Chapter 4
4.0 Taxonomy of Software Evolution

4.1 Introduction

Evolution is critical in the life cycle of all software systems particularly those serving highly volatile business domains such as banking, e-commerce and telecommunications. An increasing number of evolution mechanisms and tools are becoming available and many systems are being built with some change support in place [32]. Because of this, there is a need for a common vocabulary and conceptual framework to categorize and compare the evolution support offered by the various tools and techniques.

More than two decades ago, Lientz and Swanson [33] proposed a mutually exclusive and exhaustive software maintenance typology that distinguishes between perfective, adaptive and corrective maintenance activities. This typology was further refined into an evidence-based classification of 12 different types of software evolution and software maintenance: evaluative, consultive, training, updative, reformative, adaptive, performance, preventive, groomative, enhancive, corrective and reductive. This work is very important and relevant, in that it categorizes software maintenance and evolution activities on the basis of their purpose (i.e., the why of software changes).

In this section, we will take a complementary view of the domain, by focusing more on the technical aspects, i.e., the when?, where?, what? and how? of software changes. These questions are used as a basis to propose a taxonomy of the characteristics of software change mechanisms and the factors that influence these mechanisms. By change taxonomy we mean “A system for naming and organizing things . . . into groups which share similar qualities” [40]. With change
mechanisms we refer to the software tools used to achieve software evolution and the algorithms underlying these tools (although it is intended that this taxonomy should be extended to consider the formalisms used and the methods employed to carry out software evolution).

The purpose of this taxonomy is manifold:

4.1.1 To position concrete software evolution tools and techniques within this domain;

4.1.2 To provide a framework for comparing and combining individual tools and techniques;

4.1.3 To evaluate the potential use of a software evolution tool or technique for a particular maintenance or change context and thus;

4.1.4 To provide an overview of the research domain of software evolution. Each of these purposes is essential, given the proliferation of tools and techniques within the research field of software evolution.

![Diagram of Dimensions of Software Evolution]

**Figure 4.1. Dimensions of Software Evolution**
4.2 Taxonomy

4.2.1 Temporal Properties (When?)

The when question addresses temporal properties such as when a change should be made, and which mechanisms are needed to support this.

4.2.1.1 Time of Change

Depending on the programming language, or the development environment being used, it is possible to distinguish between different phases of the software life-cycle, such as compile-time, load-time, run-time, and so on. These phases have been indirectly used as a basis for categorizing software evolution tools. Using these phases, at least three categories become apparent, based on when the change specified is incorporated into the software system. Particularly:

4.2.1.1.1 Compile-time. The software change concerns the source code of the system. Consequently, the software needs to be recompiled for the changes to become available.

4.2.1.1.2 Load-time. The software change occurs while software elements are loaded into a executable system.

4.2.1.1.3 Run-time. The software change occurs during execution of the software.

The traditional approach to software maintenance, where the programmer edits or extends the source code of a software system, and re-compiles (possible incrementally) the changes into a new executable system, is compile-time evolution. Here, typically, a running software system has to be
shut down and restarted for the change to become effective. Instead of compile-time evolution, one often uses the term static evolution.

In contrast, run-time evolution considers the case where the changes are made at runtime. Here, systems evolve dynamically for instance by hot-swapping existing components or by integrating newly developed components without the need for stopping the system. Run-time evolution has to be either planned ahead explicitly in the system or the underlying platform has to provide means to effectuate software changes dynamically. Run-time evolution is often also called dynamic evolution.

Load-time evolution sits between these two extremes. It refers to changes that are incorporated as software elements become loaded into an executable system. In general, load-time evolution does not require access to the source code, but instead applies changes directly to the binaries of a system. Load-time evolution is especially well-suited for adapting statically compiled components dynamically on demand, so that they fit into a particular deployment context. The most prominent example for a load-time evolution mechanism is Java'sClassLoader architecture. It is based on classfile modifications on the byte-code level. Depending on whether load-time coincides with run-time (like in Java) or it coincides with a system's startup-time, load-time evolution is either static or dynamic. Obviously, the time of change heavily influences the kind of change mechanism needed. For example, systems that allow dynamic evolution must ensure, at runtime, that the system's integrity is preserved and that there is an appropriate level of control over the change. Otherwise, when the changes are implemented, the running system will crash or behave erroneously.
4.2.1.2 Change History

The change history of a software system refers to the history of all (sequential or parallel) changes that have been made to the software [5]. Tools that make this change history explicitly available are called version control tools, and are used for a wide variety of purposes. One can distinguish between mechanisms that support versioning and those that do not provide means to distinguish new from old versions. In completely unversioned systems, changes are applied destructively so that new versions of a component override old ones. In this scenario, old versions get lost in the evolution process. In systems that support versioning statically, new and old versions can physically coexist at compile-time or load-time, but they are identified at run-time and therefore cannot be used simultaneously in the same context.

In contrast to this, fully versioned systems do not only distinguish versions statically, they also distinguish versions at runtime, allowing two different versions of one component being deployed simultaneously side by side. This is particularly relevant for the dynamic evolution of systems. Here, safe updates of existing components often require that new clients of the component use the new version whereas existing clients of the old component continue to use the old one. In such a context, two versions of a component coexist until the old version reaches a quiescent state which allows the safe removal of the old version.
Such versioning mechanisms have been extensively used for schema evolution in object-oriented databases in order to support forward and backward compatibility of applications with schemas and existing objects. We will now classify the different kinds of versioning, and the mechanisms needed to support them. Software changes may be carried out sequentially or in parallel (see Fig. 4.2). With sequential software evolution, multiple persons cannot make changes to the same data at the same time. To enforce this constraint, we need some form of concurrency control (e.g., a locking or transaction mechanism) to avoid simultaneous access to the same data. With parallel evolution, multiple persons can make changes to the same data at the same time. Parallel evolution is needed when different software developers simultaneously make changes to the same software component.
Within parallel evolution, one can further distinguish between synchronous changes and asynchronous changes (see Fig. 4.2). In the synchronous case, the same data is shared by all persons. This is often the case in computer-supported collaborative work. It requires mechanisms such as a shared work surface, a permanent network connection to the server where the data resides, etc.

In the asynchronous case, all persons that change the data in parallel work on a different copy. Because the data is not shared anymore, we can have convergent changes or divergent changes (see Fig. 4.3). With convergent changes, parallel versions can be merged or integrated together into a new combined version. For divergent changes, different versions of the system co-exist indefinitely as part of the maintenance process. This is, for example, the case in framework-based software development, where different invasive customizations of an application framework, i.e. destructive changes directly applied to the framework itself by different customers, may evolve independently of one another.

Please note that an asynchronous convergent change process subsumes a completely synchronous process, if one only looks at the final product and not at the creation history, where in the asynchronous case different versions of some data may exist temporarily.

4.2.1.3 Change Frequency

Another important temporal property that influences the change support mechanisms is the frequency of change. Changes to a system may be performed continuously, periodically, or at arbitrary intervals. For example,
in traditional management-information systems (MIS), users frequently request changes but these changes may only be incorporated into the system periodically, during scheduled downtimes. Other systems (for example interpreted systems), allied with less-formal change processes, may allow developers to incorporate changes continuously, as they are required.

The frequency of change is important, because it influences the change mechanisms used. For example, if a system is being changed continuously, the need for fine-grained version control over the system is increased. Otherwise, it would become very difficult to roll-back the system to specific earlier versions, when required.

4.2.2 Object of Change (Where)

The second group in our taxonomy addresses the where question. Where in the software can we make changes, and which supporting mechanisms do we need for this?

4.2.2.1 Artifact

Many kinds of software artifacts can be subject to static changes. These can range from requirements through architecture and design, to source code, documentation and test suites. It can also be a combination of several or all of the above. Obviously, these different kinds of software artifacts influence the kind of change support mechanisms that will be required. Dynamic evolution mechanisms evolve runtime artifacts like modules, objects, functions, etc.
4.2.2.2 Granularity

Another influencing factor on the mechanisms of software change is the granularity of the change. Granularity refers to the scale of the artifacts to be changed and can range from very coarse, through medium, to a very fine degree of granularity. For example, in object-oriented systems, coarse granularity might refer to changes at a system, subsystem or package level, medium granularity might refer to changes at class or object level and fine granularity might refer to changes at variable, method, or statement level. Traditionally, researchers have distinguished only between coarse grained and fine grained artifacts with the boundary specified as being at file level. Anything smaller than a file was considered a fine-grained artifact.

4.2.2.3 Impact

Related to the granularity is the impact of a change. The impact of a change can range from very local to system-wide changes. For instance, renaming a parameter of a procedure would only be a local change (restricted to the procedure definition), while renaming a global variable would have, in the worst case, an impact on the whole source code.

Sometimes, even seemingly local changes in the software may have a global impact because the change is propagated through the rest of the code. The impact of a change can span different layers of abstraction, if we are dealing with artifacts of different kinds. For example, a source code change may require changes to the documentation, the design, the software architecture, and the requirements specification.
4.2.2.4 Change Propagation

To address all the above problems, we need to resort to mechanisms or tools that help with change impact analysis, change propagation, traceability analysis and effort estimation. Change propagation refers to the phenomenon where a change to one part of the software creates a need for changes in other parts of the software system. For example, a change to the implementation may have an overall effect in the source code, but may also impact the design, the documentation and the requirements specification. In this way, a single change to one system part may lead to a propagation of changes to be made throughout the entire software system. Change impact analysis aims to assess or measure the extent of such change propagation. Traceability analysis can help with change impact analysis, since it establishes explicit relationships between two or more products of the software development process. Like impact, the traceability relationship can remain within the same level of abstraction (vertical traceability) or across different levels of abstraction (horizontal traceability). In many cases, changes with a high impact also require a significant effort to make the changes. This effort can be estimated using effort estimation techniques. In some situations the effort can be reduced by automated tools. For example, renaming entities on the source code level is typically a global change with a high change impact, but the corresponding change effort is low because renaming can proceed in an automated way.

4.2.3 System Properties (What)

A logical grouping of factors that influence the kinds of changes allowed as well as the mechanisms needed to support these changes has to do with the
properties of the software system that is being changed, as well as the underlying platform, and the middleware in use.

4.2.3.1 Availability

Most software systems evolve continuously during their lifetime. Availability indicates whether the software system has to be permanently available or not. For most software systems, it is acceptable that the system is stopped occasionally to make changes (e.g., to enhance the functionality) by modifying or extending the source code. Alternatively, some software systems, for instance telephone switches, have to be permanently available. Therefore they cannot be stopped to incorporate changes. Such systems require more dynamic evolution mechanisms such as dynamic loading of component updates and extensions into the running system (run-time evolution).

4.2.3.2 Activeness

The software system can be reactive (changes are driven externally) or proactive (the system autonomously drives changes to itself). Typically, for a system to be proactive, it must contain some monitors that record external and internal state. It must also contain some logic that allows self-change based on the information received from those monitors. A system is reactive if changes must be driven by an external agent, typically using some sort of user interface. In this way, the system can respond to external events initiated by the user.

For a system to be proactive the time of change must be runtime (and hence it must be dynamic software evolution). If this is not the case, then the
system would not be able to detect its own monitors and trigger the specified change on itself.

4.2.3.3 Openness

Software systems are open if they are specifically built to allow for extensions. Open systems usually come with a framework that is supposed to facilitate the inclusion of extensions. While they support unanticipated future extensions (statically or dynamically), it is difficult to come up with a framework that allows for flexible extensions without being too restrictive concerning all possible evolution scenarios.

In general, a system cannot be open to every possible change. Closed systems on the other hand do not provide a framework for possible extensions. Such systems are self contained, having their complete functionality fixed at build time. This does not imply that closed systems are not extensible, simply that they were not specifically designed for it. So it is possible to evolve closed systems, but usually with more effort. Operating systems are probably the most prominent open systems. For these systems, the ability to create and run user programs that extend the functionality of the underlying operating system is essential. A second example of open systems are extensible programming languages. Extensibility in languages is either supported with explicit reflective capabilities (e.g., Smalltalk, Lisp) or with static meta-programming (e.g., OpenJava).

Similarly, some database systems e.g., KIDS, Navajo and SADES support incorporation of new functionality or customization of existing functionality by using component-based and aspect-oriented techniques. An
example of a partially open system is a system that allows for plug-ins at runtime. While the plug-in modules may be unknown in advance, the ability to add them to the system at runtime is explicitly provided. A plug-in based system is not fully open since it exposes limited capacity for "extensions". An open system would allow you to do subtractions and modifications too in a clearly defined framework.

4.2.3.4 Safety

In the context of continuous evolution, safety becomes an essential system property. We distinguish between static and dynamic safety. The system features static safety if we are able to ensure, at compile time, that the evolved system will not behave erroneously at runtime. The system provides dynamic safety if there are built-in provisions for preventing or restricting undesired behavior at runtime. Note that there are many different notions of safety. One of them is security, for example to protect the software from viruses (in the case of dynamic evolution), or to prevent unauthorized access to certain parts of the software or to certain resources. A good example for such a mechanism is Java's concept of security managers, mainly exploited in web browsers for restricting access of dynamically loaded applets to the local machine.

Another is behavioral safety, in the sense that no crashes, unpredictable or meaningless behavior will arise at runtime due to undetected errors. Yet another notion is backward compatibility which guarantees that former versions of a software component can safely be replaced by newer versions without the need for global coherence checks during or after load-time. Directly related to this is the well-known fragile base class problem in class-
based object-oriented programming, where independently developed subclasses of a given base class can be broken whenever the base class evolves.

4.2.3.4.1 The structural variant of this problem is dealt with in IBM's SOM approach, by allowing (in some cases) a base class interface to be modified without needing to recompile clients and derived classes dependent on that class. This is clearly a static form of safety.

4.2.3.4.2 The semantic variant of the problem is more complex and requires a dynamic approach, because the implementation of the base class can be changed as well. This gives rise to the question how a superclass can be safely replaced by a new version while remaining behaviorally compatible with all of its subclasses.

Obviously, the kind and degree of safety that is required has a direct influence on the change support mechanisms that need to be provided. For example, a certain degree of static safety can be achieved by a programming language's type system at compile-time, while dynamic type tests can be used for those cases that are not covered by the static type system. Moreover, systems that support dynamic loading need additional coherence checks at load-time to ensure that new components "fit" the rest of the system.

Such checks are even necessary for systems that guarantee certain aspects of safety statically because of components' separate compilation. As a final example, systems where two versions of a single component can coexist
together during a transition phase not only need dynamic checks to ensure consistency: They also need some form of monitoring which is capable of mediating between the two versions actively. Object database systems, for example, provide mechanisms for adapting instances across historical schema changes.

4.2.4 Change Support (How)

During a software change, various support mechanisms can be provided. These mechanisms help us to analyze, manage, control, implement or measure software changes. The proposed mechanisms can be very diverse: automated solutions, informal techniques, formal representations, process support, and many more. This section describes some orthogonal dimensions that influence these mechanisms or that can be used to classify these mechanisms.

4.2.4.1 Degree of automation

We propose to distinguish between automated, partially automated, and manual change support. In the domain of software re-engineering, numerous attempts have been made to automate, or partially automate, software maintenance tasks. Typically, these are semantics-preserving transformations of the software system. In reality, however, these automated evolutions incorporate some form of manual verification and thus, can only be considered partially automated. Within the specific domain of refactoring (i.e., restructuring of object-oriented source code), tool support also ranges from entirely manual to fully automated. Tools such as the Refactoring Browser support a partially automatic approach while other researchers have demonstrated the feasibility of fully automated tools.
4.2.4.2 **Degree of formality**

A change support mechanism can either be implemented in an ad-hoc way, or based on some underlying mathematical formalism. For example, the formalism of graph rewriting has been used to deal with change propagation and refactoring.

4.2.4.3 **Process support**

Process support is the extent to which activities in the change process are supported by automated tools. As an example of process support, one can consider a refactoring tool as a way to automate the activity of refactoring, which is a crucial part of the extreme programming process. By resorting to such a tool, as opposed to performing the refactorings manually, potential errors can be reduced significantly. The process support dimension is orthogonal to both previous dimensions. First, the degree of automation of a change process can range from fully manual to automatic. Second, we can have a formal change process that relies on an underlying mathematical formalism by resorting to formal methods. Their mathematical basis makes it possible to define and prove notions like consistency, completeness and correctness.

4.2.4.4 **Change type**

The characteristics of the change itself can influence the manner in which that change is performed. Because an extensive typology of software changes was already presented in, we will restrict ourselves here to the distinction between structural and semantic changes only. This distinction is
an important influencing factor on the change support mechanisms that can be defined and used.

Structural changes are changes that alter the structure of the software. In many cases, these changes will alter the software behavior as well. A distinction can be made between addition (adding new elements to the software), subtraction (removing elements from the software), and modification (modifying an existing element in the software, e.g., renaming). Intuitively, it seems likely that addition is better suited to late, runtime preparation than subtraction and alteration. Subtraction and alteration suggest that changes will occur within the existing system, whereas addition suggests that extra functionality can be hooked onto the existing system.

Next to structural changes, a distinction should be made between semantics-modifying and semantics preserving changes. In object-oriented systems, for example, relevant semantic aspects are the type hierarchy, scoping, visibility, accessibility, and overriding relationships, to name a few. In this context, semantics-preserving changes correspond to the well-known concept of software refactoring. In the wider context of re-engineering, semantics-preserving changes are accommodated by restructuring activities, such as the replacement of a for loop by a while loop, or the removal of goto statements in spaghetti code.

Note that a change may only be semantics-preserving with respect to a particular aspect of the software semantics, while it is semantics-modifying when taking other aspects of the semantics into account. For example, a
typical refactoring operation will preserve the overall input-output semantics of the software, but may modify the efficiency or memory usage, which is another aspect of the software semantics that may be equally important.

The type of change is clearly orthogonal to the previous dimensions. First, some semantics-preserving changes can be fully automated, while semantics-modifying changes typically require a lot of manual intervention. Second, semantics-preserving changes can be supported by a formal foundation or not. Third, semantics-preserving changes can be a crucial part in the change process, as in the case of extreme programming.

<table>
<thead>
<tr>
<th>Group</th>
<th>Dimension</th>
<th>Section</th>
<th>Characterizing Factor</th>
<th>Influencing Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporal properties [when]</td>
<td>time of change</td>
<td>2.1.1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>change history</td>
<td>2.1.2</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>change frequency</td>
<td>2.1.3</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>object of change [where]</td>
<td>artifact</td>
<td>2.2.1</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>granularity</td>
<td>2.2.2</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>impact</td>
<td>2.2.3</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>change propagation</td>
<td>2.2.4</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>system properties [what]</td>
<td>availability</td>
<td>2.3.1</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>activity</td>
<td>2.3.2</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>openness</td>
<td>2.3.3</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>safety</td>
<td>2.3.4</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>change support [how]</td>
<td>degree of automation</td>
<td>2.4.1</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>degree of formality</td>
<td>2.4.2</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>process support</td>
<td>2.4.3</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>change type</td>
<td>2.4.4</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1. Dimensions That Characterize or Influence The Mechanism Of Change

4.2.6.5 Dimensions as Characterizing Factors

In determining the dimensions that characterize the change mechanism, we adhered to two simple heuristics. The first was to review existing literature to determine if the dimension had been frequently used to position software evolution
tools. So, for example, in the literature, software tools have often been introduced as run-time or load-time (with compile-time being implicitly accepted otherwise).

The second heuristic, was to put the dimension in a simple sentence of the form: "The change mechanism is <dimension>". If such a sentence makes sense, then the dimension must reflect the essence of the change mechanism and is thus classified as a characterizing mechanism. For example, because we can say that "The change mechanism is compile-time/load time/run-time", the time dimension is a characterizing factor. In a similar way the dimensions of 'change history', 'activeness', 'degree of automation', 'process support' and 'type of change' are classified as characterizing factors.

Perhaps the most surprising of these is the 'activeness' dimension, as it refers to the system under change. However, it should be noted that, when a system is active in changing itself, it is also part of the change mechanism. Thus, a sentence of the form: "The change mechanism is proactive" can be sensibly formed.

4.2.6.6 Dimensions as Influencing Factors

In determining if the dimension was an influencing factor we followed a third heuristic. For each dimension, the group tried to find an example of a way in which it could influence the change mechanism. For example, system 'availability' could typically affect the change mechanism. If a system is required to be highly available, then this would suggest a run-time change mechanism. Low availability would allow run-time or compile-time changes. It is no coincidence that Table 1 classifies all dimensions of group 'object of change' as influencing factors, as these dimensions are in general independent of concrete change mechanisms and rather describe particular properties that can be supported or unsupported by a concrete change mechanism.
Note that being a characterizing factor and being an influencing factor are not mutually exclusive. For example, 'time of change', apart from being a characterizing mechanism, also influences the change mechanism by prompting additional change activities like state management.

4.3 MODELS FOR SOFTWARE EVOLUTION

4.3.1 INTRODUCTION

The phenomenon of software evolution has been described with several models of different nature being proposed to understand and explain the empirical observations. Some of these models purport to be universally applicable to all software development processes. However, the models in this book were built mainly observing software developed using a traditional centrally-managed waterfall development process, or one of its variants.

This area has been approached from two different perspectives. One considers the processes, methods and techniques to implement and evolve software (the how). The other applies systematic observation of empirical data to achieve an understanding of the causes and general characteristics of the phenomenon (the what/why). Both perspectives are related: the study of the what/why is important in order to achieve and appropriate plan, manage and control the various software engineering activities; and to guide the development of new processes, methods and tools, that is, to guide the how.

The link between the how and the what/why perspectives is governed by several guidelines are derived from Lehman’s laws of software evolution. The output of
the what/why study is exemplified by empirical generalizations such as Rajlich and Bennett’s model of the lifecycle and Lehman’s laws of software evolution [52]. Therefore, we expand and refine the empirical hypothesis presented in the staged model of software evolution so that it can be applied to FLOSS projects. For this, we compare and contrast the existing empirical knowledge (e.g. as derived from studies of proprietary systems evolved under traditional processes with the emergent FLOSS paradigm. This revision will help FLOSS developers and practitioners to characterize any FLOSS system, in terms of which phase it is currently undergoing, or which phase it will more likely move to.

4.3.2 THE STAGED MODEL

The staged model for software evolution represents the software lifecycle as a sequence of steps. Based on the traditional commercial projects, the core idea of the model is that software systems evolve through distinct stages:

Initial development, or alpha stage, includes all the phases (design, first coding, testing) achieved before the first running version of the system. In this stage, usually no releases are made public to the users.

4.3.2.1 Evolutionary pressures enhance the system with new features and capabilities in the phase of the evolution changes: binary releases and individual patches are made available to the users, and feedback is gathered to further enhance the system.

4.3.2.2 As long as the profitability of either new enhancements or changes to the existing code base is overcome by the costs of such modifications, the servicing phase is reached. The system is considered mature, changes are added to the code base, but no further enhancements (apart from patches) are provided to the end users.
4.3.2.3 When the service is discontinued and no more code patches are released, the stage of phase-out is meant to declare the system's end. This can be associated with the presence of a new enhanced system substituting the old one.

4.3.2.4 The old system serves as a basis for the new one and then it is closed down.

4.3.3 FLOSS STAGED EVOLUTION MODEL

![Staged Model Adapted To FLOSS Systems](image)

Figure 4.4: Staged Model Adapted To FLOSS Systems

It is possible to analyze each of the phases of the cited model, and observe when and how differences and commonalities arise.

4.3.3.1 Initial Development

Traditional commercial systems are typically built by a fixed amount of designers, developers and testers. Also, software is usually published (as executables) only after the first release is deemed as "correctly running". The
recipients of this release are the end users. It is a rare event that source code access is granted too: we are aware of just one specific case in which a commercial software house, using an agile development process, gives the possibility for users to download the application from the public versioning system repository. FLOSS systems, on the other hand, typically start with a small amount of early developers, and eventually new developers join after a certain initiation period. FLOSS systems, instead, may be released (in binaries as well in source code) well before they are complete or working. Recipients are not only end-users, but also any other developer: read-only access to the versioning system is given to anybody, which leads to the already mentioned continuous release even before the first official release. In the following, results from empirical case studies are reported to justify the mentioned differences.

4.3.3.2 Evolution Changes

Several releases are observed both in traditional commercial and FLOSS systems. The staged model, as proposed, also incorporated the possibility of evolution through several branches of releases. In traditional commercial systems, most of the changes are distributed and applied as patches on the existing code base. New versions of the software systems are distributed regularly, albeit a higher frequency is perceived as an instability factor. Feedback is provided by users in the form of requests for change or bug signaling, and collected as a set of new requirements for new releases, or in intermediate code patches.

In FLOSS systems, new releases of systems and patches are available more often, and this is usually perceived by FLOSS developers as a vitality factor. Although traditionally many FLOSS projects published a new release "once it is
ready”, in recent times several FLOSS projects have moved to a time-based release planning, offering a new stable version for end-users on a periodic basis (for Ubuntu and GNOME, for instance, every six months). Feedback is provided by users in the same forms as in commercial systems, but also under the form of code patches that users write themselves, and which possibly will be incorporated into new releases of the system.

The loop of evolution changes presented in Figure 4.4 may be accomplished through many years. Both traditional commercial and FLOSS systems have shown the characteristics of long-lived software. For instance, operating systems like OS360, the various flavors of UNIX, or the Microsoft Windows, as well as the FLOSS Linux kernel, FreeBSD or OpenBSD, have been successfully evolving for decades. It is noticeable that while the evolution loop can be found both in commercial and FLOSS environments, research has shown that growth dynamics in both cases differ significantly, at least in the case of large projects. Some of the FLOSS projects have a superlinear growth rate (for example, Linux), while a majority of the large projects studied grow linearly. Both behaviors (superlinearity and linearity) seem to be in contradiction with Lehman’s laws of Software Evolution [52] that implies that size over time shows a decelerated pattern.

4.3.3.3 Servicing

This phase was first described for traditional commercial systems, when new functionalities are not added to the code base, whilst fixes to existing features are still performed. The transition from evolution to servicing is typically based on the economic profitability of the software system. When revenues from a software product are not balanced by the costs of its
maintenance, the system is no longer evolved, and it may become a legacy system. For FLOSS systems, on the other hand, the evolutionary behavior shows often stabilization points, where the size of the overall system does not change, albeit several releases are made available, and a long time interval is achieved. Although a servicing stage could be detected, a new evolution period is later found. Researchers have also assessed that some 20% of the overall number of projects within Source-Forge are “tragedies of growth”, i.e. they evolved for a period, but then fewer and fewer additions were made.

The presence of servicing stages has been detected in the past through an overall small increase of the code base (say, less than 10% over several releases and temporal time). It was observed, in some of the systems, a very fast increase in size, and a corresponding fast evolution, followed by a stabilization phase which lead to the complete abandonment of the project. For some other analyzed systems, instead, (e.g. the Gaim system), albeit the same initial fast growth rate, and a transition from evolution to servicing, were observed, a new period of evolution was also found. In yet other systems (the Grace system), it was possible to observe an overall growing pace even when the system was abandoned by the initial lone developer, and was handed over a new team of developers, followed by a later stabilisation period. The dashed arc between the evolution and servicing stages of Figure 4.4 is displaying this possibility.

4.3.3.4 Phase Out

In traditional commercial systems, the phase out of a software system happens when the software house declares that neither new functionalities,
nor the fixing of existing ones will be performed. The system becomes then a legacy application. The same behaviour is detectable in FLOSS systems, when development teams declare their intention not to maintain the system any more. The main difference between the traditional commercial approach and the FLOSS cases is the availability of source code. In some, specific cases new developers may take over the existing system, and with the availability of source code, bring it to a new stage of evolution.

Many FLOSS projects have been reported to renew their core groups. For instance, three different categories of FLOSS projects were identified: code gods, generations and mixed behavior. Code gods projects are maintained by the same group of developers during the whole lifetime of the project. Generations projects exhibit a renewal in the core group of developers; the group of people that were the main developers at an early moment in the lifetime are not the main developers in posterior moments. Therefore, a generational relay has taken place in the project. Mixed projects exhibit neither a pure code god or generations profile, but a intermediate state among those two extremes.

4.3.4 SOFTWARE EVOLUTION IN SOFTWARE DESIGN AND SOFTWARE EVALUATION TECHNIQUES

In this section, we describe what we mean by the gap between software design/evaluation techniques and design patterns/styles. We consider the most well-known design and evaluation techniques and describe how identifying the context of the evolution problem helps in the choice of the software evolution mechanisms.
4.3.4.1 The Unified Process

The Unified Process is a use-case driven, iterative and architecture centric software design process. The life of a system is composed of cycles and each cycle concludes with a product. Each cycle is divided into four phases. The first phase is the inception phase and in this phase the requirements are analyzed and a general vision about the product is developed. This phase is followed by the elaboration phase in which the architectural baseline of the product is developed. During the third phase the product is built and this phase is labeled as construction.

The last step, called transition, involves the manufacturing of the product. To support evolution in Unified Process, there must be link between the transition phase of the previous cycle and the inception and elaboration phases of the current cycle. With this link, the designer, while gaining a perspective about the old system, can also develop ideas about integrating new requirements to the system. That is, with this link the designer can identify the evolution problem he is faced with, select the suitable evolution technique and then apply this technique to the design. For example, if the new requirements extend the current system, the designer can choose to delegate the current system with new requirements. Thus, the new system can be designed using means of delegation mechanisms like call forward protocols.

4.3.4.2 Software Architecture Synthesis Process

The Software Architecture and Synthesis process (Synbad) is an analysis and a synthesis process, which is a widely used process in problem
solving in many different engineering disciplines. The process includes explicit steps that involve searching solutions for technical problems in solution domains. These domains contain solutions to previously solved, well established, similar problems. Selection of which solution to use from the solution domain is done by evaluating each solution according to quality criteria. The method consists of two parts, which are solution definition and solution control. The solution definition part involves identification and definition of solutions. In this part client requirements are first translated into a technical problems; these are the problems that are actually going to be solved. These technical problems are then prioritized and a technical problem is selected according to this priority order. The solution process involves identifying the solution domain for the problem and searching possible solution abstractions in this domain. Selected solution abstractions are, then, extracted from the solution domain and specified to solve the problem in consideration. In the last step of the solution definition part, the specified solutions are composed to form the architectural description of the software. The solution abstractions may cause new problems to be found; thus there is a relation, labeled as 'discover', between solution abstraction and technical problem.

The solution control part of Synbad represents the evaluation of the solutions. The evaluation conditions (e.g. constraints on applying the solution) are provided by the sub-problem and by the solution domain. The solutions extracted from solution domains are expressed as formal models for evaluation. Then optimizations are applied to the formal model in order to meet the constraints and the quality criteria. The output of these optimizations is then used to refine the solution. Synbad treats each problem
separately and the solutions of each problem are composed to form the solution of the overall problem the software is going to solve. Thus, this process inherently supports the addition of new requirements to evolve the software. When new requirements arrive, their technical problems are analyzed and the solution abstractions for these technical problems are extracted from the solution domain. Each extracted solution abstraction causes a new technical problem to be identified, which can be stated as "given a solution, what are the techniques to compose this solution to the system". For this problem, the solution abstraction and the system define the quality criteria and constraints. Here, the quality criteria are the non-functional requirements of the system. The constraints, on the other hand, are the factors that affect the selection of the composition mechanisms. For example, if the extracted solution is already implemented and its source code can not be changed, then the composition mechanism should be a run-time solution.

4.3.4.3 Scenario-Based Evaluation Techniques

There are many scenario-based techniques that evaluate software architectures with respect to certain quality attributes. Scenario-based Architecture Analysis Method (SAAM), for example, is a method for understanding the properties of a system's architecture other then its functional requirements. The inputs to SAAM are the requirements, the problem description and the architecture description of a system. The first step of SAAM is scenario creation and software architecture description.

During this, all stakeholders of the system must be present; scenarios are considered to be complete when a new scenario doesn't affect the
architecture. In the last step, scenarios are evaluated by determining the components and component connections that need to be modified in order to fulfill the scenario. Then the cost of modifications for each scenario is estimated in order to give an overall cost estimate. In recent years, SAAM has been specialized to focus on a quality attribute like modifiability and extended to find the trade-off between several quality attributes.

These methods can easily be used or adapted to find the impact of evolution requests. Though, after finding the impact, software engineers are faced with the problem of finding the mechanisms that are applicable to the evolution problem in consideration. When with scenarios certain components are found to be hard to evolve, how can the software engineer make them easier to evolve? For this, the evolution problem should be analyzed in detail; the constraints of the software system and the evolution mechanisms should be identified and the most applicable mechanisms should be used to replace/change the components. That is, the context of the software evolution should be identified in order to select the applicable mechanisms. Currently, none of the evaluation techniques has steps that include such analysis. Here, we identify the contexts of evolution problems and mechanisms. Thus, after finding the impact, the software engineer can find the applicable evolution mechanisms by selecting the context of the problem he is dealing with.
4.3.4.4 Design Pattern and Styles

In the software engineering domain there are many mechanisms that can cope with various evolution problems. Some design patterns, for example, make it is easier to add new behavior to the system. In their study of comparing design patterns to simpler solution for maintenance, Prechelt et al [38] concludes that due to new requirements design patterns should be used, unless there is an important reason to choose the simpler solution, because of the flexibility they provide. Mens and Eden list some of these evolution mechanisms and determine how helpful they are for some evolution situations. Their analysis shows that these mechanisms are very costly to use for certain evolution problems while for others they are not. This shows that there are contexts for these techniques.

Thus, identifying these contexts and then selecting the mechanism to use may greatly ease the procedure for the evolution of software. The problem here is that these contexts are not analyzed. We know that design patterns and styles can ease evolution operations but what the applicable mechanisms are for a given evolution problem is not known.
4.4 Challenges in Software Evolution

4.4.1 Introduction

Today's information technology society increasingly relies on software at all levels. This dependence on software takes place in all sectors of society, including government, industry, transportation, commerce, manufacturing and the private sector. Productivity of software organisations and software quality generally continue to fall short of expectations, and software systems continue to suffer from symptoms of aging as they are adapted to changing requirements. One major reason for this problem is that software maintenance and adaptation is still undervalued in traditional software development processes.

The only way to overcome or avoid the negative effects of software aging is by placing change in the center of the software development process. Without explicit and immediate support for change and evolution, software systems become unnecessarily complex and unreliable. The negative influence of this situation is rapidly increasing due to technological and business innovations, changes in legislation and continuing internationalisation.

One must therefore advance beyond a restricted focus on software development, and provide better and more support for software adaptation and evolution. Such support must be addressed at multiple levels of research and development. It requires:

4.4.1.1 Basic research on formalisms and theories to analyse, understand, manage and control software change;

4.4.1.2 The development of models, languages, tools, methods, techniques and heuristics to provide explicit support for software change;
4.4.1.3 More real-world validation and case studies on large, long-lived, and highly complex industrial software systems.

4.5 Classification of Challenges

To increase the readability, the challenges are classified according to a number of more or less orthogonal dimensions. Table 4.1 gives a comprehensive overview of all these challenges using this classification.

4.5.1 Time horizon. Is a short, medium or long-term effort required in order to achieve results?

4.5.2 Research target. Is the challenge related to the management, control, support, understanding or analysis of software evolution?

4.5.3 Stakeholders. Who is interested in, involved in, or affected by the challenge? Given the diversity of challenges, many different people can be involved: managers, developers, designers, end-users, teachers, students, researchers, and so on.

4.5.4 Type of artifact under study. Which type of artifact(s) does the challenge address? Artifacts should be interpreted in the broad sense here since they can refer to formalisms, tools, techniques, models, meta models, languages, programs, processes, systems, and many more.

4.5.5 Type of support needed. Which type of support is needed in order to address the challenge? Although this question is completely different from the previous one, the list of possible answers is the same. One can provide
formal support, tool support, language support, process support and so on in order to address software evolution.

4.6.1 Enumeration of Challenges

4.6.1 Preserving And Improving Software Quality

The phenomenon of software aging, coined by Dave Parnas, and the laws of software evolution postulated by Manny Lehman [52] agree that, without active countermeasures, the quality of a software system gradually degrades as the system evolves. In practice, the reason for this gradual decrease of quality (such as reliability, availability and performance of software systems) is for a large part caused by external factors such as economic pressure. The negative effects of software aging can and will have a significant economic and social impact in all sectors of industry. Therefore it is crucial to develop tools and techniques to reverse or avoid the intrinsic problems of software aging. Hence, the challenge is to provide tools and techniques that preserve or even improve the quality characteristics of a software system, whatever its size and complexity.

4.6.2 A Common Software Evolution Platform

A major difficulty when trying to address the previous challenge has to do with scalability. The need is to develop solutions that are applicable to long-lived, industrial-size software systems. Many of the tools that must be built to manage the complexity intrinsic to software evolution are too complex to be built by single research groups or individuals. Therefore, a closely related challenge is to develop and support a common application framework for doing joint software evolution research. This challenge raises issues such as tool integration and interoperability, common exchange formats and standards, and
so on. As an example of such a shared framework that served as a common software evolution research vehicle within the RELEASE network is the Moose reverse engineering environment. A concrete goal could be to try and extend this framework with tools to analyse, manage and control software evolution activities.

Another candidate that may serve as a common platform is Eclipse. It has the advantage of visibility and industrial acceptance and also permits reuse of certain components (e.g., Java parsing). An important disadvantage is its lack of control over releases. One researcher once mentioned that he had to keep several versions of the platform because not all plug-ins work on all versions. There is also the issue of exploratory prototyping, which is better supported by environments like Smalltalk. Both options should probably co-exist, although this of course implies duplication of effort.

4.6.3 Supporting Model Evolution

Although support for software evolution in development tools can still be advanced in many ways, there are already a number of success stories. One of them is program refactoring, introduced by John Opdyke [75] in the early 1990s as a way to improve the structure of object-oriented programs without affecting their desired external behaviour. Since the publication of Martin Fowler’s book on refactoring, this program transformation technique has gained widespread attention. Today, refactoring support has been integrated in many of the popular software development environments.

Unfortunately, it is observed that almost all existing tool support for software evolution is primarily targeted to programs (i.e., source code). Design and
modelling phases (supported by UML CASE tools, for example) typically provide much less support for software evolution. Taking the example of refactoring, we didn’t find any modelling tool providing adequate means for refactoring design models. Research in model refactoring is just starting to emerge. This can be generalised into the following challenge: Software evolution techniques should be raised to a higher level of abstraction, in order to accommodate not only evolution of programs, but also evolution of higher-level artifacts such as analysis and design models, software architectures, requirement specifications, and so on. Since the advent of model-driven software engineering, this challenge becomes increasingly more relevant, and techniques and tools for dealing with model evolution are urgently needed.

4.6.4 Supporting Co-evolution

A challenge that is related to the previous one is the necessity to achieve co-evolution between different types of software artifacts or different representations of them. Modification in one representation should always be reflected by corresponding changes in other related ones to ensure consistency of all involved software artifacts. To give but a few examples, support for coevolution is needed between:

4.6.4.1 Programs (source code) and design models or software architectures;
4.6.4.2 Structural and behavioural design models. This is for example the case with UML, where different models are used to express structure (e.g., class diagrams) and behaviour (e.g., sequence diagrams and state-transition diagrams);
4.6.4.3 Software (at whatever level of abstraction) and the languages in which it is developed. Whenever a new version of the programming, modeling
or specification language is provided, it is quite possible that programs that worked perfectly in a previous version of the language fail to function in the new version

4.6.4.4 Software and its business, organisational, operational and development environment. Changes in each of these environments will impact the software and conversely. This feedback loop is well known in software evolution research

4.6.4.5 Software and its developer or end-user documentation

To provide better support for co-evolution, it is worthwhile to take a look at other domains of science that can hopefully provide better insights in the matter. Linguistic theory and the history of natural language evolution may increase understanding in how software languages evolve. In order to better understand software co-evolution, it could be interesting to look at co-evolution in biology. In fact, the term coevolution originated in biology, and is borrowed by computer scientists to describe a similar situation in software development.

4.6.5 Formal Support For Evolution

According to Brownsworth [74], “a formal method of software development is a process for developing software that exploits the power of mathematical notation and mathematical proofs.” For several decades, formal methods have been advocated as a means to improve software development, with an emphasis on software specification, verification and validation. Nevertheless, as Robert Glass [76] observes “Formal methods have not, during that extended period of time (well over 30 years by now), had any significant impact on the practice of software engineering.” He
points out a major cause of this problem: “What in fact most practitioners
tell me about specifications is that the needs of the customers evolve over
time, as the customer comes to learn more about solution possibilities, and
that what is really needed is not a rigorous/rigid specification, but one that
encompasses the problem evolution that inevitably occurs.”

Unfortunately, existing formal methods provide very poor support (or even
none at all) for evolving specifications. Let us take the example of formal
verification, which aims to prove mathematically that the implementation of
a software system satisfies its specification. Specialists that were consulted
in relation to this question agreed that even today there are no truly
incremental verification approaches available.

With current verification tools, even if small localized changes are made to
the specification of a program, the entire program needs to be verified again.
This makes the cost of verification proportional to the size of the system.
What is desired is that it is proportional to the size of the units of change.
This leads to the next challenge in software evolution: In order to become
accepted as practical tools for software developers, formal methods need to
embrace change and evolution as an essential fact of life.

Besides the need for existing formal methods to provide more explicit
support for software evolution, there is also a clear need for new formalisms
to support activities specific to software evolution
4.6.6 Evolution As A Language Construct

As a very interesting research direction, programming (or even modelling) languages should provide more direct and explicit support for software evolution. The idea would be to treat the notion of change as a first-class entity in the language. This is likely to cause a programming paradigm shift similar to the one that was encountered with the introduction of object-oriented programming. Indeed, to continue the analogy, one of the reasons why object-oriented programming became so popular is because it integrated the notion of reuse in programming languages as a first-class entity. The mechanisms of inheritance, late binding and polymorphism allow a subclass to reuse and refine parts of its parent classes.

It can be pointed out that explicit support for software evolution is considerably easier to integrate into dynamically typed languages that offer full reflective capabilities. Classboxes are also a new module system that controls the scope of changes in an application. Changes can be introduced in a system without impacting existing clients; changes are only visible to new clients desiring to see the changes.

4.6.7 Support for Multilanguage Systems

Schmidt [16] pointed out that another crucial, and largely neglected, aspect of software evolution research is the need to deal with multiple languages. Indeed, in large industrial software systems it is often the case that multiple programming languages are used. More than three languages is the rule rather than the exception. Therefore, software evolution techniques must provide more and better support for multi-language systems. One way to tackle this problem is to provide techniques that are as language-
parametric (or language-generic, or language-independent) as possible. Note that this challenge is becoming increasingly more relevant as the number of languages needed or used in software systems is increasing. Programming languages, modelling languages, specification languages, XML-based languages for data interchange, domain-specific languages, business modeling languages, and many more are becoming ever more widely used.

4.6.8 Integrating Change In The Software Lifecycle

It is important to investigate how the notion of software change can be integrated into the conventional software development process models. A typical way to include support for change into a more traditional software process models is by resorting to an iterative and incremental software development process. So-called agile software processes (including the well-known extreme programming method) already acknowledge and embrace change as an essential fact of life. Other processes, such as the staged lifecycle model, have been proposed as an alternative that provides explicit support for software change and software evolution.

4.6.9 Increasing Managerial Awareness

Besides better understanding of, and better support for, evolutionary process models, there is also a need to increase awareness of executives and project managers of the importance and inevitability of software evolution. Training is needed to convince them of the importance of these issues, and to teach them to plan, organise, implement and control software projects in order to better cope with software changes. We suggest explaining the importance of software evolution through the SimCity metaphor. This computer game simulates a city and is a typical example of a highly complex
dynamic system where continuous corrective actions of the "manager" are needed in order to avoid deteriorating the "quality" of the city and, ultimately, its destruction or desertion.

4.6.10 Need For Better Versioning Systems

Although support for software evolution in software development tools can still be improved in many ways, there are already a number of success stories. One of them is version management. Version control is a crucial aspect in software evolution, especially in a collaborative and distributed setting, where different software developers can (and will) modify the program, unaware of other changes that are being made in parallel.

A wealth of version control tools is available, commercial as well as freeware. The most popular one is probably CVS (www.cvs.org). Nevertheless, for the purpose of analysing the evolution of software systems, these version repositories clearly fall short because they do not store enough information about the evolution. Therefore, the challenge is to develop new ways of recording the evolution of software that overcome the shortcomings of the current state-of-the-art tools. When addressing this challenge, it is necessary to communicate and coordinate with the research community on Software Configuration Management, that is trying to address very related issues.

4.6.11 Integrating Data From Various Sources

One promising approach to reason about the evolution history of software systems, is the integration of data from a wide variety of sources:
bug reports, change requests, source code, configuration information, versioning repositories, execution traces, error logs, documentation, and so on. Besides all of the above information, it is equally important to take into account information about the software process during the analysis of change histories: the software team (size, stability, experience, ...), individual developers (age, experience, ...), the project structure (hierarchical, surgical, flat, ...), the process model (waterfall, spiral, agile, ...), the type of project (e.g., open source), and so on. Indeed, Conway’s law postulates that the architecture of a software system mirrors the structure of the team that developed it (and, more generally, the structure of a system tends to mirror the structure of the group producing it). The main challenge here is to find out how these different kinds of data can be integrated, and how support for this integration can be provided. Having a flexible and open-meta model as the one of the Moose reengineering environment supporting entity annotations and version selection should be regarded as a first step in that direction.

4.6.12 Analysing Huge Amounts Of Data

Given the sheer amount of data that needs to be processed during the above analysis, new techniques and tools are needed to facilitate manipulation of large quantities of data in a timely manner. In order to achieve this, one can probably borrow from related areas of computer science that deal with similar problems. For example, one may consider using data mining techniques as used by the database community, or techniques related to DNA sequence analysis as used in bio-informatics.
These techniques could be implemented as an extension of current tools such as CodeCrawler that already supports the management of large data sets via polymetric views (i.e., views enriched with semantical information). Another attempt that has been made to address this challenge is a technique that suggests to the developer changes to be performed based on the co-occurrence of past changes.

4.6.13 Empirical Research

In the context of software evolution there is a need for more empirical research. Among others, comparative studies are urgently needed to measure the impact of:

4.6.13.1 Process models: in an industrial setting, which software process is most appropriate for which type of evolution activity?

4.6.13.2 Tools: to which extent does the use of a tool facilitate the execution of a particular evolution activity compared to the manual execution of the same activity; how does one compare the performance of different tools to carry out the same evolution activity?

4.6.13.3 Languages: what is the impact of the programming language on the ease with which certain evolution activities can be performed? For example, dynamically typed languages with reflective capabilities seem to be more suited than other languages to support the task of runtime evolution.

4.6.13.4 People: to which extent does the experience, background and training of a software developer contribute to his ability to carry out certain software evolution activities?
4.6.14 Need For Improved Predictive Models

Predictive models are crucial for managers in order to assess the software evolution process. These models are needed for predicting a variety of things: where the software evolves, how it will evolve, the effort and time that is required to make a change, and so on. Unfortunately, existing predictive models, typically based on software metrics, are far from adequate.

To counter this problem, Woodside CM [45] suggested looking at metrology research, the science of measurement, which explicitly takes into account the notion of uncertainty that is also inherent in software evolution. Girba’s “Yesterday’s Weather” measurement is another step in the same direction. This measurement characterizes the climate of changes in a system and helps assessing the trust that may be placed in the continuity of changes (based on the assumption that assets that have changed in the recent past are more likely to change in the near future).

4.6.15 Evolution Benchmark

In order to adequately test, validate, and compare the formalisms, techniques, methods, and tools to be developed for the other challenges, it is useful to come up with, and reach a consensus on, a common set of evolution benchmarks and case studies which, together, are representative for the kinds of problems needing to be studied. Given the amount of long-lived, industrial-size, open-source projects available today, it should be feasible to come up with such a benchmark.
4.6.16 Teaching Software Evolution

One of the best ways to place change and evolution in the center of the development process is to educate the future generations of software engineers. However, classroom programming exercises usually are well specified, have a single release, and are small in size. Capstone projects are more amenable to convey the need for dealing with software evolution, but on the one hand they are often supervised in a rather loose mode, and on the other hand it is preferable to prepare students in earlier courses with the concepts and tools they need to handle changes in their project. Therefore, a big challenge for everyone involved in teaching concerns how to integrate the ideas, formalism, techniques and tools for software evolution into our computer science curriculum in a meaningful way. As a community, we need to decide upon what we want to teach, how we want to teach it, and provide the necessary course material for it.

4.6.17 A Theory Of Software Evolution

It seems that often researchers either do empirical investigations into the evolution of a given system over its life-time, or propose tools and techniques to facilitate evolution. But it is not always clear what one gets from all the collected data nor if the tools actually are informed by typical evolution patterns. One needs to study and compare evolution activities before and after the installation of some tool supporting such activities. To undertake such studies, it is necessary to develop new theories and mathematical models to increase understanding of software evolution, and to invest in research that tries to bridge the gap between the what (i.e., understanding) of software evolution and the how (i.e., control and support) of software evolution.
Nevertheless, this theory still remains to be formalized and enriched. For example, Lehman [50] suggested that many software failures are due to changes that impact on the initial (often implicit) assumptions, and therefore a theory of software evolution must take assumptions into account.

4.6.18 Postdeployment Runtime Evolution

Maintenance and evolution of continuously running systems have become a major topic in many areas, including embedded systems, mobile devices, and service infrastructures. There is an urgent need for proper support of runtime adaptations of systems while they are running, without the need to pause them, or even to shut them down. For that, further steps are needed to make the deployment, application, and exploration of dynamic adaptations more comprehensible. Dynamic Service Adaptation (DSA) is a promising approach trying to address these issues by providing appropriate means to introspect and navigate basic computational structures and to adjust them accordingly. While evolution support at runtime via dynamic adaptation addresses many of the requirements stated above, it does not address program or system comprehension. On the contrary, systems that are changed dynamically are harder to understand using contemporary approaches. Proper tool support is needed for the exploration and manipulation of both basic and enhanced runtime structures as well as change events and their history.