Architectural support in industry: a reflection using C-POSH

R. J. Bril, R. L. Krikhaar, and A. Postma

SUMMARY

Software architecture plays a vital role in the development (and hence maintenance) of large complex systems (containing millions of lines of code) with a long lifetime. It is therefore required that the software architecture is also maintained, i.e., sufficiently documented, clearly communicated, and explicitly controlled during its life-cycle. In our experience, these requirements cannot be met without appropriate support.

Commercial-off-the-shelf support for architectural maintenance is still scarcely available, if at all, implying the need to develop appropriate proprietary means. In this paper, we reflect upon software architecture maintenance taken within three organizations within Philips that develop professional systems. We extensively describe the experience gained with introducing and embedding of architectural support in these three organizations. We focus on architectural support in the area of software architecture recovery, visualization, analysis, and verification.

In our experience, the support must be carried by a number of pillars of software development, and all of these pillars have to go through a change process to ensure sustainable embedding. Managing these changes requires several key roles to be fulfilled in the organization: a champion, a company angel, a change agent, and a target. We call our reflection model C-POSH, which is an acronym for Change management of the four identified pillars of software development: Process, Organization, Software development environment, and Humans. Our experiences will be presented in terms of the C-POSH model. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: software architecture; development process; organization; software development environment; change management
1. INTRODUCTION

1.1. Context

Philips is an electronics company that operates worldwide and develops high-volume electronics (HVE) consumer products, such as TV sets and set-top-boxes, as well as (low-volume) professional systems, such as business communication systems and medical imaging systems. All of these systems are becoming more and more software intensive.

The professional systems developed within Philips are examples of large complex systems with a long lifetime. The economic lifetime of a delivered private branch exchange (PBX, i.e., a telephony system in a business environment) developed by Philips Business Communications (PBC) is these days approximately 15 years. Software development in the digital PBX area within PBC spans over two decades, and the maintenance obligations for installed software packages may last over ten years. Similar figures hold for the medical imaging systems developed by Philips Medical Systems (PMS).

The size and complexity of these professional systems combined with their long lifetime has two main consequences. Firstly, subsequent generations of systems are typically developed in an evolutionary way. Hence, development may be viewed as maintenance and all categories of maintenance (see, for example, [1]) apply for these systems, not only the more-common corrective, adaptive, and perfective categories (which are explicitly covered by the definition of maintenance in [2]), but also the less-common preventive maintenance category. Secondly, the importance of the software architecture [3] of these systems increases rapidly due to new and changing market requirements, amongst others. Therefore, the software architecture should be sufficiently documented, clearly communicated, and explicitly controlled. In our experience, these requirements cannot be met without appropriate support at the architectural level.

1.2. Architectural support

Software architecture is still an emerging discipline (see [4,5]), and architectural support (AS) is therefore far from mature. Commercial-off-the-shelf (COTS) tools for either forward or backward (i.e., reverse) software architecting are still scarcely available, if at all. Maintaining architectural descriptions without appropriate means is hard, and systems with software developments spanning over two decades, therefore, typically do not have an up-to-date software architecture descriptions. During the evolution of software, the source code evolves, but the description of the software architecture typically does not evolve accordingly. As a consequence, the initial conformance relation between the software architecture as specified and the software system as implemented decays. Tooling for architectural support is required to keep the code conforming to the software architecture.

Although software architecture is becoming more important, it is unclear how to manage the software architecture within a real software development organization from a technical or from a managerial perspective [6]. Within Philips, we have gained quite a lot of experience in the area of software architecture recovery, visualization, analysis, and verification. This paper is mainly concerned with AS in those areas.

1.3. Technology development and transfer

Given the state of affairs of software architecture and the needs of Philips’ Business Units (BUs), Philips Research (PR) started investigations in the area of software visualization and analysis in the
early 1990s, and closely co-operated in a number of joint projects with Philips BUs to alleviate their pressing needs. PR therefore played an important role in transferring state-of-research results in the area of software architecture towards Philips’ BUs, thereby improving the state-of-practice within industry to state-of-the-art.

In an earlier paper [7], we reported that organizational issues are considerably more demanding than technical issues when developing and subsequently introducing means for AS, i.e., an architectural methodology with an accompanying toolset. In [8], we described influential elements for AS, however, this paper is entirely dedicated to the C-POSH model for embedding AS within organizations.

1.4. The C-POSH model

The business success of large and complex systems with a long lifetime is to a large extent determined by their architecture [9], and for software systems by their software architecture. As mentioned above, this requires that the software architecture for a software system is maintained. In our experience, AS must be carried by a number of pillars of software development in order to make the introduction and subsequent embedding of the AS in industry successful. In this paper, we discuss four pillars that we experienced to play a major role in the cases:

- Process;
- Organization;
- Software development environment (SDE);
- Humans (people).

Typically, each pillar has to go through a change process to ensure that the AS will actually be sustained; see Figure 1.

1.5. Related work

The importance of non-technical issues in AS for a business has received some attention in the literature.
For example, the BAPO-CAFCR approach described by Muller et al. [9] is an architecture-centric approach, which stresses the multitude of views and influential factors (most of which are non-technical) that play a role when defining an architecture. BAPO is an acronym that summarizes the influential factors for introducing an architecture, Business, Architecture, Process, and Organization, and CAFCR is an acronym that includes the different views on architecture, the Customer, Application, Functional, Conceptual, and Realization view.

The BAPO-CAFCR approach is complementary to the C-POSH model in that the focus of C-POSH is on maintaining an architecture, whereas the BAPO-CAFCR approach concentrates on defining an architecture.

Sametinger [10] mentioned the importance of non-technical aspects when applying software reuse in an organization. In particular, Sametinger highlights the economical aspect: it should only be embedded in an organization if it is expected to pay off. More generally, software reuse should only be embedded and/or sustained if it yields sufficient (economical or strategic) benefit for the organization. So, a regular evaluation should take place to see whether software reuse is (still) beneficial. This evaluation aspect as a justification for embedding is also valuable for AS. It is not yet part of the C-POSH model, but would be a desirable future extension. In this paper, we have restricted ourselves to businesses in which large, complex, and long-living systems are being produced, and have assumed that the continuing need for AS is evident. As will become apparent from one of the case studies, this is not necessarily true (see Section 5.1 for more details).

Sametinger also identified the importance of a number of other non-technical aspects, like legal issues, organizational issues, and measurement issues. These issues play a role when, for example, introducing software reuse in an organization. In this context the legal issues concern the rights and responsibilities of providers and consumers of reusable components. The reuse process is an economic model of supply and demand including producers, consumers, and a distribution mechanism. Also, organizational factors can greatly affect the implementation of reuse programs—think of a centralized group making reusable components. It is impossible to manage what we cannot measure. Therefore, Sametinger distinguishes the measurement issues as a non-technical aspect for a software reuse program.

In [11], three perspectives were presented that are of value when thinking about change management:

- internally driven change (push) versus externally driven change (pull);
- change directed at products and services versus those directed at design and production processes;
- incremental (or evolutionary) versus revolutionary change.

In the case studies described in Section 5, the change management processes were internally driven and the change was directed at design and production processes. In two cases, the changes were revolutionary, whereas in one case, an evolutionary approach was chosen.

1.6. Overview

This paper, has the following structure. In Section 2, we provide a definition of ‘software architecture’ that serves our purposes, i.e., that allows us to describe and classify the embedded AS considered in this paper. In Section 3, we describe the overall process of technology development and transfer within Philips in general and for AS in particular. Embedding results in industry is the final stage.
ARCHITECTURAL SUPPORT IN INDUSTRY: A REFLECTION USING C-POSH

Figure 2. The 4+1 view model.

of this process. The C-POSH model is presented in Section 4, Section 5 constitutes the main body of this paper, and is devoted to a presentation of three case studies in which our empirical C-POSH model is used to reflect on the efforts to embed AS. Conclusions are presented in Section 6.

2. ARCHITECTURAL SUPPORT

The following definition of software architecture serves our purposes for this paper.

The software architecture of a system comprises software components, connections (or relationships) between those components, and constraints on the connections.

A software architecture may be described using different concurrent views (see [12] for an empirical or [3] for a synthesized model), where each view addresses a specific set of concerns. In this paper, we will use the ’4+1 view model’ described in [3], which distinguishes the following views (see also Figure 2, which has been taken from [3]):

- the logical view, describing the services the system provides to the end-user;
- the process view, describing the system’s concurrency and synchronization aspects;
- the development view, describing the system’s static organization;
- the physical view, describing the mapping of the software onto the hardware, reflecting its distributed aspects;
- scenarios, showing how the other four views work together.

Note that the interpretation of the terms components, connections, and constraints in the definition of software architecture given above depends on the particular view. For example, for the development view, the components are the modules, the connections are the uses and part-of relations between modules (see [13]), and the constraints are the restrictions imposed on these relations.

Support can be given for forward or backward software architecting. Examples of support for forward software architecting are given in [14,15]. A component framework approach to achieve architectural variability in PMS is discussed in [14], and [15] presents object-oriented requirements modeling that is appropriate for the application domain of PMS as well. For software architecture maintenance, we also need support for backward software architecting. This paper focuses on AS for software maintenance in general, and in the area of software architecture recovery, visualization, analysis, and verification in particular.
Architecture support tools for software maintenance can be described in terms of the ‘extract–abstract–present’ paradigm (which was inspired by [16]; see Figure 3). The three activities shown in Figure 3 can be characterized briefly as (see [17]):

- **extract**—extracting relevant information from the system software, system history and system experts;
- **abstract**—abstracting extracted information to a higher (design) level;
- **present**—presenting abstracted information in a developer-friendly way, taking into account their current topic of interest.

In [8], we presented influential elements for AS. This paper introduces our empirical C-POSH model and presents a reflection on embedding AS within three software development organizations within Philips based on C-POSH.

### 3. TECHNOLOGY DEVELOPMENT AND SUPPORT

#### 3.1. Industrial research

PR is an *industrial* research centre, having an intermediary role between academic research and industrial developments within Philips’ BUs. PR uses and builds on academic results when available, and investigates complementary areas or areas that are not sufficiently mature for transfer towards Philips’ BUs where needed. Academic research typically focuses on application domain independent issues, whereas Philips’ BUs are primarily concerned with application domain specific issues. PR bridges this gap by their involvement in the transfer of application domain independent solutions in an application domain specific context. Figure 4 shows the relations in terms of feed-forward and feed-backward information flows between academic research, industrial research, and industrial development.
3.2. Research and development

With respect to software maintenance, the prime focus of PR is an architecture- and technology-oriented one.

Obviously, Philips’ BUs do not have such a prime focus. They also consider business, process, organizational, and development aspects. We are convinced that for the adoption of research results in industry in general, and the embedding of AS in particular, the latter four aspects are at least as important as architecture and technology, and may even be decisive for success.

3.3. Stages towards embedding

Technology transfer has multiple stages, such as problem identification and investigation, solution space exploration, evaluation of the feasibility of particular solutions, and embedding of appropriate solutions in an organization. The main responsibility for and the bulk of work associated with the initial stages of technology transfer are assigned to PR, whereas for the final stages they are assigned to Philips’ BUs.

For further information on the theoretical foundations laid and experience gained by PR in the area of software architectural recovery, visualization, analysis, and verification, the interested reader is referred to [18–22].

3.4. Embedding in industry

For the embedding in industry, i.e., the final stage of transfer, there are two main phases: construction and deployment.

During the construction phase, the tool-support is developed in isolation, i.e., in parallel with the development of the actual systems. The construction may either go through a succession of prototypes followed by regular development of a tool, or follow an incremental development trajectory. Typically, the tools are first run on a fixed data set (using a particular version of the actual system). As a next step, the tools are run on other versions of the system (usually the most recent versions), and finally the tools are integrated into the build process. To minimize interference with the development of the actual
system, typically no actions are initiated based on the initial results found or the support provided by the constructed tools. The involvement of PR is typically phased out during the construction phase.

During the deployment phase, the tools gradually become available to the receiving organization. As an example, the deployment of architectural verification tools, which check the conformance of a system to a set of architectural rules, may take the following steps. Initially, individuals perform checks on an incidental basis to gauge the system. As a second step, a software integrator performs checks to determine whether or not updates will be accepted. As a third step, software developers perform checks before they deliver their updates to the integrator. As a final step, software developers are ‘constrained’ by the rules, i.e., a local build may fail when an update violates the rules.

4. THE C-POSH MODEL

In this section, we take a closer look at the key pillars that need to carry support at the architectural level and the changes these pillars have to go through to ensure that AS will actually be sustained.

Within Philips, at several departments AS has been introduced with different degrees of success. To learn from these experiences we developed an empirical model to evaluate this. In this paper we reflect on introducing and embedding AS.

In order to successfully improve software development in general one should be aware of the technical aspects as well as non-technical aspects. In the C-POSH model we distinguish four non-technical aspects to be important for architectural support: Process, Organization, SDE, and Humans. Furthermore, to sustainably embed AS, one should manage change carefully.

4.1. Change management

Deployment of AS within an organization requires changes to all four pillars of software development, and such changes need to be carefully managed. Management for change typically involves a number of key roles that need to be filled in [23], i.e.:

- a champion believing in and attempting to obtain support for change;
- a company angel (or sponsor) authorizing change;
- a change agent implementing change;
- a target (in our case the software architects and software developers) needing the change.

Clearly, a single individual may have multiple roles, and in the case studies presented in the subsequent sections an architect (i.e., a target) typically also has the role of champion and/or change agent.

4.2. Process

Philips has adopted the Capability Maturity Model (CMM) [23] for their software process improvement efforts. A particular CMM-level does not ensure that AS will actually be sustained in an organization. It is, however, very likely that an organization assessed at level 1 will be less successful at achieving sustainability than an organization assessed at level 2 or higher.

From level 2 to level 3, organizations have to pay attention to subjects such as standards, software inspections, process models, and advanced configuration management topics. Note that the latter
include architecture and design changes. Embedding software architecture in an organization requires changes to each of these subjects, and without proper coverage it is unlikely that AS can be embedded sustainably.

4.3. Organization

A prerequisite for the embedding of AS is the recognition of the vital importance of software architecture by management. An organization lacking architectural focus is considered a major obstacle. A minimal requirement is therefore management support (both at the level of software development organization leaders as well as project leaders), and clearly defined roles of software architects. Documents such as [24] may help to initiate awareness.

4.4. SDE

Software development spanning two decades will encounter major changes in the (host) development environment (including the platform). It has long been a necessity and therefore the standard for the professional industry to develop and use a suite of dedicated proprietary tools on mainframe computers. The advance of standard desktop environments, increased market pressure, and the need for productivity increase has led to major changes in SDE during the past decade. The professional industries considered in this paper moved from SDEs on mainframes to SDEs on (networked) PCs that are to a large extent based on COTS tools. This move towards COTS tools where deemed appropriate may also be viewed as a strategic move; it allows a healthy focus of Philips’ BUs on their core business, which lies in the telecommunications and medical domains. From a software development point of view, the changes in the SDE may be considered as adaptive maintenance, necessitated by the changing environment.

Considering the desired focus on core business, development of proprietary means for AS may come as a surprise. As in the situation two decades ago, the means required are not available at present, and those means are considered indispensable for the maintenance of our professional legacy systems. This has three main consequences. Firstly, we have to develop those means ourselves. Secondly, in order to get those means up to the mark and accepted within the organization, they need to be fully integrated into the SDE. Finally, it becomes necessary to build-up experience in developing tools for the new SDE. These three consequences are the main reasons why the SDE influences AS.

4.5. Humans

Typically, software architects are the main target of AS. However, maintenance of the software architecture does not require the maintenance of the software architectural artifacts alone. There should also be a conformance relation between the software architecture as specified and the software system as implemented. An active participation of the software developers is therefore of paramount importance to ensure that the AS serves its purposes. All persons involved in software development must be made aware of the vital role of software architecture. Software developers must believe in the added value of software architecture to make it successful. Moreover, when an architectural methodology and accompanying tool-set are introduced, the target group must be involved right from the start, and sufficiently trained.
5. CASE STUDIES

In this section, we present and reflect on the efforts to embed AS in three software development organizations within Philips, one from PBC (denoted by ‘P’), and two from PMS (denoted by ‘Q’ and ‘R’). The professional systems developed by each of these three organizations may be viewed as large, complex, and software intensive systems with a long lifetime. The software core of each of the systems has a size of roughly 3 MLOC (million lines of code) mainly written in C and C++. From these three systems, that from ‘Q’ has existed for approximately 10–15 years, and the other two have existed for almost 25 years. For each of these systems more than 100 software developers were involved. The reflection on the efforts to embed AS in these three organizations is structured according to the four main pillars of C-POSH.

5.1. Case study PBC: ‘P’

5.1.1. Introduction

In the early 1980s, PBC started the development of a new range of digital telephony systems for the business market termed SOPHO. These systems were very successful, hence became an asset, and particular care was taken to maintain this legacy. In [7], an overview is given of the ongoing efforts to improve the maintainability and quality of the software core of this legacy. Moreover, the paper describes the identification of the need, development, and subsequent deployment of support for program understanding and complexity control at the architectural level for SOPHO. That support is provided by a basic set of proprietary architectural tools for SOPHO, termed URSA‡. URSA has been developed as a joint effort between PR and PBC. The time span from the start of the initial explorations for the desired AS until the completion of the dissemination of the final conceived model including the tool-support is about five years, i.e., from 1994 until 1999. We have already stated that the organizational issues are considerably more demanding than technical issues when developing and subsequently introducing a tool-set like URSA. In this section, we evaluate the sustainability of the envisioned changes.

In the following, we first briefly recapitulate the situation medio 1999 as described in [7]. We then describe the developments during the past four years and characterize the current situation. We summarize our experiences in a final section.

5.1.2. AS

Using the terminology of [3], URSA contains the following proprietary tools.

- **Jolly Jumper**: a clickable Message Sequence Chart (MSC) viewer, visualizing a scenario in terms of actors and their communication, and providing (hyper-)links to the source code. The Jolly Jumper links scenarios with the process view and development view [22].

‡Understanding and Recovery of the SOPHO Architecture.
• **MAB**: a Module Architecture Browser, visualizing the structure and the intradependencies of the system by means of tables. The MAB supports the development view [25]. The notion module architecture may be considered as part of the development view [17].

• **ArchiSpy**: a module architecture verifier, determining whether or not the software conforms to a set of architectural rules (or constraints) for the development view, and reporting breaches. ArchiSpy is basically a quality assurance tool at the architectural level, and is based on a relational calculator [26].

The Jolly Jumper is meant for all software developers, and primarily used for program understanding during both corrective and perfective maintenance activities. The MAB is used for both program understanding and (qualitative) complexity control. The MAB has been used for a number of perfective maintenance activities, and has proven to serve its purposes. ArchiSpy is exclusively used for (quantitative) complexity control. Medio 1999, it was the exception rather than the rule that the consequences of maintenance activities on the module architecture were mapped out from the beginning and explicitly controlled during all stages of these activities. Moreover, although ArchiSpy was integrated into the development environment, it did not constrain software developers. As a consequence, the conformance check performed by ArchiSpy regularly reveals breaches as a reflection on the result of a maintenance activity. Both the MAB and ArchiSpy are, in the first instance, primarily aimed at software architects and designers; the average developer has less affiliation with this complexity control support of URSA. URSA became available to all developers at the end of 1998.

As described in [7], the Jolly Jumper did not reach the same level of quality as the MAB during the development of URSA. During the past four years, the Jolly Jumper was re-implemented from scratch using SDE-specific technology, motivated by its quality, amongst others. The Jolly Jumper is now fully integrated with the SDE, and used in daily practice.

Unlike the Jolly Jumper, neither the MAB nor ArchiSpy are in use any longer, and neither of these tools is currently operational.

### 5.1.3. Pillars of C-POSH

**Process.** In the summer of 1999, ‘P’ was very close to a classification as a CMM level 2 organization. It fully satisfied most (but only partly satisfied some) of the key process areas (KPAs) of level 2, and already satisfied some KPAs of level 3. During the past four years, ‘P’ invested in a Software Process Improvement (SPI) programme. In 2002, it received the internal Philips SPI Award for effort and accomplishment. A recent CMM quick scan assessment resulted in a classification as a full CMM level 3 organization.

The organization, therefore, has subjects as (coding) standards and inspections of software artifacts in place, and satisfies all the basic requirements for embedding AS sustainably.

**Organization.** Management within the organization of PBC recognized that the price to be paid for the success of its legacy system was the steadily increasing complexity of the system and related difficulty in understanding it. In the literature (e.g., [4,5,24]), it is stressed that software architecture plays a vital role in the development (and hence maintenance) of large software systems. Management therefore decided in the mid 1990s to explicitly introduce the element of architecture in general, and
of software architecture in particular, within PBC. Upper management became the sponsor, and a local champion was chosen as the change agent. The change agent was asked to make the organization aware of the role of architecture, and roles were defined for the product architect (carrying the main responsibility for the logical view of the system) and the software architect (carrying the responsibilities of the other views). Moreover, the quest for program understanding combined with the need to manage the software architecture explicitly on the one hand and the lack of appropriate COTS tools at the architectural level on the other hand led to the development of URSA. Support at the programming level was left to standard tools that are readily available on the market. In order to give all persons involved in software development a common understanding of the role of software architecture within PBC, an architecture awareness course was developed. The need to control architectural changes and the use of URSA is also explained within the course.

Four years ago, both the company angel as well as the change agent left PBC. Moreover, program management (i.e., managers who decide which new functionality will be developed) became responsible for the software architecture and the software architects also became accountable to program management. The management of the department that actually developed the software was neither responsible nor involved any longer. During the past four years, the software architectural focus vanished within the organization, and the role of software architect gradually disappeared.

SDE. As described in [7], the software core was ported from a proprietary operating system towards a COTS real-time operating system, and was entirely written in C++ medio 1999. The changes in the operating system and the programming language may be considered as preventive maintenance, improving future maintainability and reliability, and providing a better basis for future enhancements. Although preventive maintenance is said to be ‘rare in the software world’ (see, for example, [1]), we are convinced that this legacy system is still maintainable and commercially successful due to, amongst other things, these preventive maintenance activities.

During the past four years, the software core was also ported from the target-platform towards the host-platform. This latter port made all the standard testing and debugging aids on PCs available to the software developers. Moreover, whereas in the past many tests could only be performed using the target-platform in combination with applications running on PCs, it is now possible to perform these tests on the host-platform only. This combination of steps resulted in major improvements in software development. In particular, it reduced the lead-time of testing by an order of magnitude, and resulted in substantial savings on the number of target-platforms needed.

Nowadays, the target-platform is exclusively used for function integration tests (e.g., can the caller actually hear ringing tone when the callee is ringing) and stress-tests.

Humans. Within PBC, it is common practice that software developers participate in all phases of the development cycle, from concept and feasibility (which includes architectural issues) up to and including testing. Due to their involvement, there is a strong sense of co-ownership of all software artifacts. Moreover, software developers participate actively in the service organization to solve problems that are reported in the field as a compulsory service. They therefore get feedback from the field on the results of their efforts, and have a strong incentive to produce high-quality software. Historically, the organization has a low turnover of personnel, implying that the average developer is not only highly motivated, but also highly experienced with both the product as well as its supporting tools.
The deployment of URSA required changes in the development process. Knowledgeable representatives of URSA’s target group (the ‘early adopters’) were therefore involved right from its early conception and their requirements, and the feedback on intermediate versions of the tools was treated as those of regular customers, and concluded with an acceptance test. The architecture awareness course mentioned above was also considered a necessary and essential step for the deployment. Architectural awareness alone is not sufficient, however. An active participation of the software developers is of paramount importance to ensure that the state-of-affairs of the software architecture does not deteriorate, and preferably even improves. Getting the methodological aspects and the underlying principles across turned out to be hard, and a lack of coaching contributed to the failure to embed complexity control at the architectural level sustainably within PBC.

5.1.4. Summary

In retrospect, the envisioned changes in the development process to support program understanding and complexity control at the architectural level by URSA turn out to be only partially sustainable. Whereas the Jolly Jumper is fully integrated with the current SDE and used in daily practice, both the MAB and ArchiSpy are no longer in use. This may not come as a surprise given the observation that the deployment of software architecture within PBC was not successful. The software architectural focus vanished over the past four years, and the role of the software architect gradually disappeared. As a result, both the MAB and ArchiSpy, which were primarily aimed at software architects and meant for complexity control, disappeared as well. This lack of success can partially, but certainly not exclusively, be traced back to the premature departure of both the company angel and the change agent. Embedding complexity control at the architectural level turned out to be hard, not from a tool perspective, but from a methodological, organizational, and human perspective. Moreover, these tools did not pay-off to the same extent to the average developer as the port of the software core to the host-platform. Viewed alternatively, the preservation of the Jolly Jumper illustrates its added value for the organization. The fade-out of the other two tools illustrates their lack of value for the organization to justify preservation, in particular when compared to other improvement actions that were instantiated over the past four years.

5.2. Case study PMS: ‘Q’

5.2.1. Introduction

‘Q’ produces and sells a variety of medical imaging systems, ranging from low-end systems used for diagnostic purposes to high-end systems used for complex medical interventions. These systems may be viewed as distributed, heterogeneous systems. In the past, these systems were mainly developed independently from each other. Their increasing size and complexity combined with demands for a decreasing time-to-market required the development efforts to be bundled. It was therefore decided in 1996 to set up a product line architecture that could serve as a basis to build this variety of systems, using the architectures of the existing products as the input. Hence, only the logical view of the software architecture remained unaltered. Setting up this product line architecture from scratch was an excellent opportunity to make a major step forward in technology as well: coming from VAX/VMS, it was decided to build the new systems on Windows NT using C++, MFC and Microsoft DCOM.
(Distributed Component Object Model) as the realization technology. More details about this product line architecture and the realization technology used can be found in [27].

This revolutionary approach of developing a new family of medical imaging systems from scratch using state-of-the-art technology may be viewed as preventive maintenance.

5.2.2. AS

The vital role of the software architecture of this new product line was recognized right from the start of this preventive maintenance activity. The software architecture should not only be maintained, but the software architecture as realized in the implementation of the system should also conform to the software architecture as specified.

PR has developed a sophisticated set of proprietary tools termed FAVOR, which check the conformance of the system to a fixed set of system-wide so-called architectural rules for the development view. Apart from reporting breaches, the tool-set also automatically visualizes a number of important dependency structures by means of graphs, and provides support to link breaches at the architectural level to individual C++ methods at the code-level. More information about module architecture verification and the tools that support this process are given in [28].

In 2001, the construction phase of the embedding of FAVOR was completed. FAVOR is currently in its deployment phase, and the conformance checks are performed on demand.

5.2.3. Pillars of C-POSH

Process. The adaptation of a revolutionary changing the system resulted in a fall of the organization’s CMM level. As a result of the changes in ‘Q’, some KPAs at level 2 were no longer fully satisfied, and formally the organization is therefore currently at CMM level 1.

Organization. The development process for the product line is based on the approach described in [29], and therefore required substantial organizational changes.

Moreover, in order to build up expertise in the new technology, the organization invested in an education programme for all software developers, and organized on-site training in Windows NT, C++, MFC, and COM.

In this organization, the system architect was both the champion and change agent for the AS. He convinced management of the importance of the support, and the need to develop the methodology and associated tools in close co-operation with PR. The system architect was also the main customer of the AS, and formulated the major requirements for the tools. Although the software architects and software developers are intended users as well, they have not actively been involved until the deployment stage of the embedding phase of the transfer. Recently, the system architect left the organization.

SDE. The revolutionary approach caused major changes in the SDE, and tuning the standard build and integration process to the needs of the organization required ample time. The changes in the SDE clearly influenced the development and deployment of FAVOR as well. On the one hand, the use of Microsoft C++ in combination with COM provided additional technical challenges to FAVOR (compared with systems that are built using plain C). Whereas dependencies in the source code can
simply be deduced from the include structure in C, one needs source browser files and proprietary COM definitions to extract these dependencies for Microsoft C++ in combination with COM. On the other hand, integrating FAVOR in a converging (rather than a stable) SDE requires regular adaptations to keep it operational. FAVOR has been transferred and is maintained by ‘Q’.

**Humans.** In general, people are reluctant to change, and changes are therefore hard to carry through, even when these changes are for the better. Many software developers had trouble with the number of changes caused by the revolutionary approach. Moreover, building an entire new product line is a huge investment. Given the amount of effort involved, it could not be accommodated by the existing developers alone, temporarily resulting in an unusual balance between fixed and hired personnel. This combination of aspects resulted in a relatively high turnover of personnel. Finally, it is disputable whether the introduction of AS was experienced by the average software developer as a desirable aid in development, or as just another complicating factor.

5.2.4. **Summary**

The revolutionary change to the development of the software of the medical imaging systems had a huge impact on the process, the organization, the SDE, and the humans. Although the AS is extremely suited for the approach, it is just another change on top of a pile of changes, putting its sustainability at stake. Time will tell whether or not the envisioned changes are sustainable within the organization.

5.3. **Case study PMS: ‘R’**

5.3.1. **Introduction**

The software developed by ‘R’ has been under development for approximately 20 years. An important step was the department-wide introduction of object-oriented programming in 1998, replacing C with C++ (and recently also with C#) as one of the main languages. C and other languages are still used, especially when hardware specific code is required, but C++ is preferred if possible. Due to market demands, competition considerations and new technologies, a new system (a so-called preferred configuration, being a member of the product family) is released once a year. This has resulted in a continuously evolving code base that has adopted many new technologies of which COM and .NET are the most important.

5.3.2. **AS**

Several proprietary tools have been developed to provide AS both for the **development view** and the **process view** of the architecture. More details about the tools that support the development view are given in [20,30,31].

For explanatory purposes, we first introduce some terminology to ease the subsequent presentation of the AS.

The decomposition structure of the software is an unbalanced tree of architectural entities that are called **building blocks** (BBs). The system consists of hundreds of BBs, and each leaf BB consists of approximately eighty **modules**. In Figure 5, the decomposition structure of a simple system \( S \) is shown.
consisting of 10 BBs (i.e., $S$, $A$, $B$, $A_1$, $A_2$, $A_3$, $B_1$, $B_2$, $A_{21}$, and $A_{22}$). The arrows in Figure 5 visualize the part-of relation, e.g., $A$ is part-of $S$. A tree-cut is a line crossing each path from the finest grain BBs (e.g., $B_1$) to the root (i.e., $S$) exactly once, where each path is crossed ‘through’ a BB (see [32]). The BBs on the tree-cut are termed subsystems (see Figure 5).

The overall structure for documentation, source code, and planning has been standardized on BBs. Interfaces are defined (both syntactically as well as semantically) and explicitly managed at the subsystem level. These interfaces are the responsibility of the software developers maintaining the subsystems. The dependencies between subsystems, i.e., the uses relation between subsystems, are constrained by the software architects. Constraints on the uses relation within subsystems are the responsibility of the software designers, and automated tool support for those constraints is currently under construction.

The actual software (i.e., code) is constrained by coding standards. Adherence to the coding standards is enforced through reviews, and no software artifact can enter the archive without approval.

The performance (in terms of CPU-usage, disk-access and network-bandwidth) and memory usage of the BBs is monitored on a daily basis.

5.3.3. Pillars of C-POSH

Process. Last year, ‘R’ was assessed as a CMM level 3 organization by an accredited third party.

Organization. The organization has a clear architectural focus; management at both the department level as well as the project level recognize the vital role of software architecture for their products, and the organization has well-defined roles for software architects. The organization is also conscious of the importance to manage change, and the key roles are filled in to accomplish change, such as the embedding of the technology to constrain the dependencies between subsystems. Changes are typically introduced in an evolutionary setting, such as the introduction of new software architecture artifacts.
The technology transfer process has been split into two parts, a front-end up to and including the feasibility stage, and a back-end covering the embedding stage only. Whereas PR may be involved in the front-end part, the back-end part of the process is exclusively done by ‘R’ itself. In particular, tools developed by PR are re-implemented by ‘R’ in the construction phase of the embedding stage to ensure that the tools meet the standards of ‘R’.

**SDE.** Whereas the architecture verification tools described in Sections 5.1 and 5.2 only report breaches as a reflection on a maintenance activity, the AS of ‘R’ prevents violations. Prevention is accomplished by means of proprietary build management tools (reflecting the ‘make’ command). Prevention is enabled through so-called include scoping, that is realized by means of ‘compiler flags’. All modules of a particular subsystem are only allowed to use the interfaces of subsystems on an ‘allowed to include’ list that is managed by the software architect. Statements like ‘#include full path name’ are not only prohibited by the coding standards, but also do not work on the separate build server.

**Humans.** The software developers are used to work according to standards and procedures, and they are technology-oriented. Moreover, SPI activities have already been known (and accepted) for many years, developers actively participate in SPI activities, and SPI is therefore a natural part of daily work.

### 5.3.4. Summary

‘R’ is a mature software development organization, that is used in SPI activities. Moreover, the organization recognizes both the vital role of architecture and the importance of managing change. Although embedding AS sustainably did not happen by itself, it went as a matter of course given a champion. Finally, this organization builds on the experience gained by ‘P’, amongst others, and has adapted its processes accordingly. As examples, it has split the technology transfer process into two parts and its AS prohibits violations rather than reports breaches of architectural rules.

### 5.4. Comparing the cases

Table I gives an overview of the case studies presented in Sections 5.1–5.3.

In Figure 6, we have illustrated the maturity to adopt and sustain AS in the three organizations considered in the case studies. The figure uses four dimensions, corresponding to the C-POSH pillars that carry sustainable AS, as described in Section 4. The three dots on the lines of the dimensions represent ‘low’, ‘medium’, and ‘high’ maturity to adopt AS (from the centre of the figure to its boundary). Although the figure is based on a subjective judgement of the organizations by the undersigned, we believe that it fairly reflects our experiences.

For all three cases, the changes come from within the Philips organization, making it an internally driven change. For all three cases, the changes are directed at design and production processes. It is disputable, however, if the changes were incremental or revolutionary in nature. In the following sections we first show that the approach taken in ‘R’ is relatively evolutionary compared to the approaches taken in ‘P’ and ‘Q’. We then present a summary of our findings.
Table I. C-POSH for three cases (rank: + = high, □ = medium, − = low).

<table>
<thead>
<tr>
<th>Case</th>
<th>Pillar</th>
<th>Rank</th>
<th>Reasons</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘P’</td>
<td>Process</td>
<td>+</td>
<td>CMM level 3</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>−</td>
<td>Invested in SPI</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>−</td>
<td>Less software architecture focus</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>−</td>
<td>Company angel disappeared</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>−</td>
<td>Change agent disappeared</td>
</tr>
<tr>
<td></td>
<td>SDE</td>
<td>+</td>
<td>SW test environment on developer machines (+)</td>
</tr>
<tr>
<td></td>
<td>SDE</td>
<td>+</td>
<td>Move from proprietary towards COTS tools (+)</td>
</tr>
<tr>
<td></td>
<td>SDE</td>
<td>+</td>
<td>Move from VAX/VMS to Windows operating system (+)</td>
</tr>
<tr>
<td></td>
<td>Humans</td>
<td>−</td>
<td>Focus on design rather than on architecture</td>
</tr>
<tr>
<td></td>
<td>Humans</td>
<td>−</td>
<td>Lack of coaching</td>
</tr>
<tr>
<td>‘Q’</td>
<td>Process</td>
<td>−</td>
<td>CMM level 1</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>□</td>
<td>System architect was both the change agent and company angel (+)</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>□</td>
<td>System architect recently left the organization (−)</td>
</tr>
<tr>
<td></td>
<td>SDE</td>
<td>□</td>
<td>Changes in SDE influenced AS tool development as well</td>
</tr>
<tr>
<td></td>
<td>SDE</td>
<td>□</td>
<td>Many changes due to revolutionary approach</td>
</tr>
<tr>
<td></td>
<td>Humans</td>
<td>−</td>
<td>Unusual balance between hired and fixed personnel</td>
</tr>
<tr>
<td></td>
<td>Humans</td>
<td>−</td>
<td>High personnel turnover</td>
</tr>
<tr>
<td>‘R’</td>
<td>Process</td>
<td>+</td>
<td>CMM level 3</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>□</td>
<td>Organization has clear architectural focus</td>
</tr>
<tr>
<td></td>
<td>Organization</td>
<td>□</td>
<td>Evolutionary changes</td>
</tr>
<tr>
<td></td>
<td>SDE</td>
<td>+</td>
<td>AS prevents violations via proprietary build management tools</td>
</tr>
<tr>
<td></td>
<td>Humans</td>
<td>□</td>
<td>Developers are used to work with standards and procedures</td>
</tr>
<tr>
<td></td>
<td>Humans</td>
<td>□</td>
<td>SPI is known and accepted</td>
</tr>
</tbody>
</table>

Figure 6. Maturity to adopt and sustain AS.
Scope implications on humans

Relation algebra with multi-relations (see [17,20,21]) provides the theoretical basis for the AS for the development view as introduced in ‘P’ and ‘Q’. The relational approach can be applied to all levels of the decomposition structure of a system, not only to the top (i.e., architectural) levels of the hierarchy, but all the way down to the methods at the programming level, thereby spanning the design and implementation levels as well. From a technical perspective, this may be viewed as a blessing, because it allows the application of a single approach for the entire hierarchy. From an embedding perspective, it involves a major risk. When the approach is applied indiscriminately, it will influence the entire group of software developers rather than just a few (i.e., those responsible for the architecture). By extending the scope of the approach, more people have to be involved in organizational learning. The extended scope may be viewed as another cause for the lack of success of the embedding of the MAB and ArchiSpy at ‘P’, and as a risk for the embedding of FAVOR at ‘Q’. Note that ‘R’ limited the scope of the AS by making a clear distinction between architectural entities (i.e., BBs from the top of the decomposition structure until the tree-cut) and other entities, and assigned responsibilities for those entities and the relations between them accordingly.

Process versus SDE

Methodological and automated tool-support for AS are not only complementary in nature, but are also partly exchangeable. Without appropriate tool-support, greater demands are made on the software process. As examples, ‘gaps’ in the automated tool-support may be bridged by inspections, and exceptions to rules have to be approved by software architects. Because the tools introduced in ‘P’ and ‘Q’ only report breaches to rules, whereas the tools of ‘R’ prohibit violations to rules, ‘P’ and ‘Q’ had to pay more attention to their process than ‘R’.

Technology development implications on SDE

Depending on the particular goals one has in mind with the AS, the amount of innovation and development needed may vary considerably. Whereas the AS developed for ‘P’ and ‘Q’ is quite sophisticated, the support for ‘R’ is very modest.

6. CONCLUSIONS

In this paper, we reported upon the experience gained in embedding AS in three organizations within Philips developing professional systems. Although all three organizations belong to the same company and all three are developing large complex systems with long lifetimes, the experiences gained are quite diverse, although there are certainly also similarities, such as embedding AS does not happen by itself and has a long lead-time.

We noticed to our surprise that the AS embedded in one organization was only partially sustained. Unlike our own expectation, part of the AS was apparently not indispensable for the maintenance of the system in the organization. This emphasizes the need for a regular evaluation of whether AS is still
beneficial for the organization. If this is not the case (anymore), then sustaining the embedded AS is no longer justified.

For the cases considered, we identified four C-POSH pillars that need to carry AS in order to ensure its sustainability: Process, Organization, SDE, and Humans. Moreover, we described the essential role of change management, and the importance of performing evolutionary changes (as opposed to revolutionary). Managing change of these pillars turned out to be at least as challenging as, and probably even more than, the development of the AS itself.

6.1. C-POSH experiences

Experience reveals that software architecture maintenance based on a mature process alone does not work, because the software architecture as realized may easily deviate from the software architecture as specified. Similarly, tool-support alone does not work either. A combination of a mature process and appropriate tools is a necessary, but not a sufficient condition for success. Without additional organizational support and humans carrying co-responsibility for the software architecture, software architectural erosion is hard to prevent. The four pillars in our C-POSH model have to be addressed to perform software architecture maintenance and to sustainably embed AS in industry.

In our experience, an organization as well as its people must have an architectural focus in order to sustainably embed AS. Similar conclusions have been drawn elsewhere for other kinds of support. As an example, [33] states that organizations and people with a process focus are more likely to accept process-centered environments (PCEs) and process-centered frameworks (PCFs). Likewise, organizations and people with a development focus are more likely to accept CASE-tools. The fact that the architectural focus has vanished within ‘P’ during the past four years contributed to the disappearance of the support of complexity control at the architectural level.

The revolutionary approach to the development of systems of ‘Q’ had a huge impact on all four pillars in general, and the process and humans in particular. The revolutionary nature of this approach is therefore a major risk for the sustainability of FAVOR.

Finally, we observed that the approach taken by ‘R’ is relatively evolutionary compared with the approaches of both ‘P’ and ‘Q’, and therefore more likely to be sustained. Note that this is a change management issue that is orthogonal to the elements distinguished in Figure 6.

The C-POSH pillars are non-technical aspects that are often difficult to address by purely technical people. It is therefore required for organizations to build-up capabilities that support the C-POSH pillars.

Introducing and embedding AS asks for a lot of effort and the C-POSH pillars should all be addressed. On the other hand, when the embedding has succeeded, it does not require a lot of effort to sustain its support. By nature people do not change their behaviour easily, which also means that when the AS is embedded well, it will last without too much effort for maintenance.

6.2. Future work

We argue that the C-POSH model is also feasible for a lot of other software engineering related subjects. A successful introduction and sustainable embedding of object orientation depends on more than just the selection of an object-oriented method and tools that support that method.
Similar to embedding AS, one may expect that C-POSH contains valuable elements to guide the embedding of object orientation in a company.

An important consideration is how much effort should be spent in change management of the four pillars of software development. For the changes to be sustainable, each of the four C-POSH pillars should be sufficiently changed. A ‘critical mass’ of change has to be achieved, after which the effort spent to change management can be reduced. Case ‘P’ showed that the departures of the company angel and the change agent were premature, hence, the AS was not sustainably embedded.

Of course, as stated before, regular evaluation of the (economic and/or strategic) benefit of AS is desirable. Like many other quality improvement processes, one should follow the quality cycle or Demming wheel (plan–do–check–act) (see [34]). If the AS yields insufficient benefit for the organization, the change management effort should be stopped.

ACKNOWLEDGEMENTS

The authors would like to thank Eelco Rommes, Tobias Rötschke, Jan Gerben Wijnstra, Pierre America, and Jaap van der Heijden from PRE, Ben Pronk and Jan-Willem Dijkstra from PMS, and Erik Oerlemans, Piet Hubers, and Enno Douma from PBC. Furthermore, we wish to thank the anonymous reviewers of an earlier version of this paper.

REFERENCES


AUTHORS’ BIOGRAPHIES

Reinder J. Bril received a BSc and a MSc (both with honours) from the Department of Electrical Engineering of the University of Twente, The Netherlands. Since 1985, he has been with Philips. He has worked in both Philips Research as well as Philips Business Units, on various topics, including fault tolerance, formal specifications, and software architecture analysis, and in different application domains. For the last four years, he has worked at Philips Research Laboratories Eindhoven (PRLE), the Netherlands, in the area of Quality of Service (QoS) for consumer devices, with a focus on dynamic resource management in receivers in broadcast environments (such as digital TV sets and set-top boxes). He is currently detached for a period of one year by Philips to the Eindhoven University of Technology (TU/e), Department of Mathematics and Computer Science, Group System Architecture and Networking (SAN).
René L. Krikhaar is a software architect at Philips Medical Systems in Best, the Netherlands. He has more than 15 years experience in software engineering. His research interests concern product families, recovery, analysis and improvement of software architectures, development processes and software evolution. He holds a Masters degree in Computing Science from the University of Nijmegen, a Masters degree in Knowledge Engineering from the University of Middlesex and a PhD in Software Architecture Reconstruction from the University of Amsterdam.

André Postma is senior scientist at Philips Research Laboratories Eindhoven, the Netherlands. He has 10 years experience in software engineering. His research interests include analysis of software architecture, and reliability and performance engineering. He works in these research areas in close co-operation with one of the Philips Business Units. He holds a Masters degree in Computer Science from the Hogeschool Heerlen and Technical University of Eindhoven, and a PhD degree in ‘Classes of Byzantine Fault-tolerant Algorithms for Dependable Distributed Systems’ from the University of Twente.