6 Impact Analysis

Per Jönsson and Mikael Lindvall

Abstract: Software changes are necessary and inevitable in software development, but may lead to software deterioration if not properly controlled. Impact analysis is the activity of identifying what needs to be modified in order to make a change, or to determine the consequences on the system if the change is implemented. Most research on impact analysis is presented and discussed in literature related to software maintenance. In this chapter, we take a different approach and discuss impact analysis from a requirements engineering perspective. We relate software change to impact analysis, outline the history of impact analysis and present common strategies for performing impact analysis. We also mention the application of impact analysis to non-functional requirements and discuss tool support for impact analysis. Finally, we outline what we see as the future of this essential change management tool.

Keywords: Impact analysis, Software change, Traceability analysis, Propagation of change, Non-functional requirements, Metrics.

6.1 Introduction

It is widely recognized that change is an inescapable property of any software, for a number of reasons. However, software changes can, and will, if not properly controlled, lead to software deterioration. For example, when Mozilla's 2,000,000 Source Lines of Code (SLOC) were analyzed, there were strong indications that the software had deteriorated significantly due to uncontrolled change, making the software very hard to maintain [17].

Software deterioration occurs in many cases because changes to software seldom have the small impact they are believed to have [40]. In 1983, some of the world's most expensive programming errors each involved the change of a single digit in a previously correct program [38], indicating that a seemingly trivial change may have immense impact. A study in the late 90s showed that software practitioners conducting impact analysis and estimating change in an industrial project underestimated the amount of change by a factor of three [26]. In addition, as software systems grow increasingly complex, the problems associated with software change increase accordingly. For example, when the source code across several versions of a 100,000,000 SLOC, fifteen-year-old telecom software system was analyzed, it was noticed that the system had decayed due to frequent change. The programmers estimating the change effort drew the conclusion that the code was harder to change than it should be [13].

Impact analysis is a tool for controlling change, and thus for avoiding deterioration. Bohner and Arnold define impact analysis as "the activity of identifying the potential consequences, including side effects and ripple effects, of a change, or estimating what needs to be modified to accomplish a change before it has been made" [3]. Consequently, the output from impact analysis can be used as a basis for estimating the cost associated with a change. The cost of the change can be used to decide whether or not to implement it depending on its cost/benefit ratio.

Impact analysis is an important part of requirements engineering since changes to software often are initiated by changes to the requirements. In requirements engineering textbooks, impact analysis is recognized as an essential activity in change management, but details about how to perform it often left out, or limited to reasoning about the impact of the change on the requirements specification [20, 23, 27, 32, 35]. An exception is [40], where Wiegers provides checklists to be used by a knowledgeable developer to assess the impact of a change proposal.

Despite its natural place in requirements engineering, research about impact analysis is more commonly found in literature related to software maintenance. In this chapter, we present impact analysis from a requirements engineering perspective. In our experience, impact analysis is an integral part of every phase in software development. During requirements development, design and code do not yet exist, so new and changing requirements affect only the existing requirements. During design, code does not yet exist, so new and changing requirements affect only existing requirements and design. Finally, during implementation, new and changing requirements affect existing requirements as well as design and code. This is captured in Fig. 6.1. Note that in less idealistic development processes, the situation still holds; requirements changes affect all existing system representations.

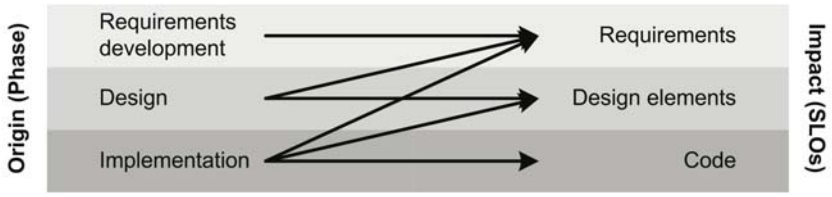


Fig. 6.1 Software life-cycle objects (SLOs) affected (right) due to requirements changes in different phases (left)

The chapter is organized as follows. In the remainder of this section, we define concepts, discuss software change and outline the history of impact analysis. In Sect. 6.2, we present common strategies for impact analysis. Sect. 6.3 discusses impact analysis in the context of non-functional requirements. We explore a number of metrics for impact analysis and give an example of an application of such metrics in Sect. 6.4. In Sect. 6.5, we look at tool support for impact analysis and discuss impact analysis in requirements management tools. Finally we outline the future of impact analysis in Sect. 6.6 and provide a summary of the chapter in Sect. 6.7.

6.1.1 Concepts and Terms

Throughout this chapter, we use several terms and concepts that are relevant in the field of impact analysis. In this section, we briefly visit these terms and concepts, and explain how each relates to impact analysis and to other terms and concepts.

Software life-cycle objects (SLOs –also called software products, or working products) are central to impact analysis. An SLO is an artifact produced during a project, such as a requirement, an architectural component, a class and so on. SLOs are connected to each other through a web of relationships. Relationships can be both between SLOs of the same type, and between SLOs of different types. For example, two requirements can be interconnected to signify that they are related to each other. A requirement can also be connected to an architectural component, for example, to signify that the component implements the requirement.

Impact analysis is often carried out by analyzing the relationships between various entities in the system. We distinguish between two types of analysis: dependency analysis and traceability analysis [3]. In dependency analysis, detailed relationships among program entities, for example variables or functions, are extracted from source code. Traceability analysis, on the other hand, is the analysis of relationships that have been identified during development among all types of SLOs. Traceability analysis is thus suitable for analyzing relationships among requirements, architectural components, documentation and so on. Requirements traceability is defined and discussed in Chap. 5. It is evident that traceability analysis has a broader application within requirements engineering than dependency analysis; it can be used in earlier development phases and can identify more diverse impact in terms of different SLO types.

It is common to deal with sets of impact in impact analysis. The following sets have been defined by Arnold and Bohner [3]:

- The System Set represents the set of all SLOs in the system – all the other sets are subsets of this set.
- The Starting Impact Set (SIS) represents the set of objects that are initially thought to be changed. The SIS typically serves as input to impact analysis approaches that are used for finding the Estimated Impact Set.
- The Estimated Impact Set (EIS) always includes the SIS and can therefore be seen as an expansion of the SIS. The expansion results from the application of change propagation rules to the internal object model repeatedly until all objects that may be affected are discovered. Ideally, the SIS and EIS should be the same, meaning that the impact is restricted to what was initially thought to be changed.
- The Actual Impact Set (AIS), finally, contains those SLOs that have been affected once the change has been implemented. In the best-case scenario, the AIS and EIS are the same, meaning that the impact estimation was perfect.

In addition to the impact sets, two forms of information are necessary in order to determine the impact of a change: information about the dependencies between objects, and knowledge about how changes propagate from object to object via dependencies and traceability links. Dependencies between objects are often captured in terms of references between them (see Chap. 5). Knowledge about how change propagates from one object to another is often expressed in terms of rules or algorithms.

It is common to distinguish between primary and secondary change. Primary change, also referred to as direct impact, corresponds to the SLOs that are identified by analyzing how the effects of a proposed change affect the system. This analysis is typically difficult to automate because it is mainly based on human expertise. Consequently, little can be found in the literature about how to identify primary changes. It is more common to find discussions on how primary changes cause secondary changes, also referred to as indirect impact.

The indirect impact can take two forms: Side effects are unintended behaviors resulting from the modifications needed to implement the change. Side effects affect both the stability and function of the system and must be avoided. Ripple effects, on the other hand, are effects on some parts of the system caused by making changes to other parts. Ripple effects cannot be avoided, since they are the consequence of the system's structure and implementation. They must, however, be identified and accounted for when the change is implemented.

We have previously mentioned architectural components as an example of SLOs. The software architecture of a system is its basic structure, consisting of interconnected components. There are many definitions of software architecture, but a recent one is "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [2]. Several other definitions exist as well (see [34]), but most echo the one given here. Software architecture is typically designed early in the project, hiding low-level design and implementation details, and then iteratively refined as the knowledge about the system grows [10]. This makes architecture models interesting from a requirements engineering and impact analysis point-of-view, because they can be used for early, albeit initially coarse, impact analysis of changing requirements.

6.1.2 Software Change and Impact Analysis

Software change occurs for several reasons, for example, in order to fix faults, to add new features or to restructure the software to accommodate future changes [28]. Changing requirements is one of the most significant motivations for software change. Requirements change from the point in time when they are elicited until the system has been rendered obsolete. Changes to requirements reflect how the system must change in order to stay useful for its users and remain competitive on the market. At the same time, such changes pose a great risk as they may cause software deterioration. Thus, changes to requirements must be captured, managed and controlled carefully to ensure the survival of the system from a technical point of view. Factors that can inflict changes to requirements during both initial development as well as in software evolution are, according to Leffingwell and Widrig [23]:

- The problem that the system is supposed to solve changes, for example for economic, political or technological reasons.
- The users change their minds about what they want the system to do, as they understand their needs better. This can happen because the users initially were uncertain about what they wanted, or because new users enter the picture.
- The environment in which the system resides changes. For example, increases in speed and capacity of computers can affect the expectations of the system.
- The new system is developed and released leading users to discover new requirements.

The last factor is both real and common. When the new system is released, users realize that they want additional features, that they need data presented in other ways, that there are emerging needs to integrate the system with other systems, and so on. Thus, new requirements are generated by the use of the system itself. According to the "laws of software evolution" [24], a system must be continually adapted, or it will be progressively less satisfactory in its environment.

Problems arise if requirements and changes to requirements are not managed properly by the development organization [23]. For example, failure to ask the right questions to the right people at the right time during requirements development will most likely lead to a great number of requirements changes during subsequent phases. Furthermore, failure to create a practical change management process may mean that changes cannot be timely handled, or that changes are implemented without proper control.

Maciaszek points out: "Change is not a kick in the teeth, unmanaged change is" [27]. In other words, an organization that develops software requires a proper change management process in order to mitigate the risks of constantly changing requirements and their impact on the system. Leffingwell and Widrig discuss five necessary parts of a process for managing change [23]. These parts, depicted in Fig. 6.2, form a framework for a change management process allowing the project team to manage changes in a controlled way.

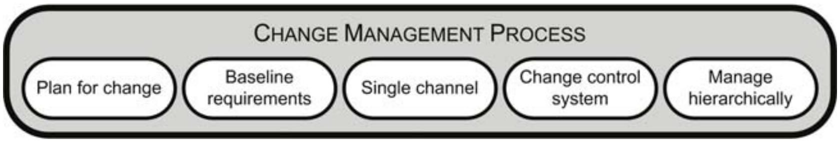


Fig. 6.2 Change management process framework [23]

Plan for change involves recognizing the fact that changes occur, and that they are a necessary part of the system's development. This preparation is essential for changes to be received and handled effectively.

Baseline requirements means to create a snapshot of the current set of requirements. The point of this step is to allow subsequent changes in the requirements to be compared with a stable, known set of requirements.

A *single channel* is necessary to ensure that no change is implemented in the system before it has been scrutinized by a person, or several persons, who keep the system, the project and the budget in mind. In larger organizations, the single channel is often a change control board (CCB).

A *change control system* allows the CCB (or equivalent) to gather, track and assess the impact of changes. According to Leffingwell and Widrig, a change must be assessed in terms of impact on cost and functionality, impact on external stakeholders (for example, customers) and potential to destabilize the system. If the latter is overlooked, the system (as pointed out earlier) is likely to deteriorate.

To *manage hierarchically* defeats a perhaps too common line of action: a change is introduced in the code by an ambitious programmer, who forgets, or overlooks, the potential effect the change has on test cases, design, architecture, requirements and so on. Changes should be introduced top-down, starting with the requirements. If the requirements are decomposed and linked to other SLOs, it is possible to propagate the change in a controlled way.

This framework for the change process leaves open the determination of an actual change process. Requirements engineering textbooks propose change management processes with varying levels of detail and explicitness [27, 32, 35]. The process proposed by Kotonya and Sommerville is, however, detailed and consists of the following steps [20]:

1. Problem analysis and change specification
2. Change analysis and costing, which in turn consists of:
    1. Check change request validity
    2. Find directly affected requirements
    3. Find dependent requirements
    4. Propose requirements changes
    5. Assess costs of change
    6. Assess cost acceptability
3. Change implementation

Impact analysis is performed in steps 2b, 2c and 2e, by identifying requirements and system components affected by the proposed change. The analysis should be expressed in terms of required effort, time, money and available resources. Kotonya and Sommerville suggest the use of traceability tables to identify and manage dependencies among requirements, and between requirements and design elements. We discuss traceability as a strategy for performing impact analysis in Sect. 6.2.1.1.

6.1.3 History and Trends

In some sense, impact analysis has been performed for a very long time, albeit not necessarily using that term and not necessarily resolving the problem of accurately determining the effect of a proposed change. The need for software practitioners to determine what to change in order to implement requirement changes has always been present. Strategies for performing impact analysis were introduced and discussed early in the literature. For example, Haney's paper from 1972 on a technique for module connection analysis is often referred to as the first paper on impact analysis [18]. The technique builds on the idea that every module pair of a system has a probability that a change in one module in the pair necessitates a change in the other module. The technique can be used to model change propagation between any system components including requirements. Program slicing, which is a technique for focusing on a particular problem by retrieving executable slices containing only the code that a specific variable depends on, was introduced already in 1979 by Weiser [39]. Slicing, which is explained in Sect. 6.2.1.2, can be used to determine dependencies in code and can be used to minimize side effects. Slicing can also be used to determine dependencies between sections in documents, including requirements, which is described below. Requirements traceability was defined in ANSI/IEEE Standard 830-1984 in 1984 [1]. Traceability describes how SLOs are related to each other and can be used to determine how change in one type of artifact causes change in another type of artifact. The notion of ripple effect was introduced by Yau and Collofello in 1980 [41]. Their models can be used to determine how change in one area of the source code propagates and causes change in other areas.

Impact analysis relies on techniques and strategies that date back a long time. It is however possible to identify a trend in impact analysis research over the years. Early impact analysis work focused on source code analysis, including program slicing and ripple effects for code. The maturation of software engineering among software organizations has led to a need to understand how changes affect other SLOs than source code.

For example, Turver and Munro [37] point out that source code is not the only product that has to be changed in order to develop a new release of the software product. In a document-driven development approach, many documents are also affected by new and changed requirements. The user manual is an example of a document that has to be updated when new user functionalities have been provided. Turver and Munro focus on the problem of ripple effects in documentation using a thematic slicing technique. They note that this kind of analysis has not been widely discussed before. The same approach can be applied to the requirements document itself in order to determine how a new or changed requirement impacts the requirements specification.

In 1996, Arnold and Bohner published a collection of research articles called Software Change Impact Analysis [3]. The purpose of the collection was to present the current, somewhat scattered, material that was available on impact analysis at the time. Reading the collection today, nearly ten years later, it becomes apparent that it still is very relevant. Papers published after 1996 seem to work with the same ideas and techniques. We do not mean to depreciate the work that has been done, but it indicates that the field is not in a state of flux. Rather, the focus remains on adapting existing techniques and strategies to new concepts and in new contexts. Impact analysis on the architectural level is an example of this.

When the year 2000 approached, the Y2K problem made it obvious that extensive impact analysis efforts were needed in order to identify software and parts of software that had to be changed to survive the century shift. This served as a revelation for many organizations, in which the software process previously had not included explicit impact analysis [4].

Today, software systems are much more complex than they were 25 years ago, and it has become very difficult to grasp the combined implications of the requirements and their relationships to architecture, design, and source code. Thus, a need for impact analysis strategies that employ requirements and their relationships to other SLOs has developed. Still, dependency webs for large software systems can be so complex that it is necessary to visualize them in novel ways. Bohner and Gracanin present research that combines impact analysis and 3D visualization in order to display dependency information in a richer format than is possible with 2D visualization [5]. Bohner also stresses the need to extend impact analysis to middleware, COTS software and web applications. The use of these types of software is becoming more common, moving the complexity away from internal data and control dependencies to interoperability dependencies. Current impact analysis strategies are not very well suited for this type of dependencies [4].

6.2 Strategies for Impact Analysis

There are various strategies for performing impact analysis, some of which are more germane to the requirements engineering process than others. Common strategies are:

- Analyzing traceability or dependency information
- Utilizing slicing techniques
- Consulting design specifications and other documentation
- Interviewing knowledgeable developers

We divide these impact analysis strategies into two categories: *automatable* and *manual*. With automatable strategies, we mean those that are in some sense algorithmic in their nature. These have the ability to provide very fine-grained impact estimation in an automated fashion, but require on the other hand the presence of a detailed infrastructure and result at times in too many false positives [30]. With manual strategies, we mean those that are best performed by human beings (as opposed to tools). These require less infrastructure, but may be coarser in their impact estimation than the automatable ones. We recognize that the two categories are not entirely orthogonal, but they do make an important distinction; the manual strategies are potentially easier to adopt and work with because they require less structured input and no new forms of SLOs need to be developed.

A previous study indicated that developers' impact analyses often result in optimistic predictions [26], meaning that the predicted set of changes represents the least possible amount of work. Thus, the work cannot be easier, only more difficult. The study also identified the need for conservative predictions and establishing a "worst level" prediction. The real amount of work will lie between the optimistic and the conservative level. An improvement goal would be to decrease variation as the impact analysis process stabilizes and becomes more mature.

The cost associated with producing a conservative prediction depends on its expected accuracy. Since conservative predictions identify such a large part of the system, developers often cannot believe they are realistic. The benefit of having a conservative prediction is the ability to determine a most probable prediction somewhere between the optimistic and the conservative prediction. An ideal impact analysis approach would always provide an optimistic and a conservative estimate. By collecting and analyzing empirical data from the predictions as well as the actual changes, it can be established where in that span the correct answer lies.

6.2.1 Automatable Strategies

Automatable impact analysis strategies often employ algorithmic methods in order to identify change propagation and indirect impact. For example, relationship graphs for requirements and other SLOs can be used with graph algorithms to identify the impact a proposed change would have on the system. The prerequisite for automatable strategies is a structured specification of the system. By structured, we mean that the specification is consistent and complete, and includes some semantic information (for example, type of relationship). Once in place, such a specification can be used by tools in order to perform automatic impact analysis. Requirements dependency webs and object models are examples of structured specifications.

The strategies presented here, traceability and dependency analysis and slicing, are typically used to assess the Estimated Impact Set by identifying secondary changes made necessary because of primary changes to the system. They are not well suited for identifying direct impact.

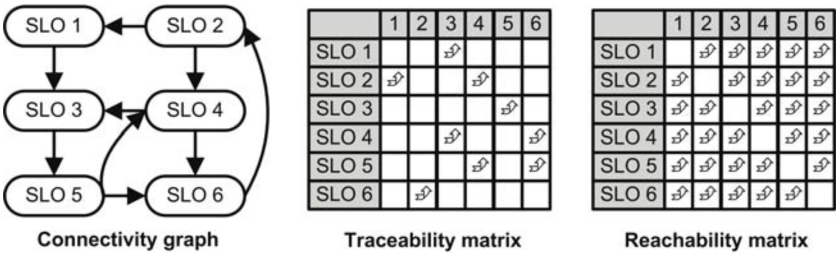
6.2.1.1 Traceability/Dependency Analysis

Traceability analysis and dependency analysis both involve examining relationships among entities in the software. They differ in scope and detail level; traceability analysis is the analysis of relationships among all types of SLOs, while dependency analysis is the analysis of low-level dependencies extracted from source code [3]. Requirements traceability is discussed further in Chap. 5.

By extracting dependencies from source code, it is possible to obtain call graphs, control structures, data graphs and so on. Since source code is the most exact representation of the system, any analysis based on it can very precisely predict the impact of a change. Dependency analysis is also the most mature strategy for impact analysis available [3]. The drawback of using source code is that it is not available until late in the project, which makes dependency analysis narrow in its field of application. When requirements traceability exists down to the source, it can, however, be very efficient to use source code dependencies in order to determine the impact of requirements changes. A drawback is that very large systems have massive amounts of source code dependencies, which make the dependency web difficult to both use and to get an overview of [5].

Traceability analysis also requires the presence of relationship links between the SLOs that are analyzed. Typically, these relationships are captured and specified progressively during development (known as pre-recorded traceability). The success of traceability analysis depends heavily on the completeness and consistency of the identified relationships. However, if traceability information is properly recorded from the beginning of development, the analysis can be very powerful.

A common approach for recording traceability links is to use a *traceability matrix* (see, for example, [20], [23] and [40]). A traceability matrix is a matrix where each row, and each column, corresponds to one particular SLO, for example a requirement. The relationship between two SLOs is expressed by putting a mark where the row of the first SLO and the column of the second SLO intersect. It is also possible to add semantic information to the relationship between SLOs. For example, the relationship between a requirement and an architectural component can be expanded to include information about whether the component implements the requirement entirely, or only partially.



	1	2	3	4	5	6
SLO 1			✓			
SLO 2	✓			✓		
SLO 3					✓	
SLO 4			✓			✓
SLO 5				✓		✓
SLO 6		✓				

	1	2	3	4	5	6
SLO 1		✓	✓	✓	✓	✓
SLO 2	✓		✓	✓	✓	✓
SLO 3	✓	✓		✓	✓	✓
SLO 4	✓	✓	✓		✓	✓
SLO 5	✓	✓	✓	✓		✓
SLO 6	✓	✓	✓	✓	✓	

Fig. 6.3 Three views of the relationships among SLOs

Ramesh and Jarke report that current requirement practices do not fully embrace the use of semantic information to increase the usefulness of relationships between SLOs [31]. A relationship stating that two SLOs affect each other but not how, will be open to interpretation by all stakeholders. According to Ramesh and Jarke, different stakeholders interpret relationships without semantic information in different ways. For example, a user may read a relationship as “implemented-by,” while a developer may read the same relationship as “puts-constraints-on.”

To further illustrate the need for semantics in traceability links, we have created an example with six interconnected SLOs. Figure 6.3 shows the SLOs in a connectivity graph (left), where an arrow means that the source SLO affects the destination SLO. For example, SLO 2 affects, or has an impact on, SLO 1 and SLO 4.

The connectivity graph corresponds exactly to a traceability matrix, shown next in the figure. An arrow in the traceability matrix indicates that the row SLO affects the column SLO. Both the connectivity graph and the traceability matrix show direct impact, or primary change needed, whereas indirect impact, or secondary change needed, can only be deduced by traversing the traceability links. For systems with many SLOs, the amount of indirect impact quickly becomes immense and hard to deduce from a connectivity graph or a traceability matrix. In order to better visualize indirect impact, the traceability matrix can be converted into a reachability matrix, using a transitive closure algorithm[1]. The reachability matrix for our example is also in Fig. 6.3, showing that all SLOs eventually have impact on every other SLO. Consequently, the reachability matrix for this example is of limited use for assessing indirect impact. Bohner points out that this problem is common in software contexts, unless some action is taken to limit the range of indirect impact [4].

One way of limiting the range of indirect impact is to add distances to the reachability matrix. By doing so, it becomes possible to disregard indirect impacts with distances above a predefined threshold. This is a simple addition to the normal creation of reachability matrices, but it fails to address the fact that different types of traceability relationships may affect the range of indirect impact differently. Another solution is to equip the traceability matrix with traceability semantics and adjust the transitive closure algorithm to take such information into account. The algorithm should consider two SLOs reachable from each other only if the traceability relationships that form the path between them are of such types that are expected to propagate change.

Traceability analysis is useful in requirements engineering, which we view as an activity performed throughout the entire software lifecycle. Initially, traceability links can only be formed between requirements, but as design and implementation grow, links can be created from requirements to other SLOs as well.

6.2.1.2 Slicing Techniques

Slicing attempts to understand dependencies using independent slices of the program [16]. The program is sliced into a *decomposition slice*, which contains the place of the change, and the rest of the program, a *complement slice*. Slicing is based on data and control dependencies in the program. Changes made to the decomposition slice around the variable that the slice is based on are guaranteed not to affect the complement slice. Slicing limits the scope for propagation of change and makes that scope explicit. The technique is, for example, used by Turver and Munro [37] for slicing of documents in order to account for ripple effects as a part of impact analysis. Shahmehri *et al.* [33] apply the technique to debugging and testing. Pointer-based languages like C++ are supported through the work of Tip et al. and their slicing techniques for C++ [36]. Slicing tools are often based on character-based presentation techniques, which can make it more difficult to analyze dependencies, but visual presentation of slices can be applied to impact analysis as shown by Gallagher [15].

Architectural slicing was introduced by Zhao [42], and is similar to program slicing in that it identifies one slice of the architecture that is subject to the proposed change, and one that is not. As opposed to conventional program slicing, architectural slicing operates on the software architecture of a system. As such, it can be employed in early development, before the code has been written. The technique uses a graph of information flows in order to trace those components that may be affected by the component being changed. In addition, those compo-

[1] The **transitive closure** of a graph is a graph where an edge is added between nodes A and B if it is possible to reach B from A in the original graph.

nents that may affect the component being changed are also identified. This means that there must be a specification of the architecture that exposes all the information flows that it contains.

Slicing techniques can be useful in requirements engineering to isolate the impact of a requirements change to a specific part of the system. In order to provide a starting point for the slicing technique, the direct impact of the change must first be assessed.

6.2.2 Manual Strategies

Manual impact analysis strategies do not depend as heavily on structured specifications as their automatable counterparts do. Consequently, there is a risk that they are less precise in their predictions of impact. On the other hand, they may be easier to introduce in a change management process and are, in our experience, commonly employed in industry without regard to their precision.

The strategies presented here, using design documentation and interviewing, are primarily used for assessing the Starting Impact Set by identifying direct impact. The identification of secondary impact is possible, but is better handled by automatable strategies. Note that manual strategies, like the ones described here, can be used to capture traceability links between SLOs to be used in traceability analysis.

6.2.2.1 Design Documentation

Design documentation comes in many different forms, for example as architecture sketches, view-based architecture models, object-oriented UML diagrams, textual descriptions of software components and so on. The quality of design documentation depends on the purpose for which it was written, the frequency with which it is updated, and the information it contains. It is far too common in industry that design documentation is written early in a project only to become shelfware, or that the documentation is written after the project, just for the sake of writing it. To perform impact analysis and determine how a new or changed requirement affect the system based on design documentation requires the documentation to be up-to-date and consistent with any implementation made so far. In addition, a prerequisite for using design documentation to assess direct impact is the possibility of relating requirements to design SLOs found in the documentation. The success and precision of this activity depends on a number of factors:

- *The knowledge and skills of the persons performing the analysis.* Persons with little insight into the system will most likely have problems pinpointing the impact of changed requirements in the system.
- *The availability of the documentation.* Documentation that is "hidden" in personal computers or stored in anonymous binders may be overlooked in the analysis.
- *The amount of information conveyed in the documentation.* Simple design sketches are common, but fail to express the semantics in connections between

classes or architectural components. Ill-chosen naming schemes or inconsistent notation makes the analysis task arduous.
- *Clear and consistent documentation.* Ambiguous documentation is open for interpretation, meaning, for example, that the impact of a proposed change is coupled with great uncertainty, simply because another interpretation would have yielded different impact.

If the factors above have been taken into account, impact analysis of a requirements change can be performed by identifying the design SLOs that implement or in any other way depend on the requirements affected by the change. Additional measures that can be taken in order to alleviate the impact analysis effort are:

- *Keep a design rationale.* A design rationale is documentation describing why decisions are made the way they are. Bratthall et al. performed an experiment on the effect of a design rationale when performing impact analysis [7]. The results from the experiment suggest that a design rationale in some cases can shorten the time required for impact analysis, and increase the quality of the analysis.
- *Estimate impact of requirements as soon as the requirements are developed.* The estimated impact is necessarily coarse to begin with, but can be improved incrementally as knowledge about the system increases.

Of course, structured design documentation can also be used with traceability analysis (see Sect. 6.2.1.1) to identify indirect impact. For example, Briand et al. propose a method for performing impact analysis in UML models, where they use a transitive closure algorithm to find indirect impacts in the models [8]. They do point out, however, the essential criterion that the UML models are updated as the system undergoes changes.

6.2.2.2 Interviews

Interviewing knowledgeable developers is probably the most common way to acquire information about likely effects of new or changed requirements according to a study on impact analysis [25]. The study found that developers perceive it as highly cost-effective to ask a knowledgeable person instead of searching in documents or other forms of information sources. Extensive communication between developers was also mentioned by developers as a success factor for software development projects. Analysis of source code was the second most common way of acquiring information about the likely impact of new or changed requirements. While all developers said they interviewed other developers and consulted source code, about half of the developers answered that they also consulted information, such as use-case models and object models, stored in the CASE tool in use. When asked why information in object models was not used more extensively, the developers answered that the information in object models was not detailed enough for impact analysis. In addition, they did not believe that the information in the models was up-to-date. "Source code, on the other hand, is always up-to-date." Among some developers, especially newcomers, the attitude towards using object models as the basis for determining change as an effect of new or changed requirements was less than positive. Object models (and the particular CASE tool that was used) were, however, mentioned as a good tool for documenting impact analysis and for answering questions about the relation between requirements and design objects using the support for traceability links.

6.3 Non-Functional Requirements

Requirements are often divided into *functional* and *non-functional requirements*. Non-functional requirements, or quality requirements, are those requirements "which are not specifically concerned with the functionality of the system" [20]. Non-functional requirements are often harder to deal with than functional ones, because their impact is generally not localized to one part of the system, but cuts across the whole system.

A non-functional requirement that, for example, relates to and calls for high security, often requires fundamental support in the software architecture, as it may constrain data access, file management, database views, available functionality and so on. Changes to functional requirements may also affect non-functional requirements. For example, if a change involves replacing a data transfer protocol to one that is more data intensive, overall system performance may be degraded. One approach for dealing with non-functional requirements is to convert them into one or more functional requirements [6]. For example, a requirement stating that "no unauthorized person should be allowed access to the data" may be broken down into the more tangible requirements "a user must log into the system using a password" and "the user's identity must be verified against the login subsystem upon data access." Not all non-functional requirements can be converted in this way, however, which means that changes to them still have system-wide impact. Unfortunately, most impact analysis techniques deal exclusively with changes that can be initially pinpointed to a specific component, class or the like.

Lam and Shankararaman stress the distinction between functional impact analysis and quality impact analysis, i.e. impact analysis for functional and quality requirements, respectively [21]. They suggest the use of Quality Function Deployment (QFD) for dealing with changes to both functional and non-functional requirements. In QFD, a matrix connecting customer requirements with design features is constructed. A change to a requirement can be mapped to design features through the QFD matrix.

Cleland-Huang et al. accomplish performance-related impact analysis through event-based traceability [9]. In their approach, requirements are interconnected as event publishers to subscribing performance models. Whenever a change to a requirement is proposed, the relevant performance models are re-calculated. The resulting impact analysis is subsequently compared to constraints in the requirements specification. If several requirements are linked to the same performance model, they will all be verified against the impact analysis.

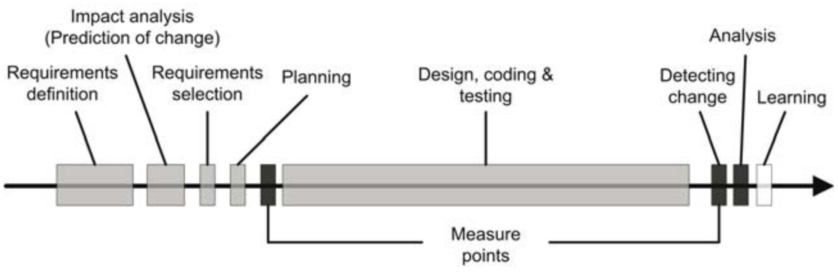


Fig. 6.4 Measuring impact using metrics

The impact of non-functional requirements is commonly dealt with in software architecture evaluation. Bosch has created a software architecture design method with a strong focus on non-functional requirements [6]. In the method, an initially functional architecture is progressively transformed until it is capable of meeting all non-functional requirements posed on the system. Parts of the method lend themselves well to impact analysis, since they deal with the challenge of assessing the often system-wide impact that non-functional requirements have. For most operational non-functional attributes (for example performance and reliability), a profile consisting of usage scenarios, describing typical uses of the system-to-be is created. The scenarios within the profile are assigned relative weights, in accordance with their frequency or probable occurrence. In scenario-based assessment, an impact analysis is performed by assessing the architectural impact of each scenario in the profile. For performance, the impact may be expressed as execution time, for example. Based on the impact and the relative weights of the scenarios, it is possible to calculate overall values (for example, throughput and execution time) for the quality attribute being evaluated. These values can be compared to the non-functional requirements corresponding to the quality attribute, in order to see whether they are met or not. Furthermore, they serve as constraints on the extent to which non-functional requirements can change before an architectural reorganization is necessary. Also, should a functional requirement change, it is possible to incorporate the change in a speculative architecture, re-calculate the impact of the scenarios in the scenario profile, and see whether the non-functional requirements are still met or not.

6.4 Impact Analysis Metrics

Metrics are useful in impact analysis for various reasons. They can, for example, be used to measure and quantify change caused by a new or changed requirement at the point of the impact analysis activity. Metrics can also be used to evaluate the impact analysis process itself once the changes have been implemented. This is illustrated in Fig. 6.4, in which two measure points are depicted; one after the requirements phase has ended and design is about to start, and one when testing has been completed. Using these measure points, one can capture the predicted impact (the first point) and compare it to the actual impact (the second point). This kind of measurement is crucial for being able to do an analysis and learn from experiences in order to continuously improve the impact analysis capability. The figure is simplified and illustrates a learning cycle based on a waterfall-like model. As discussed earlier, impact analysis can be used throughout the life cycle in order to analyze new requirements and the measure points can be applied accordingly: whenever a prediction has been conducted and whenever an implementation has been completed.

6.4.1 Metrics for Quantifying Change Impact

Metrics for quantifying change impact are based on the SLOs that are predicted to be changed as an effect of new or changed requirements. In addition, indicators of how severe the change is can be used. Such measures of the predicted impact can be used to estimate the cost of a proposed change or a new requirement. The more requirements and other SLOs that are affected, the more widespread they are and the more complex the proposed change is, the more expensive the new or changed requirement will be. Requirements that are costly in this sense but provide little value can, for example, be filtered out for the benefit of requirements that provide more value but to a smaller cost.

Change impact can be measured based on the set of requirements that is affected by the change. For example, the number of requirements affected by a change can be counted based on this set. The affected requirements' complexity often determines how severe the change is and can be measured in various ways. Examples are the size of each requirement in terms of function points and the dependencies of each requirement on other requirements. For other SLOs, the metrics are similar. For architecture and design, measures of impact include the number of affected components, the number of affected classes or modules, and number of affected methods or functions. For source code, low-level items such as affected lines of code can be measured and the level of complexity for components, classes, and methods can be measured using standard metrics such as cyclomatic complexity and regular object-oriented metrics.

In determining how severe or costly a change is, it is useful to define the *impact factor*. Lindvall defined the impact factors in Table 6.1 to measure the impact of a suggested change [25]. The impact factor is based on empirical findings in which it was determined that changes to different types of SLOs can be used as an indicator of the extent of the change. The higher the impact factor, the more severe the change. For example, changes that do not affect any other type of SLO but the design object model are relatively limited in scope. Changes that affect the use-case model are instead likely to require changes that are related to the fundamentals of the system and are therefore larger in scope. In addition, changes to the use-case model most likely also involve changes of all other SLOs making this kind of changes even more severe.

Table 6.1 Impact factors

Impact Factor	Impact	Description
M1	Change of the design object model.	These changes regard the real or physical description of the system and may generate change in the software architecture about the size of the change in the model.
M2	Change of the analysis object model.	These changes regard the ideal or logical description of the system. A small change here may generate change in the software architecture larger than the change in this model.
M3	Change the domain object model.	These changes regard the vocabulary needed in the system. A small change here may generate large change in the software architecture.
M4	Change the use-case model.	These changes require additions and deletions to the use-case model. Small changes here may require large change in the software architecture

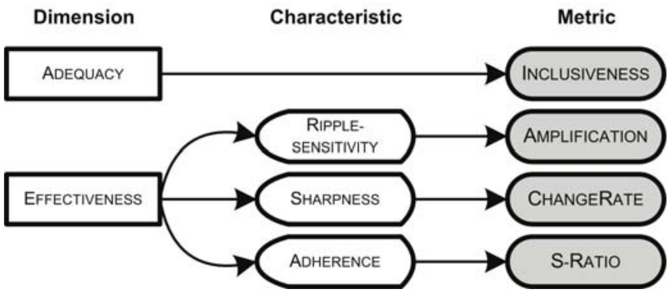


Fig. 6.5 Tree of impact analysis metrics

6.4.2 Metrics for Evaluation of Impact Analysis

Bohner and Arnold proposed a number of metrics with their introduction of impact sets [3]. These metrics are relations between the cardinalities of the impact sets, and can be seen as indicators of the effectiveness of the impact analysis approach employed (# denotes the cardinality of the set):

1. #SIS / #EIS, i.e. the number of SLOs initially thought to be affected over the number of SLOs estimated to be affected (primary change and secondary change). A ratio close to 1 is desired, as it indicates that the impact is restricted to the SLOs in SIS. A ratio much less than 1 indicates that many SLOs are targeted for indirect impact, which means that it will be time-consuming to check them.
2. #EIS / #System, i.e. the number of SLOs estimated to be affected over the number of SLOs in the system. The desired ratio is much less than 1, as it indi-

cates that the changes are restricted to a small part of the system. A ratio close to 1 would indicate either a faulty impact analysis approach or a system with extreme ripple effects.

3. #EIS / #AIS, i.e. the number of SLOs estimated to be affected over the number of SLOs actually affected. The desired ratio is 1, as it indicates that the impact was perfectly estimated. In reality, it is likely that the ratio is smaller than 1, indicating that the approach failed to estimate all impacts. Two special cases are if AIS and EIS only partly overlap or do not overlap at all, which also would indicate a failure of the impact analysis approach.

Fasolino and Visaggio also define metrics based on the cardinalities of the impact sets [14]. They tie the metrics to properties and characteristics of the impact analysis approach, as per the tree in Fig. 6.5.

Adequacy is the ability of the impact analysis approach to estimate the impact set. It is measured by means of the binary metric *Inclusiveness*, which is strictly defined to 1 if all SLOs in AIS also are in EIS and 0 otherwise. *Effectiveness* is the ability of the approach to provide beneficial results. It is refined into *Ripple-sensitivity* (the ability to identify ripple effects), *Sharpness* (the ability not to over-estimate the impact) and *Adherence* (the ability to estimate the correct impact).

Ripple-sensitivity is measured by *Amplification*, which is defined as (#EIS - #SIS) / #SIS, i.e. the ratio between the number of indirectly impacted SLOs and the number of directly impacted SLOs. This ratio should preferably not be much larger than 1, which would indicate much more indirect impact than direct impact. Sharpness is measured by *ChangeRate*, which is defined as #EIS / #System. This is the same metric as the second of Arnold and Bohner's metrics presented previously. Adherence is measured by *S-Ratio*, which is defined as #AIS / #EIS. S-Ratio is the converse of the third of Arnold and Bohner's metrics presented previously.

Lam and Shankararaman propose metrics that are not related to the impact sets. These metrics are more loosely defined and lack consequently recommended values [21]:

- *Quality deviation*, i.e. the difference in some quality attribute (for example, performance) before and after the changes have been implemented, or between actual and simulated values. A larger than expected difference could indicate that the impact analysis approach failed to identify all impact.
- *Defect count*, i.e. the number of defects that arise after the changes have been implemented. A large number of defects could indicate that some impact was overlooked by the impact analysis approach.
- *Dependency count*, i.e. the number of requirements that depend on a particular requirement. Requirements with high dependency count should be carefully examined when being subjected to change.

Lindvall [25] defined and used metrics in a study at the Swedish telecom company Ericsson AB in order to answer a number of questions related to the result (prediction) of impact analysis as conducted in a commercial software project and performed by the project developers as part of the regular project work. The study was based on impact analysis conducted in the requirements phase, as Fig. 6.4 indicates, and the term *requirements-driven impact analysis* was coined to capture this fact. The results from the impact analysis was used by the Ericsson project to estimate implementation cost and to select requirements for implementation based on the estimated cost versus perceived benefit. The study first looked at the collected set of requirements' predicted and actual impact by answering the following questions: "How good was the prediction of the change caused by new and changed requirements in terms of predicting the *number* of C++ classes to be changed?" and "How good was this prediction in terms of predicting *which* classes to be changed?" The last question was broken down into the two sub questions: "Were changed classes predicted?" and "Were the predicted classes changed?"

There were a total of 136 C++ classes in the software system. 30 of these were predicted to be changed. The analysis of the source code edits showed that 94 classes were actually changed. Thus, only 31.0% (30 / 94) of the number of changed classes were predicted to be changed.

In order to analyze the data further, the classes were divided into the two groups *Predictive group* and *Actual group*. In addition, each group was divided into two subgroups: *Unchanged* and *Changed*. The 136 classes were distributed among these four groups as shown in Table 6.2.

Table 6.2 Predicted vs. actual changes

		Predictive Group		
		Unchanged	**Changed**	
Actual Group	**Unchanged**	A: 42 **(30.9%)**	B:0 **(0.0%**	A+B: 42 **(30.9%)**
	Changed	C; 64 **(47.1%)**	D: 30 **(22.1%)**	C+D: 94 **(69.1%)**
		A+C: 106 **(77.9%)**	B+D: 30 **(22.1%)**	N: 136 **(100.0%)**

Cell A represents the 42 classes that were not predicted to change and that also remained unchanged. The prediction was correct as these classes were predicted to remain unchanged, which also turned out to be true. The prediction was implicit as these classes were indirectly identified –they resulted as a side effect as complement of predicting changed classes.

Cell B represents the zero classes that were predicted to change, but actually remained unchanged. A large number here would indicate a large deviation from the prediction.

Cell C represents the 64 classes that were not predicted to change, but turned out to be changed after all. As with cell B, a large number in this cell indicates a large deviation from the prediction.

Cell D, finally, represents the 30 classes that were predicted to be changed and were, in fact, changed. This is a correct prediction. A large number in this cell indicates a good prediction.

There are several ways to analyze the goodness of the prediction. One way is to calculate the percentage of correct predictions, which was (42 + 30) / 136 =

52.9%. Thus, the prediction was correct in about half of the cases. Another way is to use Cohen's Kappa value, which measures the agreement between two groups ranging from -1.0 to 1.0. The -1.0 figure means total discompliance between the two groups, 1.0 means total compliance and 0.0 means that the result is no better than pure chance [11]. The kappa value in this case is 0.22, which indicates a fair prediction. We refer to [26] for full details on the Kappa calculations for the example. A third way to evaluate the prediction is to compare the number of classes predicted to be changed with the number of classes actually changed. The number of classes predicted to be changed in this case turned out to be largely underpredicted by a factor of 3. Thus, only about one third of the set of changed classes was identified. It is, however, worth noticing that all of the classes that were predicted to be changed were in fact changed.

The study then analyzed the predicted and actual impact of each requirement by answering similar questions for each requirement. The requirements and the classes that were affected by these requirements were organized in the following manner: For each requirement, the set of classes predicted to be changed, the set of changed classes and the intersection of the two sets, i.e. classes that were both predicted and changed. In addition, the sets of classes that were predicted but not changed and the set of classes that were changed but not predicted were identified.

The analysis showed that in almost all cases, there was an underprediction in terms of number of classes. In summary, the analysis showed that the number of changed classes divided by the number of predicted classes ranged from 1.0 to 7.0. Thus, up to 7 times more classes than predicted were actually changed.

Estimating cost in requirements selection is often based on the prediction like it was in the Ericsson case, which means that requirements predicted to cause change in only a few entities are regarded as less expensive, while requirements predicted to cause change in many entities are regarded as more expensive. This makes the rank-order of requirements selection equal to a requirements list sorted by the number of items predicted. By comparing the relative order based on the number of predicted classes with the relative order based on the number of actually changed classes, it was possible to judge the goodness of the prediction from yet another point of view. In summary, the analysis on the requirements level showed that a majority of the requirements were underpredicted. It was also clear that it is relatively common that some classes predicted for one requirement are not changed because of this particular requirement, but because of some other requirement. This is probably because the developers were not required to implement the changed requirements exactly as was specified in the implementation proposal resulting from the impact analysis. The analysis of the order of requirements based on number of predicted classes showed that the order was not kept entirely intact; some requirements that were predicted to be small proved to have a large change impact, and vice versa.

In order to try to understand the requirements-driven impact analysis process and how to improve it, an analysis of the various characteristics of changed and unchanged classes was undertaken. One such characteristic was size, and the questions were: "Were large classes changed?", "Were large classes predicted?" and "Were large classes predicted compared to changed classes?"

The analysis indicated that large classes were changed, while small classes remained unchanged. The analysis also indicated that large classes were predicted to change, which leads to the conclusion that class size may be one of the ingredients used by developers, maybe unconsciously, when searching for candidates for a new or changed requirement.

6.5 Tool Support

The complexity of the change management process makes it necessary to use some sort of tool support [27, 35]. A change management tool can be used to manage requirements and other SLOs, manage change requests, link change requests to requirements and other SLOs, and monitor the impact analysis progress. A simple database or spreadsheet tool may be used as basic change management support, but still requires a considerable amount of manual work, which eventually may lead to inconsistencies in the change management data. If the tool support is not an integral part of the change management process, there is always a risk that it will not be used properly. A change management system that is not used to its full extent cannot provide proper support to the process.

A problem with many change management tools is that they are restricted to working with change and impact analysis on the requirements level. Ideally, a change management tool would support impact analysis on requirements, design, source code, test cases and so on. However, that would require the integration of requirement management tools, design tools and development environments into one tool or tool set. In a requirements catalog for requirements management tools, Hoffmann *et al*. list both *traceability* and *tool integration* as high-priority requirements, and *analysis functions* as a mid-priority requirement, confirming the importance of these features [19].

In a survey of the features of 29 requirements management tools supporting traceability, we could only find nine tools for which it was explicitly stated on their web sites that they supported traceability between requirements and other SLOs, such as design elements, test cases and code. Depending on the verbosity and quality of the available information, this may not be an exact figure. However, it indicates that in many cases it is necessary to use several different tools to manage traceability and perform impact analysis, which can be problematic depending on the degree of integration between the tools.

There are tools that extract dependency information from existing system representations, for example source code and object models, but the task of such tools is nonetheless difficult and often requires manual work [12]. Higher-level representations may be too coarse, and source code may have hidden dependencies, for instance due to late binding. Egyed, for example, proposes an approach for extracting dependencies primarily for source code [12]. Input to the approach is a set of test scenarios and some hypothesized traces that link SLOs to scenarios. The approach then calculates the footprints of the scenarios, i.e. the source code lines they cover, and based on footprints and hypothesized traces generates the remaining traces. The approach can also be used when no source code exists, for example by simulating the system or hypothesizing around the footprints of the scenarios.

Tools that deal with source code are mostly used in software maintenance contexts, and are obviously of limited use within the development project. Natt och Dag et al. have studied automatic similarity analysis as a means to find duplicate requirements in market-driven development [29]. In addition to the original field of application, they suggest that their technique can be used to identify dependency relationships between requirements, for example that two requirements have an "or" relation, or that several requirements deal with similar functionality. How to deal with natural language requirements is further explored in Chap. 10. Tools that aid in performing impact analysis can be synonymous with the underlying methods. Methods that rely on traceability analysis are well suited for inclusion in tools that try to predict indirect impact. For example, Fasolino and Visaggio present ANALYST, a tool that assesses impact in dependency-based models [14]. Lee *et al*. present another tool, ChAT, which calculates ripple effects caused by a change to the system [22]. Many such tools are commonly proof-of-concept tools, constructed to show or support a particular algorithm or methodology. What is lacking is the integration into mainstream change management tools.

6.6 Future of Impact Analysis

Most strategies for impact analysis work under the assumption that changes only affect functionality. It is thus more difficult to assess the impact of changes to non-functional requirements, or changes where non-functional requirements are indirectly affected. Some work on this topic exists (see [9] and [21]), but a stronger focus on impact analysis for non-functional requirements is needed.

As we have pointed out, impact analysis is mostly referred to in software maintenance contexts. We have argued that impact analysis is an essential activity also in requirements engineering contexts, and that standard impact analysis strategies apply in most cases (for example, traceability approaches are commonly exercised for requirements). There is still, however, a need for more research focusing on the requirements engineering aspects of impact analysis, for example, how to relate requirements to other SLOs and how to perform change propagation in this context. Most automatable strategies for impact analysis assume complete models and full traceability information. Since it is common in industry to encounter models that are not updated and traceability information that is only partial, there is a need for more robust impact analysis strategies that can work with partial information. Egyed has proposed one such approach [12]. Existing tools for impact analysis are often proof-of-concept tools, or work only with limited impact analysis problems, such as the extraction of dependencies from system representations. Some mainstream requirements management tools incorporate impact analysis of not only requirements, but also design, code and test, but far from all these things. Full-scale impact analysis must be an integral part of requirement management tools in order for change to be dealt with properly. Impact analysis needs to be adapted to the types of systems that become increasingly common today, such as web applications and COTS software. Web applications, for example, often consist of standalone components that connect to a central repository, such as a database. Thus, there are few control dependencies between components, and instead rich webs of data dependencies towards and within the central repository. The fact that such repositories can be shared among several distinct systems introduces interoperability dependencies that impact analysis strategies especially tailored for these technologies must address in order to be effective.

6.7 Summary

Impact analysis is an important part of requirements engineering since changes to software often are initiated by changes to the requirements. As the development process becomes less and less waterfall-like and more of new and changed requirements can be expected throughout the development process, impact analysis becomes an integral part of every phase in software development. In some sense, impact analysis has been performed for a very long time, albeit not necessarily using that term and not necessarily fully resolving the problem of accurately determining the effect of a proposed change. The need for software practitioners to determine what to change in order to implement requirement changes has always been present. Classical methods and strategies to conduct impact analysis are dependency analysis, traceability analysis and slicing. Early impact analysis work focused on applying such methods and strategies onto source code in order to conduct program slicing and determine ripple effects for code changes. The maturation of software engineering among software organizations has, however, led to a need to understand how change requests affect other SLOs than source code, including requirements, and the same methods and strategies have been applied. Typical methods and strategies of today are based on analyzing traceability or dependency information, utilizing slicing techniques, consulting design specifications and other documentation, and interviewing knowledgeable developers. Interviewing knowledgeable developers is probably the most common way to acquire information about likely effects of new or changed requirements. Metrics are useful and important in impact analysis for various reasons. Metrics can, for example, be used to measure and quantify change caused by a new or changed requirement at the point of the impact analysis activity. Metrics can also be used to evaluate the impact analysis process itself once the changes have been implemented. In determining how severe or costly a change is, it is useful to determine the impact factor as it indicates the likely extent of a change to a certain type of SLO. To summarize: Impact analysis is a crucial activity supporting requirements engineering. The results from impact analysis feed into many activities including estimation of requirements' cost and prioritizing of requirements. These activities feed directly into project planning, making impact analysis a central activity in a successful project.

Acknowledgements

We would like to thank Jen Dix for proof reading, and the anonymous reviewers for helping to improve the chapter.

References

1. ANSI/IEEE Std 830-1984 (1984) IEEE guide to software requirements specifications, Institute of the Electrical and Electronics Engineers
2. Bass L, Clements P, Kazman R (2003) Software architecture in practice, Addison Wesley
3. Bohner SA, Arnold RS (1996) Software change impact analysis, IEEE Computer Society Press
4. Bohner SA (2002) Extending software change impact analysis into COTS components. In: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, December 4−6, Greenbelt, USA, pp.175−182
5. Bohner SA, Gracanin D (2003) Software impact analysis in a virtual environment. In: Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop, December 2−4, Greenbelt, USA, pp.143−151
6. Bosch J (2000) Design & use of software architectures - Adopting and evolving a product-line approach. Pearson Education, UK
7. Bratthall L, Johansson E, Regnell B (2000) Is a design rationale vital when predicting change impact? - A controlled experiment on software architecture evolution. In: Proceedings of the 2nd International Conference on Product Focused Software Process Improvement, June 20-22, Oulo, Finland, pp.126−139
8. Briand LC, Labiche Y, O'Sullivan L (2003) Impact analysis and change management of UML models. In: Proceedings of the International Conference on Software Maintenance, September 22−26, Amsterdam, Netherlands, pp 256−265
9. Cleland-Huang J, Chang CK, Wise JC (2003) Automating performance-related impact analysis through event based traceability. Requirements Engineering 8(3):171−182
10. Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J (2003) Documenting software architectures: Views and beyond. Addison Wesley, UK
11. Cohen J (1960) A coefficient of agreement for nominal scales, educational and psychological measurement 20(1):37−46
12. Egyed A (2003) A scenario-driven approach to trace dependency analysis. IEEE Transactions on Software Engineering 29(2):116−132
13. Eick SG, Graves L, Karr AF, Marron JS (2001) Does code decay? Assessing the evidence from change management data. IEEE Transactions on Software Engineering 27(1):1−12
14. Fasolino AR, Visaggio G (1999) Improving software comprehension through an automated dependency tracer. In: Proceedings of the 7th International Workshop on Program Comprehension, May 5−7, Pittsburgh, USA, pp 58−65
15. Gallagher KB (1996) Visual impact analysis. In: Proceedings of the International Conference on Software Maintenance, November 4−8, Monterey, USA, pp 52−58
16. Gallagher KB, Lyle JR (1991) Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17(8):751−761

17. Godfrey LW, Lee EHS (2000) Secrets from the monster - Extracting Mozilla's software architecture. In: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, Limerick, Ireland, pp 15−23

18. Haney FM (1972) Module connection analysis - A tool for scheduling software debugging activities. In Proceedings of AFIPS Joint Computer Conference, pp 173−179

19. Hoffmann M, Kühn N, Bittner M (2004) Requirements for requirements management tools. In: Proceedings of the 12th IEEE International Requirements Engineering Conference, September 6−10, Kyoto, Japan, pp 301−308

20. Kotonya G, Sommerville I (1998) Requirements engineering - Processes and techniques. Wiley and Sons, UK

21. Lam W, Shankararaman V (1999) Requirements change: A dissection of management issues. In: Proceedings of the 25th EuroMicro Conference, September 8−10, Milan, Italy, Vol. 2, pp.244−251

22. Lee M, Offutt JA, Alexander RT (2000) Algorithmic analysis of the impacts of changes to object-oriented software. In: Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems, July 30−Aug 4, Santa Barbara, USA, pp 61−70

23. Leffingwell D, Widrig D (1999) Managing software requirements - A unified approach. Addison Wesley

24. Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution - The nineties view. In: Proceedings of the 4th International Software Metrics Symposium, November 5-7, Albuquerque, USA, pp 20−32

25. Lindvall M (1997) An empirical study of requirements-driven impact analysis in object-oriented systems evolution. Ph.D. thesis no. 480, Linköping Studies in Science and Technology, Sweden

26. Lindvall M, Sandahl K (1998) How well do experienced software developers predict software change?, Journal of Systems and Software 43(1):19−27

27. Maciaszek L (2001) Requirements analysis and system design - Developing information systems with UML, Addison Wesley

28. Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: Proceedings of the International Conference on Software Maintenance, October 11-14, San Jose, USA, pp 120−130

29. Natt och Dag J, Regnell B, Carlshamre P, Andersson M, Karlsson J (2002) A feasibility study of automated support for similarity analysis of natural language requirements in market-driven development. Requirements Engineering 7:20−33

30. O'Neal JS, Carver DL (2001) Analyzing the impact of changing requirements. In: Proceedings of the International Conference on Software Maintenance, November 6-10, Florence, Italy, pp.190−195

31. Ramesh B, Jarke M (2001) Towards reference models for requirements traceability. IEEE Transactions on Software Engineering 27(1): 58−93

32. Robertson S, Robertson J (1999) Mastering the requirements process. Addison Wesley, UK

33. Shahmehri N, Kamkar M, Fritzson P (1990) Semi-automatic bug localization in software maintenance. In: Proceedings of the Conference on Software Maintenance, November 26-29, San Diego, USA, pp 30−36

34. Software Engineering Institute (2004): How do you define software architecture?, http://www.sei.cmu.edu/architecture/definitions.html, Accessed November 19, 2004.

35. Sommerville I, Sawyer P (1997) Requirements engineering - A good practice guide. John Wiley and Sons, London

36. Tip F, Jong DC, Field J, Ramlingam G (1996) Slicing class hierarchies in C++. In: Proceedings of Object-Oriented Programming, Systems, Languages & Applications Conference, October 6-10, San Jose, USA, pp 179−197

37. Turver RJ, Munro M (1994) An early impact analysis technique for software maintenance. Journal of Software Maintenance Research and Practice 6(1):35−52

38. Weinberg GM (1983) Kill that code. Infosystems 30: 48−49

39. Weiser M (1979) Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan, Michigan, USA

40. Wiegers KE (2003): Software requirements. Microsoft Press

41. Yau SS, Collofello JS (1980) Some stability measures for software maintenance. IEEE Transactions on Software Engineering 6(6): 545−552

42. Zhao J (1998) Applying slicing technique to software architectures. In: Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems, August 10-14, Monterey, USA, pp.87−98

Author Biography

Per Jönsson is a Ph.D. student in Software Engineering at the School of Engineering at Blekinge Institute of Technology in Sweden, where he also received his Degree of Master of Science in Software Engineering in 2002. His main research interest is impact analysis on a software architecture level. This touches the boundary between requirements engineering and software architecture, and includes questions about how requirements affect the architecture, but also how architectures are created, changed, maintained and merged.

Dr. Mikael Lindvall is a scientist at Fraunhofer Center Maryland. He manages the center's participation in NASA's High Dependability Computing Project. He heads test bed development for experimenting with and determining technologies' impact on software dependability and studies how best practices, lessons learned and other experience and knowledge management strategies are best applied in software engineering. He studies software architecture evaluation and evolution to efficiently understand software architectures and to identify architectural violations. Lindvall received a Ph.D. from Linköping University, Sweden 1997, on impact analysis and evolution of object-oriented systems at Ericsson Radio in Sweden.