Chapter 5
Practical Experiences

This chapter describes several practical experiences we have made over the last 10 years with different parts of the product quality control approach described in this book. The first three experience reports concentrate on building quality models and using them for quality requirements and evaluating quality: the Quamoco base model, the maintainability model for MAN Truck and Bus, the security model for Capgemini TS and an activity-based quality model for a telecommunications company. Next, we describe the application of quality prediction models, in particular reliability growth models, at Siemens COM. Finally, in the last experience report, we focus on applying analysis techniques: We apply static analysis, architecture conformance analysis and clone detection at SMEs.

5.1 Quamoco Base Model

The Quamoco base model is the largest and most comprehensive application of the Quamoco approach today. Therefore, we start with the experiences we have made in building it and applying it to several open source and industrial systems.

5.1.1 Context

Quamoco was a German research project sponsored by the German Federal Ministry of Education and Research[1] from 2009 to 2013. You can find all information on the project also on its website http://www.quamoco.de. The project was conducted by a consortium of industry and academia, all with prior experiences with quality models and quality evaluation. The industrial partners were Capgemini, itestra,

[1] Under grant number 01IS08023

S. Wagner, *Software Product Quality Control*, DOI 10.1007/978-3-642-38571-1_5,

SAP and Siemens. The academic partners were Fraunhofer IESE and Technische Universität München. JKU Linz participated as subcontractor for Siemens.

As this was a large project with many partners, there were different motivations and goals in joining the consortium. Therefore, we created various different contributions in the project. The main aim of all members of the consortium, however, was to build a generally applicable model that covers (a large share of) the main quality factors for software. Over time, we decided to call this the *base model*, because in parallel we were working on several specific quality models targeted at the domains of the industrial partners. For example, Siemens built a quality model for software in embedded systems [146].

What is also remarkable about Quamoco is the strong commitment to tool development. We built a quality model editor that supports in building such comprehensive quality models including all additional descriptions and quality evaluation specifications. It allows different views on the model to aid understanding, and it even has a one-click solution to perform the evaluation of a software system for the base model. This is connected to the integration with ConQAT (see Sect. 4.1) which executes all analysis tools, collects the results for the analyses and calculates the evaluations using the evaluation specifications from the model. Although the user-friendliness of the tool chain could be improved, the overall integration of model building and quality evaluation is probably unique.

5.1.2 Building the Model

The base model should be a widely applicable and general quality model. Therefore, we could not directly follow the model building approach of Sect. 3.1. The approach is more suitable for building specific quality models. Nevertheless, the main steps are still there but with rather broad results. The relevant stakeholders of the base model are all common stakeholders in software engineering: users, customers, operators, developers and testers.

Hence, the general goal is to build high-quality software. We referred to ISO/IEC 25010 (Sect. 2.3) as relevant document to elicit our quality goals which were the quality characteristics of the standard quality model. They formed our quality aspect hierarchy in the Quamoco base model. We chose these characteristics despite the problems we discussed, such as overlap, because we assume that using the standard will result in a broad acceptance.

We experimented with various artefacts created in software development but decided to concentrate on the main artefact developers work with and which has a strong effect on the user: the source code. Hence, the base model is source-code-focused. To be able to finish the model in the project duration, we decided, driven by the industry partners, to concentrate on Java and C# for modelling the measurements. With more time, however, we would have liked to include other artefacts to better reflect their influence on the quality aspects as well as other programming languages to make the base model more widely usable. With the

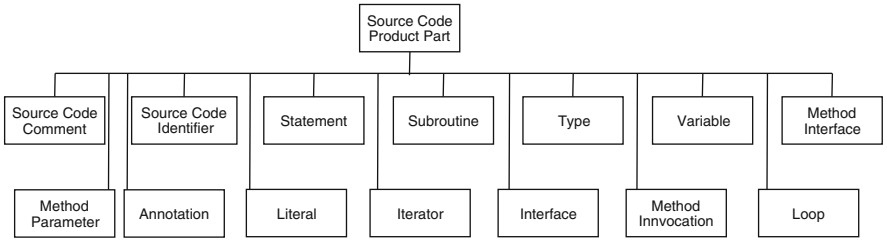

Fig. 5.1 An excerpt of the entity tree of the Quamoco base model

decision for source code, the resulting entities are almost all source code parts. An exception is, for example, the entity identifier which is also possible in other kinds of artefacts. We created a hierarchy for the source code parts which was partly built *bottom-up* while adding product factors and measures and partly structured *top-down* to keep an understandable entity tree. Figure 5.1 shows the layer of entities below the source code part which contains most entities. There would be some refactorings possible, for example, to organise the method-related entities in a part-of hierarchy, but we found that this does not improve readability.

To come up with appropriate product factors, we drew mainly from the descriptions of analysis rules of static analysis tools we employed, the experiences we have made in other contexts and our own expert opinion. This resulted in a wide range of product factors including [Class Name | CONFORMITY TO NAMING CONVENTIONS], [Duplication | SOURCE CODE PART] and [Class | SYNCHRONIZATION OVERHEAD].

The project had the length of 3 years, and we decided early on to work in three iterations. Hence, we rebuilt and fundamentally restructured the whole model two times. We worked on specific aspects in a large number of workshop sessions with a varying number of participants from most of the consortium's members. We had many and long heated discussions about identifying and naming product factors as well as the influences of them onto quality aspects. We believe these long discussion resulted in a highly readable and understandable model.

As the base model is a general model, we could not formulate concrete quality requirements but had to model what is "normal" and generally applicable. Therefore, we searched for ways to accomplish that. For the weights of the quality aspects, which usually are specific for a product, we chose to use the results of a survey we have also done in the context of Quamoco [213]. It represents the average opinion of over hundred experts. Therefore, we modelled the weights corresponding to the importance of the quality aspects given by the survey respondents.

Finally, to determine the evaluation specifications for the product factors, we used a calibration approach based on open source systems. The assumption was that by using a large number of existing systems, we will cover many different contexts and, on average, get a widely applicable evaluation. We chose for the C# part 23 systems and selected over hundred systems with varying sizes from an open source collection of Java systems. We needed to determine the *min* and *max* for the evaluation specifications (Sect. 4.2) which we chose as the 25 % and 75 % percentiles of all the results we got from measuring the open source systems.

5.1.3 Quality Evaluation and Measurement

It was important for the Quamoco consortium not only to build abstract quality aspects and some corresponding product factors but to have it completely operationalised. To make this possible, we had to concentrate on two programming languages: Java and C#. For these two languages, there are measures for all product factors and corresponding evaluation specifications up to the quality aspects.

The process of measurement and quality evaluation is well supported by the Quamoco tool chain consisting of the quality model editor and an integration with the dashboard tool ConQAT. We can add evaluation specifications to any product factor and quality aspect and define how it is evaluated depending on measurement results or influencing product factors. For that, we can define a maximum number of *points* a product factor can get, distribute the points among the related measures to express weights and finally define linear increasing or linear decreasing functions with two thresholds. The actual number of points, which is then normalised to a value between 0 and 1, is determined by running the evaluation specifications in ConQAT. The model editor allows to export a corresponding specification for it.

Let us look at the example of cloning (Sect. 4.4.2) and how it is represented in the base model. We chose to call the corresponding product factor [Source Code Part | DUPLICATION]. It is not directly language-specific but could be valid for any kind of system or programming language. The model defines it as "Source code is duplicated if it is syntactically or semantically identical with the source code already existing elsewhere". We cannot measure the semantic identity directly. It might be possible to find semantic clones in reviews, but it is not very reliable. Instead, we focus on the syntactic redundancy.

Figure 5.2 shows the screen from the quality model editor in which we specify the evaluation of the product factor. We have two measures available in the model: clone coverage and cloning overhead. As their meaning is similar, we chose only to include clone coverage to measure duplication. Therefore, the maximum defined number of points (here: 100) goes to clone coverage (CP in the table). Clone coverage was the probability to find a duplicate for any randomly chosen statement. Hence, the larger the clone coverage, the higher the degree of duplication. Therefore, we chose a linear increasing function.

This leaves us with deciding for the *min* and *max* thresholds. In this case, below the *min* threshold, the product factor receives 0 points, and above the *max* threshold, it receives 100 points. Based on the calibration for Java, we found that most systems have a clone coverage between 0.0405 and 0.5338. This was consistent also with our observations. Hence, we used them as thresholds for the linear increasing function.

Let us look at a concrete example of applying the evaluation to a real system. The Java open source library *log4j*[2] is often used to implement logging. We analysed version 1.2.15 using our base model. ConQAT gave us a clone coverage result

[2] http://logging.apache.org

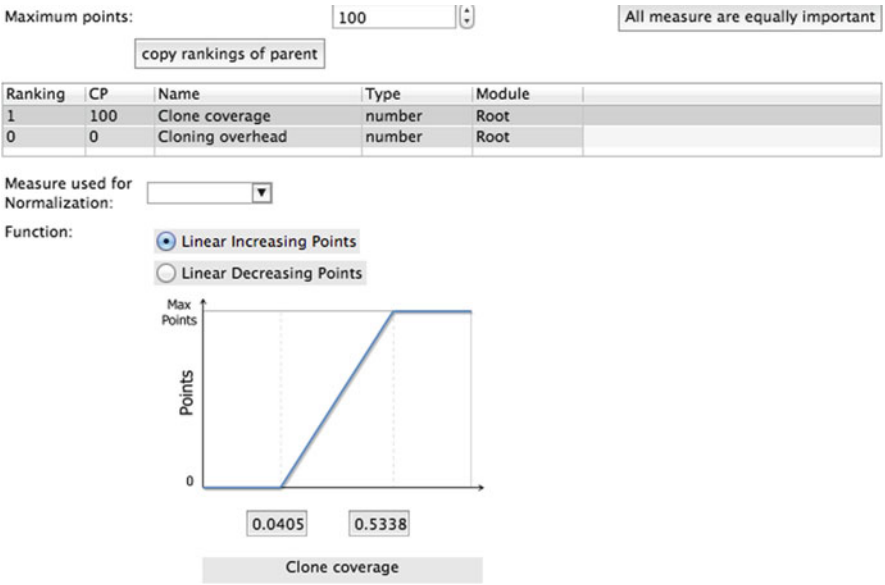

Fig. 5.2 The evaluation specification for [Source Code Part | DUPLICATION] in the quality model editor

of 0.086 which is comparably small but above the threshold. Putting this clone coverage in the linear increasing evaluation function resulted in 9.2 points or, normalised between 0 and 1, an evaluation of the product factor of 0.092. This is very low and has a positive impact on analysability and modifiability.

Analysability, for example, is influenced by more than 80 product factors. They are weighted based on an expert-based ranking which results in a weight of only 1.06 of 100 for our product factor. For the example of log4j, although the result for the duplication was very good, the other product factors brought it down to an evaluation of 0.93 which we normalise and transform into a school grade of 4.3 (where 1 is best and 6 is worst).

As we use ConQAT for evaluating based on the base model, we directly get a dashboard created. Figure 5.3 shows an example output of the dashboard. It recreates the hierarchies of quality aspects and product factors and allows to drill down to each individual measure. In addition, it shows configuration and execution time information of the evaluation.

5.1.4 Applications and Effects

Inside the Quamoco project, we applied the Quamoco quality evaluation based on the base model to several open source and industrial systems to investigate the

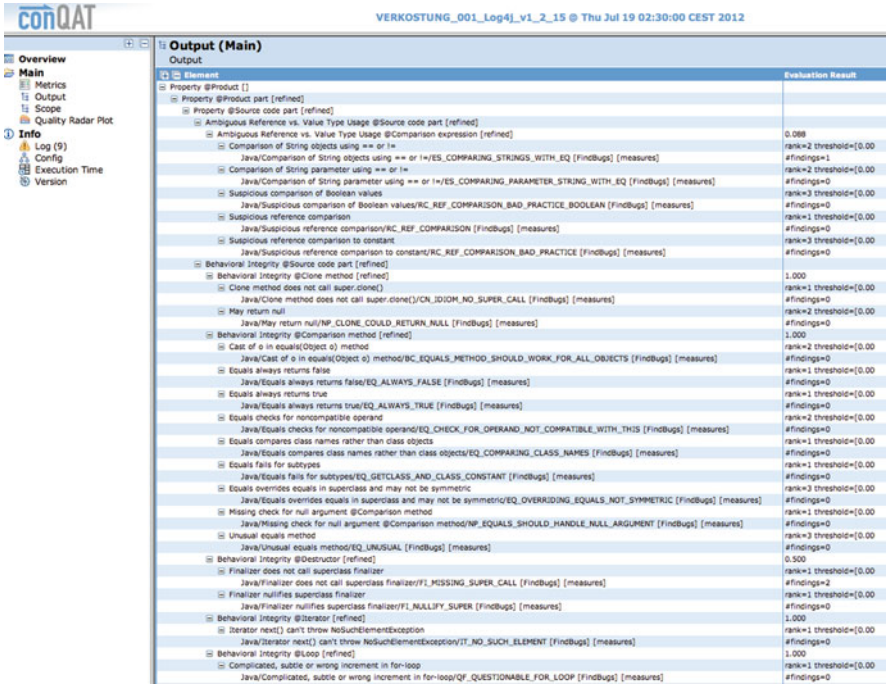

Fig. 5.3 Dashboard for the quality evaluation

Table 5.1 Comparison of the evaluations using the base model and the expert evaluations for five open source products [211]

Product	LOC	Quamoco grade	Quamoco rank	Expert rank
Checkstyle	57,213	1	1	1
log4j	30,676	3	2	2
RSSOwl	82,258	3	2	3
TV-Browser	125,874	4	4	4
JabRef	96,749	5	5	5

validity of the evaluations as well as how well practitioners understand the quality model and the evaluation results.

To be able to investigate the validity of the quality evaluation, we need software systems with an existing, independent evaluation. Our first application was to the five open source Java products *Checkstyle*, *RSSOwl*, *log4j*, *TV-Browser* and *JabRef*. For these products, there have been an event where experts evaluated their quality [79]. We then applied our Java base model to the same versions and compared the resulting quality ranking from our Quamoco evaluation and the evaluation from experts. You can find the comparison in Table 5.1. The Quamoco evaluation produced almost exactly the same ranking as the experts did. We were only not able to distinguish between RSSOwl and log4j.

Table 5.2 Comparison of the evaluations using the base model and expert evaluations for five subsystems of an industrial product [211]

Subsystem	Quamoco rank Quality	Expert rank Quality	Quamoco rank Maintainability	Expert rank Maintainability
Subsystem D	1	1	1	1
Subsystem C	2	4	3	4
Subsystem B	3	2	5	3
Subsystem E	4	4	4	3
Subsystem A	5	2	2	2

Table 5.3 Quamoco evaluation results for consecutive versions with explicit quality improvements [211]

Version	Quamoco grade
2.0.1	3.63
2.0.2	3.42
2.1.0	3.27
2.2.1	3.17

Next, we looked at an industrial software and an expert ranking for its five subsystems. We used that again to compare it to the corresponding Quamoco evaluation results. You can see the comparison for quality overall as well as maintainability, which is the strongest part of the base model, in Table 5.2. The agreement in the rankings is not as clear as above, but the general trends are similar. There is an agreement in the best ranked subsystem D. Subsystem A, however, came out last in overall quality, while the expert saw it much better. One explanation could be that the expert put more emphasis on the maintainability of the product in which the Quamoco evaluation also judged subsystem A as better. Overall, this application also supported our choice of contents and calibration of the base model.

A third application of the base model was to another industrial system that underwent quality improvements specifically in some releases. We were interested in the question whether the Quamoco evaluation results would be able to show a quality improvement. Table 5.3 shows the small but gradual improvement as evaluated by the Quamoco base model. Hence, at least for this case, the base model is well enough calibrated to support project managers in quantitatively controlling quality improvements.

Our final application was to eight industrial systems in which we evaluated the systems and afterwards interviewed experts on the systems. You can find details on the questionnaire and results in [210], but we will only highlight some of the results. The experts judged our base model to be a more transparent definition of quality than only ISO/IEC 25010, and it also could be more easily adopted and applied in practice. They also found the relationships in the quality model acceptable and the calculations in the evaluation transparent. It was seen as problematic to understand the rankings we used for determining weights in the evaluation specifications and the

calibration thresholds. It was seen positively that the evaluation used school grades and that it clarifies metrics as well as the abstract quality characteristics of ISO/IEC 25010. Overall, we received very positive feedback which supports our confidence that the Quamoco base model is ready to be applied in practice.

5.1.5 Lessons Learned

There were several problems in building and applying the Quamoco base model. First, we learned that building a fully operationalised quality model means that we had to build a really large model. As any other artefact, the quality model then becomes hard to maintain, for example, when we wanted to add new tools and measures which then had the effect that we needed to restructure the product factors. With the help of the quality model editor and some practice, this was manageable, but it should not be underestimated.

Second, it was hard to build such a detailed model in consensus with a group of different people from different domains. We spent many hours discussing various aspects of the meta-model, the structuring of the quality model, the impacts and even names of entities, product factors and measures. In the end, however, I believe that this has led to the good feedback we received from practitioners that they were able to understand the chosen relationships and that it helped in clarifying ISO/IEC 25010. It seems like it is a necessary process.

Third, the large amount of work we spent in building the operationalised quality model also payed back in the sense that we actually have now an almost fully automatic way to evaluate software quality for Java and C#. The model is not well equipped for quality factors that have a strong dynamic side, like performance efficiency or reliability, but even for those, we could reach reasonable results. In addition, we have a set of manual measures that need to be collected in reviews. It seems these manual measures are important for a good evaluation, but this is subject to further research.

5.2 MAN Truck and Bus

MAN Truck and Bus was one of the first applications of the activity-based quality model approach. Because of continuing development, the model we built there had been the most refined quality models before the start of Quamoco. The model concentrates on describing maintainability with a focus on embedded, automotive systems modelled in Matlab Simulink and Stateflow that have the aim to generate code directly from the functional models. The original description of this model can be found in [51].

5.2.1 Context

MAN Truck and Bus is a German-based international supplier of commercial vehicles and transport systems, mainly trucks and busses. It has over 30,000 employees worldwide of which 150 work on electronics and software development.

The organisation brought its development process to a high level of maturity by investing enough effort to redesign it according to best practices and safety-critical system standards. The driving force behind this redesign was constantly focusing on how each activity contributes to global reliability and effectiveness. Most parts of the process are supported by an integrated data backbone developed on the eASEE framework from Vector Consulting GmbH. On top of this backbone, they have established a complete model-based development approach using the tool chain of Matlab/Simulink and Stateflow as modelling and simulation environment and TargetLink of dSpace as C-code generator.

Matlab/Simulink is a model-based development suite aiming at the embedded systems domain. It is commonly used in the automotive area. The original *Simulink* has its focus on continuous control engineering. Its counterpart *Stateflow* is a dialect of statecharts that is used to model the event-driven parts of a system. The Simulink environment already allows to simulate the model to validate it.

In conjunction with code generators such as Embedded Coder from MathWorks or TargetLink by dSpace, it enables the complete and automatic transformation of models to executable code. This is a slightly different flavour of model-based development than the MDA approach proposed by the OMG.[3] There is no explicit need to have different types of models on different levels, and the modelling language is not UML. Nevertheless, many characteristics are similar and quality-related results could easily be transferred to an MDA setting.

5.2.2 Building the Model

Here we follow the approach for model building introduced in Sect. 3.1.

Define General Goals

MAN Truck and Bus, especially the stakeholder *management*, has the general goal of a global development with high reliability and effectiveness. The other relevant stakeholders in this case are the *developers* and *testers*. Both are interested in an efficient and effective development and test of their systems.

[3] http://www.omg.org/mda/

Analyse Documents Relevant for General Goals

To further understand and refine these general goals, we analysed the existing development process definition, guidelines and checklists, as well as slide decks describing the strategy. In addition to the document analysis, the major source of input was in-depth interviews with several of the stakeholders. The major refinement of the goals was the commitment to a completely model-based development with Simulink/TargetLink and the central management of all development-related artefacts and data in the data backbone eASEE. In addition, the interviews showed that the main interest at present was the analysis and assurance of the maintainability of the built models.

Define Activities and Tasks

As we refined the general goals to the maintainability of the models, we consider *maintenance* the top level activity we are interested in. For that, we use a standard decomposition of maintenance activities from IEEE Standard 1219 [88]. Furthermore, we extended the activity tree to match the MAN development process by adding two activities (*Model Reading* and *Code Generation*) that are specific for the model-based development approach.

Define Quality Goals

As we already narrowed the general goals to the analysis and assurance of maintainability, our main quality goal is to support the maintenance activity. The most important activity is code generation, because if we cannot generate meaningful source code, the model is useless. Then we are highly interested to be able to quickly read and understand the models. Finally, also the easy test of the models and the resulting code is of high importance.

Identify Affected Technologies and Artefacts

From the general and quality goals, we derive that the main artefacts that are affected are the Simulink/TargetLink models of the automotive functions. In addition, as we are interested also in testing the system, the resulting source code could be important as well. Furthermore, as MAN uses the data backbone, the additional data in the backbone can influence our quality goals.

Analyse Relevant Materials

The material we analysed for building the quality model consists of three types:

1. Existing guidelines for Simulink/Stateflow
2. Scientific studies about model-based development
3. Expert know-how of MAN's engineers

Specifically, our focus lies on the consolidation of four guidelines available for using Simulink and Stateflow in the development of embedded systems: the MathWorks documentation [144], the MAN-internal guideline, the guideline provided by dSpace [55], the developers of the TargetLink code-generator and the guidelines published by the MathWorks Automotive Advisory Board (MAAB) [145]. There is now also a MISRA guideline for Simulink [152] and TargetLink [153] which were not available when we built the model.

Define Product Factors

Because of confidentiality reasons, we are not able to fully describe the MAN-specific model here. However, we present a number of examples that illustrate parts of the model. Overall, we modelled 87 product factors (64 new entities) that describe properties of entities not found in classical code-based development. Examples are states, signals, ports and entities that describe the graphical representation of models, e.g. colours.

We started with a simple translation of the existing MAN guidelines for Stateflow models into the maintainability model. For example, the MAN guideline requires the current state of a Stateflow chart to be available as a measurable output. This simplifies testing of the model and improves the debugging process. In terms of the model, this is expressed as [Stateflow Chart | ACCESSIBILITY] $\xrightarrow{+}$ [Debugging] and [Stateflow Chart | ACCESSIBILITY] $\xrightarrow{+}$ [Test].

We describe the ability to determine the current state with the property ACCESSIBILITY of the entity Stateflow Chart. The Stateflow chart contains all information about the actual statechart model. Note that we carefully distinguish between the *chart* and the *diagram* that describes the graphical representation. In the model the facts and impacts have additional fields that describe the relationship in more detail. This descriptions are included in generated guideline documents.

Specify Quality Requirements

Finally, we classified most of the product properties in properties that *must not* or *should not* hold. This classification is sufficient for generating review guidelines and using the results for a manual maintainability analysis.

5.2.3 Effects

Consolidation of the Terminology

At MAN, we found that building a comprehensive quality model has the beneficial side effect of creating a consistent terminology. By consolidating the various sources of guidelines, we discovered a very inconsistent terminology that hampers a quick understanding of the guidelines. Moreover, we found that even at MAN the terminology has not been completely fixed. Fortunately, building a quality model automatically forces the modeller to give all entities explicit and consistent names. The entities of the facts tree of our maintainability model automatically define a consistent terminology and thereby provide a glossary.

One of many examples is the term *subsystem* that is used in the Simulink documentation to describe Simulink's central means of decomposition. The dSpace guideline, however, uses the same term to refer to a *TargetLink subsystem* that is similar to a Simulink subsystem but has a number of additional constraints and properties defined by the C-code generator. MAN engineers, on the other hand, usually refer to a *TargetLink subsystem* as *TargetLink function* or simply *function*. While building the maintainability model, this discrepancy was made explicit and could be resolved.

Resolution of Inconsistencies

Furthermore, we are able to identify inconsistencies not only in the terminology but also in contents. For the entity *Implicit Event*, we found completely contradictory statements in the MathWorks documentation and the dSpace guidelines.

- *MathWorks [144]* "Implicit event broadcasts [...] and implicit conditions [...] make the diagram easy to read and the generated code more efficient".
- *dSpace [55]* "The usage of implicit events is therefore intransparent concerning potential side effects of variable assignments or the entering/exiting of states".

Hence, MathWorks sees implicit events as improving the readability, while dSpace calls them intransparent. This is a clear inconsistency. After discussing with the MAN engineers, we adopted the dSpace view.

Revelation of Omissions

An important feature of the quality meta-model is that it supports inheritance. This became obvious in the case study after modelling the MAN guidelines for Simulink variables and Stateflow variables. We model them with the common parent entity Variable that has the attribute LOCALITY that expresses that variables must have the smallest possible scope. As this attribute is inherited by both types of variables,

we found that this important property is not expressed in the original guideline. Moreover, we see by modelling that there was an imbalance between the Simulink and Stateflow variables. Most of the guidelines related only to Simulink variables. Hence, we transferred them to Stateflow as well.

Integration of Empirical Research Results

Finally, we give an example of how a scientific result can be incorporated into the model to make use of new empirical research. The use of Simulink and Stateflow has not been intensively investigated in terms of maintainability. However, especially the close relationship between Stateflow and the UML statecharts allows to reuse results. A study on hierarchical states in UML statecharts [44] showed that the use of hierarchies improves the efficiency of understanding the model in case the reader has a certain amount of experience. This is expressed in the model as follows: [Stateflow Diagram | STRUCTUREDNESS] $\xrightarrow{+}$ [Model reading].

5.2.4 Usage of the Model

At MAN, we concentrated on checklist generation and preliminary automatic analyses. We chose them, because they promised the highest immediate pay-off. The Quamoco tool chain did not yet exist.

Checklist Generation

We see quality models as central knowledge bases for quality issues in a project, company or domain. This knowledge can and must be used to guide development activities as well as reviews. The model in its totality, however, is too complex to be comprehended entirely. Hence, it cannot be used as a quick reference. Therefore, we exploit the tool support for the quality model to select subsets of the model and generate concise guidelines and checklists for specific purposes.

The MAN engineers perceived the automatic generation of guideline documents to be highly valuable as we could structure the documents to be read conveniently by novices as well as experts. Therefore, the documents feature a very compact checklist-style section with essential information only. This representation is favoured by experts who want to ensure that they comply to the guideline but do not need any further explanation. For novices the remainder of the document contains a hyperlinked section providing additional detail. Automatic generation enables us to conveniently change the structure of all generated documents. More importantly, it ensures consistency within the document which would be defect prone in handwritten documents.

Automatic Analyses

As the model is aimed at breaking down product factors to a level where they can be evaluated and they are annotated with the degree of possible automation, it is straightforward to implement automatic analyses. For the product properties that can be automatically evaluated, we were able to show that we can check them in Simulink and Stateflow models.

For this, we wrote a parser for the proprietary text format used by Matlab to store the models. Using this parser we are able to determine basic size and complexity metrics of model elements like states or blocks, for example. Moreover, we can use the parser to automatically identify model elements that are not satisfactorily supported by the C-code generator. We also implemented clone detection specifically for Simulink and Stateflow models and included it into the quality assessment from the model.

By integrating these analyses in our quality controlling toolkit ConQAT [50], we are able to create aggregated quality profiles and visualisations of quality data. We have not yet used the integrated quality evaluation approach described in Sect. 4.2. This would be the next logical step.

5.2.5 Lessons Learned

Overall, the MAN engineers found the approach of building the model for maintainability useful. Especially the model's explicit illustration of impacts on activities was seen as beneficial as it provides a sound justification for the quality rules expressed by the model. Moreover, the general method of modelling – that inherently includes structuring – improved the guidelines: Although the initial MAN guideline included many important aspects, we still were able to reveal several omissions and inconsistencies. Building the model, similar to other model building activities in software engineering [180], revealed these problems and allowed to solve them.

Another important result is that the maintainability model contains a consolidated terminology. By combining several available guidelines, we could incorporate the quality knowledge contained in them and form a single terminology. We found terms used consistently as well as inconsistent terminology. This terminology and combined knowledge base were conceived useful by the MAN engineers.

Although the theoretical idea of using an explicit quality meta-model for centrally defining quality requirements is interesting for MAN, the main interest is in the practical use of the model. For this, the generation of purpose-specific guidelines was convincing. We not only built a model to structure the quality knowledge, but we are able to communicate that knowledge in a concise way to developers, reviewers and testers. Finally, the improved efficiency gained by automating specific assessments was seen as important. The basis and justification for these checks are given by the model.

Table 5.4 Sample projects [138]

Project	Description
A	Public sector, desktop application
B	Private partner, desktop application
C	Private partner, web application
D	Private partner, software component
E	Private partner, web application
F	Public sector, software component

5.3 Capgemini TS

Based on the experiences with modelling maintainability at MAN Truck and Bus, we stepped onto new terrain with modelling security at Capgemini TS [138] who has its main focus on the individual development of business information systems.

5.3.1 Context

Capgemini TS is the technology service entity of Capgemini. Their main focus is on custom business information systems and therefore the systems are very different from what we encountered at MAN Truck and Bus.

As at Capgemini TS there are numerous projects, we cannot immediately build a quality model valid for all those projects. We therefore decided to restrict ourselves to a small initial sample for the model building. We interviewed contact persons from 20 projects. From this analysis, we found six projects (A–F) that were suitable for building the quality model since they were able to offer sufficient data. The sizes of the selected projects range from one person month to 333 person years; the average size was 163 person years (Table 5.4).

5.3.2 Building the Model

Define General Goals

Similarly to the huge number of projects and their diversity at Capgemini TS, there are various general goals by the existing stakeholders. In this particular case, we restricted ourselves to the business goal to ensure customer satisfaction by protecting their valuable assets. The business information systems of Capgemini TS often include large amounts of partly very sensitive information of their customers as well as perform vital services which are often exposed to the Internet. The protection of this data and services is essential to keep the customer satisfied.

Analyse Documents Relevant for General Goals

The aim of this step is to refine the general goals to derive relevant activities and tasks. The protection of valuable assets is what we usually describe by *security*. Therefore, the main document we use as basis for the quality model is the security model from Sect. 2.6.2. It defines as activity hierarchy a classification of attacks on software systems which we can reuse for the Capgemini quality model. In addition, we compare the attacks from the security model with the existing requirements specifications of the analysed projects to remove irrelevant attacks or detect omissions in the activities. Capgemini TS especially uses the German *IT-Grundschutz* manual [29] of the BSI as a basis for security requirements.

Define Activities and Tasks

The basic activity hierarchy is the same as in the model in Sect. 2.6.2. In particular, we extended it by the activities that we found in the *IT-Grundschutz* manual. It defines threats, such as the abuse of user rights, which we incorporated into the activity hierarchy.

Define Quality Goals

We restricted ourselves in the Capgemini quality model early on to security-related activities. We have the quality goal that all attack activities in our activity hierarchy that are relevant for a particular software system are hard to perform. For example, the abuse of user rights needs to be difficult.

Identify Affected Technologies and Artefacts

We then analysed what technologies and artefacts we need to describe by product properties to reach these goals. In the context of Capgemini TS, we needed to look at the programming and markup languages Java, JSP, ASP.NET, JavaScript, HTML and CSS. All artefacts created in these languages can be subject to attacks. Furthermore, the user and rights management is a key artefact in a software system for ensuring its robustness against attacks. Finally, although it might not be directly a part of the analysed software system, there is a network and operating system that influence attacks. Finally, there can be additional security systems such as firewalls.

Analyse Relevant Material

To find relevant product factors for the technologies and artefacts, we go back to the material we have already analysed for activities. The security model from

Sect. 2.6.2 contains product factors and impacts to attacks from existing collections. The *IT-Grundschutz* manual describes *safeguards*, which are product properties to prevent threats: our attack activities. Finally, Capgemini TS has many experiences with building security-critical software systems. This expert knowledge from practical experience is also a valuable source for product factors. We, therefore, also analyse the existing requirements specifications of our sample projects.

Define Product Factors

Corresponding to the threats, which we modelled as activities, the manual describes so-called *safeguards*, which are counter measures to these threats. We modelled the safeguards as product factors.

Overall, we defined several hundred product factors for the identified technologies and artefacts. It starts with general product factors that hamper attacks. For example, that passwords should never be saved or shown as clear text:
[Password | CONCEALMENT] $\overline{\longrightarrow}$ [Attack]

Then we described product factors for specific technologies. For example, in most Capgemini TS systems, there are database systems. We defined that instead of dynamically built SQL, a developer should use:

- An OR mapper
- Prepared statements
- Static SQL statements

[Database Access | APPROPRIATENESS] $\overline{\longrightarrow}$ [SQL injection]

Another example is the session handling in web applications. We defined that URLs do not contain session IDs and that each session has a time-out. Both have a negative impact on hijacking the sessions:
[URL | APPROPRIATENESS] $\overline{\longrightarrow}$ [Session Hi-Jacking]
[Session Length | LIMITEDNESS] $\overline{\longrightarrow}$ [Session Hi-Jacking]

As final example, we defined product factors that relate to the network connections of the software system. For valuable assets, such as sensitive user data, it works against many attacks to encrypt the transfer of these assets over networks:
[Asset Transfer | GUARDEDNESS] $\overline{\longrightarrow}$ [Attack]

Specify Quality Requirements

As last step, some of the product factors need to be refined to concrete quality requirements. For example, [Session Length | LIMITEDNESS] $\overline{\longrightarrow}$ [Session Hi-Jacking] is not complete as it only prescribes that there needs to be some limit of the session length. In a quality requirement for a concrete system, we gave a session limit of 5 min. Other product factors, such as the concealment of passwords, could be directly used as quality requirements.

Table 5.5 Reuse potential [138]

Project	# Sec. Reqs.	Reuse ratio
A	127	–
B	23	0.87
C	48	0.27
D	5	0.60
E	29	0.52
F	408	0.18
Mean	106.67	0.47

5.3.3 Effects

Comprehensive and Concise Specification

We could improve the specification documents since the structure of the model prescribes a uniform way to document each requirement with two major effects:

1. Avoidance of redundancy
2. Explicit rationales

The clear structure prevents redundant or similar parts in general.

Reuse

Another important effect is that the quality model can act as a repository for security or other quality requirements. This central repository ensures requirements specifications with a high quality that can be reused and thereby reduce the possibility of defects in the requirements. We deliberately built the quality model from analysing one project after another. This way, we could analyse the overlap between the security requirements of the projects. This overlap is the potential reuse that can be realised by exploiting the quality model. Table 5.5 shows the results for our sample projects. The column *# Sec. Reqs* gives the number of security requirements in the quality model, and *Reuse Ratio* shows the share that could have been reused. The range of the size of the requirements is high which stems from the diversity of the analysed systems and their specifications. On average more than 100 security requirements were specified per project.

We found that on average 47 % of the security requirements could have been reused using the quality model. The range goes from 0.18 up to 0.87. The ratio, however, depends strongly on the size of the specification. In smaller specifications, it is by far easier to come to a high reuse ratio. With an average reuse ratio of 47 %, almost every second specified requirement could have been reused employing the quality model. Still, as we have no baseline of reuse that would be possible without the model approach, we cannot give an improvement caused by the approach but only show its potential. Among other factors, the size of the specification

has a large effect on the reuse ratio. The model repository can only deliver as many requirements as are contained in it. Hence, the reuse potential for a large specification seems to be lower. Another effect, especially found in project F, is that larger specifications tend to have more redundancies. Because of the fixed structure of requirements in the model, this redundancies can largely be reduced.

Beyond these findings, we found further aspects to be discussed. First, in several cases we could reveal omissions in the specified security requirements. We derived additional requirements that were not contained in the original requirements specifications. Finally, we observed during the case study that the efforts of applying the model approach decreased while including the requirements of more and more projects into the repository. A major cause was easy access to quality requirements via goals. On the long run, such a central quality model can contribute to an efficient quality knowledge transfer to companies.

5.3.4 Usage of the Model

Capgemini TS uses the quality model very differently from MAN. While MAN focuses on the evaluation of their Simulink/TargetLink models using the quality model, Capgemini aimed at using the quality model to elicit and specify quality requirements. Both activities are closely related but can have positive effects independently.

5.3.5 Lessons Learned

Overall, also the engineers at Capgemini TS found the quality model approach useful to structure and reuse their quality requirements. Even though the quality evaluation possibilities are not exploited, the quality model is already useful as a means for structuring quality requirements. It ensures

1. the completeness of the quality requirements by employing existing experiences and
2. the existence of rationales for each requirements in the form of the activities, or attacks, it influences.

5.4 Telecommunications Company

A further practical experience, I want to report on, is about introducing a quality model and especially a dashboard at a company. This was part of an investigation of introducing quality control at a telecommunications company [67].

5.4.1 Context

The company we worked with in this case was a German telecommunications company which provides Internet, phone and TV services to end users. We had a small collaboration on improving and introducing quality management and control techniques in their software development process. Their software systems mainly consist of internal billing systems as well as web applications for the customers.

The idea of the collaboration was, among others, to understand the current state of software development in that company, find problems related to quality control, derive a suitable quality model and implement it using corresponding quality assurance techniques and measures. This collaboration took place before the Quamoco project and, hence, we were not able to use the fully fleshed-out Quamoco approach and tool chain. Instead, we concentrated on building an activity-based quality model to understand what data should be collected and implementing corresponding analyses. The results were supposed to be applicable in principle to all software systems of the company, but we concentrated on a specific larger system as an example.

5.4.2 Building the Model

As this collaboration had been before the Quamoco approach and the actual scope was a bit bigger, we did not follow the model building approach from Sect. 3.1 directly. Instead, we started with an interview to investigate the state of the software development at the company and learn about current quality problems. We interviewed 12 people from development, project management, requirements engineering, quality assurance and operation.

We found that the company successfully works with a mostly waterfall-like process with specification, design, implementation and testing sequentially. In this waterfall, there is an established procedure for quality gates after each phase. Only risk management is missing from the process description. In the testing phase, they start with unit tests which are done by the developers. Later integration and system tests are done by the quality assurance team. During development, they already used code reviews and coding guidelines as well as, in some projects, test-driven development. Especially the use of automated unit tests varied strongly over the different projects. If automated, tests are highly used for regression testing which is seen as very beneficial because problems are detected early. The test progress is controlled by the number of succeeded and failed test cases as well as by tracking assigned, resolved and closed change requests.

We discussed and proposed several measures and asked for the interviewees' opinions. The measures considered very important were test coverage, degree of dependency between modules, structuredness of the code, loading time of the program, code clones and execution time. Only of medium importance were

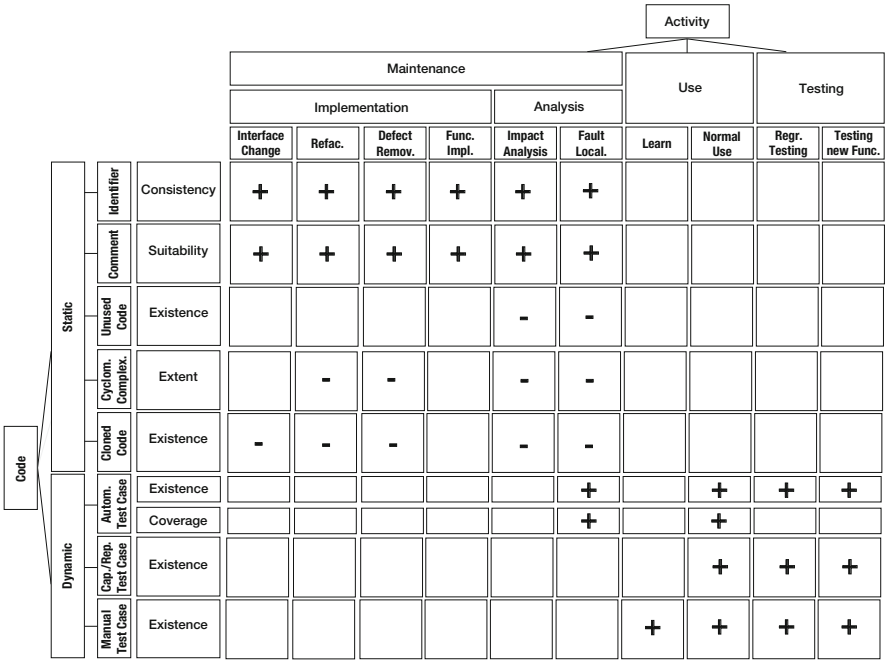

Fig. 5.4 The derived activity-based quality model for the telecommunication company

considered the comment ratio (code comments to code), dead code (because it is not important at run time) and number of classes or interfaces. Also size measures in general were not considered important at all apart from giving a general impression of the size of the system. Finally, we also let them classify quality factors to derive a corresponding quality model. They judged functional suitability, security, reliability, usability and modifiability as most important.

We used the information from the interviews to derive an activity-based quality model and corresponding measures to evaluate the product factors in the model. As shown in Fig. 5.4, we modelled if a product factor (left-hand side) has a positive (+) or negative (−) impact on the activities (top). We did not implement a complete evaluation aggregation along these pluses and minuses. Instead, we defined measures for the product factors and informally assumed that good measurement values for the product factors mean a good or bad impact on the activities.

As you can also see in Fig. 5.4, we concentrated on code as the root entity and structured it into dynamic aspects of the code and static aspects. In the static aspects, we included the entities Cloned Code, Cyclomatic Complexity, Unused Code, Comment and Identifier. For the dynamic aspects, we included Manual Test Case, Capture/Replay Test Case and Automated Test Case. In the Quamoco approach, we would probably model the entities differently, for example, using Method and COMPLEXITY as its property to form a product factor. This quality model shows, however, that it is not

necessarily important to follow the modelling principles of Quamoco closely and still derive benefit from building an explicit quality model.

As the entities were already rather specific, most of their properties are only EXISTENCE, but we also used CONSISTENCY, SUITABILITY, EXTENT and COVERAGE. Following from the important quality characteristics, we included the activities Maintenance, Use and Testing in the quality model. Each of these activities were broken down in several sub-activities to make the impacts from product factors more clear. For example, the [Cyclomatic Complexity | EXTENT] has a negative influence on Refactoring and Defect Removal but not the Functionality Implementation of new features.

Finally, we had to define measures for each of the product factors together with thresholds for what is considered good or bad. In the Quamoco approach, we would map this to a linear increasing or decreasing function to fully operationalise the quality evaluation. Here, we just implemented the analysis in ConQAT and showed red, yellow or green traffic lights for each measure. For example, for the comment ratio, we considered a ratio between 0 % and 15 % as well as between 85 % and 100 % as red, between 15 % and 30 % as well as 70 % and 85 % as yellow and between 30 % and 70 % as green. Similarly, we defined thresholds for measures such as clone coverage, architecture conformance, cyclomatic complexity and test coverage.

5.4.3 Effects

After the introduction of the measures and analyses, we conducted a survey among the previously interviewed engineers and beyond to judge the benefit generated. They saw very high benefit in better architecture specifications, triggered by the architecture conformance analysis, and additional dynamic tests, triggered by the monitoring of tests. Similarly, the engineers judged the consequent definition, conformance and updating of architecture specifications, test-driven development and overall the new static analysis as highly beneficial. They realised high benefits from analysing code clones, unit test results and test coverage. Only medium benefit brought the analysis of unused code and cyclomatic complexity which seemed to not add so much more additional information. Overall, most of the engineers answered that they check the analyses in the ConQAT dashboard every day; only some check them only every week.

5.4.4 Lessons Learned

We found that the developers were overall happy about the additional information about the quality of their systems that we derived using the activity-based quality model. Especially architecture specification and analysis was underdeveloped and, therefore, a welcome addition. The explicit modelling of relationships in the

activity-based quality model helped to make the quality goals transparent. Today, we would use the Quamoco evaluation approach and toolset to support a complete operationalisation of the model. Nevertheless, even this rather pragmatic model helped in structuring the quality evaluation.

5.5 Siemens COM

We worked with the network communication division of Siemens on a very specific quality model and quality evaluation. The aim was to analyse and predict failures in the first year in the field. The quality control loop concentrated on the last phases, especially system test and field test. It shows the use of testing as data source for quality evaluation. We reported these experiences initially in [207].

5.5.1 Context

Siemens Enterprise Communications is now a joint venture of the The Gores Group and Siemens AG, but it was still part of Siemens AG at the time of our collaboration. The company has a strong history in voice communication systems and expanded into other areas of communication and collaboration solutions. They have offices worldwide and employ about 14,000 people.

We worked with the quality assurance department which is responsible for system and field testing. The aim was to analyse and optimise system tests and field tests by predicting the number of field failures in the first year of a new product release.

5.5.2 Quality Model

As we had a very concrete and focused goal, we did not build a broad Quamoco quality model to capture all potentially interesting product factors, but we identified the essential activities and properties. The overall goal was to minimise the disruptions of the users of the Siemens systems by software failures. Hence, in an activity-based quality model, our quality goal was to maximise the effectiveness of the interaction of the users with the system. In a product quality view from ISO/IEC 25010, we wanted to maximise reliability.

Similar to the experiences at MAN and Capgemini, we could have built a quality model that describes all product factors that influence the effectiveness of the interaction of the user, use that to specify detailed quality requirements and analyse the artefacts of the system. In this case, however, we had a very focused prediction aim, the number of field failures in the first year. To come to valid predictions,

we decided to concentrate on a single product factor: faultiness. Faults in the product cause the failures we are interested in to predict. Therefore, we needed to estimate the number of faults and their failure rate to predict reliability. This gives us the measures we need. For a concrete prediction of the effectiveness of interaction or reliability, we needed a stochastic model that captures the relationship between faults and failures.

5.5.3 Stochastic Model

The description of the software failure process using stochastic models is a well-established part of software reliability engineering. We call those models commonly software reliability growth models, because we assume that over time software reliability grows as faults are removed by fixes. All kinds of stochastic means have been tried for describing the software failure process and for predicting reliability. You can find more details on any aspects of these models in a variety of books [139, 158].

For Siemens, we looked at four stochastic models that had shown to be good predictors in other contexts as well as a new stochastic model, which we developed specifically for Siemens based on the kind of relationship between faults and failures we found in their data.

Musa Basic

The Musa basic execution time model is one of the most simple and therefore easy to understand reliability growth models, because it assumes that all faults are equally likely to occur and are independent of each other and there is a fixed number of faults. Hence, it abstracts from introducing new faults, for example, by bug fixes. The intensity with which failures occur is then proportional to the number of faults remaining in the program and the fault correction rate is proportional to the failure occurrence rate.

Musa–Okumoto

The Musa–Okomoto model, also called logarithmic Poisson execution time model, was first described in [160]. It also assumes that all faults are equally likely to occur and are independent of each other. Here, the number of faults, however, is not fixed. The expected number of faults is a logarithmic function of time in this model, and the failure intensity decreases exponentially with the expected failures experienced. Hence, the software will experience an infinite number of failures in infinite time.

NHPP

There are various models based on a non-homogeneous Poisson process [172]. They all are all Poisson process which means that they assume that failure occurs independently. We do not model faults but the number of failure up to a specific point in time. These models are called *non-homogeneous*, because the intensity of the failure occurrence is changing over time.

Littlewood–Verall Bayesian

This model was proposed for the first time in [136]. It also does not take faults explicitly into account. Its assumptions are that successive times between failures are independent random variables each having an exponential distribution and each failure is then following a gamma distribution.

Fischer–Wagner

We developed the last model specifically for the context of Siemens and their communication systems. They had observed a geometric sequence (or progression) between failure rates of faults. NASA and IBM had already documented similar observations [1, 164, 165]. We used this relationship to develop a fifth stochastic model that included an estimation of faults. There are more details on this model available in [206, 207].

5.5.4 Time Component

A critical part of predicting reliability is to analyse time correctly. We are interested in the occurrence of failures in a certain time interval. For software, however, calendar time is usually meaningless, because there is no wear and tear. The preferable way is to use execution time directly. This, however, is often not possible. Subsequently, a suitable substitute must be found. With respect to testing this could be the number of test cases or for the field use the number of clients. Figure 5.5 shows the relationships between different possible time types.

The first possibility is to use in-service time as a substitute. This requires knowledge of the number of users and the average usage time per user. Then the question arises how this relates to the test cases in system testing. A first approximation is the average duration of a test case.

In the context of the Siemens communication systems, the most meaningful time are incidents, each representing a usage of the system. In the end, however, we want a prediction of failures over 1 year. To convert these incidents into calendar time

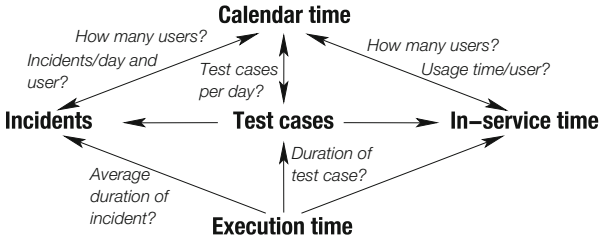

Fig. 5.5 The possible relationships between different types of time [207]

it is necessary to introduce an explicit time component. It contains explicit means to convert from one time format into another.

The number of incidents is, opposed to the in-service time, a more task-oriented way to measure time. The main advantage of using incidents, apart from the fact that they are already in use at Siemens, is that in this way, we can obtain very intuitive metrics, e.g. the average number of failures per incident. There are usually some estimations of the number of incidents per client and data about the number of sold client licences.

5.5.5 Parameter Estimation

The stochastic models describe the relationship of faults in the product to failures in the field. They rely, however, on parameters such as the number of faults to make a prediction of the reliability of the system. We need to estimate these parameters accurately to get a valid prediction.

There are two techniques for parameter determination currently in use. The first is prediction based on data from similar projects. This is useful for planing purposes before failure data is available. The second is for prediction during test, field trial, and operation based on the sample data available so far. This is the technique most reliability models use and it is also statistically most advisable since the sample data comes from the population we actually want to analyse. Techniques such as Maximum Likelihood estimation or Least Squares estimation are used to fit the model to the actual data.

For the application at Siemens we chose the Least Squares method for estimating the parameters of the models. In that method an estimate of the failure intensity is used and the relative error to the estimated failure intensity from the model is minimised. For our Fischer–Wagner model, we implemented it separately. The other models are implemented in the tool SMERFS [60] that we used to calculate the necessary predictions.

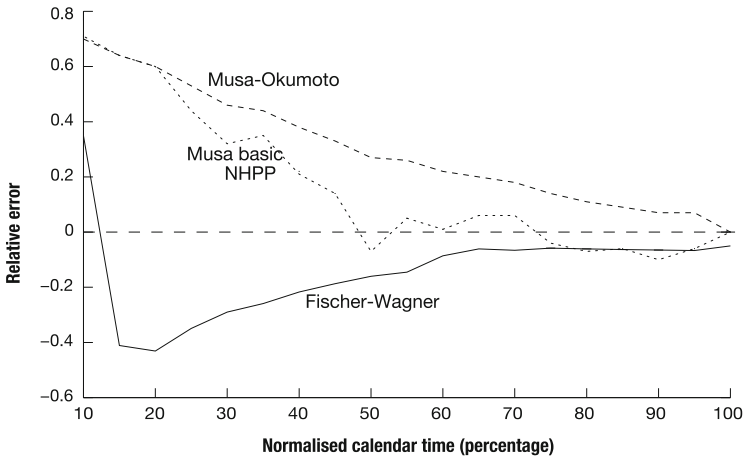

Fig. 5.6 Relative error curves for the models based on the Siemens 1 data set [207]

5.5.6 Suitability Analysis

There is no one-size-fits-all reliability growth model. We always need to run different models to find the most suitable one for any given context. For this we relied on original data from Siemens from the field trial of two products. To anonymise the projects we call them *Siemens 1* and *Siemens 2*.

We followed [159] and used the *number of failures approach* to analyse the validity of the models for the available failure data. We assume that there have been q failures observed at the end of test time (or field trial time) t_q. We use the failure data up to some point in time during testing $t_e (\leq t_q)$ to estimate the parameters of the mean number of failures $\mu(t)$. The substitution of the estimates of the parameters yields the estimate of the number of failures $\hat{\mu}(t_q)$. The estimate is compared with the actual number at q. This procedure is repeated with several t_es.

For the comparison we plot the relative error $(\hat{\mu}(t_q) - q)/q$ against the normalised test time t_e/t_q (see Figs. 5.6 and 5.7). The error will approach 0 as t_e approaches t_q. If the relative error is positive, the model tends to overestimate, and vice versa. Numbers closer to 0 imply a more accurate prediction and hence a better model.

5.5.7 Results

We give for each analysed project a brief description and show a plot of the relative errors of the different models.

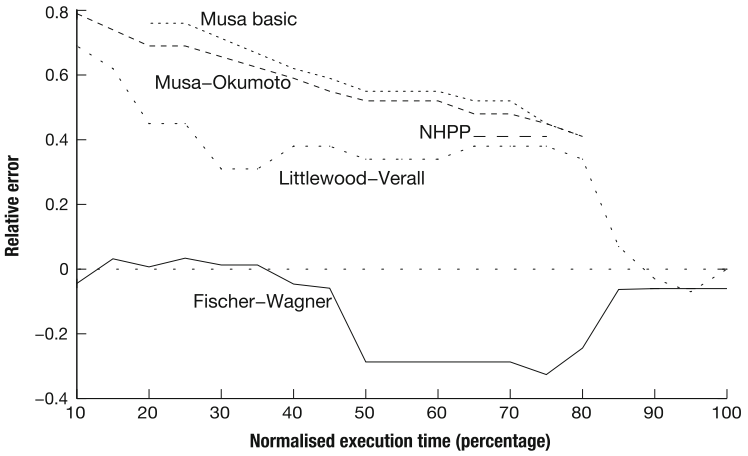

Fig. 5.7 Relative error curves for the models based on the Siemens 2 data set [207]

5.5.7.1 Siemens 1

This data comes from a large Siemens project that we call *Siemens 1* in the following. The software is used in a telecommunication equipment.

We only look at the field trial because this gives us a rather accurate approximation of the execution time which is the actually interesting measure regarding software. It is a good substitute because the usage is nearly constant during field trial. Based on the detailed dates of failure occurrence, we cumulated the data and converted it to time-between-failure (TBF) data. This was then used with the Fischer–Wagner, Musa-basic, Musa–Okumoto and NHPP models. The results can be seen in Fig. 5.6. In this case we omit the Littlewood–Verall which made absurd predictions of over a thousand future failures.

The Musa-basic and the NHPP models yield similar results all the time. They overestimate in the beginning and slightly underestimate in the end. The Musa–Okumoto model overestimates all the time; the Fischer–Wagner model underestimates. All four models make principally usable predictions close to the real value from about half the analysed calendar time. The Fischer–Wagner model has a relative error below 0.2 from 45 % on, the Musa basic and the NHPP models even from 40 % on.

5.5.7.2 Siemens 2

Siemens 2 is a web application for which we only have a small number of field failures. This makes predictions more complicated as the sample size is smaller.

It is interesting to analyse, however, how the different models are able to cope with this. For this, we have plotted the results in Fig. 5.7.

Again not all models were applicable to this data set. The NHPP model only made predictions for a small number of data points; the Musa basic and the Musa–Okumoto models were usable mainly in the middle of the execution time. All models made comparably bad predictions as we expected because of the small sample size. Surprisingly, the Fischer–Wagner model did well in the beginning but worsened in the middle until its prediction became accurate in the end again. Despite this bad performance in the middle of the execution time, it is still the model with the best predictive validity in this case. Only the Littlewood–Verall model comes close to these results. This might be an indication that the Fischer–Wagner model is well suited for smaller sample sizes.

5.5.8 Lessons Learned

The experience in quality control with Siemens was special as we did not aim to install a broad quality model in a quality control loop but to analyse a very focused area. The aim was to optimise and control system and field testing. For this, we installed only a very focused subset of an Quamoco quality model: product faults and failures which are a reduction of the interaction between the user and the system. Furthermore, we had the clear goal to have a concrete quantification of a prediction of these failures.

We employed the existing work on software reliability growth models to select and build stochastic models that can quantify the relationship between faults and failures. Based on data from system and field testing, we were able to predict the future distribution of failures. We compared several stochastic models to get the best predictive validity.

The predictive validity we observed for two Siemens systems was not extremely high with only the data from initial testing. Most models, however, showed that they can predict reasonably accurately with enough data. Especially the context-specific model performed well.

Overall, such a concrete quantification of quality goals needs investment in data collection and stochastic analysis. For reliability, there are existing models and tools which make their application easier. For other quality goals, this is probably more difficult. But even for reliability, a basic understanding of statistics and these models is necessary to successfully apply them. Especially, a suitable time component that allows an intuitive notion of time for the system under analysis is essential. If these problems are handled, however, this approach provides good predictions for planning and optimisation.

5.6 Five SMEs

Small- and medium-sized enterprises (SMEs) are important in the global software industry. They usually have not the same level of resources available for quality control. Hence, it is especially important to support them with (semi-)automatic techniques. We introduced and evaluated a set of static analysis tools in five SMEs. The following is based on our existing report [75] where you can also find more details.

5.6.1 Context

Experiences from technology transfer are only useful with information about the corresponding contexts. Therefore, we illustrate the SMEs which took part in the technology transfer, the employed static analysis techniques and the software systems we used as example applications for the transfer.

Before the technology transfer project, we had been in discussions with various SMEs about software quality control in general and newer analyses techniques such as clone detection. We realised that there is a need as well as an opportunity for SMEs and quality control because of its high potential for automation. Therefore, after discussions and workshops with several SMEs, we started a collaboration with five SMEs from the Munich area. Our goal was to transfer the static analysis techniques *bug pattern detection*, *clone detection* and *architecture conformance* to the SMEs and document our experiences.

Following the definition of the European Commission [58], one of the participating SMEs is micro-sized, two are small and two are medium-sized considering their number of employees and annual turnover. The SMEs cover various business and technology domains including corporate and local government controlling, form letter processing as well as diagnosis and maintenance of embedded systems. Four of them are involved in commercial software development, one in software quality assurance and consulting only. Hence, the latter could not provide an own software project.

Following the suggestion of the partner without a software project, we instead chose the humanitarian open source system OpenMRS,[4] a development of the equally named multi-institution, non-profit collaborative. This allowed us to analyse the five software systems briefly described in Table 5.6 in the technology transfer project. For some information, we had to contact the core developers of Open MRS directly who were responsive to our requests thankfully.

The analysed software systems contain between 100 and 600 kLOC. The developments of the systems 1–4 were done by the SMEs themselves and had

[4] http://www.openmrs.org

Table 5.6 Analysed systems [75]

System	Platform	Sources	Size [kLOC]	Business domain
1	C#.NET	Closed, commercial	≈100	Corporate controlling
2	C#.NET	Closed, commercial	≈200	Embedded device maintenance
3	Java	Open, non-profit	≈200	Health information management
4	Java	Closed, commercial	≈100	local government controlling
5	Java	Closed, commercial	≈560	Document processing

started at most 7 years earlier. The project teams contained less than ten persons. The development of the systems 1 and 2 had already been finished before our project started.

5.6.2 Introduction of Static Analysis

We used a carefully developed procedure to introduce the static analysis techniques and gather corresponding experiences in the SME context.

First, we conducted a series of workshops and interviews to convince industrial partners to participate in our project and to understand their context and their needs. In an early information event, we explained the general theme of transferring QA techniques and proposed first directions. With the companies that agreed to join the project, we conducted a kick-off meeting and a workshop to create a common understanding, discuss organisational issues and plan the complete schedule. In addition, the partners each presented a software system that we could analyse as well as their needs concerning software quality. To intensify our knowledge of these systems and problems, for each partner we performed an interview with two interviewers and a varying number of interviewees. We then compared all interview results to find commonalities and differences. Finally, we had two consolidation workshops to discuss our results and plan further steps.

Second, we retrieved the *source code* for at least three versions of the systems, in particular major releases chosen by the companies, for applying our static analysis techniques. For bug pattern detection and architecture conformance analyses, we retrieved or built executables packed with debug symbols for each of these configurations. For architecture conformance we also needed an appropriate architecture documentation. To accomplish all that, the SMEs had to provide project data as far as possible including source code, build environment and/or debug builds as well as documentation of source code, architecture and project management activities.

Third, we applied each technique on the systems by running the tools and inspecting the results, i.e. findings and statistics. To accomplish this step, the partners had to provide support for technical questions by a responsible contact or by personal attendance at the meetings.

Fourth, we conducted a short survey using a prepared questionnaire to better gather the experiences and impressions of our collaboration partners at the SMEs. These helped us to further judge our experience and to set them into context.

Bug Pattern Detection

By now, bug pattern detection is a rather conventional static analysis technique, although it is still not as widely used as we would expect considering its low effort needed and its long history. Therefore, we chose proven and available bug pattern detection tools in our technology transfer project. For Java-based systems, we used FindBugs 1.3.9 and PMD[5] 4.2.5. For the C#.NET systems, we used Gendarme[6] 2.6.0 and FxCop 10.0. We determined the tool settings during preliminary analysis test runs and experimented with different rule sets. Categories and rules which we considered as not important – based on discussion with the partners as well as requirements non-critical to the systems' application domains – were ignored during rule selection.

We chose to use a very simple classification of the rules to simplify classifying the found defects. We only distinguish between rules for *faults* (potentially causing failures), *smells* (simple to very complex heuristics for latent defects) and *minor* (less critical issues with focus on coding style).

For additional and language independent metrics (lines of code without comments, code-comment ratio, number of classes, methods and statements, depth of inheritance and nested blocks, comment quality) as well as for preparing results and visualising, we applied ConQAT.

We analysed the finding reports from the tool runs. This step involved the filtering of findings as well as the inspection of source code analysing the severity and our confidence of the findings and determining how to correct the found problems. To get feedback and to confirm our conclusions from the findings, we discussed them with our partners during a workshop.

Code Clone Detection

We used the clone detection feature of ConQAT 2.7 for all systems. In case of conventional clone detection, the configuration consists of two parameters: the minimal clone length and the source code path. In case of gapped clone detection (see Sect. 4.4.2), additional gap-specific parameters such as maximal allowed number of gaps per clone and maximal relative size of a gap are required. Based on our earlier experiences and initial experimentation, we set the minimal clone length to ten lines of code, the maximal allowed number of gaps per clone to 1 and the

[5] http://pmd.sourceforge.net

[6] http://www.mono-project.com/Gendarme

maximal relative size of a gap in our analysis to 30 %. After providing the needed parameters, we ran the analysis.

To inspect the analysis metrics and particular clones, we used ConQAT. It provided a list of clones, lists of instances of a clone, a view to compare files containing clone instances and a list of discrepancies for gapped clone analysis. We employed this data to recommend corrective actions. Also in a series of runs of clone detection over different versions of respective systems, we monitored how several parameters evolve over time.

Architecture Conformance Analysis

We also used ConQAT for this technique. For each system, we first configured the architecture conformance part of ConQAT with the path to the source code and corresponding executables of the system. Then, we created the architecture models based on the architectural information given by the enterprises. In our case, ConQAT is only able to analyse *static* call relationships between components out of the box. Next, we ran the ConQAT architecture conformance analysis which compares the relationships given in the architecture model with the actual relationships in the code. At last, we analysed the found architecture violations and discussed them with our partners.

We used an architecture model consisting of hierarchical components and allowed and disallowed call relationships between these components. The modelled components needed to be mapped to code parts (e.g. packages, namespaces or classes) as basis for the automated analysis. We excluded code parts from the analysis that did not belong directly to the system (e.g. external libraries). ConQAT is then able to analyse the conformance of the system to the architecture model. Every existing relationship that is not allowed by the architectural rules represents a defect. The tool visualises defects both on the level of components and on the level of classes which allows a detailed and a more coarse-grained analysis. To eliminate tolerated architecture violations and to validate the created architecture models, we discussed every found defect with our partners at the SMEs. This allowed us to group similar defects and to provide a general understanding.

5.6.3 Experiences with Bug Pattern Detection

We made the experience that bug patterns are a powerful technique to gather a vast variety of information about potentially defective code. We also found, however, that it is necessary to carefully configure it specific for a project to avoid false positives and get the most out of it.

First, we had difficulties in determining the impact of findings on quality factors of interest and their consequences for the project (e.g. corrective actions, avoidance

or tolerance). The rule categories, severity and confidence information by the tools helped but also were confusing sometimes.

Second, some rules exhibited many false positives either because their technical way of detection is fuzzy or because a precise finding is considered irrelevant in a project-specific context. The latter case requires an in-depth understanding of each of the rules, the impacts of findings and, subsequently, a proper classification of rules as minor or irrelevant. We neither measured the rates of false positives nor investigated costs and benefits thereof as our focus lay on the identification of the most important findings only.

Third, despite our workshops and discussions with our collaboration partners at the SMEs, we only had a limited view on the systems, their contexts and evolution. This, together with the limited support in the tools for selecting and filtering the applied rules, hampered our efforts in configuring the analyses appropriately.

We addressed the first two issues by discussing with our partners and selecting and filtering rules as far as possible so that they do not show irrelevant findings for the systems under analysis. To compensate the third issue, we had to put in manual effort for selecting the right rules. As most finding reports were technically well accessible, we utilised ConQAT to gain statistical information for higher-level quality metrics.

We achieved the initial setup of a single bug pattern tool in less than an hour. This step required knowledge about the internal structure of the system such as its directory structure and third party code. We used ConQAT to flexibly run the tools in a specific setting and for further processing of the finding reports. Having good knowledge of ConQAT, we completed the analysis setup for a system (selection of rules, adjustment of bug pattern parameters and framework setup) in about half a day.

The runs took between a minute and an hour depending on code size, rule selection and other parameters. Hence, bug pattern detection is definitely suitable to be included into automated build tasks. Part of the rules are computationally complex and some tools frequently required more than a gigabyte of memory, however. The manual effort after the runs can be split into review and reconfiguration. The review of a report took us from a few minutes up to half an hour. An overview of the effort can be found in Table 5.7.

As a result of the review of the analysis findings, we were able to identify a variety of defects of all categories and severities. We will not go into details about then numbers of findings, but we summarise the most important findings grouped by programming language [75]:

C# Among the rules with highest numbers of findings, FxCop and Gendarme reported *empty exception handlers*, *visible constants* and *poorly structured code*. There was only one consensually critical kind of finding related to correctness in system 2: unacceptable loss of precision through *wrong cast during an integer division* used for accounting calculations.

Java Among the rules with the highest numbers of findings, FindBugs and PMD reported *unused local variables*, *missing validation of return values*, *wrong use*

Table 5.7 Effort spent per system for applying each of the techniques [75]

Phase	Work step	Clone detection	Bug pattern detection	Architecture conformance
Introduction (configuration) and calibration	Analysis tools	≤ 0.5 h	≤ 1 h	≤ 0.5 h
	Aggregation via ConQAT	n/a	$\leq 0.5\,rmd$	≤ 0.5 h
	Recalibration, x-times	n/a	$\leq x \cdot 0.5$ h	n/a
Application (analysis)	Run analysis	≤ 5 min	1 min $\leq \cdot \leq$ 1 h	≤ 10 s
	Inspection of results	≤ 1 h, more for gapped CD	5min $\leq \cdot \leq$ 0.5 h	5 min $\leq \cdot \leq$ 0.5 h

of serialisable and extensive *cyclomatic complexity*, *class/method size*, *nested block depth* and *parameter list*. There have only been two consensually critical findings, both in system 5, related to correctness: foreseeable *access of a null pointer* and an *integer shift beyond 32 bits* in a basic date/time component.

Independent of the programming language and concerning security and stability, we frequently detected the pattern *constructor calls an overwritable method* in four of the five systems and found a number of defects related to *error-prone handling of pointers*. Concerning maintainability, the systems contained *missing or unspecific handling of exceptions*, manifold *violation of code complexity metrics* and various forms of *unused code*.

In addition, we learned from the survey at that end of the project that all of the partners considered our bug pattern findings to be relevant for their projects. The sample findings we presented during our final workshop were perceived as being non-critical for the success of the systems but would have been treated if they had been found by such tools during the development of these software systems. The low number of consensually critical findings correlated well with the fact that the technique was known to all partners, that most of them had good knowledge thereof and regularly used such tools in their projects, i.e. at least monthly, at milestone or release dates. Nevertheless, three of them could gain additional insights into this technique. Overall, all of the enterprises decided to use bug patterns as an important QA technique in their future projects.

5.6.4 Experiences with Code Clone Detection

Code clone detection turned out to be the most straightforward and least complicated of the three techniques. It has some technical limitations, however, that could hinder its application in certain software projects. A major issue was the analysis of projects containing both markup and procedural code like JSP or ASP.NET. Since ConQAT

supports either a programming language or a markup language during a single analysis, it is required to aggregate the results for both languages. To avoid this complication and to concentrate on the code implementing the application logic, we took into consideration only the code written in the programming language and ignored the markup code. Nevertheless, it is still possible to combine the results of clone detection of the code written in both languages to get more precise results.

Another technical obstacle was filtering out generated code from the analysed code basis. In one system, large parts of the code were generated by the parser generator ANTLR. We excluded this code from our analysis using ConQAT's feature to ignore code files specified by regular expressions.

The effort required to introduce clone detection is small compared to bug pattern detection and architecture conformance analysis. Introducing clone detection is easy because configuring it is simple. In the simplest case, it only needs the path to the source code and the minimal length of a clone.

For all systems, it took less than an hour to configure the clone detection, to get the first results and to investigate the longest and the most frequent clones. Running the analysis process itself took less than 5 min. In case of gapped clone detection, it could take a considerable amount of time to analyse if a discrepancy is intended or if it is a defect. To speed up the process, ConQAT supports that the intended discrepancies can be fingerprinted and excluded from further analysis runs. An overview of the efforts is shown in Table 5.7.

The results of conventional clone detection can be interpreted as an indicator of bad design or of bad software maintainability, but they do not point at actual defects. Nevertheless, these results give first hints which code parts must be improved. The following three design flaws were detected in all analysed systems to a certain extent: cloning of exception handling code, cloning of logging code and cloning of interface implementation by different classes.

Table 5.8 shows the clone detection results for three versions of each study object (SO), sorted by version. In the analysed systems, clone coverage (Sect. 4.4.2) varied between 14 % and 79 %. Koschke [128] reports on several case studies with clone coverage values between 7 % and 23 %. He also mentions one case study with a value of 59 %, which he defines as extreme. Therefore, the SOs 1, 3 and 5 contain normal clone rates according to Koschke. The clone rate in SO 2 is higher than the rates reported by Koschke, and for SO 4 it is extreme. Regarding maintenance the calculated blow-up for each system is an interesting value. It shows by how much the system is larger than it needs to be if the cloning would be removed. For example, version III of SO 4 is more than three times bigger as its equivalent system containing no clones. SO 4 shows that cloning can be an increasing factor over time, while SO 3 reveals that it is possible to reduce the amount of clones existing in the system code.

Interesting values are also the longest clones and the clones with the most instances. The longest clones show if only smaller code chunks are copied or whole parts of the system. Those largest clones are also interesting candidates for refactorings because they allow to reduce the most redundancy with only tackling a small number of clones. They are usually measured in *units* which are essentially

Table 5.8 Results of code clone detection [75]

SO	Version	Blow-up [%]	Clone coverage [%]	Longest clone [units]	Most clone instances
1	I	119.5	22.2	112	39
	II	118.9	23.0	117	39
	III	119.2	24.0	117	39
2	I	143.1	40.5	63	64
	II	150.2	45.4	132	47
	III	137.4	36.7	89	44
3	I	114.5	18.2	79	21
	II	111.2	15.1	52	20
	III	110.0	13.7	52	19
4	I	238.8	68.0	217	22
	II	309.6	77.6	438	61
	III	336.0	79.4	957	183
5	I	122.3	24.8	141	72
	II	122.7	25.3	158	72
	III	122.8	25.5	156	72

statements in the code. Also interesting to investigate are the clones with the most instances. That means which piece of code has the most copies. Those clones are often the most dangerous because it is very easy to forget to make changes to all copies. Therefore, they are also interesting candidates for the first refactorings.

Cloning is especially considered harmful, because it increases the chance of unintentional, inconsistent changes which can lead to faults in a system [113]. These changes can be detected when applying gapped clone detection. We found a number of such changes in the cloned code fragments, but we could not finally classify them as defects, because we lacked the knowledge needed about the software systems. Also the project partners could not easily classify these discrepancies as defects which confirms that gapped clone detection is a more resource demanding type of analysis. Nevertheless, in some clone instances, we identified additional instructions or deviating conditional statements compared to other instances of the same clone class. Gapped clone detection does not go beyond method boundaries, since experiments showed that inconsistent clones that cross method boundaries in many cases did not capture semantically meaningful concepts [113]. This explains why metrics such as clone coverage may differ from values observed with conventional clone detection. Table 5.9 shows our results of gapped clone detection.

Following the feedback obtained from the questionnaire, two enterprises had limited prior experience with clone detection; the others did not know about it at all. Three enterprises estimated the relevance of clone detection to their projects as very high; the others estimated it as medium relevant. One company stated that "clones are necessary within short periods of development". Finally, all enterprises evaluated the importance of using clone detection in their projects as medium to high and plan to introduce this technique in the future.

Table 5.9 Results of gapped code clone detection [75]

SO	Version	Blow-up [%]	Clone coverage [%]	Longest clone [units]	Most clone instances
1	I	119.9	22.3	34	39
	II	117.9	21.5	37	52
	III	117.4	22.1	52	52
2	I	116.3	19.0	156	37
	II	123.2	25.0	156	37
	III	123.7	25.3	156	37
3	I	124.4	18.2	73	123
	II	120.0	15.1	55	67
	III	118.6	20.5	55	64
4	I	192.1	58.6	42	34
	II	206.2	59.8	51	70
	III	211.1	59.5	80	183
5	I	117.4	20.7	66	68
	II	118.0	21.3	85	78
	III	118.2	21.5	85	70

5.6.5 Experiences with Architecture Conformance Analysis

We observe two kinds of general problems that prevent or complicate each architectural analysis: the absence of an architecture documentation and the usage of dynamic patterns.

For two of the systems there was no documented architecture available. In one case the information was missing, because the project was taken over from a different organisation that was not documenting the architecture at all. They reasoned that any later documentation of the system architecture would be too expensive. In another case, the organisation was aware that their system was severely lacking any architectural documentation. Nevertheless, they feared that the time involved and the sheer volume of code to be covered exceeds the benefits. The organisation additionally argued that they are afraid of having to update the documentation within several months as soon as the next release is coming out.

In system 2, a dynamic architectural pattern is applied, where nearly no static dependencies could be found between defined components. All components belonging to the system are connected at run time. Thus, our static analysis approach could not be applied.

Architecture conformance analysis needs two ingredients apart from the architecture documentation: the source code and the executables of a system. This could be a problem because the source has to be compilable to analyse it. Another technical problem occurred when using ConQAT. Dependencies to components solely existing as executables were not recognised by the tool. For that reason all rules belonging to compiled components could not be analysed. Beside these

Table 5.10 Architectural characteristics of the study objects [75]

SO	Architecture	Version	Violating component relationships	Violating class relationships
1	12 Components	I	1	5
	20 Rules	II	3	9
		III	2	8
2	Dynamic	n/a	n/a	n/a
3	Undocumented	n/a	n/a	n/a
4	14 Components	I	0	0
	9 Rules	II	1	1
		III	2	4
5	Undocumented	n/a	n/a	n/a

problems, we could apply our static analysis approach to two systems without any technical problems. An overview of all systems with respect to their architectural properties can be found in Table 5.10.

For each system, the initial configuration of ConQAT and the creation of the architecture model in ConQAT could be done in less than 1 h. Table 5.10 shows the number of modelled components and the rules that were needed to describe their allowed connections. The analysis process itself finished in less than 10 s. The time needed for the interpretation of the analysis results is dependent on the amount of defects found. For each defect, we were able to find the causal code parts within 1 min. We expect that the effort needed for bigger systems will only increase linearly but stay small in comparison to the benefit that can be achieved using architecture conformance analysis. An overview of the efforts can be found in Table 5.7.

As shown in Table 5.10, we observed several discrepancies in the analysed systems over nearly all version. Only one version did not contain architectural violations. Overall, we found three types of defects in the analysed systems. Each defect represents a code location showing a discrepancy to the documented architecture. The two analysable systems had architectural defects which could be avoided if this technique had been applied. In the following we explain the types of defects we classified together with the responsible enterprises. The companies rated all findings as critical.

- *Circumvention of abstraction layers:* Abstraction layers (e.g. a presentation layer) provide a common way to structure a system into logical parts. The defined layers are hierarchically dependent on each other, reducing the complexity in each layer and allowing to benefit from structural properties like exchangeability or flexible deployment of each layer. These benefits vanish when the layer concept is harmed by dependencies between layers that are not connected to each other. In our case the usage of the data layer from the presentation layer was a typical defect we found in the analysed systems.

- *Circular dependencies:* We found undocumented circular dependencies between two components. We consider these dependencies – whether documented or not – as defects themselves, because they affect the general principle of component design. Two components that are dependent on each other can only be used together and can thus be considered as one component, which contradicts the goal of a well-designed architecture. The reuse of these components is strongly restricted. They are harder to understand and to maintain.
- *Undocumented use of common functionality:* Every system has a set of common functionality (e.g. date manipulation) which is often grouped into components and used across the whole system. Consequently, it is important to know where this functionality is actually used inside a system. Our observation showed that there were such dependencies that were not covered by the architecture.

As a result from the final survey, we observed that four of the five participating enterprises did not know about the possibility of automated architecture conformance analysis. Only one of them already checked the architecture of their systems, however in a manual way and infrequently. Confronted with the results of the analysis, all enterprises rated the relevance of the presented technique relevant. One of them stated that as a new project member, it is easier to become acquainted with a software system if its architecture conforms to its documented specification. All enterprises agreed on the usefulness of this technique and plan its future application in their projects.

5.6.6 Lessons Learned

First, we observed that code clone detection and architecture conformance analysis have been quite new to our partners as opposed to bug pattern detection which was well known. This may result from the fact that style checking and simple bug pattern detection are standard features of modern development environments. We consider it as important, however, to know that code clone detection can indicate critical and complex relationships residing in the code at minimum effort. We also made our partners aware of the usefulness of architecture conformance analysis, both in the case of an available architecture specification and to reconstruct such a documentation.

Second, we conclude that all of the three techniques can be introduced and applied with resources affordable for small enterprises. We assume that, except for calibration phases at project initiation or after substantial product changes, the effort of readjusting the settings for the techniques stays very low. This effort is compensated by the time earned through narrowing results to successively more relevant findings. Moreover, our partners perceived all of the discussed techniques as useful for their future projects.

Third, we came across interesting findings from the analysed systems. We found large clone classes, a significant number of pattern-based bugs aside from smells and pedantry as well as unacceptable architecture violations.

In summary, in our opinion static analysis tools can efficiently improve quality assurance in SMEs, if they are continuously used throughout the development process and are technically well integrated into the tool landscape.