Chapter 2

The Nature of Software Evolution

Gabriele Taentzer, Michael Goedicke, Barbara Paech, Kurt Schneider, Andy Schürr, Birgit Vogel-Heuser

In this chapter, we consider the nature of software evolution: What kinds of software systems are evolved? Which quality aspects of software systems play a role throughout evolution? What kinds of software changes exist and which evolution processes are considered? What are the respective change impacts? The purpose of this chapter is to clarify the fundamental aspects of software evolution being taken up again in the following chapters. Hence, this chapter shall provide a basic terminology for this book. To a small extent, it also provides a domain analysis of the area of software evolution. And finally, for more details, further scenarios, and examples of fundamental aspects of software evolution, the reader can find references to follow-up chapters. By this way, this chapter helps to identify how the contributions of subsequent chapters fit into the big picture of software evolution.

2.1 Introduction

The main purpose of this chapter is to present a conceptual basis for core aspects of software evolution. Evolution is a natural phenomenon in the life cycle of software systems according to diverse reasons for change. Software evolution occurs in incremental development where large systems are achieved in small steps, and as reaction to changes in the environment, purpose, or use of the considered software system. We clarify the core aspects of evolution processes. Changes of a software system may have impact on its quality referring to aspects such as correctness, consistency, usability, and maintainability. Evolving software shall preserve or even improve its quality (defined in the ISO standard 25000 on software product quality [**ISO**]) throughout software changes. Our considerations of the nature of software evolution are largely independent of application domains for software systems. Throughout the book, however, two application domains are focused, namely business information systems [**heinrich2015cocome**] and product automation [**legat2013evolution**].

2.2 Software Systems

As a conceptual basis, we consider fundamental aspects of software systems. *Application domains* and *system scopes* set the environments of software systems; *artefacts* and potential software *variants* refer to the ingredients of software systems or even software product lines [**clements2002software**].

2.2.1 Application domains

An application domain for software systems is a problem field being characterized by common requirements, terminology, processes, and functionality for software systems. Throughout this book, various application domains for software systems are considered. They are mostly considered from a rather technical point of view. In particular, two domains, business and product automation, occur very prominently in the subsequent chapters due to our case studies Common Component Modeling Example (CoCoME) and extended Pick and Place Unit (xPPU). They are introduced in Chapter 4.

2.2.2 Scopes and environments of software systems

A *software system* is a set of coherent components that provides services (or features) to users. A software system needs a *platform* to run consisting of hardware and further software components such as operating systems, libraries, and special software components provided by the environment. The hardware comprises computers of any kind in the first place but also networks of computers (especially the Internet). Dependent on the domain, additional hardware may come into play, such as mechanical and electrical components. The *scope of a software system* defines a range of items that can be shaped and designed when developing software systems [**CPREGlossary**]. Besides the code for the system itself it comprises e.g. the system requirement specification, any kinds of system documentation, models, data sets, and test suites.

The *environment* of a system does not only contain the platform for running the system but also any other part being relevant for the software system and its scope such as users on which the system has impact to and regulations that shall be obeyed. An explicit consideration of the environment is important when it comes to evolution since various kinds of environment changes can occur such as new versions of the underlying operating system or programming language, related software components, external regulations that shall be obeyed by the software, and many more.

Two interesting examples of software systems are the following: In Chapter 5 and Chapter 6, the authors investigate the evolution of *socio-technical systems* where developers and/or users are explicitly considered within the system scope. The interrelation of social and technical aspects and their joint optimization are of special relevance here. A very different form of system are mechatronic systems such as automated Production Systems (aPS) which consider the interplay of mechanics, electronics, and software (in Chapter 10 and Chapter 8).

2.2.3 Software artefacts

Software development and software changes usually involve a number of *software artefacts.* Even the kinds of software artefacts are manifold: Analysts elicit requirements and write requirement specifications which may comprise analysis models. Software architects take these specifications into account to develop the design of a software system, often by constructing design models. Software engineers and programmers develop models and write code which are structured in various files and directories. Moreover, they write test cases and documentations being organized in additional file structures. Once a software system is deployed, it may produce even further artefacts for, e.g. reporting about continuously running processes. The system behaviour at runtime and its ad-hoc changes, for example, are considered in Chapter 10. To summarize, there are usually a vast number of artefacts of various kinds in the scope of a software system.

Software artefacts are usually not isolated but inter-related. Hence we have to take care of consistency relationships between them and we have to maintain them throughout software evolution. For example, the evolution of requirement specifications, design decisions (comprising design knowledge about problems, solution, context and rationale), and architecture specification models is in the focus of Chapter 6.

In software engineering, there are quite a number of languages used to create software artefacts. Besides programming languages such as Java and C, there are various modelling languages as the Unified Modeling Language (UML) and Matlab/Simulink. Documentations are usually semi-structured natural text, often written in HTML, LaTeX or Word. Moreover, there are domain-specific languages, especially for specific modelling purposes, such as AutoFocus [**legat2014interface**; **rosch2015model**; **teufl2015efficient**] for embedded system development presented in Chapter 11 and variability-modelling languages like decision models [**schmid2011comparison**], orthogonal variability models [**pohl2005software**], and feature models [**kang1990featureoriented**] as considered in Chapter 7 and Chapter 8. Furthermore, there exist specific languages to describe the syntax and semantics of modelling languages as, e.g. EMF [**EMF**], and to specify differences resp. transformations between models to formally express their evolution as, e.g. Henshin [**henshin**]. They are used in Chapter 10 to understand historical evolutions between different versions of models as well as to recommend future evolutions based on these historic evolutions.

2.2.4 Software variants

Most modern software systems occur in several *variants*. If a software has several locations where the resulting system can be configured, there is already some form of variability supported. Each configuration forms a variant of the software system. They may be used to tailor a software system to different platforms, contexts and users' needs. A collection of software variants that share common artefacts that are commonly processed, is called a *software product line*. A software variant is called *product* in this context. Variants of a software system can occur independently of any time periods while chronologically changed software is usually called a *version*. Version management is specifically considered in Section 2.4.

In Chapter 7, statechart models are presented which are able to integrate all product-variant behaviour into one model. A feature model serves as configuration specification; the product line is implemented by preprocessor-based C-code. Similarly, in Chapter 8, software variants are explicitly considered for evaluation of performance. In particular, strategies for performance evolution are discussed for variants co-existing at the same time and versions that are the result of software evolution.

2.3 Software Quality

Preserving and improving software quality are often the main drivers of software evolution, such as improving the performance of a software solution. The ISO 25000 standard [**ISO**] defines software quality based on a number of aspects covering functional and non-functional ones. *Functional software quality* refers to the extent the software conforms to a given functional requirement specification. Aspects of functional software quality are e.g. correctness, consistency, dependability, and usability. *Non-functional software quality* tells us how well a software system meets non-functional requirements concerning, e.g. performance (cf. Chapter 8, Chapter 10), maintainability, and security aspects (cf. Chapter 9). In the following, we recall the main quality aspects of software systems and point to examples.

2.3.1 Consistency

As there may be various artefacts in the scope of a software system, an immediate question is: *Do the various software artefacts belonging to the scope of a software system are consistent with each other?* Artefact relations may be purely *syntactical* such as models conforming to their meta-models. Software artefacts may also be related w.r.t. behaviour. The most prominent shape of behaviour consistency is *behavioural equivalence* (also known as bi-simulation). Weaker notions of behavioural equivalence like conditional and relational equivalence are introduced as consistency notions in Chapter 11. Besides this *outer consistency* being established in between several artefacts, there is also an *inner consistency* considering the content of just one artefact. Here, consistency means that *an artefact does not contradict itself* [**easterbrook1996using**]. Inner consistency comprises, for example, the internal consistency of requirements within one requirement specification or the declaration of a variable before its use in a program.

Even if artefacts are consistent on creation, changes to one software artefact may not necessarily be reflected immediately in all related artefacts that are affected by the same modification. This means that the quality aspect of consistency is endangered by changes. If changes are made in one place, consistency may call for changes in several other artefacts. There is the resulting challenge of keeping systems consistent over time. As consistency cannot be always (re)established easily, there is also the general need for *inconsistency management*. Intermediate inconsistency gives developers the flexibility and the freedom to postpone the re-establishing of consistency for increasing productivity. If explicit relationships between artefacts, i.e. traces, are considered, a form of traceability link management is needed here [**feldmann2016comprehensive**]. Traceability is explicitly considered in the context of identifying and extracting tacit knowledge in software evolution (Chapter 5, Chapter 10) and continuous design support (Chapter 6) caring about the consistency between architecture and code.

Inconsistency may also affect system variants. A necessary condition for software product lines (defining software variants) is often the following: If a feature model is available for the system, it is typically assumed to be consistent, i.e. there must be at least one valid combination of features (Chapter 7 and Chapter 8). In Chapter 8, performance is only measured for configurations that are valid according to the feature model. The product implementation derivable from valid combinations of features must also be consistent with further development artefacts such as quality-assurance artefacts. If product implementations evolve, for instance, corresponding test suites must be updated accordingly (Chapter 7).

2.3.2 Correctness

To validate correctness of a software system we should ask: *Does the system do what I want it to do?* This question shows that correctness relates to the system's functional requirement specification. A software system is considered correct w.r.t. its requirement specification if it behaves as specified by its requirements. Hence, correctness can also be considered as a kind of consistency, here of code (and other artefacts) with the requirement specification. As correctness is such a central consistency aspect of software systems, it is usually considered explicitly. In most software projects, functional requirements are validated by testing a software. In contrast to validating system functionalities, there is also the possibility to verify them formally. Correctness in the presence of evolution plays a central role in Chapter 7 which is concerned with software testing of evolving SPLs, and in Chapter 11 as this chapter is concerned with the formal verification of evolving automated production systems.

2.3.3 Dependability

Dependability comprises quality aspects such as reliability, availability, safety, and security [**avizienis2004dependability**; **littlewood2000software**]. A high dependability allows us to rely on a system functioning as required, even under hampered conditions such as software and hardware faults. The notion of dependability has been discussed very broadly in literature, dependent on different perspectives of various stakeholders. [**febrero2016software**] gives a literature overview.

Considering *reliability*, we ask: *Does the system show correct behaviour all the time or for a specific time period?* Reliability is closely related to *availability* which is typically described as the ability of a component or system to function at a specified moment or interval of time. Reliability is also considered as the probability of success. In addition, dependability comprises *safety* which shows the degree of hazards' prevention that may result from the operation of the system and threatens users or environment [**lauber1999prozessautomatisierung**]. Evolution and safety is discussed in Chapter 10. In contrast, *security* mainly refers to the absence of unauthorized access from users or environment that threaten the operation of the system [**reussner2006toward**; **lauber1999prozessautomatisierung**]. Chapter 9 is mainly concerned with maintaining security in the presence of software evolution.

2.3.4 Performance

Performance of a software system is considered by asking questions like: *Does the system perform the indicated behaviour as fast as required?* Performance engineering comprises all kinds of optimizing the timing behaviour and resource consumption of a software system as well as guaranteeing available or specified resource limitations. Considered aspects are e.g. the througput, latency, memory usage, and energy consumption of software systems. Performance for the specific case of automated production systems is discussed in Chapter 10. Performance issues can result in loss of productivity for the user. When software engineers start improving the systems's performance, corresponding evolution steps may lead to cost overruns due to tuning or redesign. Moreover, it is likely that tuning may disrupt the original software architecture or its behaviour.

Considering a software system with variants, there are often variants with better or worse performance. Here, checking performance refers to the accessible computation effort and resulting impact on resource usage and timeliness of a system variant (Chapter 8).

2.3.5 Usability

Users expect a software that is easy to learn as well as pleasant and efficient to use. Moreover, they appreciate a system which easily recovers from usage errors and whose usage can be easily memorized after some period of not using it. To check usability, the degree to which a software system can be used by specified consumers should be investigated to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a determined context of use. In Chapter 5, usability is a major aspect in the sense that expectations or assumptions about the usability and the functionality of a system are derived from the users' behaviour.

2.3.6 Maintainability

A software system is well maintainable if it can be easily changed with respect to its environment to e.g. correct defects, realize new requirements, or adapt the system to a changed platform. Specific aspects of maintainability are testability, analyzability, and changeability. A software is well testable if its artefacts support testing in given test contexts. Often, testability is a question of good software design featuring strong cohesion and loose coupling. Testability of variant-rich software systems is a key aspect in Chapter 7 as it is concerned with model-based testing of evolving SPLs. A software system is considered analysable if system parts causing deficiencies or failures of the system can be easily identified. In Chapter 11, analysable models are considered to bridge the conceptual gap between requirements and target system implementations. And centrally for software evolution, software shall be easily changeable to be adaptable to continuously occurring changes in the environment with considerable effort.

2.4 Software Evolution

Software system changes show a wide variety which has been investigated and classified in the literature such as [**lientz1980software**; **chapin2001types**; **buckley2005towards**]. In [**buckley2005towards**] the authors present a taxonomy for software evolution distinguishing four different dimensions of system change: They consider temporal properties (i.e. when do changes happen), objects of change (i.e. ,where in the system do we make changes), system properties (i.e. what is changed), and change support (i.e. how is it changed). They do not consider who is doing system changes and why; this has already been done before in [**chapin2001types**]. This split-up of dimensions is driven by the basic idea that activities and processes form the core of software engineering methods. The purpose of taxonomies as the ones found on software evolution is, among others, to provide a framework for comparing and combining individual tools and techniques and to provide an overview of the research domain of software evolution. We take it up in this section: Considering different *kinds of software change* in the following, we will focus on reasons for change as well as participating artefacts and users, i.e. the *why* and *what*. Thereafter we consider *evolution processes* where temporal properties, change support, and stakeholders, i.e. the *when*, *how*, and *who*, are focused. Finally, configuration management is considered to capture all changes of software artefacts that emerge throughout evolution.

2.4.1 Kinds of software change

Software changes have been studied since a long time; comprehensive works in this direction are [**lientz1980software**; **chapin2001types**] where types of software evolution are classified along the kind of artefacts changed as well as the reason for change. The authors focus on code versus documentation changes; reasons for change are functionality changes, adaptations to the environment as well as performance and maintenance issues. Documentation comprises all kinds of software artefacts except of the code.

Early works such as [**lientz1980software**] and the ISO/IEC standard for software maintenance [**ISOM**] propose to distinguish software changes into *corrective*, *adaptive*, *perfective*, and *preventive* modifications.

- Corrective modifications subsume all kinds of *bug fixing* to eliminate system failures and *feature requests* as long as they reflect corrected requirements.
- Adaptive modifications refer to *changes of system environments* as well as additional *requirement elicitation*. More recently, studies of *adaptive systems* have led to further kinds of evolution activities being runtime adaptations, i.e. system modifications at runtime [**lemos2013software**].
- Perfective modifications subsume all kinds of system improvements such as *performance optimization*, *structure re-engineering or optimization* (refactoring), and all kinds of documentation activities especially *knowledge extraction* from the software system.
- Preventive modifications summarize all changes which prevent problems from software systems before they occur.

Software changes may take place continuously, as planned or ad-hoc changes.

Throughout this book, various kinds of system changes are presented: Chapter 4 discusses a variety of concrete evolution steps as they occur in the case studies. Chapter 5 is concerned with detecting and reducing mismatches between stakeholder's mental models during software evolution. The basic problem is that the system may gradually diverge from a given specification or customer demand. This deviation may come from incomplete implementation of requirements - or from changing requirements that are not complemented by a corresponding change of the system. Such deviations shall be reduced. In Chapter 6 continuous software engineering is considered being a special kind of software evolution. Chapter 7 discusses implementation changes and corresponding updates of quality-assurance artefacts in software product lines such that consistency is preserved. Maintaining performance as a prerequisite for evolving software artefacts is considered in Chapter 8. Analysis strategies are presented which can efficiently assess and predict the system's performance. On this basis, performance improvements over time are considered. Moreover, software variants with the best performance are identified. Chapter 9 is dedicate to maintaining security throughout changing requirements and changing environments such that changes do not affect the system's level of security. The maintenance of safety is addressed in Chapter 10. Capturing and transferring knowledge to next software versions and projects are addressed in Chapter 6 and Chapter 10, in particular ad-hoc changes with respect to learning are presented in Chapter 10. To be able to distinguish wanted from unwanted system changes, the maintenance of correctness is considered in Chapter 11 by applying formal verification techniques to show the correctness of evolving software systems. Newer revisions of the software must not violate existing software properties and should comply with even more.

Change may also take place during runtime; A knowledge elicitation technique, well known in software engineering, is the Post-Mortem Analysis (PMA) [**stalhane2003post**]. PMA of a system's runtime behaviour simply consists of gathering knowledge about a process and to analyse it in order to improve the next runs of this process in future. An example application of PMA is presented in Chapter 10.

2.4.2 Evolution processes

Several iterative and incremental software development processes have been proposed such as the Unified Process [**kruchten2003rational**], V-ModelXT [**vogel2015evolution**], and agile software development [**beck1999embracing**]. *Agile software development* processes already acknowledge and embrace change as an essential fact of life. One of the agile development principles is to welcome changing requirements, even in late development. Furthermore, software development shall be sustainable and software quality shall remain high. How do software development processes actually incorporate evolution? According to [**mens2008software**] (referring to [**lehman2003software**]), the *software evolution process* is a multi-loop, multi-level, multi-agent feedback system that cannot be treated in isolation. A specific form of evolution process is *round-trip engineering* (also called *horse shoe process* [**kazman1998requirements**]) where developers alternate between models and code. This process consists of three phases: The *reverse engineering* phase is needed to understand the structure and behaviour of a larger part of legacy code by means of models. In the subsequent *restructuring* phase (a part of) the software is redesigned on the level of models, and finally, *forward engineering* is needed to implement the new design and to integrate it into the existing system.

While the clear separation between development and maintenance has already dissolved in agile software development, this is even more the case in *continuous engineering* [**Fitzgerald2017**]. Continuous activities are meant to eliminate discontinuities which occur from following development activities in a specific order. Continuous engineering specifically includes continuous improvement and innovation. An early proposed activity which can be considered as continuous innovation activity is that of beta testing, which became a widespread practice, even in industrial software development. It is used to elicit early customer feedback prior to formal release of software products [**cole2002from**]. Following the trend of continuous engineering, software engineers have commonly accepted that software must continually evolve according to changes. Otherwise, the software does not fulfil its ever changing requirements and therefore, will become outdated earlier than expected.

Change impact analysis techniques can identify system parts that are likely to be affected by additional changes. These techniques support knowledge elicitation from change histories to inform all interested stakeholders. On this basis, these techniques can also give an estimation of how costly an intended change will be and how risky it is to make that change. This analysis is used to decide whether it is worthwhile to carry out that change. The risk has a strong relation to software quality. If proper support for measuring quality is available, a measurement report can provide crucial information to determine whether the software quality is degrading, and to take corrective actions if this turns out to be the case.

Throughout the book the following aspects of evolution processes are tackled: The roundtrip model is used in Chapter 6, Chapter 7, Chapter 8, and Chapter 10. In Chapter 6, we show that the small iterations of continuous engineering support light-weight design decision capture and use. Chapter 7 considers an SPL evolution scenario covering a complete family of software products to be evolved. The efficient performance analysis of software variants and versions based on monitoring and model extraction is focused in Chapter 8. The dynamic nature of running self-adaptive systems and their environments requires continuous validation and verification to assess the system at run-time, which was traditionally done at development-time and which requires new and efficient techniques for the run-time case [**lemos2013software**]. An example for evolving self-adaptive systems is given in Chapter 10. In Chapter 11, regression verification is applied to evolving systems again based on a roundtrip model.

2.4.3 Configuration management

To capture all changes throughout software evolution, emerging changes of software artefacts are usually managed with the help of development tools. *Change management* refers to a systematic consideration of change requests which may be bug reports and feature requests. To ensure that the most urgent and cost efficient change requests are prioritized, each request is collected and assessed first and addressed along its priority thereafter. Especially for software product lines where versions of variants may occur a systematic management of change requests is necessary.

Version management is needed to store and track emerging versions of software artefacts. Moreover, it allows developers to work on these versions concurrently in a coordinated way. To save memory, subsequent versions may be stored in a list of *deltas*. A delta just stores the differences of one version to its successor. Applying these deltas to a root version (usually the newest one), the other versions can be computed. Several developers are allowed to work on the same artefacts concurrently. The version management system tracks the edited artefacts, ensures that changes to one and the same artefact do not get lost, and supports the resolution of conflicting changes. To allow developers working in isolation, the artefacts in the scope of a software system may be duplicated into several *branches* (of the version tree). The ability of branching implies the later facility to merge changes back onto one branch. The usage of branching in the context of continuous integration is tackled in Chapter 6.

Release management is the process by which source code is converted to a final software product, often being built for a specific environment. Version management is usually involved and is recommended but is not a requirement. A reliable release process is as much automatic as possible and supports a quick and frequent deployment, a prerequisite for *continuous integration*. Recently, continuous integration has emerged as a practice to eliminate discontinuities between development and deployment. However, continuous integration is not yet used in aPS (only 33% of companies use it to some extent and 15% by default) [**bougouffa2017scalable**]. In a similar vein, the recent emphasis on *DevOps* recognizes that the integration between software development and its operational deployment needs to be a continuous one [**Fitzgerald2017**]. The concept of continuous deployment, i.e. the ability to deliver software more frequently to customers, enables frequent customer feedback which has become very attractive to companies, in the area of production automation, however, it is often not implemented due to confidentiality.

As configuration management may becomes very complex for software product lines, this problem is subject in Chapter 7 and Chapter 10. Each variant, or in more detail each feature, can occur in various versions which have to be integrated in a consistent way. In Chapter 10, the evolution of variants is considered focusing specifically on the continuous correctness of the system.