# Proposing and empirically validating change impact analysis metrics

Arvanitou, Elvira Maria

# Chapter 1 - Introduction

In the literature one can identify various ways to define the term "software quality". According to Kitchenham et al. (1995), software quality is a complex and multifaceted notion, which can be recognized, but not easily defined. For example, from the viewpoint of the end-user, quality is related to the appropriateness of the software for a particular purpose. From the software engineer's point of view, quality deals with the compliance of software to its specifications. From the product viewpoint, quality is related to the inherent characteristics of the product, while from a monetary viewpoint, quality depends on the amount that a customer is willing to pay to obtain it. To ease the management of software quality, stakeholders (e.g., software engineers, end-users, customers, etc.) usually negotiate and specify certain quality attributes (QAs) of interest for their projects.

Quality attributes are organized into quality models, which in the majority of the cases are organized in a hierarchical manner (ISO-9126, 2001; ISO-25010, 2011; McCall et al., 1977; Bohem et al., 1978; Bansiya and Davis, 2002): high-level (HL) quality attributes are decomposed into Lower-Level (LL) ones (some quality models include more than one levels of LLs), which are subsequently mapped to quality properties that are directly quantified by software metrics.


**Figure 1.a: ISO 25010 Hierarchical Structure**

For example, in the ISO/IEC 25010 model, product quality is defined as follows (see Figure 1.a):

- ***the first level*** (HL / characteristics) separates product quality into eight QAs (e.g., Maintainability, Functional suitability, etc.);
- ***the second level*** (LL / sub-characteristics) decomposes each quality attribute into sub-characteristics, e.g., Maintainability is decomposed into Testability, Modifiability, etc.);

The LL sub-characteristics can be evaluated by measuring internal quality properties (typically static measures of intermediate products), or by measuring external quality properties (typically by measuring the behaviour of the code when executed), or by measuring quality in use properties (when the product is in real or simulated use) (Figure 1.b) (ISO-25010, 2011). Figure 1.b shows the relationship between measurable internal object-oriented software properties, in which we focus in this thesis, and external quality attributes (ISO-25010, 2011).

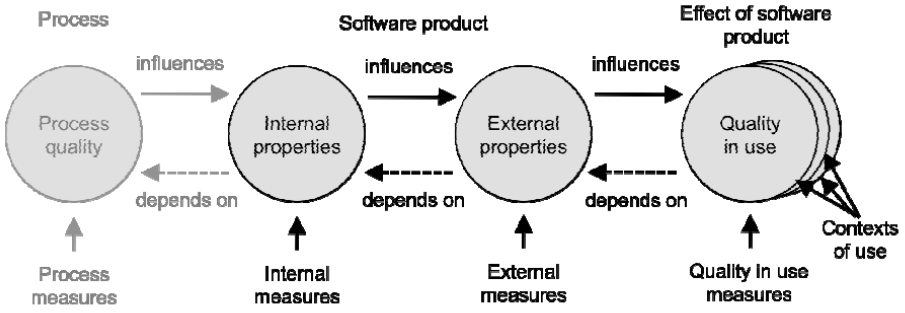


**Figure 1.b: Product Quality (Internal and External) and Quality in Use (ISO-25010, 2011)**

## 1.1 Software Quality Models, Attributes and Metrics

ISO-25010 is one of the most well-known international standards for assessing software quality. ISO-25010 defines a set of software quality attributes (i.e., characteristics) and metrics (ISO-25010, 2011). Specifically, it identifies eight (8) main quality attributes that compose *product quality*, defined as follows (ISO-25010, 2011):

- ***Functional Suitability*:** The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.
- ***Performance Efficiency:*** The performance relative to the amount of resources used under stated conditions.
- ***Usability:*** The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
- ***Compatibility:*** The degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment.
- ***Maintainability:*** The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
- ***Reliability:*** The degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
- ***Security***: The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
- ***Portability:*** The degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

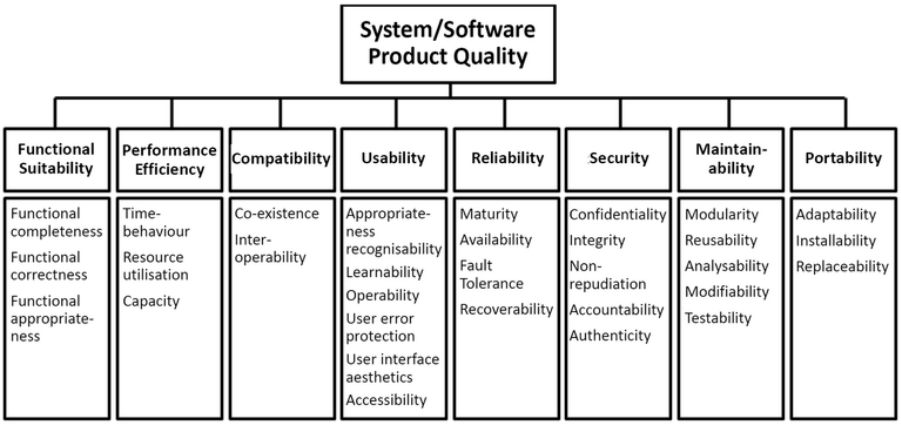


**Figure 1.1.a: ISO 25010 Product Quality Model**

These quality attributes are decomposed into 31 sub-QAs, and subsequently the standard defines metrics that assess these sub-QAs. For example (see Figure 1.1.a), maintainability is decomposed to: modularity, reusability, analysability, modifiability, and testability.

Software metrics can be calculated at various levels of granularity and on different artifacts. The most commonly used metrics in practice are source-code (i.e., those calculated on classes, methods, etc.) and design metrics (i.e., those that can be calculated on design artifacts—e.g., UML class diagrams) (Arvanitou et al., 2017a). The basic advantage of source-code level metrics is that they provide an insight into the system being developed and help to understand which parts of the source-code need maintenance (e.g. refactoring). Source-code level metrics are highly accurate, but can only be calculated during the implementation phase. On the contrary, metrics at the design level are not so accurate, but can be calculated earlier, and provide estimates on the final quality of the software. Additionally, a precondition for using such metrics is that a software engineering team should have access to design artifacts. For example, supposing that the object-oriented development paradigm is used, artifacts that describe class and object definitions, class hierarchies, etc. would be required. More details on these metrics can be found in Chapter 2.

## 1.2 Software Maintainability, Instability, Change Proneness

In this thesis we focus on one of the QAs defined in the ISO-25010 model, namely maintainability. Maintenance is one of the most effort-consuming activities in the software engineering lifecycle, in the sense that it consumes 50 - 75% of the total time / effort budget of a typical software project. Therefore, monitoring and quantifying the maintainability of a software system is crucial. In this PhD thesis, we adopt the ISO-25010 definition for maintainability as the "*software quality characteristic concerning the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers*". ISO-25010 decomposes maintainability to six sub-QAs (ISO-25010, 2011):

- ***Modularity***, i.e., the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

- ***Reusability***, i.e., the degree to which an asset can be used in more than one system, or in building other assets.
- ***Analysability***, i.e., the degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- ***Modifiability***, i.e., the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing quality.
- ***Testability***, i.e., the degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

From the aforementioned sub-QAs, we further focus on software modifiability, and in particular on one of its sub-characteristics, namely **stability** (and it's opposite: *instability*) (ISO-25010, 2011). Based on ISO-9126, stability "*characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes*" (ISO-9126, 2001). According to Galorath (2008) and Chen and Huang (2009) maintenance costs are increased by up to 75% if the software is unstable. In the literature, one can identify a term similar to instability, namely **change proneness**; however the two notions differ as follows:

- Change proneness is a measurement of all changes that occur to an artifact (e.g., new requirements, debugging, change propagation, etc.) (Jaafar et al., 2014), whereas stability only refers to the last type of change (propagation of changes to other artifacts).
- Change proneness is usually calculated from the actual changes that occur in an artifact (a posteriori analysis), whereas stability can be calculated a priori.

Although instability and change proneness are closely related concepts that can be characterized as two sides of the same coin, there may be cases in which they are not correlated. For example, a class heavily depending on other classes would be highly unstable; however, if this class does not actually change, then its change proneness would be low.

In order to quantify change proneness, two specific parameters need to be assessed: (a) the change proneness of the artifact that emits the change (e.g., a class), and (b) the instability of the connector between this artifact and the ones that depend upon it (e.g., classes that inherit the source class). For example, in Figure 1.2.a, we consider a system of four artifacts (e.g., classes, packages, requirements, etc.). Artifact A, can be changed for two reasons: (a) due to internal reasons (e.g., a bug is identified in it, a change in its requirements occur, etc.), or (b) due to a change in another artifact that propagates to it (e.g., from Artifact B1, B2, or B3) through a dependency (external probability to change). Subsequently, to quantify the probability of a change occurring in Artifact B1 to propagate to Artifact A one needs to consider: (a) internal probability of B1 to change (***change proneness***), and (b) the strength of the dependency between A and B1 (***instability***).

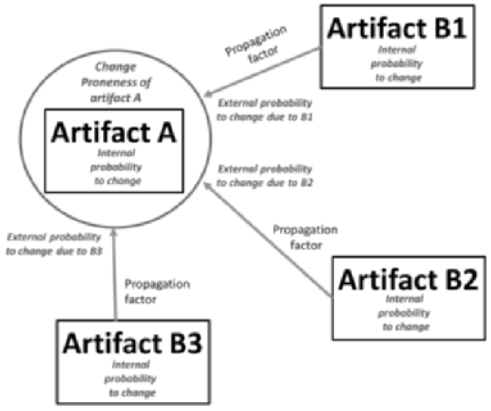


**Figure 1.2.a: Change Proneness and Instability Relation**

The main usefulness of stability and change proneness metrics are for performing Change Impact Analysis (CIA): this is the process of investigating the undesired consequences of a change in a software module (Bohner, 1996). Change impact analysis can be useful both before and after the application of the change. Before the application of the change, CIA can be useful for effort estimation; for example, knowing how many classes will need to be checked, after changing a specific module, can be an indicator of the maintenance effort (Haney, 1972). After the application of a change, CIA can be useful for test case prioritization; for example, having in mind which requirements are related can

be used as an efficient way to integrate specific test cases in the test planning of a software release (Rovegard et al., 2008).

## 1.3 Research Design

In Chapter 1.3.1 we discuss the problem statement that is addressed in this thesis. Next, in Chapter 1.3.2 we present the employed research methodology, whereas in Chapter 1.3.3 we present the research questions that the thesis deals with. Finally, in Chapter 1.3.4 we present the used empirical research methods, and in Chapter 1.3.5 we conclude with an overview of this research.

### 1.3.1 Problem Statement

In the literature, only a limited number of metrics for instability and change proneness have been proposed (more details on design-time quality metrics are presented in Chapter 2). In particular, change proneness and instability have been quantified by eight measures at the implementation level (e.g., (Black, 2008)), six at the detailed-design level (e.g., (Yau and Collofello, 1981)), and none at the architecture and requirements level (Arvanitou et al., 2017a). Due to the lack of metrics at these two levels, change impact analysis cannot be performed based on objective / quantitative data. Consequently, there is a ***need to introduce change proneness and instability metrics at the architecture and requirements level***.

Furthermore, at the detailed-design and implementation level we have identified two limitations. First, the ***accuracy of the metric-based approaches is rather low***, since they do not take into account both change proneness and instability so as to combine their predictive power; consequently, low accuracy of the metrics results in ineffective and inefficient change impact analysis. Second, most of the existing approaches lack applicability, in the sense that ***they do not provide tools***. Thus, since the calculation of existing metrics is not automated, they cannot be applied to large-scale systems.

Concluding, the state-of-the-art on change proneness and instability measures, suffers from the following limitations:

a. There are no metrics available for requirements and architecture development phases.

b. The metrics that exist for the detailed-design and implementation levels are not accurate enough, since they do not combine change proneness and instability.

c. There is limited tool support for assessing change proneness and instability.

Therefore, the problem statement that this PhD thesis attempts to resolve can be summarized as follows:

"*Current change impact analysis practices that are based on instability and change proneness, are not supported: (a) by metrics for requirements and architecture development phases, (b) by metrics that consider both change proneness and instability, and (c) by automated tools that quantify change proneness and instability*"

### 1.3.2 Design Science as Research Methodology

In this chapter we present the research approach that has been used, namely *Design Science*. In this dissertation, we have adopted the design science framework described by Wieringa (2009)—as outlined in Figure 1.3.2.a.

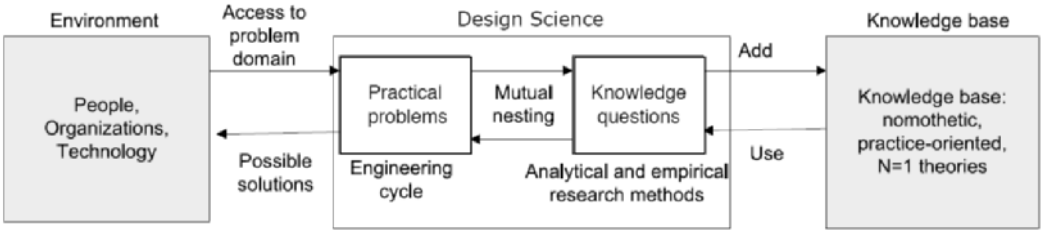


**Figure 1.3.2.a: Research Methodology Outline**

As observed in the previous figure, design science is inherently practice-oriented, getting inspiration for identifying needs from the environment (e.g., people, organizations, technology, etc.), as a starting point and decomposes the identified problem statement into two type of problems: (a) practical problems, and (b) knowledge questions (see Design Science box in Figure 1.3.2.a). A practical problem is defined as "*a difference between the way the world is experienced by stakeholders and the way they would like it (the world) to be*"; a knowledge question is "*a difference between current knowledge of stakeholders about the world and what they would like to know*". A practical problem is usually solved by applying an engineering cycle (Wieringa, 2009), whereas

knowledge questions are answered with empirical or analytical research methods. For example, in the software measurement domain, a practical problem could be: *Tailor a cohesion metric that is calculated at the class level, to make it applicable at the method level with the ability to signify the need for splitting a long method*. The above problem is a practical problem one, in the sense that it aims at proposing a new metric. However, it also implicitly entails at least two knowledge questions: *What are the available cohesion metrics at the class level*? and for evaluation purposes: *Does the proposed metric capture the expected properties of cohesion (i.e., signify the need for splitting a large artifact)*? These are knowledge questions, because they aim at increasing the knowledge that we already have on the practical problem (i.e., by adding or using existing knowledge bases). This is one example for the nested nature of practical problems and knowledge questions (see Figure 1.3.2.a). Applying design science is an iterative process, in the sense that the researcher starts from a practical problem statement, extracts and analyzes a practical problem, proposes a solution, evaluates the solution, and then starts over again, or digs even further by investigating possibly nested problems. These iterations are termed design cycles (Hevner, 2007). The design science framework is particularly suitable for describing long-term research like PhD thesis, because it allows to present the evolution of research questions and solutions at the same time.

### 1.3.3 Practical Problems and Knowledge Questions

In this chapter, we present the practical problems and knowledge questions addressed in this thesis, and how each one follows up on another. Figure 1.3.3.a depicts the problems and questions: grey boxes represent knowledge questions and white boxes represent practical problems. Moreover, hollow arrows denote sequence whereas solid arrows denote decomposition. We refer to both practical problems and knowledge questions as research questions. The main research questions are labeled with Arabic numbers from one to three. The research sub-questions are numbered with lowercase letters. A special case is $RQ_3$, which is decomposed into four levels: there are three sub-questions, one for the source-code level, one for the architecture level, and one for the requirements level; next, each one of these sub-questions deals is further decomposed. Instability has been separately investigated only at the source-code level ($RQ_{3.a.i}$): at the other levels (architecture and requirements),

instability metrics have been incorporated when proposing the change proneness metrics. Thus, no questions on instability are set for $RQ_{3.b}$ and $RQ_{3.c}$.

As already explained in Chapter 1.3.1, the major goal of this thesis is to support the calculation of change proneness and instability metrics (through methods), and the provision of corresponding tools that can automate these calculations. The methods and tools will be able to guide the change impact analysis process, along the requirements, architecture, and implementation phases. As a first step towards achieving this goal, we have reviewed the literature in order to explore the relevant quality attributes and *identify existing metrics* that are able to quantify instability and change proneness. In fact, we investigated all design-time qualities (instead of only stability and change proneness), because we aimed at a more comprehensive study, to make sure that we do not miss studies related to change impact analysis (since quality attributes are sometimes referred with a different name). Thus, we set a broader research question, stated as follows, $RQ_1$: *Which are the most important design-time quality attributes, and how can they be measured*? To answer this knowledge question, we investigated two sub-questions: (a) $RQ_{1.a}$: *Which design-time quality attributes should be considered in a software development project*? (b) $RQ_{1.b}$: *Which metrics can be used for assessing/quantifying design-time QAs*?

Based on the answer to $RQ_{1.a}$, we have observed that instability and change proneness are among the most studied quality attributes for all development phases; however (based on $RQ_{1.b}$), the metrics that have been proposed for assessing them are limited. These results strengthen the main problem statement as they substantiate the importance of the selected QAs (instability and change proneness) and the lack of metrics for quantifying them. In other words, we have been able to provide evidence on the relevance of the problem statement, and at the same time we built a corpus of related work that can be used for the rest of the dissertation.

Based on the main finding of $RQ_{1.b}$—the lack of metrics for some development phases (particularly requirements and architecture), as well as the plethora of available metrics at source-code level— as a next step we explored whether code metrics can be applied at the architecture phase. To this end, we needed to investigate: (a) the applicability of source-code metrics for the architecture phases and (b) the aggregation functions that can be used for elevating metrics

from the source-code to the architecture level. Consequently, we investigated a set of metrics that have been identified as maintainability predictors (the best available, based on the literature). Similarly to $RQ_1$, we selected to open the scope of this study to maintainability-related metrics, rather than instability and change proneness only, in order not to miss any relevant metrics.

Thus, the first practical problem that we investigated is $RQ_2$: *Are maintainability prediction source-code metrics applicable at the architecture phase?* In particular, we approach the metric selection, based on the ability of a metric to capture fluctuations of metric scores along evolution, by considering that fine-grained changes are more probable to be important at the method and class level (i.e., implementation phase), whereas, architecture metrics should be sensitive only to more coarse-grained changes. Apart from the formula of metrics calculation, another parameter that we consider is the use of aggregation functions that can be used for elevating source-code metrics to the architecture phase. To answer this RQ, we divided it into 4 sub-questions:

(a) $RQ_{2.a}$: *Are maintainability prediction metrics able to capture fine- or coarse-grained changes that are expected to occur in different development phases*? This question helped us to understand which metrics are capable of capturing small-scale and which large-scale changes, which are expected to occur at different levels of granularity (e.g., an architectural metric should be sensitive only to extensive changes, whereas a class metric should be sensitive to even the smallest changes in the code bases);

(b) $RQ_{2.b}$: *Can different aggregation functions lead to differences in the way maintainability prediction metrics are able to capture fine- or coarse-grained changes*? Next, we focused on the most common aggregation functions (e.g., average, sum, etc.) that can be used for aggregating metrics in artifacts from a fine-grained level of granularity (e.g., class) to a coarse-grained or architectural level (e.g., package). In particular, we investigated if the use of different aggregation functions can lead to changes in the previously mentioned metrics fluctuation;

(c) $RQ_{2.c}$: *Propose a metric property that can assess the suitability of metrics in a specific development phase*. To objectively assess the ability of metrics to capture the aforementioned fluctuations we proposed a metric property, called Software Metrics Fluctuation (SMF);

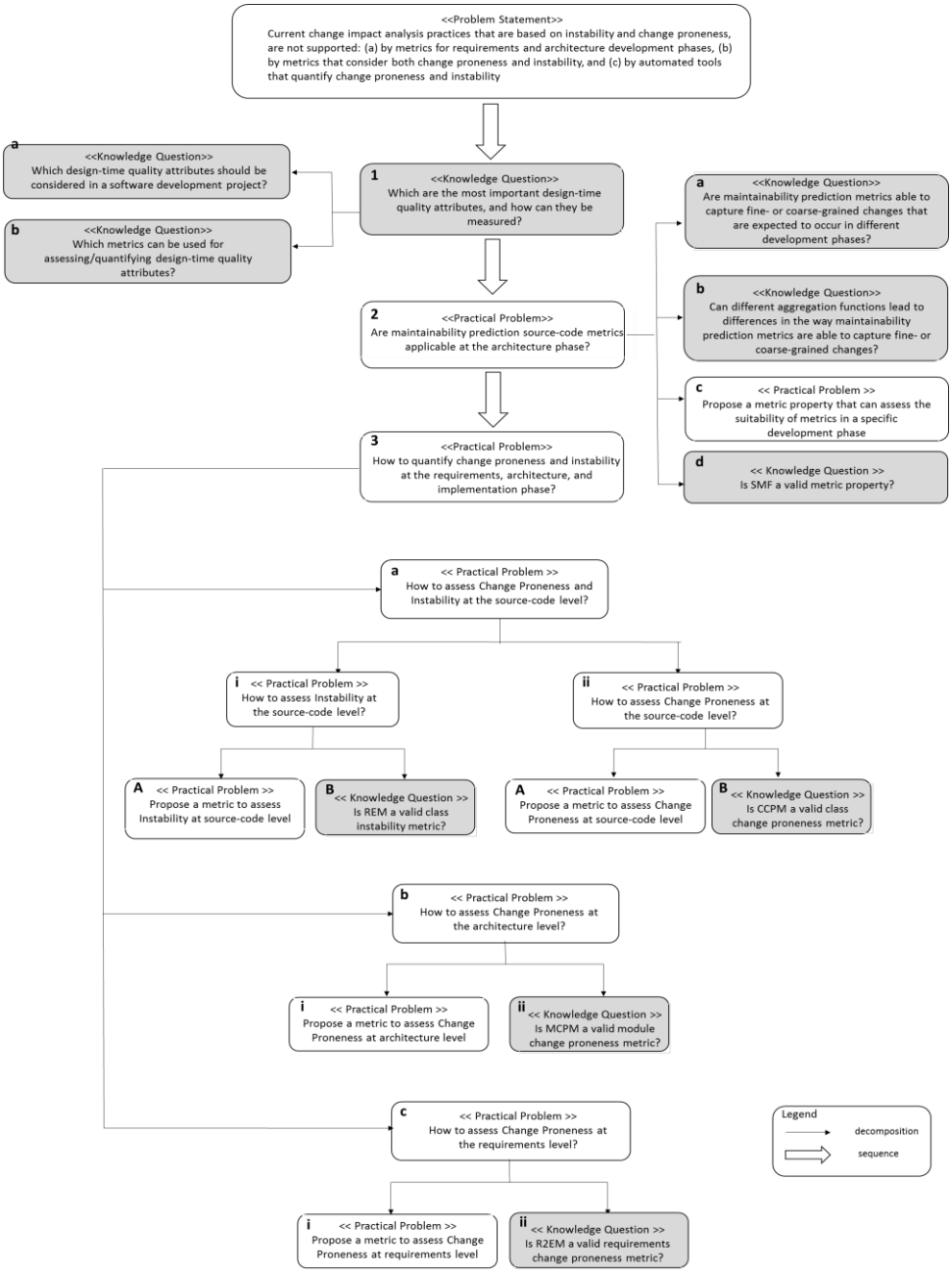


**Figure 1.3.3.a: Research questions addressed in this research PhD thesis**

(d) $RQ_{2.d}$ (*Is SMF a valid metric property*?). Specifically we assessed whether the SMF is a metric property that correlates to the expert opinion of software engineers.

As a result of $RQ_2$, we identified only one maintainability prediction source-code metric that can be applicable to the architecture level and is related to instability. However, this metric is not purely instability-related, since it takes into account both the dependencies to other classes, but also the size of the class. Therefore, we concluded that a novel architecture-level metric should be introduced that focuses on instability and change proneness characteristics.

Based on the answers to both $RQ_1$ and $RQ_2$, we concluded that there is a need for the ***introduction of dedicated, high-accuracy metrics for change proneness and instability for the requirements, architecture, and source-code level***. To proceed in this direction we have used as input the results from $RQ_1$ and $RQ_2$, as follows. From the first research question, we collected a set of proposed metrics for instability and change proneness quantification; we are thus able to compare their levels of validity to the metrics we derive in $RQ_3$ (*How to quantify change proneness and instability at the requirements, architecture, and implementation phase?*). From answering the second research question, we understood that a different metric is required for each development phase, and that we should pay special attention in metric construction on the selection of aggregation functions; this is exactly what we did when introducing the new metrics in $RQ_3$. $RQ_3$ is decomposed into three levels: requirements, architecture and implementation.

First, in ($RQ_{3.a}$: *How to assess Change Proneness and Instability at the source-code level*?) we focused at the source-code level. As explained at the end of Chapter 1.2, in order to be able to assess change proneness, we first need to assess instability. Thus, in $RQ_{3.a.i}$ (*How to assess Instability at the Source-Code level*?) we proposed and evaluated the Ripple Effect Measure (REM), which is an assessor of the probability of one class to change, due to changes in another class of the system, responding to $RQ_{3.a.i.A}$ (*Propose a metric to assess Instability at source-code level*). The proposed metric is theoretically validated and empirically compared to existing coupling metrics ($RQ_{3.a.i.B}$: *Is REM a valid class instability metric*?). Next, in $RQ_{3.a.ii}$ (*How to assess Change Proneness at the Source-Code level*?), REM is used as a parameter for defining the Class Change

Proneness Measure (CCPM) ($RQ_{3.a.ii.A}$: *Propose a metric to assess Change Proneness at source-code level*). The proposed metric is empirically compared to existing coupling metrics ($RQ_{3.a.ii.B}$: *Is CCPM a valid class change proneness metric*?).

Second, in $RQ_{3.b}$ (*How to assess Change Proneness at the architecture level*?), we propose the Module Change Proneness Measure (MCPM) to assess the change proneness of architectural modules ($RQ_{3.b.i}$: *Propose a metric to assess Change Proneness at architecture level*). The proposed metric is empirically compared to existing architecture metrics ($RQ_{3.b.ii}$: *Is MCPM a valid module change proneness metric*?).

Third and final, in $RQ_{3.c}$ (*How to assess Change Proneness at the requirements level*?), we proposed the Requirements Ripple Effect Metric (R2EM), which can be used as an indicator of test case prioritization ($RQ_{3.c.i}$: *Propose a metric to assess Change Proneness at requirements level*). The proposed metric is empirically evaluated in an industrial setting using the expert opinion of software engineers ($RQ_{3.c.ii}$: *Is R2EM a valid requirements change proneness metric*?).

### 1.3.4 Using Empiricism to Answer Knowledge Questions

Empirical Software Engineering (ESE) research focuses on the application of empirical studies on any phase of the software development lifecycle. As empirical, we characterize research methods that use experiences and/or observations for retrieving evidence from a real-world context or an artificial setting suitable for investigating a phenomenon of interest (Tichy and Padberg, 2007). Empiricism is considered valuable in software engineering research and practice, because of the plethora of available software engineering methods and tools that can be used for treating the same problem. To this end, an empirical study can for example determine whether claimed differences among alternative software techniques are actually observable (Basili and Selby, 1991). The most common reasons for performing empirical software engineering research are the following (Tichy and Padberg, 2007):

- search for relationships between different variables (e.g., the relation between size of the code to be changed and development effort) by using, e.g., correlation studies;

- use the aforementioned relationships to support decision making mechanisms (e.g., cost estimates, time estimates, reliability estimates) by using prediction and optimization models;
- test hypotheses (e.g., whether development time is saved or quality improved by using inspections, design patterns, or extreme programming) by using experiments.

In this thesis, we have used predominantly the case study method. Case studies are used for monitoring real-life projects, activities or assignments. In case study research, usually different data collection methods are used. The goal is to seek convergence of evidence (from multiple, complementary data sources), a process that is often called triangulation. The case study is normally aimed at tracking a specific attribute or establishing relationships between different attributes (Wohlin et al., 2012). Regarding data collection, we used three methods (Lethbridge et al., 2005):

- ***Analysis of Work Artifacts*** is based on the observation of outputs or by-products of software engineers' work. Common examples of such work outputs (i.e., artifacts) are source-code, documentation, and reports, whereas by-products are defined as outputs created along software development (e.g., feature requests, change logs, etc.). A main advantage of analysis of work artifacts technique is that it requires minimal time or commitment from the study participants (usually software engineers). On the other hand, the collected data might be outdated, in the sense that they might relate to systems or processes that have been significantly changed. Due to the above, this technique should be supplemented by other techniques to achieve research goals.
- ***Interviews & Questionnaires*** are performed through asking a series of questions. Questions can be closed-ended, i.e., multiple-choice such as yes/no or true/false, or they can be open-ended, i.e., conversational responses. Open-ended questions leave the answer entirely up to the respondent and therefore provide a greater range of responses. To implement interviews and questionnaires effectively, questions and forms must be crafted carefully to ensure that the data collected is meaningful (DeVaus, 1996). In order to produce good statistical results from interviews or a questionnaire, a sample must be chosen that is representative of the population of inter-

est. One advantage of these methods is that people are familiar with answering questions, either verbally or on paper, and as a result they tend to be comfortable and familiar with this data collection method. However, interviews and questionnaires rely on respondents self-reporting their behaviors or attitudes.

- In ***Brainstorming***, several people get together and focus on a particular issue. The idea is the group of people tries to find a solution for a specific problem by gathering a list of ideas spontaneously contributed by its members. It works best with a moderator because the moderator can motivate the group and keep it focused. Furthermore, the best way to work this method, is a simple trigger question to be answered and everybody is given the chance to contribute whatever comes to their mind, initially on paper. ***Focus Groups*** are similar to brainstorming. However, a focus group is a group discussion on a particular topic (not just generate ideas). It uses moderators to focus the group discussion and make sure that everyone has an opportunity to participate. One advantage of these methods is that they are excellent data collection methods to use when one is new to a domain and looking for ideas for further exploration. However, if the moderator is not very well trained, brainstorming and focus groups will become too unfocused.

Regarding subject selection, in the majority of our case studies we have used a wide variety of open-source projects. The use of OSS projects enabled us to develop large datasets that could not have been obtained using closed-source. More details on the selection of OSS projects are provided in the corresponding case study designs (e.g., see Chapter 3.5.1). In cases when the data collection was meant to include experts' opinion, we referred to industries that were interested in our projects and involved experienced software engineers as subjects (e.g., see Chapter 7).

During the last years and mainly due to the rise of the Evidence-Based Software Engineering (EBSE) Paradigm (Kitchenham et al., 2004), another type of empirical research has become extremely popular, namely **Secondary Studies**. Secondary studies can be further classified into two major types:

- ***Systematic Literature Reviews:*** Systematic Literature Reviews (SLRs) use data from previously published studies for the purpose of research syn-

thesis, which is the collective term for a family of methods for summarizing, integrating and, where possible, combining the findings of different studies on a topic or research question. Such synthesis can also identify crucial areas and questions that have not been addressed adequately with past empirical research. It is built upon the observation that no matter how well-designed and executed, empirical findings from single studies are limited in the extent to which they may be generalised (Kitchenham et al., 2009).

- ***Systematic Mapping Studies:*** Mapping studies use the same basic methodology as SLRs but aim to identify and classify all research related to a broad software engineering topic rather than answering questions about the relative merits of competing technologies that conventional SLRs address. They are intended to provide an overview of a topic area and identify whether there are sub-topics with sufficient primary studies to conduct conventional SLRs and also to identify sub-topics where more primary studies are needed (Kitchenham et al., 2011).

For the purpose of this thesis, the systematic mapping study approach has been employed. An overview of the empirical research methods that were used for answering each knowledge question is provided in Table 1.3.4.a.

**Table 1.3.4.a: Empirical methods used to answer the knowledge questions**

| Code | Knowledge Question | Empirical Method | Data Collection | Subject | Described in |
|---|---|---|---|---|---|
| RQ1.a | Which design-time quality attributes should be considered in a software development project? | Mapping Study | Manual Inspection | Existing Literature | Chapter 2.3 |
| RQ1.b | Which metrics can be used for assessing/quantifying design-time quality attributes? | | | | |
| RQ2.a | Are maintainability prediction metrics able to capture fine- or coarse-grained changes that are expected to occur in different development phases? | Case Study | Artifact Analysis | Open-source | Chapter 3.5.1 |

| Code | Knowledge Question | Empirical Method | Data Collection | Subject | Described in |
|---|---|---|---|---|---|
| RQ2.b | Can different aggregation functions lead to differences in the way maintainability prediction metrics are able to capture fine- or coarse-grained changes? | Case Study | Artifact Analysis | Open-source | Chapter 3.5.1 |
| RQ2.d | Is SMF a valid metric property? | Case Study | Questionnaires | Practitioners | Chapter 3.6.1 |
| RQ3.a.i.B | Is REM a valid class instability metric? | Case Study | Artifact Analysis | Open-source | Chapter 4.6.1 |
| RQ3.a.ii.B | Is CCPM a valid class change proneness metric? | Case Study | Artifact Analysis | Open-source | Chapter 5.4 |
| RQ3.b.ii | Is MCPM a valid module change proneness metric? | Case Study | Artifact Analysis | Open-source | Chapter 6.4 |
| RQ3.c.ii | Is R2EM a valid requirements change proneness metric? | Case Study | Interviews Questionnaires Focus Group | Practitioners | Chapter 7.4 |

### 1.3.5 Overview of the Dissertation

The main body of this dissertation contains six chapters. Table 1.3.5.a presents the research questions and the chapters, in which they are addressed.

**Table 1.3.5.a: Overview**

| Research Question | Chapter |
|---|---|
| RQ1: Which are the most important design-time quality attributes, and how can they be measured? | Chapter 2 |
| RQ2: Are maintainability prediction source-code metrics applicable at the architecture phase? | Chapter 3 |
| RQ3.a.i: How to assess Instability at the source-code level? | Chapter 4 |
| RQ3.a.ii: How to assess Change Proneness at the source-code level? | Chapter 5 |
| RQ3.b: How to assess Change Proneness at the architecture level? | Chapter 6 |
| RQ3.c: How to assess Change Proneness at the requirements level? | Chapter 7 |

Chapters 2 to 7 are based on scientific journal or conference articles, five of them published, and one currently under review. In all the publications, the

PhD student was the first author and main contributor; other authors include the 3 supervisors as well as industrial collaborators. In the following, each chapter is briefly outlined:

- Chapter 2 is based on a paper published in the Journal of Systems and Software (JSS) (Arvanitou et al., 2017a). This study provides an overview of the literature on design-time quality attributes and the corresponding metrics. The paper was selected to be presented as ***Journal First*** in the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18). JSS is one of the top venues in the software engineering community, whereas SANER is among the top-2 venues in the software maintenance community.
- Chapter 3 is based on a paper published in Information and Software Technology (IST) (Arvanitou et al., 2016). The study proposes and evaluates a method for assessing metrics' fluctuation, through a case study conducted with students and Open-Source Software projects. IST is one of the top venues in the software engineering community.
- Chapter 4 is based on a paper published in the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM' 15) (Arvanitou et al., 2015). In this study we proposed and theoretically and empirically evaluated a metric that can be used to assess the probability of a random change occurring in one class, to propagate to another. ESEM is the top conference of the empirical software engineering community.
- Chapter 5 is based on a paper published in the 21st International Symposium on Evaluation and Assessment in Software Engineering (EASE' 17) (Arvanitou et al., 2017b). In this study we proposed and evaluated a method for assessing the change proneness of classes, through a case study performed with five open-source projects. The paper was awarded the ***Best Full-Paper Award*** for the Conference.
- Chapter 6 is based on a paper published in the 1st International Workshop on Emerging Trends in Software Design and Architecture (WETSODA' 17) (Arvanitou et al., 2017c). This study proposes and evaluates a method for assessing the change proneness of architectural modules. To validate the proposed method, we performed a case study on five open-source projects.

- Chapter 7, is based on a paper currently submitted to the IEEE Transactions on Software Engineering (TSE) (Arvanitou et al., 2018). The paper proposed and evaluates a method for assessing the probability of one requirement to be affected by a change in another requirement as an indicator of its priority to be tested.