mass products for a while (Pohl et al. 2005); however, not all of the people want the same kind of car. Thus, industry was challenged with the rising interest for individual products, which was the beginning of mass customization.

Thereby, many companies started to introduce common platforms for their different types of products, by planning beforehand which parts will be used in different product types. Thus, the use of platforms for different products led to the reduction in the production cost for a particular product kind. The systematic combination of mass customization and common platforms is the key for product lines, which is defined as a "*set of software-intensive systems that share a common, managed feature set, satisfying a particular market segment's specific needs or mission and that are developed from a common set of core assets in a prescribed way*" (Clements and Northrop 2001).

### *4.1 Software Product Line Essential Activities*

Software product lines include three essential activities: *Core Asset Development* (CAD), *Product Development* (PD), and *Management* (Clements and Northrop

2001). Some authors (Pohl et al. 2005) use other terms for CAD and PD, for example, Domain Engineering (DE) representing the CAD and Application Engineering (AE) representing the PD. These activities are detailed as follows.

**Core Asset Development (Domain Engineering)** This activity focuses on establishing a production capability for the products (Clements and Northrop 2001). It is also known as Domain Engineering and involves the creation of common assets, generic enough to fit different environments and products in the same domain.

The core asset development activity is iterative, and according to Clements and Northrop (2001), some contextual factors can impact in the way the core assets are produced. Some contextual factors can be listed as follows: product constraints such as commonalities, variants, and behaviors; and production constraints, which is how and when the product will be brought to market. These contextual factors may drive decisions about the used variability mechanisms.

**Product Development (Application Engineering)** The main goal of this activity is to create individual (customized) products by reusing the core assets. This activity is also known as Application Engineering and depends on the outputs provided by the core asset development activity (the core assets and the production plan).

Product engineers use the core assets, in accordance with the production plan, to produce products that meet their respective requirements. Product engineers also have an obligation to give feedback on any problem within the core assets, to avoid the SPL decay (minimizing corrective maintenance), and keep the core asset base healthy and viable for the construction of the products.

**Management** This activity includes technical and organizational management. Technical management is responsible for the coordination between core asset and product development and the organizational management is responsible for the production constraints and ultimately determines the production strategy.

## *4.2 Commonalities and Variabilities in SPL*

SPL establishes a systematic software reuse strategy whose goal is to identify commonality (common functionality) and variability points among applications within a domain, and build reusable assets to benefit future development efforts (Pohl et al. 2005). Linden et al. (2007) separate the variability in three types, as follows:

- Commonality: common assets to all the products.
- Variability: common assets to some products.
- Specific products: it is required for a specific member of the family (it cannot be integrated in the set of the family assets).

In the SPL context, both commonalities and variabilities are specified through features. SPL engineers consider features as central abstractions for the product

configuration, since they are used to trace requirements of a customer to the software artifacts that provide the corresponding functionality. In this sense, the features communicate commonalities and differences of the products between stakeholders, and guide structure, reuse, and variation across all phases of the software life cycle (Apel et al. 2013).

The variability is viewed as being the capability to change or customize a system. It allows developers to delay some design decisions, that is, the variation points. A variation point is a representation of a variability subject, for example, the type of lighting control that an application provides. A variant identifies a single option of a variation point. Using the same example, two options of lighting control can be chosen for the application (e.g., user or autonomic lighting) (Pohl et al. 2005).

Although the variation points identified in the context of SPL hardly change over time, the set of variants defined as objects of the variability can be changed. This is the Software Product Lines Engineering (SPLE) focus, that is, the simultaneous use of variable artifacts in different forms (variants) by different products (Gurp et al. 2001).

The variability management is an activity responsible to define, represent, explore, implement, and evolve SPL variability (Linden et al. 2007) by dealing with the following questions:

- Identifying what varies, that is, the variable property or variable feature that is the subject of the variability
- Identifying why it varies, based on needs of the stakeholders, user, application, and so on
- Identifying how the possible variants vary, which are objects of the variability (instance of a product)

Considering the previous example from the car domain (Fig. 1), once created the feature model with the commonality and variability of the domain, the design, and implementation can be performed, for example, using techniques such as conditional compilation, inheritance, or aspect-oriented programming. Based on conditional compilation, `ifdefs` sentences can be used to separate optional code. Thus, in the product development activity, automation tools (such as Ant[6]) can be used to generate different products based on the features selection.

## *4.3 Future Directions in SPL and Software Reuse*

**Dynamic Software Product Lines (DSPL)** Emerging domains, such as mobile, ubiquitous computing, and software-intensive embedded systems, demand high degree of adaptability from software. The capacity of these software to reconfigure and incorporate new functionality can provide significant competitive advantages.

[6] http://ant.apache.org/

This new trend market requires SPL to become more evolvable and adaptable (Bosch and Capilla 2012). More recently, DSPL became part of these emerging domains.

The DSPL approach has emerged within the SPLE field as a promising means to develop SPL that incorporates reusable and dynamically reconfigurable artifacts (Hallsteinsen et al. 2008). Thus, researchers introduced the DSPL approach enabling to bind variation points at runtime. The binding of the variation points happens initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment (Hallsteinsen et al. 2008).

According to Hinchey et al. (2012), the DSPL practices are based on: (1) explicit representation of the configuration space and constraints that describe permissible configurations at runtime on the level of intended capabilities of the system; (2) the system reconfiguration that must happen autonomously, once the intended configuration is known; and (3) at the traditional SPLE practices.

The development of a DSPL involves two essential activities: monitoring the current situation for detecting events that might require adaptation and controlling the adaptation through the management of variation points. In that case, it is important to analyze the change impact on the product's requirements or constraints and planning for deriving a suitable adaptation to cope with new situations. In addition, these activities encompass some properties, such as automatic decision-making, autonomy, and adaptivity, and context awareness (Hallsteinsen et al. 2008; Bencomo et al. 2012).

The runtime variability can help to facilitate automatic decision-making in systems where human intervention is extremely difficult or impossible. For this reason, the DSPL approach treats automatic decision-making as an optional characteristic. The decision to change or customize a feature is sometimes left to the user (Bosch and Capilla 2012). However, the context awareness and the autonomy and adaptability are treated in the same way.

The adoption of a DSPL approach is strongly based on adapting to variations in individual needs and situations rather than market forces and supporting configuration and extension capabilities at runtime (Hallsteinsen et al. 2008; Hinchey et al. 2012). Given these characteristics, DSPL would benefit from research in several related areas. For example, it can provide the modeling framework to understand a self-adaptive system based on Service-Oriented Architecture (SOA) by highlighting the relationships among its parts, as well as, in the automotive industry, where the need for post-deployment, dynamic and extensible variability increases significantly (Bosch and Capilla 2012; Baresi et al. 2012; Lee et al. 2012).

Bencomo et al. (2012), Capilla et al. (2014) present important issues related to the state-of-the-art in DSPL.

Other promising directions for software product lines are related to **Search-based Software Product Lines** and **Multiple Product Lines (MPLs)**. Search-Based Software Engineering (SBES) is a discipline that focuses on the use of search-based optimization techniques, such as evolutionary computation (genetic algorithms), basic local searches, and integer programming to solve software

engineering problems. These techniques are being investigated and used in different SPL areas such as testing and product configuration. Harman et al. (2014) and Herrejon et al. (2015) present important issues related to the state-of-the-art in SBES in the context of product lines.

As we discussed along this chapter, the typical scope of existing SPL methods is "building a house," that is, deriving products from a single product line that is considered autonomous and independent from other possibly related systems. On the other hand, as in some situations, this perspective of "building a house" in SPL is no longer sufficient in many environments; an increasing number approaches and tools have been proposed in recent years for managing Multi-Product Lines (MPLs) representing a set of related and interdependent product lines (Holl et al. 2012).

Holl et al. (2012) define a multi-product line (MPL) as a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system. The main point is that the different product lines in an MPL can exist independently but typically use shared resources to meet the overall system requirements. They are based on several heterogeneous subsystems that are managed in a decentralized way. These subsystems themselves can represent product lines managed by a dedicated team or even organizational unit. Holl et al. (2012) present the main directions in the area of MPL based on a systematic literature review and expert survey.

## 5 Conclusion

The reuse of products, processes, and other knowledge can be important ingredients to try to enable the software industry to achieve the pursued improvements in productivity and quality required to satisfy growing demands. However, these efforts are often related to individuals and small groups, who practice it in an ad hoc way, with high risks that can compromise future initiatives in this direction. Currently, organizations are changing this vision, and software reuse starts to appear in their business agendas as a systematic and managerial process, focused on application domains, based on repeatable and controlled processes, and concerned with large-scale reuse, from analysis and design to code and documentation.

Systematic software reuse is a paradigm shift in software development from building single systems to application families of similar systems. In this chapter, we presented and discussed the fundamental ideas of software reuse from its origins to the most effective approaches. Moreover, we presented some important directions in the area of Software Product Lines, the most effective approach to achieve large-scale reuse with applications in the same domain.

## References

Almeida, E.S., Alvaro, A., Lucrédio, D., Garcia, V.C., Meira, S.R.L.: A survey on software reuse processes, International Conference on Information Reuse and Integration (IRI) (2005)

Almeida, E.S., Alvaro, A., Garcia, V.C., Mascena, J.C.C.P., Burégio, V.A.A., Nascimento, L.M., Lucrédio, D., Meira, S.R.L.: C.R.U.I.S.E. – Component Reuse in Software Engineering (2007)

Apel, S., Batory, D., Kastner, C., Saake, G.: Feature-Oriented Software Product Lines; Concepts and Implementation. Springer, Berlin (2013)

Baresi, L., Guinea, S., Pasquale, L.: Service-oriented dynamic software product lines. IEEE Comput. **45**(10), 42–48 (2012)

Barnes, B., et al.: A framework and economic foundation for software reuse. In: Tracz, W. (ed.) IEEE Tutorial: Software Reuse—Emerging Technology. IEEE Computer Society Press, Washington, DC (1998)

Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. Commun. ACM. **39**(10), 104–116 (1996)

Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B.J., Sirkin, M.: The GenVoca model of software-system generators. IEEE Softw. **11**(5), 89–94 (1994)

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: PuLSE: A Methodology to Develop Software Product Lines, Symposium Software Reusability (SSR) (1999)

Bencomo, N., Hallsteinsen, S.O., Almeida, E.S.: A view of the dynamic software product line landscape. IEEE Comput. **45**(10), 36–41 (2012)

Biggerstaff, T.J.: A perspective of generative reuse. Ann. Softw. Eng. **5**, 169–226 (1998)

Bosch, J., Capilla, R.: Dynamic variability in software-intensive embedded system families. IEEE Comput. **45**(10), 28–35 (2012)

Burégio, V.A.A., Almeida, E.S., Lucrédio, D., Meira, S.R.L.: A Reuse Repository System: From Specification to Deployment, International Conference on Software Reuse (ICSR) (2008)

Capilla, R., Bosch, J., Trinidad, P., Ruiz Cortés, A., Hinchey, M.: An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Softw. **91**, 3–23 (2014)

Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, p. 608. Addison-Wesley (2001)

Cox, B.J.: Planning the software industrial revolution. IEEE Softw. **7**(06), 25–33 (1990)

Czarnecki, K., Eisenecker, U.: Generative Progamming: Methods, Tools, Applications. Addison-Wesley, Boston (2000)

Davis, T.: The reuse capability model: A basis for improving an organization's reuse capability. International Workshop on Software Reusability (1993)

Diaz, R.P., Freeman, P.: Classifying software for reusability. IEEE Softw. **4**(1), (1987)

Ezran, M., Morisio, M., Tully, C.: Practical Software Reuse, p. 374. Springer, Heidelberg (2002)

Favaro, J.: What Price Reusability? A Case Study, First International Symposium on Environments and Tools for Ada, California, pp. 115–124, March 1991

Frakes, W.B., Fox, C.J.: Quality improvement using a software reuse failure modes model. IEEE Trans. Softw. Reuse. **23**(4), 274–279 (1996)

Frakes, W.B., Isoda, S.: Success factors of systematic reuse. IEEE Softw. **11**(5), 14–19 (1994)

Frakes, W.B., Kang, K.C.: Software reuse research: Status and future. IEEE Trans. Softw. Eng. **31**(7), 529–536 (2005)

Frakes, W.B., Nejmeh, B.A.: Software reuse through information retrieval. ACM SIGIR Forum. **21**(1–2), 30–36 (1986)

Frakes, W.B., Succi, G.: An industrial study of reuse, quality, and productivity. J. Syst. Softw. **57**(2), 99–106 (2001)

Frakes, W.B., Terry, C.: Software Reuse: Metrics and Models, ACM Computing Survey (1996)

Garcia, V.C.: RiSE Reference Model for Software Reuse Adoption in Brazilian Companies, Ph.D. Thesis, Federal University of Pernambuco, Brazil (2010)

Gurp, J.V., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines, Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 45–54, Amsterdam, Netherlands, August, 2001

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. IEEE Comput. **41**(04), 93–95 (2008)

Harman, M., Jia, Y., Krinke, J., Langdon, W.B., Petke, J., Zhang, Y.: Search based software engineering for software product line engineering: A survey and directions for future work. 18th International Software Product Line Conference (SPLC), pp. 5–18, Italy, August, 2014

Herrejon, R.E.L., Linsbauer, L., Egyed, A.: A systematic mapping study of search-based software engineering for software product lines. Inf. Softw. Technol. J. **61**, 33–51 (2015)

Hinchey, M., Park, S., Schmid, K.: Building dynamic software product lines. IEEE Comput. **45**(10), 22–26 (2012)

Holl, G., Grunbacher, P., Rabiser, R.: A systematic review and an expert survey on capabilities supporting multi product lines. Inf. Softw. Technol. J. **54**, 828–852 (2012)

Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley (1997)

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute (SEI), Technical Report, p. 161, November 1990

Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann. Softw. Eng. **5**, 143–168 (1998)

Kim, Y., Stohr, E.A.: Software reuse: Survey and research directions. J. Manag. Inf. Syst. **14**(04), 113–147 (1998)

Kitchenham, B.A., Dybå, T., Jørgensen, M.: Evidence-Based Software Engineering, International Conference on Software Engineering (ICSE) (2004)

Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992)

Lau, K.K., Wang, Z.: Software component models. IEEE Trans. Softw. Eng. **33**(10), 709–724 (2007)

Lee, J., Kotonya, G., Robinson, D.: Engineering service-based dynamic software product lines. IEEE Comput. **45**(10), 49–55 (2012)

Lim, W.C.: Effects of reuse on quality, productivity, and economics. IEEE Softw. **11**(05), 23–30 (1994)

Lim, W.C.: Managing Software Reuse. Prentice Hall, Upper Saddle River, NJ (1998)

Linden, F.V., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer (2007)

Lucrédio, D., Almeida, E.S., Prado, A.F.: A Survey on Software Components Search and Retrieval, 30th IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), Component-Based Software Engineering Track, pp. 152–159, Rennes, France, August/September 2004

McIlroy, M.D.: Mass Produced Software Components, NATO Software Engineering Conference Report, pp. 79–85, Garmisch, Germany, October 1968

Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. IEEE Trans. Softw. Eng. **21**(6), 528–562 (1995)

Mili, A., Mili, R., Mittermeir, R.: A survey of software reuse libraries. Ann. Softw. Eng. **05**, 349–414 (1998)

Mili, H., Mili, A., Yacoub, S., Addy, E.: Reuse Based Software Engineering: Techniques, Organizations, and Measurement. Wiley, New York (2002)

Morisio, M., Ezran, M., Tully, C.: Success and failure factors in software reuse. IEEE Trans. Softw. Eng. **28**(4), 340–357 (2002)

Neighbors, J.M.: The draco approach to constructing software from reusable components. IEEE Trans. Softw. Eng. **10**(5), 564–574 (1984)

Parnas, D.L.: On the design and development of program families. IEEE Trans. Softw. Eng. **2**(1), 1–8 (1976)

Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques, p. 467. Springer, New York (2005)
Poulin, J.S.: Measuring Software Reuse, p. 195. Addison-Wesley, Boston, MA (1997)
Poulin, J.S.: The Business Case for Software Reuse: Reuse Metrics, Economic Models, Organizational Issues, and Case Studies, Tutorial Notes, Torino, Italy, June, 2006
Roshandel, R., van der Hoek, A., Rakic, M.M., Medvidovic, N.: Mae – A system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13**(2), 240–276 (2004)
Sametinger, J.: Software Engineering with Reusable Components, p. 275. Springer, Berlin (1997)
Sim, S.E., Gallardo-Valencia, R.G.: Finding Source Code on the Web for Remix and Reuse. Springer, New York (2013)
Sommerville, I.: Software Engineering, Addison-Wesley (2006)
Tracz, W.: Confessions of a Used Program Salesman: Institutionalizing Software Reuse, Addison Wesley, Reading, MA (1995)
van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Comput. **33**(3), 78–85 (2000)
Weiss, D., Lai, C.T.R.: Software Product-Line Engineering. Addison Wesley, Reading, MA (1999)
Ye, Y., Fischer, G.: Promoting Reuse with Active Reuse Repository Systems, International Conference on Software Reuse (ICSR) (2000)
Zand, M., Basili, V.R., Baxter, I., Griss, M.L., Karlsson, E., Perry, D.: Reuse R&D: Gap Between Theory and Practice, Symposium on Software Reusability (SSR), pp. 172–177, Los Angeles, May, 1999