Facets of Software Evolution

Roland T. Mittermeir


4.1 Introduction

Software and evolution are very general terms. Hence, people use them in different contexts with quite different semantics, with the risk of inconsistent conclusions. To avoid this problem, this chapter focuses on terminology in the area of software evolution. It is intended to motivate other software engineering researchers to precisely define the scope of the problems they are addressing in their research and to help students understand the breadth of the topic.

On the basis of reflections on the nature of software and on aspects of evolution, a categorisation of software is proposed to help practicing software engineers to choose a proper evolution strategy, depending on the nature of both, the system at hand and the change to be performed.

To advance in any discipline requires that one emphasises the role of efficient communication. Interactive speech acts allow the communicating partners to probe directly whether the frame of reference is adequately adjusted so that the terms used in the communication are properly understood. With written communication, one has to be more careful. Notably, when using highly generic terms, speaker and listener take a risk to find themselves, perhaps unknowingly, caught in a homonym trap. Different mental specialisations of the generic term might be so far apart that miscommunication will eventually set in. Software and evolution are both generic terms. It is, therefore, no wonder that in discussions involving these terms, participants talk at cross purposes, unaware of the fact that they are addressing different problems and proposing different cures.
Sharpening the semantic focus of these terms is the central aim of this chapter. It first discusses the semantic content of the term software and considers the various interpretations one might attach to the word evolution. After this initial consideration, the term software is mapped to five different levels of a size/complexity spectrum. On the basis of this stratification, approaches to handle evolution are mentioned to exemplify the situation and show approaches to cope with it. The difference in these approaches is due to humans’ limitations in information processing. The arguments raised are demonstrated in the context of a sizable software system.

4.2 What is Software?

Answering the question: ‘What is software’? should be easy for software professionals. However, an ad hoc experiment conducted by Osterweil in preparing a panel for ICSE 2001 [1] showed that even a sample of renowned software engineering researchers could not come up with a common definition for this term. Instead, analogies and relations to other artefacts were voiced.

4.2.1 Software: A Technical Artefact

When introducing new concepts, notably those related to methodology, software scientists resort to analogies with other engineering artefacts. For instance, when arguing that prototyping should become an accepted part of software development methodology, the spectrum of analogies used ranged from assembly line produced items like cars, via complex industrial products like airplanes to strictly custom-specified unique objects, like architect-designed houses [2]. This indicates that the concepts people have in mind when talking about software cover a wide range. A point of consensus might be that software, at least software that is subject to evolution, is a technical artefact created in a human thought process. The opinion, software is executable mathematics, though occasionally voiced, will, as pointed out by Parnas [3], not apply to evolutionary software. But while analogies serve well for demonstrative purposes, they are less adequate as basis for definitions.

When coining a definition for a tangible object, one usually refers to aspects such as the material it is made of, its external appearance, such as shape, colour or size, or to its purpose, that is, the function it is to serve. For software, this does not yield convincing results.

As generally agreed, software has no physical substance. It is immaterial. Thus, it lacks shape and colour. Nevertheless, one talks about the size of software. However, size needs qualification when referring to software. Some people would measure it by Lines Of Code (LOC), others by Kilo Delivered Source Instruction (KDSI) [4–6]. In another context, one might refer to the amount of physical memory consumed on disk or on working-storage. Intellectual complexity might also be an important indicator of size. However, it can only be measured indirectly by referring to the volume of code or documentation or by measures derived from code, such as the cyclomatic number [7].

4.2.2 Software: A Utility

Taking purpose as key criterion, one might consider the overall utility software-based systems serve for end-users or for society at large. This approach fails in so far, as
systems serving some end-user need never consist of software alone. They need at least hardware on which this software is to run. Thus, defining software as ‘the thing that makes computers behave in some prescribed way’ seems to hit a point. This is at least in agreement with several earlier attempts at defining software. It comes close to the utilitarian approach that claims function dominates material [8, 9]. But it is, once again, quite unsatisfactory to define something by just focusing on the relationship between this still undefined something and some other well-defined class of objects. Moreover, focusing exclusively on functionality may lead to contradictions when separating the required functional properties from nonfunctional requirements that must also be met [10]. It will also be totally at odds with established architectural principles claiming that the architecture of the building has to consider amongst other things the construction material to be used. This principle has not only aesthetic merit but it is also of major technical concern for the maintenance process and hence for everything that has to do with evolution. Thus, even when accepting that software is immaterial, one has to look for a positive answer concerning its very nature.

4.2.3 Software: A Text, Reflecting Reality

Focussing on size as measured by LOC or KDSI points to source code’s textual representation. But is not design, the intermediate result of an early stage of development, also software? So are binaries, the result of a later development stage? The linguistic representation also crosses with the purpose, since it can be changed drastically (e.g. by replacing an algorithm with a more efficient one) without changing the functionality of the software system. Likewise, different compilers will yield different binaries from the same source code. The differences might be due to variations in the optimisation strategies or might result from compiling for different machines. This should raise further questions against the argument that the relationship between machine and observable behaviour is the very nature of software. At least from the source code perspective, it is twice the same product. Looking at binaries, though, two different products and hence different relationships are seen.

A teleological perspective on the software development process might yield an answer though. Sizable software is not built in a single step that takes one from problem statement to executable binaries. It is rather constructed in an iterative process. In transforming the output of one step to that of a following step, information is added and layers of representations of the very same software result. This information initially stems from the problem domain. It is elicited during domain modelling and during requirements engineering. In later steps (design, programming), it rather relates to the solution domain. During compilation and when loading and interacting with the runtime environment, the information is likely to relate to the machine on which this software is to be executed.

While some of the information added in early phases serves only scaffolding purposes and is, therefore, removed in later steps, all information added throughout the process is added on purpose. Hence, taking a teleological perspective, one builds software systems to reflect in the information domain something that happens (or something the client

---

2 In line with [4, 11] the term software is in the sequel not restricted to code only. It encompasses also upstream products of textual or diagrammatical form as well as alternative downstream representations, such as carefully designed test-suites describing the desired behaviour.
wishes to happen) in the real world. Executable software is the ultimate reflection of
some reality in a chain of intermediate products reflecting this reality. These intermediate
reflections differ in their degree of precision, their granularity, as well as in the extent that
they take into account the machine on which the executable software is finally to be run.
Considering software as a specifically constrained form of a linguistic expression allows
this term to encompass not only binaries but also source code and all upstream products
necessary to derive an operational software system from an initial problem statement.

4.2.4 Software is Information

The contents of these intermediate representations are apparently information to be passed
to software professionals of the next development stage. To account for the individuals
involved in this process, one must leave the communications-theoretic definition of infor-
mation given by Shannon and Weaver [12] and move towards an interpersonal definition.
From this perspective, information does not exist per se but only in relation to a recipient.
The information content of a message may then be regarded as the degree to which the
data it contains influences the state or behaviour of the recipient of this message. That
recipient can be a human or a machine.

The state space of the receiver can be described by a probabilistic measure attached to
the individual values it might assume. Thus, the probabilities attached to this state space
may change upon receipt of the message. Hence, if the message cannot be interpreted or
if the interpreted message does not lead to a change in the receiver’s state space, the data
contained in the message does not qualify as information from the receiver’s subjective
perspective. If, on the other hand, the message leads to a state change, the degree of that
change characterises the subjective information content of the message.

Conceptualising software as information yields several benefits:

(a) It provides a clue why software is such a complex entity that is not ‘soft’ at all. It
hints at why software is difficult to write and even more difficult to modify. Lehman’s
funnel-concept [13] not only wants to cure the problem of its being mistakenly per-
ceived as soft by ‘hardening’ it. It also leads to a materialised separation of concerns.
Middleware systems, properly used, come close to this ideal.
(b) It also helps to clarify some limits faced in both, software development and soft-
ware evolution. With information as defined above, one is shifting the focus from the
machine to the humans who have to deal with software systems. This has a direct con-
sequence for tool builders, methodology developers as well as for technical managers.

Identification of software as information raises two issues:

(a) Since programs are also data with the information encoded in it arranged to pro-
cess other information encoded in some other form (data), the overall relationship
represents a tricky recursion.
(b) That recursion will be recognised as even more subtle when one observes that software
reflects some reality. In doing so, it reacts with this reality and thus becomes part of it.

The former loop is central to the concept of the von Neuman computer and thus central
to the nature of software. It certainly constitutes one of the difficulties encountered when
writing software. The second loop has been broadly discussed (e.g. Lehman’s *E-type systems* [14] or Yeh’s definition of *wicked systems* [15]). Taken together, these loops remind one of the artfully interwoven loops one finds in Bach’s fugue or in Escher’s drawings as discussed in Hofstadter’s book relating recursive structures in mathematics, drawing, and music to computing [16].

Before dwelling on these considerations, the next section focuses on the term *evolution*.

### 4.3 Evolution

This section describes two issues of importance for evolution of software systems: the phenomenon of evolution and drivers for evolution.

#### 4.3.1 Principles

According to the COBUILD dictionary [17], *evolution* refers to ‘a process of gradual change that takes place over many generations’ or ‘a process of gradual and uninterrupted change and development in a particular situation over a period of time’. Evolution is thus distinct from both, revolution (‘a complete, pervasive, usually radical change in something’) and complete standstill. The word evolution is also often used in the context of the activity supporting a particular transition in a gradual change process, as, for example, in the phrase ‘evolution of a notion or an idea’ or, in software engineering, the evolution from assembly to compilable (high-level) languages or from batch to interactive systems. This latter usage occurs sometimes in the context of presenting some methodology (e.g. [18]) or tool (e.g. [19]) conceived to aid the evolution of software.

Studying the phenomena of sustained change, that is, using the term evolution as noun, is less frequent. Amongst this usage, the ground-breaking analysis of the IBM OS 360 by Belady and Lehman [20] as well as Lehman’s subsequent work, most recently pursued in the FEAST projects [21], Parnas’s ICSE keynote on software aging [3] or Tamai’s analysis [22] are to be mentioned as exceptions in the vast literature on software maintenance where the word evolution appears repeatedly.

In studying change processes, software engineers have learned that, even if developers and maintainers are quite often caught by surprise, change does not occur at random. Lehman has shown that only a certain portion of a system changes between releases [23]. Its size remains stable within a rather constant bandwidth. Investigations by Baker and Eick [24] and by Gall *et al.* [25] have shown that the statistically constant change effort is unevenly distributed over modules. The reason for these observations may differ from situation to situation. Sometimes, managerial decision based on adherence to a predictable sequence of releases and employing a relatively constant workforce for system maintenance may be the primary source of the observed behaviour. In the other situations, the different exposure of system components to different change drivers is likely to cause this effect.

The assumption of different change drivers operating at different times is substantiated by the work of Antón and Potts [8]. Studying the evolution of features of a telephone system, they identified a set of specific evolution patterns. Among other results, this study showed that evolution of systems in terms of integration of new features does not progress gradually and linearly. New features tend to be introduced in bursts, followed by withdrawal of some of them (premature or misjudged developments). Such abrupt
stepwise (Antón and Potts call them ‘saltationist’) evolutionary stages are followed by a period of calm, during which the system is stabilised, at least from the point of view of features. Considering Lehman’s laws of system evolution [26] helps to interpret some of these results. These laws state that in order to remain alive, systems need to evolve. This necessitates periodic restructuring, that is, work must be applied to restore or improve the systems’ internal technical quality. Considering the constant activity law, one must assume that there are periods of feature extension, of feature stabilisation and of internal (mainly architectural) stabilisation.

Relating these observations, two interlinked driving forces for system evolution can be identified as follows:

(a) external market factors (or comparable social phenomena);
(b) internal feedback factors (while of technical nature, they too are influenced by human concerns).

4.3.2 Evolution Drivers

Market factors depend on human decisions. They involve the expectation and anticipation of decisions by other humans. Antón and Potts identified withdrawal periods immediately after substantial feature extensions. This suggests causes beyond mere technical problems. Rather, the social system was not ready to accept the full extent of the change offered. Human acceptance of change via the social system determines to a large extent what kind of new features are introduced at which time and at which level within the system evolution phase identified in [27].

With other systems, it is not the marketplace that drives evolution. Systems that have to keep up with legislative changes fall into this category. However, even in these cases, it is the social system that drives the change. Technical progress, too, is controlled by social processes of market forces as well as by the technology adoption behaviour of society. On a more detailed level, limited capacity of the maintenance/evolution staff determines the extent to which new technology (of whatever kind) is accepted as a driver for changing existing operational software.

From these reflections, the following hypotheses are proposed.

**Hypothesis 1:**

*Human-based considerations are the main external driving factor behind systems (and software) evolution.*

Considering software as information structured in a particular form in order to express or implement some reflection of reality leads to:

**Hypothesis 2:**

*Human (and social) limitations in information processing (acquisition, restructuring) limit the extent of software evolution.*

From the software producers/maintainers’ perspective, this applies to both, evolution stemming from external change drivers, and evolution due to the feedback properties of
very large software [28]. The second hypothesis brings size and complexity into play. If software is regarded as structured information, the distinction between size and complexity need not be considered directly. The intellectual reach of the individual user and the intellectual reach that an individual software developer can span will be of primary concern. The literature on programmer productivity and on attempts to provide indicators for planning the duration of a given development or maintenance task (e.g. [4–6]) provide clues as to what this intellectual span might be. The difference between the productivity for writing new code as opposed to maintaining old code [29] points to the fact that an author of new code just has to relate the intermediate result from a previous development step to his or her own ideas, whereas a maintenance programmer has to try to recover the encodings somebody else has made, interpret them and relate them to her or his own ideas.

To bring mere size and complexity to a comparable level, one might resort to Halstead’s definition of program volume [30]. As the detailed operationalisation of V and V* can be questioned, this chapter does not delve deeper into Halstead’s theory. Nevertheless, the broad brush notion of these concepts capture the basis on which rest the ensuing considerations.

4.4 Strata of Software-Size and Complexity

The categorisation of software into different strata, which possibly follow different evolutionary patterns, will be defined by the number of people being involved with developing or maintaining the respective entity. Although this is a rather coarse measure, it is directly observable. Knowing that the number of people to be productively employed in solving a given software development task can be varied only within certain limits [31] adds sufficient credibility to this measure as long as one may assume that, for good engineering reasons, the real observable exemplars are of about the size and complexity to be handled by an adequately skilled person or by an adequately composed group of software engineers.

For the sake of differentiating strata of software evolution the following categories are proposed:

- module,
- design unit,
- architecture,
- system,
- system-of-systems.

Readers might note that these categories do not define a linear order. Notably, architecture is a sidestep on a size dimension one might conceptualise between module and system-of-systems. However, the abundant work on software architecture and not least the arguments raised in [27] justify to distinguish at the system level between the system in its entirety and those engineering key decisions and ‘load bearing walls’ [32] that define its architecture. On the basis of Hypothesis 2 proposed above, these five categories are characterised as follows:

4.4.1 Module

A module is a unit of work produced or maintained by an individual programmer. Thus, the information content of a module is within the limits of this programmer’s intellectual
span. Therefore, evolution on the module level need not be a smooth process at all. If some external technological change requires a module to be changed, this change may be radical. The module may even be replaced by a completely new one. The only (evolutionary) requirement is that the new module adheres to the interfaces of the environment it is to be placed into.

The feasibility of module replacement as evolution strategy does not only follow from the established teachings on information hiding. It is also a strategy in development-with-reuse [33] and got new impetus from COTS-based development [34].

One should note that both development-with-reuse and COTS-based development might yield good arguments that neither size nor any other complexity metric is used directly in the definition of the module given above. If, for whatever reason, the developer feels sufficiently at home with the component to integrate it in her/his software and if maintenance could be done in a way that this component is not opened up by the maintainer but rather replaced in its entirety by another component satisfying the new specification, the definition given above is satisfied for this particular environment.

Perceived independently though, this component might be classified as subsystem or even as a system on its own. A classical example of software that is a complex system on its own but can be seen in a particular system-context just as a module is a database management system. Given clean interfaces, it is quite feasible that a single individual replaces the currently used relational DBMS by the relational DBMS of another vendor, even if both DBMS’ are highly complex systems on their own, developed by large structured groups. It is possible to perceive this software just as a module in the context of an application system, since relational theory in conjunction with the respective DBMS’s description allow the maintenance programmer to abstract from most of the details contained in these software entities. The information that needs to be kept mentally active at a time is sufficiently small for a single individual to cope with. This example can be extended to other situations, where a COTS-based developer does not need to see all the details. Sufficient information can be presented at a higher level of abstraction to trust that the component serves the functionality and role needed. The particular representational form of this information is of secondary concern as long as it allows for a trustworthy and concise specification.

4.4.2 Design Unit

The term design unit was introduced in the context of work on software reusability to denote an entity more complex than a module, but falling short of the properties of a system or of a fully operational subsystem [35]. While a design unit might not have all the closure properties one assumes when referring to a subsystem, it is a component big enough to warrant some kind of formal design, but small enough that this design can follow a single mastermind.

Thus, a design unit is a component consisting of several interacting subcomponents (modules, classes, procedures) that interact among themselves to achieve a common purpose in the context of a system. This description does neither focus on a particular representational form nor on a particular stage in the development process. The component constituting a design unit can be represented in the form of code as well as in the form of a formal design with all associated additional documentation and test-suits. But while the closure properties of a module (e.g. encapsulation of a distinct portion of
the state space controlled by the system) make modules suitable candidates as reusable software components, this requirement is not necessarily given for a design unit. It might be a complex object and thus reusable, but it might also be just a data-flow connected portion in the afferent or efferent part of a conventional system [36, 37]. There might be a specific small, single-minded team responsible for maintaining this part and for evolving it according to change requests against the system, even if this part is neither general enough nor sufficiently self-contained to qualify as reusable component. The design effort needed for such components is justified for either one of the following two reasons:

- The design unit is developed by a team and this team needs a documented design to allocate work-packages and to define interfaces.
- The component is developed by a single individual who needs an extensive period of time. Then, formal design is needed by this person in order to stick to his or her own interface agreements, agreements that might otherwise be challenged by fading memory.

In short, documented design is needed as a communication device (information), because on the code level, the volume of a design unit would exceed an individual’s intellectual span. On the more abstract and, therefore, more compact design level though, a single individual can oversee the various decisions that this design incorporates.

With design units, one should refrain from unconstrained revolutionary modification since the unit is too big for an individual to change it. However, evolutionary bursts might be observed because in a concerted effort, a group will still be able to perform arbitrarily radical changes. These ‘arbitrarily radical changes’ will be limited by the design unit’s environment. Within limits of arbitration, one might either radically change the design units internals but keep its interface to the rest of the system more or less unchanged (that is interface changes are limited to the changes directly driven by internal modifications) or the designer’s intellectual capability might be distributed adequately among changes to be coordinated with others and changes under the design owner’s independent authority.

4.4.3 Architecture

*With architecture, one refers to the system’s skeletal structure, that is to its ‘load bearing walls’ [32, 38]. This definition does not depend on a specific architectural phase in the development process [39] nor on some specific language or notation [40]. Whether this architecture is described in terms of some modern architectural description language or whether it is only implicit, to be potentially recovered by some re-architecting venture, is relatively unimportant for this discussion. Architecture here refers to key structural properties observable in the actual system. These structural properties, however neat or ugly they might be, are present in any system at any time. They matter to the extent that many subordinate decisions depend on the particular nature of such ‘load bearing walls’. It has to be understood, though, that architecture does not need to be explicitly present (with physical artefacts one would say ‘materialised’) in code. Software development seen as a stepwise decision process (c.f. [31]) requires decisions that have scaffolding purpose only. Their effects might remain only implicit in the final system. Perry and Wolf require that architectural descriptions also contain rationale among the central aspects [32]. This is to ensure that information of structural importance for the overall system is not forgotten, even if it remains only implicit in the final code.*
With software like with houses, you do not remove or arbitrarily modify a load-bearing wall without either taking precautions for everything resting on this wall or being severely punished by breakdowns or costly repair operations thereafter. Hence, architectures remain relatively stable over a long period of time and evolution. If architectural changes take place, they will be rather in the form of extensions. As a consequence, hybrid structures, comparable to what can be seen in many European churches, may result. There, generations of builders have made extensions, some of them being rather square to the initial architect’s plans. However, even when parts are torn down and replaced by newer realisations, the original mastermind and thus the basic style used in the initial construction will remain noticeable over centuries. With software systems, the centuries considered as yardstick for physical buildings might pass in quick motion during decades or even during biennia. All other aspects of the analogy remain valid though.

Therefore, normal architectural changes will be evolutionary. Over time, some spikes of activity will result, though, when somebody decides to rearchitect the system. However, such evolutionary spikes do not occur too often during a system’s lifetime. The degree of aging [3] due to regular maintenance operations will largely determine whether and when such evolutionary spikes will happen in a system’s architecture. Thus, within limits, technical arguments will be the key driver for architectural evolution.

4.4.4 System

System, in this context, refers not just to a skeleton but to the complete (delivered and operational) software in its entirety with all kind of documentation shipped with this system or kept somewhere else to support future maintenance operations or related questions. This entirety obviously consists of lower granularity entities (design units, modules). However, because of the diseconomies of scale in software engineering, systems need to be discussed separately. Moreover, the system is not only the unit produced by an organisation, it is also the unit of presenting a product to the customers. Hence, on this level the customers’ ability to absorb changes will be the delimiting factor for system-level changes. This suggests a pattern like the one observed by Antón and Potts on the feature level. However, before reaching a conclusion too quickly, the structure of the user base has to be considered.

A telephonic system has a large community of users with every user requesting more or less the same service. Hence, to model thousands of users by one single representative seems to be legitimate. In corporate environments though, one is usually confronted with different user groups sharing needs within the group but renouncing common needs on all other users. In this case, one can assume intellectual limits of users to serve as limiting factors for evolutionary speed only if the different user groups can be considered as largely interdependent. Thus, the flow (and nature) of user-driven change requests depends on the organisational environment of the system. Seen in conjunction with Lehman’s ‘constant effort’–law [26], the actually observable evolution strategy results from an interaction between the relatively constant workforce of system maintenance and the more erratically arriving change requests from users or their representatives.

4.4.5 System-of-Systems

The concept of systems-of-systems was introduced in Zemel and Rossak’s definition of mega-systems [41, 42]. They identified a system-of-systems as a particular form of a
conglomerate mega-system. Such huge agglomerations are suitable for reflecting on evolution and evolvability at instances where complete systems are to be integrated.

Integration of highly complex systems happens usually in the context of mergers and acquisitions when already relatively huge information systems need to be integrated. This definitely causes a very particular challenge. The challenge is not only due to difficulties in the integration of software per se; the challenge also involves integrating the data repositories accumulated at the organisation(s) in question as well. Thus, a distinction between software proper and the data processed by this software becomes at least at this level of complexity and size inadequate when system consolidation is aimed for. This gives further evidence to the information-based definition of software proposed earlier. In such a consolidation process and in any evolution process to be followed thereafter, not only will convincing system technicians be the limiting factor. It might be as difficult to convince application experts who used the system for years that a now feature, proposed by somebody outside of their peer group, is a valuable feature indeed. At least initially, they might rather consider it a bug, prohibiting them to continue operations with some workaround they might have discovered in the mean time.

Thus, at least at this stratum, Lehman’s dictum [43] that, once a system exceeds a certain size, it is no longer the product manager who controls system evolution finds its justification. He proposed that it is rather feedback forces of system evolution that are controlling the product manager. Evolution finds its limitation by the extent a system’s (social) environment is ready to tolerate (and on the same token to create) change. These considerations call for a change in the yardstick. What was initially introduced as the intellectual span of an individual now becomes the intellectual span of an organisation or society. This will be larger than an individual’s span, but by all means far less than the sum of the spans of the individuals concerned. Results such as those of Fischer, pointing out that individual users apply only a very limited amount of features presented by very complex systems, can be used as an argument in support of a distinction between the change requests voiced by a user community and change tolerance accepted by individual users [44].

This is not the place to speculate on a precise metric of such a combined span. It is fair to assume, however, that it is indirectly bound by the information processing capacity of the individuals concerned. Therefore, the closer the environment of such a system-of-systems has interlinked itself, the lesser is its capacity to tolerate big leaps in system evolution.

4.4.6 Discussion

When focusing on large entities, a distinction between the system in its entirety and those aspects of the system one might consider to be its backbone has been proposed. Again,
this distinction does not refer to a specific representational form. The distinction between the system’s architecture and the system’s final realisation in the form of executable code seems warranted, though, at least for the reason that changing X lines of code is (should be!) quite a different activity, if this causes a modification in the application’s surface structure or if the modification addresses the system’s architectural core.

Organisationally, an entity is at the system stratum if it exceeds the size and complexity to be reliably handled by an individual group. Thus, the system- and architecture stratum refer to something that needs a levelled organisation for building and maintaining it (c.f. the definition of ‘large program’ as one requiring ‘an organisation of at least two levels of management for its development or maintenance’ in [45]). However, to distinguish it from systems-of-systems, a system’s high-level conceptualisation might still follow a single masterminded plan. The importance of such a plan and the mastermind behind it influence the nature of system evolution [27]. As long as the mastermind behind the development can control changes and additions to an already operational product such that the system’s initial architecture is preserved, the system is in an offensive evolution stage. Once this person controlling development and progressive evolution has left, the system enters a rather defensive servicing stage. The servicing stage continues until system support finally comes to a halt and the system is phased out, possibly replaced and closed down.

Not least with reference to the above statements does it seem necessary to state that scientific discussion of evolution, notably the empirical work of Lehman, started with entities in mind that are referred to here as systems. However, over the years people used the word evolution to refer simply to the phenomenon of enduring change, disregarding the size or nature of the software artefact. Hence, this chapter also discusses these smaller granules for which the laws of software evolution were originally not defined and where they will not apply to their full extent.

On the system-of-systems stratum one has to recognise though that the usual viewpoint – reality is given and the software- and information system has to be a more or less a faithful reflection of this reality – changes. Of course, the relationship between software and the reality it reflects has to be carefully maintained always. But with big systems, notably with systems-of-systems, the potential of prescriptive power will change. Whenever it is easier to change the organisation (the real world reflected in a software system) than changing the system, the organisation might follow what the software system prescribes. If this is unacceptable, either the software system has to die (thus, something of a revolutionary nature will take place in the software) or even the organisation might perish because of its inability to co-evolve with its environment.

4.5 Approaches to (R-)evolve
As argued in the previous paragraph, the nature of evolution depends on the nature of evolution drivers. These, in turn, depend on the scale of the artefact under consideration. Arriving from these general considerations to concrete hints for ways to cope with evolution, one has to consider other factors too. Some of them are environmental, such as the volatility of requirements. While the considerations mentioned in the above paragraph put constraints on this volatility, volatility can still vary within a considerable bandwidth. This has to be taken care of. Other factors are system-immanent.

The most important among the system-immanent factors is whether software is changed off-line or in a running system.
(a) In most situations, software is changed and tested off-line. Only after it has been tested on the development machine with specially prepared test data, will it be transferred to the hardware environment where real user data is processed. To ensure consistency, this roll-out is made at a time when the operational system can be brought to a brief temporary halt. Then it is reinitialised with its previous (possibly transformed or adjusted) state. In this case, the evolution of the software in the proper sense of this word (i.e. change of the shell encapsulating the data describing relevant parts of the reality reflected) happens in a situation somehow resembling an artificial laboratory situation. Therefore, one might refer to it as \textit{in vitro} evolution.

(b) In contrast to this comparatively keen situation, certain systems, notably real-time systems, cannot be brought to a complete halt when switching to a new version. Evolution of such a system has to take place during operation where not only the software proper changes but also the data this software operates on has to be changed in sync. This case has to be handled with much more care, since it is obviously more complex and the proper evolutionary steps cannot be tested off-line [46]. To contrast this situation from the one described in (a), one might call it \textit{in vivo} evolution.

In order not to lose track, the rest of the chapter will abstract from these considerations and concentrate on the five categories presented in the previous section. Without attempting to achieve completeness, some strategies to cope with evolution are mentioned for demonstration purposes.

4.5.1 \textit{Changes in Modules}

On the module stratum, clean interfaces and adherence to classical design principles will make modules robust against evolution in other parts of the system. Information hiding and, more general, design for reuse will be adequate strategies. With design for reuse or design for component-based development, planning for the possibility of revolutionary change by completely replacing a module with another one is specifically highlighted.

Thus, the overall design of the system has to be in such a way that with individual components, even revolutionary change cannot radically shake the system. Thus, the strategy to cope with evolution is not inward directed, considering the component itself. It rather considers the relationship between the component and its software-technical environment. Thus, strong cohesion and consequently minimal coupling have to be interpreted such that the individual module provides single-minded semantics and has a clear interface. Obviously, object-orientation provides an important set of concepts supporting this goal.

However, not only reuse or object-orientation are to be considered in this context. Developments in the realm of high-level programming languages also matter in this respect. On a keystroke-level, changing a symbol is always just of unitary nature in a textual representation. In terms of informational content, the complexity of the change will depend on the complexity of the semantics attached to this symbol. In terms of intellectual span, finally, one has to consider how well understood the abstraction attached to this symbol (set of symbols) is, to assess whether the textually observable change causes high or low intellectual effort. As evidence for this claim, one might refer either to Halstead’s work or to the various language calibrated conversion tables from function points [47] or modified versions of this concept to either line of code or to effort [5].
4.5.2 Modifying Design Units

With design units, precautions for evolution have to be such that change requests from the design level can be easily accounted for at the implementation level. From this stratum onwards, one has to consider that change needs to be propagated from the complex stratum down to lower (closer to implementation) levels. Thus, what might be bursts of evolution for a design unit might be revolutionary change for some of the modules constituting the respective design unit. Hence, loose coupling of the individual components is one key strategy, state separation is another one. Thus, classical design wisdom leading finally to strategies recommended as design for reuse [33] will serve as a basis for economical evolvability. Further, at this level, one should already consider something like volatility management, that is, a well-defined strategy of how to allocate those parts of the system that are most likely to change repeatedly over the system’s lifetime. This issue is too tricky and too domain dependent as to give a general recommendation. Single-minded components versus encapsulation of volatility laden aspects might be conflicting design strategies that can only be resolved considering a more global perspective on the design space.

4.5.3 Evolution on the Architectural Level

On the architecture stratum, the distinction between components and connectors (with connectors being special purpose communication components) can at first be seen as the distinction between living rooms and working rooms versus hallways and walls. With the walls, a further distinction has to be made. They can be either load-bearing walls of the system or they might just highlight and assure separation of concern, being software analogues to easily movable Japanese screens. This distinction has to be consciously made and clearly documented. Thus, design for volatility, a side consideration at the level of design units, becomes a major concern in architectural design.

One can witness this design for volatility on the architecture level also when considering the discussions in the intersection of research on software architecture with research on product lines [48]. With the definition of a product line, the system structure is basically partitioned into those parts that are robust with respect to varying user populations and those parts where substantial variations between different subdomains of a common application domain occur. Thus, the homogeneity of the user population (application subdomain) is considered as a factor limiting the volatility and, therefore, the need and speed of evolution. A good product line architecture will consider partitioning of the overall market. When each instance of the product line is targeted towards a homogeneous subdomain of this market, product evolution can be controlled more easily. Thus, market segmentation determines product segmentation. The argument that establishing product lines yields a high reuse potential for the (sub)systems has been put forward in Northrop’s keynote to ICSE 01 [49].

4.5.4 System-Level Evolution

At the system level, one has to consider again that a software system is the realisation of reflections, charted coarsely at more abstract levels during initial steps of the development process. These high-level descriptions are progressively refined and augmented with implementation-relevant information till development reaches the fine granular, detailed
level of executable code. Thus, what is charted on a high level in such a way that some highly skilled individual(s) can intellectually capture an all-encompassing perspective, exceeds this span when all items are represented at an executable level. Hence, various evolution support strategies can be followed. These strategies, however, will be only supplementary to the strategies mentioned on the design unit and on the architecture level.

Given that the system is small enough, using frameworks can be mentioned as a strategy. Frameworks yield some standardisation, a standardisation that will bear on the lower-level components. Thus, frameworks and patterns might be considered as matching pairs. But while patterns are a rather scale-independent concept, there will be limits of the size of systems, where frameworks can provide an adequate answer.

Another low-level idea that scales up after generalisation and re-transformation is parameterisation. Parameterisation is more powerful than its routine use for parameters in the data space. Procedural parameterisation does scale up to a certain degree, if adequate instrumentation assures security walls comparable to those that a strong type system establishes for conventional parameterisation. Attempts reported in [50] and [51] are initial steps in a direction that might be termed meta-parameterisation.

4.5.5 Evolution of Systems-of-Systems

On the system-of-systems stratum, one can observe that change is of a dual nature. To some extent, these large entities will be constantly subjected to gradual change. Individual change drivers will be mainly related to individual systems contained in this conglomerate. Therefore, strategies of confinement are important. However, radical changes within the reality these systems are reflecting, will come seemingly at random points in time. Mergers and acquisitions might be considered as examples of such radical changes in the real world.

Accepting this duality, one has to also accept that it will be insufficient to allocate evolution support only within the system. One has to take care of evolvability on a more general, that is, on a strategic level to prepare these mega-systems for eventual radical changes.

Standardisation might be the strongest mechanism in this context. It can be conceived as a strategy to allow modular, that is, revolutionary evolution behaviour of components even if their internal complexity is beyond the grasp of an individual. However, standardisation is not and cannot be on the scale of mega-systems. Hence, loose and indirect coupling of those parts that lend themselves at least to some extent to standardisation might be the other key strategy to be followed in preparing mega-systems for changes happening in reality. Notably for changes that cannot be accommodated by small incremental steps within the system, this might be a worthwhile strategy. However, this advice contains a certain contradiction in itself. On the one hand, mega-systems are so big that their mere size prohibits radical change. On the other hand, they are so big that certain changes, whatever the architecture of the system might be, will be very big too. Perhaps the point where this drive for evolution and blockage against evolution overlap will eventually define the limit of growth for mega-systems.

On a smaller scale, the phenomenon that the system defines reality and not vice versa happens already when organisations put complex ERP-systems into operation. Such systems do not provide standardisation in the strict sense of this term. The reverse effects these systems have as change drivers seem interesting though, since the naïve statement that software has to keep up with reality does no longer apply. In this context, one is reminded that what seems as external change driver on first glance might not only have
sources independent from the software system under consideration. Feedback loops are operational, such that a software system, once fielded in an environment causes changes in this environment, which lead to changes in the requirements to be covered by the respective system [14].

4.6 An Example

In this section, we discuss the aforementioned issues through an example: the evolution of the SESAM/AMEISE system, a teaching and research environment to practice software project management [52]. SESAM\(^4\) simulates software development according to a process model comprising continuous as well as discrete process elements. It constitutes the core of the AMEISE system\(^5\) [53] developed by a consortium of three Austrian universities. Taken together, SESAM/AMEISE’s lifetime extends over 15 years.

In the sequel, a selection of evolutionary aspects of this system is discussed along the strata defined in Section 4.4. Placing interesting changes to the system into historical and organisational background should highlight the evolutionary aspect and show the broad picture. But software evolution, perceived to be continuous from a long-term perspective, happens in discrete steps of releases or updates to a configuration [54]. Thus, any single-step perspective will necessarily lead to the impression that the instance discussed just amounts to a more or less complex maintenance operation. The narrower the focus of observation becomes, the more this criticism applies.

4.6.1 A System-of-Systems?

Considering its size of roughly 150,000 LOC, SESAM/AMEISE certainly does not qualify as a system-of-systems. However, the fact that the development has been distributed over different groups (two in the same city at different institutions, the remaining about a one-day train ride apart) with different backgrounds (Ada versus Java development environments) and two different chief-engineers certainly determined the trail of evolution. For example, the original developers would certainly have implemented AMEISE’s multi-user support in the SESAM-core. The AMEISE-group avoided touching this Ada-core as much as it could. At the expense of performance, it achieved multi-user functionality by saving the sizable state space of individual users and reloading it into the wrapped SESAM-core. This costly operation was justified by both, lack of Ada-experts and by the need to keep the various instances of the state space in a persistent database, for various user support features they built.

Likewise, requirement modifications were made in the light of system properties. An AMEISE needs statement called for developing ‘simpler models to allow using the system in introductory classes’. This would require empirical work and revalidation of such small-scale models\(^6\). As it was unclear whether the associated costs are warranted by didactical gains, it was decided to implement support features that allow instructors to vary the

---

4 SESAM (Software Engineering Simulated by Animated Models) can be obtained via //www.informatik.uni-stuttgart.de/se/research/sesam/index_e.html.

5 The extensions and modifications made in the AMEISE project (A Media Education Initiative for Software Engineering) can be perused at//ameise.uni-klu.ac.at. AMEISE has been funded by bm:bwk under NML-1/77.

6 The current process model has been defined and validated for developments in the range of 200 to 1000 adjusted function points (AFPs). It consists of about 25,000 LOC in a proprietary language.
complexity of the assignment by varying the magnitude of support mechanisms provided to students.

While trade-offs of risky requirements against safe flexibility can be made at any stratum, they are most likely to happen in complex system or development situations. The example also represents cases where evolving requirements leave the range foreseen at system conception. Instead of linearly extending the range of some parameters, system designers open other avenues by providing features that recapture requirements within the range the system has originally been designed for.

4.6.2 System-Level Changes

The SESAM history started in 1990 at the University of Stuttgart. Initially, a Smalltalk-80/Visual Works 2.0 prototype helped to shape the basic ideas of teaching software project management by a quantitative simulation system. In 1997, a complete re-implementation in Ada95 has been undertaken under the direction of the original chief architect, comprising currently over 75 KLOC. In 2001, a consortium of Austrian universities decided to build its AMEISE tool for SE-project management following the concepts of SESAM and building directly upon SESAM. Among other aspects, AMEISE should provide a new user interface, group support and various features deemed interesting for didactical reasons. Currently, AMEISE’s extensions encompass 72 KLOC Java code.

Building on top of SESAM allowed AMEISE to become already operational in spring 2002. Since then, it has been substantially extended till, eventually, limits were reached that required to shift effort from feature extension to internal purification of the system.

In AMEISE, multi-user functionality was needed. Some further new requirements were as follows:

- Students should be able to operate AMEISE without direct instructor supervision.
- AMEISE should become operational via a web interface.
- Support for student- and class management should be provided.

This led to reconsiderations at the architecture level and required a clear distinction between the legacy system and the new one.

A client-server architecture with a completely new student interface, a newly built instructor interface, a data repository and some load-balancing device had to be built. On the other hand, the SESAM-user interface had to be untangled from the system. Because of the ingenuity of the original design, this could be carried out in a straightforward manner. However, as call-backs were handled differently from straight inputs, some cutting and gluing was necessary. Likewise, changes in the dictionary of SESAM’s pseudonatural language interface became necessary.

For consistency reasons AMEISE features its own graphical user interface as well as SESAM’s traditional textual user-interface, relying heavily on call-backs. Performance monitoring has shown that providing for those call-backs is quite costly and user supervision demonstrated that the textual user-interface is hardly ever used. Sacrificing the textual interface will reduce communication complexity. This allows for scrapping a sizable portion of interface code and structural improvements within the client. Hence, it will be a forthcoming step in system-level evolution.
4.6.3 Architectural Decisions

Architecture can be discussed on several levels. On the domain-level, SESAM provides a clear separation into

- the model, containing all entities (persons, documents, activities) relevant in a software development project, and rules establishing qualitative and quantitative relationships among them;
- the simulator, executing commands of the user acting as project manager, and
- the interface(s) for students and for instructors operating the system.

This separation proved helpful in early prototyping and remained stable for the single-user architecture of SESAM as well as for the multi-user AMEISE system.

Technically, AMEISE followed its own architectural decisions, which can be seen from two perspectives:

- A client-server architecture, allows $k$ independent clients (each one on its own machine) to connect via one (or more) load-balancing components to $n$ wrapped SESAM cores (on $n$ machines) cooperating with one database server. This architecture fully exploits SESAM’s simulation and modelling functionality while being free in terms of hardware-based and AMEISE’s group support features. Further, it provides flexibility concerning performance aspects.
- A data model has been designed as semantic architectural backbone. While required for student and class management, it had become the focal point for functional system evolution. Besides the wrapped SESAM cores and the load balancer, it is the only logically central feature. Its main sections, class management, user management, model management, support-features management, and user-run management allow individual AMEISE features to keep relatively tight and clean interfaces and to be (with few exceptions) memory less. So far, the data model became the key evolution facilitator.

Up to now, these architectural decisions have withstood the proof of concept by requests for several extensions of support features during AMEISE’s two years of operation. Most of these extensions were feature extensions. Multi-lingual support for German and English could also be easily integrated. Handling French as a third language caused ripples though. To allow for handling of accents, constraints in the parser had to be weakened.

4.6.4 Design Units

Design units, as defined in Section 4.4.2 are relatively closed portions of a system, big enough to require a team effort to develop them. Thus, they require some documented design. In AMEISE, the database, the load balancer, and the user-client are such units. Various administrative clients or monitoring devices might also be seen as software design units. Design units that rather have the character of data are the simulation model (a set of several hundred rules), the dictionary and the explanation component with its specific aid tables. Among those, the SESAM-core as well as the simulation model are under the strict authority of the group in Stuttgart. The user-client contains modules developed by two different sites in Klagenfurt. For post-development work, it has been placed under the control of the Technikum Klagenfurt. The rest of the AMEISE system is under the
responsibility of the Klagenfurt University. The dictionary plays a special role, as it is a cornerstone between model, user interface and explanation component. Hence, dictionary modifications require consensus among all partners and utmost care in version control.

From an evolution perspective, the client might be of particular interest. Its internal architecture is a simple façade-like structure [55]. Its only interface to the rest of the system is via the load balancer. Its components though, encompassing the GUI-based user-interface, the textual user-interface, and graphical user-interfaces subordinate to various explanatory components are rather complex. Functional evolution of the system is of course always reflected in the client. Its structure allowed for easy integration of these extensions though.

Like the original SESAM user-interface, the client is stateless. However, to accommodate the full functionality of the pseudonatural language legacy interface, a number of bookkeeping operations are necessary to resolve ambiguities by call-backs. The graphical user-interface allows sending only complete and syntactically correct commands. Hence, dropping the communication intensive textual interface will allow structural improvements also in several other user-related client components. After these forthcoming structural cleanups, only the ‘friendly peer’, an agent observing the last user actions in a window of limited depth, keeps local memory. It remains to be seen how much its communication protocol with the database at start-up and shut-down can be further streamlined within the overall protocol simplification.

The explanation component was specifically designed to allow for continuous evolution. To accommodate changes in or replacements of simulation models, its design follows the interpreter pattern [56] in a two-level recursive manner. On the basis of the current user state, it constructs a set of SQL queries into tables monitoring the user’s actions and into specific aid tables. The results of these queries are used to construct another SQL query. Out of these results, the message displayed to the user is composed. This general principle allows experts of a different kind (instructor, model-builder) to build new or change existing explanations of model effects by just modifying some database entries and by letting the system worry that the individual changes are properly reflected on the user-interface. This design decision is in line with the overall AMEISE architecture of having the database as the evolutionary backbone of the system. It allowed shifting design-unit (or even system level) evolution to the level of strictly confined modular changes. This has also implications on the organisational level. While such changes definitely involve client, server, and model (i.e. organisationally three teams), this specific decision is one aspect that allows passing the maintenance responsibility for the client to one team only, though the modules it contains were built by two distinct teams. As long as changes of the explanation component, be it corrective maintenance or be it extensions in explanatory power, stayed within the given (textual and graphical) syntactical framework, the concept kept up with the requirements. Accommodating French, though, necessitated levying the restriction that the dictionary’s parser accepts only strict ASCII code.

The database schema itself is also considered as a design unit. As a component of architectural significance, it has been designed by a group of three with a single mastermind and was heavily reviewed before being implemented. Considering the individual relations as modules it has been heavily revised on the module level since. Apart from actually implementing the multi-lingual aspects which where architecturally foreseen though, the
changes required so far were confined to only few interrelated tables each. Because of the
logical centrality of the database, its proper versioning needs to be done with utmost care.

To summarise, the specific aspects mentioned rest on the basis of organisational and
technical decisions that allowed not only letting the system grow in the direction originally
foreseen, but also to accommodate requirements originally not anticipated. Examples of
those are the French version as well as various requirements resulting from users and from
developers. An example, where user and developer requirements could be accommodated
by a common feature was the user requirement to allow students to outsource certain
development tasks and the developer’s need to develop a test-bed for performance tests.
The ‘external software house’—feature satisfies both. Reaping such benefits is possible
only when change management from requirements onwards till feature integration allows
planning. One might term this evolution control.

In terms of evolution control, the explanation component’s history is worth consider-
ing. Nothing comparable was foreseen in SESAM. AMEISE introduced the concept and
adopted the design mentioned above. As this proved useful, a tool was developed that
generates the database entries from higher-level descriptions. But there is still no direct
coupling to the model driving simulations. Hence, on the very long term agenda is the
development of a component that allows creating the chunked SQL-scripts and explanatory
hints of the explanation component directly from the rules defined in the model or concur-
rently with the definition of these rules. This will raise consistency and ease development
of new applications. But the respective decision has to be taken at the architecture level.

4.6.5 Modules

Module evolution took place in various forms so far. In most cases evolution involved
incremental changes in both, requirements and implementation. To highlight the discussion
of Sections 4.4.1 and 4.5.1, three exemplary cases are mentioned though.

- **Behaviour preserving revolutionary change**: The SESAM Tcl/Tk user interface had
  been completely replaced by a Java interface initially. This allowed to experiment
  with the initial version of a simple client-server architecture consisting of wrapped
  SESAM-core, simple DB-structure for state dismemberment, load balancer and simple
  user interface. Later, this simple textual interface had been replaced by a client sys-
  tem, hosting this interface and various prototypes of support features. This eventually
evolved to the current client architecture hosting various components that are on the
module or the design-unit level. Thus, user functionality was fully preserved through-
out various versions of the system while technically this functionality was provided by
three completely different generations of components.

- **Technical revolution with evolutionary behaviour**: The transition from the Tcl/Tk user-
  interface to the Java Interface was certainly a revolutionary change. Replacing the
textual user-interface by an interface based on selection from dynamically created
menus could also be considered revolutionary on a technical level. However, for consist-
ency reasons, the textual interface remains available and the composed text is visible
to the user. Both modules\(^7\) reside currently in the user-client and either one can be

\(^7\) The textual user-interface and the graphical user-interface are modules in the sense of Section 4.4.1 as they
can be overseen by a single individual. In object-oriented terminology, they might be called packages consisting
of several classes.
selected by means of a pull-down menu. Hence, from a user’s perspective, the change was just evolutionary.

- **Revolutionary behaviour change by technical evolution:** Contrary to these changes, development of a new model (e.g., a maintenance model) might seem to be a radical change from the user’s perspective. Technically though, it required just some new entries in the dictionary module (evolution), new versions of some tables in the specific aid portion of the database and of course a completely new model with new rules and new quantitative parameters. The remainder of the system is sufficiently parametric to cope with these changes though.

### 4.6.6 Discussion

The SESAM/AMEISE system has evolved over 15 years. With currently about 150 KLOC programs, integrated components such as a DBMS, and an application model comprising 24 KLOC it is beyond the size a single individual can oversee completely. As a living system, it is still growing. Because of legacy concerns, several evolutionary decisions might not qualify as standard textbook material. Had the AMEISE-consortium produced a SESAM-3 system from scratch, quite a number of decisions would probably have been taken in a more appealing way. However, this was not the situation. The AMEISE team was more than happy to build its extensions on an already sizable legacy system. In (re-)defining the architecture of the system, AMEISE designers had to consider that emerging design units and modules are small enough to be completed in term projects of groups of students or within MA-theses of individual students. Further, design units had to be small enough to be overseen by a single supervisor. Therefore, decisions were taken in the light of available staffing, available competence and given organisational dispersion.

As with extensions to physical buildings, architectural and design decisions had to be made carefully considering legacy decisions. Likewise, when new things were added, one had to consider how to interface the old with the new. Interfacing between the old and the new involves turning down parts of the existing construction and rebuilding them in a new style and it implies compromises and respect for organisational and intellectual constraints. This applies equally to evolving a successful software system as it does to a valuable ancient mansion.

### 4.7 Summary

The chapter departs from the perspective that software is always a reflection of some reality and as such it rests on the statement that software, in its substance and in its effects, is structured information. On this basis, a bridge is established between evolution and human’s information processing capability or evolution and society’s information processing capability respectively. Building on these assumptions, a stratification of software according to the information processing capability of individuals, groups and organisations is given. The strata described are at the levels of system-of-systems, system, architecture, design unit and module.

The discussion of concepts and tools supporting evolution at these levels shows that it will be futile to aim for the ideal overall evolution support strategy. On the contrary, a spectrum of evolution support mechanisms, each mechanism adequately scoped, will be needed to solve the problem at a given level of system size or system complexity.
References


