Experiences in evolvability research

Pierre America \(^a\), Piërre van de Laar \(^b\,*\), Gerrit Muller \(^b\,\!c\)

\(^a\) Philips Research, Eindhoven, The Netherlands
\(^b\) Embedded Systems Institute, Eindhoven, The Netherlands
\(^c\) Buskerud University College, Kongsberg, Norway

**Abstract**

Many technical products and systems nowadays have functionality that is largely determined by software. Such systems range from very small (e.g., implantable hearing aids), via small (e.g., mobile phones), medium-sized (e.g., television sets), and large (e.g., medical MRI scanners), to very large (e.g., battle ships). Because of the dominance of software, we call them software-intensive systems. Although for many of these systems software takes up a large part of the development effort, several other disciplines are also essential, for example, electronics, mechanics, optics, or chemistry. Moreover, since these systems are developed for a purpose, disciplines related to the application area are also important, for example, medicine, photography, or defense. This interplay of several different disciplines typically leads to complexity in several areas, such as requirements, design, project organization, and team communication.

The requirements for software-intensive systems change over time. The changes can originate from various sources. There can be a change in the application domain, for example in the applicable medical procedures. The user may wish to diagnose a larger number of patients per week. The environment, for example the hospital infrastructure, can change. The manufacturer may want to increase market share. A supplier may stop producing certain components. Whatever the source of the requirements change, the system will have to be adapted to deal with it, which means the system must evolve. We define evolvability as the ability of the system to respond to such changes.

**1. Introduction**

Many technical products and systems nowadays have functionality that is largely determined by software. Such systems range from very small (e.g., implantable hearing aids), via small (e.g., mobile phones), medium-sized (e.g., television sets), and large (e.g., medical MRI scanners), to very large (e.g., battle ships). Because of the dominance of software, we call them software-intensive systems. Although for many of these systems software takes up a large part of the development effort, several other disciplines are also essential, for example, electronics, mechanics, optics, or chemistry. Moreover, since these systems are developed for a purpose, disciplines related to the application area are also important, for example, medicine, photography, or defense. This interplay of several different disciplines typically leads to complexity in several areas, such as requirements, design, project organization, and team communication.

The requirements for software-intensive systems change over time. The changes can originate from various sources. There can be a change in the application domain, for example in the applicable medical procedures. The user may wish to diagnose a larger number of patients per week. The environment, for example the hospital infrastructure, can change. The manufacturer may want to increase market share. A supplier may stop producing certain components. Whatever the source of the requirements change, the system will have to be adapted to deal with it, which means the system must evolve. We define evolvability as the ability of the system to respond to such changes.

People commonly use biologically inspired terminology when discussing evolvability, suggesting that a system evolves autonomously in a long and tedious trial-and-error manner \([1,2]\). However, system development requires a purposeful process that involves limited time and resources and is reasonably predictable.

Improving evolvability of complex software-intensive systems was the goal of Darwin \([3,4]\), a cooperation project between Philips Healthcare, Philips Research, five Dutch universities, and the Embedded Systems Institute. It ran from October 2005 to September 2010 and was co-funded by the Dutch government. This project focused specifically on one kind of complex software-intensive systems: medical MRI scanners. These systems are representative for complex, software-intensive systems because they show the same phenomena in their development and evolution.

In this paper we intend to share some of our experiences, insights, and results.

At a first glance, one would think that evolvability depends only on the system itself, but some reflection reveals other factors that determine evolvability. We denote this wide context as Business, Architecture, Processes, and Organization (BAPO) \([5]\). Then evolvability is characterized not as a system property, but as a BAPO property. We still focus on the evolution of the system, but in the wide context of BAPO. Here are some ways in which the elements of BAPO influence evolvability.

**Business:** When the situation of customers changes, and therefore their expectations towards a system, there is no general law that dictates that the system must be adapted. Rather it is a business decision to determine whether, and to what degree, the
system will be adapted to meet these changed expectations. In an extreme case, a business might decide that it seems more advantageous to withdraw from a part of the market than to do a costly redesign of a system. Another important factor is how well the business can predict or even shape particular changes in requirements.

Architecture: By architecture, we mean the most important technical decisions about a system [6]. Such decisions determine, among others, the structure of the system (the way it is composed from subsystems) and the choice of technical solutions for the main concerns about the system. It is clear that a structure that keeps changes localized to a particular subsystem makes it easier to respond to a change in requirements than a structure that necessitates changes to many different subsystems. Similarly, where one choice for an individual technology may lead to design changes to deal with changed requirements, another choice may lead to a system that can satisfy the changed requirements without design changes.

Processes: A development organization often has well-defined processes: ways to perform activities that recur regularly. These processes typically also determine the way changes are made to a system and therefore influence the time and effort required. They include aspects such as whether documentation (for requirements, designs, tests, etc.) is completely updated (which may require a large effort and make it difficult to understand the changes with respect to the previous version) or a delta document is written, describing only the changes (in which case people easily lose the overview of the current version). In addition, the process for dealing with a change may or may not require many steps involving multiple stakeholders, which ensures that drastic changes are carefully considered before they are carried out, but cause a relatively large overhead in effort for simple changes.

Organization: This encompasses aspects such as the division of responsibilities, skills, and knowledge, the communication patterns among people, and their general attitudes and value systems (in other words, the organization’s culture). It is clear that if a change requires the cooperation of various people in different continents who have different objectives and do not communicate very often, this is more difficult than in a situation where the change can be handled by just a few people sharing a single room. As we will see, the way knowledge is acquired, consolidated, and shared is of utmost importance where evolvability is concerned. Moreover, the maturity of the organization regarding evolvability determines which kinds of measures can improve evolvability.

The fact that evolvability is not a system property but a BAPO property means that it is extremely difficult to find an adequate quantification for it, which would allow quantitative specifications and measurements to serve as a yardstick for the success of our research. In the literature and in our own research we did not find anything that came close to such a quantification of evolvability. We believe that this is a serious obstacle in evolvability research.

Nevertheless, in the beginning of the Darwin project, we developed a vision of where the focus of the research should be. This is represented in Fig. 1. We envisaged that evolvability would improve by research on the following subjects:

- Extracting (mining) knowledge from past and existing systems.
- Developing new mechanisms, patterns, and guidelines to improve system evolvability.
- Representing knowledge into a reference architecture.
- Supporting architectural decision making on an economic basis.

The rest of the article describes our experiences and results in more detail. In particular, in the following section we relate evolvability to knowledge and introduce the topics of extraction of knowledge, representation of knowledge, and deciding with knowledge. These are discussed in more depth in Sections 3, 4 and 5, respectively. We end with our conclusions in the final section.

2. Relating evolvability to knowledge

It is obvious that developing a complex system requires a large amount of knowledge. Developing such a system is typically a task for a team of people that have extensive training and experience in various different disciplines. But a large fraction of the relevant knowledge is not written down in textbooks or other publications, rather it is related to and often even specific for the system itself and the role it plays in its environment. Although a development team usually does a decent effort to document its work, it can never document all details, simply because there are too many of them. The result is that the largest part of that knowledge remains tacit, i.e., implicit and confined to the heads of individual developers.

Fig. 2 illustrates that the focus of developers changes over the abstraction levels. The abstraction level with most details copes with the mono-disciplinary realization artifacts such as parts, connections, and lines of codes. Higher abstraction levels have fewer details but consider multidisciplinary relationships among components. At the system level, developers think in terms of relatively few system properties, which directly relate to system requirements.

Even the knowledge that is explicitly documented is not necessarily easily available to the developers who need it. Typically, that
knowledge is scattered over many different documents in different formats such as source code archives and documentation in natural language. These documents themselves are sometimes distributed over formal project repositories, informal wikis, private file systems, and mailboxes. In an early study, Alexander Douglas, a Darwin colleague, counted ten different sources of documents in a single development department. Moreover, the relationships between information in the various documents are often not sufficiently documented to enable traceability. (Douglas also observed this phenomenon in practice.) In this way it can be a real challenge to find the document that represents the particular piece of knowledge that a developer needs in the actual state of a development project.

The problem is not only that the knowledge is difficult to find. Most organizations suffer from a loss of overview, caused by organization size and complexity and by the age of the architecture. In larger and more complex organizations employees tend to become more specialized; this narrow focus is the enemy of overview. The age of the architecture plays a role because most architectures gradually degrade when evolution step is stacked on evolution step. The rationale of the current situation may have been lost over time, and the rationale may not be logical from a contemporary viewpoint.

All this knowledge is not only needed when developing a new system, but also when an existing system is adapted to respond to changes. In a complex system, it is already difficult to identify the exact technical modifications that are needed to adapt the system to the external change. It is even more difficult not to break the system, or more precisely, to ensure that all quality aspects of the system that initially had acceptable values do not become unacceptable. In many cases the initial system has been carefully designed to meet those quality requirements. In other cases the quality requirements were already met more or less by coincidence. But a modification in the system may inadvertently upset the multiple relationships between the components in such a way that the quality requirements are no longer met. For example, the increased power dissipated in one component may lead to a higher temperature in a nearby one so that it runs out of its usable temperature range. A new gantry on an MRI examination table may necessitate a new position for an electromotor, which disturbs the carefully equalized magnetic field in the scanner. For these reasons, a large amount of multidisciplinary system knowledge shared among most people involved in system development is an essential precondition for evolvability of the system. From our own involvement in many complex system development efforts, we would even like to claim that in most cases, the lack of shared knowledge is an important limiting factor for evolvability.

We explored the following steps to get from tacit knowledge to knowledge that can be used for evolvability.

- Extracting knowledge, both from humans and from artifacts. This step converts tacit knowledge into explicit knowledge. It also has to deal with the fact that some knowledge is no longer present or even missing: Some employees may have left the company while source code and documentation typically do not explain the design rationale behind them.
- Representing knowledge. This includes capturing, articulation, and visualization of knowledge. These are necessary to make the knowledge accessible to developers so that it can be shared effectively.
- Deciding with knowledge. This involves analysis of decision-making processes to see which knowledge is actually used and which knowledge should ideally be available for well-founded decision making.

These steps are elaborated further in the next three sections.

As can be seen from Fig. 1, the Darwin project also worked on new mechanisms, patterns, and guidelines to improve evolvability. Some of this work is described in another paper in this special issue [7]. Although some interesting results came out of this work, practitioners did not eagerly adopt them. We suspect that initially the need for shared knowledge is perceived as the limiting factor for evolvability and that the usefulness of new mechanisms, patterns, and guidelines increases as the shared knowledge in the organization grows.

3. Extracting knowledge

In the previous section, we argued that a complex system, like an MRI scanner, is the result of the combined effort of hundreds of people from many different disciplines. And, these hundreds of people have already been working for decades on this complex, evolving system by continuously adopting, extending, and improving it to ensure an optimal fit for its stakeholders in a changing environment. While evolving a complex system, these people consult, use, and develop a large amount of knowledge. Due to the sheer size of this knowledge, it can never be documented in all details. Even worse, since it is often difficult, if not impossible, to predict the relevant details to be documented, the documentation will often provide too little background knowledge to predictably, efficiently, and effectively adapt a system to meet its changed requirements. In this section, we will discuss methods to augment the explicit knowledge (i.e., knowledge consolidated in documents or models) to ensure a successful adaptation of the system. The methods we discuss are

- Mining tacit knowledge.
- Abstracting realization data into knowledge.
- Mining system-generated data, and
- Mining process-related data.

3.1. Mining tacit knowledge

We first discuss the mining of tacit knowledge as a method to augment the explicit knowledge. As said earlier, a large amount of knowledge about a complex system is only available in the heads of the developers. In other words, the developers must be involved when the knowledge must be shared and consolidated. We observed that in particular multidisciplinary design decisions are marginally documented and that the linkage between systems requirements and realization is mainly in the heads of the developers [8, Section 2.4]. This observation is depicted in Fig. 3.

In the Darwin project, the researchers frequently mined for tacit knowledge by interviewing and asking specific questions to the
system developers [9–14]. Our observations of the Darwin researchers in action can be summarized as follows:

- The process of mining tacit knowledge is labor intensive. The knowledge contained in the heads of the developers is not always complete, correct, and consistent. Multiple iterations with different developers are needed to make the knowledge sufficiently complete, correct, and shared.
- The tacit knowledge that can be mined is limited. Regularly, we noticed that parts of the knowledge are lost, due to e.g. developers having left the company and the limitations of human memory.
- Various interview styles were used. Some interviews asked the developers open questions. Others presented the explicit knowledge as understood by the researcher. The latter kind of interviews turned out to be more effective despite limited understanding by the researcher.

Based on our observations, we recommend that tacit knowledge is mined only when needed to adapt a system to changed requirements. We also consider the consolidation and documentation of the recovered tacit knowledge an essential part of mining, since documents will immediately elicit feedback, ensure shared knowledge, and might be valuable sources of knowledge for future requirements changes.

3.2. Abstracting realization data into knowledge

Second, we discuss abstracting detailed data into knowledge as a method to augment the explicit knowledge. Most opportunities to find suitable data are related to the realization: Compared to the design decisions, the realization is typically well documented, and often even described in a machine readable format, such as source code or CAD drawings. These opportunities are depicted in Fig. 4.

In the Darwin project, the researchers regularly tried to abstract available knowledge using existing or prototype research tools [15,16]. Our observations can be summarized as follows:

- Abstracting data into knowledge benefits from conventions and guidelines and, especially, the strict adherence to them in the realization artifacts. In other words, the more structure exists in the data, the easier it is to transform it into knowledge.
- Abstracting can only lead to knowledge about the realization itself, such as call graphs, containment relationships, and applied patterns. However, not all knowledge is typically present in the realization [17], e.g., consider design rationales, design alternatives, design decisions, and (traceability to) system requirements and system qualities. Fortunately, realization knowledge can help in recovering this other knowledge. For example, discovery of a pattern leads to the question why this was applied.

Based on our observations, we recommend limiting the ambition in abstracting realization data into knowledge. To achieve more with automatic abstraction requires including more information in the realization, e.g. by using annotations and attributes.

3.3. Mining system-generated data

Third, we discuss mining system-generated data as a method to augment the explicit knowledge. While the system is operating, a large amount of data is generated. This data includes the different log files and the events as captured by system monitoring tools. The system-generated data is not part of the system requirements, design decisions, and realization. In other words, it is outside the pyramid as drawn in the previous three figures. Yet, by analyzing and interpreting system-generated data we can construct a description of the dynamic behavior of the system. This can provide valuable insights about the system.

In the Darwin project, the researchers tried systematically to mine system-generated data using prototype research tools [18]. Our observations can be summarized as follows:

- The dynamic behavior of the system was barely documented. Furthermore, both the developers and the available documentation were far more static-oriented than dynamic-oriented. This must be due, at least partly, to the difficulty of describing a system’s dynamic behavior, in all its variations, in a static document. Moreover, over the last decades we see a trend in programming paradigms (from imperative, over object-oriented and component-based, to service-oriented architectures) where the high-level abstractions are more and more removed from the dynamic behavior. This also leads the designers to focus on the static structure rather than the dynamic behavior.
- The content and structure of the log files emerged from years of incremental realizations; they were often not architected or designed. For example, regularly, log files contained data specific to debug a particular issue of an earlier version of the system that is solved in the current version.
- System-generated data can be transformed into more abstract information. This information can be discussed with domain experts, trigger further investigations, and so lead to knowledge.
- Mining system-generated data benefits from the domain-specific abstractions as available in the log files.
- Mining system-generated data benefits from the availability of logging concepts such as transactions with unique identifiers and well-defined beginnings and endings. The effectiveness of these concepts depends on the diligence of the developers with respect to logging, in general, and the compliance with the transaction design in all situations, in particular. Logging concepts apply not only to the normal mode of operation, sometimes called the happy flow, but also to exceptions.

Based on our observations, we recommend to architect the system-generated data. Create the relevant scenarios and execution viewpoints and provide the necessary abstraction in the log files. Furthermore, use well-known logging concepts, such as transactions, and prevent duplication (of development effort and data) by using general purpose system monitoring tools. If this is done properly, the system-generated data can be used to monitor the dynamic quality aspects of the system. For example, the techniques developed by our colleague Trosky Callo were adopted by...
Philips Healthcare to benchmark the startup sequence of the MRI scanner [19].

3.4. Mining process-related data

As the last method to augment the explicit knowledge, we discuss the mining of process-related data. While developing and evolving a complex system, developers not only explicitly capture knowledge but also generate a large amount of data, ranging from e-mails and chat messages to changes to the source code archive. The process-related data is not part of the system requirements, design decisions, and realization. In other words, it is outside the pyramid as drawn in the previous three figures. Yet, mining the process-related data might yield valuable insights about the system, possibly ranging from requirements to realization.

In the Darwin project, the researchers frequently tried to mine process-related data using prototype research tools [20,21]. Our observations of the Darwin researchers in action can be summarized as follows:

- Sometimes process-related data can be transformed into interesting information. This information can be discussed with domain experts, trigger further investigations, and so lead to knowledge. For example, by looking at simultaneous code changes across modules, our colleague Adam Vanya found a misplaced header file.
- Interpretation of process-related data requires good understanding of the way the developers work. This can be consolidated into a process model. The information obtained by mining process-related data strongly depends on the quality of this process model. Developing a high-quality model is labor-intensive, and requires considerable domain knowledge on the process and organization. For example, the modeler should determine how a system change is checked into the version control system. There are many alternatives in the number of people involved and the timing of the actions. Moreover, in large organizations and when the process-related data covers a long period of time, the model has to reflect the differences among organizational units and changes over time in the process and organization, e.g., changes in system and software development methods, programming paradigms, and programming languages.

Based on our observations, we only recommend the mining of process-related data when the relevant aspects of the process are well known, shared, and adhered to throughout the organization. In other words, mining process-related data is only valuable when a high-quality model that describes how the data is actually produced can be made. On the other hand, if the organization makes good use of process-related data, this is a good incentive to optimize the official processes and improve compliance to them.

4. Representing knowledge

4.1. Reference architectures

Reference architectures capture the essence of the architecture of a collection of systems [22]. The purpose of reference architectures is to provide guidance for the development of architectures for new versions of the system or extended systems and product families. Fig. 5 shows how reference architectures can guide the development of actual systems in a few steps.

The initial vision in the Darwin project was that reference architectures therefore would be well suited to facilitate evolvability by transforming essential knowledge from the past into guidelines for the future. One of the core ideas of reference architectures is that their credibility is based on past experience. The initial step to create a reference architecture is to study past systems and their architectures, and to filter out the architecture relevant knowledge, for instance in the form of architecture patterns.

The researchers in the Darwin project were challenged to lift the knowledge obtained by mining to the level of a reference architecture view. We observed that extensive discussions with designers and architects are needed for this abstraction step. Borches [9] and Douglas recovered many different views of the acquisition subsystem in an effort to create a reference architecture for acquisition subsystems. Their efforts evolved into A3 architecture overviews described in Section 4.3.

The main lessons learned from the efforts of creating a reference architecture confirm the conclusions in [22]:

- Many views (e.g. physical, functional, quantified) are needed to capture architectures and reference architectures [23].
- A lot of interaction with experts is needed to mine previous architectures. Mining involves recovering knowledge, filtering out architectural relevant knowledge and to transforming it into guidelines.

4.2. Modeling

Modeling is an alternative to verbal descriptions to capture knowledge and to make it accessible. The term modeling is used for a broad spectrum of models, ranging from informal sketched models to fully formalized executable models that can even be used to generate code for the system itself. Models can be used to communicate, analyze, facilitate decision-making, and to verify. In the Darwin project research was done how models can be used to capture knowledge and in this way facilitate evolvability. Some of the research efforts were:

- Visual workflow modeling to enable early and crucial feedback from non-technical end-users [12]. Here simple workflow visualizations and 3D animations were used simultaneously to understand and discuss how existing systems could be connected effectively. The use of animations was especially effective in communicating with clinical (non-technical) stakeholders.
- Formal modeling with explicit syntax and semantics to allow automatic analysis and code generation was applied with various formalisms. Theunissen [13] used a state-based formalism to model the patient communication and audio subsystem to elicit early feedback from non-technical end-users. This...
subsystem evolved from a simple intercom into a multitasking audio system. Gülesir [14] modeled the dynamic behavior, in particular the data and control flow, of software intensive systems.

Developers look with interest at the formal modeling efforts. They agree with the argument that modeling creates and captures insight that supports evolvability. However, developers’ interest is insufficient to entice them to try this kind of modeling themselves. From our team’s efforts in modeling we conclude that modeling can be considered as a promising means to support evolvability. We speculate, based on our interaction with developers, that modeling is perceived as a specific skill. Hence, it requires a modeling competence (knowledge of formalisms, tools, and techniques, and the skills to use them) in the development crew that in many cases is not yet fully developed. Future research is needed on how to help develop modeling competence and how to provide practical guidelines to make appropriate models.

4.3. A3 architecture overviews

Capturing knowledge of a system requires many different views. Architects at the System Architecting Forum, a biannual meeting of international architects (www.architectingforum.org) agreed that 10–12 views are used actively to document a system [24]. Even more views are relevant, but practical constraints, such as the capability of the stakeholders to cope with the number of views, limit the number of actively used views. Bonnema [25] states that physical view, functional view, and quantification are a practical minimum subset.

Our colleagues Daniel Borches and Alexander Douglas took Bonnema’s work as starting point to study evolvability in the Darwin project. They extracted the knowledge about the communication of data acquisition in MRI systems by reading documents and interviewing designers. The results were documented as large physical and functional block diagrams, one diagram per large sheet of paper. These diagrams were well appreciated by the organization as consolidation of the current situation. However, for the design of the next generation communication infrastructure these large sheets with one view were not used by the organization. Borches tested several techniques and modeling formalisms, such as Design Structure Matrix (DSM) and SysML without positive response from the organization. The use of A3s to capture the architecture overview, however, was met with enthusiasm. A3s are well known from the LEAN product development world [26], where they are used to capture designs and project status, among others. Borches developed a method to capture A3 architecture overviews and documented it as double-sided A3 cookbook [27]. We describe this A3 approach in more detail than other topics because there is more evidence that it is useful in practice (see the survey results in this A3 approach in more detail than other topics because there is more evidence that it is useful in practice (see the survey results in [28]).

The initial A3s need visual models. Visual models trigger immediate interaction in meetings. Physical view, functional view, and quantification are commonly applicable. Actual A3 content and layout requires customization per domain and the topic that is addressed. Many architects struggle with making good visualizations. Physical view (or partitioning in case of software) tends to dominate; this is not different from “normal” architectures. Empty space on A3s invites interaction by sketching or scribbling. Designing A3s at multiple abstraction levels helps to keep focus on A3 and helps to provide context (zoom out) and depth (zoom in). A3s can well be applied on the system context, with for instance concept of operations, workflows, scenarios, etc. A3s work well in reverse architecting current systems. Developers are willing to spend time around A3s to recreate an understanding of the current system in a limited number of iterations. This reverse architecting effort around A3s triggers many discussions about the current architecture and results in improvement proposals. An example can be found in [28].

The main idea behind A3s is that the size limitation of the paper sheet forces the authors to reduce the information to the essence. At the same time, A3s are large enough to show multiple views concurrently, facilitating discussions across views. Borches proposes to use A3s double sided: one side with visual models, as shown in Fig. 6, and one side textual information. These two representations are complementary.

The findings of this research were that A3s are well received by the designers and managers. In practice, the conventional many page documents that were used to describe architectures were often not read; writing such documents is mostly done to satisfy the process. One of the main problems for most designers is the continuous time pressure in combination with a large information overload. A3s contain a digestible amount of information. When A3s are available in meetings they are actively used by meeting participants.

The success of the A3 research inspired several companies to experiment with A3s for architecture overviews. The original company where the A3 overviews were researched introduced A3s per functional cluster. Several other companies in the same research network did similar experiments, for example one of these companies made one A3 per system quality. We also applied and evaluated A3 use in several Norwegian companies. In all these companies the experiments were sufficiently successful to continue with A3s. Some findings of the broader application of A3s in the different companies are:

- The initial A3s need visual models. Visual models trigger immediate interaction in meetings.
- Physical view, functional view, and quantification are commonly applicable.
- Actual A3 content and layout requires customization per domain and the topic that is addressed.
- Many architects struggle with making good visualizations.
- Actual A3 content and layout requires customization per domain and the topic that is addressed.
- Physical view (or partitioning in case of software) tends to dominate; this is not different from “normal” architectures.
- Empty space on A3s invites interaction by sketching or scribbling.
- Designing A3s at multiple abstraction levels helps to keep focus on A3 and helps to provide context (zoom out) and depth (zoom in).
- A3s can well be applied on the system context, with for instance concept of operations, workflows, scenarios, etc.
- A3s work well in reverse architecting current systems. Developers are willing to spend time around A3s to recreate an understanding of the current system in a limited number of iterations. This reverse architecting effort around A3s triggers many discussions about the current architecture and results in improvement proposals. An example can be found in [28].
- People in the organization ask for some guidance in making A3s; in some companies conventions in creating A3s did evolve. More research is needed to evaluate the effectiveness in specific instances, and to determine if more generic guidelines can be developed to support developers in using A3s.
- As any form of documentation, also A3s benefit from a clear target audience and goal definition. Given the size limitation, the target audience and goal determine the appropriate content.

A3s are well suited to make architecture overviews. We see architecture overview as a prerequisite for evolvability, because it helps us predict the consequences of design changes, at least to some degree.
5. Supporting decision making

Assuming that relevant knowledge has been captured and made available to the appropriate stakeholders, the next step is to act on that knowledge, or in other words, to make decisions based on that knowledge. Here we refer mainly to architects’ decisions that affect the architecture. These decisions, when they are implemented, change the system, which is, of course, a form of evolution. Larger changes can also change the evolvability of the system, and our goal is to make sure that the evolvability becomes better, not worse. Actually, when it comes to those far-reaching decisions affecting the architecture of the system, they are rarely taken by architects alone, but typically by architects and managers together, where each looks at specific aspects of the decision dependent on their own role.

Let us look now at such a decision, which for simplicity’s sake we consider as a binary one: Whether or not to carry out a development effort, typically in the form of a project. So we assume that the architect has made all the technical choices as best he could and presented the result in the form of a project for a go/no go decision. Speaking in broad categories, we can divide such projects into feature development projects, which aim to change the user-observable characteristics of the system, and architecture renovation projects, which aim to improve the architecture of the system, in principle without changing its user-observable characteristics. In practice, all kinds of mixtures are possible, but one could split each mixed project into a feature development part and an architecture renovation part and decide on those separately. In theory, all these go/no go decisions are independent, but in practice there are many interdependencies and there is a limited development budget so that a subset of all possible development projects must be chosen. Ideally, there should be a balance between the feature development projects and the architecture renovation projects; otherwise the architecture deteriorates over time and many quality aspects, including evolvability, become worse. This concept is called technical debt [29,30]. In this terminology, as the technical debt increases, evolvability (and typically other quality attributes as well) gets worse. Therefore the challenge in decision-making is to keep the technical debt within limits.

In order to support architectural decisions, together with our colleague Ana Ivanovic we first investigated what kind of knowledge, or more generally, information, is needed. First we interviewed architects and managers and asked them what information they consider important [31,32, Chapter 4]. It turns out that a relative large number of information types are considered important, but there is quite a difference between architects and managers: Architects focus more on system-related information, whereas managers are more interested in financial data.

Then we did an online experiment to investigate what information is really taken into account in such decisions [32, Chapter 5]. Over a hundred people (architects, managers, and others) participated. The results were somewhat worrying. Of the various information items offered to the participants, only a few were actually used (quality and uncertainty), but they were not necessarily used in line with business objectives. These results suggest that there is a need for a systematic approach to taking architecture investment decisions.

As a basis for such an approach, we looked at ways to assess the economic impact of a development project. Simply put, the impact is equal to the added value of the project minus its cost. For estimating cost there are well-known methods [33] and we also observed in practice that project cost is often estimated in detail. However, estimating the added value of a project (the business case) is rarely done with the same rigor. Therefore we investigated a number of approaches to measure the added value of improving quality attributes by architecture renovation. For quality attributes that can be observed by the user, such as performance or usability, we found ways of estimating the added value [34], based on marketing concepts such as value-in-use and market segmentation.

For quality improvements that are not observable by the user, such as evolvability, estimating the value is much more complicated. In principle, real option theory offers an appealing model to quantify the added value [35,36]: improving the evolvability by architecture renovation corresponds to buying an option and implementing new features on the new architecture corresponds to executing the option. Unfortunately, in practice this approach turns out to be infeasible to estimate the value of architecture renovation, because the various parameters in the theoretical model cannot be estimated with sufficient accuracy. A much simpler method based on Net Present Value is already challenging enough [37].
We conclude that more research into estimating the value of architecture improvement is needed [38], but also that estimating the added value of development projects deserves a lot more attention in practical decision-making. Even in the cases where we had a coaching role, there was relatively little willingness to spend effort in value estimation compared to cost estimation.

6. Conclusions

This article gives an account of our investigations into the evolvability of complex systems in the Darwin project. In Section 3 we focused on extracting knowledge from various sources. We saw that mining knowledge is not at all straightforward. It benefits from a rigid and clear design and process: Not only the effort to mine is reduced, but also the quality of the mined information increases. However it is unclear whether more rigid designs and processes improve the evolvability of a system: do the advantages in mining outweigh the disadvantages of the increased rigidity?

In Section 4 we discussed ways of representing knowledge in the form of a reference architecture. We explored various ways of doing that, such as various modeling techniques and the A3 approach. Although we believe that the proposed approaches are beneficial, it is still true that preparing knowledge to be readily accessible in the development organization is a laborious endeavor.

Section 5 investigated how knowledge about the system and its surroundings can improve decision making about development projects. We saw that even if the subject of the decision is reduced to a simple yes-or-no question on whether a particular project should be carried out, it is very difficult (but not impossible) to support the decision by estimating the added value of the architectural improvements that the project realizes.

This brings us back to the postulate in Section 2: evolvability is related to knowledge. We believe that a good overview of the complete system, together with a deep understanding of the parts that are intended to be modified, is an important prerequisite for a successful development project. Hence the overall vision of our project was based on four cornerstones, three of which are about knowledge (see also Fig. 1):

- Extracting knowledge from existing systems. We tried various methods to do this, one of which was adopted by practitioners in our industrial environment [19].
- Developing new mechanisms, patterns, and guidelines to improve evolvability. We actually believe that these may become more useful as soon as an organization has sufficient knowledge to deploy them effectively.
- Consolidating knowledge into a reference architecture. Among the various approaches that we tried, the A3 approach was most successful. It has been adopted by several development organizations and has inspired several new research activities.
- Supporting architectural decision making on an economic basis. The work in the Darwin project increased the awareness of practitioners that economic impact of architectural investments should be analyzed systematically, where not only the easily quantifiable cost but also the more uncertain benefits should be taken into account.

Early on in our research on architectural measures for improving evolvability, we realized that this would not be easy, because evolvability is not just a system property, but a BAPo property, i.e., it is determined by business, architecture, processes, and organization. This makes it very difficult, perhaps even impossible, to quantify and measure evolvability, which would be necessary to obtain a rigorous evaluation of the effectiveness of any proposed improvement measures. Although investigations into quantitative approaches should be encouraged, we probably will have to live with the fact that evolvability, like so many other important things in life, cannot be measured, but we can still strive to achieve the best we can.

Acknowledgments

This work has been carried out as part of the DARWIN project at Philips Healthcare under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK program.

References


