

Finalization and presentation of results: Finalization of the assessment report and management presentation of the results.

The result of this assessment method is a qualitative report indicating the general maturity of the engineering strategy regarding form and structuredness, strategy objects that should have been considered in the engineering strategy, the completeness and maturity of strategic statements for each important strategy object, those strategy objects where existing strategic statements are too weak or few with respect to the relevance of the strategy object, gaps in the engineering strategy in the sense of strategy objects important for the company or division without coverage by strategic statements.

In order to validate the identified strategy objects regarding completeness and coverage of relevant organizational processes, the strategy objects have been mapped against the key process areas of CMMI [CMM06]. As CMMI is a widespread process improvement maturity model for the development of products and services that aims at a wide coverage of engineering disciplines, it was assumed that the process areas described there cover a wide range of organizational processes. The detailed mapping of strategy objects against process areas is described in [PSN08].

As in particular the *strategy key area “Process”* groups all strategy objects that deal with the management of processes in general, with value chain management, quality management, etc., this grouping allows a customized view on strategic objects and strategic statements from the point of view of process engineering. It thus facilitates capturing and understanding the strategic constraints for the process engineering activity as set by the engineering strategy of an organization.

3 Software Quality Engineering

At about the same rate as software systems have been introduced in our everyday life, the number of bad news about problems caused by software failures increased. For example, last year at the opening of Heathrow’s Terminal 5, in March 2008, technical problems with the baggage system caused 23,000 pieces of luggage to be misplaced. Thousands of passengers were left waiting for their bags. A fifth of the flights had to be cancelled and—due to these problems—British Airways lost 16 million pounds. An investigation revealed that a lack of software testing has to be blamed for the Terminal 5 fiasco (ComputerWeekly.com⁴, 08 May 2008).

In August 2003 a massive blackout cut off electricity to 50 million people in eight US states and Canada. This was the worst outage in North

⁴ <http://www.computerweekly.com/Articles/2008/05/08/230602/lack-of-software-testing-to-blame-for-terminal-5-fiasco-ba-executive-tells.htm>

American history. USA Today reported: “FirstEnergy, the Ohio energy company . . . cited faulty computer software as a key factor in cascading problems that led up to the massive outage.” (USA Today⁵, 19 Nov 2003).

These and similar reports are only the tip of the iceberg. A study commissioned by the National Institute of Standards and Technology found that software bugs cost the U.S. economy about \$59.5 billion per year [Tas02]. The same study indicates that more than a third of these costs (about \$22.2 billion) could be eliminated by improving software testing.

The massive economic impact of software quality makes it a foremost concern for any software development endeavor. Software quality is in the focus of any software project, from the developer’s perspective as much as from the customer’s. At the same time, the development of concepts, methods, and tools for engineering software quality involves new demanding challenges for researchers.

In this chapter we give an overview of research trends and practical implications in software quality engineering illustrated with examples from past and present research results achieved at the SCCH. Since its foundation, SCCH has been active in engineering of high quality software solutions and in developing concepts, methods, and tools for quality engineering. A number of contributions have been made to following areas, which are further elaborated in the subsequent subsections.

- Concepts of quality in software engineering and related disciplines.
- Economic perspectives of software quality.
- Development of tool support for software testing.
- Monitoring and predicting software quality.

Concepts and Perspectives in Engineering of Software Quality 3.1

Definition of Software Quality

Software quality has been an issue since the early days of computer programming [WV02]. Accordingly a large number of definitions of software quality have emerged. Some of them have been standardized [IEE90]⁶, but most of them are perceived imprecise and overly abstract [Voa08]. To some extent, this perception stems from the different viewpoints of quality inherent in

⁵ <http://www.usatoday.com/tech/news/2003-11-19-blackout-bug-x.htm>

⁶ The IEEE Standard 610.12-1990 defines software quality as “(1) *The degree to which a system, component, or process meets specified requirements.* (2) *The degree to which a system, component, or process meets customer or user needs or expectations.*”

the diverse definitions. As a consequence, the ISO/IEC Standard 9126:2001 [ISO01] and its successor ISO/IEC Standard 25000:2005 [ISO05] decompose software quality into process quality, product quality, and quality in use. The standard recognizes software as product and reflects Garvin's general observation about different approaches to define product quality [Gar84].

Process quality: Software processes implement best practices of software engineering in an organizational context. Process quality expresses the degree to which defined processes were followed and completed.

Product quality: Software products are the output of software processes. Product quality is determined by the degree to which the developed software meets the defined requirements.

Quality in use: A product that perfectly matches defined requirements does not guarantee to be useful in the hands of a user when the implemented requirements do not reflect the intended use. Quality in use addresses the degree to which a product is fit for purpose when exposed to a particular context of use.

Quality Models

Measurable elements of software quality, i.e. quality characteristics, have to be defined in order to assess the quality of a software product and to set quality objectives. A series of attempts to define attributes of software products by which quality can be systematically described (see [Mil02]) has been combined in the ISO/IEC standards 9126:2001 [ISO01] and 25000:2005 [ISO05] respectively. The standards provides a quality model with six quality characteristics, namely functionality, reliability, usability, efficiency, maintainability and portability, which are further refined in sub-characteristics (see Figure 10).

Bugs, i.e. defects, indicate the deviation of the actual quantity of a quality characteristic from the expected quantity. Defects are often associated with deviations in the behavior of a software system, affecting its functionality. The quality model, however, makes clear that defects concern all quality characteristics of a software system. Hence, a deviation from a defined runtime performance is therefore as much a defect as a deviation from the expected usability or a flawed computation.

Quality models are a valuable vehicle for systematically eliciting quality requirements and for adopting a quality engineering approach covering all relevant qualities of a software product. For example, in the research project *WebTesting*, a guideline for methodical testing of Web-based applications (see [RWV⁺02] and [SRA06]) has been derived from a domain-specific quality model.

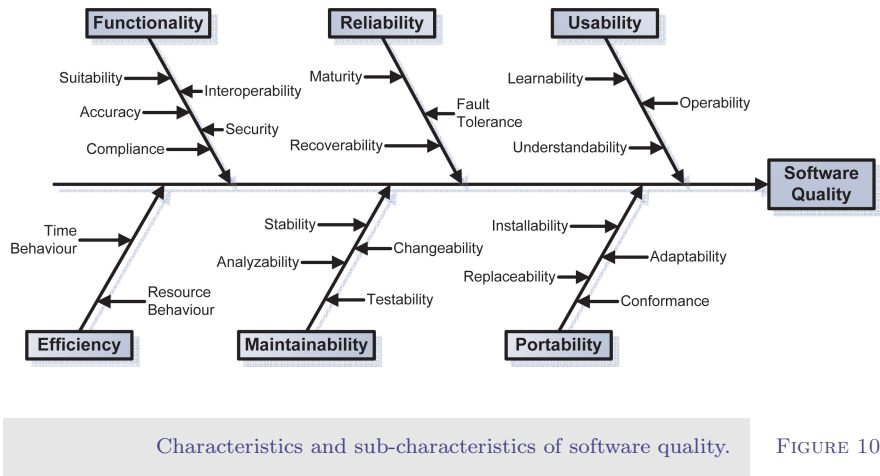


FIGURE 10

Quality Assurance Measures

Quality must be built into a software product during development and maintenance. Software quality engineering [Tia05] ensures that the process of incorporating quality into the software is done correctly and adequately, and that the resulting software product meets the defined quality requirements.

The measures applied in engineering of software quality are constructive or analytical in their nature. Constructive measures are technical (e.g., application of adequate programming languages and tool support), organizational (e.g., enactment of standardized procedures and workflows), and personnel measures (e.g., selection and training of personnel) to ensure quality a priori. These measures aim to prevent defects through eliminating the source of the error or blocking erroneous human actions. Analytical measures are used to assess the actual quality of a work product by dynamic checks (e.g., testing and simulation) and static checks (e.g., inspection and review). These measures aim to improve quality through fault detection and removal.

Economic Perspective on Software Quality

Applying quality assurance measures involves costs. The costs of achieving quality have to be balanced with the benefits expected from software quality, i.e., reduced failure costs and improved productivity. Engineering of software quality, thus, is driven by economic considerations, entailing what Garvin [Gar84] described as “value-based approach” to define quality.

Value-based software engineering [BAB⁺05] therefore elaborates on the question “How much software quality investment is enough?” [HB06]. In [RBG05] we describe how an analysis of the derived business risks can be used to answer this question when making the investment decision, which can be stated as trade-off. Too little investments in quality assurance measures incur the risk of delivering a defective product that fails to meet the quality expectations of customers and results in lost sales. This risk has to be opposed with the risk of missed market opportunities and, thus, lost sales due to too much quality investments prolonging the time-to-market. Neither too little nor too much quality investments are economically reasonable. From an economic perspective a “good enough” approach to software quality [Bac97] is considered the optimal solution.

Engineering of software quality in practice has to be coherent with economic constraints. Hence, in any application-oriented research, the economic perspective of software quality is a dominant factor. Further examples about economic considerations will be presented in the next subsections as part of the discussion about manual versus automated testing and the prioritization of tests based on the prediction of defect-prone software modules.

3.2 Management and Automation of Software Testing

Software testing is one of the most important and most widely practiced measures of software quality engineering [LRFL07] used to validate that customers have specified the right software solution and to verify that developers have built the solution right. It is a natural approach to understand a software system’s behavior by executing representative scenarios within the intended context of use with the aim to gather information about the software system. More specifically, software testing means executing a software system with defined input and observing the produced output, which is compared with the expected output to determine pass or fail of the test. Accordingly, the IEEE Standard 610.12-1990 defines testing as “the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component” [IEE90].

Compared to other approaches to engineer software quality, testing provides several advantages, such as the relative ease with which many of the testing activities can be performed, the possibility to execute the program in its expected environment, the direct link of failed tests to the underlying defect, or that testing reduces the risk of failures of the software system. In contrast, however, software testing is a costly measure due to the large number of execution scenarios required to gather a representative sample of the real-world usage of the software system. In fact, the total number of possible execution scenarios for any non-trivial software system is so high that

complete testing is considered practically impossible [KFN99]. Test design techniques (e.g., [Bei90, Cop04]) are therefore applied to systematically construct a minimal set of test cases covering a representative fraction of all execution scenarios. Still, testing can consume up to 50 percent and more of the cost of software development [HB06].

As a consequence, automation has been proposed as a response to the costly and labor-intensive manual activities in software testing. Test automation [FG99] has many faces and concerns a broad variety of aspects of software testing: The automated execution of tests, the automated setup of the test environment, the automated recording or generation of tests, the automation of administrative tasks in testing. In all these cases, tool support promises to reduce the costs of testing and to speed up the test process.

In the following, we present results from research projects conducted at SCCH that involved tool-based solutions addressing different aspects of test automation.

- The first example, *TEMPPO*, outlines the tool support for managing large test case portfolios and related artifacts such as test data, test results and execution protocols.
- In the second example, a framework for the automation of unit tests in embedded software development has been used to introduce the paradigm of test-driven development to a large software project in this domain.
- We conclude this subsection with a study about balancing manual and automated software testing subsuming ongoing observations and lessons learned from several research and industrial projects. In addition, we present a tool-based approach (*TestSheets*) for user interface testing as an example for blending automated and manual testing.

Tool Support for Test Management

Testing tools are frequently associated with tools for automating the execution of test cases. Test execution, however, is only one activity in the software testing process, which also involves test planning, test analysis and design, test implementation, evaluating exit criteria and reporting, plus the parallel activity of test management. All of these activities are amenable to automation and benefit from tool support.

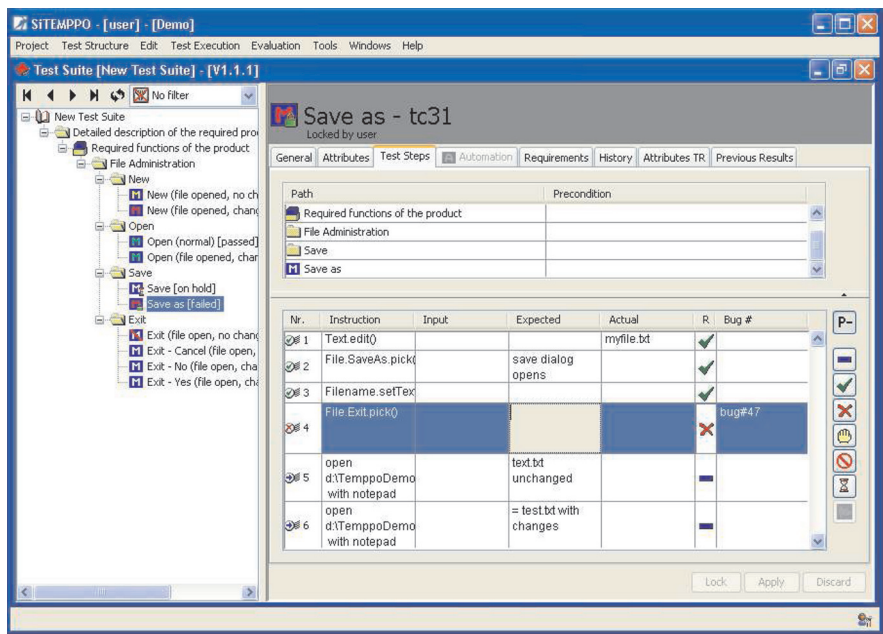
In the following we describe a tool-based approach specifically for test management and present some results from the research project TEMPPO (Test Execution Managing Planning and rePorting Organizer) conducted by Siemens Austria and SCCH. The project results are an excellent example for the sustaining benefit that can be achieved by linking science and industry. The project fostered a fruitful knowledge exchange in both directions. Requirements for managing testing in step with actual practice in large soft-

ware development projects have been elicited by Siemens, and appropriate solution concepts have been developed by researchers at SCCH. The cooperation led to a prototype implementation of a test management environment that addressed a number of research issues significant for tool-based test management in industrial projects.

- A light-weight test process for managing the different stages in the genesis of test cases had to be defined, providing support for the inception of the initial test ideas based on a software requirements specification, the design of test cases and their implementation, the manual test execution as well as the automated execution in subsequent regression testing.
- An efficient structure for organizing and maintaining large hierarchical portfolios of up to several thousand test cases had to be developed. The high volumes of related data included an extendable set of meta-information associated to test cases and a range of artifacts such as associated test scripts, test results and execution protocols accumulated over the whole software development and maintenance lifecycle.
- Changes of the test structure and test cases are inevitable in any large software project once new requirements emerge or test strategies are updated. To accommodate these changes, an integrated versioning and branching mechanism became necessary. It makes sure that results from test executions are linked to the executed version of the test cases even after changes took place.
- Sophisticated query and grouping aids had to be applied for constructing test suites combining a set of test cases for execution. Results from several consecutive test executions had to be merged in a coherent test report for assessing and analyzing the project's quality status.
- Test management as the coordinating function of software testing interacts with a variety of other development and testing activities such as requirements management and change and defect management. For example, the integration of test management and unit testing is described in [RCS03]. These integrations imply interfaces that realize a synchronization between the underlying concepts and workflows of test management and the intersecting activities, which go beyond a mere data exchange between the involved tools.

The prototype developed in the joint research project has been extended with additional features by Siemens and evolved to an industry-strength test management solution. SiTEMPPO⁷ (Figure 11) has been successfully applied in projects within the Siemens corporation all over the world, and it is licensed as commercial product for test management on the open market with customers from a broad range of industrial sectors and application domains.

⁷ <http://www.pse.siemens.at/SiTEMPPO>



The test management solution SiTEMPPO. FIGURE 11

Automation of Unit Testing in Embedded Software Development

Test-driven development (TDD) [Bec02] has been one of the outstanding innovations over the last years in the field of software testing. In short, the premise behind TDD is that software is developed in small increments following a test-develop-refactor cycle also known as red-green-refactor pattern [Bec02].

In the first step (test), tests are implemented that specify the expected behavior before any code is written. Naturally, as the software to be tested does not yet exist, these tests fail – often visualized by a red progress bar. Thereby, however, the tests constitute a set of precisely measurable objectives for the development of the code in the next step. In the second step (develop), the goal is to write the code necessary to make the tests pass – visualized by a green progress bar. Only as much code as necessary to make the bar turn from red to green should be written and as quickly as possible. Even the intended design of the software system may be violated if necessary. In the third step (refactor), any problematic code constructs, design violations, and duplicate code blocks are refactored. Thereby, the code changes performed in the course of refactoring are safeguarded by the existing tests. As soon as change introduces a defect breaking the achieved behavior, a test will fail

and indicate the defect. After the refactoring has been completed, the cycle is repeated until all planned requirements have finally been implemented.

Amplified by the paradigm shift towards agile processes and the inception of extreme programming [BA04], TDD has literally infected the developers with unit testing [BG00]. This breakthrough is also attributed to the framework JUnit⁸, the reference implementation of the xUnit family [Ham04] in Java. The framework provides the basic functionality to swiftly implement unit tests in the same programming language as the tested code, to combine related tests to test suites, and to easily run the tests or test suites from the development environment including a visualization of the test results.

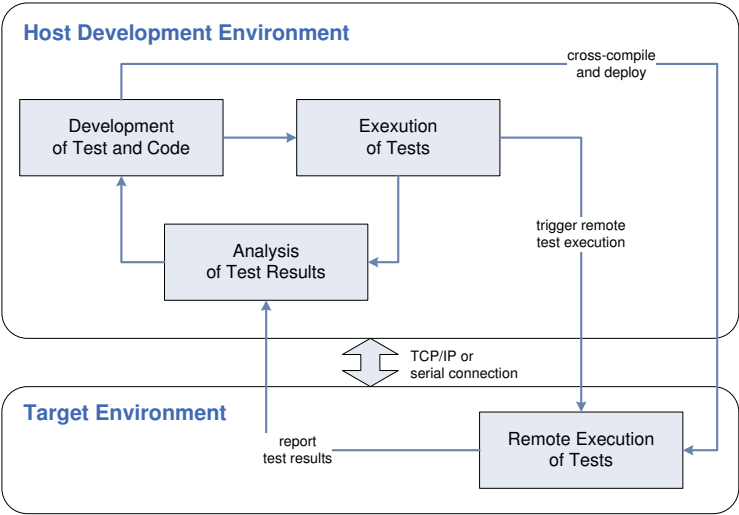
TDD has been successfully applied in the development of server and desktop applications, e.g., business software or Web-based systems. The development of embedded software systems would also benefit from TDD [Gre07]. However, it has not been widely used in this domain due to a number of unique challenges making automated unit testing of embedded software systems difficult at least.

- Typical programming languages employed in embedded software development have been designed for runtime and memory efficiency and, thus, show limited support for writing testable code. Examples are limitations in error and exception handling, lack of comprehensive meta-information, rigid binding at compile-time, and little encouragement to clearly separate interfaces and implementation.
- The limiting factor is usually the underlying hardware with its harsh resource and timing constraints that forces the developers to design for runtime and memory efficiency instead for testability. When the code is tuned to produce the smallest possible memory footprint, debugging aids as well as additional interfaces to control and to introspect the state of the software system are intentionally removed.
- Cross-platform development with a separation between host development environments and target execution platforms is a typical approach in building embedded software systems. The development tools run in a host environment, usually including a hardware simulator. Larger increments are cross-compiled and tested on the actual target system once it becomes available.
- In addition, unit testing is concerned with a number of domain-specific issues causing defects that demand domain-specific test methods and tool support. In embedded software development, these specific issues include, for example, real-time requirements, timing problems, and asynchronous execution due to multi-threaded code or decentralized systems.

The goal of the project was to tackle these challenges and to introduce the concept of TDD to the development of embedded software for mobile and handheld devices. Together with the partner company we developed a framework for automated unit testing with the aim to resemble the design of the

⁸ <http://www.junit.org>

xUnit family as closely as possible, so unit tests could be written in the restricted C++ language variant used for programming embedded devices. Beyond that, the framework comprises extensions such as to run as application directly on the mobile device or to remotely execute unit tests on the target device via a TCP/IP or a serial connection, while the test results are reported back to the the development environment on the host (Figure 12). Many defects only prevalent on the target hardware can so be detected early in development, before the system integration phase.



Workflow for unit testing in the host development environment as well as on the target device.

FIGURE 12

Balancing Manual and Automated Software Testing

Questions like “When should a test be automated?” or “Does test automation make sense in a specific situation?” fuel an ongoing debate among researchers and practitioners (e.g. [BWK05]). Economic considerations about automation in software testing led to the conclusion that – due to generally limited budget and resources available for testing – a trade-off between manual and automated testing exists [RW06]. An investment in automating a test

reduces the limited budget and, thus, the number of affordable manual tests. The overly simplistic cost models for automated testing frequently found in the literature tend to neglect this trade-off and fail to provide the necessary guidance in selecting an optimally balanced testing strategy taking the value contribution of testing into account [Ram04].

The problem is made worse by the fact that manual and automated testing cannot be simply traded against each other based on pure cost considerations. Manual testing and automated testing have largely different defect detection capabilities in terms of what types of defects they are able to reveal. Automated testing targets regression problems, i.e. defects in modified but previously working functionality, while manual testing is suitable for exploring new ways in how to break (new) functionality. Hence, for effective manual testing detailed knowledge about the tested software system and experience in exploring a software system with the aim to find defects play an important role [BR08]. In [RW06] we propose an economic model for balancing manual and automated software testing and we describe influence factors to facilitate comprehension and discussion necessary to define a value-based testing strategy.

Frequently, technical constraints influence the feasibility of automaton approaches in software testing. In the project *Aragon*, a visual GUI editor as a part of an integrated development environment for mobile and multimedia devices, has been developed [PPRL07]. Testing the highly interactive graphical user interface of the editor, which comprises slightly more than 50 percent of the application's total code, involved a number challenges inherent in testing graphical user interfaces such as specifying exactly what the expected results are, testing of the aesthetic appearance, or coping with frequent changes.

While we found a manual, exploratory approach the preferable way of testing the GUI, we also identified a broad range of different tasks that can effectively be automated. As a consequence we set up the initiative *TestSheets* utilizing Eclipse cheat sheets for implementing partial automated test plans embedded directly in the runtime environment of the tested product [PR08]. This integration enabled active elements in test plans to access the product under test, e.g., for setting up the test environment, and allows to tap into the product's log output. Test plans were managed and deployed together with the product under test.

We found that partial test automation is an effective way to blend manual and automated testing amplifying the benefit of each approach. It is primarily targeted at cumbersome and error-prone tasks like setting up the test environment or collecting test results. Thereby, partial automation enhances the capability of human testers, first, because it reduces the amount of low-level routine work and, second, because it provides room for exploring the product under test from various viewpoints including aspects like usability, attractiveness and responsiveness, which are typically weakly addressed by automated tests.

Monitoring and Predicting Software Quality 3.3

Software quality engineering is an ongoing activity. Beyond measures to achieve software quality, it requires paying close attention to monitor the current quality status of software systems and to anticipate future states as these software systems continue to evolve. In the following we show how a research project integrating software engineering data in a software cockpit can provide the basis for monitoring and predicting software quality of upcoming versions of software products.

Software Cockpits

Continuous monitoring and management of software quality throughout the evolution of a software system [MD08] requires a comprehensive overview of the development status and means to drill-down on suspicious details to analyze and understand the underlying root causes. Software cockpits (also known as dashboards or software project control centers [MH04]) have been proposed as key to achieve this vision by integrating, visualizing and exploring measurement data from different perspectives and at various levels of detail. Typical sources of measurement data are software repositories and corporate databases such as versioning systems, static code and design analysis tools, test management solutions, issue tracking systems, build systems, and project documentation.

Each of these repositories and databases serves a specific purpose and provides a unique view on the project. For a holistic view on software quality, the relevant aspects of these individual views have to be integrated. Thereby, in order to support the analysis of the project situation, it is not enough to simply present the data from different sources side by side. The integration requires modeling and establishing the relationships between the different software repositories and databases at data level [RW08]. The topic of data integration has been successfully addressed by the concept of data warehouses with its associated ETL (extract, transform, load) technologies in database research and practice [KC04].

Data warehouses are the basis for business intelligence solutions, which support managers in making decisions in a dynamic, time-driven environment based on information from diverse data sources across an organization. Test managers and quality engineers operate in a similar environment under pressure to meet high-quality standards and, at the same time, to deliver in a tight schedule and budget. Hence, as partner in the competence network

*Softnet Austria*⁹ we investigated and adopted the idea of business intelligence for software development and quality engineering [LR07].

In a study of existing approaches and solutions offering software cockpits for testing and quality management, we found an overemphasis of the reporting aspect. The main purpose of most of the studied cockpits was to generate static views of aggregated data, usually retrieved from a single data source. In contrast, Eckerson [Eck05] illustrates the nature of cockpits as the intersection between static reporting and interactive analysis. We therefore implemented a software cockpit with the objective to further explore the requirements and solution concepts for interactive data analysis. We based the cockpit on an open source data warehouse as platform for integrating project-specific data sources from development and test tools. The retrieved data was harnessed in customized software metrics and models [Kan02], which were visualized and analyzed via the cockpit. Our first prototype implementation of the software cockpit supported data extraction from open source software engineering tools such as the issue tracking tool Bugzilla or the versioning system CVS.

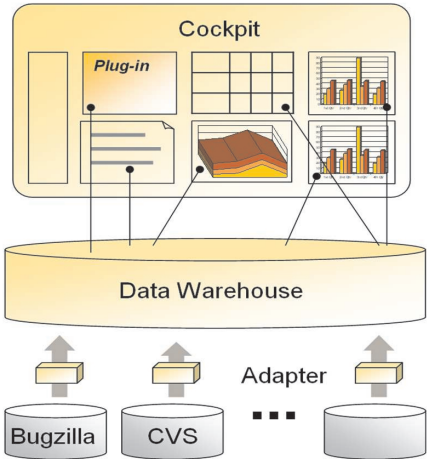
The three main tiers of the cockpit's architecture are shown in Figure 13 (from bottom to top):

1. *Data adapters* periodically extract relevant data from different repositories and databases, e.g., Bugzilla's issue database or the change log of CVS. The data is transformed to a standard data structure and stored in the central data warehouse.
2. *The data warehouse* organizes the data as cubes amenable for on-line analytical data processing. The data schema supports recording the project history for analyzing the evolution and forecasting of trends.
3. *The user interface* of the cockpit visualizes aggregated information and offers the flexibility to customize views, metrics and models. The Web-based implementation provides easy access to visual representation of the integrated data.

The first prototype of the cockpit has been developed in close cooperation with an industrial software project pursuing an iterative development process. Over a series of rapid prototyping cycles, the Web-based user interface (Figure 14) has evolved including a number of features to visualize and to analyze quality-related measurement data. Building on these results, the software cockpit has been successfully adopted in other projects and organizations, for example, a software product company developing business software involving a development team of more than 100 persons [LRB09].

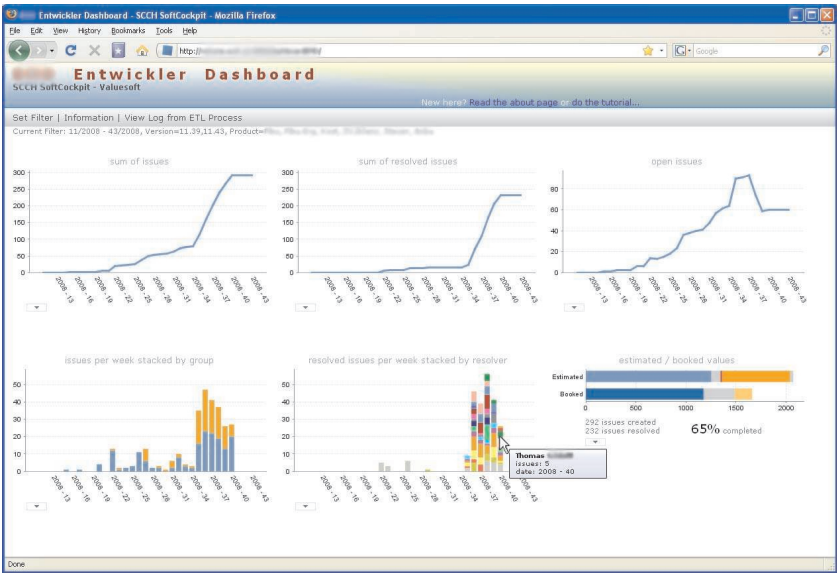
We identified a number of features that constitute key success factors for the successful implementation and application of software cockpits in practice.

⁹ <http://www.soft-net.at/>



System architecture of the software cockpit.

FIGURE 13



Interface of the software cockpit for developers.

FIGURE 14

- A user-centered design that supports the users' daily activities keeps the administrative overhead at a minimum and is in line with personal needs for feedback and transparency.
- A comprehensive overview of all relevant information is presented as a set of simple graphics on a single screen as the lynchpin of the software cockpit. It can be personalized in terms of user specific views and filters.
- The presented information (i.e. in-process metrics from software development and quality engineering) is easy to interpret and can be traced back to the individual activities in software development. Abstract metrics and high-level indicators have been avoided. This encourages the users to reflect on how their work affects the overall performance of the project and the quality status of the product.
- In addition, the interactive analysis of the measurement data allows drilling down from aggregated measurements to individual data records and in-place exploration is supported by mechanisms such as stacked charting of data along different dimensions, tooltips showing details about the data points, and filters to zoom in on the most recent information.

Predicting Defect-prone Modules of a Software System

Data about the points in time where defects are introduced, reported, and resolved, i.e. the lifecycle of defects [Ram08], is gathered in the data warehouse and can be used to construct the history and current state of defective modules of a software system. The data about the software system's past states can also serve as the basis for predicting future states of a software system, indicating which modules are likely to contain defects in upcoming versions.

The rationale for identifying defect-prone modules prior to analytical quality assurance (QA) measures such as inspection or testing has been summarized by Nagappan et al.: "During software production, software quality assurance consumes a considerable effort. To raise the effectiveness and efficiency of this effort, it is wise to direct it to those which need it most. We therefore need to identify those pieces of software which are the most likely to fail—and therefore require most of our attention." [NBZ06] As the time and effort for applying software quality assurance measures is usually limited due to economic constraints and as complete testing is considered impossible for any non-trivial software system [KFN99], the information about which modules are defect-prone can be a valuable aid for defining a focused test and quality engineering strategy.

The feasibility and practical value of defect prediction has been investigated in an empirical study we conducted as part of the research project *Andromeda*, where we applied defect prediction for a large industrial software system [RWS⁺09]. The studied software system encompasses about 700

KLOC of C++ code in about 160 modules. Before a new version of the system enters the testing phase, up to almost 60 percent of these modules contain defects. Our objective was to classify the modules of a new version as potentially defective or defect-free in order to prioritize the modules for testing. We repeated defect prediction for six consecutive versions of the software system and compared the prediction results with the actual results obtained from system and integration testing.

The defect prediction models [KL05] we used in the study have been based on the data retrieved from previous versions of the software system. For every module of the software system the data included more than 100 metrics like the size and complexity of the module, the number of dependencies to other modules, or the number of changes applied to the module over the last weeks and months. Data mining techniques such as fuzzy logic-based decision trees, neural networks, and support vector machines were used to construct the prediction models. Then, the models were parametrized with the data from the new versions to predict whether a module is defective or defect-free.

Preliminary results showed that our predictions achieve an accuracy of 78 (highest) to 67 percent (lowest). On average 72 percent of the modules were accurately classified. Hence, in case testing has to be stopped early and some modules have to be left untested, a test strategy prioritizing the modules based on the predicted defectiveness is up to 43 percent more effective than a strategy using a random prioritization. Even in with the lowest prediction accuracy the gain can be up to 29 percent compared to a random testing strategy when only 60 percent of all modules are tested. The gain over time is illustrated in Figure 15. The testing strategy based on average defect prediction results (blue) is compared to the hypothetical best case—a strategy ordering the modules to be tested according to their actual defectiveness (green)—and the worst case—a strategy ordering the modules purely random (red).

The depicted improvements in testing achieved by means of defect prediction are intermediate results from ongoing research. So far, the prediction models have been based on simple metrics derived from selected data sources. Combining the data in more sophisticated ways allows including additional aspects of the software system's history and, thus, promises to further increase the prediction performance [MGF07]. In a specific context of a project, the results can be improved even further by tuning of the applied data mining methods. For the future, we plan to extend this work to a larger set of industrial projects of various sizes and from different domains.

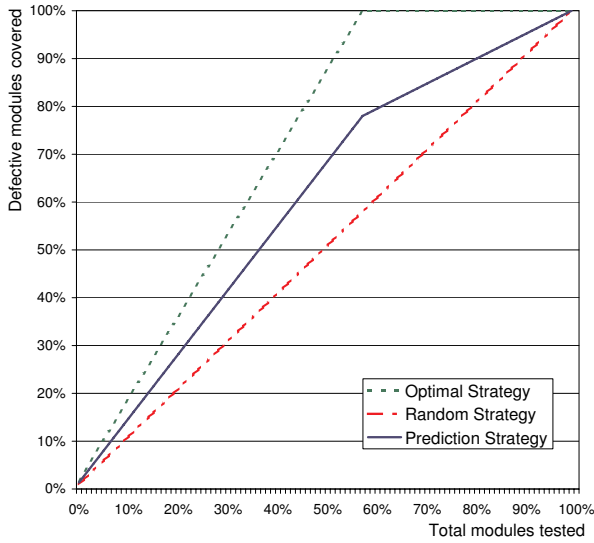


FIGURE 15 Improvement gain achieved by defect prediction.

4 Software Architecture Engineering

A software system’s architecture is an abstraction of its implementation, omitting details of implementation, algorithm and data representation, see [BCK03]. The architecture of a software system is often represented by different models, each consisting of abstract elements and relationships. Each of these models can be used to describe a particular abstract view of important structural relationships, facilitating understanding and analysis of important qualities of a software system. The fact that the abstraction defined by an architecture is not made up by one but by different structures and views providing different perspectives on a software system is reflected in a widely accepted definition of software architecture provided by the Bass et al. [BCK03], where software architecture is defined as “the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them”. Architecture is the result of design. This is reflected in a broader definition provided by Medvidovic et al. [MDT07] which state that “a software system’s architecture is the set of principal design decisions about a system”. This includes design decisions related to structure, behavior, interaction, non-functional properties, the development process, and to a system’s business position (see [MDT07]).

Hagenberg Research

Buchberger, B.; Affenzeller, M.; Ferscha, A.; Haller, M.;
Jebelean, T.; Klement, E.P.; Paule, P.; Pomberger, G.;
Schreiner, W.; Stubenrauch, R.; Wagner, R.; Weiß, G.;
Windsteiger, W. (Eds.)

2009, VIII, 488 p., Hardcover

ISBN: 978-3-642-02126-8