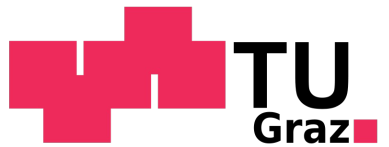

Nadja Lauritsch

Software Quality

Evaluation and Improving Projects in an Industrial Setting

Master Thesis

Fulfillment of requirements for the degree of
Master of Science

Study
Computer Science

Graz University of Technology
Faculty of Computer Science

Advisor
Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Heinz Veitschegger

Institute for Software Engineering
Research Group for Software Engineering

Graz, 13.10.2013

If you have any questions or problems and need help please contact the author (**EMAIL**).

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende wissenschaftliche Arbeit selbstständig angefertigt und die mit ihr unmittelbar verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für wissenschaftliche Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die während des Arbeitsvorganges gewährte Unterstützung einschließlich signifikanter Betreuungshinweise ist vollständig angegeben.

Die wissenschaftliche Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Nadja Lauritsch, Graz 13.10.2013

Acknowledgements

It is a pleasure to thank the many people who made this thesis possible, especially Heinz Veitschegger and Jörg Schuntermann from the Infineon Technologies AG and professor Dr. Franz Wotawa from TU Graz, for their guidance and support. I also have to thank my parents, Angelika More, Alexander Maessen and many more, who gave me inspiration and reviewed my thesis. I am truly grateful for the opportunity to have had such a dynamic and capable thesis committee.

Abstract

Software quality has become more and more important, but how is this quality measured and what does software quality mean? In literature it is not defined clearly. There are many techniques, which have an influence on the quality of a software project. From developing processes, coding guidelines, different test methods to software metrics, they all play a role for the resulting quality. This thesis covers these techniques and focuses on calculating static code metrics. Finally an evaluation for two projects (c# and java) is done with Sotoarc and Sotograph.

Keywords: Software quality, software metrics, coding guidelines, testing techniques, Sotoarc, Sonograph

Contents

1 Introduction **1**

1.1 Aim of the master thesis 1

1.2 Objectives 3

1.3 Overview 3

2 Preliminaries **5**

2.1 Software quality 5

2.1.1 Importance of software quality 7

2.1.2 Costs to repair defects 7

2.2 Product quality 8

2.2.1 Constructive quality assurance 8

2.2.2 Documentation 15

2.2.3 Analytical quality assurance 15

2.3 Process quality 17

2.3.1 Software infrastructure 17

2.3.2 Management process 17

2.3.3 Maturity model 20

2.4 Design Patterns 23

2.4.1 Singleton 23

2.4.2 Observer Pattern 24

2.4.3 Structural Violations 27

2.5 Testing 29

2.5.1 Planning and Monitoring 29

2.5.2 Manual test 31

2.5.3 Unit test 31

2.5.4 Functional test 33

2.5.5 Smoke test . . . 33
2.5.6 System test . . . 33
2.5.7 Acceptance test . . . 34
2.5.8 Regression test . . . 34
2.6 Chapter summary . . . 35

3 Metrics **37**
3.1 Object oriented principles . . . 39
3.1.1 Abstraction, encapsulation and information hiding . . . 39
3.1.2 Coupling . . . 40
3.1.3 Cohesion . . . 42
3.1.4 Inheritance . . . 42
3.1.5 Polymorphism . . . 43
3.2 Common metrics . . . 43
3.3 Software quality evaluation . . . 46
3.3.1 Evaluation of the questionnaires . . . 47
3.3.2 DND . . . 49
3.3.3 ITec . . . 52
3.4 Chapter summary . . . 59

4 Tools **61**
4.1 Sotoarc and Sotograph . . . 61
4.1.1 Sotoarc . . . 61
4.1.2 Sotograph . . . 69
4.1.3 Choice of metrics . . . 71
4.1.4 Metrics prioritization . . . 76
4.2 Visual Studio 2010 . . . 78
4.2.1 Metrics in VS 2010 . . . 78
4.2.2 Code coverage . . . 78
4.3 Comparison of Sotograph and VS 2010 . . . 79
4.4 Comparison of Analyst4j and VizzMaintenance . . . 79
4.4.1 Metric results . . . 80
4.4.2 Analysis . . . 80
4.5 Usability . . . 81

4.5.1 Sotoarc and Sotograph 81
4.5.2 Visual Studio 2010 82
4.5.3 Analyst4j 82
4.5.4 VizzMaintenance 82
4.6 Chapter summary 82

5 Conclusion and future work **83**
5.1 Conclusion 83
5.2 Future work 84

List of Figures **85**

Bibliography **87**

1 Introduction

When facing the task of building a skyscraper the first step contains a lot of planning work. Every possibility must be carefully examined to avoid any unexpected surprises in the construction process. In other words, all necessities should be discussed and clearly defined before starting the construction work. Questions, like the material used, companies involved, time management and numerous more have to be answered initially, so that everything goes smoothly and the end result is a high quality skyscraper. The same method should be applied for the software development process. It must be predictable, understandable and controllable to a high software quality level. This scenario is optimal, but sometimes reality looks completely different. There is not enough time to develop all the requirements, especially as far as testing and documentation is concerned. To ensure high software quality, many books advise taking time and using several analyzing and testing methods for preventing failures. If you follow that advices, it takes you more time at the beginning, but at the end of the project you have a real benefit. Experience has shown that the problem in this case is that developers have too little time to keep the quality assurance process in mind. Under the pressure of a deadline, the solution must be ready for the customer and the developer turns a blind eye to the existing failures. Everything that is not really necessary for this release will be postponed. There must be a solution to meet the deadline and also develop high quality software.

This master thesis is about evaluating existing applications. Figure 1.1 shows how many factors play a role in software development for producing software on a high quality level. Software quality [Hof09] is divided into product and process quality. Many techniques can be used to improve the quality, such as using unique guidelines, testing and source code analyzing.

1.1 Aim of the master thesis

The challenge of this master thesis was to analyze existing projects from the Infineon IT-department. By analyzing and evaluating these projects, I proposed some techniques to software engineers in order to improve their current situation. The focus was on source code analysis and improving software quality, because they are long-lived projects with a lot of erosion. These two projects are iTec (Java) and DND (C#). Chronological iTec started in 2000 and DND in 2003.

During my work I defined the following assumptions:

- A1: Coding guidelines have an effect on software quality.
- A2: It is possible to measure software quality with software metrics.

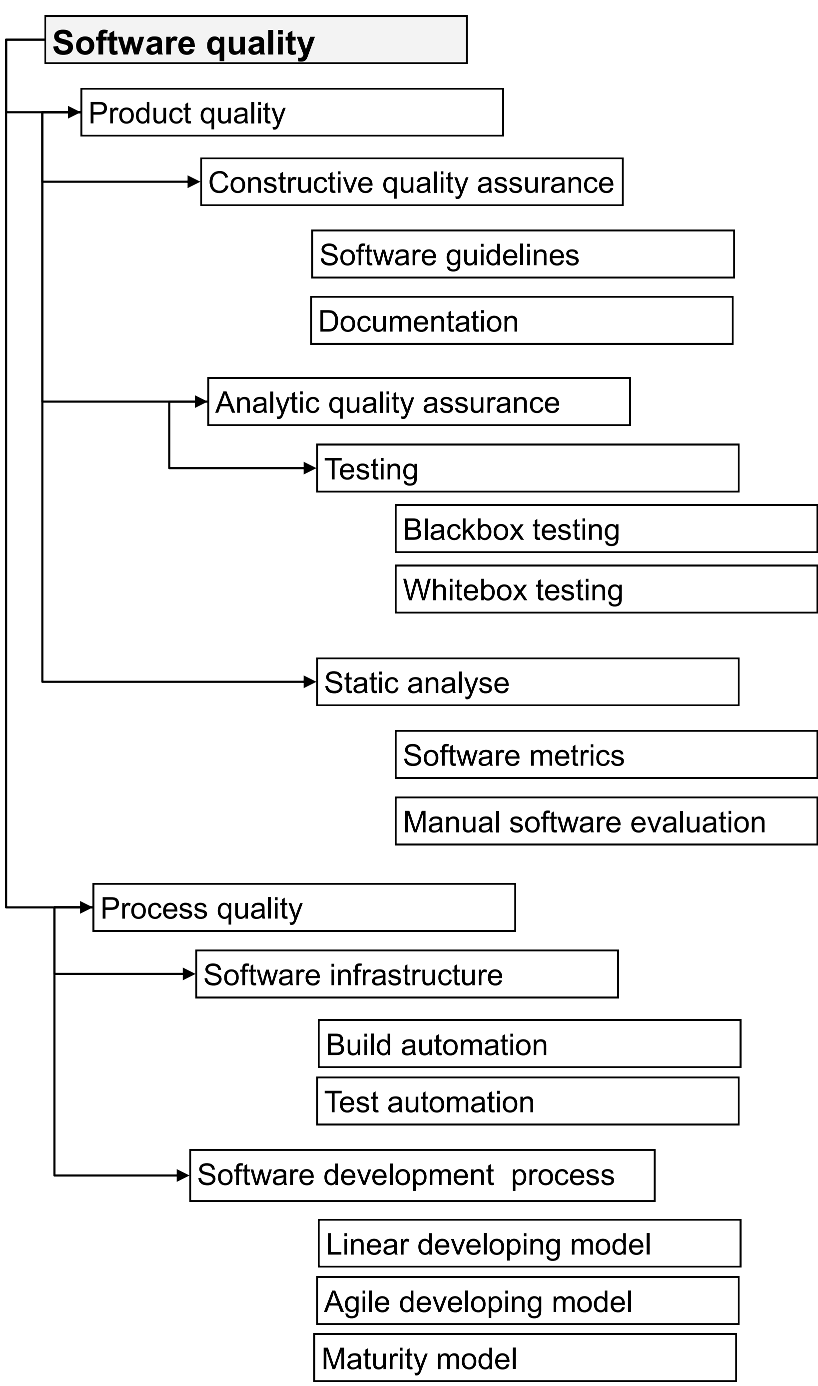

Fig. 1.1: Techniques for improving software quality

1.2 Objectives

The primary objectives for this master thesis were finding a tool for analyzing projects and calculating software metrics. In addition to analyzing selected projects, finding a set of metrics which can be checked regularly, and improving software quality. A condition for this thesis was to evaluate the tools by their usability and another was to find factors, which influence and describe the software quality.

1.3 Overview

The thesis is structured in the following way:

Chapter 2 covers the topics *product and process quality. Product quality* summarizes coding guidelines, documentation and analytical quality assurance. In contrast, *process quality* deals with software infrastructure. Both, product and process quality, have an influence on software quality in the form of source code readability and how the requirements can be developed and implemented. It also contains a section about design patterns, as they are measured by Sotograph and are clarified by an example in order to get a better understanding. Finally, various test techniques are explained briefly.

Chapter 3 starts with a theoretical background about software metrics and object oriented principles. At the end of this chapter the questionnaire and evaluation of the selected projects DND and iTec are listed.

Chapter 4 includes the tools Sotoarc, Sotograph, Visual Studio 2010, Analyst4j and finally VizzMaintenance. Differences are found by comparing calculated metric results.

Finally, **Chapter 5** comprises of the conclusion, summary and future research suggestions.

2 Preliminaries

In this chapter the term software quality is explained and described on the basis of software criteria. Software quality can be divided into product and process quality, which are explained. Finally, some basic design patterns and test techniques are described.

2.1 Software quality

The term software quality seems to be a simple expression, however several definitions for it exist. Ambiguities and misunderstandings are preprogrammed [BLL04]. One software quality definition is "meeting the requirements". Requirements must be measurable and will either be met or not met. Software quality can also be described as the conformance to functional and non functional requirements. Another definition is the degree to which a system, component, or process meets the customer expectations. All of the project's necessary software criteria must be fulfilled for an error-free usage. There is not only one criterion for measuring the software quality, but also other criteria, such as functionality, reliability, usability, performance, maintainability, portability and reusability. The customer and developer have different opinions on software quality or what is important for the quality of their projects. These opinions are based undoubtedly on their own experiences. An overview [ISO03a], [ISO03b] of the quality criteria is shown in figure 2.1 [ISO01], which is described in the ISO\IEC 9126.

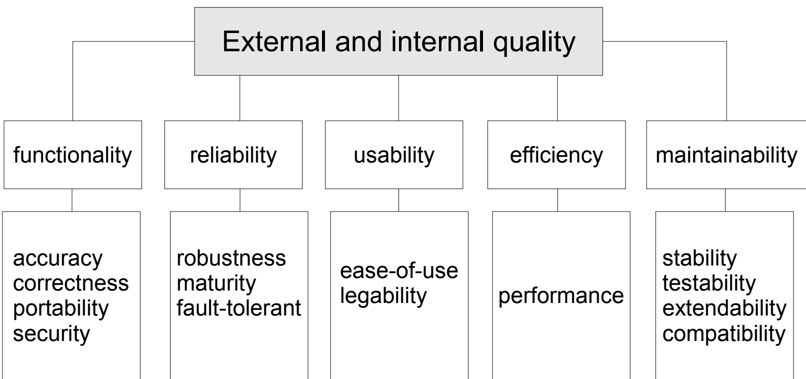

Fig. 2.1: Internal and external quality criteria

- **Functionality** describes how the required functionality is really put into practice and what the software does to fulfill the customer needs. It consists of accuracy, correctness, portability and security. **Accuracy** describes the ability to provide the proper results with the needed degree of precision. **Correctness** is the degree to which the software performs the required functions accurately, and so the goal of each program is to operate correctly. The ability to transfer the software from one environment to another is defined as **portability**, e.g. can the application run on 32 and 64 bits machines? Does it run only on Win 7? **Security** is the ability of a system to protect the information from unauthorized persons with respect to confidentiality and integrity.
- The definition for **reliability** is the capability of the software to run under specific condition for a stated period of time. Reliability contains robustness, maturity and fault-tolerance. **Robustness** is the measure of how often the software freezes or crashes. **Maturity** is the capability to avoid failures caused by software faults. The software product has to maintain a specific level of performance in cases of software faults. In the event of a component failing, a backup component can take its place without any loss of service. These properties determine the **fault-tolerant** criteria.
- **Usability** is how easily the customer can handle the software. It contains ease-of-use and legibility. **Ease-of-use** mean that the customer can use the software intuitively. **Legibility** is characterized by the facility of reading the source code. Guidelines help the developers implement a uniform style of code.
- For **performance**, it is important how the application fulfills its purpose without any waste of resources. Time behavior and the efficiency are relevant for this criterion.
- **Maintainability** describes the ease in understanding the software and contains testing, extending, modifying and verifying it. Maintainability is a very important aspect and quite difficult to quantify. **Stability** is characterized as the capability of the software product to avoid unexpected effects from modifications of the software, e.g. how often the developer needs to fix problems. The definition for **testability** is the assurance that functions work correctly and do not occur within unexpected errors. It is a method for evaluating the source code and finding errors and failures. **Extendibility** amounts to further growth. It should be easy to extend the software with new modules. **Reusability** is the ability to reuse some modules from an existing software project into other projects and also deals with compatibility. **Compatibility** is the ability of using a new version of the application with an older version‘s data.

There are two groups of software quality criteria [Hof09]. The first one is the external criteria, which are directly reflected to the customer. These criteria are functionality, performance, reliability and usability. When it comes to the customer‘s buying behavior, these criteria are essential. The external criteria are important for management, considering that they can observe the progress of the project. The second group of software quality criteria is internal criteria like testability, maintainability and portability. Only customers with a deeper understanding for the software development can orient themselves on the internal criteria. Both, the internal and the external criteria are very important. It‘s not possible to meet all these criteria at the same time, for example portability and performance interfere with each other. Ronald A. Fisher formulated this already 1958 and Gerald M. Weinberger quoted it like:

"The better adapted a system is to a particular environment, the less adaptable it to new environments."

Gerald M. Weinberg [Wei98]

Every project has to fulfill certain software quality criteria. Knowing which techniques can be used to measure these criteria is essential.

2.1.1 Importance of software quality

Before listing the aspects of the software quality importance, the following factors explain why software quality can be bad and how the developer can handle that [Hof09].

The core problem in software development is the increasing complexity, due to intricacy of software projects, which tends to be high. Any system with a higher degree of complexity will have difficulty to reaching a certain level of reliability. The next factor is the average size of projects, which is continuously growing as well. Therefore, the projects get more inflexible. In earlier days of computer engineering a single developer could control and understand a project; nowadays, this can only be managed by a large team. Another problem is in the process of time many different developer generations are working on one and the same project. If the project lasts about 10 years, the second or third generation is developing it. Each developer has his own coding style, which is directly reflected in the source code. As a result, long-living software is losing clarity and structure. In summary the software developers are fighting against complexity and growing software size to save clarity and structure while simultaneously simplify the maintainability. Brian Kernighan described it as follows:

"Controlling complexity is the essence of computer programming."

Brian Kernighan [KP76]

Some techniques exists, including testing, refactoring and measuring software with metrics, which helps to control the software problems. Many factors play a role in the quality of a software product, but software quality is also a team effort. The software quality has an effect on the costs as well.

2.1.2 Costs to repair defects

Defects should be detected and removed as early as possible in the software life cycle. The later they are discovered, the harder to solve and the higher the costs to fix them. In [Zel06] there is an analysis about the costs of detecting defects.

Phase when defect is detected	Relative cost to repair
Requirements analysis	1
Design	5
Coding	10
Unit test	20
Acceptance test	50
Maintenance	200

Tab. 2.1: Relative cost to repair defects

Figure 2.2 shows if the defect is discovered and corrected during development, it is vastly less expensive than a defect released to production. The source is from IBM Systems Sciences Institute.

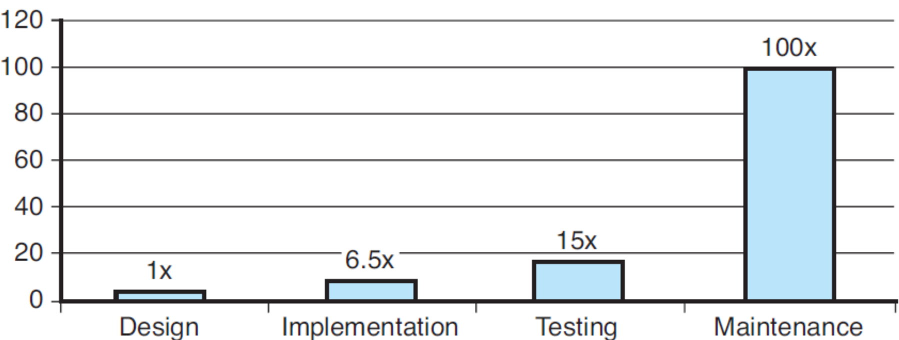

Fig. 2.2: The cost of a defect in the software lifecycle [Lev11]

2.2 Product quality

Besides process quality, product quality has an influence on software quality. It contains constructive quality assurance and analytical quality assurance [Hof09]. This includes everything that has an effect on software, especially on source code such as coding guidelines, documentation and testing.

2.2.1 Constructive quality assurance

For constructive quality assurance it is essential to fulfill the criteria for a software product. This involves using the same software guidelines for a programming language within the team in addition to documenting the source code.

Software guidelines

Software guidelines are important for programming within a team. If every team member is using the same coding style, the programmer feels more comfortable with the code written by others. If there is a new member joining the team, then it is easier to read and understand the source code.

The most important rules for programming are extracted from the Clean Code Developer book, which are listed below [Mar08]:

Meaningful names

- Choosing good names
  The name of a variable, function, or class should answer all big questions. The goal is to clarify why the variable exists, what it does and how it is used.
- Avoid leaving false clues, that confound the meaning of the code, e.g. the variable accountList should be a List.
- The names should be pronounceable, so that the team is able to discuss it.
- Single letter variable names are only okay for loops, e.g. i or j, hence mental mapping should be avoided.

- Class and object names should have noun or noun phrase names like Customer and Product.
- Methods should have verb or verb phrase names for instance showProducts() or print().
- Avoid using the same verb for two purposes e.g do not use add for both, a method that adds two values and returns a result and another method that puts a parameter into a collection.
- Avoid drawing every name from the problem domain. Use computer science terms, algorithm names, pattern names or math terms.
- Place names in context by enclosing them in well-named classes, functions or namespaces.

Functions

- Keep your functions small. Each function should hardly ever be 20 lines and longer.
- Functions should only do one thing. Switch statements always do n-things. They can be tolerated, if they appear only once. They are used to create polymorphic objects and are hidden behind an interface relationship, so that the rest of the system cannot see them.
- Do not mix levels of abstraction within one function, e.g. abstract concepts and string manipulation.
- The code should be read from top to bottom, so that every function is followed by those at the next level of abstraction.
- Using more then two arguments for a function should be avoided. Instead of a long list of parameters, use instance variables.
- Avoid flag arguments; a boolean argument is an indicator that the function does more than one thing.
- When a function seems to need more than two arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.
- Functions should either do something or answer something and not both! Separate the command from the query.
- Separate the bodies of the try and catch blocks out into functions of their own, otherwise error processing and normal processing would be mixed into one function.
- Do not repeat yourself! Concentrate code into base classes that would otherwise be redundant.
- Every function, and every block within a function should have one entry and one exit. This means that there should only be one return statement in a function, no break or continue statements in a loop and never any goto statements. This is valid for large functions. If you keep your functions small, then the occasional multiple return, break or continue statement does no harm.

Comments

- Comments are necessarily bad and should be avoided. The older a comment is, the farther away it is from the code it describes. Code tends to be refactored, although comments do not.

- Comments do not make up for bad code. It is better to spend time in writing comments, that explain the code, so make sure code is clean!
- Examples for good comments:
  - Legal comments, such as copyright and authorship statements.
  - Informative comments like those explaining a regular expression, which is intended to match.
  - Explanation of why something was implemented in a certain way
  - Warn other developers about consequences, e.g. by using a design pattern. It is a proofed solution, but it also has consequences.
  - TODO comments, but do not clutter your code with them.
  - Javadocs in public APIs, but beware that they can be just as misleading as any other comment, if they are not up to date.
- Examples for bad comments:
  - Redundant comments like *String customerName; //Name of the customer*.
  - Misleading comments including statements, that are not precise enough to be true.
  - Mandated comments - it is just plain frivolous to have a rule that says every function must have a javadoc or every variable a comment.
  - Journal comments - as nowadays we have source control, there is no need to log changes of the source file within the respective file anymore.
  - Banners like //*******Service executions***** followed by a number of methods handling various services.
  - Closing brace comments, e.g. } // end if (user.isLoggedOn). Shorten your functions and then these kinds of comments become obsolete.
  - Commented out code, if you do not need it anymore, delete it. If you need the source again after you have deleted it, you can get it from the source control.
  - HTML comments - It is not the responsibility of the programmer to adorn the comments with appropriate HTML.
  - If you have to write a comment, then make sure it is near the code it describes. Do not offer system-wide information in the context of a local comment.
  - No historical discussions or irrelevant descriptions of details in your comments.
  - The connection between a comment and the code it describes should be obvious.
  - Function headers - A well-chosen name for a small function, that does one thing is usually better than a comment header.
  - Javadocs in nonpublic code.

Formatting

- Take care that your code is nicely formatted. Have an automated tool, which can apply formatting rules for you. Style and discipline survives, however code does not.

- Using the newspaper metaphor:
  - The name of a source file should be simple, but explanatory.
  - The upper most part of the source file should provide the high-level concepts and algorithms.
- Separate concepts with line blanks, e.g functions.
- Lines of code, which are closely related, should appear vertically dense.
- Variables should be declared as close as possible to their usage.
- If one function calls another, they should be vertically close and the caller should be above of the callee.
- Conceptual affinity can be a group of functions performing a similar operation such as assertTrue() or assertFalse().
- The line edging is depending on your screen solution, but it never should be more than 100-120 characters.
- Make hierarchy of scopes visible by indenting the lines of source code in proportion to their position in the hierarchy.

Objects and data structures

- Objects hide their data behind abstractions and expose function to operate on that data.
- Data structure exposes their data and has no meaningful functions.
- Choose the right thing for the right purpose:
  - Procedural code makes it hard to add new data structures, as all the functions have to be changed.
  - Object oriented code makes it hard to add new functions, as all of the classes have to be changed.
- A module should not know about the innards of the object it manipulates. A method f of a class C should only call the methods of:
  - C
  - an object created by f
  - an object passed as an argument to f
  - an object held in an instance variable of C
- Avoid train wrecks like opaHandler.getOpa("XY").getValue().equals("Y") and split it up.
- Data transfer object is a class with public variables and no functions.

Error handling

- Error handling is important, but if it obscures logic, then it is wrong.
- Use exceptions rather than return code.
- Create informative error messages and pass them along with your exceptions. Log the error in your catch.

- Wrap APIs, especially third party APIs. Make sure that it returns a common exception type. The information sent with the exception can distinguish the errors. Use different classes, only if there are times when you want to catch one exception and allow the other ones to pass through.
- Use the special case pattern. Create a class or configure an object so that it handles a special case. That way, the client does not have to deal with exceptional behavior.
- Do not return null. Instead it is better to consider throwing an exception or returning a special case object.
- Do not pass null.

Boundaries

- Use generics.
- When you use a boundary interface like Map, keep it inside the class or close to family classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs.
- Use learning tests that call the third party API to verify if the third party packages work the way, they are expected to. When there are new releases of the third party package, run the learning test to see whether there are behavioral differences.
- Use the Adapter pattern to encapsulate the interaction with an API, whose result has only a single place to change when the API evolves.

Unit tests

- There are three laws of TDD (Test Driven Development):
  - You may not write production code until you have written a failing unit test.
  - You may not write more of a unit test than is sufficient to fail and not compiling is failing
  - You may not write more production code than is sufficient to pass the currently failing test.
- Keep tests clean! Tests have to change as the production code evolves. The dirtier the tests, the harder they are to change.
- Tests enable change and so if you have tests, do not fear making changes to the code.
- Clean tests have to be simple, succinct and expressive, though test code does not need to be as efficient as production code.
- Use the Build-Operator-Test pattern:
  1. The first part builds up the test data.
  2. The second part operates on the test data.
  3. The third part checks if the operation yielded the expected results.
- Minimize the number of asserts per test.
- Test a single concept in each test function.

- Another five rules of clean tests:
  1. Fast - Tests should be fast, in the interest of running them frequently.
  2. Independent - Tests should not depend on each other. One test should not be set up on the conditions of the next test.
  3. Repeatable - Tests should be repeatable in any environment.
  4. Self-Validating - Test should have a boolean output for pass or fail.
  5. Timely - Unit tests should be written just before the production code.

Classes

- Classes should be small.
- Class organization in the following order:
  – Public static constants
  – Private static variables
  – Private instance variables
  – Public functions
  – Private functions
- Maintain privacy. Only use loose encapsulation as a last resort.
- Use the single responsibility principle; every class should have onley one responsibility.
- Classes should have a small number of instance variables. Each of the methods of the class should manipulate one or more of those variables. When classes lose cohesion, you have to split them up.
- Classes should be open for extensions, but closed for modifications. Incorporate new features by extending the system and not by making modifications to existing code.
- Introduce interfaces and abstract classes.
- Dependency Inversion Principle - classes should depend upon abstraction, instead of concrete details.

Systems

- Appropriate levels of abstraction: some people are responsible for the big picture, while other focus on the details.
- Separate the startup process. When the application objects are constructed and the dependencies are "wired" together, from the runtime logic that takes over after startup. Achieve this by:
  – Moving all aspects of construction to main() or functions called by main().
  – Using factories to make the application responsible for when an object is created.
  – Using dependency injection and inversion of control.
- It is a myth that we can get the systems right on the first try. Focus on implementing only today‘s stories, then refactor and expand the system to implement new stories tomorrow.

- Postpone decisions until the last possible moment. This lets you make informed choices with the best possible information.
- Use Domain-Specific languages to minimize the communication gap between the domain concept and the code, that implements it.
- When designing a system, use the simplest thing that can possibly work.

Emergence A design is simple when it:

- Runs all the tests
  Making the system testable pushes you toward a design where the classes are small and have only a single purpose. The fact that we have tests eliminates the fear, that cleaning the code will break it.
- Contains no duplication
  Use the Template Method pattern to remove higher-level duplication.
- Expresses the intent of the programmer
  The clearer the author can make the code, the less time others will have to spend understanding it. Use standard nomenclature and design patterns.
- Minimizes the number of classes and methods
  High class and method counts are sometimes the result of pointless dogmatism. Although it is important to keep class and function count low. It is more important to have tests, eliminate duplication and express yourself.

Concurrency

- Separate concurrency design from the rest of the code.
- Keep your synchronized sections as small as possible.
- Avoid using more than one method on a shared object.
- Copy objects and treat them as read-only.
- Threads should be as independent as possible, sharing no data with any other thread.
- Use the Java thread-safe collections.
- Write tests that have the potential to expose problems and run them frequently with different configurations and load. If tests ever fail, track down the failure.
- Do not ignore system failure as one-offs.
- Allow the number of threads to be easily tuned.
- To encourage task swapping, run with more threads than processors or cores.
- Run your threaded code on all target platforms early and often.
- Use jiggling strategies while another implementation generates a random number to choose between sleeping, yielding or just falling through.

Microsoft also provides guidelines for developers [mic11]. Design guidelines for developing class libraries are for library development that extends and interacts with the .NET Framework. This guideline describes rules for the naming conventions and how to name types and members in

class libraries. It also includes the information on how to use static and abstract classes, how to design libraries, that can be extended and how to design exceptions.

2.2.2 Documentation

Documentation [Hof09] is divided into external and internal documentation. For the costumers external documentation is important, like the specification document, user manuals or online help. Internal documentation is not visible to the customers; it is more attractive to developers. Each developer has its own documentation style. In most cases it is seen as irksome and needless. If only one implements the project, he knows the functionality of the project. But what happens when he leaves the company? The result is possibly a complex source code, which may be difficult for new programmers to understand. Solving this problem, documentation is extremely vital. It is one essential part of the software product and has an effect on quality as well.

2.2.3 Analytical quality assurance

Analytical quality assurance contains testing methods and static analysis performed on an existing software system. The main goal is to evaluate and analyze the quality properties of the systems. It contains software testing and statical analysis.

Software testing

For measuring software product, quality criteria are defined. For measuring these criteria, they are divided into external and internal. Table 2.2 shows the above mentioned criteria and which testing method could be used to measure each criterion. If there is a check mark next to the criterion, this means it is visible to the user or developer (dev).

Name	Dev	User	Measuring with
Functionality	✓	✓	Unit test, acceptance test, integration test, regression test, system test
Efficiency	✓	✓	Performance test, load test,stress test
Reliability	✓	✓	Stress test, integration stress test
Portability	✓	✓	Compatibility test
Integrity	✓	✓	Data test
Usability	0	✓	Acceptance test, prototyping, usability test
Maintainability	✓	0	Unit test, acceptance test, coding guidelines
Documentation	✓	✓	Inline documentation, user manual
Legibility	✓	0	Coding guidelines
Scalability	✓	✓	Performance test

Tab. 2.2: Measuring quality criteria

The above mentioned testing techniques are explained in chapter 2.5.

Statical analysis

In most cases, statical analysis is performed on source code. It guarantees that the software meets all the quality requirements. It can be performed manually or automatically. Software metrics are a key topic. In chapter 3, you can find a detailed description of metrics, which are a way to analyze your source code automatically. Manual software evaluation [Hof09] works

without any computer support. It contains walkthroughs, reviews and inspections, which are applied to perform statical analyses.

The principle goal of walkthroughs is to evaluate parts of the source code by two or more developers. A discussion takes place about the existing source code and how to improve it. Applying the principle of multiple-eye verification, it is possible to find errors, which the author does not see anymore. Walkthrouhgs can be combined with the search for failures. A review is an formal variation of the walkthrough. The evaluation is based on check lists, which guarantee completeness and comparableness. The most formal variation of manual software evaluation is the inspection. For this different phases are defined: planning phase, overview phase, preparation phase, inspection, post processing and checking phase.

The following enumeration compares the different ways an analysis can be performed.

- Informal vs. formal
  For informal evaluations, the know-how of developers plays a role and there are no strict rules on how the evaluation should be performed. Whereas the formal evaluation includes a planned sequence and defined rules. Typically a walkthrough is informal and review and inspection are formal.
- Spontaneous vs. planned
  Spontaneous evaluation is started by developers, when they have the feeling that an evaluation is necessary. Planned evaluation is initiated by the management.
- Moderated vs. unmoderated
  Walkthrough and reviews are unmoderated and all members are equal to each other. The inspections are moderated by assigning the members to roles with different tasks. One of the members is the moderator, who leads the meeting and tries to have a regular meeting routine.

2.3 Process quality

In order to develop software, it is important to define software infrastructure, management process and maturity model.

2.3.1 Software infrastructure

Software infrastructure contains all processes and tools that enables developers to implement in a productive way. This contains a version control system and build automation. Most of the developer teams use SVN or Team Foundation Server (TFS) 2010 as a version control system. For build automation in Java, ANT is used.

2.3.2 Management process

There are a lot of different definitions for building a high quality software product. The software development process also has an influence on the quality. In each definition, the company has to make a standard for the development life cycle. The used development process should be described in the standard and the developers get their information about the policy there. It is very important to keep it up to date and not use an outdated version. Increasing the complexity of modern software systems makes it necessary to carry out the development in the form of best practices. In this paper the linear an agile models are described. Numerous software processes exist, such as the waterfall model, the Infineon internal model named SEM-I, CMM, CMMI and SCRUM.

Waterfall model

The first description of similar development phases was made by Herbert D. Benington [Hof09]. The basic concept was used from Winston W. Royce in 1970, where he developed the waterfall model. He described this model as implementation steps to develop a large program for delivery to a customer. Eleven years later, that was further shaped by Barry W. Boehm. The waterfall model gets its name from the fact it can graphically model as a cascade of six phases. The development model is a linear model and is document driven and so the result of each phase is a finished document.

Figure 2.3 shows the following phases in the waterfall model:

Each activity must be finished before going on the next phase. There are predefined starting and ending points for each phase with clearly defined results. One advantage is that the model is easy to understand. There is no chaotic development anymore as a result of the development process being divided into several phases. It also allows you to take a step backward and correct the errors. One of the disadvantages of this model was the missing flexibility. The execution is very strict in the single phases. Analyses of the requirements are not completed at the beginning and hold for the whole project duration without change. The implementation and testing is separated and the testing is only considered at the end. It is very important to find the errors as soon as possible, and due to that testing has to be a parallel process in every phase instead of only at the end of the project.

SEM-I

The basis for the SEM-I is the procedure model of Siemens Austria (PSE) [Inf00]. The SEM-I contains six different phases, which must fulfill the specific aims. In each phase there are

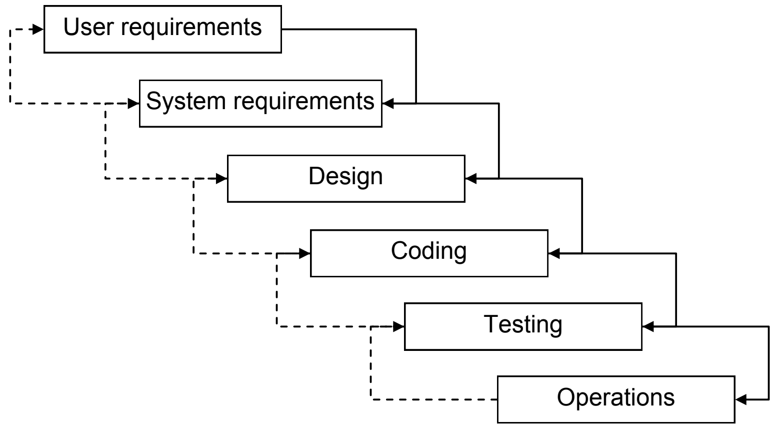

Fig. 2.3: Waterfall model

requirements, milestones and activity results involved. The phases can be repeated, because of e.g. change requests or iterative organization of the project life cycle. There are different options available:

- Waterfall sequence model
- Spiral sequence model
- Prototype sequence model
- Evolution sequence model
- Expansion stages model

The advantage of the spiral model compared to the waterfall model is the division of the lifecycle into several steps that can be evaluated. Especially in large projects with a long period of development this can reduce the risk of incorrect development.

SCRUM

Scrum is an agile approach of software development [Sch04]. It is characterized by small sprints, which last for 30 consecutive days. Everyday the developing team comes together for a 15 minutes inspection and discuss the progress, especially what has changed since the last daily Scrum.

There are three distinct roles defined:

1. Product Owner,
2. Scrum Master and the
3. Scrum Team

Product Owner represents the business and tries to improve the trust relationship to the customers. The Scrum Master mind is on coaching the team and helping the members to use the Scrum framework. The Scrum Team implements the Sprint Backlog. The team is supported by the Scrum Master and the Project Owner. It is self organizing, so that each person can

contribute in whatever ways they best can in order to complete the work of each sprint. Now the work flow is described in more detail.

At the beginning the Product Owner has the product idea. He works on the idea as long as he gets a product vision out of it. In the end the Product Vision contains the basic idea. The next step is to discuss the product functionality and record it in a list, the so-called Product Backlog. After collecting all of the items, they get rated on their financial profit. Each Product Backlog has to be regarded by its size by the Scrum Team. The team consists of all software developers, who implement the item. They also have to evaluate the size and feasibility and discuss their solution with the assessment from the Product Owner. If the vision gets an okay from all necessary instances, the work can start. At the beginning of each Sprint, the team discusses the functionality from the backlog and associates items to a Sprint on which functionality is going to be implemented. At the end of each Sprint there has to be software, which is ready to be delivered to the customer. In the Scrum terminology this is called potential shippable business functionality or usable software.

Figure 2.4 shows a part of the Scrum workflow. It shows how the functionality is developed step-by-step. There is always a check on feasibility as well as on on time management.

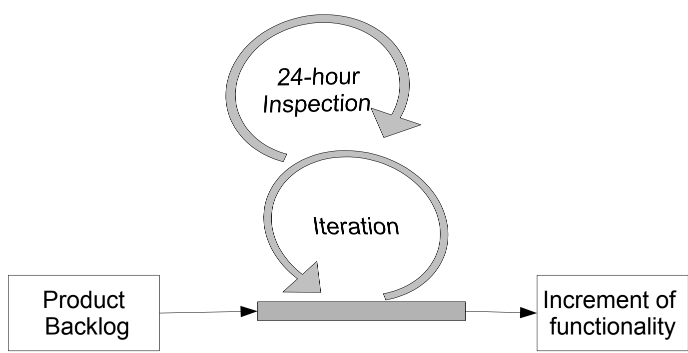

Fig. 2.4: Scrum skeleton

At first, a Sprint Plan Meeting 1 takes place, where the Product Owner, the Team, the management and the customer come together to talk about the Sprint Goal. All of the requirements to implement and the functionality are discussed in detail. That way, at the end of the meeting, everybody has a clear idea about the desired work. The Items for the next Sprint are gathered together in the Selected Product Backlog. At the Sprint Planning Meeting 2, the team has a chance to discuss the requirements and the result of this meeting is the Sprint Backlog. Then the implementation phase from the team can start. Every day, they come together and discuss what happened yesterday, which problems occurred and which tasks are to be implemented for today. This conversation should be at least 15 minutes and is moderated by the Scrum Master, who tries to help the team achieve the aim. To show the progress of the project, a Scrum board is used. On this board, the actual progress is shown, including which story cards are still in progress. Furthermore, each developer can pick a task and put their name on it. The story card

describes the functionality and contains estimation about the complexity of this task and how much time and effort is spent on it.

2.3.3 Maturity model

The maturity model is for rating and optimizing the software process. Capability Maturity Model (CMM) and Capability Maturity Model Integration (CMMI) belong to this categorization.

Capability Maturity Model (CMM)

The Capability Maturity Model (CMM) [Hof09] [BLL04] was developed by the Software Engineering Institute (SEI). The first version was published in 1991. The CMM is a model for judging the maturity of a software process in an organization. The main goal is to identify the key practices and increase the maturity of these processes. In the year 2000 CMM was replaced by the Capability Maturity Model Integration (CMMI). A maturity level indicates process capability and contains key process areas (KPA).

Each maturity level depends on capabilities organizations can reach with the usage of a software process. The key process areas are expected to achieve goals and are organized by common features. The goals of the KPAs are reached by following the key practices. In the same way, a maturity level is approached by meeting all of the goals of all of the KPAs at that level. The common features of key practices indicate whether the implementation of a KPA is effective, repeatable or lasting. For this model, there are five maturity levels. The first level is the default level, which is set initially for the organization. Each level is based on previous levels. Reaching the third level indicates the fulfilling of the prerequisite first, second and third level.

- Initial
  The initial phase is the starting point of the process and it is characterized as ad hoc. The organization typically does not provide a stable environment for developing and maintaining software. In this level few processes are defined.

- Repeatable
  Basic project management processes are established to track costs, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. The key activities in level 2 are the software configuration management, quality assurance, project tracking, oversight, project planning and requirements management. Common understanding between customers and developers and establish review processes are important for the management of requirements. The project planning consists of creating and following a reasonable plan for realizing and managing projects. It also contains sourcing of objects. The artifacts for the configuration management have to be managed and the changes must be documented.

- Defined
  At this level the documentation plays a key role. The software process is documented, standardized and integrated into a standard software process for the organization. Metrics and measures will be reviewed to take account of how much time will be spend on test activities. Testing efficiency, inspection rate for deliverables and variance between actual and planned management effort are measured. The KPAs are training units for the employees. Furthermore, the integrated software management should suit every project need.

There is an organizational acceptance of standard processes. Inspections and walkthroughs are performed to detect errors at an early stage.

- Managed
  Detailed measures of the process and product quality are collected and analyzed. It is possible to measure the qualities with metrics. Therefore they play a key role at this level.
- Optimizing
  It consists of improving the process continuously by the feedback. This includes defect prevention and identifying useful techniques, tools and methodologies. It is also essential to improve the organization‘s process to increase the quality. The KPA‘s are defect prevention, improving the process to have a positive impact on quality, productivity and development time.

Capability Maturity Model Integration (CMMI)

The CMM was successfully adopted in the software field, but it was not an optimal process for development, due to missing flexibility. The successor of the CMM is the Capability Maturity Model Integration. The CMMI is now used to have an eye on the process improvement throughout an organization. The difference between the CMM and the CMMI is that the CMMI is not only developed for the software area. The main core is the CMMI-SE/SW, which covers the software engineering partition. The maturity model extends the software engineering field with the integrated product and process development (IPD) and supplier sourcing (SS). It uses the most successful elements from CMM. The following information is included in the CMMI:

- Capability maturity model for software (CMM-SW)
- Integrated product development capability maturity model (IPD-CMM)
- Capability maturity model for system engineering (CMM-SE)
- Capability maturity model for supplier sourcing (CMM-SS)

The disadvantage of the CMM was that in this process, the phases were strict and fixed. The CMMI provides a continuous representation, which eliminates the problem with the overly strict and fixed phases. Both models cover the same content.

2.4 Design Patterns

Design patterns make it easier to reuse successful designs and architectures. For new developers it is easier to understand the new software by using the patterns. The used pattern vocabulary gives the new developer a good idea of how the project works, even by hearing the pattern's name. In [GHJ+05] Christopher Alexander explained that each pattern has a problem description, which occurs over and over again in the environment. Then, he describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
A pattern has four essential elements:

- Pattern name
  The name expresses the problem, its solution and consequences in a few words.
- Problem
  The problem describes when to apply the pattern. It explains the problem and its context.
- Solution
  The solution is characterized by the elements that make up the design, their relationships, responsibilities and collaborations. It does not describe a particular concrete design or implementation, because the pattern is a template and it can be applied in many different situations.
- Consequences
  The consequences are the results and trade-offs of applying the pattern. They are important for evaluating design alternatives and for understanding the benefits of applying the pattern. That includes its impact on a system‘s flexibility, extensibility and portability.

Design patterns are typically grouped in creational, structural and behavioral patterns. As an example and demonstration for using patterns two examples are picked out. As a creational pattern, Singleton is described and as a behavioral pattern, the Observer pattern is demonstrated [GHJ+05].

2.4.1 Singleton

The Singleton pattern is a way to create only one instance of an object with a global point of access to it [GHJ+05].

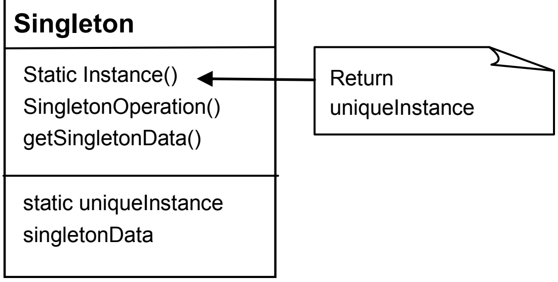

Fig. 2.5: Singleton Pattern

The consequences of using this pattern:

- Controlled access to sole instance. Because the Singelton class encapsulates its sole instance, it can have strict control over how and when clients request it.
- Reduced name space. It is an improvement over global variables.
- Permits refinement of operations and representations. The Singleton class may be subclassed, and it is simple to configure an application with an instance of this extended class at run-time.
- Permits a variable number of instances. It is easy to change your mind and allow more than one instance of the Singelton class.
- More flexible than class operations.

The implementation looks as follows:

```
public class Singleton {
  private static Singleton uniqueInstance;

  private static Singleton getInstance () {
      if (uniqueInstance == null) {
          synchornized (Singleton.class) {
              if (uniqueInstance == null) {
                  uniqueInstance = new Singleton();
              }
          }
      }
      return uniqueInstance;
    }
}
```

Src. 2.1: Implementation of the Singleton pattern

2.4.2 Observer Pattern

The Observer pattern is also known as Publish-Subscribe Pattern [GHJ+05] and it belongs to the behavioral patterns. It defines a one-to-many dependency among objects, so that when one object changes its state, all of its dependents are notified and updated automatically. If the data is visualized by two or more windows and you do not want to care of changing the data at different places. To keep the objects consistency, the Observer pattern should be used. Another argument is if you do not want to keep the objects coupled tightly.

The participants of the pattern are:

- Subject
  The Subject knows its observers. Any number of Observer objects may observe a Subject. It provides an interface for attaching and detaching Observer objects.
- Observer
  The Observer defines an updating interface for objects, that should be notified if a subject changes.
- ConcreteSubject
  The ConcreteSubject is the "real" subject, which stores the states and sends a notification to its observers, when its state changes.

- ConcreteObserver
  The ConcreteObserver is referenced as a ConcreteSubject object. It stores the state, which should stay consistent with the subjects state. It implements the Observer updating interface.

Figure 2.6 shows the interaction between the participants.

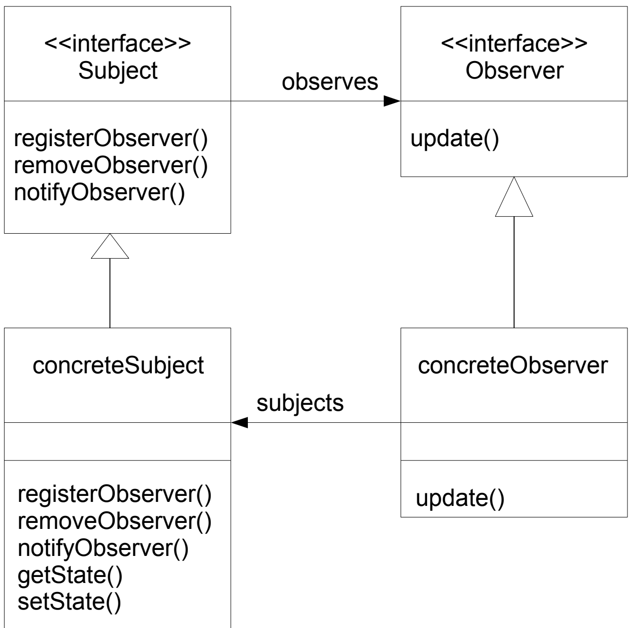

Fig. 2.6: Observer Pattern

The consequences of using the Observer patterns are:

- Abstract coupling between Subject and Observer. The Subject only knows that it has a list of Observers, but it does not know the concrete class of any Observer.
- Support for m:n communication models. The notification is broadcast automatically to all interested objects subscribed. The subject it not interested in how many objects exit, it is only responsible for notifying the Observer. At any time, new Observers can be added or removed. The Observer has to handle or ignore the notifications.
- Unexpected updates work in a way that allows Observers to be blind to the ultimate cost of changing the subject. This can lead to spurious updates, which can be hard to track down.

Implementation of the Observer pattern in Java and the following exercise to build a weather station [FFB+04].

Subject interface and its concreteSubject implementation:

```
package Observer;
public interface Subject {
  public void registerObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}

package Observer;
import java.util.ArrayList;
public class WeatherData implements Subject {
  private ArrayList observers;
  private float temperature;
  private float humidity;
  private float pressure;


  public WeatherData() {
    observers = new ArrayList();
  }

  public void registerObserver(Observer o) {
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    int i = observers.indexOf(o);
    if (i >=0) {
      observers.remove(i);
    }
  }

  public void notifyObservers() {
    for (int i=0; i < observers.size(); i++){
      Observer observer = (Observer)observers.get(i);
      observer.update(temperature, humidity, pressure);
    }
  }

  public void measurementsChanged() {
    notifyObservers();
  }

  public void setMeasurements(float temp, float hum, float pres){
    this.temperature=temp;
    this.humidity=hum;
    this.pressure=pres;
    measurementsChanged();
  }
}
```

Src. 2.2: Implementation of the Subject and concreteSubject

The following source code shows the implementation of the Observer and its concrete Observer named CurrentConditions.

```
package Observer;
public interface Observer {
  public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement {
```

```
  public void display();
}

public class CurrentConditionsDisplay implements Observer, DisplayElement{
  private float temperatures;
  private float humidity;
  private Subject weatherData;

  public CurrentConditionsDisplay(Subject weatherData){
    this.weatherData = weatherData;
    weatherData.registerObserver(this);
  }

  public void display() {
      System.out.println("Current conditions: " + temperatures + "F degrees and " + humidity + "\%
           humidity");
  }

  public void update(float temp, float humidity, float pressure) {
    this.temperatures=temp;
    this.humidity=humidity;
    display();
  }
}
```

Src. 2.3: Implementation of the Observer and its ConcreteObserver

The WeatherStation implements the Main and the interaction between the Observer and the Subject.

```
package Observer;
public class WeatherStation {

  public static void main (String[] args) {
    WeatherData weatherData = new WeatherData();
    CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherData);

    weatherData.setMeasurements(80, 60, 30.4f);
    weatherData.setMeasurements(82, 70, 29.2f);
    weatherData.setMeasurements(78, 90, 29.3f);
  }
}
```

Src. 2.4: Implementation of the WeatherStation

Java provides an implemented Observer pattern. The most general are the Observer interface and the Observable class in the java.util.package. The implementation is quite similar to the implemented Observer pattern above.

2.4.3 Structural Violations

Using the layer architecture is a way to structure applications in groups of subtasks in which each group of subtasks is at a particular level of abstraction [BMRS96]. The most important principle is that layers are strictly separated from each other, in the sense that no component may spread over more than one layer. If layers are not separated, this leads to structural violations.

Figure 2.7 shows an example of a layered architecture from the DND project.

The following rules are a way to structure your application, but not every step is mandatory:

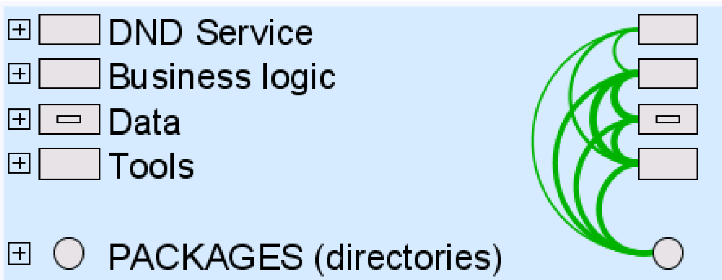

Fig. 2.7: DND layer architecture

1. Define the abstraction criterion for grouping tasks into layers.
2. Determine the number of abstraction levels according to your abstraction criterion.
3. Name layers and assign tasks to each of them.
4. Specify the services.
5. Refine the layering and iterate steps 1 to 4.
6. Specify an interface for each layer. If a layer L should be a black box for layer L+1, there has to be an interface that offers all of L's services.
7. Structure individual layers, because there is a kind of chaos inside. If it is possible, it should be broken into separate components.
8. Decouple adjacent layers so that there is only a one-way coupling.

The architectural layer pattern has the following benefits:

- It is possible to reuse layers.
- Clearly-defined levels of abstraction enable the development of standardized tasks and interfaces.
- Dependencies are kept local.

Each pattern has not only benefits, but also some liabilities as in the following:

- Lower efficiency. High-level services in the upper layers rely heavily on the lowest layer and all relevant data must be transferred through a number of intermediate layers.
- Unnecessary work. The lower layer performs excessive or duplicated work, which is not actually required by the higher layer. This has an impact on performance.
- The challenge is to find correct granularity of layers.

2.5 Testing

Testing is the process of exercising a software component using a selected set of test cases with the intent of revealing defects and evaluating quality [Bur03]. Defects occur and have a negative effect on the software quality, which is why it is so important to start testing as soon as possible in the software process. Tests are executed when the corresponding code is available, but testing activities start earlier, as soon as the specifications are available. Figure 2.8 shows the error discovery rates for different stages of tests [Bur03]. Although each application system is different, most errors are found during integration and system testing [DWR08].

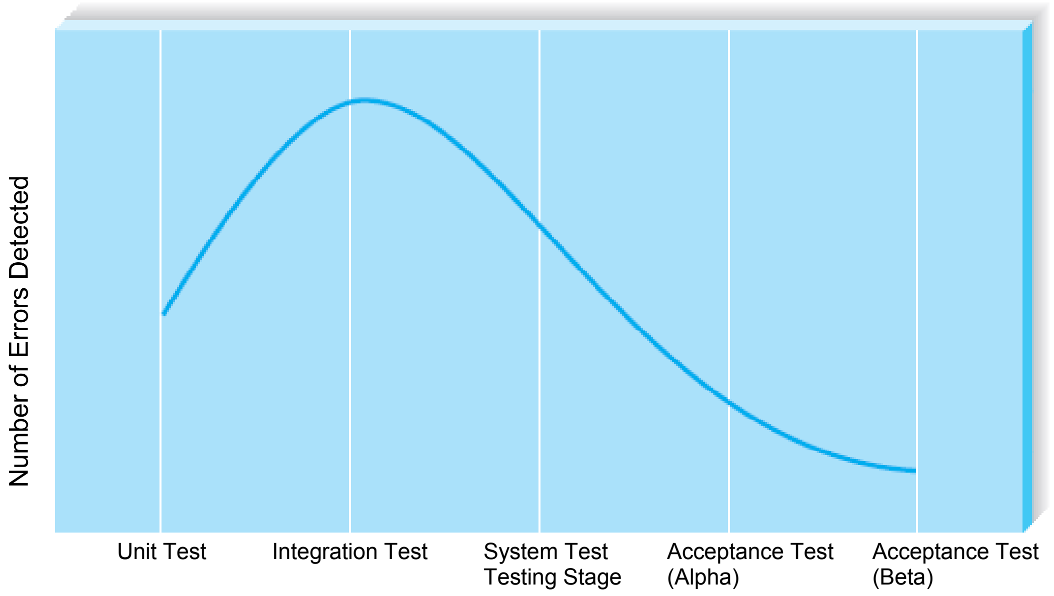

Fig. 2.8: Error discovery rates [DWR08]

In the next sections the following test methods are explained: Manual test, unit test, functional test, database unit test, smoke test, load test, system test, acceptance and regression test.

2.5.1 Planning and Monitoring

Testing is not an intuitive process; it has to be planned. Process visibility plays a key role in the development [PY07]. It is primarily for team members and management, so that they can see the actual process state. This includes how many tests pass or fail and also reviewing and evaluating the metrics result. The architects are able to check the design and can decide if code refactoring is necessary. For the team members it is beneficial to get feedback. Figure 2.9 shows what the plan of a project can look like and which techniques are used.

A combination of several testing techniques is always a good choice, due to the fact that a single test does not cover everything. A good way to design test cases is, when the implementation has not started e.g test driven development. You have to think about input and output parameters, how the method should act and what the result is. Tests are independent from source code. They can highlight inconsistencies and incompleteness in the corresponding software specification.

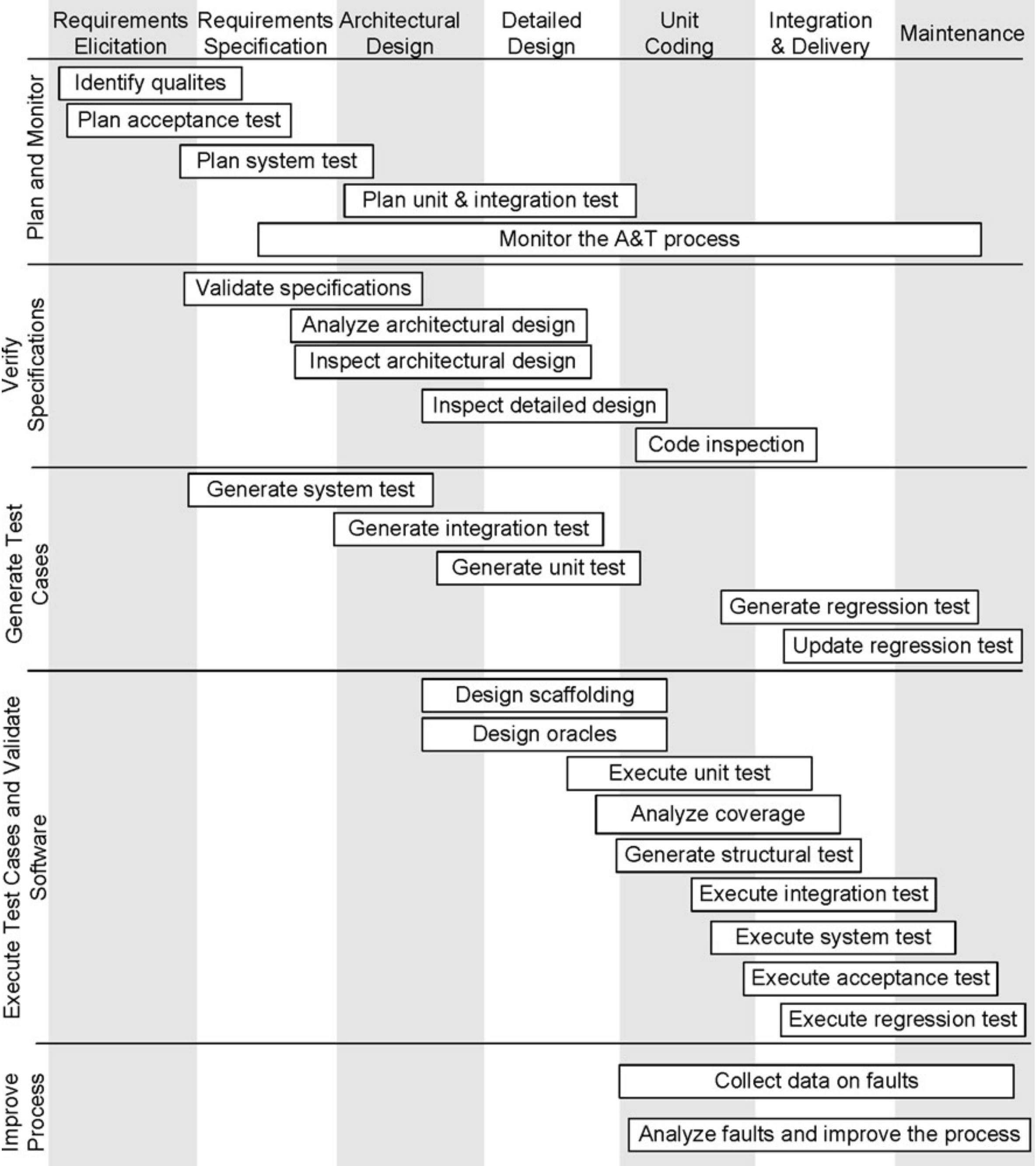

Fig. 2.9: Testing techniques through software life cycle

A test plan should answer the following questions:

- What quality activities will be carried out?
- What resources are needed and how will they be allocated?
- How will the process be monitored to maintain an adequate assessment of quality and early warning of quality and schedule problems?

2.5.2 Manual test

The oldest and the simplest type of testing is the manual test [KS10]. The tester writes test cases to validate the requirements without using any automation tool. It plays a role in the following scenarios:

- There is not enough budget for automation
- The tests are more complicated to convert into automated tests
- Not enough time to automate the tests
- Automated test would be time consuming to create and run

2.5.3 Unit test

The goal of the unit test is to examine the smallest piece of source code [KS10]. It isolates the source from the remainder of code and checks if the behavior is exactly the same as expected. Unit tests are written and run by developers as they write code. They are typically automated as a suite to be run on code check-in (continuous integration) and used by testers as part of integration tests and regression tests. It is crucial to run these tests and catch the defects in an early stage of the software development cycle.

A unit test is a functional class method test, which calls a method with the parameters and finally compares the actual results with the expected results.

In Visual Studio 2010 it is possible to generate unit test methods and classes during the implementation of the class.

```
namespace TestLibrary {

  public class Class1 {

      public double CalculateTotalPrice (double quantity) {
        double totalPrice;
        double unitPrice;
        unitPrice = 16.0; //Todo get unit price. For test it is hard coded.

        totalPrice = unitPrice * quantity;
        return totalPrice;
      }

      public void GetTotalPrice() {
        int qty = 5;
        double totalPrice = CalculateTotalPrice(qty);
        Console.WriteLine("Total Price: " + totalPrice);
      }
    }
  }
```

Src. 2.5: Example source

In Visual Studio you can generate a unit test and modify the source code, shown in source 2.6 [KS10]

```
namespace TestProject1 {

    [TestClass]
     public class Class1Test {

        private TestContext testContextInstance;

        public TestContext TestContext {
            get {
              return testContextInstance;
            }
            set {
              testContextInstance = value;
            }
        }

        [TestMethod]
        public void TestMethod1() {
          Class1 target = new Class1();
          double quantity = 0F;
          double expected = 0F;
          double actual;
          actual = target.CalculateTotalPrice(quantity);
          Assert.Inconclusive("Verify the correctness of this test method.");
        }

        [TestMethod]
        public void GetTotalPriceTest() {
          Class1 target = new Class1();
          target.GetTotalPrice();
          Assert.Inconclusive("A method that does not return a value cannot be verified.");
        }
      }
    }
```

Src. 2.6: Unit test example

For java projects, if you need a lot of database connections for unit tests, it is very time consuming. First, you have to set up test data, then perform the test and finally a rollback. To get rid of that, it is possible to use Mockito [Fab13] or PowerMockito [HK13] open-source test framework. It is possible to simulate the behavior of certain objects. For example you have a Product interface. Behind the interface, there are several implementations, which contain a lot of logic, dependencies to other classes and connections to the database. For your test, it is only important to get a Product object. In source 2.7 a Product object is mocked. For simulating the behavior when().thenReturn() is used.

```
@RunWith(MockitoJUnitRunner.class)
public class MockTest {

import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

@Test
public void testExampleTest() {

//mock a concrete Object
```

```
Product testProduct = mock(Product.class);

//stubbing
when(testProduct.getPrice()).thenReturn(42);
when(testProduct.buyProduct(anyInt())).thenReturn(true);
//***** perform your test ******

RealClassToTest realClass = new RealClassToTest();
realClass.byProduct(1);

//***** returns a list of products *****
List<Product> productList = realClass.getAllBoughtProducts();
Product resultingProduct = productList.get(0);

//***** verify results *****
assertEquals(resultingProduct.getPrice(), 42);
}
```

Src. 2.7: mock objects

2.5.4 Functional test

Functional testing is a type of black box testing. It [PY07] derives test cases from the program specification. It is based on program specifications and not on the internals of source code. Functional test case design should begin as part of the specification requirements process, and continue through each level of design and interface specification. It is the only test design technique with such a wide and early applicability. Moreover, functional testing is effective in finding some classes of fault that typically elude so-called white-box or glass-box techniques of structural or fault-based testing.

2.5.5 Smoke test

Smoke test [Lev11] is a kind of acceptance testing. It is made as a first test after modifications to ensure that it will not fail catastrophically.

Load test

Load testing [Lev11] is the process that subjects the system under testing to a work level approaching the limits of its design specification. Load testing is usually performed in a controlled lab environment where accurate measurements can be taken under repeatable conditions. You can also perform load testing in the field to obtain a qualitative assessment of system performance in the "real world".

2.5.6 System test

System test examines the entire system to ensure that the requirements have been met. This test includes the functional and also the non functional requirements. The test should be performed n an environment that closely reflects the physical environment that the production system runs in [Lev11].

In Visual Studio 2010 this can be done with the Lab Management and by executing automated test causes.

2.5.7 Acceptance test

Acceptance testing [Lev11] is a functional trial performed on a completed increment of functional software before it is accepted and deemed ready for release to the market or delivery to the end user. The acceptance testing process is designed to replicate the anticipated real-life use of the product to ensure that what the consumers or end users receive is fully functional and meets their needs and expectations. In traditional predictive processes, this is usually at the end of the product development cycle, whereas with agile development, process acceptance testing is done at the conclusion of each development iteration. Figure 2.10 shows you that acceptance testing is the pinnacle of building quality [Lev11].

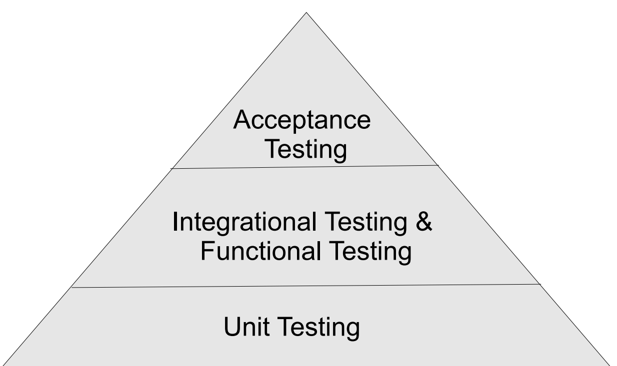

Fig. 2.10: Testing pyramid

2.5.8 Regression test

After changes in the project a Regression test should be made, where all the test cases, designed for previous versions, are executed. Even simple modifications of the data structures can have an impact of the execution and so test cases may not be executable without corresponding changes.

In Visual Studio 2010 Test Impact Analysis was built for performing regression tests.

2.6 Chapter summary

After a closer look, the simple expression ”software quality” seems very complex. Software quality is considered good, if all the requirements are met. Furthermore, it also includes when software is quite error-free. For error-free software testing is necessary. For testing, source code has to be understandable. It has to be kept in mind, that code reading is a big part of the developers daily job. It is necessary to use some coding guidelines and style templates to save time in understanding the source code. There is a description in the ISO\IEC 9126 for the criteria of software quality. These criteria are divided into functionality, reliability, usability, efficiency and maintainability. Then a definition for software quality for the project has to be defined.

Now, the following parts for software quality are found:

- Software quality criteria
- Testing
- Clean code development
- Coding guidelines

Obviously, the software developer has a substantial influence on implementation of the requirements, if they are well-defined. Also, the software development process has an influence on the resulting quality.

3 Metrics

In the domain of software development, quality assurance takes considerable effort. It is important to identify those pieces of software which are most likely to fail and therefore requires most of the developer‘s attention. Metrics are widely used in software evaluation tools, because using them is a way to reflect software quality. By definition a metric is a function whose input is software data and whose output is a single numerical value. The result can be interpreted for a given attribute how it affects quality [KMB04].

If the quality of a system suffers and needs to be re-engineered, they can be used for detecting problems. Quality criteria are evaluated, but the interpretation seems to be though in practice. Nowadays there are many metric suits available, which include several, different kinds of metrics. The main challenge is to find a suitable set of metrics as well as the correct interpretation of calculated results. The developer uses metric suites in order to gain the insight required for understanding and evaluating the structure and quality of a system.

"You can‘t manage, what you can‘t control and you can‘t control what you can‘t measure."
Tom DeMarco [DeM86]

Metrics only have a few requirements; they have to be simple and easy to understand for developers. Nobody wants to use metrics, which are really difficult to interpret. The following three statements are examples of how metrics can be interpreted or misunderstood [Wil11], [Pei11]:

1. Statement 1: The higher the error density, the worse the source code is.
   To rate a project only on the fact of error density is not decisive. There is a difference if you evaluate a project_A with 500 lines of code and 25 errors, compared to a project_B with 5000 lines of code and 30 errors. The density of project_A is higher than in project_B, but it does not say anything about the quality of the source code.
2. Statement 2: The higher the test automation, the more effective the test process is.
   For this statement the cost-benefit ratio has to be considered. You have to check which test cases paid off for test automation.
3. Statement 3: The higher the value of the measured cyclomatic complexity metric is, the worse the maintenance is.
   Maintenance becomes a challenge if too many cycles are in the project. In a worst case scenario, one change in source code could lead to several problems on other different positions, which are included in the cycle.

Figure 3.1 shows several states of development essentials for an software evaluation [Pei11]. The states of development should be declared by the company. The stakeholder and the goals

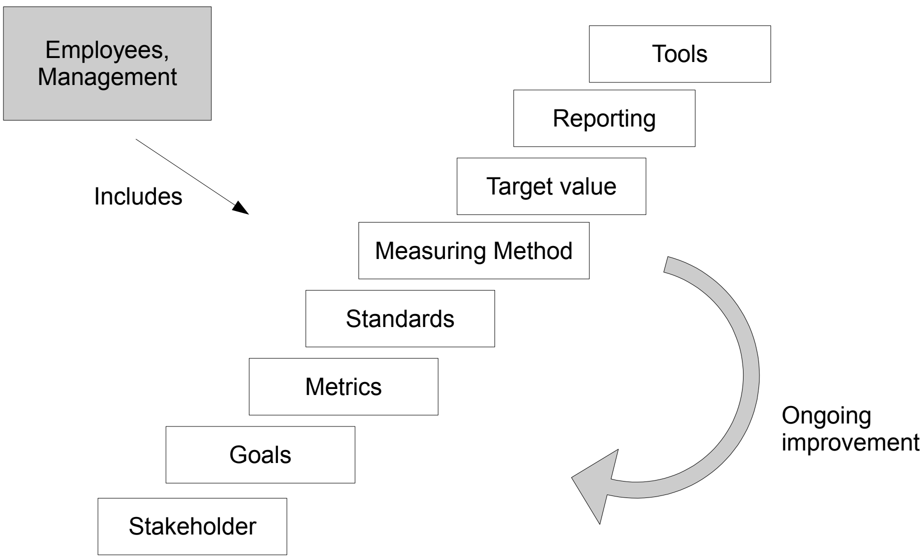

Fig. 3.1: States of development

have to be identified. The next aspect is to declare a metric suite, which fits the requirements. For developing software it is necessary to define standards, guidelines and measuring methods. Furthermore, it is important to specify target values where the limits are given for good or bad metric results. All the measures should be reported and included in milestones. Before starting the metric calculation, there are some questions for software architects and developers to answer. These questions are for a self-evaluation if everything is clear in the development. The following questions arose while the interviews about software quality took place at the Infineon.

Questions for **software architect**:

- Is the actual project state the status-quo or the desired state?
- Do architectural violations appear in the project and which are they?
- Can the project be extended easily?
- Are there some quality standards given for the project and are these standards monitored?

Questions for **software developer**:

- Does each software developer know about the system structure?
- What will happen if something in the source code is changed, because of dependencies?
- Are there some coding guidelines or quality standards used?

With metrics you can measure your software and you are able to calculate a trend by comparing results. When trying to improve a process, the company must start with a desire to make a change. If a company does not want to change, gathering metrics is not useful and it adds a

further waste of time to the already wasted time [Lev11]. There are some pros and cons of using metrics. It is very useful to check yourself and have an eye on the source code. Metrics are an indicator of some vulnerability in source code and also a sign to improve the development process. The greatest challenge is to interpret these metrics, which is quite difficult. At the end of this chapter the ranges of different metric suites are specified.

The best way is to find a minimal set of metrics, which covers problems of the software project and measures internal properties of an object oriented product. Several studies found that there is a correlation between software metrics and quality [BWDP00]. For the evaluation of the IT-projects (Java, C# and C++) the object oriented metrics are relevant.

The design characteristics for measuring are [CS09]:

1. Abstraction
2. Encapsulation
3. Information hiding
4. Coupling
5. Cohesion
6. Inheritance
7. Polymorphism

In the following sections these object-oriented principles are explained. Finally common metrics like Halstead, Maintainability Index, McCabe and the Chidamber and Kemerer (CK) metrics are described.

3.1 Object oriented principles

3.1.1 Abstraction, encapsulation and information hiding

Abstraction is a way to simplify the complexity of a program [Doo11]. The basic principle is to hide complex details and keep things simple. By using this principle you can say what a program does, without saying how it works in detail. It is a process of generalization where the details and all the inessential information are eliminated. **Encapsulation** means to bundle a group of services defined by their data and behaviors together as a module and keep them together. This group should be coherent and all elements should clearly belong together. An interface is provided to access the services and data in this module. As a result you have high cohesion. **Information hiding** is the concept of isolating information. The class only gets to know what it needs to know in order to interact with other classes. It provides selective information access to classes.

For better understanding the differences between encapsulation and information hiding, the following source code is an example of representing data of a point on the Cartesian plane [Mar08]:

```
public class Point
{
  //encapsulation
  public double x;
```

```
  public double y;
}

public interface Point
{
  //information hiding
  double GetX();
  double GetY();
  void SetCartesian (double x, double y);
  double GetR();
  double GetTheta();
  void SetPolar (double r, double theta);
}
```

Src. 3.1: Encapsulation and information hiding

3.1.2 Coupling

The definition for coupling [YK10] is to measure the degree of interaction between two software components. A good software system should have a low coupling value. If there are too many dependencies between components, the software is hard to understand, maintain and reuse. Coupling is one of the important design principals in object-oriented software development, because it provides the sharing property. On the other hand, it makes the software more error-prone and unreliable. Reliability is a crucial quality criterion and contains fault prevention, fault detection, fault removal and reliability maximization. Coupling itself is also closely related to complexity.

There are the following types of coupling:

- Single coupling
  Single coupling provides access of data from one class to another class. Sharing is only between two classes.
- Multiple coupling
  In multiple coupling one class has access to the data of two or more classes.
- Hierarchical coupling
  A single class is shared by more than one class.
- Multilevel coupling
  In this kind of coupling the output from one class becomes the input of the next class.
- Hybrid coupling
  In hybrid coupling different kinds of coupling are combined.
- Cyclic coupling
  The sharing of data and methods form cycles among classes.

To reduce the coupling value metrics like cyclomatic complexity or Weighted Methods per Class (WMC) are used. But WMC does not measure the object oriented aspect. Some problems with cycles are sharing data and methods several times. This increases complexity and contains functional dependencies, which cause retesting and rework. One way to improve coupling is refactoring. Code refactoring is the way of restructuring an existing body of code without changing the behavior of the software. Refactoring also leads to a higher level of complexity, because re-engineering for only one part of the software is applied. A class that is highly sensitive

Name	graphical representation
Single coupling	A — B
Multiple coupling	A1, A2, A3 — A
Hierarchical coupling	B — B1, B2, B3
Multilevel coupling	A — B — C
Hybrid coupling	A — B, D — C

Tab. 3.1: Different kinds of coupling

means having more chances for errors, which need to be re-engineered. The method used very often should be kept in a separate class, e.g. utility program.

3.1.3 Cohesion

Cohesion is the expressed when a method shares at least one single attribute [PR10]. If a large sized class only has a low level of cohesion, it is a symptom off bad software design. The choice of the right cohesion metric is always critical. There are five categories of cohesion metrics.

- Disjoint component-based metrics (LCOM1, LCOM3, LCOM4)
  Disjoint component-based metrics count the number of disjointed sets of methods or attributes in a given class.
- Pairwise connection-based metrics (LCOM2, RLCOM, CR, TCC, LCC)
  Pairwise connection-based metrics count the number of connected or disjointed method pairs.
- Connection magnitude-based metrics (LCOM, SCOM, LCOM*)
  Connection magnitude-based metrics count the accessing methods per attribute and indirectly find a sharing index in terms of the count.
- Decomposition-based metrics (CBMC)
  Recursive decomposition is used for counting the decomposition-based metrics. They are generated by removal of pivotal elements that keep classes connected.
- Interface-based metrics (CAMC, CM)
  Interface-based metrics are based on information gathered from method signatures.

The lack of Cohesion (LCOM), especially the LCOM1 is the number of disjoint sets formed by the intersection of n access sets of instance variables, corresponding to n methods. LCOM2 is a highly controversial cohesion metric. If there are not any shared instances, the computed value is zero, instead of negative. The Cohesion Ratio (CR) is the number of method pairs sharing instance variables in ratio to the total number of method pairs. LCOM, TCC and LCC include the constructor, destructor and accessors in cohesion computation. This should not influence the cohesion value of a class. The new metric, which excludes these factors, is Cohesion Based on Member Connectivity (CBMC). Metrics from connection magnitude-based category capture cohesion more accurately compared to metrics from the other categories. Disconnected component-based metrics capture variations in class cohesion and they detect disparate classes and also count the number of disconnected components.

3.1.4 Inheritance

Inheritance [SS07] [Bre07] is the most powerful feature of object-oriented design. Inheritance means that one class inherits the characteristics like behavior or attributes of another class. The inheritance metrics influence the complexity and maintenance time. Through structuring, you get a classification hierarchy and classes are organized in a tree structure. The tree depth is calculated from the root to the class node and measures ancestor classes. The deeper the tree hierarchy is, the more the complexity increases and therefore, testing gets more difficult. It is recommended that the maximum length of the tree does not exceed 7. There are two complementary roles of inheritance in an object-oriented application:

- Specialization: Extending the functionality of an existing class.
- Generalization: Sharing commonality between two or more classes

Some classes only act as a holder, because they do not represent a concrete type. Those classes are abstract classes. Instead of creating instances, an abstract class is a superclass for other classes and contains abstract methods. It is suggested to have at least 15% of abstract classes within the project. In an object-oriented system, there is a one to many relationship between a method and its implementation. There are three ways in which a derived class can implement polymorphic methods:

- A derived class can implement a polymorphic method by inheriting it unchanged.
- Replacing it with different implementation (overriding).
- By extending it, adding to the existing implementation.

3.1.5 Polymorphism

Polymorphism [CW85] allows programs to process objects that share the same superclass in the hierarchy as if they are all objects of the superclass. It is the ability to create an object that has more than one form. Because a single name can represent different code, that name can express many different behaviors. It allows the developer to extend the system easily. There are four types of polymorphism:

- Inclusion Polymorphism
- Parametric Polymorphism
- Overriding
- Overloading

Inclusion polymorphism is a way to redefine a method in classes that are inherited from a base class. You can therefore call on an object's method without having to know its intrinsic type. Parametric polymorphism is the ability to define several functions using the same name, although using different parameters and the correct method is automatically selected. An override is a type of function which occurs in classes that inherit from another class. It replaces a function inherited from the base class. The method does different things depending on which class was used to instantiate an object. With overloading, you can declare the same method multiple times. This only differs in the number of parameters.

3.2 Common metrics

In this section metrics [CS09], [KB11] are described. Popular metric suites are Halstead‘s Complexity Measure, Maintainability Index, McCabe‘s Cyclomatic Complexity and Chidamber & Kemerer (CK).

Halstead [SSB10] was the first one interested in code complexity. He took over the term complexity of the communication theory from Shannon, who defined the complexity from messages as the ratio from many different signs to the length of the message. To use this definition for developing software, the source code is complex, if there are many different operands and operation in ratio to the sum of commands. Operators and operands are defined by their relationship

to each other. An operator carries out an action and an operand participates in such an action. An example is an operator that carries out an operation using zero or more operands. An operand may participate in an interaction with zero or more operands. The metrics can operate on method, class and package level. The Halstead Effort is used in the Maintainability index calculation.

Maintainability Index (MI) [SSB10] was designed at the university of Idaho in 1991 by Oman and Hagemeister. The aim of the MI was to determine how easy it would be to maintain the source code. It combines the Halstead Effort (E) per module, McCabe cyclomatic complexity (VG) and Lines of Code (LOC and the commented lines of code CMT). The formula is $171 - 3,42 * \ln(E) - 0,23 * VG - 16,2 * \ln(LOC) + 0,99 * CMT$.

McCabe metrics also called cyclomatic complexity [BD08], were developed by Thomas J. McCabe in 1976. He saw his source code as a directed graph with nodes and edges. McCabe metrics are used to indicate the complexity of a program. The cyclomatic complexity is computed using the control flow graph where every independent path is counted. McCabe metrics are calculated by counting the number of edges minus knots plus not connected components two times. The complexity increases with the number of branches through a program. From practical experience those modules with a cyclomatic complexity higher than 10 tend to be prone to higher defect rates. The formula for the cyclomatic complexity is $V(s) = e - n + 2p$.

The Mood (Metrics for Object-Oriented Design) [Bre07][Nag04] suite contains metrics for measuring object oriented aspects such as method and attribute inheritance, polymorphism, coupling, hiding factor and attribute hiding factor. These metrics operate on system level and consist of the following metrics:

- **MHF - Method Hiding Factor** - the number of visible methods
- **AHF - Attribute Hiding Factor** - the number of visible attributes
- **MIF - Method Inheritance Factor** - the ratio of the sum of inherited methods to the total of number of methods
- **AIF - Attribute Inheritance Factor** - the ratio of the sum of inherited attributes to the total number of attributes
- **PF - Polymorphism Factor** - the degree of method overriding in the class inheritance tree
- **Coupling Factor** - the actual number of couplings among classes in relation to the maximum number of possible couplings

The Chidamber & Kemerer (CK) metric suite [CK94] originally contains the six following metrics: WMC, DIT, NOC, RFC, CBO, RFC and LCOM. They are the "de-facto" standard for measuring properties of classes and objects. These metrics measure different properties and should be independent. WMC, DIT and NOC are for identification of classes. WMC, RFC, LCOM are for the semantics of classes and RFC and CBO is for the relationships between classes.

- **LOC - Lines Of Code** - Counts the number of lines in a software program.
- **RLOC - Relevant Lines Of Code** - Only the relevant lines of codes are counted while comments are excluded.

- **WMC - Weighted Methods per Class** - WMC is the sum of methods in a class. The complexity of each class is its sum of the methods. The larger the number of methods in a class, the greater the impact of their children is. Children will inherit all the methods defined in class. The reuse of classes with too many methods is difficult.
- **DIT - Depth of Inheritance Tree** - DIT defines the longest path from the class to the root hierarchy. If the path is too large, it gets more and more complex to predict its behavior and there is a higher probability of errors in the source code. If it is too low, that implies less complexity but also less code reuse through inheritance.
- **NOC - Number of Children** - NOC is the number of direct descendants of a class. This metric measures the scope of the influence of a class on its subclasses due to inheritance. A class with a large number of children requires more testing effort.
- **RFC - Response For a Class** - This metric counts all the number of different methods that can be executed when an object of that class receives a message. Both, the direct and indirect calls are counted. The larger the RFC value, the greater the complexity of the class is.
- **CBO - Coupling Between Objects** - This metric counts the number of other classes which are coupled to the current class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types and exceptions. If the classes depend on a lot of other classes, the maintenance becomes a challenge. The more independent a class, the easier to reuse and maintain it is.
- **LCOM - Lack of Cohesion of Methods** - This metric shows how closely the local methods of a class are related to each other. High cohesion implies simplicity and high reusability.
- **LCC - Loose class Cohesion** - It measures the percentage of pairs of public methods to the class that uses common attributes. A high value of LCC is desirable.

Some considerations and ranges of CK metrics [Mac13] [GGM11]:

1. **DIT range(5 +/-1) - find the right balance**. If the DIT has a high value it means that the design complexity increases, while if it is too low you cannot reuse the code and use the inheritance principle. In the case of e.g. Java, how should the interfaces be handled?
2. **NOC range (9 +/-3)**. The greater the NOC, the greater the reuse is. That means, inheritance is also a form of reuse. By increasing the NOC you could make a lot of mess in the design, so testing also has to increase. Therefore reduce the number of children.
3. **CBO range(38+/-12) - low values are good**. If the CBO is high, it means that your source code becomes more difficult to maintain. The more links between the classes, the more complex it is and as a result, testing becomes more difficult. The calculation is only in one direction, because if A references B and B references A, it is only counted once. The CBO should be as low as possible.
4. **LCOM value should be low**, because it indicates cohesiveness. A lack of cohesion implies that you should split up the classes into more subclasses. For calculating LCOM you have to know when the LCOM is high or low. This depends on the amount of method pairs. With three methods, you can have three pairs, with four methods, you can still

have six pairs. You have to know the number of method pairs to decide if the LCOM is high or low.

5. **WMC range(224+/-111)- a little confusing**. The larger the number of methods, the more they have a potential impact on their children because they inherit all methods defined in the class. It is a predictor of how much time and effort is required for testing.
6. **RFC range(88+/-31) - low value**. If the value of RFC is high, it is a very useful indicator of potential problems. If you have a large number of methods which are invoked in response to a message, the testing becomes complicated and challenging. It also increases the complexity of the class.

Figure 3.2 from [LMD10] shows an Overview Pyramid, which covers e thmain aspects of an object-oriented system by quantifying complexity, coupling and usage of inheritance. The three aspects are closely related and mutually influence each other. While size, complexity and coupling characterize every software system, inheritance is specific to object-oriented software and combines coupling, size and complexity.

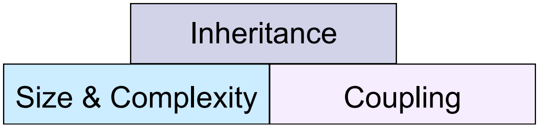

Fig. 3.2: Overview pyramid

The basic idea behind the overview pyramid is having the most significant measurements together. Every engineer can see, interpret and get an impression of the system.

The following metrics are the most important metrics and should be checked regularly:

- DIT
- NOC
- RFC
- CBO

Combine these metrics with LOC per method or class, cyclomatic complexity, testing and code coverage. This should improve your software quality and assure that you meet the quality standard.

3.3 Software quality evaluation

One aspect of this thesis was the creation of a questionnaire and an interview with the developers of the Infineon IT-department. The goal of this evaluation was to find out which problems the employees have during software development and which aspects could improve the actual situation.

3.3.1 Evaluation of the questionnaires

The following answers are results of the questionnaire from 14 respondents of the Infineon IT-department. The respondents include developers, project leaders and managers. The questionnaire consists of 20 questions, which are structured into common questions, questions about the developer team, team size, code quality, about testing and which quality assurance mechanism are used. Finally it contains questions about which project should be evaluated and where the main problems of software development within the IT-department were.

Common questions

1. Which programming languages are used in the projects?
   The major popular programming languages in use are: C#, C++, VB6, JavaScript and Java. The main focus is to improve the software quality on C#, C++ and Java.
2. How many years of programming experience do you have? How many programming languages do you know?
   In the IT-department the average programming experience is about 13 years. The language knowledges is comprised of all kinds of object-oriented languages like Java and C# and other programming languages like Cobol, Fortran, Assembler and so on.
3. What does software quality mean to you?
   The respondents define software quality with understandability, testability, simplicity of the source code and documentation.
4. The respondents had to find a ranking for the software quality criteria. On the first rank is the most important criterion, and on the 6th rank is the most dispensable criterion.
   a) functionality
   b) reliability
   c) performance
   d) maintainability
   e) usability
   f) reusability
5. How often do you analyze the software quality and the system?
   Only 5 out of 14 people analysis their software system approximately 2 times a year.

Questions about the team

6. How many people are in the development team?
   The average number of people working in a team is about five.
7. Are there any qualifications concerning the team, e.g. education, workshops, certificates?
   Half of the respondents, also including their team members, who do not take part of the interview, have graduated from a university or Fachhochschule (university of applied sciences).
8. Are there many changes within the team (fluctuations)?
   Most of the team members are stable and work for 4.5 years in the company.

9. How many experiences, concerning projects, does each team member have?
   85% indicate, that they have proper team work experiences.

Questions about code quality

10. How is quality assurance being applied?
    Quality assurance is mainly done with testing and code reviewing. Half of the respondents use coding guidelines from Microsoft, the internal guidelines from the IT-department or CCD and UNT. The IT-guideline are not completely up-to-date.
11. Which tools are used for quality assurance?
    Two of 14 people are evaluating the code quality via code reviews. The other part makes an evaluation of the source code based on their feelings. The assessment of the code quality is between middle and excellent.
12. Do you use code metrics, like LOC, complexity, Methods per class and so on?
    No metrics or cyclic detection mechanisms are used. Only naming conventions are used by 57 percent.
13. Do you avoid cycle buildings and the occurrence of cycle inheritance structures between directories, namespaces and architecture modules?
    43% take care of the architecture and possible architecture violations. 29% pay attention to the design.
14. How often do you perform redesigns and refactoring?
    Refactoring and redesigns are done by 71% percent of the respondents, but they are practiced regularly.
15. Do you review your code regularly?
    85% are using code reviews for quality assurance. From this 85%, only 39% are doing these reviews efficiently.

Questions about testing

16. Do you create a test plan?
    A test plan is created by 85% of the interviewees. As an administration tool, Hp Quality Center, Excel and a Wiki is used.
17. How are the test cases being created?
    In most cases, the developers create the test cases on their own. This could be problematic, because the developer gets blinkered in his work. The test sequence creates its own test plan and finds the right requirements for the test cases. Brainstorming is used to find the testing requirements.
18. Which test methods do you use?
    Unit tests, system tests, integration tests, regression tests and UAT are applied by all of the respondents. In some teams, a simulation is used to test the result before and after a change and also to simulate the productive load.
19. Which testing tools do you use in which environment?

    The following tools are used for testing purposes:

    - Excel

- Simulator for the productive load
- Hp Quality Center
- Test Director
- Visual Studio 2010 testing tools

20. Which quality assurance mechanisms are practiced as well?
Everybody uses a version control tool and inline documentation. If a bug occurs, a bug tracking system like Remedy, Trips or TFS is used.

3.3.2 DND

Table 3.2 shows the system overview metrics.

Metric	Value
SysFiles	546
SysLOC	104.261
SysMetadata	1791
SysMethods	4442
SysNestedClasses	460
SysPackages	98
SysPackagesNotUsed	12
SysPckgInCycles	36
SysProperties	1080
SysReferences	82909
SysRelevantLines	48458
SysSubsystems	18
SysAttributes	3768
SysCalls	16370
SysClasses	1153
SysClassInCycles	525
SysCommentblocks	5749
SysCommentedOutCodelines	2970
SysConstantAttributes	213
SysCSharpDocComments	3130
SysDelegates	33
SysDistributedNamespaces	15
SysDuplicatedCodeBlocks	90
SysEvents	62
SysIndexers	15
SysInheritances	554

Tab. 3.2: System Overview in Sotograph

The developer has an idea about the project's dimension. It has 104.261 LOC with 546 files and so it has quite an average project size.

SysMetadata: A simple description for the SysMetadata is data that describes data. In this context it is a collection of programmatic items that constitute the EXE, such as the types declared and the methods implemented. The reason for using metadata is that it allows the .NET runtime to know at run time what types will be allocated and what methods will be called. This enables the runtime to properly set up its environment to more efficiently run the application. The means by which this metadata is queried is called reflection. In fact, the .NET framework class libraries provide an entire set of reflection methods that enable any application to query another application‘s metadata.

The following metrics are more detailed and discussed in the following section:

Rules

The notation of the following metrics is read as metric name, followed by metric‘s value and the calculated value.

Attribute metrics

AttrExternalUsageRule (0) 88

88 methods from external classes read or write to the current attribute.
Example: Data.cs, attribute: Repository
Solution: A solution could be the usage of static constructors.

Package metrics

PckgAttributeOverrideRule (0) 2

The number of attributes which are hidden in derived classes. It is not common to override an attribute from the super class. In most of these cases, it is a mistake because you cannot say which attribute you use anymore. It causes more errors than it helps. Example:

```
class A { public String s = "Test";} // not static
class B extends A {
public String s = "Test23"; // not static
public void doSomething
{
  s = "23";
}

public void doSomething2
{
  ...
  deleteFrom(s); // Test? Test 23? -> Test 23
  //Which attribute did you want to delete, from class A or B? What will happen if you rename B manually
      or delete it?
}
```

Src. 3.2: Attribute override

PckgClassKnowingDerivedRule (0) 2

Identifies all classes of the current package that have any knowledge about directly or indirectly derived classes. This is considered bad style because changes to derived classes should never affect a base class.
Example: Package function
Solution: There is no general rule for what to do in this case. As a standard solution you can introduce abstract classes or use interfaces.

Measures

MethChars (1000) 15.512
These metrics shows the number of characters in the method body.
Example: OperatorGuideRuleTest.cs, GetProfileEntitiyList()
Solution: Split the method in several small methods.

MethLOC (100) 283
The number of lines of code in the method body.
Example: OperatorGuideRuleTest.cs, GetProfileEntitiyList()
Solution: Split the method in several small methods.

MethCyclomaticComplexity (10) 59
It calculates the complexity of the current method. The decision points have a big influence on the complexity. Example: OperatorGuideRuleTest.cs, GetProfileEntitiyList()
Solution: Increase the number of unit tests or refactor the source code into smaller pieces to reduce the complexity. If you have smaller pieces, it is easier to test.

MethParameters (5) 18
It counts the number of parameters within the current method. For this metric it is not the sum of the parameters that is important, but the different types. Example: Exclusion.cs, IsExclusion()

```
public bool isExlusion(Lot lot, string LotName, string Product, string Route, string Facility, string
    Wafersize, string Type, string BasicType, string ProcessClass, string BusinessSegment, string
    ProcessGroup, string epa, string setup, string layer, string operation, ExclusionData ed, bool
    exception, string Owner)
```

Src. 3.3: Too many parameters in the method

Solution: In this class there are a lot of parameters with the same type. This can cause errors, because you can not distinguish between string Product or string Route. A solution for this problem could be using objects instead of 20 string parameters.

Class metrics

ClassAttributes (20) 137
It counts the number of attributes independent of its visibility.
Example: WS_Lot.cs

ClassExcessiveMethodOverloading (3) 21
For a class, this metric counts the maximum number of times a method is overloaded.
Example: Example source code

```
class A {}
class B extends A {}
class C
{
  public void doSomething (A a) {...};
  public void doSomething (B b) {...};
}
class D
{
  public void doSomething () {new C().doSomething(new A());} //still simple
  public void doSomething () {new C().doSomething(new B());} // still simple
  public void doSomething () {A a = new B(); new C().doSomething(a);} //which is now chosen?
```

```
}
```

Src. 3.4: Method overloading

Solution: A solution for this violation is using inheritance and interfaces methods.

ClassOutboundRefClass (50) 114
For a single class, it counts the number of distinct classes from which at least one symbol is directly referenced. Example: Package Dispatcher, Service.cs

ClassPrivateMethodNotUsed (0) 35
This metric counts the unused private methods, which can be deleted. Example: Package: PushPull, Calculator.cs

ClassPublicAttributes (0) 137
The number of public attributes in a class is counted. Example: WS_lot.cs

ClassPublicMethodNoGetSet (20) 58
The public methods without getters and setters are counted. Example: Package LinqRuleData, LinqData.cs

File metrics

FileLOC (1.000) 3.114
The metric counts the lines of code of a file.
Example: Batch.cs

FileTodo (0) 7
It counts the number of strings containing the phrase "todo" or "to do". Example: RXENT.cs

3.3.3 ITec

Table 3.3 shows an overview of the system metrics from iTec.

In the iTec project, it is significant that it has 1.056 classes, and 679 of these classes are in a cycle. More than 50% are in a cycle, which means if you change one class, it has an influence on the other files in a cycle.

The following metrics offer a more detailed view of the iTec project. The results are described as such: the first part is the name of the calculated metric in Sotoarc and Sotograph and the value within the brackets is the maximum value of the metric. The second part is the actual value of this metric. The order of evaluating the metrics is first the rules, then the measures, the cyclomatic dependencies and finally the bad smells.

Rules

First the rules are viewed and evaluated. The most significant violations are in the attribute and file metrics. You should always avoid having violations in the rules. The rules are divided into attribute, file and package rules.

Attribute metrics

AttrExternalUsageRule(0) 798
798 methods from external classes read or write to the current attribute.
Solution: A solution could be the using of static constructors.

Metric	Value
SysFiles	838
SysLOC	155.880
SysMetadata	34
SysMethods	6652
SysNestedClasses	217
SysPackages	38
SysPackagesNotUsed	0
SysPckgInCycles	19
SysReferences	78.957
SysRelevantLines	66025
SysSubsystems	8
SysAttributes	2910
SysCalls	21.832
SysClasses	1.056
SysClassInCycles	679
SysCommentblocks	12.717
SysCommentedOutCodelines	1.476
SysConstantAttributes	1.096
SysDeprecAttributes	0
SysDeprecClasses	0
SysDeprecMethods	0
SysViol	13.726
SysDuplicatedCodeBlocks	217
SysInheritances	26
SysJavaDocComments	6.504
SysInheritance	680
SysAnonymousClasses	145

Tab. 3.3: iTec - System Overview in Sotograph

File metrics

FileAssignmentInOperandRuleViolation(0) 7

In operands assignments are made. This can make the source code more complicated and harder to read. This is a PMD-rule.
Example: AddOnsImpl.java, readStandardOutput()

```
try{
...
  while((i=inread(buf, 0, BUFFER)) != -1) {
  }
}
```

Src. 3.5: Assignment

Solution:

```
if (i=3) // compiler error
if (i==3) // o.k.
if ((i=3)==3) // do you want to do an assignment or a comparison?
---
i= inread(buf, 0, BUFFER);
while (i != -1)
{
  ...
  i= read(buf, 0 , BUFFER);
}
In this example if the assignment is made outside the while, it's easier to understand, but if you forget
      the second assignment this could lead in an endless loop. In the most cases this violation happens
     if you iterate over java.io.*InputStreams.
```

Src. 3.6: Assignment solution

FileAvoidDeeplyNestedIfStmtsRuleViolations (0) 10

The nested if..then statements are hard to read.
Example: WaitForEventTreahType3.java, handleReceivedData()
Solution: Split the if-statements for a clearer structure.

FileCyclomaticComplexityRuleViolations (Scale 1-4 low complexity, 5-7 moderate complexity, 8-10 high complexity, 10+ very high complexity) 17

It counts all the decision points in a method plus one for the method entry. Decision points are: if, while, for and case labels.
Example: YMParse.java, checkBracket(int lineNr)
Solution: Split the decision points, so it has low or moderate complexity.

FileEmptyCatchBlockRuleViolation (0) 37

The rule is violated if empty catch blocks are found, where an exception is caught, but nothing else is done.
Example: Equipment.java, checkDriverRegistration()

```
try {
    tuiMgr.infoHandler.logProcessInfo(2, "--------> Port" + Integer.toString(socketHandler.getPort()) + "
         : " + call);
    }
    catch (Exception e) {
    }
```

Src. 3.7: Empty catch block

Solution: Insert an exception message, so that it becomes clear why and where the exception is thrown.

```
try {
    }
    catch (Exception e) {
      System.err.println("logging Error" + e.getMessage());
    }
```

Src. 3.8: Exception message in a catch block

FileExceptionTypeCheckingRuleViolations (0) 7

At some places an exception is caught and then a check with instanceof is performed. Each exception type that is caught, should be handled on its own catch clause.
Example: SocketHandlerSTC.java

```
if (Tui.DEBUG_STCSERVER_IF) {
  String excepname;
  if (exception instanceof UnkownHostException)
      excepname = "UnkonHostException";
  else if (exception instanceof IOException)
      excepname = "IOException";
}
```

Src. 3.9: Exception with instanceof

Solution:

```
try {
}
  catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
  }
```

Src. 3.10: Exception handling

FileFinalFieldCouldBeStaticRuleViolations (0) 3

If a final field is assigned to a compile-time constant, it could be made static for saving overhead in each object. Example: WaitSoakTimeDialog.java

```
private final String FONT_NAME = "Ms Sans Serif";
```

Src. 3.11: Final field

Solution:

```
private final static String FONT_NAME = "Ms Sans Serif";
```

Src. 3.12: Final field with static

FileSignatureDeclareThrowsExceptionRuleViolation (0) 28

It is unclear which exception the method should throw. It might be difficult to understand and document the vague interfaces.
Example: TestProcessFlow.java, startYodaServer()

```
protected void startYodaServer() throws Exception {}
protected void sendCreateProcessJobCommand() throws Exception {}
```

```
protected void sendStartProcessJobCommand() throws Exception {}
```

Src. 3.13: unclear which Exception is thrown

Solution:

```
protected void startYodaServer() throws IOException {}
protected void sendCreateProcessJobCommand() throws ArrayIndexOutOfBoundsException {}
protected void sendStartProcessJobCommand() throws IOException {}
```

Src. 3.14: Exception handling

FileSwitchDensityRuleViolations (0) 1
A switch statement with a higher ratio of statements to labels, implies that this statement has to do too much work.
Example: YMDialogSelect.java
Solution: Consider moving the statements either into a new method or create subclasses based on the switch variable.

Package

PckgAttributeOverrideRule (0) 2
The number of attributes that are hidden in derived classes.
Example: Package core

PckgClassKnowingDerivedRulde (0) 2
Identifies all classes of the current package that have any knowledge about directly or indirectly derived classes. This is considered bad style because changes to derived classes should never affect a base class.
Example: Package function

PckgClassNestingRule (0) 2
This metric counts all classes in a package, which are nested in more than two levels. Example: Package: gui, level 1 class: BinClassMonitoringWindow.java, BinClassMonitoringWindow$MyBarRenderer, BinClassMonitoringWindow$MyBarRenderer$1

PckgCyclomaticComplexityRuleViolations (Scale 1-4 low complexity, 5-7 moderate complexity, 8-10 high complexity, 10+ very high complexity) 169
The number of decision points in a method plus one for the method entry. Example: Package core, contactLoopHandler.java, actionPerformed() has a cyclomatic Complexity of 22
Solution: Redesign the code and use the KISS (Keep it short and simple) principle.

PckgMethodNamingRule (0) 3
The number of methods in a package, which start with a lowercase letter first. Example: Package igxl, Equipment.java. GPIBSendRequest()

Measures

Method metrics

MethChars (1000) 22.952
This metric shows the number of characters in the method body.
Example: MessagePresenter.java, putInformation()
Solution: Split the method in several small methods.

MethLOC (100) 521
The number of lines of code in the method body.
Example: MessagePresenter.java, putInformation()
Solution: Split the method in several small methods

MethCyclomaticComplexity (10) 70
It calculates the complexity of the current method. The decision points have a big influence on the complexity. Example: ContactLoopWindow.java, viewPassFailChannels()
Solution: Increase the number of unit tests or refactor the source code into smaller pieces to reduce the complexity. If you have smaller pieces, it is easier to test.

MethParameters (5) 21
It counts the number of parameters of the current method. For this metric, it is not the sum of the parameters that is important, but the different types.
Example: ContactLoopWindow.java, ContactLoopWindow()

```
ContactLoopWindow(int left, int top, int width, int height, Site [][] siteConfiguration, int maxColCount,
      int maxRowCount, String[] colorZeroValues, String[] colorOneValues, Color colorOne, Color colorTwo,
      Color backColorZero, Color backColorOne,
Color backColorTwo, ActionListener actionListener, WindowListener windowListener, TreeWillExpandListener
     expansionListener, TreeExpansionListenerr treeExpansionListener, TreeSelectionListener
     selectionListener, TuiMgr tuiMgr, TreeModelListener modelListener)
```

Src. 3.15: Too many parameters in the method

Solution: Instead of using so many parameters with the same type, you can use objects.

Class metrics

ClassAttributes (20) 188
This metric counts the number of attributes of a class independent of its visibility (private, protected or public).
Example: SocketHandler.java
Solution: For better understanding you should try to keep your classes simple, which also includes the amount of attributes.

ClassPublicAttributes (0) 177
The number of public attributes is counted, but attributes inherited from superclasses are not considered. The object-oriented principle forbids using public attributes, because this violates the data hiding principle.
Example: SocketHandler.java Solution: A better way is using methods (get-, set-, is-) or in C# with properties (get-, set-).

ClassPublicMethodNoGetSet (20) 182
The number of public methods without getter or setter, are counted. This is a measure for the interface size of a class. The point of getters and setters is that they are only meant to be used to access the private variable, which they are getting or setting. For later modifications, you only have to modify the getter or setter.
Example: FunctionFactory()
Solution: Use getters and setters.

ClassOutboundRefClass (50) 322
This metric counts the number of distinct classes from which at least one symbol is directly

referenced. Only direct calls are considered.
Example: OpaHandler.java

File metrics

FileInstanceOf (0) 41
The occurrences of "instanceof" in the source code is counted. Each usage of it violates the once and only once principle. In such cases the inheritance relationship is not only encoded in the inheritance graph but also in the clients. The usage of instanceof in equals methods usually makes sense.
Example: MessagePresenter.java
Solution: Usage of equals instead of instance of usually makes sense.

FileLOC (1.000) 3.721
It counts the number of lines of code of a file. Example: ContactLoopWindow.java with 3.721 LOC, it is clearly too many LOC so it is getting more and more complex to understand.
Solution: Split the file in several sub-files.

Duplicated Code Metrics

FileDuplicatedCodeBlocks (0) 5
This metric counts the number of distinct duplicated code blocks, within one file. Solution: The duplicated code is redundant, so evaluate the source code and then eliminate the duplication. Sometimes the GUI elements are also counted as duplicated code blocks, but these duplications are okay.

FileDuplicatedCodeFile (0) 5
The number of files with a duplicated code relationship is counted.

Architectural Metrics

File metrics FileInboundArchViol (0) 2383
The violation shows the number of architectural violations going into this file. Example: TuiMgr.java

3.4 Chapter summary

In this chapter, various metric suits like Chidamber & Kemerer, Metrics for Object-Oriented Design, McCabe and Maintainability Index are explained. The idea of measuring software quality with metrics is good, but the interpretation is difficult. After spending a lot of time in literature research, I found some ranges for CK metrics. As a result of my work, metrics are project dependent. As a proposal for a metrics set, checked regularly, DIT, NOC, RFC and CBO are chosen. Nowadays, there are a lot of tools available, which provide calculation of different metrics. For the Infineon IT-department I did an evaluation for Sotoarc and Sotograph. Before starting the evaluation, I interviewed the employees, to get a better overview of the actual project situation and how testing is performed. Another topic in the questionnaire was about software quality, to get an idea about the employee‘s attitude. Two projects (java, c#) were evaluated by Sotoarc and Sotograph. As a result, the most important metrics are explained and a solution is provided.

4 Tools

In this chapter Sotoarc, Sotograph, Visual Studio 2010, Analyst4j and VizzMaintenance are described and evaluated for improving the software quality of projects from the IT-department.

4.1 Sotoarc and Sotograph

Sotoarc and Sotograph are software analysis tools from hello2morrow. They are specialized in analyzing architectural and structural violations. Therefore, the focus is on metrics which cover the architecture and structure. For this, tool version P4.1.2 release date 110624 with the database version 3.017, is used. The following sections about Sotoarc and Sotograph are referenced from [hel08], [hel10]. The license for the IT-department implies Java, C# and C++ parser. The parser extracts the source code and stores it in a database from where the structural information is extracted. In the database all occurring artifacts and references between these, are stored. Rule violations are calculated with external code checkers, integrated via the code checker plug-in interface. Sotograph for Java comes with a preconfigured PMD integration. PMD is a code-checker tool for scanning the Java source code and for looking at potential problems like dead code, overcomplicated expressions, duplicated code, suboptimal code and possible bugs like empty try/catch/finally/switch statements [PMD11]. PMD is integrated with JDeveloper, Eclipse, JEdit, JBuilder, BlueJ, CodeGuide, NetBeans/Sun Java Studio Enterprise/Creator, IntelliJ IDEA, TextPad, Maven, Ant, Gel, JCreator, and Emacs.

Figure 4.1 shows the Sotoplatform. It is divided into Sotoarc, Sotograph, Sotoreport and Sotoweb. For the purpose of evaluation, only the Sotoarc and Sotograph are relevant. For the architectural analyses and modeling, Sotoarc is used. There, you can model your architecture and have a look at the cyclic dependencies, which are shown in a graph. The information about cyclic dependencies between classes, files, packages and subclasses are calculated at fill time. In Sotograph the quality analysis is performed. There are metric suites provided to get detailed information about the system. It is possible to calculate a trend over several versions of a software system.

Figure 4.2 shows the integration of source code in Sotoarc and Sotograph. A parser extracts the relevant information into the repository. After filling in the data, code checker tools, like PMD, are executed. It is also possible to integrate Sotoarc/Sotograph into Eclipse.

4.1.1 Sotoarc

In Sotoarc you can model your system, check the architectural violations and get an overview of the system metrics. If you want to change your architectural layout and move files from one layer to another, you can inspect the resulting influences. Changes are only visible in Sotoarc

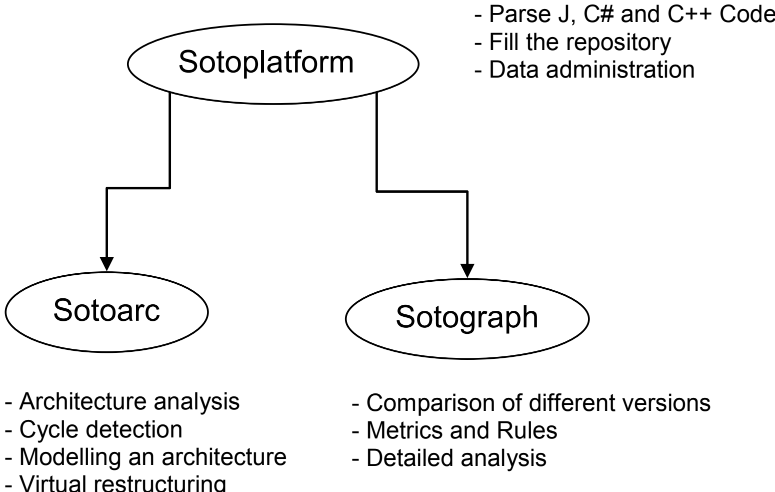

Fig. 4.1: Soto platform

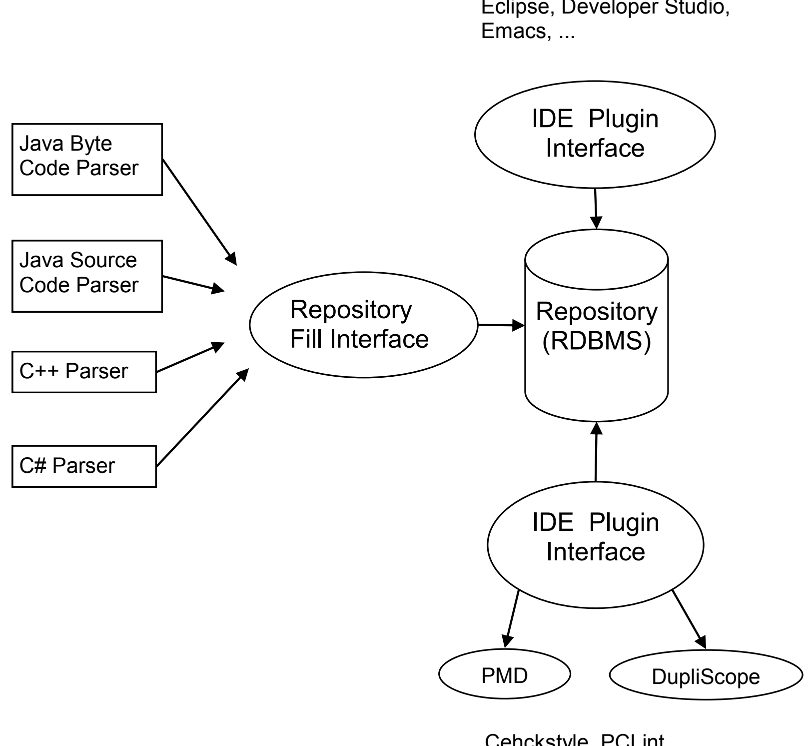

Fig. 4.2: Soto integration

and Sotograph, but it does not change anything in source code. First start Sotoarc and load the project.

If it is a Java project, you have to load the following files:

- Class files
- Directories or directory trees containing class files
- Java archives containing class files (jar, ear, war, zip)

You can generate the packages from the Java packages or from the directories.

If it is a C# project, you have to provide the following files:

- Source Code
- Visual Studio C# solution files or CTP capture tool project files
- Extra directories for the assemblies

You can generate the packages from the C# namespaces or from directories.

From the source code, a lot of information is extracted and saved as artifacts such as classes, methods, attributes and relations about inheritance, read and write access and call-callee dependencies.

After successfully loading the software project in Sotoarc you can switch to the "Architecture menu" and start modeling your system. It is essential map the system in the way it really looks and not how it should look. You want to improve something and therefore, it is very important to model the actual state. The modeling itself does not follow the UML syntax.

The next step is to assign the classes into Sotoarc specific modules.

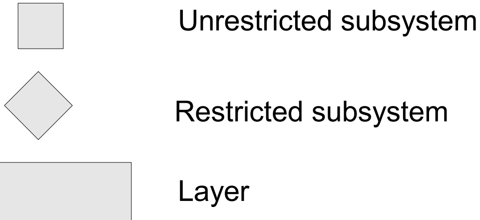

Fig. 4.3: Different modules in Sotoarc

There are three types of modules in Sotoarc, shown in figure 4.3:

- An unrestricted subsystem is a module without restrictions on the dependencies on its sibling modules.
- An independent subsystem is a module which must not depend on its sibling modules, i.e. it must not have references to sibling modules (modules which are located in the same container).
- A layer is a module that forms a layered structure together with its sibling modules. The first (highest) module represents the first layer of that structure; the last (lowest) module represents the lowest layer of that structure. This layered structure defines that higher layers may access lower ones but not vice versa.

Within one module you should only define the same kind of modules. This could be a layer with all unrestricted or independent modules.

After you have modeled your architecture, green and red arcs will arise. If there is a green arc between the nodes, left-handed, this means that downward references derived from the analyzed system. There are one or several references going from artifacts represented by the upper node to artifacts represented by the lower node. If there is a green arc, right-handed, this means that upward references, are derived from the analyzed system. There are one or several references going from artifacts represented by the lower node to artifacts represented by the upper node. Moreover, red arcs indicate architectural violations which should be solved. In chapter 2.4.3 Architectural Violations there are some examples illustrated, where an existing reference in the code is not allowed by the architectural model.

In figure 4.4 you have to define the following kind of modules:

1. Create subsystems
2. Create layer
3. Define the private area
4. Define the allowed depth a layer may have access to

Defining an architecture in Sotoarc needs a lot of time. If you want to check your system and assign it into layers, you have to look at each file and decide carefully in which layer you want to put it. It should be done in several meetings. The first step is to discuss the architecture and find a rough outline and to do the fine tuning in the coming meetings.

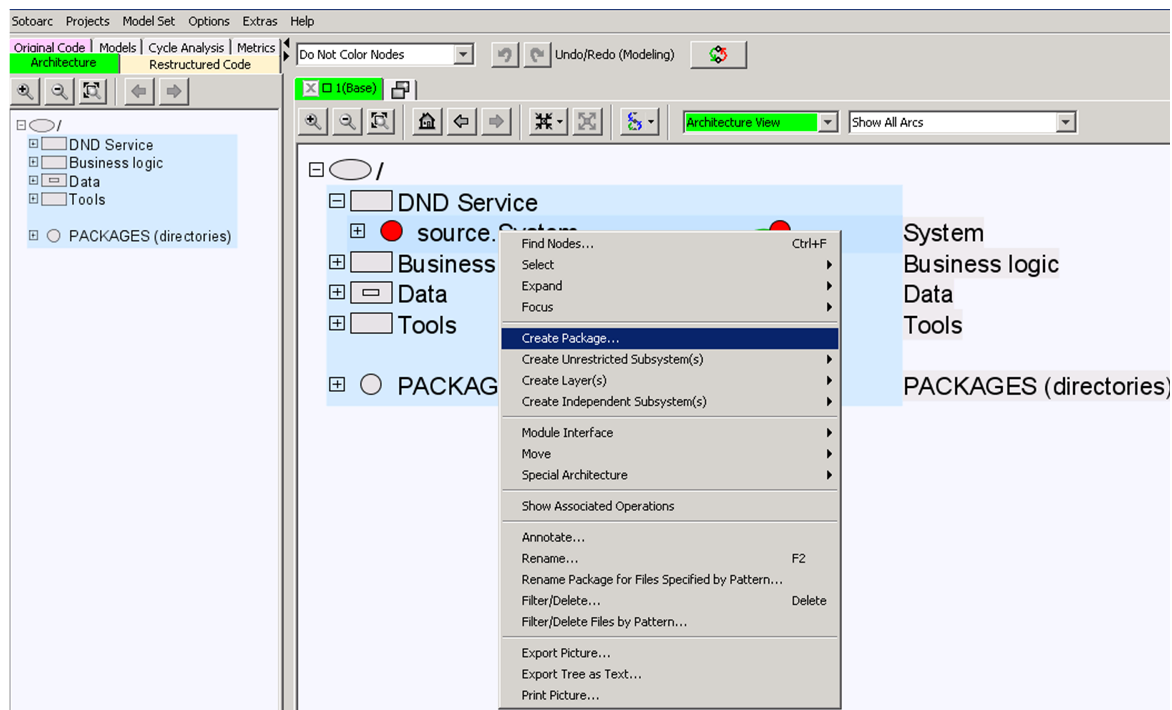

Fig. 4.4: Different modules in your system

Now moving from the theoretical part to a more practical one. In figure 4.5 you can see the architecture from the iTec project. For this example the red arc shows a architectural violation

that should be solved. Finally if there are no red arcs anymore, your architectural design can be continued and analyzed.

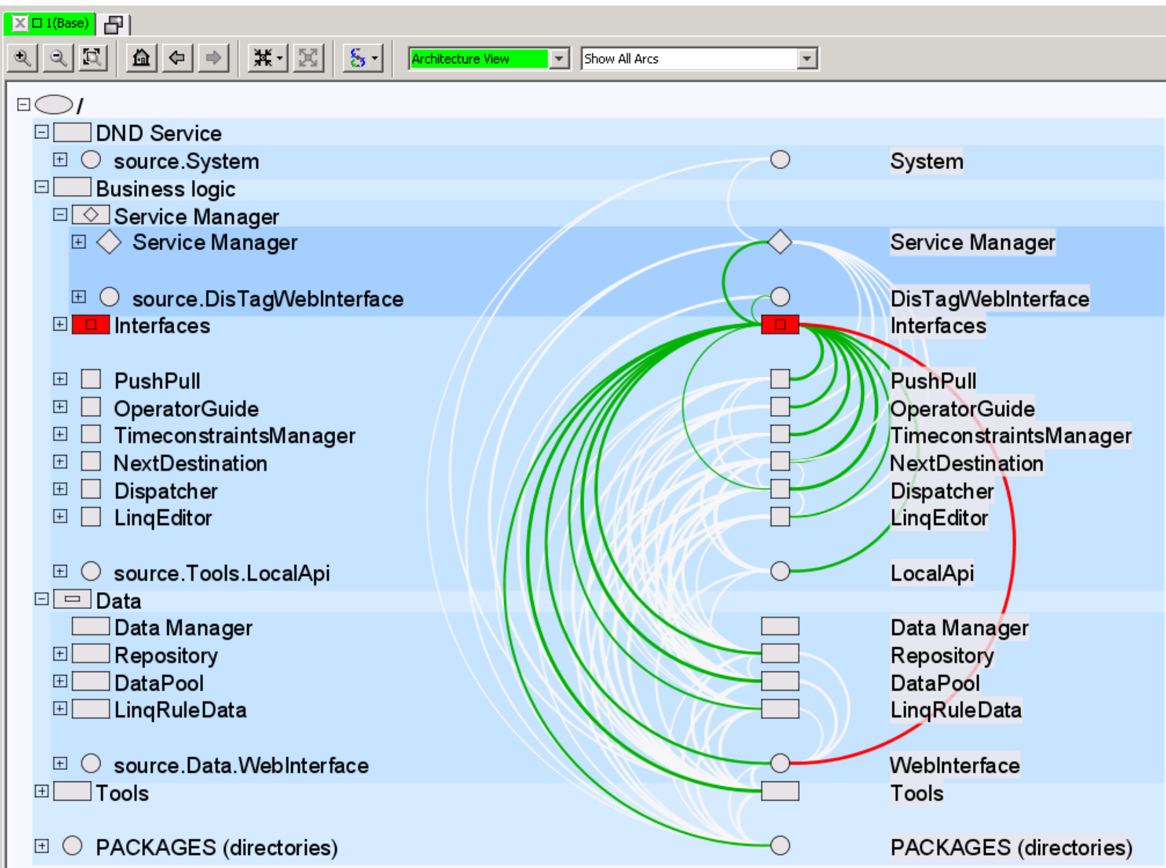

Fig. 4.5: Architectural view

At default you see all references from your files. If you want to see only the violations you can switch the view to: Show Only Violating and Marked Arcs. In figure 4.6 there are only the red arcs shown.

If you want to see only the references between one class and the others, you can focus on dependencies with right click and choose Focus ->Focus on Dependencies. Figure 4.7 shows you the resulting view.

To get details about the architectural violation you can click on the red arc and a tool tip will appear, which is shown in figure 4.8.

In Sotoarc you can check the cyclic dependencies of your project. At first it is better and easier to check the file dependencies. In figure 4.9 you can see a cycle with 5 files, involved in two packages. On the right side there is a table with the current cyclicity. You can show how these values change if you cut one dependency. One cycle group may split into several cycle groups by cutting some dependencies.

- CYC: the remaining cyclicity of nodes in the graph
- CycNodes: the remaining number of nodes belonging to cycle groups of the graph

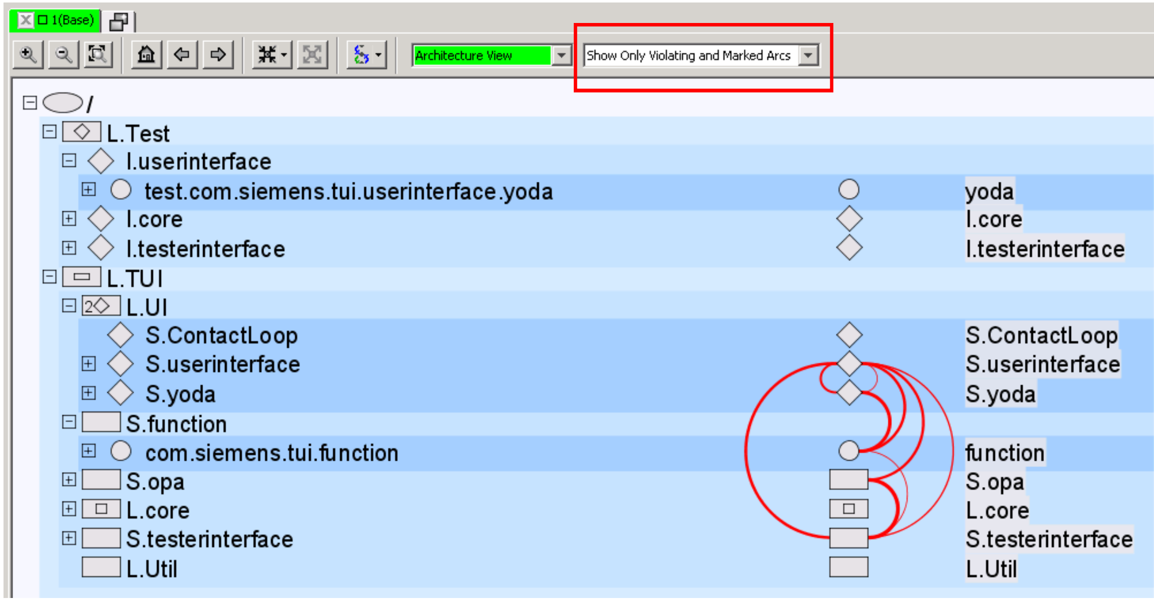

Fig. 4.6: Show only violated arcs

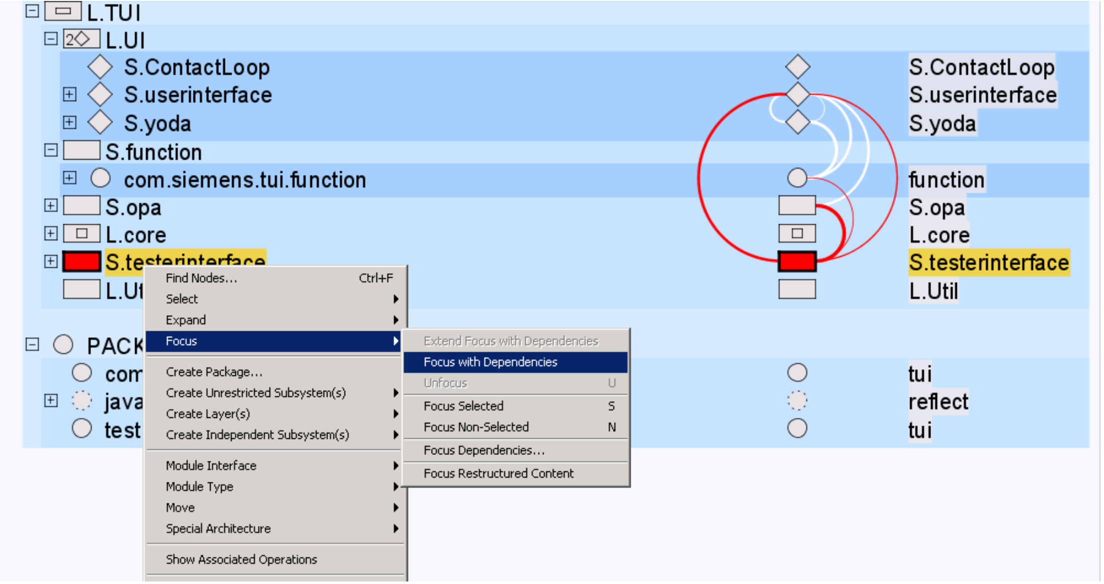

Fig. 4.7: Focus on dependencies

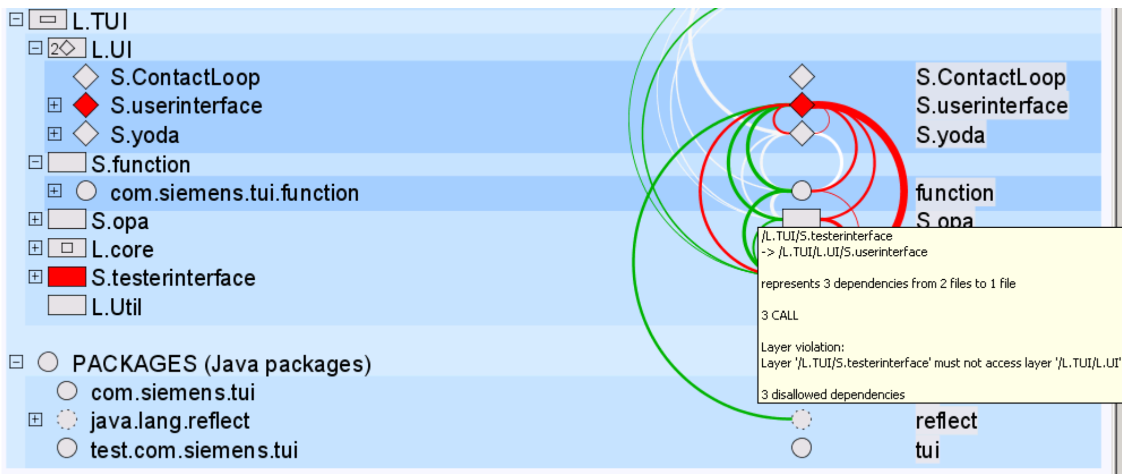

Fig. 4.8: Tool tip about layer violation

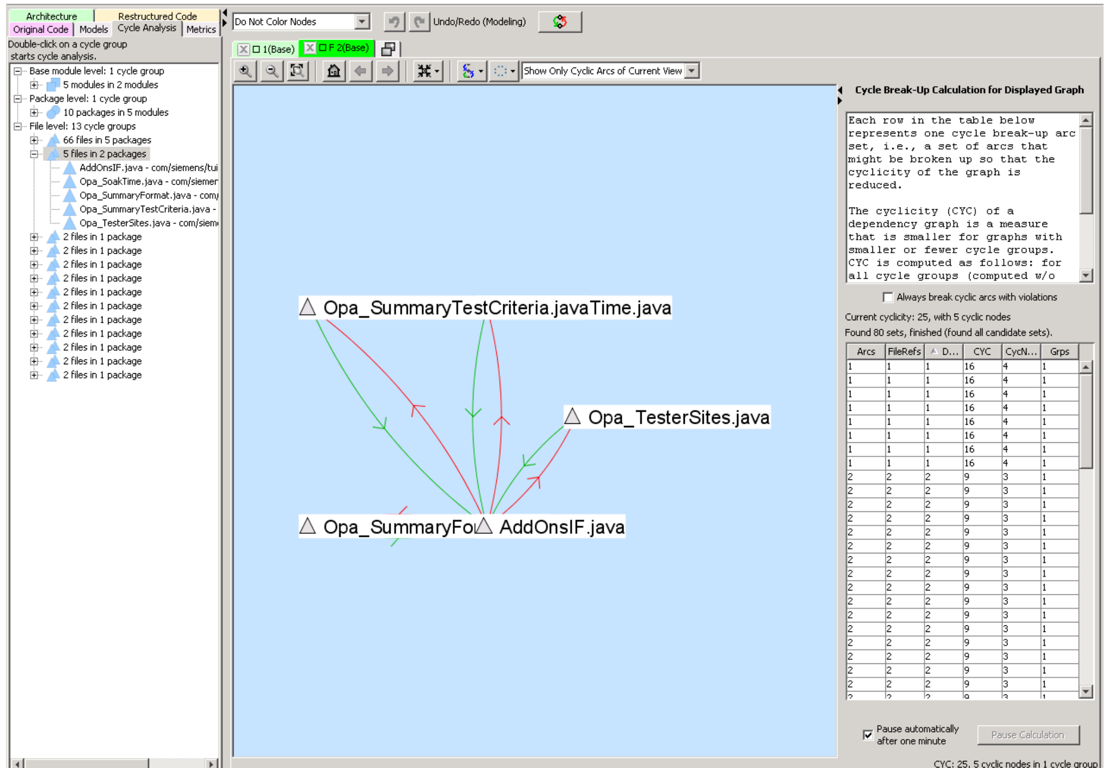

Fig. 4.9: File dependence visualized in a graph

- Grps: the number of cycle groups in the graph

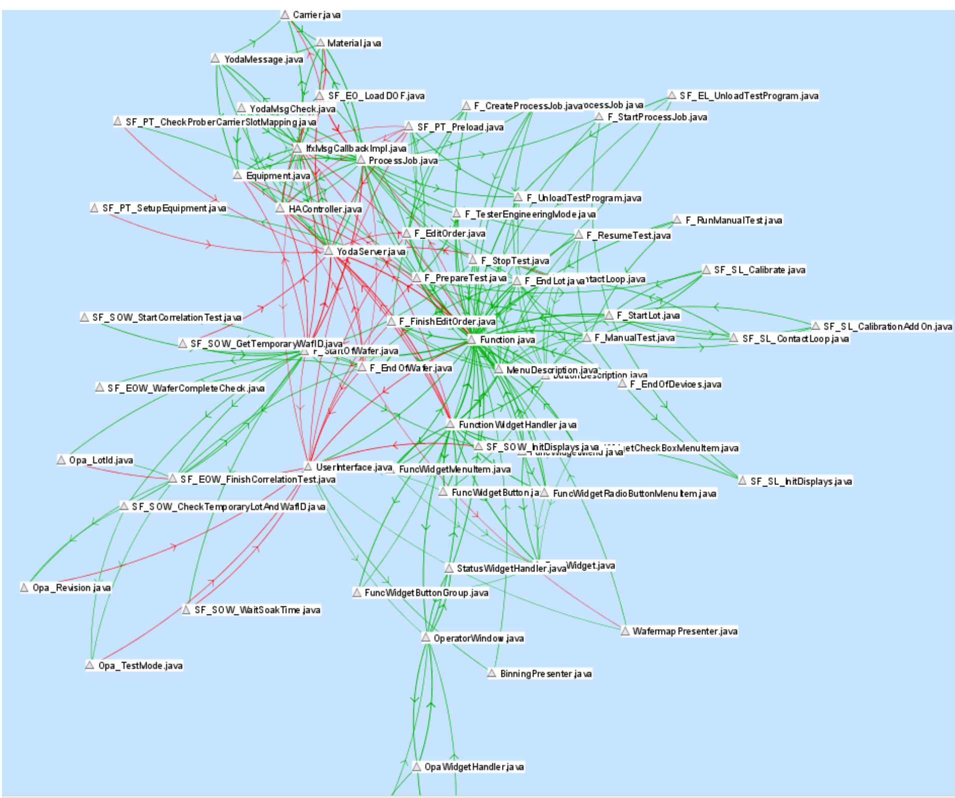

Fig. 4.10: Cyclic dependencies visualized in a graph

For further details check the Sotoarc hands-on. After finishing the modeling activities, you can print out a report. Select 'Extras ->Generate Overview Report...' from the menu bar. Then you have a document where all the changes, which have been made in the architecture, are presented. The challenge is to change the source code in a way, that it fits the architectural model.

Loading a new project into Sotoarc for calculating trends and comparing metric results you have to choose "Project ->Parse project". When you have already parsed a version of the current project you can use the history option to retrieve the entered data again. Next add the byte code and source files.

If you want to change your database, you can use the migration wizard. First start Sotoarc, load Project1_1 and open (click on) the menu bar "Projects ->Admin...". In the Sotoadmin window click in the menu bar "Database ->Migrate "Project X". Finally a migration wizard appears and you can select the target database into which information from the old database will be migrated.

The most important part is the architectural evaluation. You can check cycle dependencies to get an overview of all files involved. For more detailed software evaluation, you have to switch to Sotograph.

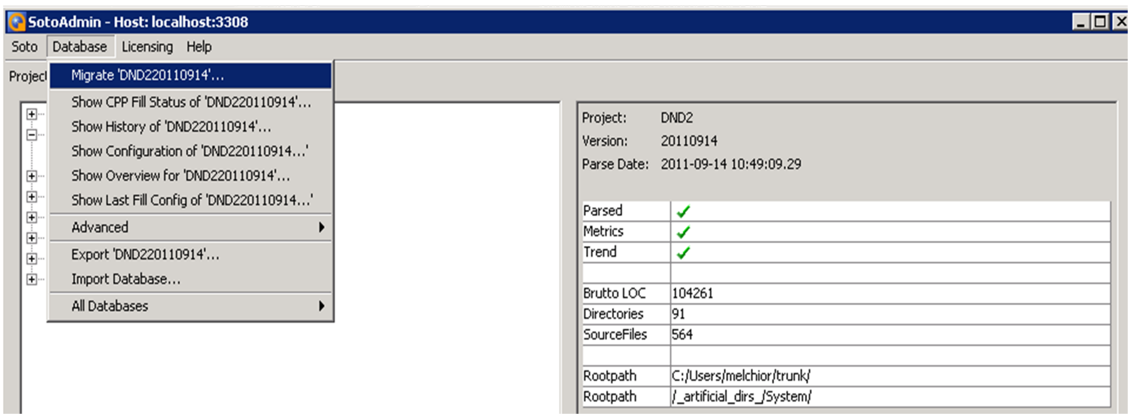

Fig. 4.11: Project migration in Sotoarc

4.1.2 Sotograph

Sotograph is used for fine evaluation of software metrics. The strategy for analyzing the metric value in Sotograph is first to get an overview of the system metrics. The most important part of Sotograph is the MetricScope. You can switch between different scopes via the menu bar, like GraphScope, QueryScope, ResultScope, TrendMetricScope and XrefScope. These scopes are explained before starting with the MetricScope, which is the most important tool in Sotograph.

GraphScope

In the GraphScope you can generate graphs and visualize parts of your system. First you have to configure the graph settings. You can select the dependency type "Call", "Inheritance" or "Package Nesting" as referenced kind. In order to obtain a better overview of the involved packages in the graph, you can switch from system to subsystem or packages level.

QueryScope

In this scope, you inspect your system and browse between different categories, like "Bad Smells", "Patterns", "Trend for 2 last version" and many more. In the category Bad Smells, you can evaluate the bottlenecks and you get an overview of which files have many in and outs.

ResultScope

The purpose of this tool is to depict analysis results of all varieties in the form of tables.

TrendMetricScope

The TrendMetricScope has a similar manner as the MetricScope. It additionally displays metric results for different versions of the software system. Therefore, you can observe the values of various metrics between two versions, shown in figure 4.12

It is also possible to generate a trend chart of two versions, where you can see how metric values changed between two versions, shown in figure 4.13

XrefScope

The XrefScope is a high level cross-referencer which enables you to analyze relationships between sets of artifacts. You can choose an artifact (class, file, package or subsystem) as an anchor of

ID	Artifact Name	Artifac...	ID	ID	Metric Name	V1	V2	Diff
SY	execute	METHOD	FL	PA	MethChars	7751.0	7849.0	98.0
FL	InfoObjMsg.java	FILE	FL	PA	FileInboundArchViol	455.0	551.0	96.0
FL	InfoObjMsg.java	FILE	FL	PA	FileInboundViol	455.0	551.0	96.0
SY	unHideLimitsTreeSites	METHOD	FL	PA	MethChars	2652.0	2747.0	95.0
SY	update	METHOD	FL	PA	MethChars	2274.0	2368.0	94.0
SY	updateValue	METHOD	FL	PA	MethChars	7682.0	7773.0	91.0
SY	put	METHOD	FL	PA	MethChars	6921.0	7824.0	903.0
SY	DEBUG_TESTERIF	ATTRI...	FL	PA	AttrExternalUsageRule	229.0	238.0	9.0
SY	TuiMgr	CLASS	FL	PA	ClassMethods	121.0	130.0	9.0
FL	MenuDescription.java	FILE	FL	PA	FileInboundArchViol	27.0	36.0	9.0
FL	MenuDescription.java	FILE	FL	PA	FileInboundViol	27.0	36.0	9.0
SY	isEnabled	METHOD	FL	PA	MethChars	1022.0	1031.0	9.0
SY	sortMenu	METHOD	FL	PA	MethChars	1570.0	1579.0	9.0
SY	mouseReleased	METHOD	FL	PA	MethChars	2127.0	2136.0	9.0
SY	createComboBoxContainer	METHOD	FL	PA	MethChars	1145.0	1154.0	9.0
SY	createMenu	METHOD	FL	PA	MethChars	5434.0	6315.0	881.0
SY	updateFunctionWidgets	METHOD	FL	PA	MethChars	1873.0	2753.0	880.0
SY	updateUI	METHOD	FL	PA	MethChars	3180.0	4022.0	842.0
SY	Tui	METHOD	FL	PA	MethChars	8841.0	8924.0	83.0
SY	hideLimitsTreeSites	METHOD	FL	PA	MethChars	3266.0	3349.0	83.0
SY	updateFctStatus	METHOD	FL	PA	MethChars	2306.0	3132.0	826.0
SY	parseColorMapping	METHOD	FL	PA	MethChars	1593.0	1674.0	81.0
SY	parseColorMapping	METHOD	FL	PA	MethChars	1593.0	1674.0	81.0
SY	parseColorMapping	METHOD	FL	PA	MethChars	1593.0	1674.0	81.0
SY	invertSiteChannel	METHOD	FL	PA	MethChars	4308.0	4389.0	81.0
SY	userInterface	ATTRI...	FL	PA	AttrExternalUsageRule	64.0	72.0	8.0
SY	StatusHandler	CLASS	FL	PA	ClassNeighborsInCycle	55.0	63.0	8.0
SY	WaitForEventThreadTyp5	CLASS	FL	PA	ClassOutboundRefClass	16.0	24.0	8.0
SY	OpaHandler	CLASS	FL	PA	ClassOutboundRefClass	314.0	322.0	8.0
SY	TuiMgr	CLASS	FL	PA	ClassPublicMethods	112.0	120.0	8.0
FL	SF_PT_Preload.java	FILE	FL	PA	FileOutboundArchViol	3.0	11.0	8.0
FL	SF_PT_Preload.java	FILE	FL	PA	FileOutboundViol	3.0	11.0	8.0
SY	isCurrentTOVAppropriate	METHOD	FL	PA	MethChars	1092.0	1100.0	8.0
SY	createOpas	METHOD	FL	PA	MethLOC	317.0	325.0	8.0
PA	function	PACKAGE	ID	PA	PckgDuplicatedCodeBlocks	27.0	35.0	8.0
PA	opa	PACKAGE	ID	PA	PckgFiles	307.0	315.0	8.0
SY	execute	METHOD	FL	PA	MethChars	1521.0	1599.0	78.0
SY	updateOpaUI	METHOD	FL	PA	MethChars	2847.0	3622.0	775.0
SY	createNodes	METHOD	FL	PA	MethChars	1580.0	1656.0	76.0
SY	checkReSelectPorts	METHOD	FL	PA	MethChars	2087.0	2805.0	718.0
SY	EventHandler	CLASS	FL	PA	ClassOutboundRefClass	41.0	48.0	7.0
FL	SocketKeyNotFoundException.java	FILE	FL	PA	FileInboundArchViol	4.0	11.0	7.0
FL	Function.java	FILE	FL	PA	FileInboundArchViol	172.0	179.0	7.0
FL	UserInterface.java	FILE	FL	PA	FileInboundArchViol	101.0	108.0	7.0
FL	SocketKeyNotFoundException.java	FILE	FL	PA	FileInboundViol	4.0	11.0	7.0
FL	Function.java	FILE	FL	PA	FileInboundViol	172.0	179.0	7.0

Fig. 4.12: Trendmetric in Sotograph

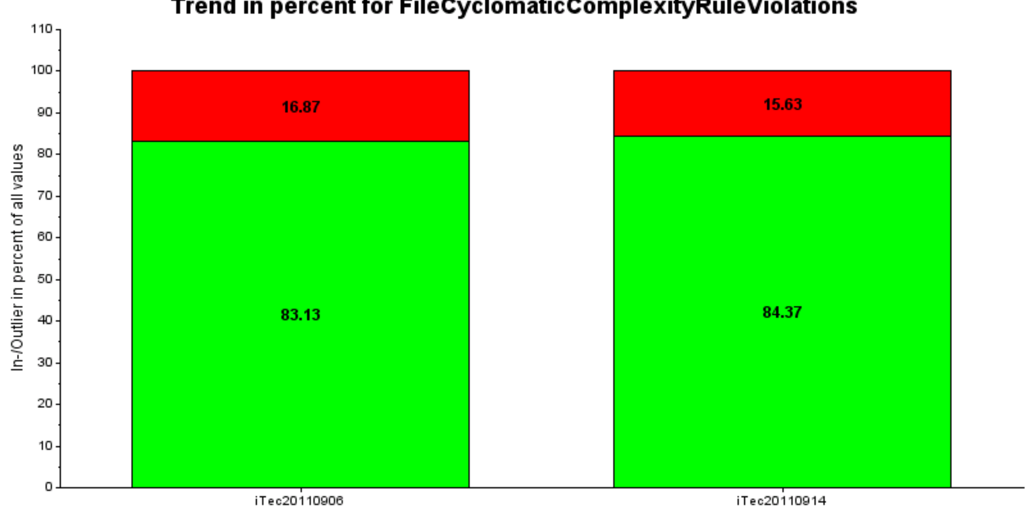

Fig. 4.13: Trend chart

a xref query. Then you have to specify the "Search Direction" of the query, i.e anchor refers to the result to find artifacts that are referencing the anchor.

MetricScope

The MetricScope enables you to analyze a software system by measuring metrics and rules of a quality model. A quality model defines a set of metrics and rules which is tailored for analyzing software implemented in a certain programming language or even a single project. For metrics it is possible to specify the boundaries that separate uncritical and critical value ranges.

First, figure 4.14 shows the Sotograph MetricScope view.

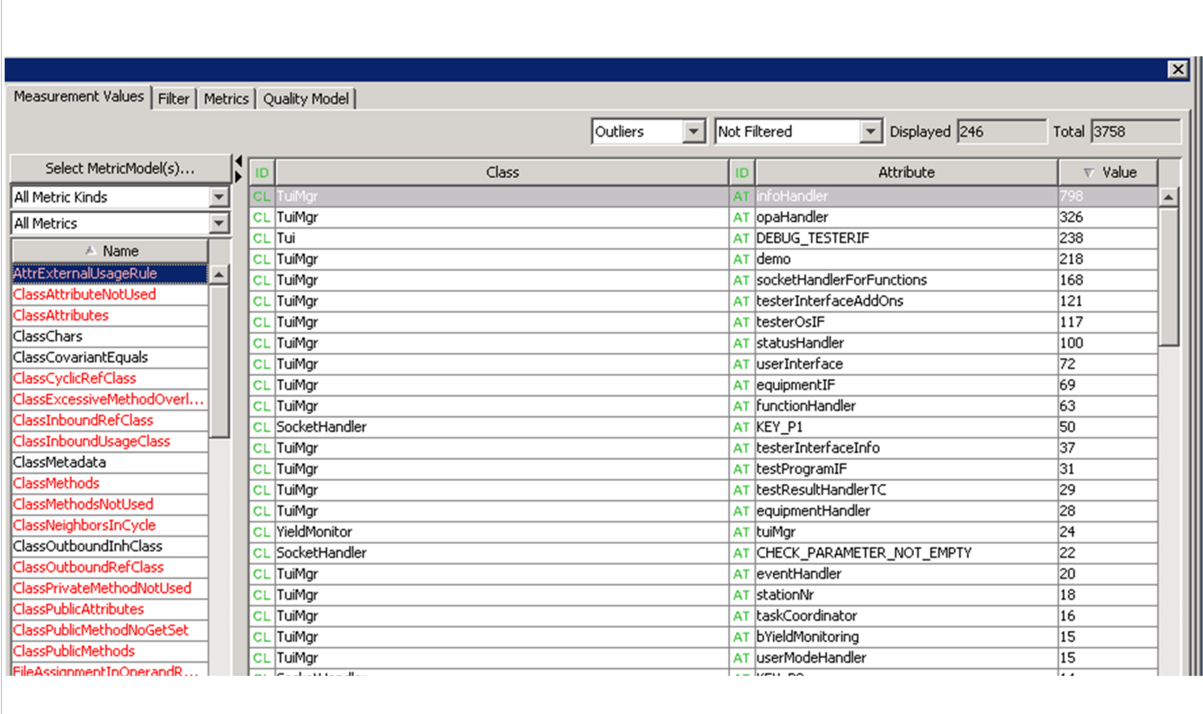

Fig. 4.14: Sotograph explanation

Switch to the MetricScope. At first check the duplicated code metrics, rules and finally the measures, shown in figure 4.15.

You only have to check the metrics in red, because they indicate a violation shown in figure 4.16.

4.1.3 Choice of metrics

In Sotograph [hel08], software analysis takes place as well as detailed metrics calculations. To get an overview of the system metrics, they can be checked in the Sotograph MetricScope-Metric Values -⟩ Show System Metrics Overview.

Metrics are divided into:

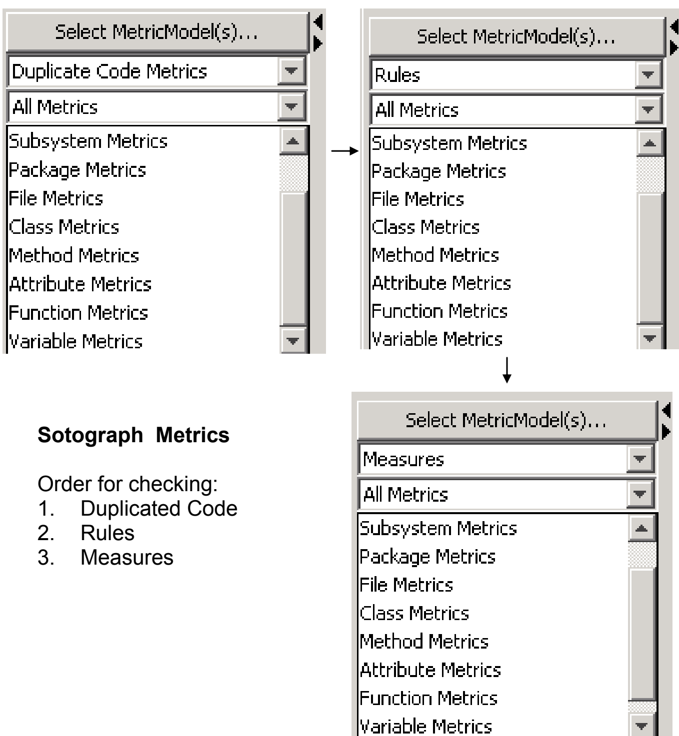

Fig. 4.15: Order of metrics in Sotograph

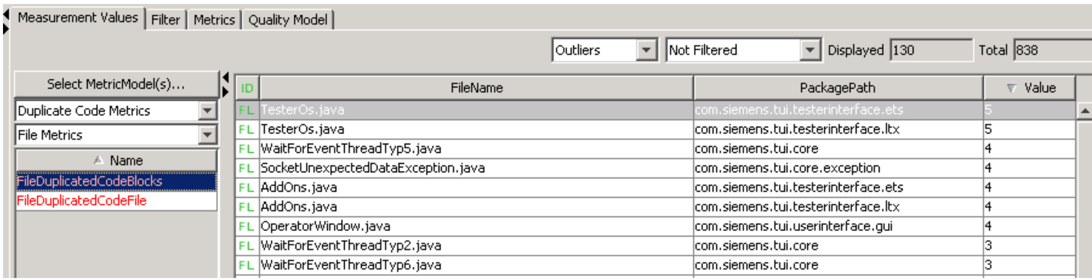

Fig. 4.16: Violated metrics

- Architectural Metrics
- Cycle Metrics
- Duplicate Code Metrics
- Measure
- Rules

Rules define conditions that have to be fulfilled by structural aspects of a software system. Rules violations should always be fixed. You can distinguish between architectural and structural level rules. Measures find artifacts on any abstraction level which point out problems. In the literature, measures are usually called metrics. In Sotoarc/Sotograph, artifacts are classes, methods, attributes, functions, variables, source files, directories, Java packages and namespaces.

Before starting a detailed analysis, it makes sense to visualize the cyclic metrics to a graph for getting a better overview, for e.g. on subsystem or package level. The problem with cyclic dependencies is that it is hard to maintain the software. There are two options as how to handle cyclic problems:

- Decide if the graph is still manageable and avoid that further packages are added to the cycle.
- Start cleaning up. The best way to start is to find semantic information. Defining a layered architecture model and illegal arcs is a good starting point for cleaning up.

The following rules and violations are a suggestion from hello2morrow, which are the most important metrics based on their experience. The next step is to check duplicated code metrics. In this section the **SysDuplicatedCodeBlocks** is relevant. Therefore, the number of distinct duplicated code blocks in the system are counted.

The next step is to evaluate the rules. There is a difference between C# and Java source code. For C# there are default metrics available and for Java you can switch to detailed metrics. In the drop-down menu "All metrics" the following metrics are important:

- Attribute Metrics: AttExternalUsageRule
  The number of methods from external classes are counted, which read from or write to the current attribute.
- Class Metrics: ClassCovariantEquals (JAVA)
  It counts classes, that define a covariant equals method, i.e., one in which the parameter type equals the class in which the method is defined should also override equals (java.lang.Object). Mistakenly defined a covariant equals() method without overriding the equals(java.lang.Object) can produce unexpected runtime behavior.
- File Metrics
  There are the following rules in file metrics to check:
  - FileAvoidDeeplyNestedIfStmtsRuleViolation
    This metric counts the deeply nested if..then statements, which are difficult to read.
  - FileEmptyCatchBlockRuleViolations
    This metric counts the sum of all empty catch blocks for every file, where nothing is done.

  - FileExceptionTypeCheckingRuleViolation
    This metric counts all the rule violations where at some places exceptions are caught and then a check with instanceof is performed. It is better to catch all the specific exceptions instead.
  - FileFinalFieldCouldBeStaticRuleViolation
    If a final field is assigned to compile-time constant, it could be made static saving overhead in each object.
  - FileSignatureDeclareThrowsExceptionRuleViolation
    The problem concerning this violation is that it is unclear which exception can be thrown from the methods. Use either a class derived from RuntimeException or a checked exception.
  - FileSwitchDensityRuleViolation
    For every file, this metric counts the sum of all rule violations for the rule SwitchDensity. This rule is a high ratio of statements to labels in a switch statement, which implies that the statement is responsible for too many functions. If the statements are too nested, it is better to move some of the functionalities into new methods or create subclasses.

From the package metrics the following metrics are of primary importance:

- PckgAttributeOverrideRule
  This rule contains the number of attributes, that are hidden in derived classes.
- PckgClassConstructorRule
  The number of methods in a package with the same name as their class, that are not a constructor.
- PckgClassEqualsAndHashCodeRule
  Number of classes in the package overriding method hashCode() or equals (), but not both.
- PckgClassIllegalInRule (JAVA)
  Number of classes in a package that are direct or indirect subclasses of java.lang.Error, java.lang.Throwable, java.lang.RunTimeException.
- PckgClassKnowingDerivedRule
  Identifies all classes of the current package that have any knowledge about directly or indirectly derived classes. This is considered bad style because changes to derived classes should never affect a base class.
- PckgClassNestingRule
  Number of classes in the package that are nested more than two levels, because then it gets more and more complex.
- PckgFinalFieldCouldBeStaticRuleViolation
  For every package, this metric counts the sum of all rule violations for FinalFieldCouldBeStatic. This contains the final assigned constants, which could be static to save overhead in each object.

- PckgMethodMainRule (JAVA)
  The number of methods in a package, which are called main but are not "public static void main (java.lang.String[])".

The same rule violation as for package level will be violated on the file level. For the system level, these two metrics should be considered:

- SysRefDeprecClassRule
  This rule counts all those directly referenced by deprecated classes. Every call, via objects or static methods, is considered.
- SysRefDeprecMethAttrRule
  The metric counts all directly called deprecated methods or directly used deprecated attributes of classes.

The next step is to switch from "All Metric Kinds" to "Measures" in the drop-down-box. Method Metrics:

- MethCyclomaticComplexity
  It is determined by the number of decision points in a method plus one for the method entry.
- MethLOC
  The sum of code lines within a method is counted.
- Methparameter
  This metric counts the number of current method parameters.

Class Metrics:

- ClassExcessiveMethodOverloading
  This metric counts the maximum number of times a method is overloaded in a class.
- ClassOutboundRefClass
  It counts the number of distinct classes from which at least one symbol is directly referenced.
- ClassPrivateMethodNotUsed
  The private method that is not used, can be deleted.
- ClassPublicAttributes
  The number of public attributes is counted.
- ClassPublicMethodNoGetSet
  The number of public methods without any getters and setters is counted.

File Metrics:

- FileCommentedOutCodeLines
  It counts the number of files where code is commented.
- FileHack
  All strings of the source code are checked for containing the word "hack".
- FileLOC
  It counts the lines of code in a file.

- FileNewInstance
  It counts the number of occurrences of the string "newInstance" in the source code.
- FileToDo
  This metric counts the number of strings containing the phrase "todo" or "to do" in a file.

Package Metrics:
PckgOutboundRefPckg: It counts the number of other packages, it refers to within one package.

System Metrics:

- SysDeprecAttributes
  This metric counts all non library deprecated attributes.
- SysDeprecClasses
  It counts all non library deprecated classes.
- SysDeprecMethods
  It counts all non library deprecated methods.
- SysMetadata
  This metric counts the number of all used metadata, which are annotations in Java and attributes in C#.

4.1.4 Metrics prioritization

For better understanding you can divide the metrics into different categories. For a categorization only inheritance, coupling/cohesion, complexity and polymorphism from the object oriented principles are relevant. In addition, there are two more categories for the structure violations and duplicated code metrics.

Structure metrics

Structure metrics are violations in the structure of source code, such as coding style errors or general source code information. The following metrics belong to this category.

- FileEmptyCatchBlockRuleViolation
- FileExceptionTypeCheckingRuleViolation
- FileFinalFieldCouldBeStaticRuleViolation
- FileSignatureDeclareThrowsExceptionRuleViolation
- PckgClassConstructorRule
- PckgClassEqualsAndHashCodeRule
- PckgClassIllegalInRule
- PckgFinalFieldCouldBeStaticRuleViolation
- PckgMethodMainRule
- SysRefDeprecClassRule
- SysRefDeprecMethAttrRule
- MethLOC

- Methparameter
- ClassPrivateMethodNotUsed
- ClassPublicMethodNoGetSet
- FileCommentedOutCodeLines
- FileHack
- FileLOC
- FileNewInstance
- FileToDo
- SysDeprecAttributes
- SysDeprecClasses
- SysDeprecMethods
- SysMetadata

Duplicated code metrics
SysDuplicatedCodeBlocks

Inheritance
PckgClassKnowingDerivedRule

Coupling/Cohesion
- AttExternalUsageRule
- ClassOutboundRefClass

Complexity
- FileAvoidDeeplyNestedIfStmtsRuleViolation
- FileSwitchDensityRuleViolation
- PckgClassNestingRule
- MethCyclomaticComplecity
- PckgAttributeOverrideRule
- MethCyclomaticComplexity
- ClassExcessiveMethodOverloading

Polymorphism
- Methparameter
- ClassExcessiveMethodOverloading

In 4.1.3 Choice of metrics, metrics are listed from the smallest to the largest unit. The most important metrics, which should be checked regularly, are listed below. Please mind that the Sotoarc and Sotograph metrics are focused on architectural and structural violations.

Prioritization of Sotograph metrics:

1. AttExternalUsageRule
2. FileAvoidDeeplyNestedIfStmtsRuleViolations
3. MethCyclomaticComplexity
4. MethLOC
5. Methparameter
6. FileEmptyCatchBlockRuleViolations
7. PckgClassKnowingDerivedRule
8. ClassExcessiveMethodOverloading
9. ClassPublicAttributes
10. ClassPublicMethodNoGetSet

The following list is a set of metrics and threshold that have proved to be best practice in many Java projects [MvZ11].

1. Normalized cumulative component dependency (project) <= 7
2. Number of types (package) <= 50
3. Number of public types (package) <= 30
4. Lines of code (compilation unit) <= 700
5. Number of methods (type) <= 50
6. Cyclomatic complexity (method) <= 20
7. Number of parameters (method) <= 7

4.2 Visual Studio 2010

Visual Studio provides some powerful tools for improving source code such as Test and Lab Manager. For this evaluation Microsoft Visual Studio 2010 version 10.0.30319.1 is used.

4.2.1 Metrics in VS 2010

In the menu bar "Analyze" you are able to run code analysis on your project. Per default the Microsoft Basic Design Guideline Rules are selected, which include the Maintainability Index (MI), Cyclomatic Complexity (CC), Depth of Inheritance, Class Coupling and Lines of Code. After compiling you get an overview about the most important metrics. The challenge here is to interpret the results. Checking whether or not the metrics are within the ranges is explained in chapter 3.2 Common Metrics.

4.2.2 Code coverage

Code coverage means to figure out the percentage of code that is covered by the last execution [KS10]. To enable code coverage, you have to open the test settings, go to "Data and Diagnostics" and enable it. The following figure 4.17 shows you a screen shot with the code coverage results from the test run.

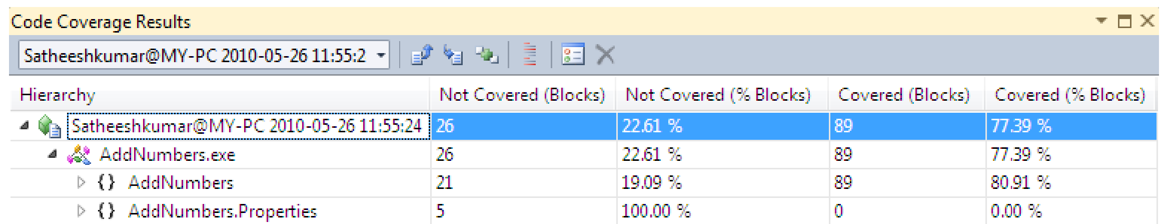

Fig. 4.17: Code Coverage

You can also enable the code coverage setting for unit testing. It shows you the code that has been covered by unit testing. Therefore, you have to specify the configuration in the Data and Diagnostics wizard. Select the test and run test again setting.

4.3 Comparison of Sotograph and VS 2010

Table 4.1 shows a comparison of Sotoarc/Sotograph and Visual Studio 2010. This comparison is concerned with metrics and architectural evaluation.

Name	Sotoarc/Sotograph	VS 2010
Multi languate	C, C++, C#, Java	C, C++, Visual Basic
Generates architectural design	✓	✓
Generates dependency graph	✓	✓
Generates directed graph	✓	✓
Calculates metrics	✓	✓
Detailed metric evaluation	✓	
Tips for solving violations	✓	

Tab. 4.1: Comparison of Sotoarc/Sotograph an VS 2010

4.4 Comparison of Analyst4j and VizzMaintenance

Figure 4.18 shows a list of metric tools evaluated in [com08]. The 'x' marks that a metric can be calculated by the corresponding metric tool.

To analyze software projects there are several metric tools available online. The following two tools are compared:

1. Analyst4j [Cod13]
2. VizzMaintenance [Ari13]

The installation as such was very easy. Analyst4j is available as a stand-alone version and also as an Eclipse IDE plug-in. It features search, metrics analyzing quality and report generation for Java programs. VizzMaintenance is used as an Eclipse plug-in. VizzAnalyzer is not longer available as a stand-alone and VizzMaintenance is its successor.

Tools	Metrics								
Name	CBO	DIT	LCOM-CK	LCOM-HS	NOC	NOM	RFC	TCC	WMC
Analyst4j	x	x	x		x	x	x		x
CCCC	x	x			x	x			
Chidamber & Kemmerers Java Metrics	x	x	x		x	x	x		
Dependency Finder		x			x	x			
Eclipse Metrics Plugin 1.3.6		x		x	x	x			x
Eclipse Metrics 3.4			x	x					x
OOMeter	x	x	x		x			x	
Semmle		x	x	x	x	x	x		
Understand for Java	x	x	x		x	x			
VizzAnalyzer	x	x	x		x	x	x	x	x

Fig. 4.18: Tools and metrics in evaluation [com08]

4.4.1 Metric results

For source code evaluation, a small project of my own is used, because more different tools are tested also from [com08], which are considered. By comparing the results for LOC, there are huge differences. Analyst4j computed 2.628 LOC, while VizzMaintenance got 3.170 LOC. Obviously, even LOC is tool-dependent. The other metrics are not so easy to compare. Results provided by Analyst4j are only average values. In contrast, in VizzMaintenance, there are metrics calculated for each file. In either case, there are no range values defined to see which metric values are acceptable and which are not. The interpretation is still the hardest part and for this you have to know how each metric is calculated and how to interpret the computed results.

4.4.2 Analysis

In [com08] the computation of different metrics also returned different results depending on the used metric tool. They picked out a small class from the jTcGUI project. For calculation the scope is important, which means for each metric you have to decide if all classes, methods, constructors, calls, accesses etc. are taken into account.

Cohesion - CBO and RFC Analyst4j calculates for CBO 4 and RFC 12, because API classes are considered. For the evaluation in this paper the results are from VizzAnalyzer. For CBO it is 1 and RFC 6, due to the API not being in scope and in any case, the constructor does not count as a method either.

Inheritance - DIT Metric tools try to calculate the inheritance hierarchy of classes. There are also different results. Analyst4j returns DIT 2 while the API is not within the scope for VizzAnalyzer and therefore DIT is 0.

Size and Complexity - LOC, NOM and WMC For these metrics, source code is essential and for computation complexity all the entities and relations are considered, like loops and conditions. VizzAnalyzer calculated LOC 64 by considering the full class declaration plus class

Tool	Data	CBO	DIT	LCOM-CK	LCOM-HS	LOC	NOC	NOM	RFC	WMC
Analyst4j	Max of Value	32.000	3.000	0.997			1.000	25.000	155.000	42.000
	Min of Value	4.000	1.000	0.800			0.000	6.000	12.000	10.000
	Average of Value	17.000	2.000	0.895			0.200	12.800	73.600	24.000
C&K Java Metrics	Max of Value	25.000	6.000	664.000			1.000	37.000	118.000	
	Min of Value	0.000	0.000	1.000			0.000	6.000	14.000	
	Average of Value	8.000	3.000	165.800			0.200	15.800	55.200	
CCCC	Max of Value	13.000	2.000				1.000	25.000		
	Min of Value	4.000	0.000				0.000	0.000		
	Average of Value	8.200	1.000				0.200	9.200		
Dependency Finder	Max of Value		2.000			231.000	1.000	45.000		
	Min of Value		1.000			30.000	0.000	6.000		
	Average of Value		1.200			108.200	0.200	19.600		
Eclipse Metrics Plugin 1.3.6	Max of Value		7.000		0.917		1.000	25.000		38.000
	Min of Value		1.000		0.000		0.000	6.000		9.000
	Average of Value		4.400		0.658		0.200	12.600		22.000
Eclipse Metrics Plugin 3.4	Max of Value			10.000	0.960					38.000
	Min of Value			0.000	0.000					9.000
	Average of Value			2.000	0.648					22.200
Semmle	Max of Value		4.000	323.000	0.980	314.000	1.000	25.000	109.000	
	Min of Value		1.000	7.000	0.839	50.000	0.000	6.000	8.000	
	Average of Value		2.600	97.000	0.903	150.400	0.200	12.600	57.400	
Understand For Java	Max of Value	32.000	7.000	96.000		407.000	1.000	25.000		
	Min of Value	5.000	1.000	73.000		59.000	0.000	6.000		
	Average of Value	**17.600**	4.400	82.200		200.600	0.200	12.800		
VizzAnalyzer	Max of Value	4.000	1.000	274.000		410.000	1.000	24.000	28.000	34.000
	Min of Value	0.000	0.000	4.000		64.000	0.000	5.000	6.000	8.000
	Average of Value	**1.000**	0.200	86.400		204.800	0.200	11.800	15.600	19.600

Fig. 4.19: Differences between metric tools for project jTcGUI [com08]

comments. Analyst4j calculated WMC 17 and NOM 6. The values can be explained by counting all methods excluding constructors. VizzAnalyzer got 13, because it does not include the constructor.

4.5 Usability

In this section, the installation routine and the usability of each tool is discussed. Finally, the difficulties in interpreting the software metric results are described.

4.5.1 Sotoarc and Sotograph

Tool version is P4.1.2 release date 110624. Sotoarc and Sotograph can be run on a regular development machine. A computer with 512 MB RAM and a 1500 MHz processor is adequate to run the tools for a mid-sized software system (up to 1 million LOC). For larger software systems, one GB of RAM is recommended. A dedicated database server machine is not required.

The usage of Sotoarc and Sotograph is not self-explanatory. You have to load a java or c# project into Sotoarc. You also have to specify directories where class- and jar-files are. The first step in analyzing an unknown software system is to gain an overview of its structure. You see the packages of the project, but the use of the overview is unclear at this point. After consulting the Sotoarc‘s hands-on, it was clear to define the architecture and move each class to a specific layer. In the architecture view, you can see the dependencies and cycles between the files. You can switch to Sotograph for metric calculation. Hello2morrow also provides a Sotograph hands-on and then you get a better overview. During my work for the Infineon IT-company, the office received a Sotoarc and Sotograph training. After that training, it was easier to use it, but it takes a lot of time to analyze the architecture and evaluate the provided metrics. The conclusion

for these two tools is that they are useful, but you have to invest a lot of time to know all of their functionality.

4.5.2 Visual Studio 2010

The tool for metric calculation was already included in Microsoft Visual Studio 2010, version 10.0.30319.1. There was no extra installation necessary. It calculates only four metrics, by default. The usage was easy, after clicking in the menu bar "Analyze", you get a view of the calculated metrics. In the results there are no explanations about allowed ranges, or why this metric is violated. First, it is important to know what these metrics are calculating.

4.5.3 Analyst4j

Analyst4j was tested in version 1.5.0. For testing, I downloaded it in a standalone-version, otherwise you have to get a license for using full functionality. The installation and usage of Analyst4j was easy as an user of Eclipse programming environment. Right click on the project, Analyst4j and analyze; you get the metric calculation results. The results contain only average values. In this case you have to know what you are measuring and how to interpret these results.

4.5.4 VizzMaintenance

VizzMaintenance was tested in version 2.0.4.201002180142. The installation was uncomplicated. The plug-in had to be integrated in Eclipse and it was ready to use. A disadvantage is that there are also no explanations of calculated metrics and the reason metrics are violated are not explained.

4.6 Chapter summary

As a result of my thesis and [com08], calculation of software metrics is dependent on the tool. The knowledge and interpretation of these metrics are preconditions for measuring. Nevertheless, hello2morrow provided a lot of functionality with Sotoarc and Sotograph. Otherwise, VS 2010 includes only a small set of metrics into the framework. Every developer can check metrics regularly without any additional tool. Analyst4j and VizzMaintenance were simple to install, but interpretation without any hints is very difficult. It is not recommended for people without any metric experience.

5 Conclusion and future work

5.1 Conclusion

Software quality is a rather abstract term. Developing projects is not writing code anymore, it contains a lot of other disciplines to consider. Coding guidelines, testing, code review and measuring quality with software metrics are some key points. Each company has to define some golden rules for developing projects and that part is challenging. How can anyone guarantee high software quality without measuring it? One way is calculating static software metrics. For this master thesis I evaluated two different projects, DND (c#) and iTec (java) from Infineon. Before measuring, I interviewed 14 employees, involving team leaders and members for these two projects. Simple by answering a questionnaire, a major problem appeared: complexity. Next, I had to calculate a small set about 5 to 8 metrics, which could be useful for every project and which would be easy to understand and calculable for developers. The main problem with metrics is that you can not generalize one metric set for all projects. It is very specific for each project, because software criteria are different. The challenge is to find a set of metrics, independent from software criteria, yet helpful as an indicator for problems. For this reason, I picked out the most famous metrics from Chidamber and Kemerer, the CK-metrics. They cover inheritance, complexity, coupling and cohesion issues. If the inheritance path is too deep, this leads to confusing the developers and also make testing harder. Coupling and cohesion lead to worsening the maintainability and reusablity. Complexity, in general, makes it hard to understand the specific software part. The next task was to find a tool that would measures these sets of metrics. At the office, Sotoarc and Sotograph from hello2morrow were available for checking this software criteria. Sotoarc is a tool for checking the software architecture, so it is necessary to define software layers for each project. If there are already some layers, you have to check each one to find out if their functionality belongs together or has become chaotic. Both projects have a high cyclomatic complexity, which is an indicator for bad layer definition, as you can not change something in one class without side effects. For evaluating architectural problems, only two hours are scheduled. In these two hours using Sotoarc, you have to define an architecture and identify architectural violations in your project. If you have found violations, a solution could be adopting the previously designed architecture in small steps. The main focus was on finding metrics to improve the actual situation. Sotograph calculates static code metrics. It is an advantage to list all the metrics, that are violated in the project. It is remarkable that the allowed metric ranges are shown and the meaning of this particular metric is explained in the tool. With some help from a consultant of hello2morrow, we have defined a small set of metrics. Every developer has access to these tools and can regularly check how these metrics are changed over time.

5.2 Future work

The evaluation of these two projects has to be done for a longer period of time, in order to see if it has an effect on software quality of the whole project. It would be also interesting to see if the found set of metrics can be done by the daily build for C# and Java projects. Another step could be regularly performing code reviews every three weeks. This could improve the understanding and structure of a specific software. A focus should be set on testing and code coverage.

List of Figures

1.1 Techniques for improving software quality . . . 2

2.1 Internal and external quality criteria . . . 5
2.2 The cost of a defect in the software lifecycle [Lev11] . . . 8
2.3 Waterfall model . . . 18
2.4 Scrum skeleton . . . 19
2.5 Singleton Pattern . . . 23
2.6 Observer Pattern . . . 25
2.7 DND layer architecture . . . 28
2.8 Error discovery rates [DWR08] . . . 29
2.9 Testing techniques through software life cycle . . . 30
2.10 Testing pyramid . . . 34

3.1 States of development . . . 38
3.2 Overview pyramid . . . 46

4.1 Soto platform . . . 62
4.2 Soto integration . . . 62
4.3 Different modules in Sotoarc . . . 63
4.4 Different modules in your system . . . 64
4.5 Architectural view . . . 65
4.6 Show only violated arcs . . . 66
4.7 Focus on dependencies . . . 66
4.8 Tool tip about layer violation . . . 67
4.9 File dependence visualized in a graph . . . 67
4.10 Cyclic dependencies visualized in a graph . . . 68
4.11 Project migration in Sotoarc . . . 69

4.12 Trendmetric in Sotograph 70
4.13 Trend chart 70
4.14 Sotograph explanation 71
4.15 Order of metrics in Sotograph 72
4.16 Violated metrics 72
4.17 Code Coverage 79
4.18 Tools and metrics in evaluation [com08] 80
4.19 Differences between metric tools for project jTcGUI [com08] 81

Bibliography

[Ari13] Arisa. Vizzmaintenance 2.0, 2013. `http://www.arisa.se`.

[BD08] Manfred Bundschuh and Carol Dekkers. *The IT Measurement Compendium - Estimating and Benchmarking Success with Functional Size Measurement*. Springer-Verlag Berlin Heidelberg, 2008. ISBN 978-3540681878.

[BLL04] W.H.C. Bassetti, William E. Lewis, and Lewis E. Lewis. *Software Testing and Continuous Quality Improvement*. Auerbach Publisher Inc., 2004.

[BMRS96] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996. ISBN 978-0471958697.

[Bre07] Kadhim M. Breesam. Metrics for object-oriented design focusing on class inheritance metrics. In *2nd International Conference on Dependability of Computer Systems*, pages 231 – 237, 2007.

[Bur03] Ilene Burnstein. *Practical Software Testing*. Springer, 2003. ISBN 978-0387951317.

[BWDP00] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 2000.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metric suite for object oriented design. In *IEEE Transactions on Software Engineering*, pages 476–493, 1994.

[Cod13] CodeSwat. Analyst4j, 2013. `http://www.codeswat.com`.

[com08] *Comparing Software Metrics Tools*, ISSTA '08, New York, NY, USA, 2008. ACM.

[CS09] Kuljit Kaur Chachal and Hardeep Singh. Metrics to study symptoms of bad software designs. In *ACM SIGSOFT Software Engineering Notes*, volume 34, pages 1 – 4, January 2009.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[DeM86] Tom DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall, 1986. 978-0131717114.

[Doo11] John Dooley. *Software Development and Professional Practice*. Apress, 2011. ISBN 978-1430238010.

[DWR08] Alan Dennis, Barbara Haley Wixom, and Roberta M. Roth. *System Analysis and Design*. John Wiley & Sons, 2008.

[Fab13] Szczepan Faber. Mockito, 2013. `http://code.google.com/p/mockito/`.

[FFB+04] E. Freemann, E. Freeman, B. Bates, K. Sierra, and M. Loukides. *Head First Design Patterns*. O'Reilly Media, 2004. ISBN 978-0596007126.

[GGM11] Yossi Gil, Maayan Goldstein, and Dany Moshkovich. How much information do software metrics contain?, 2011.

[GHJ+05] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, and Craig Larman. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 2005. ISBN 978-1405837309.

[hel08] hello2morrow. Sotograph tutorial 3.4. 2008.

[hel10] hello2morrow. Sotoarc handson overview. 2010.

[HK13] Johan Haleby and Jan Kronquist. Powermockito, 2013. `http://code.google.com/p/powermock/`.

[Hof09] Dirk W. Hoffmann. *Software-Qualität*. Springer, Berlin 1. Auflage, 2009. ISBN 978-3540763222.

[Inf00] Infineon. Sem-i development manual - internal manual, 2000.

[ISO01] ISO. *Software Engineering - Product Quality - Part 1: Quality Model*. Internal Organization for Standardization, 2001. ISO/IEC 9126-1.

[ISO03a] ISO. *Software Engineering - Product Quality - Part 2: External Metrics*. Internal Organization for Standardization, 2003. ISO/IEC 9126-2.

[ISO03b] ISO. *Software Engineering - Product Quality - Part 3: Internal Metrics*. Internal Organization for Standardization, 2003. ISO/IEC 9126-3.

[KB11] Usha Kamari and Sucheta Bhasin. Application of object-oriented metrics to c++ and java: A comparative study. In *ACM SIGSOFT Software Engineering Newsletter Volume 36*, March 2011.

[KMB04] Cem Kaner, Senior Member, and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *METRICS 2004, IEEE CS*, pages 1 – 12. Press, 2004.

[KP76] B. W. Kernighan and P. J. Plauger. Software tools. *SIGSOFT Softw. Eng. Notes*, 1(1):15–20, May 1976.

[KS10] N. Satheesh Kumar and S. Subashni. *Software Testing Using Visual Studio 2010*. Packt Publishing, 2010. ISBN 978-1849681407.

[Lev11] Jeff Levinson. *Software Testing with Visual Studio 2010*. Addison-Wesley Professional, 2011. ISBN 978-0321734488.

[LMD10] Michele Lanza, Radu Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2010. ISBN 978-3642063749.

[Mac13] Virtual Machinery. WMC, CBO, RFC, LCOM, DIT, NOC - 'The Chidamber and Kemerer Metrics'. URL, 2013. http://www.virtualmachinery.com.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall International, 2008. ISBN 978-0132350884.

[mic11] Microsoft coding guidelines, 2011. http://msdn.microsoft.com/.

[MvZ11] Dietmar Menges and Alexander v. Zitzewitz. Architectural quality rules. 2011.

[Nag04] Nachiappan Nagappan. Toward a software testing and reliability early warning (strew) metric suite. In *ICSE'04*, pages 60–62, 2004.

[Pei11] Bernhard Peischl, editor. *ASQT 2011 - Ausgewählte Beiträge zur Anwenderkonferenz für Softwarequalität und Test 2011*. OCG, 2011. ISBN 978-3854032830.

[PMD11] PMD. Java source code scanner, 2011. http://pmd.sourceforge.net.

[PR10] Joshi Padmaja and Joshi K. Rushikesh. Quality analyis of object oriented cohesion metrics. In *Seventh International Conference on the Quality of Information and Communications Technology*, pages 319–324, 2010.

[PY07] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[Sch04] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004. ISBN 978-0735619937.

[SS07] K. Stroggylos and D. Spinellis. Refactoring – does it improve software quality? In *Fifth International Workshop on Software Quality*, 2007.

[SSB10] Harry M. Sneed, Richard Seidl, and Manfred Baumgartner. *Software in Zahlen: Die Vermessung von Applikationen*. Carl Hanser Verlag GmbH & CO. KG, 2010. ISBN 978-3446421752.

[Wei98] Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House, 1998. ISBN 978-0932633422.

[Wil11] Hermann Will. Metriken aus der praxis für die praxis. ASQT 2011 - 9. Anwenderkonferenz für Softwarequalität und Test, 2011.

[YK10] A. Yadav and R. A. Khan. Does coupling really affect complexity. In *International Conference on Computer and Communication Technology (ICCCT) 2010*, pages 583 – 588, 2010. ISBN 978-1424490332.

[Zel06] Marvin V. Zelkowitz. *Advances in Computers - Quality Software Development*, volume Volume 66. Academic Pr. Inc, 2006.