The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study

Martin Feilkas and Daniel Ratiu and Elmar Jürgens
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
feilkas|ratiu|juergens@in.tum.de

Abstract

Architecture defines the components of a system and their dependencies. The knowledge about how the architecture is intended to be implemented is essential to keep the system structure coherent and thereby comprehensible. In practice, this architectural knowledge is explicitly formulated only in the documentation (if at all), which usually gets outdated very soon. This leads to a growing amount of implicit knowledge during evolution that is especially volatile in projects with high developer fluctuation.

In this paper we present a case study about the loss of architectural knowledge in three industrial projects to answer the following research questions: 1) to what degree is the architectural documentation kept in conformance with the code? 2) how well does the documentation reflect the intended architecture?, 3) how big is the architectural decay?, and 4) what are the causes for nonconformances? We answer these questions by investigating the architecture documentation, the source code, and by performing interviews with developers.

The most important outcomes of our study are: the informal documentation and the source code are not kept in conformance with each other, none of them completely reflects the intended architecture, and even developers taken individually are not completely aware of the intended architecture. Quantitatively, between 70% and 90% of these nonconformances are caused by flaws in the documentation and between 10% and 30% represent architectural violations in the code.

1 Introduction

The architecture defines the structure of a software system in terms of components and (allowed) dependencies. A suitable architecture is a fundamental prerequisite for evolvable and understandable systems [5]. Developers need knowledge about the intended architecture of a system whenever they do any modification. Without this knowledge, programmers can break the architectural integrity of the system accidentally, even when making only small code changes.

Today's widely used programming languages offer only very primitive mechanisms for making the architecture in the code explicit. Therefore, in everyday industrial practice, the information about the architecture is contained in external documentation in form of diagrams and natural language texts that often originate from early phases of the system design. During system evolution, the architecture often needs to be adapted, extended and modified in response to changes to the requirements, additional features or simply new insights about shortcomings of the initial design. These changes are inevitable even if an 'optimal design strategy' is used [13]. Needless to say, this effect is amplified in an industrial environment. When these changes to the intended architecture happen, they are often (unintentionally) not introduced into the architecture documentation and not propagated to other team members [10]. This leads to a gap between the intended architecture of the system, how different developers perceive it, how it is made explicit in the documentation and how it is actually implemented in the code.

Figure 1-left illustrates the ideal situation: All developers possess accurate knowledge about the system's architecture, the architecture is accurately documented and accurately implemented in the code. The right side of the figure illustrates the situation typically encountered in industrial projects: Different developers understand the architecture of big systems in (slightly) different manners, with none of them having an accurate view of the intended architecture. Furthermore, only a part of the intended architecture is documented and only a part of the code complies with it. As depicted in Figure 1-right, the *loss of architectural knowledge* can be observed in different forms: missing architectural information in the documentation, violations of

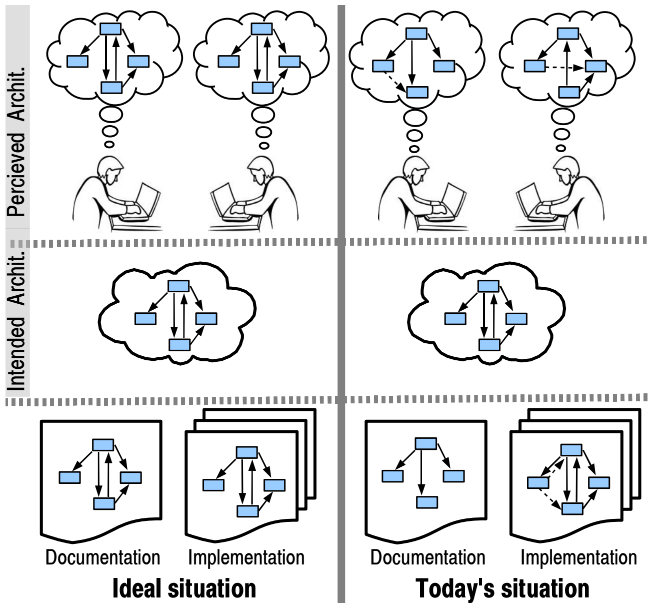


Figure 1. Loss of architectural knowledge

the architecture in the code, problems in keeping the code and documentation synchronized and different perceptions of the intended architecture by the developers.

In this paper we present an industrial case study on the loss of architectural information during the evolution of industrial software. We quantify to what degree the documentation and the code are kept in conformance. Furthermore, by means of developer interviews, we evaluate whether the differences are violations of the architecture in the implementation or if they are insufficiencies of the documentation. This case study has been done on three industrial business information systems of different age and functionality.

Outline. In Section 2 we briefly describe our approach for analyzing the conformance between the documented architecture and the implementation. In Section 3 we present three case studies we performed in a collaboration with our industry partner Munich Re. In Section 4 we discuss the lessons that we learnt from these case studies. Section 5 discusses threats to validity that could influence our conclusions. We end this paper with presenting related work in Section 6 and conclusions in Section 7.

2 Technique and methodology for architecture analysis

In this section we present our approach to describe the architecture in a machine-readable form and explain our technique to analyse the conformance between the documented architecture and the code. We exemplify our approach on C#, although the technique can be generalized easily, and in effect already supports all .NET languages and Java.

2.1 Architecture conformance analysis

Describing the architecture. The architecture is specified in terms of a set of hierarchical components $comp$ and policies pol among them. An architecture description $arch$ can hence be formalized as

$$arch = (comp, pol).$$

Components serve as the main structuring entities. The hierarchy is expressed as a predicate

$$isSubComp : comp \times comp \to bool.$$

Policies can also be regarded as a predicate

$$pol : comp \times comp \to bool.$$

The component hierarchy defines policies between parent and child components:

$$isSubComp(c_1, c_2) \Rightarrow pol(c_1, c_2)$$

If there is a policy defined between two components, the implementation elements that correspond to these components may have dependencies between each other (in the specified direction). Every dependency that is not explicitly allowed by such a policy is forbidden.

Describing the code. In object-oriented systems every element of the code is contained in a type (in .NET these types are defined as classes, structs, enums, ...). For architecture conformance analysis, a system can be regarded as a set of types $types$ and a set of dependencies dep between them:

$$sys = (types, dep)$$

The number of *dependencies* is expressed by the function:

$$dep : types \times types \to \mathbb{N}$$

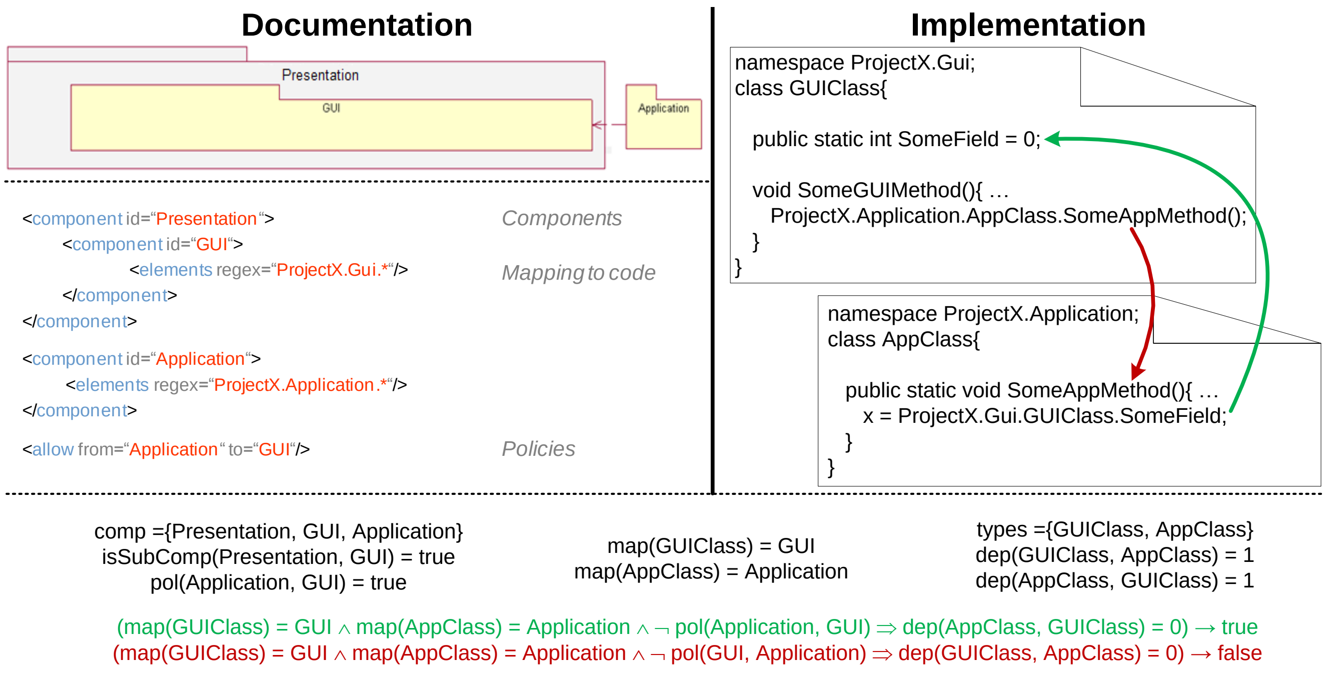
A type t_1 is dependent on another type t_2 if t_2 (or one of its elements) is used in t_1 as defined in Table 1.

Invocation of a method/constructor
Access of a property or field
Extending a class/struct, implementing an interface
Usage of a class/struct/enumeration as a type (for a field, variable or parameter)
Annotation of an attribute

Table 1. The dependencies in the code

Checking conformance. To map the architecture description to the system, we need to define a *code mapping* as a function $map : types \to comp$. If the architecture description is completely in conformance with the code the following condition must hold:

$$map(t_1) = c_1 \land map(t_2) = c_2 \land \neg pol(c_1, c_2) \Rightarrow dep(t_1, t_2) = 0$$



```
<component id="Presentation">
    <component id="GUI">
        <elements regex="ProjectX.Gui.*"/>
    </component>
</component>
<component id="Application">
    <elements regex="ProjectX.Application.*"/>
</component>
<allow from="Application" to="GUI"/>
```

```
namespace ProjectX.Gui;
class GUIClass{

  public static int SomeField = 0;

  void SomeGUIMethod(){ …
    ProjectX.Application.AppClass.SomeAppMethod();
  }
}
```

```
namespace ProjectX.Application;
class AppClass{

  public static void SomeAppMethod(){ …
    x = ProjectX.Gui.GUIClass.SomeField;
  }
}
```

comp ={Presentation, GUI, Application}
isSubComp(Presentation, GUI) = true
pol(Application, GUI) = true

map(GUIClass) = GUI
map(AppClass) = Application

types ={GUIClass, AppClass}
dep(GUIClass, AppClass) = 1
dep(AppClass, GUIClass) = 1

$(map(GUIClass) = GUI \wedge map(AppClass) = Application \wedge \neg pol(Application, GUI) \Rightarrow dep(AppClass, GUIClass) = 0) \rightarrow true$
$(map(GUIClass) = GUI \wedge map(AppClass) = Application \wedge \neg pol(GUI, Application) \Rightarrow dep(GUIClass, AppClass) = 0) \rightarrow false$

Figure 2. The architecture description mechanism

Every time when no policy is defined ($\neg pol(c_1, c_2)$) but x dependencies are identified between the types t_1 and t_2 ($dep(t_1, t_2) = x, x > 0$) then we consider these x dependencies as deviations from the architecture description.

Technical execution of the analysis. The architecture is specified in machine-readable form in an XML-file. Figure 2-left shows an example architecture: it contains three components (i.e. Presentation, GUI and Application), GUI is a subcomponent of Presentation and there is a policy defined that allows the Application component to depend on GUI. The figure illustrates how this simple example is described using XML. It contains a simple XML description of hierarchical components and their mapping to types in the source code based on regular expressions. These expressions are used to map the fully qualified names of the types in the implementation to the components. Additionally, allow-tags are defined to describe the policies of how the components may depend on each other. The right hand side of Figure 2 illustrates the implementation level: A green arrow represents an allowed dependency from the Application to the GUI component. The red arrow illustrates a dependency that violates the specification on the left hand side due to an access from the GUI to the Application component. The formalisation is given in the lower part of the figure.

All projects that are subject to the case study are implemented using the .NET framework. The analysis is performed using the Continuous Quality Assessment Toolkit (ConQAT) [2][1]. Given the XML architecture description and the binaries of the system, ConQAT performs the architecture conformance analysis and calculates the set of dependencies that are not explicitly allowed in the architecture description.

2.2 Methodology

Analysis steps. Figure 3 illustrates the steps needed to discover the intended architecture of a system:

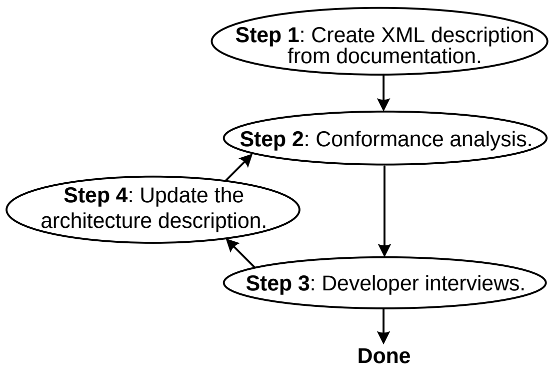

Step 1: Create XML description from documentation.
Step 2: Conformance analysis.
Step 4: Update the architecture description.
Step 3: Developer interviews.
Done

Figure 3. Analysis steps

Step 1: Translation of the architecture documentation into the machine-readable XML representation. In this step we obtain an XML description of the architecture as it was initially documented.

[1]www.conqat.org

Step 2: Checking the conformity of the code with respect to the current XML architecture description. This step uses the automatic analysis and has a list of nonconformances between the XML document and the code as a result. These nonconformances are either due to an insufficient description of the intended architecture in the current XML-file or violations of the intended architecture in the code.

Step 3: Discussion of the results with the developers. We discussed the results of step 2 with the developers to classify the differences either as violations of the intended architecture at code level or as deficiencies in the current XML architecture description. This classification uses the implicit knowledge of developers about the system's architecture. If there are differences in the output of the analysis that are not regarded as violations, the XML description of the architecture does not yet represent the intended architecture completely. In this case the architecture description has to be adapted by performing step 4. If the developers regard all of the differences as code deficiencies, then the process is complete.

Step 4: Refinement of the architecture description by considering the implicit knowledge that was not present in the documentation and revealed in step 3. After that, step 2 has to be performed.

After each iteration, the architecture defined in the XML description converges towards the intended architecture. Every modification that is necessary in the XML description (step 4) during our iterations is regarded as a flaw in the original documentation due to the changes in the architecture that were not documented (due to *architectural drift* [11]). After two to four iterations of the steps 2, 3 and 4, the architecture description was regarded as a precise specification of the intended architecture by the team members. Using the final XML description that contains the intended architecture, we are able to perform a final architecture analysis to measure the violations in the code of the intended architecture (and thereby to measure the *architectural decay* [11] of the code).

Outputs of the analysis. During the analysis process we compute the following sets:

- *Missing*: The set of components that are implemented in the system but are missing in the documentation.

- *Relocated*: The set of components that changed their super components. A component x is called a 'relocated component' if $isSubComp(x, a)$ is specified in the documentation and $isSubComp(x, b)$ reflects the intended architecture ($a \neq b$).

- *Policies*: The set of policies that were introduced or modified during the process of the analysis (in step 4).

- dep_{all}: The set of dependencies between the components in the system:

$$|dep_{all}| = \sum_{t_1 \in types} \sum_{t_2 \in types} dep(t_1, t_2)\ , \quad t_1 \neq t_2$$

- $diff_{doc}$: The subset of dependencies ($diff_{doc} \subset dep_{all}$) that represent differences between the original documentation of the architecture and the implementation. This set is computed after the first run of step 2. These differences reflect the divergence between the documented and the implemented architecture.

- $diff_{intend}$: The subset of $diff_{doc}$ that the architecture analysis revealed after the whole process was finished. The architecture description obtained after the iterations reflects the intended architecture. Therefore, all dependencies in $diff_{intend}$ are violations in the code.

3 The case study

We start this section with presenting the research questions addressed by our case study, next we describe the experimental setup, present the quantitative results of our analyses, and give an interpretation of the measured results.

3.1 Research questions

Q 1: To what degree is architecture documentation kept in conformance with the implementation during system evolution? If differences can be identified, this indicates that either the implementation violates the intended architecture or that modifications of the architecture during the system evolution are not propagated to the documentation. We answer this question by calculating the amount of differences relative to the dependencies in the code:

$$docdiff = \frac{|diff_{doc}|}{|dep_{all}|}.$$

Q 2: How well does the documentation of the architecture reflect the intended architecture? A vague or outdated architecture description is inadequate for conserving architectural knowledge for software maintenance and evolution. The documentation represents explicit knowledge about the architecture that the team members can always refer to. If no precise and up-to-date documentation of the architecture exists, new project members will have difficulties in learning the architecture. We measure the amount of implicit knowledge in terms of documentation flaws defined as:

$$flaw_{doc} = \frac{|diff_{doc}| - |diff_{intend}|}{|diff_{doc}|}.$$

Additionally, the numbers of components that were undocumented $|Missing|$ or relocated $|Relocated|$ as well as the policies that had to be modified $|Policies|$ are metrics for measuring the divergence between the documented and the intended architecture.

Q 3: How big is the architectural decay? This question should clarify to what degree violations of the intended architecture can be found in the code. The architectural decay can be measured in terms of violations:

$$flaw_{impl} = \frac{|diff_{intend}|}{|diff_{doc}|}$$

Q 4: What are the causes of nonconformances between the intended architecture and the code? We investigate why architectural violations were introduced into the code as well as why documentation is not kept up-to-date. This is a qualitative question that we answer based on the interviews with developers (compare step 3).

3.2 Experimental setup

Industrial environment. The case study was done in a collaboration between Munich Re and Technische Universität München. Munich Re is a big reinsurance company with about 39.000 employees worldwide. The insurance branch makes heavy use of individually developed software to support business processes such as sales, risk calculation or capital investments. At Munich Re, software development is done mainly by external developers. This leads to relatively high fluctuation of developers. A specially tailored RUP-like engineering process is applied to every software project. This process prescribes that an architecture documentation must be created for every project. To ensure a seamless hand-over between developers, the maintainability of the software products must be high. Thus, the systems must be implemented in a comprehensible and homogeneous way. A prerequisite for these desiderates is to manage architectural knowledge by making it explicit.

The projects. We investigated three typical business systems implemented in C#. We emphasize the fact that at Munich Re these projects are considered to be of good quality and successful. They are in productive use by 10 to 150 users in different departments of Munich Re. All of these systems have been developed by developers from different software development contractors. During the initial development there were up to 12 developers involved in each project. After these systems went into maintenance, the number of developers was reduced. The developers are constantly evolving the systems.

Project A is a typical rich client application that is further developed and maintained. The system is in production for about 5 years. It provides insurance risk calculation functionalities. Currently 5 developers maintain the system. There has been personnel turnover so that there is currently none of the initial developers in the team. The architecture documentation of Project A is a text document that contains a component diagram consisting of hierarchical components (boxes). These components are connected via arrows that represented allowed dependencies. This diagram was the most important source of architecture information.

Project B is a web-based information system. With a lifetime of 6 years in production this is the oldest of the investigated projects. It is used for doing financing and investment calculations. Currently there are 4 developers involved in maintaining the project. Like in Project A, there have been personnel fluctuations so that none of the initial developers works in the project anymore. In Project B, the architecture is described by diagrams similar to UML package diagrams.

Project C is also a web-based system. It is the youngest of the systems and under development for about 2 years. It provides functionality for managing risk information about certain clients of Munich Re. An average number of 3 developers maintain the system. It is in productive use only for about 7 months. The architecture documentation of Project C is also a text document that contains diagrams that illustrate components and their relations by boxes and arrows.

	Age	**kLOC**	**Developers**
Project A	5	454	5
Project B	6	317	4
Project C	2	495	3

Table 2. Data concerning the projects

In Table 2 we present the projects at a glance: their age, code size and current number of maintainers. We remark that the size of the projects is in the same magnitude. However, Project C is the biggest even if it was the most recently started and even if it has the least number of developers assigned for maintenance and evolution.

Table 3 shows the size of the documented architecture in terms of the number of components and policies. We should also remark the big number of policies in the case of Project A and C (they contain only 24, respectively 37, components and define many allow-policies). In contrast to projects A and C, Project B contains much more components (60), although it contains only about the same number of policies (106) as Project C. So the description of Project B has a much finer granularity and is more precise and restrictive.

	Components	**Policies**
Project A	24	79
Project B	60	106
Project C	37	102

Table 3. Initial description of the architecture

3.3 Quantitative Results

Table 4 presents the number of modifications of the architecture description during the iterative refinement. The first column shows the number of relocated components *Relocated*. For example, moving a subcomponent of the

Business component to the DataAccess component constitutes a component relocation. The second column represents the number of components that were not documented and were introduced during the analysis process (step 4). The third column shows how many policies have been modified or introduced in addition to the documented ones. No policies have been removed without substitution. This shows that the documented architecture usually has less policies than the architecture actually implemented. In other words, the documented architecture seems less coupled than the one that is really implemented.

	Relocated	***Missing***	***Policies***
Project A	0	2	8
Project B	6	2	24
Project C	2	1	9

Table 4. Modifications of the architecture description during steps 3 and 4

Table 5 shows the main results of our analyses based on the metrics defined in Section 3.1. The dep_{all} column presents the number of dependencies that the architecture conformance analysis identified in the systems. These dependencies were validated against the XML architecture descriptions. The $diff_{doc}$ column presents the results of the conformance analysis, namely the number of dependencies that did not conform to the documented architecture. The $flaw_{doc}$ and $flaw_{impl}$ columns contain the percentage of the non-conforming dependencies that are flaws in the *documentation* and respectively violations of the architecture in the *implementation*.

Proj.	dep_{all}	$diff_{doc}$	$flaw_{doc}$	$flaw_{impl}$
A	8.254	994 (12%)	90%	10%
B	4.385	376 (9%)	72%	28%
C	5.388	(2238) 1039 (19%)	88%	12%

Table 5. Results of the analysis

System A. In system A, 994 (out of 8.254) dependencies were identified as differences between the documented and the implemented architecture (12%). As Table 4 shows, 8 policies had to be added and 2 components were introduced in order to get from the documented to the intended architecture during the analysis process. After the completion of the analysis, about 10% of these differences were identified as violations of the intended architecture within the implementation. The other dependencies were mostly undocumented modifications of the architecture during the evolution of the system.

Figure 4 shows a visualization of the results of the analysis of system A. This kind of visualization, created using the graph visualization tool 'dot' [7], was integrated into the projects' dashboards and represents the entry point for obtaining feedback from the developers. Red arrows represent violations of the specified architecture, green ones are allowed dependencies. In addition to this visualization, detailed lists of the identified differences are used as inputs for discussion.

System B. The results of the analysis of Project B revealed a lower number of differences between the documented and the implemented architecture (376 out of 4.385). More adaptations of the architecture description were needed in Project B (Table 4) after the discussions with the developers to get a description of the intended architecture. The architecture description of this project was much more detailed and fine-grained in the documentation (Table 3). A relatively high share of architecture violations in the implementation (ca. 28%) of the 376 differences were identified.

System C. After the first analysis of system C based on the initial architecture description, we identified a very large number of differences (2.238 – effectively 42%). Closer investigations revealed that this extreme result had one primary cause: The system was built using the so called "data binding technique" offered by the .NET framework. This technique creates dependencies from GUI-parts directly to data access components, ignoring the intermediate BusinessLogic layer. These dependencies were not allowed in the initial documentation of the architecture. Although these dependencies are differences between the documentation and the implementation of the architecture, in contrast to usual architecture violations, these have not been introduced due to a developer's lack of knowledge about the architecture. For that reason, the XML architecture description was modified accordingly (i.e. we introduced a new policy to allow these dependencies), and a lower value of 1039 (ca. 19%) differences was measured. This value was used for the further steps of the case study. After completing the adaption process, 12% revealed to be architecture violations which is a very similar value to Project A. Also the number of adaptations of the documented architecture was in a similar range with Project A (Table 4).

3.4 Interpretation of the results and answers to the research questions

Q 1: *To what degree is architecture documentation kept in conformance with the implementation during system evolution?* Following our automatic analysis of the architecture conformance, a significant number of differences between the documented architecture and the implementation have been found in all three projects. As shown in Table 5, between 9% and 19% of all the dependencies containted in

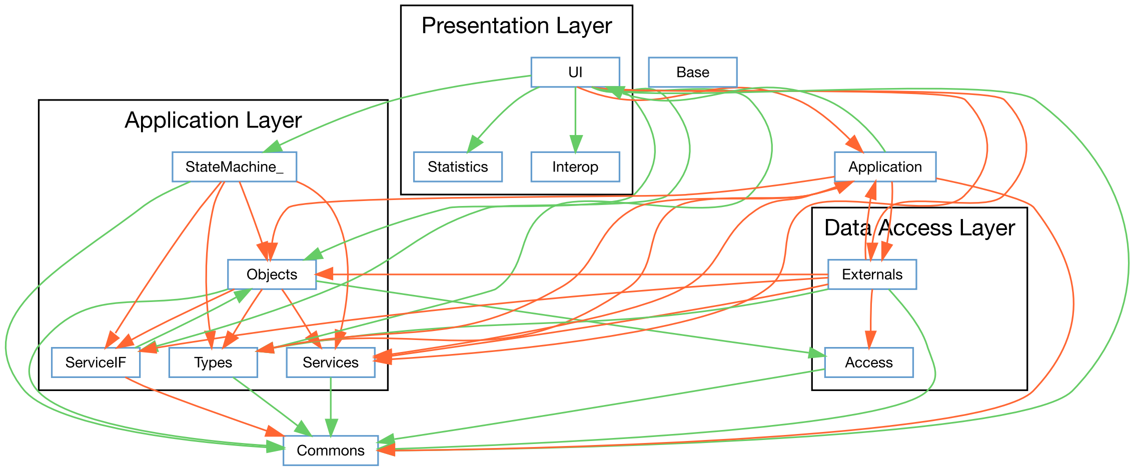


Figure 4. An example for the visualization of the results of the architecture analysis (Project A)

the implementations of the systems could not be identified in their corresponding architecture documentation. These differences represent either documentation flaws or violations of the intended architecture in the code. The *docdiff* values of the projects reflect that Project B has the least differences, but the amount of discrepancies is still significant (every tenth dependency in the system cannot be found in the documentation). In the worst case (Project C) almost one fifth of the dependencies does not conform to the documentation. This significant desynchronization between the documentation and the code led developers to regard the documentation as an unreliable source of architectural information.

Q 2: *How well does the documentation of the architecture reflect the intentions of the architect?* The documentation flaws were discovered by performing interviews with the developers (step 3). Many of these interviews caused vivid discussions among developers due to their different perceptions of the intended architecture. The outcome of these interviews were policies that had to be added to the architecture description. Table 5 shows that between 72% and 90% of the differences between the documented and the implemented architectures are due to flaws in the documentation. Table 4 summarizes the modifications of the initial documentation that were made during the analysis process. The projects A and C needed an almost equal amount of modifications. Project B caused more changes that had to be integrated into the architecture description. The reason therefore is the more fine-grained architecture definition in Project B.

In summary, most of the differences ($diff_{doc}$) must be regarded as deficiencies of the documentation. Assuming that the documentation reflects the architecture defined in the design phase, the documentation flaws are a measure of architectural drift, namely the measure in which the initially intended architecture developed further over time.

Q 3: *How big is the architectural decay?* The last column of Table 5 shows the architectural violations measured in the projects. The relative amount of architecture violations in the systems is between 10% and 28%. Although Project B has the highest relative value in that table (28%), the absolute number of violations discovered in the analysed systems is with about 100 dependencies almost equal. The fine-grained architecture description of Project B suggests that the analysis can be regarded to be more precise. Due to that more violations could be identified. Additionally, the architecture of system B is much more restrictive than the architecture description of the other projects (Table 3). As a consequence, fewer dependencies are allowed an thus developers are more likely to introduce violations.

Q 4: *What are the causes of nonconformances between the intended architecture and the code?* Most of the documentation flaws are caused by new insights during the implementation phase. For example, System A should contain no dependencies between the GUI and the DataAccess component. However, our analysis identified dependencies between these components. Interviews revealed that these were uncritical and even done on purpose because they were needed to access specific data during system startup. At startup time, the business components that are usually used for acquiring data are not yet available. This is an example for a refinement of the architecture that was performed during the implementation of the system (during the design phase it was overlooked that such a dependency is necessary). However, this knowledge has not been introduced into the documentation.

The developers regard the architecture documentation as a relict from the very beginning of the project when the architecture was initially created. So the architecture documentation is seen rather as an artifact that should ease the constructive phase (design) rather than a description of the system for long-term maintenance and evolution. The relatively high amount of documentation flaws is not due to 'laziness' of the developers. Often developers do simply not remember that there are parts in the documentation that should be modified. This can be regarded as a lack of availability of the architecture documentation based on text documents. Thus, documentation becomes a dead artifact that is used very infrequently. Many times the developers (especially in Project C) knew that the documentation was outdated. They explained that redocumentation activities are often postponed due to the higher priority of the implementation of new features or bug-fixes in their daily work.

During our analyses we identified several critical architecture violations. In Project A, for example, the architecture defines a set of interfaces as a facade to perform database access. However, our analysis revealed several accesses to the database without using this facade. Since the facade takes care of transaction handling, bypassing the facade leads to problems with the transaction handling. Thus, transactions were called in a non-adequate way (using the wrong interface). The effect was a significant loss of performance. These violations were caused by a developer that did not use the interfaces (the facade) that were intended to be used for that purpose. The explanation of the developers was that the implementer of these pieces of code did not know which components should be used to achieve these tasks. This shows that not all of the developers were aware of the intent and the adequate usage of the architecture.

Another example for a typical reason of a violation is copy and paste programming. Many times the headers of files have been copied and pasted to be used as a template for the implementation of new classes. Unfortunately, the namespace declaration was often part of the copied lines and it had been missed to modify it accordingly. However, although that seems not so critical, it is difficult for another developer to identify this as a copy&paste problem and to understand that the class should be declared somewhere else.

4 Lessons learnt

Architectural knowledge gets lost. Developers do not understand the complete architecture of the system and especially how it is reflected in the code. The main cause is the myriad of details at the code level and the big abstraction gap between architectural specifications and the implementation. Instead of performing tedious work for understanding and recovering (guessing) the intended architecture, we advocate approaches to conserve the architectural knowledge and how it is implemented in the code.

Conformance checking is a catalyzer for discussions. The case study revealed that a more structured documentation in a machine-readable form and an automatic analysis creates bigger architecture-awareness in the development team. Several discussions on the correct usage of the architecture were raised during the case study. Thereby, the different views on the intended architecture by the team members were synchronized and introduced into an explicit documentation.

Continuous architecture analysis. To ensure that the code will be kept in conformance with the architecture description in the future, we integrated the architecture conformance analysis into the nightly build of the projects. Thus, the continuously checked architecture description can be kept up-to-date as a specification of the intended architecture more easily. The architecture description is managed by the version control system so that the developers can use modify it easily. The results of the analysis can be accessed by the developers via a link in the project dashboard. Thereby, the architecture received a more central role within the projects. The continuously checked architecture documentation in machine-readable form had a better availability and visibility within the projects than the text documents. The developers agreed that due to the integration of the architecture knowledge with the system using a continuous analysis of the conformance between the architecture description and the system, a better way of conserving the architectural knowledge within the projects could be achieved.

Furthermore, the continuous assessment enables an early detection and resolution of potential architecture violations and design modifications. Thus, it can be decided very soon whether it is a violation or a design drift. So the costs of removing violations or adapting the documentation are rather low because the responsible developer still knows what she/he was working on.

Even several weeks after the main case study was done the developers reported that they inspected the results of the analysis in their project dashboards every morning. Due to the integration of the analysis into the nightly build, an active way of managing the architectural knowledge within the projects was achieved. The architecture description in XML form stayed up-to-date (synchronized with the code) at least for the time we stayed in contact with the project (which was about a year in case of Project A).

Efforts needed. The efforts of establishing the analysis, the configuration of the dashboard and the creation of the XML architecture description took about 5 days of work for each project. The most efforts took the reverse engineering and the discussions about how the appropriate architecture should look like. After these initial costs, the efforts of keeping the architecture description up-to-date using the continuous analysis were rather low. In Project A, only four architectural changes happened in about a year. But this may of course vary between different projects.

5 Threats to validity

Internal validity. Our evaluation of the loss of architectural knowledge is based on several assumptions that can potentially influence our results.

Hidden knowledge. The identification of the intended architecture of the three systems is based on iterative inspections of the difference between the documented architecture and the code. These differences represent the basis of our discussions with the developers. However, it could happen that the documentation and the code match well in a certain respect (even if they are biased from the intended architecture). In these situations our method does not identify the intended architecture. Such situations influence the completeness of our approach. However, the quantitative results are not influenced (in these situations the loss of architectural knowledge would be even bigger).

Translation from textual documentation into checkable form. The translation of the informal architecture documentation contained in text documents to the machine-readable XML-representation might affect some of the results measured. Some information contained in the informal documentation might have been overlooked or misinterpreted, hence the results measured after the initial execution of the analysis (step 2) might be biased. However, the same effect might take place when a developer that is unfamiliar with the system tries to learn the architecture by studying the documentation. Furthermore, in case of major misinterpretations, the developers should have noticed such a flaw. In that case they would have indicated such an issue.

The developers themselves do not know exactly what the architecture was intended to look like. In some situations we asked the developers and the architect about some details of the architecture and they were not able to answer our questions immediately. There have been some questions that they had to discuss within the team before they could give us precise information about the architecture-to-be. This effect may influence the results of the case study because such a team decision might not reflect the intended architecture.

Hidden dependencies. There can be dependencies at the code level, e.g. generated by the use of reflection, that we did not analyse. In these cases the measurements would be flawed. However, our manual inspection of the code and the feedback of the developers made us be confident that we considered most common types of dependencies (as shown in Table 1). We also did not encounter the use of reflection based techniques by doing manual inspections of the systems. Furthermore, hidden dependencies would affect our results only in a single-side way: If we would have analysed more dependencies, we would have identified more potential documentation flaws and violations.

External validity. There are several particularities of the investigated projects that could reduce the transferability of our results to other environments.

The environment at Munich Re might influence the results. Subject to the case study have been three projects that have been carried out by different developers that are employed at different companies. Nevertheless, each of the project was done in the environment of Munich Re. All the projects used the same development process, a similar infrastructure and similar technologies (.NET). Thus, the external validity of the results may be limited. We plan to repeat the procedure on additional projects from different environments as future work.

6 Related work

Checking architecture conformance. In the reverse engineering literature, several approaches for checking architecture conformance with respect to high-level models have been proposed [4, 9]. [4] proposes an approach to check the compliance of OO designs with the source code by mapping designs expressed in OMT to C++ programs. Our technology for specifying architecture and checking its conformance with the code is similar to the reflexion models developed by Murphy [9]. Even if our technique for checking the conformity of the architecture documentation with the code is similar, our focus is different. In this paper we investigated the loss of architectural knowledge in the systems evolution and the usefulness of explicit high-level models to make this knowledge explicit.

[3] proposes an approach to use declarative queries to specify structural dependencies and check them against the implementation. These queries are continuously run during the development. Our experience confirms the need to integrate architectural checks in the development process and thereby to prevent the loss of architectural knowledge.

Related case studies. [14] investigates how companies identify design erosion and addresses the preservation of the design. The case study is based on a qualitative analysis by performing interviews with developers. Among the major causes for design erosion the authors identify the lack of knowledge of developers about the original design decisions and too little attention to design during evolution due to release pressure. Our observations support these conclusions. We advocate the preservation of architectural knowledge through continuous conformance checking. Our approach in this paper is both quantitative (measuring the discrepancies between the documentation and the code) and qualitative (developer interviews).

The case study in [12] illustrates the difficulty involved in detecting deviations of the code from the intended design that occur in the presence of personnel fluctuation. The authors propose the usage of metrics as principal means to detect the deviations. In this paper we emphasize on the need to continuously maintain and check the conformance of an explicit representation of the intended architecture.

In [8] the authors present experiences from using architectural models in an industrial project. They report on huge efforts on keeping the models in conformance with the implementation. Our results confirm the results obtained in this case study that ensuring the conformance manually is very hard in practice. Our approach of checking the conformance between the documentation (the model) and the implementation has the potential to reduce some of these efforts.

Architectural knowledge management. The authors of [1] draw a distinction between a *personalization strategy* and *codification strategy* for architectural knowledge management. On the one hand, the personalization strategy is mainly used in industry and emphasizes on the interaction among developers. On the other hand, the codification strategy is the basis for most of the research approaches in the area of knowledge management and concentrates on identifying and storing knowledge in artifacts and repositories [6]. In our work we advocate on the usefulness of the codification based approaches since they make the knowledge explicit. This is of capital importance especially in the presence of personnel fluctuations when a pure personalization strategy is impossible. Furthermore, whenever differences are found by the analysis, they are catalyzers for discussions among developers and entry points for a *personalization strategy*.

7 Conclusions

In this paper we present our experience in evaluating the loss of architectural knowledge in three industrial projects at Munich Re. Following our study we identified three manifestations of loss of architectural knowledge: decay of the code in form of violations of the intended architecture, loss of information in the documentation and different perceptions of the intended architecture by different developers. The central outcome of this case study is that we discovered that between 9% and 19% of all the dependencies implemented in the systems did not conform to the documented architecture. These differences could be identified as insufficiencies in the documentation as well as violations in the implemented architecture. The intended architecture was buried as implicit knowledge somewhere in between these artifacts as well as the knowledge of the developers and the architect. The main lesson learnt is that in order to minimize the knowledge loss, we need to make the knowledge about the intended architecture explicit and perform automatic architecture conformance analyses continuously in order to keep the awareness of developers about the architectural knowledge.

Acknowledgments. We want to thank Benjamin Hummel and Florian Deißenböck for their efforts spent on the development of ConQAT [2] and the implementation of the architecture analysis. Furthermore, we want to thank Rudolf Vaas from Munich Re and the developers in the projects for the great collaboration.

References

[1] M. A. Babar, R. C. de Boer, T. Dingsoyr, and R. Farenhorst. Architectural knowlege management strategies: Approaches in research and industry. In *SHARK-ADI*, page 2, Washington, DC, USA, 2007. IEEE CS.

[2] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[3] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE*, pages 391–400, New York, NY, USA, 2008. ACM.

[4] R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *ICSM '98*, page 94, Washington, DC, USA, 1998. IEEE CS.

[5] D. Garlan. Software architecture: a roadmap. In *ICSE*, pages 91–101, New York, NY, USA, 2000. ACM.

[6] M. T. Hansen, N. Nohria, and T. Tierney. What's your strategy for managing knowledge? *Harvard Business Review 77 (2)*, pages 106–16, 1999.

[7] E. Koutsofios and S. North. Drawing graphs with dot, 1993.

[8] A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald. Experiences from representing software architecture in a large industrial project using model driven development. In *SHARK-ADI*, Washington, DC, USA, 2007. IEEE CS.

[9] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.

[10] S. Ornburn and S. Rugaber. Reverse engineering: resolving conflicts between expected and actual software designs. *CSM'92*, pages 32–40, Nov 1992.

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[12] R. Tvedt, P. Costa, and M. Lindvall. Does the code match the design? a process for architecture evaluation. In *ICSM '02*, page 393, Washington, DC, USA, 2002. IEEE CS.

[13] J. van Gurp and J. Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.

[14] J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17(4):277–306, 2005.