Software Evolution and Software Evolvability

Stephen Cook, He Ji and Rachel Harrison
University of Reading, UK
{ S.C.Cook, H.Ji, Rachel.Harrison}@reading.ac.uk

Abstract

In addition to managing day-to-day maintenance, information system managers need to be able to predict and plan the longer-term evolution of software systems on an objective, quantified basis. Currently this is a difficult task, particularly when an information system is deeply embedded in an organisation’s business processes. Although previous work, notably by Lehman et al., has shown that the evolution of such embedded software systems can be characterised as a complex feedback process, more work is needed to solve the practical problems of managing software evolution.

In this paper we present an approach to understanding software evolution that is based around the quantifiable concept of evolvability. This concept brings together the study of software product quality, the software evolution process, and their relationships with the organisational environment. We assess the opportunities for analysing and measuring evolvability at pre-design, architectural, and source code stages in the development of software.

We also describe work in progress on models of software evolution. We present a definition of software product evolvability in terms of a software quality model. We have also proposed a set of metrics derived from our model, which will enable objective assessments to be made of the evolutionary history and evolvability of software products. In order to validate these metrics we need a model of software evolution dynamics and we present some initial results on developing predictive models of software product evolution.

Keywords: software evolution, Lehman’s laws, quality models

1. Introduction

The recent preoccupation of the IT industry with Year 2000 issues has drawn attention to one extreme form of the software evolution problem: what are the characteristics of software that make it easier or harder to change a software system as requirements evolve? Can these characteristics be quantified and can we discover any correlations between them and measures of the evolution of the software product itself? In this paper we present our approach to these issues, based on applying the techniques of empirical software engineering to the concept of software evolvability.

Lehman and Belady first applied the term evolution to software in the 1970s and since then most investigators have used that term to refer to the long, broad view of change in software systems. More precisely, we can say that software evolution is the sequence of changes to a software system over its lifetime; it encompasses both development and maintenance. Within this broad scope a variety of approaches has been followed, ranging from detailed design prescriptions to applications of systems theory. In this paper we use the concept of evolvability to bring together some of these diverse elements and show how they can contribute to a quantitative approach to studying software evolution.

As organisations become more dependent on software and the pace of both business and IT change increases, so the effective management of software evolution becomes more critical to an organisation’s success. An important contribution to enabling that will be to demonstrate that software evolution and its drivers can be measured, and the impact on software evolution of changes in the drivers can be predicted. Our DESEL (Designing for Ease of System Evolution) project aims to achieve this by studying real-world software systems in collaboration with academic and industrial partners.

In section 2 we review the principal previous contributions to empirical approaches to software evolution. We then develop the concept of evolvability and examine its impact on different aspects of a software system. In particular, we present some work in progress on assessing the evolvability of software architectures through the use of Architectural Description Languages.

Section 4 examines in more detail how evolvability can be measured and provides examples of metrics derived from ISO/IEC standards. To be useful these metrics must be validated and we present some work in progress on developing dynamic models of software evolution for prediction of change impact.

Finally, section 5 briefly summarises our conclusions and indicates the direction of our future work.
2. Related work

This section reviews related work in four areas of empirical studies of software evolution.

2.1. Lehman’s laws of software evolution

The observation that software systems undergo continuing changes was first put forward by Belady and Lehman [4]. They termed this dynamic behaviour of software systems evolution and carried out empirical research on about 20 releases of the OS/360 operating system. The investigation led to five “laws” of software evolution: Continuing Change; Increasing Complexity; The Fundamental Law of Program Evolution; Conservation of Organizational Stability; and Conservation of Familiarity.

These laws were systematically studied by Yuen [40-42]. Several statistical methods were used to analyse different systems. One of the results of the investigation was that the characteristics observed for OS/360 did not necessarily hold for other systems, especially as the latter three laws are more based on the organizations than the software itself.

Kemerer and Slaughter [17] conducted an empirical study of software evolution using the systems of a large United States retailer. The investigation focused on the types of changes, costs and effort to evolve the software. It then analysed these factors from the point of view of Lehman’s laws. The authors detailed the analysis methods used in the study, such as comparing the time series and sequence analysis of data, as they believed that at this stage clarifying research methods is more important than obtaining a large volume of results.

More recently Lehman and his colleagues have begun new investigations into software evolution. The FEAST/1 project, beginning in 1996 and ending in 1998, aimed to construct black- and white-box models of software system evolution, with special attention to feedback phenomena. The results [23] of studying several data series from their industrial collaborators support, or at least do not contradict, the laws of software evolution formulated in the 1970s. Moreover, three new laws have been identified: Continuing Growth, Declining Quality and Feedback System. The ongoing project FEAST/2 focuses on control and exploitation of process behaviour. In [20], Lehman provided over 50 rules based on the eight laws, for application in process planning, management and implementation, and suggested tools to be developed to support these activities.

2.2. Software evolution and process improvement

As Lehman pointed out [22], their work represents that of a small group that views evolution as a noun and focuses on the nature of software evolution and the properties of the evolution phenomenon. Other groups of researchers view evolution as a verb. They focus on methods and tools to facilitate software evolution and the tasks that implement it. Many empirical studies from the latter group can be found in the literature of the software maintenance community. We do not try to give a complete review in this area, preferring instead to highlight some important points in the following paragraphs.

Kafura [16] studied the relationship between software complexity metrics and software maintenance. Two classes of complexity metrics, namely code metrics and structure metrics were used to quantify the complexity of three versions of a medium-size system. The author found that both classes of complexity grew when the system was maintained, which confirmed Lehman’s laws. The author also analysed changes in complexity at the procedure/module level, and claimed that extreme changes of complexity in one procedure/module may indicate possible flaws during the maintenance activity.

Basili et al. [1] examined 25 releases of 100 software systems to characterize the types of maintenance activities and to examine the effort distribution between them. For example, they found that corrective maintenance usually requires significant isolation activity, while enhancements required more time on inspection and certification. Effort for design, coding and unit testing were similar for the two types.

At Hewlett-Packard, Coleman et al. [5] applied two models of maintainability metrics to the daily development and maintenance processes. HPMAS (Hierarchical Multidimensional Assessment Model) [30] assesses the maintainability of software based on three dimensions: Control structure, Information structure and Typography, naming and commenting. The polynomial regression model uses a polynomial equation to express maintainability in terms of the associated metric attributes. Both models were calibrated against HP engineers’ subjective evaluation of 16 software systems. The models were used in monitoring maintenance activities, comparing software systems, and assisting the decision of buy-versus-build.

Schneidewind has been working closely with the NASA Space Shuttle project for many years. In one of his recent papers [34], a maintenance process was evaluated using reliability, risk and test metrics. The author claimed that the integration of product and process measurement serves the dual purpose of using metrics to assess and predict reliability and risk as well as to evaluate the maintenance process.

2.3. ISO software quality standards and metrics

The efforts of the ISO community have resulted in the publication of ISO/IEC 9126 Software product evaluation
3. Software evolvability

Our hypothesis is that software systems vary significantly in how easily they can be evolved to remain productive within a changing environment. We call this quality of software systems “evolvability”, defined as the capability of software products to be evolved to continue to serve its customer in a cost effective way.

The concept of evolvability brings together factors from three main areas: software product quality, software evolution processes, and the organisational environment in which the software is used. The sections below introduce the contribution which each area can make to understanding evolvability but in each case we believe that the software engineering community’s current understanding of this contribution is incomplete and largely qualitative. We draw attention to the current research directions which look most promising for deepening our understanding of evolvability and obtaining valid measures of it.

3.1. Software product quality

Some qualities of a software product affect how easily it can be changed in response to new or changed requirements. These qualities can have an impact at various levels of abstraction of a software system (conceptual, requirements, architectural, source code etc.) and at every stage of the software life-cycle.

3.1.1. Pre-design evolvability. Once a software product is in use, the more abstract forms of its representation (e.g. requirements documents) are usually less likely to change than its designs and implementations. However, when the goals or requirements of a software system do change, then the cost may be high because typically the changes will have to cascade down through all the intermediate layers of abstraction to the physical system [24].

Software engineers need to be able to measure the capability of software to withstand this kind of change. This assists the objective assessment of the lifetime costs and benefits of evolving software, and the identification of legacy situations, where a system or component is indispensable but can no longer be evolved to meet changing needs at economic cost. However, although extensive work has been done to understand the requirements engineering process, the need to continually evolve requirements has not received the same attention. Consequently, we are not yet able to say even whether it is feasible to measure the evolvability of a collection of requirements.
Figure 1  Alternative architectures

Figure 2  field_agent_A1 architecture specification
3.1.2. Architectural evolvability. At the high-level, architectural, design stage of the software process, we can be more confident that some measurements of the evolvability of the intended system can be made. In addition to measuring the evolvability of source code, it is highly desirable to be able to assess the evolvability of a software system’s architecture before any programs have been coded (or even designed). To minimise the lifetime costs of a software system, evolution pitfalls (i.e., design choices that are likely to constrain rather than enable evolution) and evolution hotspots (i.e., analysis and design patterns that are expected to change frequently [32, 33]) should be identified and mitigated at the earliest possible stage.

Architectural Description Languages (ADLs) [7, 8, 25-27, 29] can assist with this task. ADLs provide a more formal framework (compared to box-and-arrow diagrams) for specifying software architectures, comparing alternatives and measuring their properties.

To illustrate this use of ADLs, we present a very simple example in Figures 1-3 using Structural Architecture Description Language (SADL) [29]. It might be thought that this example is too trivial to be helpful but in fact it is based on a real world example.

This example describes a fragment of a multi-function work management system, representing part of a client application that is intended to be installed on personal devices (hand-held or in vehicles) that communicate with servers over a wireless network. Figure 1 shows two alternative architectures for the same functionality, namely, processing incoming messages (according to their type) from a receiver queue and appending outgoing messages to...
a transmitter queue. Architecture field_agent_A1 is monolithic, consisting of a single “black box” component. field_agent_A2 is more modular; each type of message is handled by a specialised module.

The SADL specifications are shown in Figures 2 and 3 respectively. Although SADL is rather terse (and we have omitted the details of constructs that are assumed to be imported from more generic architectural styles), we hope that readers will find that the combination of the diagrams and the textual specification makes clear the main differences and equivalences between the two architectures.

If we make the reasonable assumption that a likely evolution of this architecture will be to add more message types and functions to process them, then clearly A2 (Figure 3) is more evolvable than A1 (Figure 2) with respect to this anticipated change. Although A1 is smaller (in terms of components, connections, connectedness etc.) and superficially simpler, this makes it less adaptable than A2. If a new message type is added, then potentially all of A1 might have to be replaced because all the clauses in its specification depend on the type msg_type, which will be redefined by the change. In practice, the impact on any implementation would be difficult to assess from such a specification because it constrains its implementations so lightly. The impact of the same change on A2 is less dramatic and the changes to an implementation would be more localised and predictable.

Clearly, if we were able to make an objective appraisal of the comparative evolution risks and costs of these architectures, then A1 could be rejected at the high-level design stage. In the simplest cases, an informal diagramming approach might be sufficient to reach this kind of conclusion but the analysis of more complex architectures will benefit from the greater formality that an ADL provides. However, further work is needed to demonstrate that validated evolvability metrics can be derived from ADL specifications.

3.1.3. Detailed design and source code evolvability. If detailed designs or source code exist, then a variety of analysis approaches can be used. This allows potentially contrasting points of view to be evaluated which may generate new insights into different aspects of the evolvability of the system. Section 4 describes the set of metrics (and the quality model from which they are derived) that we are proposing for measuring evolvability at this level.

3.2. Software evolution processes

Many attempts to improve the software process are focussed on achieving repeatability. However, this manufacturing perspective may be less appropriate for software than for other engineering products, as software evolution is characterised by complex feedback and imprecise control mechanisms. Consequently, classical control theory may not be immediately relevant to achieving process improvement and more evolvable software. A focus on design activities [15] and the adaptability of the software process may be more likely to enhance the evolvability of a software system.

3.3. The organizational environment

One of the ways in which software differs from most engineering products is that it can include a model of the environment (i.e. the organisation) in which it operates. Lehman used this characteristic to distinguish the “E”-type (Embedded) programs that are of greatest concern [19]. However, the symbolic, drastically simplified models that are incorporated into software are not easily related to rich, depictive models of the real world. Consequently, the skills of information system engineers and business analysts are repeatedly required to mediate between these two kinds of models. Inevitably this contributes to delays and errors in keeping the models aligned especially when either (or both) changes rapidly or fundamentally. The consequences of misalignment between an organisation and its information systems are frequently observed but finding solutions that will enhance the evolvability of information systems remains the subject of active research.

For example, one approach has been to stress the value of co-evolution as a concept for understanding the dynamic relationships between business processes and information systems [28]. The co-evolution concept is taken from biology and refers to the complex process that arises when two or more interdependent species are evolving concurrently. The effect of this is that the evolution of each species partially determines the environmental changes that the other species are evolving in response to. Biological analogies for socio-technical processes are not always helpful but one situation where co-evolution might be a relevant model is where an organisation has chosen to build its information systems largely from COTS software products with minimal tailoring. In this situation, the alignment of business processes and information systems, in so far as it happens, is likely to be an incremental, ad hoc, bottom-up process which exhibits some of the characteristics of co-evolution.

Other investigators have examined the alignment of business and IT strategies and come to different conclusions. For example, when Sweeney and Bustard [37] investigated four organisations they found the usual discrepancies between business and IT strategies but also found that this created space for more flexible responses to rapid or unexpected change. They suggested that organisations could enhance their scope for flexibility by trying to decouple business strategies from IT systems, rather than pursue the unachievable goal of perfect alignment. They recommended that the information
system concept should be strengthened and made more autonomous by decoupling it from its supporting physical IT system, so that the information system becomes a technologically neutral description of the organisation’s information processes. This approach would appear to increase the scope for evolving information systems (by abstracting from a particular technological implementation) whilst maintaining the broadly hierarchical relationship between business strategy, information system and IT infrastructure.

We have briefly mentioned these contrasting approaches to make the point that there is still a great deal to be learned about how the evolvability of a software system is affected by relationships between the system and an organisation that uses it. In particular, further investigation is needed to discover which measurable attributes of these relationships are most critical to improving software evolvability.

4. Evolvability quality model

This section introduces some work in progress to place the concept of evolvability on a sound, empirically validated basis and presents our preliminary results of defining a quality model of evolvability.

4.1. Defining evolvability

The essence of evolving a software product is to modify the corresponding parts and elements of the product. Modifications can be classified into three categories: corrective, adaptive and perfective [36]. In our model of evolvability, we are mainly concerned with the modifiers’ point of view, since the ease with which a product can be modified is experienced primarily by the modifiers. The modifiers could either be end users who want to make some changes on their own, or professional software engineers who are contracted to evolve a system. Sometimes the user’s view is also considered. For example, a user may think that software which has a longer change turn-around time has poorer evolvability but generally, the modifier’s view is more relevant to this research.

To respond to an evolution request, the first step is to analyse the current system against the request. The aim is to understand the software system and locate which parts of it need to be changed. Then the actual change is implemented. As a software product consists of interconnected parts, changing one part may cause unexpected behaviour in other parts, so the software must be restablilised after a modification to ensure no unexpected behaviour will occur. Finally the changed product is tested to ensure the correctness of the new version. Thus we have identified four steps of evolving software products, which correspond to the four sub-characteristics of maintainability identified in ISO/IEC 9126 (1991): Analysability, Changeability, Stability and Testability. The new draft of the standard includes a new sub-characteristic, Compliance, to measure the compliance with standards or conventions. The following list gives the definitions of the sub-characteristics, taken from the standard with a minor modification in the case of Analysability (the original definition restricts this to corrective modification).

a) Analysability: The capability of the software product to enable the identification of the part(s) to be modified.

b) Changeability: The capability of the software product to enable a specified modification to be implemented.

c) Stability: The capability of the software product to avoid unexpected effects from modifications of the software.

d) Testability: The capability of the software product to enable modified software to be validated.

e) Compliance: The capability of the software product to adhere to standards or conventions relating to maintainability.

4.2. Measuring evolvability

The latest draft of ISO/IEC 9126, follows the GQM approach [2, 3] to derive metrics for evolvability. Each sub-characteristic acts as a goal that generates questions that have to be answered to achieve the goal. Then metrics are defined to provide the information to answer each question. Interpretations of measured value are also provided in the standard, which help to answer the questions and finally evaluate whether the goals have been met.

Often quality sub-characteristics can be seen from both internal and external points of view [6]. The corresponding methods and scales are termed internal metrics and external metrics in the new draft of the ISO/IEC 9126 standard. Unlike most previous maintenance measurement methods, which either focus on internal metrics or mix external and internal metrics together, the new ISO/IEC 9126 draft has a separate set of metrics for both internal attributes and external attributes. This separation not only makes clear what the data collected corresponds to (external attributes or internal attributes), but also helps to establish the relationship between the internal attributes and the evolvability of a software system.
However, we also identify some weaknesses in the current draft. It focuses on the user’s view of software evolvability. For example, most external metrics collect the elapsed time of a system’s evolution. We think this may be misleading as elapsed time may be a result of management decisions setting a lower priority on particular changes etc. Instead, we recommend using effort (person-hours for instance) in related external metrics. Another obvious weakness is the omission of some important internal metrics; in particular, there is no mention of any sort of complexity, cohesion or coupling metrics, which clearly impact on evolvability and could serve as good predictors of it.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Purpose of the Metrics</th>
<th>Measurement, formula and data element computations</th>
<th>ISO 9126 ref.</th>
<th>Comments</th>
</tr>
</thead>
</table>
| 1 Software change control capability | Can the modifier easily identify revised version? Can the modifier easily change the software to resolve problems? | $X = \frac{A}{B}$  
$A = \text{No. of change log data actually recorded}$  
$B = \text{No. of change log data planned to be recorded enough to trace software changes.}$ | 9126 -2  
8.5.2 -1 | More relevant to analyzability than changeability. |
| 2 Parameterized modifiability | Can the modifier easily change parameters to change software and resolve problems? | $X = \frac{A}{B}$  
$A = \text{No. of occasions when modifier changed software by changing parameters}$  
$B = \text{Total no. of changes made.}$ | 9126 - 2  
8.5.2 - 2 | In the standard, this metric is designed as:  
$X = 1 - \frac{A}{B}$  
$A= \text{no. of occasions when modifier fails to change the software by using parameter.}$  
$B= \text{no. of occasions when modifier attempts to change software by using parameter.}$ (N.b. B is subjective and hard to collect.) |
| 3 Modification complexity | Can the modifier easily change the software to resolve problem? | $T = \frac{\text{Sum}(A/B)}{N}$  
$A = \text{Effort spent to change}$  
$B = \text{Changed software size}$  
$N = \text{No. of changes}$ | 9126 - 2  
8.5.2 - 3 | In the standard, A is the time spent on a change. Size can either be measured in modules or LOC. |
| 4 Change cycle efficiency | Can the user's problem be solved to his satisfaction within an acceptable time scale? | $T = \frac{\text{Sum(Tu)}}{N}$  
$T_u = T_{re} - T_{sn}$  
$T_{sn} = \text{Time at which user finishes sending request}$  
$T_{re} = \text{Time at which user receives a revised version}$  
$N = \text{No. of times user receives a revised version from modifier.}$ | 9126 -2  
8.5.2 -4 | In the standard, N = No. of revised versions. This metric reflects a users’ view of evolvability. |
| 5 Change implementation effort | Can the modifier easily change the software? | $X = \frac{E}{N}$  
$E = \text{Effort spent to change the software}$  
$N = \text{No. of changes made}$ | 9126 - 2  
8.5.2 - 5 | The standard uses time in the measurement and only deals with failure. Here other kinds of modifications are also taken into account. |
| 6 Outstanding change effort | How much effort is required for unresolved changes? | $X = \frac{E}{N}$  
$E = \text{Estimated effort to implement the unresolved changes.}$  
$N = \text{No. of changes which have been successfully analyzed but not implemented yet.}$ | New | Unimplemented change is also important for measuring changeability. |
| 7 Average change size | Can the modifier easily change the software? | $X = \frac{A}{N}$  
$A = \text{Size of changes that have been made}$  
$N = \text{No. of changes made}$ | New | Size can be measured in modules or LOC. |
Our research suggests improvements to the ISO/IEC 9126 metrics. We have derived external and internal metrics from each sub-characteristic. An example can be seen in Table 1, which lists external metrics of changeability [11]. The current study focuses on finding suitable complexity, cohesion and coupling metrics for abstract representations, software architectures, designs and source code. For example, more investigation is required to extract metrics from ADLs.

4.3. Models of software evolution

Ideally, a validated model of software evolution should not only be quantitative and predictive but also explanatory. In this section we present some work in progress to develop models of software evolution dynamics that will have some explanatory power.

Previous work on software evolution, notably by Lehman et al. [19, 21, 23], focussed on size (number of modules) as the principal measure of evolution. We adopted a different viewpoint and found different results. Our investigations have uncovered an interesting class of software products that share some of the characteristics of the systems that Lehman studied but which do not exhibit the steady growth in system size that he found. For example, we examined the evolution of the Berkeley DB database product [35] and found that in many releases there was no identifiable increase in system size; most changes, even additional functionality, changed existing modules rather than adding new modules.

We made estimates of the quantity of module changes per release of Berkeley DB by analysing the published change logs and comparing them with the current release of the source code. When we also re-analysed a series of Lehman’s published data as number of module additions per release (rather than the total size of the system at each release), then the patterns of evolution appeared to be similar. For example, Figure 4 re-plots Lehman’s data for the Logica-FW system to show the number of modules added in each release. Figure 5 plots our estimate of the number of module-changes in a sequence of releases of Berkeley DB. We are aware that these measurements of the evolution of Berkeley DB are crude but nevertheless the similarities between the patterns is interesting. One effect of the changed viewpoint is to obscure the long-term stability of the evolution process that Lehman identified and instead to highlight the volatility of the quantity of changes in successive releases.

When we look at evolution from the viewpoint of volume of change per release, rather than total size of the system, then the inverse square law [23, 38] is no longer an applicable model. We have therefore turned to probabilistic models of system dynamics to provide insights into this volatility.

A very wide range of industrial, commercial and social processes can be effectively modelled as queues, in which a succession of customers require some service from a number of servers [9, 31]. The behaviour of queues can be described mathematically, making it possible to simulate queuing systems with different parameters and compare their outcomes. It is possible to model the variation in size of software releases using queuing theory; we can think of a software release as the output of one or more queues of required software changes (the “customers”) over some

![Figure 3 Evolution of Logica-FW system (after Lehman [23])](image-url)
unit of time. The servers of the queues represent the development and maintenance teams who process the software changes.

We have conducted a number of exploratory experiments to assess what kinds of queuing models produce the best fit to the evolution patterns that are observable in real-world software products. For example, we considered one of the simplest queuing models in which the only source of variability in queue dynamics is a Poisson probability distribution for the inter-arrival times of the customers; in Kendall’s notation [18] this is denoted by $M/D/\infty$. However, a simulation of this model, calibrated to the same mean as Lehman’s Logica-FW data, produced a dramatically smaller variance than the Logica data (77 and 13,559 respectively) and therefore we rejected this model as implausible.

From the experiments that we have conducted so far, we have drawn the conclusion that none of the simplest classic queuing models produces a close enough fit to real-world data. We are now investigating more complex models. For example, both the Logica and Berkeley data series suggest that there may be two queues superimposed; one queue provides the “small” releases, the other queue provides the occasional “large” releases.
In Figure 6 we show an example simulation of a composite queues model. This model is composed from three elements:

a) The sizes of the minor releases are generated by a Gamma probability distribution.
b) The occurrences of the major releases are generated by a uniform probability distribution.
c) The sizes of the major releases are generated by a Poisson probability distribution.

Although this kind of queuing model can achieve a much closer fit to both the Logica and the Berkeley data, in our current understanding of the model this fit is achieved by trial and error. Further work is required both to validate the model against a broader range of software products and to identify whether there are measurable qualities of these products that correlate with the parameters of the model. If this can be achieved then we will have made substantial progress towards an explanatory model of software evolution.

5. Conclusions

This paper presents an approach to understanding software evolution based on the empirical study of software evolvability. We propose that evolvability can be measured at different levels of abstraction of software systems, i.e. at pre-design, architectural, detailed design and source code levels. Many previous approaches emphasized the lower levels of abstraction, but by providing a real world example we demonstrate the importance of introducing evolvability at higher levels. This example also illustrates our innovative approach to exploiting ADLs in the field of software measurement.

Our research also suggests a set of metrics for software evolvability based on the ISO/IEC 9126 standard. By collecting metric data at different stages in a software life cycle and from products at different levels of abstraction, we will get assessments of evolvability at different levels. These assessments will guide the improvement of evolution processes and adjust the relation between software and its environment.

Our future work will be based on this integrated approach. So far, three directions have been identified: understanding the impact of several internal attributes of a software system on evolvability, constructing predictive models of software evolution and investigating the relationship between the organisational environment and software systems.

Acknowledgements

This research was carried out within the DESEL (Designing for Ease of System Evolution) project, which is supported financially by the UK Engineering and Physical Science Research Council (grant number GR/N01859) and by the University of Reading.

References


