Fachhochschule
Kaiserslautern
University of Applied Sciences

Informatik und
Mikrosystemtechnik
Zweibrücken

Studiengang

Medieninformatik

Bachelorarbeit

Software Quality Improvement in the OMC Team at CERN

vorgelegt von

Viktor Maier

28. Februar 2014

Betreuung: Prof. Dr. Jörg Hettel

Zweitkorrektur: Prof. Dr.-Ing. Wilhelm Meier

Abstract

Physicists use self-written software as a tool to fulfill their tasks and often the developed software is used for several years or even decades. If a software product lives for a long time, it has to be changed and adapted to external influences. This implies that the source code has to be read, understood and modified.
The same applies to the software of the Optics Measurements and Corrections (OMC) team at CERN. Their task is to track, analyze and correct the beams in the LHC and other accelerators. To solve this task, they revert to a self-written software base with more than 150,000 physical lines of code. The base is subject to continuous changes as well.
Their software does its job and is effective, but runs regrettably not efficient because some parts of the source code are in a bad shape and has a low quality. The implementation could be faster and more memory efficient. In addition it is difficult to read and understand the code. Source code files and functions are too big and identifiers do not reveal the intention of the coder and are meaningless or even misleading. Thus it isn't easy to modify and extend the code.
However, this is also a common problem in software engineering. The improving of productivity and quality in software development is still a field of research.
This thesis is about the improvement of the software quality in the OMC team by adopting several techniques from professional software development. A more professional development environment was created and the quality of the existing software base was improved. Further, the members of the OMC team were sensitized to write directly clean code, to practice pair programming and to review each others source code.
To measure software quality, a tailored quality model was defined. Code modifications trigger automatically an analysis tool to create metrics, based on the quality model, and present them in a dashboard. A trend analysis of this data will show if future developments will increase the software quality. In case that it becomes worse, appropriate steps can be initiated to refactor the corresponding source code parts.
In addition a best practices guide was developed in the course of this thesis, which can be used to improve the software quality in any other team.

Kurzfassung

German version of the previous abstract.[1]

Physiker benutzen für ihre Aufgaben selbst geschriebene Software als Werkzeug und oft wird die entwickelte Software anschließend für mehrere Jahre oder sogar Jahrzehnte verwendet. Wird ein Softwareprodukt für einen längeren Zeitraum eingesetzt, muss es aufgrund externer Einflüsse geändert und angepasst werden. Dies hat zur Folge, dass der Quelltext gelesen, verstanden und modifiziert werden muss.
Das gleiche gilt für die Software des Optics Measurements and Corrections (OMC) Team am CERN. Ihre Aufgabe ist das Verfolgen, Analysieren und Korrigieren der Beams im LHC und in anderen Beschleunigern. Um diese Aufgabe zu lösen, greifen sie auf eine selbst geschriebene Softwarebasis mit mehr als 150.000 physikalischen Codezeilen zurück. Sie unterliegt ebenfalls ständigen Änderungen.
Ihre Software erledigt ihre Aufgabe und ist effektiv, aber läuft bedauerlicherweise nicht effizient, da einige Teile des Quelltextes in schlechter Form sind und eine niedrige Qualität haben. Die Implementierung könnte schneller laufen und speichereffizienter sein. Zusätzlich ist es schwer, den Code zu lesen und zu verstehen. Quelltextdateien und Funktionen sind zu groß und Bezeichner enthüllen nicht die Absicht des Programmierers und sind bedeutungslos oder sogar irreführend. Deswegen ist es nicht leicht, den Quelltext zu modifizieren und zu erweitern.
Dies ist ebenfalls ein geläufiges Problem in Software Engineering. Die Verbesserung der Produktivität und Qualität in der Softwareentwicklung ist immer noch ein Forschungsgebiet.
Das Thema der Thesis ist die Verbesserung der Softwarequalität im OMC Team durch die Einführung verschiedener Techniken aus der professionellen Softwareentwicklung. Eine professionellere Entwicklungsumgebung wurde geschaffen und die Qualität der existierenden Softwarebasis wurde verbessert. Desweiteren wurden die Mitglieder sensibilisiert, sauberen Code zu schreiben, Pair Programming zu praktizieren und gegenseitig Quelltext zu überprüfen. Um Softwarequalität zu messen, wurde ein angepasstes Qualitätsmodell

[1] German version required for formal reasons. Mostly identical to the previous abstract.

definiert. Quelltextmodifizierungen führen automatisch ein Analysewerkzeug aus, um Metriken zu messen, welche auf dem Qualitätsmodell basieren und in einem Dashboard angezeigt werden. Eine Trendanalyse dieser Daten wird zeigen, ob zukünftige Entwicklungen die Softwarequalität seigern. Im Falle, dass es schlechter wird, können entsprechende Gegenmaßnahmen eingeleitet werden, die die betreffenden Quelltextteile anpassen.
Zusätzlich wurde im Laufe dieser Thesis ein Best Practices Guide entwickelt, welcher von einem beliebigem anderen Team benutzt werden kann, um die Softwarequalität zu verbessern.

Contents

Abstract **iii**

Kurzfassung **v**

Acronyms **xi**

1 Introduction **1**
1.1 CERN . . . 1
1.1.1 BE-ABP . . . 3
1.1.2 OMC team . . . 4
1.2 Motivation . . . 4
1.3 Goal . . . 5
1.4 Overview . . . 6

2 Software Quality **7**
2.1 ISO/IEC 25010:2011 . . . 9
2.1.1 Product Quality Model . . . 9
2.2 Measuring Software Quality . . . 11
2.2.1 Code Quality Metrics . . . 12
2.2.2 Goal Question Metric . . . 13
2.2.3 Quality Attribute Scenarios . . . 14
2.3 Clean Code . . . 16

3 Software Development Models **19**
3.1 Heavyweight Models . . . 20
3.1.1 Waterfall . . . 20
3.1.2 Incremental . . . 21
3.1.3 Spiral . . . 23
3.2 Lightweight Models . . . 24
3.2.1 Scrum . . . 25
3.2.2 eXtreme Programming - XP . . . 27
3.2.3 Kanban . . . 29
3.3 Conclusion . . . 31

4 Software Testing **33**
4.1 Test Strategies . . . 34
4.1.1 Black-Box Testing . . . 34
4.1.2 White-Box Testing . . . 34
4.2 Static Testing . . . 34
4.2.1 Desk Checking . . . 35
4.2.2 Inspections . . . 35
4.2.3 Walk-Throughs . . . 36
4.3 Dynamic Testing . . . 36
4.3.1 Functional Testing . . . 36
4.3.2 Structural (Non-functional) Testing . . . 36
4.3.3 Logic Coverage Testing . . . 37
4.4 Test-Driven Development(TDD) . . . 39
4.5 Conclusion . . . 40

5 Current Situation **41**
5.1 Surveys . . . 41
5.1.1 OMC - Survey . . . 41
5.1.2 BE-ABP - Survey . . . 42
5.2 SWOT Analysis . . . 44
5.3 Own Experience . . . 45

6 Realizations towards better Software Quality **47**
6.1 Refactoring and Cleaning . . . 47
6.1.1 GetLLM . . . 47
6.1.2 Other Main Programs . . . 49
6.1.3 Robustness . . . 49
6.1.4 Cleaning Beta-Beat.src . . . 50
6.1.5 Adding Docstrings . . . 51
6.2 OMC Wiki . . . 51
6.3 Improvement of the Beta-Beat-GUI . . . 54
6.3.1 CorrectionSelection and KnobDialog . . . 54
6.3.2 Frequency Chart . . . 55
6.3.3 Action vs. Tune Chart . . . 57
6.4 Practices to improve the Software Quality . . . 58
6.4.1 Automated Tests . . . 58
6.4.2 GitHub Issue Tracker . . . 58
6.4.3 Coding Lectures and Best Practices Guide . . . 59
6.5 Continuous Quality Measuring . . . 59
6.5.1 OMC Assessment Model . . . 59
6.5.2 SonarQube vs. ConQAT . . . 61
6.5.3 Implementation of Continuous Quality Measuring . . . 62
6.6 Comparison of Beta-Beat.src . . . 65

7 Conclusion **67**

A OMC software **69**
A.1 Beta-Beat.src 69
A.2 Beta-Beat-GUI 70

B Best Practices Guide **77**
B.1 Clean Code 77
B.1.1 Names 78
B.1.2 Functions 80
B.1.3 Comments 80
B.1.4 The Stepdown Rule 82
B.1.5 The Boy Scout Rule 83
B.1.6 Red-Green-Refactor (TDD mantra) 83
B.1.7 Clean Build 85
B.1.8 Type Rich Programming 85
B.2 Review 86
B.3 Pair Programming 87
B.4 Tools 87
B.5 Software Development Process 89
B.6 Algorithm Improvement 90

C Overview CD-ROM **95**

Glossary **97**

Bibliography **99**

Acronyms

AFS Andrew File System. 50, *Glossary:* AFS

API Application programming interface. 10, 55, *Glossary:* API

BPM Beam Position Monitor. 4, 47, 55, 71, *Glossary:* BPM

ConQAT Continuous Quality Assessment Toolkit. 59, 61–66, 68, *Glossary:* ConQAT

GetLLM Get Linear Lattice functions and More. 47, 48, 50, 57, 71, 76, 79

GIL Global Interpreter Lock. 95, *Glossary:* GIL

GQM Goal Question Metric. 13, 15, *Glossary:* ConQAT

HTML HyperText Markup Language. 51–54, 61, 64, 65, *Glossary:* HTML

IDE Integrated Development Environment. 43, 51, 67, 89, 90, *Glossary:* IDE

LSA LHC Software Architecture. 55, 77, *Glossary:* LSA

MAD Methodical Accelerator Design. 4, 71, *Glossary:* MAD

PLOC Physical Lines of Code. 12, 47, 48, 60, 62, 64–66, 80, 100, *Glossary:* PLOC

RMI Remote Method Invocation. 55, *Glossary:* RMI

SDDS Self Describing Data Sets. 71, *Glossary:* SDDS

SLOC Source Lines of Code. 4, 48, 49, 66, 84, *Glossary:* SLOC

TDD Test-Driven Development. 39, 40, 43, 59, 68, 85, 86, 91

TFS Table File System. 47, 56, 71, *Glossary:* TFS

UML Unified Modeling Language. 91, *Glossary:* UML

VCS Version Control System. 83, 90, *Glossary:* VCS

WYSIWYG What You See Is What You Get. 54, *Glossary:* WYSIWYG

Chapter 1

Introduction

1.1 CERN

To unite the European scientists and share the costs of nuclear physics facilities, a council was established in 1952 to create a European laboratory. The name for this council was "Conseil Européen pour la Recherche Nucléaire". CERN is the acronym. The laboratory European Organization for Nuclear Research was founded in 1954. Although it has an official name the acronym CERN as name retained until today.
CERN is located at the swiss-french border near Geneva and has 21 member states. The main research area is particle physics. The most powerful particle accelerators and detectors are used to fulfill the research with the main goal to gain knowledge. Profit is not an issue.
The most known particle accelerator at CERN is the Large Hadron Collider (LHC) which is a circular accelerator and located underground. The depth varies between 175 m and 50m. It has a circumstance of 27 km, accelerates protons or ions, which are hadrons, and collides the particles at four interaction points. These points are the different experiments: A Large Ion Collider Experiment (ALICE), A Toroidal LHC Apparatus (ATLAS), Compact Muon Solenoid (CMS) and the Large Hadron Collider beauty (LHCb) experiment. The accelerator has two vacuum pipes in which one beam is accelerated clockwise and one beam anticlockwise. Each beam can be accelerated to an energy of 7 TeV which results in a collision energy of 14 TeV.
The LHC is the most famous accelerator but not the only one at CERN. Figure 1.1 shows the whole accelerator complex while figure 1.2 describes the accelerators and corresponding experiments.

The source of the protons is a bottle of hydrogen gas. Hydrogen atoms consists of one proton and one electron. The electron is stripped by an electric field. The linear accelerator Linac 2 accelerates the protons and injects the beam into the Booster, followed by the Proton Synchrotron (PS) and

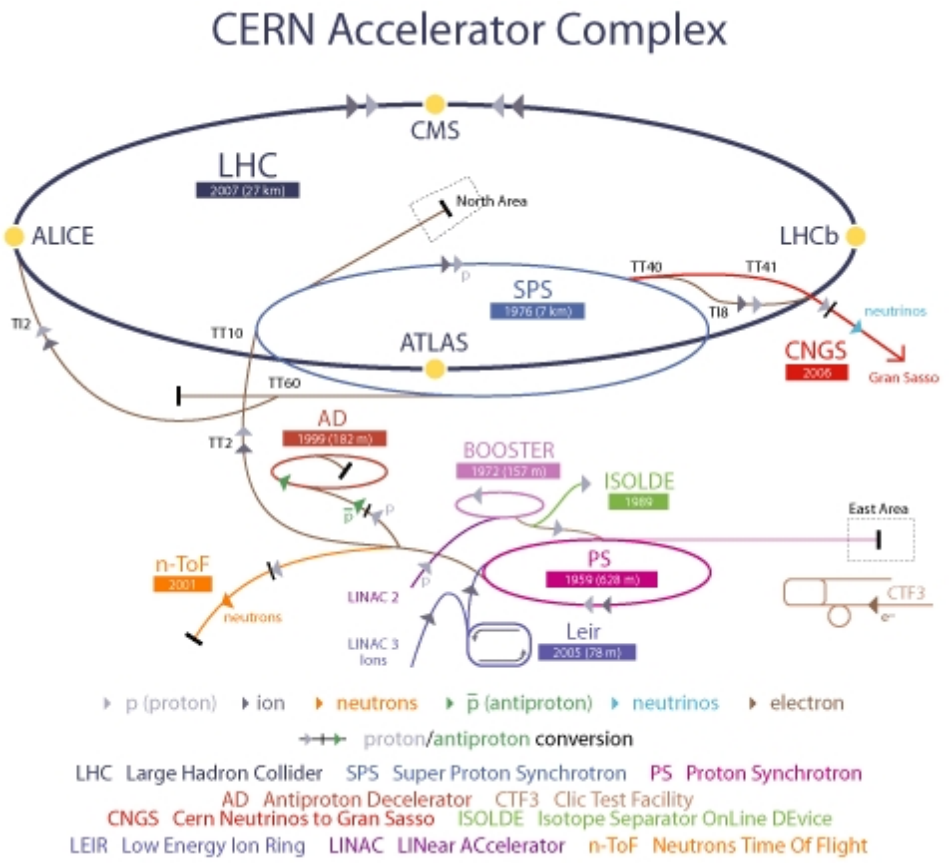

Figure 1.1: CERN's accelerator complex[1]

then the Super Proton Synchrotron (SPS). The SPS injects the beams into the two vacuum pipes in the LHC where the beams are accelerated up to 7 TeV and collide in one of the four detectors. A collision produces new particles which together have the same energy as the collision energy. The produced particles are counted, tracked and characterized to reconstruct the collision. All the detectors are built differently to analyze different particles. Since 14th of February 2013 CERN is in the first Long Shutdown (LS1) to maintain and upgrade the accelerator chain until mid 2014. In the first run from 2010 - 2012 the beams were accelerated up to 4 TeV. The LHC will run at its design energy of 7 TeV per beam in 2015. The second Long Shutdown (LS2) is planned from around 2018 until 2019.
LHC's first run period was a success. The discovery of the Higgs boson, the last missing part in the Standard Model, was announced on fourth July 2012. Peter W. Higgs and François Englert received the Nobel prize in physics 2013

[1] https://espace.cern.ch/be-dep/OP/default.aspx, visited on 16th Feb 2014.

Accelerator	Experiments	Overview
LINAC2		accelerates proton from the source and sends them to the booster (PSB)
LINAC3		accelerates ions from the source and sends them to LEIR
PSB		accelerates protons and sends to the PS and ISOLDE
PS	EA	East Area - various experiments
	nTOF	neutron time-of-flight
	DIRAC	to observe and then measure lifetimes of Muons and Kaons
	CLOUD	to study possible links between cosmic rays and cloud formation
ISOLDE		to produce a range of isotopes for research
AD	ALPHA	to make, capture and study atoms of antihydrogen and compare these with hydrogen atoms
	ASACUSA	to compare anti-protons and protons using antiprotonic helium
	ATRAP	to compare hydrogen atoms with their antimatter equivalents
LEIR		accelerates ions and sends them to the PS
SPS	NA	North Area - various experiments
	CNGS	to send muon neutrinos to to the Gran Sasso National Laboratory in Italy
	COMPASS	to study how elementary quarks and gluons work together to give particles we observe
LHC	CMS	to search for the Higgs boson, extra dimensions, and particles that could make up dark matter
	ATLAS	to search for the Higgs boson, extra dimensions, and particles that could make up dark matter
	LHCb	to understand why we live in a Universe composed almost entirely of matter, but no antimatter
	ALICE	to study a state of matter known as quark-gluon plasma
	TOTEM	to measure the size of the proton and also monitor the LHC's luminosity
	LHCf	to simulate cosmic rays to interpret and calibrate large-scale cosmic-ray experiments

Figure 1.2: Overview of the accelerators and experiments[2]

"for the theoretical discovery of a mechanism that contributes to our understanding of the origin of mass of subatomic particles, and which recently was confirmed through the discovery of the predicted fundamental particle, by the ATLAS and CMS experiments at CERN's Large Hadron Collider." [3]

Although one main goal was reached, there are even more questions to solve: "*Why is gravity so weak? Why is there more matter than antimatter in the universe? Is there more exotic physics waiting to be discovered at higher energies? Will we discover evidence for a theory called supersymmetry at the LHC? Or understand the Higgs boson that gives particles mass?*"[4] [5]

1.1.1 BE-ABP

The Accelerators and Beams Physics (ABP) Group belongs to the Beams (BE) Department and is responsible for the beam physics in the whole accelerator chain. It organizes the machine development and contributes to operation supervision and machine together with detector alignment metrology.

[2] https://espace.cern.ch/be-dep/OP/default.aspx, visited on 16th Feb 2014.

[3] http://home.web.cern.ch/about/updates/2013/10/CERN-congratulates-Englert-and-Higgs-on-Nobel-in-physics visited on 20th Dec 2013.

[4] http://home.web.cern.ch/about/physics visited on 19th Dec 2013.

[5] http://timeline.web.cern.ch/timelines/The-history-of-CERN visited on 19th Dec 2013.

This includes research plus development activities.
Further the ABP group is in charge of the development and operation of the hadron sources along with the hadron linear accelerators.
Teaching activities related to accelerator physics is a further task of the group.[6]

1.1.2 OMC team

The Optics Measurement and Corrections (OMC) team, part of the BE-ABP group, tracks, measures, analyses and corrects the beams in the LHC. The LHC accelerates 2808 bunches with 100 billion protons per bunch. One bunch is only a few centimeters long and thinner than a hair. When two bunches meet in the detectors, only 20 proton-proton collisions occur. Because the bunches travel about 11000 times around the accelerator per second, it is possible to generate 600 million collisions per second.[7] The corrections of the beam contribute to more collisions in the detectors.
The Beam Position Monitor (BPM) is the eye of an accelerator. A BPM determines the beam position. There are about 1000 BPMs in the LHC. The collected data is used to calculate several characteristics of the beam. Correction values can be calculated and then applied to the magnet strengths in the LHC by comparing the analyzed data with a model, which is calculated by the accelerator simulator Methodical Accelerator Design (MAD).
To solve the correction task, a self-written software base is used. Most of the programs are written in Python, a currently popular programming language in science. Further there is one program written in C++ which calls a procedure developed in Fortran. To simplify the correction process and to combine the invocation and the representation of data, a program with a graphical user interface (GUI), the Beta-Beat-GUI, was written in Java.
It is very important to highlight that traditional roles from software development are not present. The physicists write programs to fulfill their tasks and so there are no other users, customers or project managers. They are developers as well as users.

1.2 Motivation

In course of the research physicists encounter problems or phenomena which can be described by using mathematical equations and formulas. The amount of data related to the formulas is so enormous that it is not possible to process them manually by hand. Computers and algorithms are necessary to solve the equations.
The OMC team has already about 111,000 Source lines of code (SLOC),

[6] https://espace.cern.ch/be-dep/abp/default.aspx, visited on 19th Dec 2013.

[7] http://cds.cern.ch/record/1165534/files/CERN-Brochure-2009-003-Eng.pdf p.30, visited on 20th Dec 2013.

88,000 in Python, 1,000 in C++, 4,000 in Fortran and 17,000 in Java. CERN has even more and the amount of source code grows every day. Here is also very old code written in obsolete programming languages. The Fortran source code in the OMC team was written in 1996.
Probably the most source code was written by physicists. They are able to transform mathematical equations into a running program but they are not educated in software engineering. Thus it is very difficult to read their source code, track the logic and find out what it actually does.
But these properties are very important since the most source code is dynamic, it has to be maintained, bug-fixed, adapted or extended. Further the most programs related to memory storage and execution time are not efficient. This is not crucial for every program but the improvement of important and long running programs could save hours of time.
Especially during LS1, a lot of changes have to be adapted. Often changes and fixes deteriorate even the source code because hacks and quick fixes, instead of rewriting and refactoring, are applied to the source code. This behavior combined with several iterations can rot software and destroy whole companies[Mar09, p. 3].
Further the situation can be compared with elder software development. After creating software for several decades there are very mighty tools and techniques which support the developers but are not used here. Discussions with physicists and computer scientists outside of the OMC team revealed that this is common in teams of physicists.

1.3 Goal

The goal is to find a way to improve software quality of future developments and the adaptation of structured software development and software management tools for the OMC team with considering its specific needs.
For sure it is not possible to adapt usual proposed practices since the main purpose of the OMC team is not to produce software. Writing software is more like a tool to fulfill tasks.
Probably the hardest part is the enforcement of the changes. Every one has more or less his own work style. Changes would slow down the progress first since associated processes and tools have to be learned. To convince everybody that this effort makes the team much more efficient is an additional task.
The goal is not to adapt development procedures and work styles from professional software companies but pick several valuable and simple techniques and ideas from this field to improve the software quality of the OMC team. The results can be adapted to all similar teams at CERN and even outside of CERN.

1.4 Overview

The thesis is structured as follows. Chapter 2 introduces the term quality in general and correlate it to software by using the standard ISO 25010:2011 afterwards. In addition, how to measure software is shown and the importance of clean code for software quality is discussed. Chapter 3 introduces the most known software development models, separated in heavyweight and lightweight, and explains the connection to software quality. Chapter4 delivers the foundation of software testing. The basic testing strategies and methodologies are introduced. Test-driven development is explained in more detail and how it can improve software quality. Chapter 5 illustrates the state before own contributions and improvements. A SWOT analysis evaluates the team. The information is based on own experience and surveys in the BE-ABP group. Chapter 6 is the main part of the thesis and serves also as internship report. It describes all the work, which is done to improve the software quality of the team. Chapter 7 discusses the results and further tasks as well as further improvements.

Chapter 2

Software Quality

The aim of this thesis is to improve the software quality. This chapter introduces a clear definition of software quality and how to measure it. As source mainly served Stefan Wagners introduction to software product quality control[Wag13]
Quality is a positive notion of a person. It depends on the point of view and is subjective. For example the quality of a software product can be perceived differently from the point of view of the developers and the users. Then every user could assess the quality differently. One could be fascinated, whereby another one could be disappointed.
The ISO/IEC/IEEE 24765:2010 has six alternative definitions[Wag13, p. 6]:

1. *The degree to which a system, component or process meets specified requirements*
2. *The ability of a product, service, system, component or process to meet customer or user needs, expectations or requirements*
3. *The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs*
4. *Conformity to user expectations, conformity to user requirements, customer satisfaction, reliability and level of defects present*
5. *The degree to which a set of inherent characteristics fulfils requirements*
6. *The degree to which a system, component or process meets customer or user needs or expectations*

The broadness shows the slightly different meanings but the term requirements and synonyms are in every definition. Hence an abstract definition is that quality fulfils requirements.

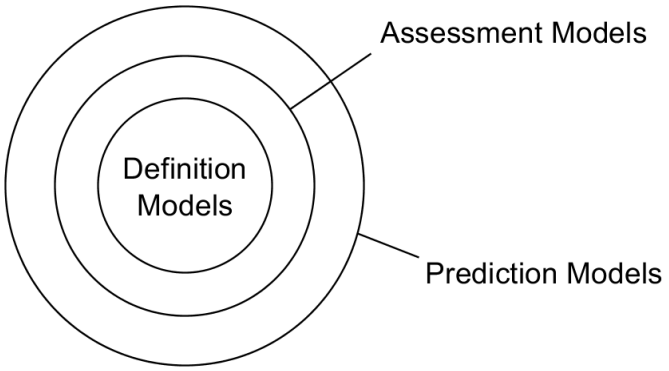

Figure 2.1: DAP classifications[1]

Software quality is defined by software quality models. This field was researched for decades and is still in research.
There are three classifications of models which have to be distinguished, see also figure 2.1:

Definition Models try to give the term software quality a detailed meaning. The usual approach is to divide the term in several characteristics(maintainability, usability, reliability...). An example is the standard ISO/IEC 25010:2011 which will be used here as well.

Assessment Models used to measure and evaluate the quality requirements. They are based on a definition model.

Prediction Models foresees a future state of the software by using an underlying assessment model and measured data together with a statistical method. An example is the number of defects. These defects are mean times between failures, repair times and maintenance efforts.

The product quality model of the ISO/IEC 25010:2011, belonging to the category definition models, is used for the concrete definition. This standard revises the old one, ISO/IEC 9126. Software quality will be classified in multiple characteristics and sub-characteristics. By using them, software can be evaluated and the quality can be measured.
It is important to measure quality to control the improvement. "You can't control what you can't measure" is the famous sentence of Tom DeMarco, but he also mentioned that it is possible to control correlated things to improve implicitly software quality without measuring quality itself[DeM09].

[1]Reproduced picture is based on [Wag13, Fig. 2.1].

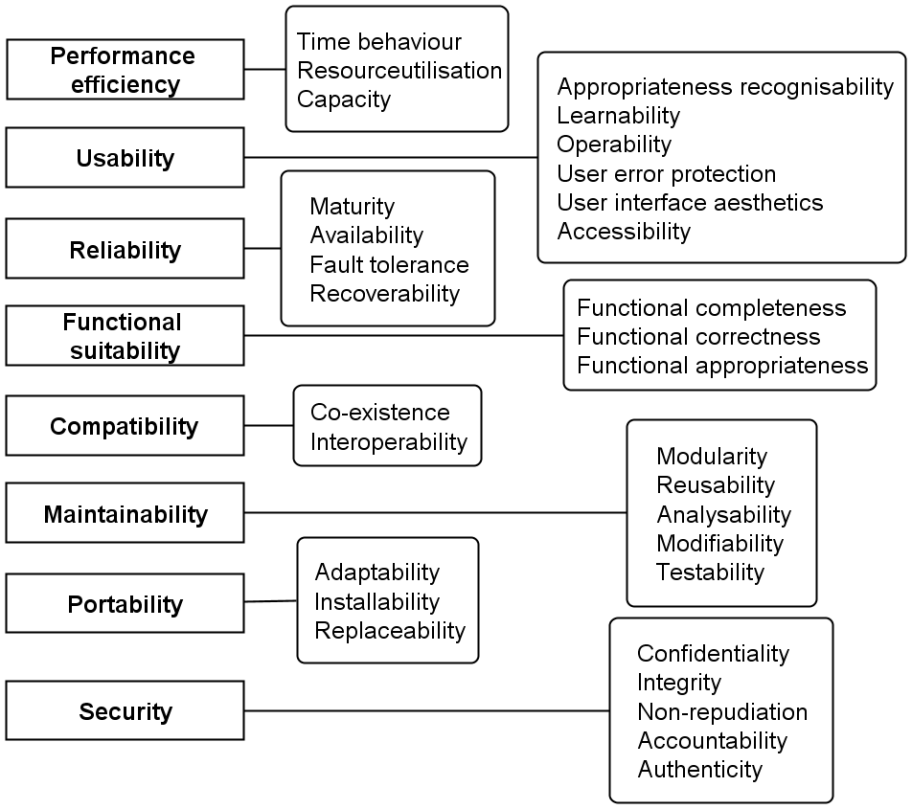

Figure 2.2: The product quality model of ISO/IEC 25010:2011[2]

2.1 ISO/IEC 25010:2011

The standard breaks software quality into smaller parts which can be described and measured. It contains the Product Quality Model and the Quality in Use Model. This section will omit the latter one since this model is interesting from the view of users which are not the developers.

2.1.1 Product Quality Model

Figure 2.2 shows the characteristics and sub-characteristics of the Product Quality Model.

Performance efficiency

It describes how well the execution, concerning time and resources, of the software is. The less time and resources, e.g. memory, a software need to fulfill a task the higher the quality.

[2]Reproduced picture is based on [Wag13, Fig. 2.9].

Since the power of computers increased enormously the last decades the explicit strive after performance is not needed or necessary in many cases. However, in many complex calculations and simulations which can take hours of execution time, the performance efficiency is a main quality factor.
After improving and refactoring of the OMC code the last years the performance is reasonable for the main software. Still there are a lot of possible improvements, especially in new software.

Usability

This describes the satisfaction of the user. It includes enjoying the use of the software by novel and advanced users and how they assess the user interface. This characteristic mainly comes from the point of view of users.
Since in this special case the software is created by the users themselves the usability plays a minor part. Further there is no software with an user interface except of the Beta-Beat-GUI, see appendix A.2 on p.70.

Reliability

Users often refer to reliability if they talk about quality. It describes how frequently the services and requirements are unavailable due to user errors, other internal and external errors or faults. If software can't be used at all, it is worse for the user than performance or usability issues, thus reliability is a very important characteristic.
Indeed this a big problem in the software of the OMC team. Many scripts crash and print a cryptic error message. For an example, see 6.1.3 on p. 49.

Functional suitability

This factor expresses that the functionality shall fulfill the users requirements and needs. It is based on the abstract definition of quality, satisfying requirements.
The OMC team needs software with functional completeness, correctness and appropriateness to do their tasks. Especially correctness is important since wrong correction would lead to worse optics and thus less collisions in the interaction points.

Compatibility

The ability to interact or exist with other software is understood as compatibility and is important for both, user and developer. If the compatibility is high, the user can use the software in combination with other software and services, e.g. a cloud service. Developers can add features by using further applications. An example of compatibility from the point of view of a developer is the existence of an API.
The OMC team needs a highly compatible software since the correction process depends on several programs which interact with each other by using the output data, see appendix A.1 on p.69.

Maintainability
The terms code quality and internal quality refers to this factor. It focuses not at the user but at the developer. Software changes continuously and so editing source code is mandatory. Maintainability describes how well the code can be analyzed, modified and tested. Modularity and reusability are sub-characteristics. A software is modular if it consists of several components which can be replaced. Such components are proper for reusing.
For the physicists it is a very important factor since the code has to be maintained if bugs or external changes occur or improvements planned for the software.

Portability
This characteristic expresses how well a software can be spread to other platforms or systems. The importance depends strongly on the product and target group. For example mobile applications should be portable to bring them to multiple platforms and reach more users.
Since the OMC code is written for own use the software does not have to be portable. The code is written to be able to run in the CERN Control Centre[3] in which the computers are running on Linux systems. However for developing purposes it is useful to have source code which is independent from the operating system.

Security
It describes how well protected the software is against malicious attacks. Further the ability of secrecy of data from users belongs to this factor. It was not a main characteristic in ISO/IEC 9126 but it became so important that it has changed in the new standard.
Security is for the OMC team immaterial since the data has no confidential content.

This thesis put emphasis on maintainability, functional suitability(especially correctness) , reliability and performance efficiency. The other factors are not very relevant for the OMC team.

2.2 Measuring Software Quality

It is easier to measure the characteristics than software quality directly. However, it is still not straightforward to measure the single factors. They need also to be divided and what actually should be measured and in which scale, depends on the product and its environment. Various programs could have very different values of the same measurement and still the same quality.

[3] http://public.web.cern.ch/public/en/spotlight/SpotlightCCC-en.html, visited on 15th Jan 2014.

After determining the single values with predetermined methods and scales for the characteristics, the values have to be combined. This is called aggregation. In theory the aggregation operator A is a function that yields the quality factor y for any n-tuple $(x_1, x_2, \ldots, x_n)$ where x_i is the measured value for characteristic i:

$$A(x_1, x_2, \ldots, x_n) = y$$

The choice of a proper aggregation operator depends strongly on the software product. A good decision is necessary since the aggregation operator has a strong influence on the results. For example the involving of security factors of an application in which security does not matter would falsify the result. As mentioned in the IEEE standard 1061 weighted sums are a possible implementation.

2.2.1 Code Quality Metrics

Code quality is the maintainability characteristic of the previous introduced ISO standard 25010:2011. This factor needs to be improved. An anecdote states that the only valid measurement respectively metric[4] for code quality is *WTFs/min*[5].
Below are just some of the many metrics related to source code:

Physical lines of code (PLOC)
: Physical lines of code (PLOC) correspond to the number of lines in the source file.

Source lines of code (SLOC)
: These are PLOC without comment and blank lines.

Methods per class
: Indicates how many methods a class has.

Blocks depth
: The number of nested blocks within a section of code, function or method.

Average depth
: The average of all measured block depths in a project.

Average statements per method
: States the average of statements for all measured methods or functions.

Cyclomatic (McCabe's) complexity
: It is the number of linearly independent paths through a graph, representing a section of source code.[6]

[4]Metric is the used synonym in software engineering for measure.
[5]http://www.osnews.com/story/19266/WTFs_m visited on 12th Jan 2012.
[6]For a full description, see http://en.wikipedia.org/wiki/Cyclomatic_complexity, visited on 20th Feb 2014.

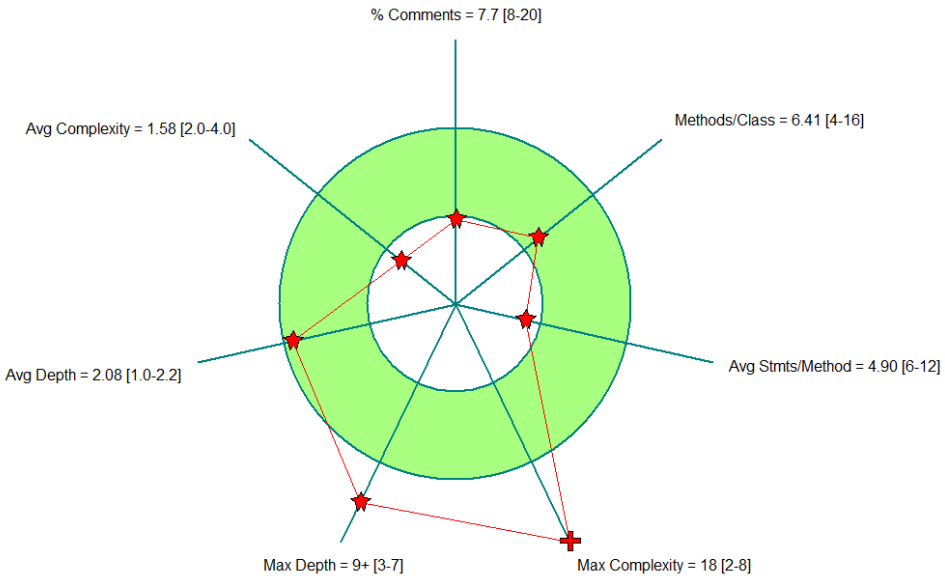

Figure 2.3: A Kiviat Metrics Graph

SourceMonitor[7] is a free tool to create these metrics. It supports several languages(C/C++, Java, etc.) but is only available for Windows. For Python source code, which is not covered by SourceMonitor, PyMetrics[8] can be used. These metrics can be used in a project to define code quality. After specifying min/max thresholds a Kiviat Metrics Graph can be created to visualize quality. Figure 2.3 shows an example of the Beta-Beat-GUI, see appendix A.2 on p.70, generated by SourceMonitor. In this example only the metrics *Methods/Class* and *average depth* are within the bounds.

The variety of free tools to measure source code facilitates the gathering of metrics. But if the measuring has no purpose, the metrics are useless. They receive only a value when a specific assessment model for the project was created and related to them. This model is generated with a purpose and thus gives the measurements a meaning.
Software metrics can describe source code very well, however other quality factors need different techniques.

2.2.2 Goal Question Metric

The Goal Question Metric (GQM)[BCR94] approach can be applied to find metrics for specific goals. These goals correspond to requirements. This approach fulfills quality by its basic definition. GQM is a top-down approach in which first of all a goal with a certain viewpoint is defined. Next a set

[7] http://www.campwoodsw.com/sourcemonitor.html, visited on 14th Jan 2014.
[8] http://sourceforge.net/projects/pymetrics/, visited on 14th Jan 2014.

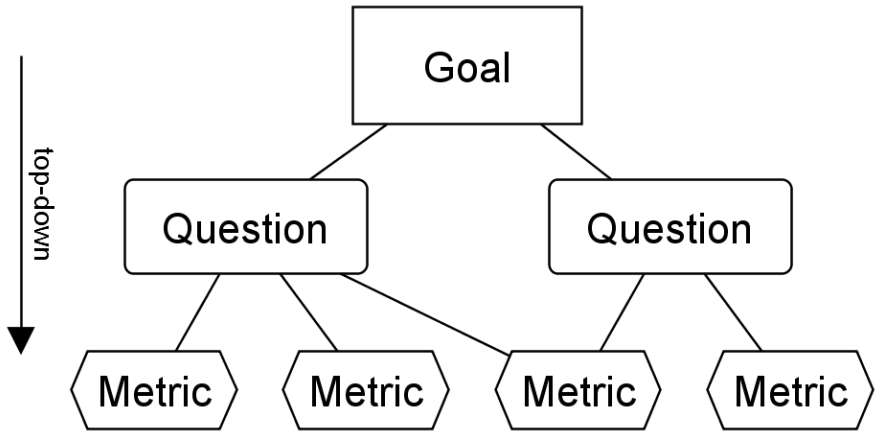

Figure 2.4: Goal Question Metric[9]

of questions is used to characterize the goal. For the last step, metrics are associated with every question, see Figure 2.4.

Goal A goal refers to an object (Product, Process, Resource), to a viewpoint and to an issue. Further the goal has a purpose.

Question One or multiple questions describe the fulfillment of the goal based on quality characteristics and from a selected viewpoint.

Metric The questions are associated with a set of data to answer the questions quantitatively. These data are metrics.

Table 2.1 shows an example.

2.2.3 Quality Attribute Scenarios

Quality Attribute Scenarios[CKK01] are a method to describe measurable scenarios. A quality attribute is an synonym for quality factor or characteristic. An example is maintainability. Scenarios are interactions between stakeholder, like user or project manager, and the software product. For a user, the scenario would describe the performing of a task. The scenario is always associated with a quality attribute. Figure 2.5 shows a simple example for the attribute performance: "*Users initiate 1,000 transactions per minute stochastically under normal operations, and these transactions are processed with an average latency of two seconds.*"[BCK03, sect. 4.3]

[9]Reproduced picture is based on [BCR94, Fig. 1].

Goal	Purpose	Improve
	Issue	the timeliness of
	Object(process)	change request processing
	Viewpoint	from the project manager's viewpoint
Question		What is the current change request processing speed?
Metric		Average cycle time Standard deviation % cases outside of the upper limit
Question		Is the performance of the process improving?
Metric		$\frac{\text{Current average cycle time}}{\text{Baseline average cycle time}} * 100$ Subjective rating of manager's satisfaction

Table 2.1: A GQM example

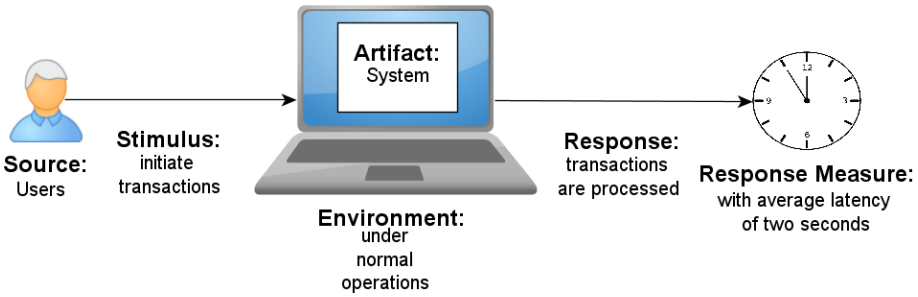

Figure 2.5: An example scenario for the performance attribute[10]

A quality attribute scenario consists of the following six parts:

The **Source of stimulus** invokes the stimuli either from external or internal sources. In our example the source is a set of users.

The **Stimulus** is the event which is invoked by the source and arrives at the system.

The **Artifact** is the target to stimulate. It can be the system or related data and services.

[10]Reproduced picture is based on [BCK03, Fig. 4.5].

The **Response** is the activity which is induced by the stimulus.

The **Response Measurement** is the analyzing of the response activity. It has to be measurable or verifiable.

Such scenarios are very useful to define quality requirements in the early state of the development.

2.3 Clean Code

Most introduced quality characteristics refers to a software product but not source code. However, the resulting software is made up of source code. For any change, the source code needs to be touched. With the GQM or quality attribute scenarios, metrics can be obtained without including the source code but to improve the software quality aspects, if they don't satisfy the requirements, the source code has to be changed.
Thus software quality is always related to code, either directly (maintenance) or indirectly (improving performance by changing code). Therefore it is important to emphasize clean code[Mar09].

There is no clear definition of clean code. Sometimes it, as well as beauty, is in the eye of the beholder. However, usually software developers get an agreement whether code is clean or not. There are multiple things which contribute to it. Some of them are universal and can be applied to any type of programming language. Others depend on a specific programming language. *The Zen of Python*[11] is an abstract description of clean code:

> *The Zen of Python, by Tim Peters*
>
> *Beautiful is better than ugly.*
> *Explicit is better than implicit.*
> *Simple is better than complex.*
> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *Although practicality beats purity.*
> *Errors should never pass silently.*
> *Unless explicitly silenced.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one– and preferably only one –obvious way to do it.*

[11] The Zen of Python is a collection of guidelines by Tim Peters and will be printed to the standard output stream by using the statement *import this*.

Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Following are some facts from [Mar09]:

- *Clean code does one thing well.* (Bjarne Stroustrup)
- *Clean code reads like well-written prose.* (Grady Booch)
- *Clean code always looks like it was written by someone who cares.* (Michael Feathers)
- *You know you are working on clean code when each routine you read turns out to be pretty much what you expected.* (Ward Cunningham)

Pollice[Pol06] thinks, there are three reason why dirty or sloppy code is written:

Time pressure
The most software projects have deadlines. If the project fall behind schedule the pressure on the team grows and developers tend to take shortcuts. Instead of giving meaningful identifiers, they name their variables with single character. Helpful comments to describe the algorithms are omitted. They only focus on getting the code work.
The OMC team does not work project-based with particular deadlines but they have to subordinate to CERN's schedules. For example, to be able to correct the beam optics, the code has to be adapted until the end of LS1 (long shutdown 1).

Lack of training
No matter where one learned to program, the university, a book or any other way, the probability is high, that one did not learn how to write clean code. Code examples are trimmed to a minimum and are always accompanied with thorough explanations. One reason is the time pressure on the educational institution. Coding style, readability and other things which contribute to clean code are omitted.
Physicists are taught to transform mathematical equations into a program. They know the basic constructs of programming, but writing clean code is not part of a physicists education.

Motivation
The aim of students is to get the code working correctly. They are rewarded for solving a problem. Style points do not exist. The focus of getting the code right and neglecting pretty code is not unique to the university. It applies to industry and can also be seen in programming

competitions. On the other side, judging code for elegance and beauty is not an easy task.
Source code is almost never shown in presentations and meetings at the OMC team and teams of physicists in general. Only the result is valued.

The source code of the OMC team is unfortunately not always clean code and this represents a big obstacle in improving certain software quality characteristics like performance. Hence this thesis will especially focus on the improvement of the software quality by creating clean code. A section about improving clean code can be found in the best practices guide, see B.1 on p. 77.

Chapter 3

Software Development Models

The last chapter defined software quality by using the product quality model of ISO/IEC 25010:2011. To improve product quality, it is necessary to improve the process quality since high-quality processes lead to high-quality products[Wag13, p.9]. A software development process is based on a model, which is an abstraction of specified phases that occur in a prescribed order. Software development models play a significant role in software development. They represent the link between project management and development. The first model is the famous Waterfall model proposed by Winston W. Royce in 1970[Roy70]. Since then a lot of different software development models occurred. This chapter presents the most important models, divided in heavyweight and lightweight.
The different models have their own approaches and characteristics but there are common phases in every model:

Analysis consists of all the processes and activities to gather the requirements and specifications of the new software. This includes also the investigation of the specific domain. This phase solves what the system shall do.

Design solves the question how the system should realize the collected requirements. The result is a designed software architecture which considers the current requirements and ensures that future requirements can be addressed.

Implementation is the activity which transforms the created design into a specific programming language. The choice of the right programming language depends on the domain and environment.

Integration is the activity of populating new features and changes into the software product.

Testing is the process which tries to find failures of a system and verifies

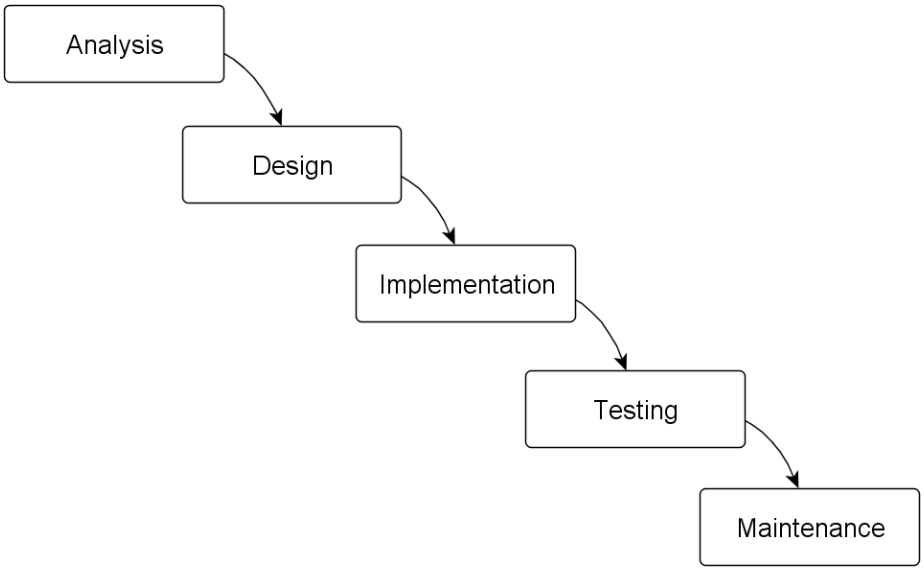

Figure 3.1: Waterfall model

that the requirements are fulfilled.

Installing/Deployment is the process of bringing the software into product environment to be available for business use.

Maintenance consists of all actions which take place after the release of software to extend, fix and adapt it.

3.1 Heavyweight Models

The models which are resistant to changes and include a lot of bureaucracy are called heavyweight. 'Static models' is a synonym therefor since these models do not adapt to changes.

3.1.1 Waterfall

The Waterfall model[Roy70,Pin08] is a highly structured model. It is considered as the traditional approach of software development and consists of the usual phases *Analysis*, *Design*, *Implementation*, *Testing* and *Maintenance*. As an straightforward process it is simply to use. The output of a phase is the input of the next phase. The process flows down the single phases,and that is where the metaphor Waterfall comes from, see Figure 3.1.

Every phase have to be completed entirely before the next phase can start. Reviews at the end of every phase evaluates the work, decide if the current phase is complete and check if the project process meets the ex-

pectations. The waterfall model is bureaucracy-intensive and produces a lot of documentation. The first phases create documents which describe what should be done. The following phases add more details about how it was done.
It is not easy to adapt changes since the created requirements from the first phase considered to be set in stone before the proceeding of the design. Thus often modifications and fixes are shifted to the maintenance phase.

Advantages

- Simple and easy to use
- Easy to manage due to the rigidity of the model
  - Each phase has specific deliverables and a review process
- System is well documented
- Phases correspond with project management phases
- Cost and schedule estimates may be lower and more accurate
- Time spent earlier on making sure that requirements and design are absolutely correct will save you much time and effort later
- Details can be addressed with more engineering effort if software is large or complex

Disadvantages

- Not robust against risks and changes
- Because the model is sequential, there is only local feedback at the transition between phases
- Significant adjustments in the scope during the development process can kill a project
- Corrections must often wait for the maintenance phase
- No working software is produced until a late state during the development process

The Waterfall model is proper for projects where the requirements are well understood from the beginning and the risk of changes is very low. It is still used, especially by governments and governmental organizations, for example by the U.S Department of Defense.

3.1.2 Incremental

The Incremental model[Pin08,Lew05] is an evolution of the Waterfall model. It is basically a series of Waterfall cycles, see figure 3.2.

All requirements should be given before the first cycle or iteration. However, requirements can be slightly changed and new and small requirements can be added. They will be addressed in further iterations. The first iteration

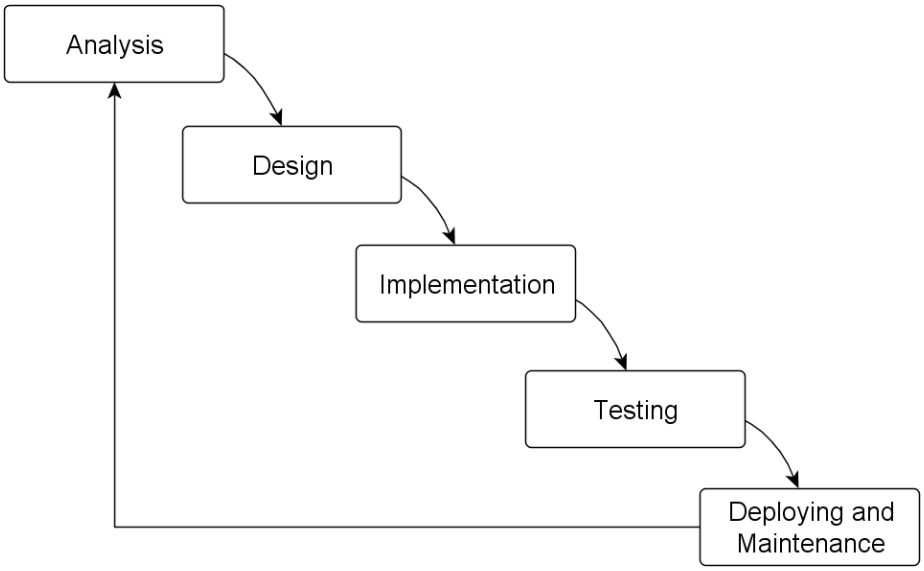

Figure 3.2: The Incremental model consists of multiple Waterfall iterations

includes the requirements which are necessary for the first release. Every succeeding cycle adds more features. If the requirements for the first release are too high, the development can be split in several iterations. Every iteration consists of the serial steps designing, implementing, integrating, testing and deploying. New functions are available within weeks, whereas a client has to wait months until receiving a product developed by the Waterfall approach. This gives additional feedback and thus there is also a good chance to find misconceptions or errors in the requirements or the software after a cycle. Every iteration serves also as the maintenance for preceding cycles.

Advantages

- Every increment delivers a new operational product
- Testing and debugging is easier
- More flexible - less costly to change scope and requirements
- Easier to manage risks
- User feedback after an iteration can be obtained
- Allows small modifications of the requirements

Disadvantages

- The majority of the requirements have to be known in the beginning
- Each release has to be incorporated

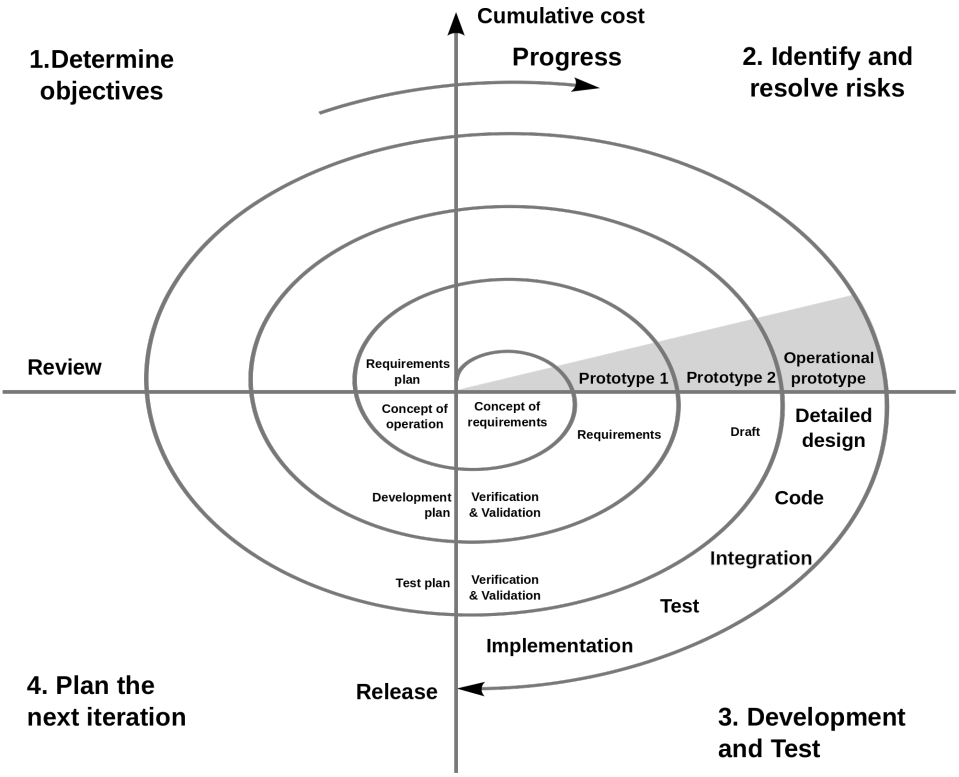

Figure 3.3: Overview of the Spiral model concept[1]

- Clients have to learn how to use the new system with each deployment
- Business operations are impacted with each release

The Incremental model is suitable for projects where the funding could be dropped since there will still be a running product. If main functionalities have to be released quickly but other requirements can wait, this model is a solution. It can also be the choice if a project contains low to medium risks.

3.1.3 Spiral

The Spiral model[Pin08, Lew05, KQK11] was proposed by Barry Boehm in 1986 and focuses highly on risk management. Concepts from Waterfall and Incremental model are present in the Spiral model. Depending on how it is implemented it can be similar to a combination of the already presented models. It is represented as a spiral in four quadrants, see figure 3.3.

[1]Figure taken from http://en.wikipedia.org/wiki/File:Spiral_model_%28Boehm,_1988%29.svg, visited on 16th Jan 2014.

This model is not structured in usual phases, like analysis, design etc., but in the following activities which represent the quadrants:

1. Determine objectives
2. Identify and resolve risks
3. Development and test
4. Plan the next iteration

This model emphasizes risk management. In every iteration risks are identified and resolved or the impact on the project is decreased. A prototype is created in the same phase. The prototype will be reviewed together with the stakeholders, which can observe the process thereby, before the next iteration is planned. Although prototypes, sketches or pictures in the early iterations, are created, software is usually first developed in the last iteration. The radius represents the cost and the spiral is the progress. The model does not easily handle concurrent event and iterations.

Advantages

- Emphasizes risk management
- Verification and validation in early stages of the project
- User/stakeholder are included in every cycle
- Good for large and mission-critical projects

Disadvantages

- The spiral model is more complex and harder to manage
- Risk analysis requires specific expertise
- Doesn't work well for smaller projects
- Does not easily handle dynamic changes in requirements

The Spiral model is appropriate for large and high budget projects and when risk assessment becomes very critical.

3.2 Lightweight Models

After a lot of years the projects, which used static models, haven't been successful yet. This led to the creation of lightweight, dynamic and agile development methods in the mid-1990s. The static models were seen as too bureaucratic and plan oriented. These were considered as obstacles in the effective work of software developers.
The development in lightweight models focuses on short iterations, interaction between people and being highly responsive to changes. The agile

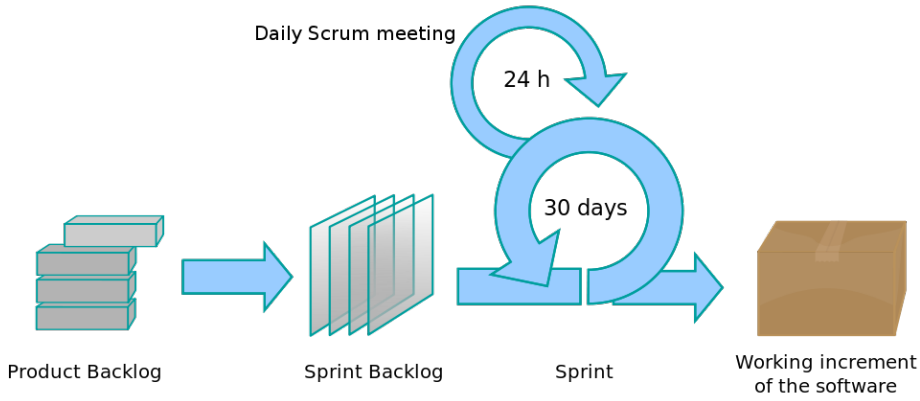

Figure 3.4: Overview of Scrum[3]

models develop software in small iterations with small increments. The key statements of the *Agile Manifesto*[2] are:

> ***Individuals and interactions*** *over processes and tools*
> ***Working software*** *over comprehensive documentation*
> ***Customer collaboration*** *over contract negotiation*
> ***Responding to change*** *over following a plan*

3.2.1 Scrum

Scrum[Pin08, KQK11] comes from a Rugby term which stands for collaborating of individual groups to form a mighty whole. It was described in a paper from Jeff Sutherland and Ken Schwaber in 1995. It is a iterative and incremental development approach. Figure 3.4 shows the structure of the included elements and activities.

The software will be developed incrementally in the so-called *Sprints*. A sprint is the developing phase and can last from around one to four weeks. However, the sprints within one project have the same duration what is called time-boxed. The *Daily Scrum meeting* takes place every day in which the team members have to answer the following questions:

- What did you do yesterday?
- What will you do today?
- What obstacles are in your way?

[2] http://agilemanifesto.org/, visited on 16th Jan 2014.

[3] Figure taken from http://en.wikipedia.org/wiki/File:Scrum_process.svg, visited on 16th Jan 2014.

One goal of this approach is that every member of the team gets an overview of the project state. It shall remain as short as possible and must not exceed 15 minutes. Problems can be discussed afterwards in a smaller circle. The *Sprint Review meeting* takes place at the end of every sprint, where the results of it will be presented. A retrospective meeting takes also place at the end of every sprint. The goal here is to discuss problems concerning the management of their work and how to improve it. The software is created incrementally in this way.
The *Product Backlog* contains all unfulfilled requirements, which are changeable, of the software sorted by priority. The *Sprint Planning meeting* is held before every sprint. During this meeting the team member decide which requirements they can accomplish within the next sprint. These requirements are moved from the product backlog to the *Sprint Backlog*. It is forbidden to add further work to the sprint backlog during a sprint. The single exception is that the team is ahead of his schedule and they could accomplish another requirement. In case of a serious issue the sprint have to be canceled and a new has to be planned. If a team cannot finish a requirement it moves back to the product backlog.
The following roles are mandatory for Scrum:

Product Owner The Product Owner represents the voice of the customer. He is the only person who is allowed to manage the product backlog. Only he can add, change or remove requirements from the product backlog.This person is officially responsible for the project. In a company this role could be the product manager or the project manager.

Scrum Master The Scrum Master is comparable to a team leader. He provides everything what the team needs to fulfill the work. He ensures that the team follows the Scrum activities and rules. He also avoids distraction from external influences

Development Team A Scrum team usually consist of four to seven developers and is self-organizing. It is up to the team how to solve the goals of the sprint. Typically role names, such as programmer, tester or architect, are not used within Scrum teams.

Advantages

- Productivity increases
- Everyone knows the detailed progress of the project
- Focus on team communications, team spirit and solidarity
- Frequent deliverables provide stakeholder feedback

Disadvantages

- Decisions are entirely up to the team
- Requires a certain level of training for all users
- Constant monitoring is necessary
- Scrum is new and different, people are resistant to changes

Scrum is very suitable for fast moving web or media projects.

3.2.2 eXtreme Programming - XP

E**x**treme **P**rogramming[Bec99, Pin08, KQK11] arose from the collaboration between Kent Beck and Ward Cunningham in the late 1980's and early 1990's. They extended their ideas to an software development approach that was adaptive and people-oriented. Kent Beck published his book *Extreme Programming Explained*[Bec99] about the description of the model. The process emphasizes programmers and technical practices. The software will be developed incrementally and in iterations, see figure 3.5.

It is called extreme because practices and principles are expanded to extreme levels[Bec99, p. 7]:

- *If code reviews are good, we'll review code all the time (**pair programming**).*
- *If testing is good, everybody will test all the time (**unit testing**), even the customers (**functional testing**).*
- *If design is good, we'll make it part of everybody's daily business (**refactoring**).*
- *If simplicity is good, we'll always leave the system with the simplest design that supports its current functionality (**the simplest thing that could possibly work**).*
- *If architecture is important, everybody will work defining and refining the architecture all the time (**metaphor**).*
- *If integration testing is important, then we'll integrate and test several times a day (**continuous integration**).*
- *If short iterations are good, we'll make the iterations really, really short – seconds and minutes and hours, not weeks and months and years (**the Planning Game**).*

XP has similar roles to Scrum: customer, programmer and coach.
The customer resembles the Scrum product owner and is responsible for the user stories. These are basically about three sentences respectively which describe requirements and interactions with the product. The customer prioritize the user stories and chooses the ones which will be processed in the

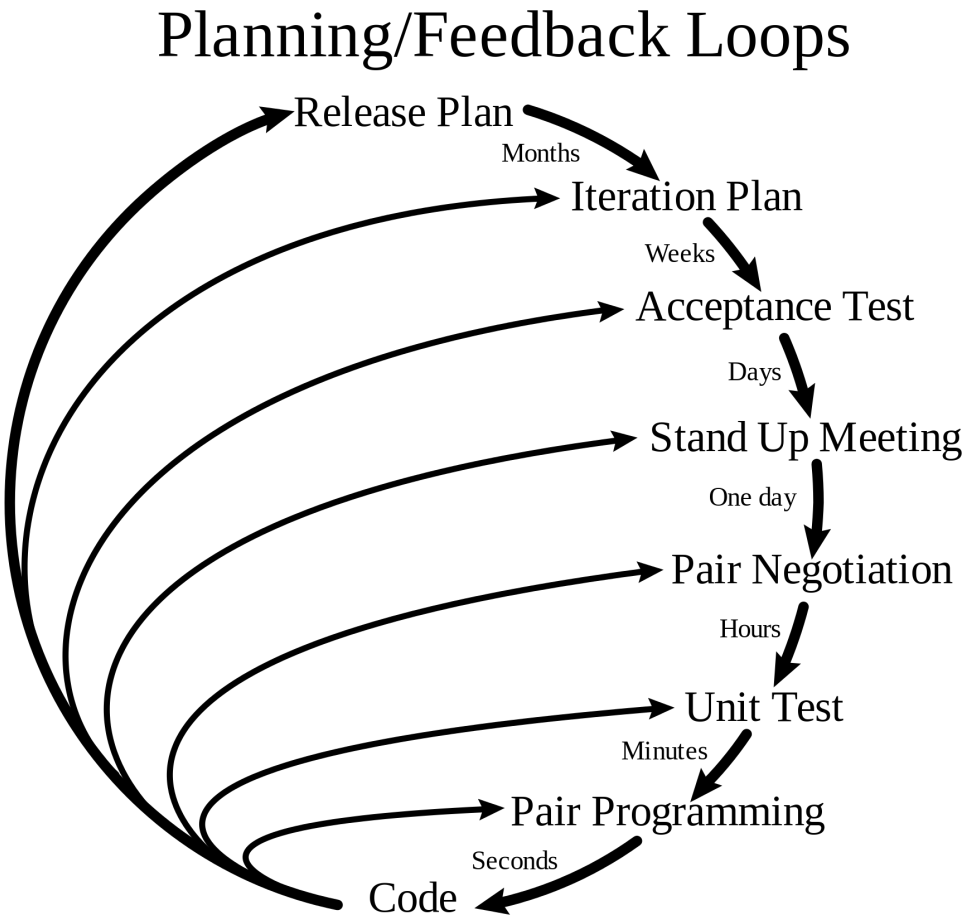

Figure 3.5: The flow chart shows feedback loops or channels of communication[4]

next iteration. One iteration lasts typically one or two weeks but not more than four. The customer is also responsible for the acceptance tests which validates the software against the requirements or user stories. He is not necessarily an user of the future system, he can also be a product manager, project manager or business analyst.

The XP programmer needs a wide field of knowledge since XP does not distinguish between designer, programmer, architects and so on. It is expected to work as a team and the responsibilities are shared. The XP programmer has to follow the mentioned practices and principles.

The XP coach is similar to the Scrum master and supports the team. He monitors and ensures the XP practices. He holds back possible interruptions and distractions from the team.

[4]Figure taken from http://en.wikipedia.org/wiki/File:Extreme_Programming.svg, visited on 16th Jan 2014.

The release planning meeting specifies the date of the next release and which user stories will be included. For every iteration plan the *Planning Game*, which basically estimates the costs of the several user stories[5], is executed. The customer decides from the set of user stories for current release which are included in the next iteration. The sum of the costs of the user stories must not exceed the value which a team can handle in one iteration. This value is decided by the team and depends on their velocity.
During an iteration the practices are followed to produce a product. A daily stand up meeting, similar to the daily Scrum meeting, is held to discuss problems and solutions. Every iteration increments the software. It is not allowed to deliver an unfinished feature or a feature with only half of the quality, as well as extending the iteration to finish features. XP emphasizes also a near connection to the customer. The customer writes the stories and the acceptance tests. He answers questions from the developers as soon as they occur.

Advantages

- Useful practices and principles (pair programming, unit testing, ...)
- Forces simplicity and simple solutions
- Test based approach to requirements
- Customer decides priority but team decides workload
- Permanent help from customer

Disadvantages

- Requires permanently a customer
- Documentation primary through face to face communication and code
- Needs experience and skill
- Provides essentially no data gathering guidance
- Difficult to scale up to large projects where documentation is essential

XP can be adopted by small to medium size projects where a permanent customer is available. It is not possible for large projects or teams.

3.2.3 Kanban

Kanban[Sha11,Sco10] is a method which does not dictate the steps of a whole process like Scrum or XP. It tries to improve the development by letting the

[5]The rules can be found here http://c2.com/cgi/wiki?PlanningGame, visited on 16th Jan 2014.

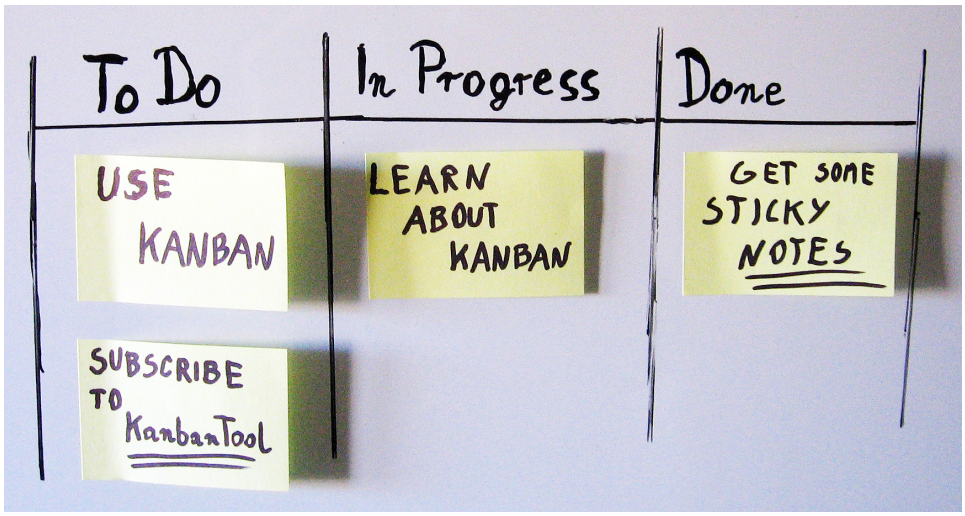

Figure 3.6: Kanban board - A possible visualization of Work in Process[6]

team identify the bottlenecks of their process. Kanban was described in detail by David J. Anderson in [And10]. The origin method was developed and applied by Toyota in 1953. Kanban is the Japanese word for visual card.
The activities and processes of a company haven't to be changed in the beginning to adapt Kanban. The approach starts by visualizing the work flow and limiting the *Work in Process* (WIP).
A visualization can be realized with a *Kanban board*, see figure 3.6.

The board consists of several columns representing the state of the task or activities in the development. Tasks, notes on simple post-its, will be put to the according column. In the course of the development they move from one column to another until they are completed. In contrast to Scrum and XP the current work is always visible, also for the management. The WIP, currently processed tasks, is limited by a number in each column. This number represents the maximal amount of tasks for the column what restraints the number of parallel executions. It is not possible to put more tasks if all columns are full. Bottlenecks can be detected by monitoring the work flow. A process impairs the flow if its column is regularly full.
The goal of Kanban is to improve incrementally the development process by removing continuously the bottlenecks in the work flow. Therefore the flow has to be managed all the time. It should be monitored, measured and reported. If the team works well and without problems the WIP limit can be reduced. Thus new bottlenecks can be discovered. Retrospective meet-

[6]Figure taken from http://en.wikipedia.org/wiki/Kanban_board, visited on 17th Jan 2014.

ings take place from time to time to discuss further possible problems and improvements.
Tasks are the breakdown of user stories. The stakeholder can prioritize the task but they cannot push them into the WIP, only into a backlog. If free space is available, the tasks are pulled on the Kanban board by the team.

Advantages

- Easy to adopt(no big changes to existing processes in the beginning)
- No pressure through time-boxed iterations
- Continuously eliminating of bottlenecks
- Teams will not be overloaded

Disadvantages

- Identifies problems in processes but doesn't solve them

Kanban can be applied to a variety of projects and teams since it aims at improving current activities but not at introducing compulsory processes.

3.3 Conclusion

The presented models and approaches are all important in software development. They try to avoid or solve problems, to deliver the best possible product with a high quality. The models mirror the evolution of software development processes. However, these models aim at teams which produce regularly software. This is not the case for the OMC team. They need a process which is easy to learn and straightforward and should avoid too much documentation. The best practices guide in appendix B.5 on page 89 proposes a tailored process which easily can be adopted. It was not accomplished to introduce this process. A good start could be making the current work visible, as it is done in Kanban. This could be achieved with the issue tracker Jira[7], which is hosted by CERN. There are some features for agile development which include also an virtual Kanban board. This could be extended with a custom issue work flow in Jira to match the proposed software development process.

[7] http://information-technology.web.cern.ch/services/JIRA-service, visited on 1st Feb 2014.

Chapter 4

Software Testing

A better software development process or a better design improves software quality but testing does not. Quality can't be tested into a product. It does not even proof the correctness of a program. Testing shows basically that assumed behavior or requirements are fulfilled. However, fulfilling requirements is the abstract meaning of quality. So, testing can prove the existence of quality and it is not possible to produce a product with high quality without testing it. [MSB12] and [EM07] served as sources for this chapter.
An appropriate definition of testing according to [MSB12, p. 6] is:

> *Testing is the process of executing a program with the intent of finding errors.*

Appropriate, because of the last part of the definition. A test should be created with the purpose to find errors. For being a successful test, it has to find errors.
Usually the programmer is not the user and should not test the software (except of unit tests). Software quality is perceived differently from this two parties and the programmer knows his program. He possibly would not push too hard on it to not 'destroy' it. However, in the case here the programmers are the users. The physicists use self-written software as a tool to fulfill their tasks.
An important fact about testing, a tester has to know, is that it is economically not possible to test everything. There are astronomically high numbers of test cases already in small applications. Tests have to be restrained to the most important test cases. Thus tests can't guarantee that the program is error free.
This chapter will present testing strategies and common test methods separated in static and dynamic tests. Test-driven development(TDD) is emphasized.

4.1 Test Strategies

Test strategies describe the approach of creating a test. Two of the most important ones are discussed here.

4.1.1 Black-Box Testing

The metaphor *black-box* stands for not knowing the internal structure and behavior of the software. Thus specifications for the input and output are used to create tests. Because of that it is also known as data-driven or input/output-driven testing. The goal of these tests is to find circumstances in which the program does not behave in accordance with the specifications.

4.1.2 White-Box Testing

White-box strategies use the knowledge of the internal structure to create tests. It is also called logic-driven testing and aims at exhaustive path testing. However, this is not always manageable.
The graph in figure 4.1 represents a very trivial program. Vertices correspond to statements and edges correspond to transitions between the statements. A path is a possible way between vertex a and vertex b. The example contains a *do-while* loop. A *if-else* statement is nested within this loop. The else-branch contains again an *if* statement. So the loop consists of three possible paths. Assuming that all decisions are independent from each other the number of unique paths is 3^n where n is the number of iterations of the do-while loop. If there would be 20 iterations then there would be 3486784401 paths to test.
Since exhaustive path testing is impracticable, it should be aimed at getting a full code coverage, execution of all statements. But a code coverage of 100% does not assure error-free programs. Program logic could be missing or wrong.
To test efficiently both, Black-Box and White-Box testing, should be used.

4.2 Static Testing

Static testing include all tests wherein the product is not executed. Usually these tests are carried out exhaustively in the beginning of a project. Static testing is not restricted to source code. Basically every document of a project can be subjected to static testing and thus it can be applied very early in development. This is a big advantage, since the later an error will be found, the more expensive it is to fix it. So, few resources makes it possible to avoid a huge amount of costs due to bug fixing.
Desk checking, *Inspections* and *Walk-throughs* are discussed here.[1]

[1] There are multiple definitions of these terms. The definition of [EM07] is used here.

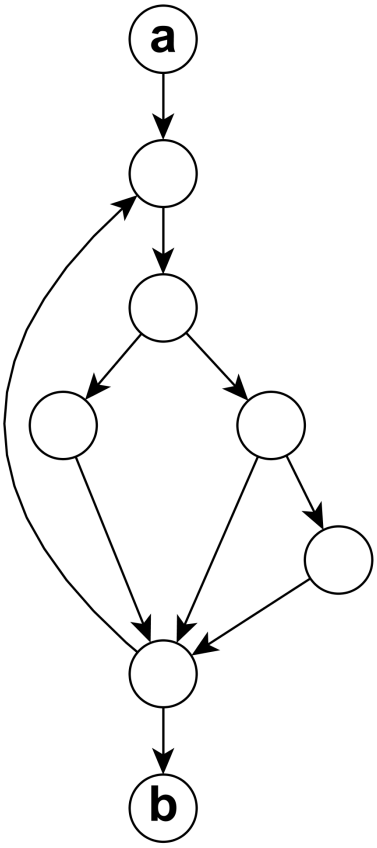

Figure 4.1: A graph belonging to a very simple program

4.2.1 Desk Checking

Desk checking is the least formal and time-consuming technique, where the author is also the tester. First tools, like spell and grammar checker, syntax checker etc., are used to clean up the cosmetic appearance. Then, the author slowly reviews his document and searches for inconsistencies, incompleteness and missing information. The author corrects found problems and cleans the cosmetic appearance in the end again.

4.2.2 Inspections

An inspection is a little bit more formal and time-consuming. The intent is that the document is checked by another pair of eyes. This role usually belongs to a more experienced programmer. He reads the document and tries to discover problems. To not being influenced by the author, the reviewer

has to separate the document and the review from the author. Found problems are documented and presented to the author in a subsequent meeting. Consequently he provides suggested solutions which are evaluated by the reviewer or a senior programmer.

4.2.3 Walk-Throughs

A walk-through is the most formal and time-consuming technique but also the most efficient at identifying content problems. The document will be sent in advance of a scheduled meeting to the participants. The participants read the document and note questions. They are the author, a moderator, senior technical staff and possibly business staff. In this meeting the author presents his document. The moderator ensures the clarification of questions from the audience and answers by the author. Further he ensures that questions are asked in an objective way. He also documents all content problems and author responses for later resolution. Before solving problems the authors suggested solutions are evaluated by senior programmer.

4.3 Dynamic Testing

Dynamic tests are all those in which source code will be executed. It is the main part of the testing activities. However, static tests should be indispensable, complementary tests. Whereby static testing reveals the cause of an error, dynamic testing shows wrong behavior. The cause has to be found.

4.3.1 Functional Testing

Functional testing includes all tests which validate the software behavior against the business functionality, specified in the software requirements or specifications. Business functionality is all the activities that support the daily business routines.
Test cases can be derived from *use cases*. Use cases describe interactions between the end user and the system. Basically they clear what the user can do with the system.

4.3.2 Structural (Non-functional) Testing

Structural testing validates the behavior of the software. The non-functional requirements of the software are tested. It includes the testing of the characteristics of software quality, see chapter 2 on p.9, if required or specified:

- Performance efficiency
- Usability
- Reliability
- Functional suitability

- Compatibility
- Maintainability
- Portability
- Security

4.3.3 Logic Coverage Testing

Logic coverage testing is a white-box testing approach. As mentioned before, exhaustive path testing is the aim of white-box testing but this is not always a realistic goal.

Statement Coverage
The test cases of a full statement coverage have to execute every statement in the program unit. This is a weak criteria for a white-box test since especially errors in conditions are not covered.

Decision(Branch) Coverage
Decision or branch coverage is satisfied if the test cases ensures that every decision evaluates to *true* and *false*. It satisfies also statement coverage. However, errors in composed conditions are difficult to find.

Condition Coverage
For condition coverage each condition in a decision should be executed to all possible outcomes at least once. The weak point is that a full condition coverage does not satisfy a full decision coverage. A simple example follows with the following code snippet:

Listing 4.1: Simple program with a compound decision

```
1 if a and b:
2     do_x()
3 else:
4     do_y()
```

The test cases to satisfy condition coverage could be:

- a = True; b = False
- a = False; b = True

The test cases would not execute the *then*-branch and thus full decision coverage is not given.

Decision/Condition Coverage
Decision/condition coverage solves the condition coverage problem. Here it is necessary that each condition in a decision takes on all possible outcomes at least once and each decision takes on all possible outcomes at least once. The weakness of this approach is that it does not exercise all outcomes of

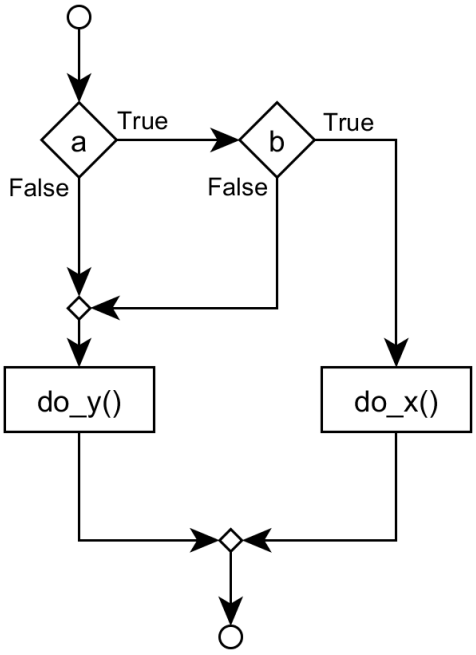

Figure 4.2: Machine code for listing 4.1

all decisions since certain conditions are masked by others.
Figure 4.2 shows the flowchart in the way a compiler would generate machine code for listing 4.1. The following three test cases would satisfy decision/condition coverage:

- a = False; b = False
- a = False; b = True
- a = True; b = True

However, the test cases fail to execute the *False* outcome of b since the *and* masks or blocks the evaluation.

Multiple-Condition Coverage

Multiple-condition coverage aims at the execution of all possible combinations of condition outcomes in each decision at least once. For instance, the decision

```
1 if a and b:
```

consists of the conditions a and b. To satisfy multiple-condition coverage, the following test cases are necessary:

- a = False; b = False

- a = False; b = True
- a = True; b = False
- a = True; b = True

Multiple-condition coverage satisfies the decision coverage, condition coverage, and decision/condition coverage criteria as well.

4.4 Test-Driven Development(TDD)

The usual way of developing is to write code and test the code afterwards. Test-Driven Development (TDD) turns the order. First, tests as automated unit tests are written and afterwards the production code. It was invented by Kent Beck and mentioned as a practice in his XP book in 1999[Bec99]. The method is based on the following two rules:

- Write new code only if you first have failing automated unit tests
- Eliminate duplication

These two simple rules imply an order to the programming tasks:

1. Red – write a small test and check that it fails, it is possible that it does not even compile
2. Green – get the test working quickly by writing as few as possible production code
3. Refactor – eliminate duplication and easier the design of the implementation

This is called the TDD mantra[Bec02, p. x].
The developer follows short cycles of testing and coding. Short means here seconds or minutes but not hours. The code remains always in an easy to maintain state. It is hard to apply in the beginning but the benefits are very valuable. The written tests will help the developers also years after the creation. Programmers fear to break existing code if no automated tests exist. Code will not be touched. Automated unit tests show if something was broken due to changes.
Since tests have to be written, TDD consumes more time. This is wrong. Indeed it takes time to write the tests, but the benefit saves much more time. The produced code is easier to maintain or extend and if changes lead to a bug, the tests will show it. TDD is perfectly suitable for regression tests[2].
It is very hard, if not even impossible, to practice TDD on an existing code base. It is easy to write tests for code which is not written yet. The opposite

[2]Regression tests are those tests which are executed after code modifications. The purpose is to determine whether the modifications caused errors in other aspects of the program.

way is much more difficult. Mostly the production code seems to be too hard to test since this code was not designed to be testable.

Advantages

- Produced code is cleaner
- Automated tests are available for future code changes
- Minimalistic design
- Easier to write tests (writing unit tests after production code is very difficult)
- Easier to maintain
- Takes the fear of changing code

4.5 Conclusion

Testing does not improve software quality directly but it ensures the fulfillment of requirements. Especially automated tests are very helpful if the code will be modified. Thus testing increase quality indirectly.
TDD is emphasized because this approach would improve the development of the OMC team and thus also the software quality in the long term. Thus an example was included in the best practices guide, see B.1.6 on page 83. It was also introduced in coding lectures, but to apply TDD, more practical training would be necessary.

Chapter 5

Current Situation

This chapter describes the state of the OMC team. Multiple contributions were done in course of the internship. Thus the current situation has already some improvements but situations and work flows before the internship are described as well.
Surveys have been done to collect further information about the OMC team and similar teams. A SWOT analysis was done to evaluate the OMC team. Special circumstances and characteristics of the team were inspected.

5.1 Surveys

Two surveys have been created to gather information and to get an overview of the situation and the work flow of physicists. The first survey was addressed to the OMC team and five responses were collected. To get a broader spectrum another, more general survey was created and addressed to the BE-ABP group[1]. 25 responses were collected. The surveys were executed in December. To that time several improvements to the OMC team have been introduced. So the second survey reflects the situation before improvements better. The surveys can be found in appendix C.

5.1.1 OMC - Survey

Five responses were collected in the OMC team. More were not expected since it is a small team of eight people and not everybody writes source code. The survey was done after some improvements. Nevertheless it gives some useful information.

Programming Skills and Environment
Every participant writes source code weekly and assess his skills to use

[1] See subsection 1.1.1 on p. 3

Python and its language tools very good. The ability to transform a mathematical equation into a program tends to *very good*. Using Object-oriented programming (OOP) and writing clean code are estimated rather *bad*. The majority still uses just a text editor to write source code. Only one person uses static analysis.
Everyone designs or plans his software by shortly thinking about the best solution and starting to write code afterwards. The question about software testing is quite interesting. Three members test every small part of the software. The interpretation of a *small part of software* seems to be different since there are no tests for small parts in the OMC code base. Computer scientists would think of one or some few functions whereby physicists perceive it as a software component which can consist of multiple classes.
Two people did never hear of TDD(test-driven development). The same number know it but can not imagine how to implement it and one person could apply it. Somebody left a comment in which he states, that he thinks it would be too time-consuming for him. It consumes some time to write appropriate tests but in the end it saves much more, see section 4.4.
A positive outcome is that all would attend lectures with live coding concerning OOP(object-oriented programming), refactoring and clean code.
The majority (four people) think that a well structured and tailored development process would improve the software quality of future developments. One commented that there is a risk, people could stop writing and submitting code, if the process is too difficult and time consuming.

Version Control and Information Handling
Since using Git was emphasized in the OMC team for several months it is not surprising that every participant uses Git. But two members assess their Git level as *bad* and even *very bad* in advanced concepts like branching and tagging. The most use the command line to work with Git but the graphical software is used as well. Still not everybody works in accordance with the proposed work flow to make changes on our common Beta-Beat.src directory, see A.1, but at least direct changes without committing are avoided.
The preferred way to solve issues is still writing an email instead of opening an issue, but the created wiki, see 6.2, is used by everybody from time to time. The idea about a central space with information about all current developing software tends to make sense and a comment even stated that it is very important if there are several developers.

5.1.2 BE-ABP - Survey

Since the OMC team is a typical team of physicists at CERN, it is helpful to get answers from physicists who develop also software, even if they don't work directly in the OMC team. That is the reason why a further survey was done with more general questions and sent to the BE-ABP group which has

about 250 members. But only a few of them are physicists who develop software and so the 25 responses are reasonable. Further it reflects the situation in the OMC team before making own contributions better. The members of the OMC team answered this survey as well.

Programming Skills and Environment
It is interesting that 56% (14 persons) of the participants write source code daily, quite more than assumed. The most used programming languages with respectively 30.9% are Python and C/C++. Fortran follows with 12.8%.
The majority asses their programming skills *good*. Using OOP and writing clean code are assessed not *very good* nor *very bad* by the most participants. A text editor like Emacs or Vim with some plugins is used mostly with 44% to develop software. It is followed by a plain text editor with 32%. A graphical Integrated Development Environment (IDE) like Eclipse or NetBeans is only used by 20%. This is the reason why 18 persons examine the output of the compiler for the static analysis. Only two participants work with advanced static analysis tools like PyLint, FindBugs or cpplint. Here is a big improvement possible.
21 people do not have a style guide at all. To design/plan software the most of the participants think shortly what could be the best solution and start to write code afterwards. Unfortunately 22 persons do not use reviews for specifications and/or code. 17 people test every small part of their software. Here varies the interpretation of *small part* again. More than a half did not hear of TDD.
18 participants would like to visit lectures with live coding. Only two persons think that a well structured and tailored development process would not help to improve the software quality and four would not obey the rules of a strict development process.

Version Control and Information Handling
Only seven people do not use a version control system. The remaining part, also seven, uses mostly Subversion. One participant uses even manually numbering versions and another one Dropbox. The overall usage level is assessed neutral, neither bad nor good. The basic work, like synchronizing and committing, is assessed more *good* and the advanced features like branching and tagging is assessed more *bad*. Again, the command line is the preferred way to work.
Collected knowledge, e.g. specific information about the usage of a script, is stored mostly as a comment in the code or in a separate README file. In any kind of issue the majority, 11 people, solve it on their own. Six participants would ask somebody face to face. One writes an email to someone and also only one opens a ticket on an issue tracker. The reason is that 20 persons do not use an issue or bug tracker and 18 do not use a wiki for

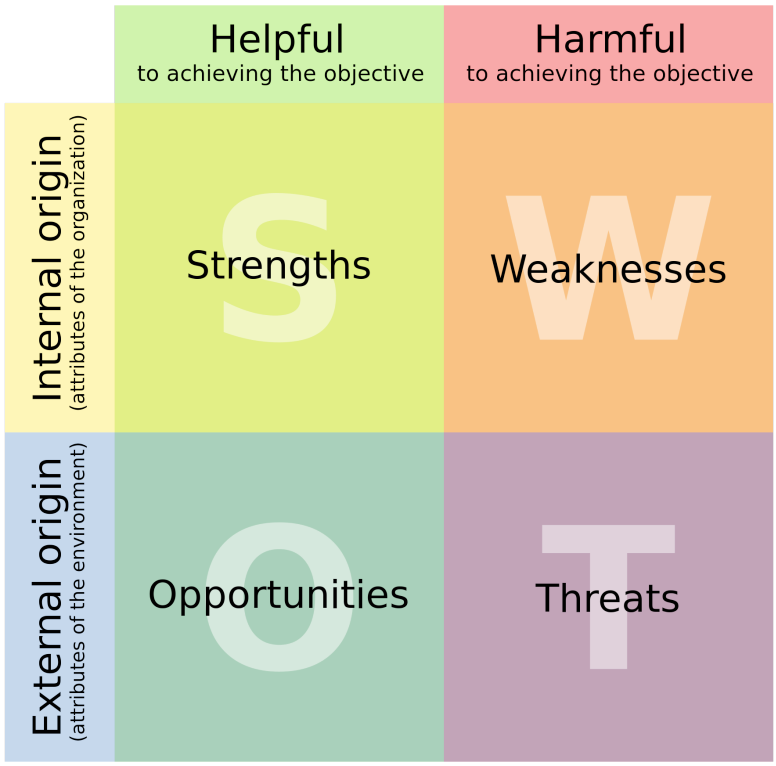

Figure 5.1: A 2x2 SWOT matrix[2]

collected information.

Overall the survey reflects mostly assumed results. It shows that the teams are far from professional development. All teams could need the same improvements as the OMC team.

5.2 SWOT Analysis

A SWOT analysis is a method to analyze a team, product, industry or place by identifying characteristics and elements. SWOT is the acronym for Strengths, Weaknesses, Opportunities and Threats. It is mainly used in economy to profile own and competitive products or companies but is not restricted to it. Characteristics and elements are divided into internal and external factors. Figure 5.1 shows an overview.

The OMC team is analyzed with respect to software development, not

[2]Picture taken from http://en.wikipedia.org/wiki/File:SWOT_en.svg, visited on 22nd Jan 2014.

physics research. Below are the elements and characteristics of it:[3]

Strengths

- Small team (few communication channels)
- Transformation of an equation into a program is no problem

Weaknesses

- Resistant to changes
- Most members are physicists and have no computer science background
- Members develop alone and without supervising
- Few information sharing concerning software development

Opportunities

- CERN provides a lot of IT services
- Frequent personnel change, easier to adopt changes to new members

Threats

- Frequent personnel change, people take knowledge when they leave
- Team is located in different offices and even buildings

Since CERN provides great IT services, they have to be used to minimize the weaknesses. But it should be avoided to create too much bureaucracy. Used tools should be supportive and not a harm.
It is important to convince the people, which are staying for a longer term, of methods and techniques which consume effort but improve software quality. They will supervise newcomers and can pass best practices to them.

5.3 Own Experience

From the view of the author the development is comparable with *code and fix* and *cowboy coding* approaches. These development approaches consider no fixed structures, practices and processes. The physicists know what they want to implement and they do it immediately. Everybody does it in his own style as he learned it.
Universities taught the physicists how to transform an equation into a program but the universities (and most books) did not show what else matters. All examples shown in the university or the books are trimmed to minimum details.[4] Further these examples are accompanied with explanations. However, source code outside of educational purposes should explain itself without foreign help or explanation, because it does not exist in a lot of cases.

[3] In this analysis external factors refer to outside the team but inside CERN. Usually external factors concern influences from outside of the company.

[4] The same holds also for computer science.

Gathering requirements from customer and users is a difficult task. Particular attention has to be paid to it in software development. This step is omitted here since client and developer are the same person. The disadvantage of this is that documentation is not created.
A big problem is the missing part between requirements retrieval and implementation, the design. This is the part in which the best solution for a given problem is considered. To avoid design errors, the intended implementation should be well-elaborated.
As already mentioned, developing software is utilized for own purposes as tool. Often some code will be used to retrieve characteristics from data and never published to others or used again. This could be expressed as a tailored solution to a unique case. A generalization of such code could provide a powerful library.
Further the most code is written by technical and doctoral students and fellows who work only for a relatively short period of about 2 months to 3 years. The result of the work is supervised by the staff.
The results are presented in meetings. Techniques and conclusions are shown but never an implemented design or source code.
Usually the development of software is accomplished alone and without supervising or reviewing.
In the beginning of the internship the version control system consisted of manually numbering of file names. To keep the overview of versions, which were in productive use, was difficult and the numbering did not protect from overriding changes. Main scripts had very long change logs. A Git repository existed already but basically nobody was using it.

The most of the mentioned problems can be avoided with a small amount of effort. However, it cannot be accomplished by a single person which purposes changes, everybody has to struggle for the quality. If the team is motivated and believes the changes will benefit then the members will adopt even if it means to be restricted in some points and to follow rules.

Chapter 6

Realizations towards better Software Quality

This chapter describes the main activities which were done in course of the internship at the OMC team. Additionally here are the practical realizations to improve the software quality.

6.1 Refactoring and Cleaning

6.1.1 GetLLM

GetLLM (Get Linear Lattice functions and More) calculates a large collection of optics functions and parameters at the BPMs, see also appendix A.1. It was the biggest script in the software base with 6252 PLOC, whereby the main function had 2004 PLOC. Figure 6.1 shows the refactored version which was split in several files. The main script *GetLLM.py* handles now only the command line options, input data and file I/O. The subscripts are invoked by GetLLM and fulfill the computations.

Table 6.1 shows a comparison of PyLint metrics between the old GetLLM and the new GetLLM with its helper modules. Except of the *refactor* issues, all types decreased drastically. Refactor issues increased because there were a few very big functions with refactor issues, like *too many arguments*, *too many branches* or *too many local variables*. After splitting these functions into smaller, the split functions were still big and caused multiple refactor issues instead of one. The task of the output was assigned to the helper class *GetllmTfsFile* which respects the format of Table File System (TFS) files and saves a lot of duplicated code. Thereby the writing of the output into files was extracted to a single spot and thus it was easily possible to produce formatted output which consumes insignificantly more memory and execution time but helps importantly during debugging since it is easily human readable.

Type	Old GetLLM	New GetLLM with helpers
PLOC	6252	6153
SLOC	4422	4097
convention	5764	2517
refactor	98	140
warning	843	33
error	22	2

Table 6.1: This table shows PyLint metrics for the old and the current GetLLM module

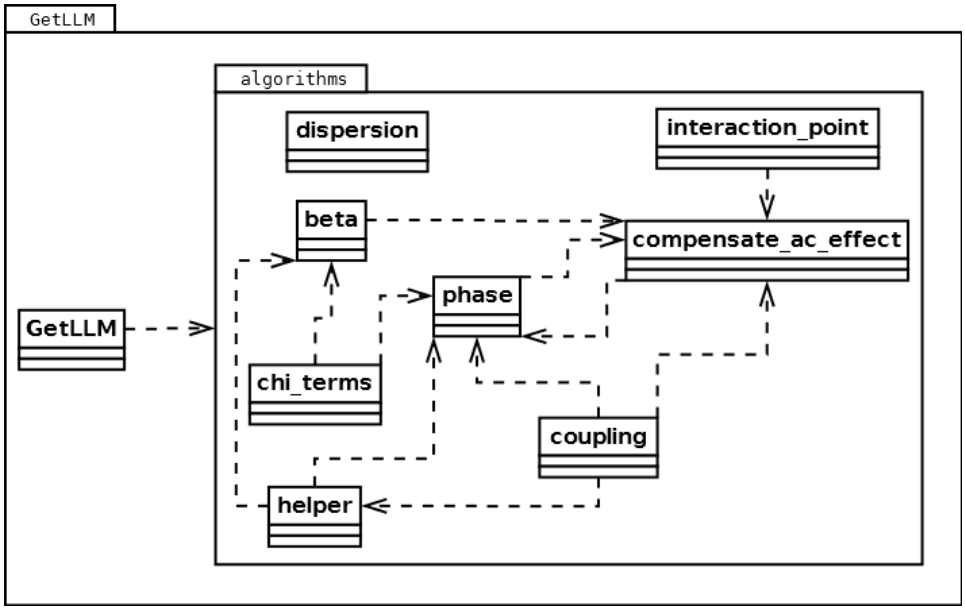

Figure 6.1: The structure of GetLLM after splitting the program in several files

Before any change was done, an automated test was written to ensure the external behavior. The test is based on the simple idea: Run the origin and modified script with the same input and compare the output. To have a high code coverage, multiple input scenarios were integrated. This led to a test duration of over one minute. This is not considered to be an unit test anymore. Unit tests should run within few seconds and avoid IO operations. However, it is not easy to populate unit tests in an established software base but to have no automated tests at all is more negligible.

6.1.2 Other Main Programs

The other main programs in the software base, *SvdClean*, *Drive*, *GenerateFullResponse* and Correction scripts, have been refactored as well, see also appendix A.1 on p.69. Again, the first step, before touching the code, was to write automated tests.
In Python it is possible to execute any statement in the global space[1]. This was abused exhaustively and thus resulted in programs having the most SLOC in the global space. The first action was to extract the logic from the global space into a function and restructure all functions to fit the *Stepdown Rule*, see appendix B.1.4 on p.82. Big functions were divided into smaller and bad identifier of variables and functions have been replaced by meaningful names. Further errors and the most warnings of static analysis tools were removed.
The main reason for the refactoring was maintainability and robustness. Since the programs run, after past improvements, in a reasonable time, performance was not the main quality factor. Profiling tools were not used but unnecessary calculations were removed wherever it was possible.

6.1.3 Robustness

Robustness describes the ability of software to be resistant against wrong user input or interaction and undefined data. A lot of errors occurred during the correction of the optics. Scripts crashed and printed cryptic error messages. Listing 6.1 shows an example of an useless stack trace.

Listing 6.1: Cryptic error message

```
1 Traceback (most recent call last):
2   File "/afs/cern.ch/work/v/vimaier/public/Beta-Beat.src/
      Correction/correct_coupleDy.py", line 423, in <module>
3    _start()
4   ...
5 numpy.linalg.linalg.LinAlgError: Arrays cannot be empty
```

This error message does not reveal the actual cause of the problem. It seems one or more arrays were empty but this is no help for the user. After tracking down the error, it became apparent that given thresholds were too strict and empty lists were the result. Listing 6.2 shows a check and the printing of an useful error message.

Listing 6.2: Cryptic error message

```
1 if 0 == len(couple_list) and 0 == len(disp_y_list):
2         print >> sys.stderr, "Couple and dispersion lists are
      empty. Are model and/or error cuts too strict?"
3         sys.exit(1)
```

[1]The global space describes the space outside of every function or class. Definitions and declarations in the global space are visible/usable everywhere in the executed program.

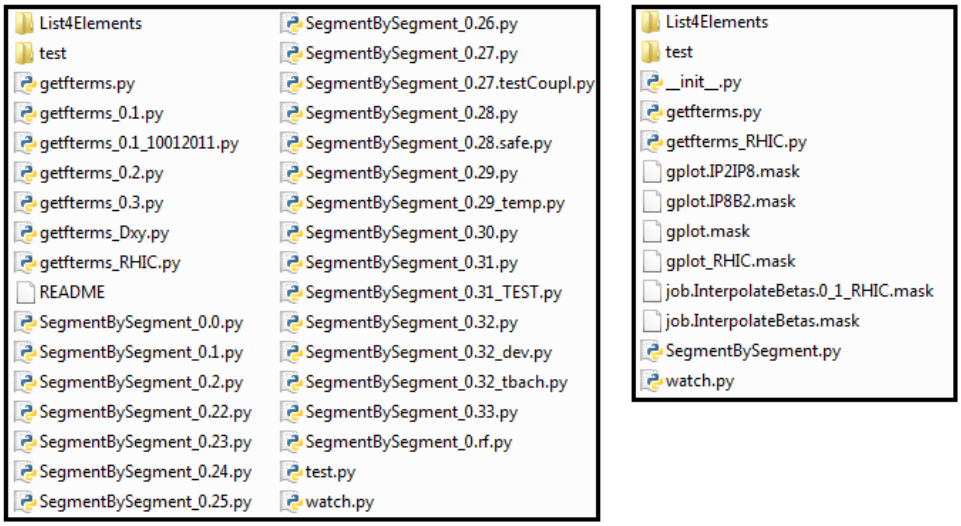

Figure 6.2: The old SegmentBySegment directory(left) compared with the current(right)

This example shows how to implement robustness by terminate the program in a controlled way with a helpful error message.
Five similar errors in GetLLM and seven errors in the other scripts have been suppressed by inserting checks, returning appropriate, empty values, printing helpful error messages and letting the program terminate in a controlled way.

6.1.4 Cleaning Beta-Beat.src

Beta-Beat.src was originally a common directory of the team hosted by CERNs Andrew File System(AFS). Today, it is a Git repository with a central copy on GitHub.[2] Before making a Git repository, the version control system consisted of manually numbering of the files. Today Git does this job, even much better. However, the different versions were still in the repository. Figure 6.2 shows the old content of the *SegmentBySegment* directory compared with the current content. The different versions clutter the directory enormously up.

To have always a working version the practice to develop existing files was to add the suffix *_dev* to files being in development. This is a bad practice and even impacts the code. Listing 6.3 shows it. The snippet checks, in a very complicated way, whether the own file is a development version and loads the appropriate module.

Listing 6.3: The impact of the file suffix *_dev*

[2] https://github.com/pylhc/Beta-Beat.src, visited on 17th Feb 2014.

```
1 if __file__.split('.')[-2][-4:]=='_dev':
2     import GetLLM_dev as GetLLM
3 else:
4     import GetLLM
```

This became obsolete because, again, Git does the job.

6.1.5 Adding Docstrings

A further enhancement was to add *Docstrings*[3] to the main scripts. The most scripts had no documentation about what they do and how to use them. Basic information have been added and Sphinx[4] was used to create documentation from source code.[5] Particularly helpful is the documentation about Drive, the C++ and Fortran program, and how to built it. The Intel compilers *icc*(Linux)/*icl*(Windows) and *ifort* are used to build the binaries. The documentation describes how to obtain them as CERN member.

6.2 OMC Wiki

The OMC team has a wiki with a lot of useful information. It addresses especially beginners but it is helpful for other team members as well.

It began with the purpose to make tutorials for the physicists. They used a text editor instead of a professional IDE to develop programs. To facilitate the changeover from a text editor to an IDE, multiple tutorials were written. It was decided to use HyperText Markup Language (HTML) web pages to make them easy accessible. They contained information about installing the Eclipse IDE with some plugins and how to use EGit[6], see figure 6.3.

After adding further information as static HTML pages, a redesign was necessary to facilitate coming changes. The disadvantage with static HTML pages is that a change in design has to be made in every single file. To avoid duplicated HTML code a small PHP[7] framework was written, see figure 6.4.

In generally, the most dynamic web sites are HTML pages which include header and banner of the web site. The content is put between header and banner, see listing 6.4.

[3]Docstrings are special comments which document and describe classes, methods, functions and variables in source code.

[4]Sphinx is a tool to create documentation from source code, see http://sphinx-doc.org/, visited on 30th Jan 2014.

[5]http://beta-beating.web.cern.ch/Beta-beating/doc/bbeat/, visited on 30th Jan 2014.

[6]EGit is a plugin which brings Git services to the Eclipse IDE.

[7]PHP is a popular general-purpose scripting language that is especially suited to web development, http://php.net/, visited on 20th Jan 2014.

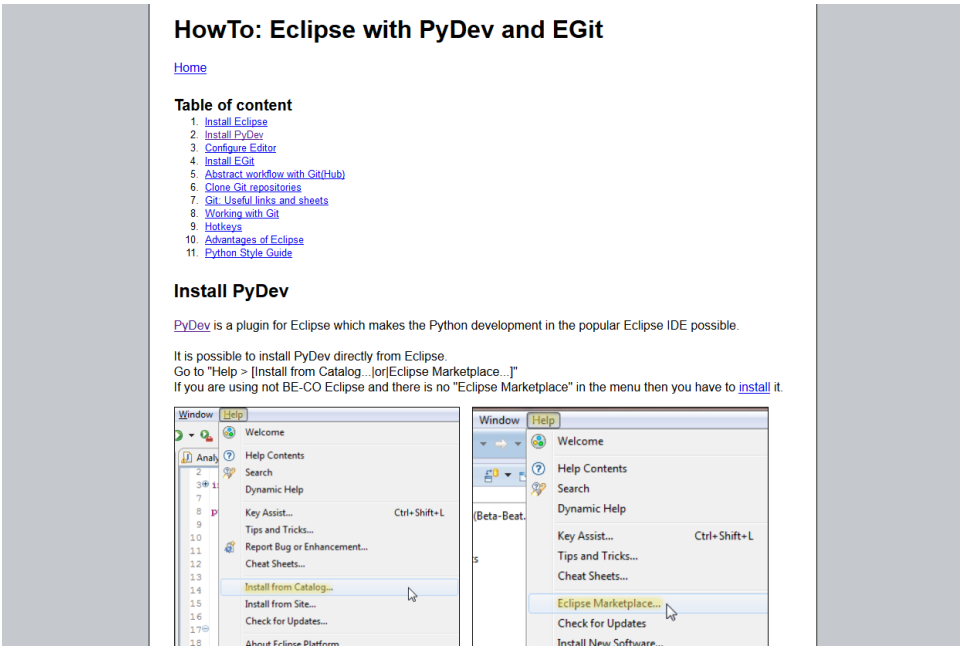

Figure 6.3: The first version of the OMC wiki which consists of static HTML web pages

Figure 6.4: The second version of the wiki. A small PHP framework is used to avoid duplicated HTML code.

Listing 6.4: Including HTML from HTML

```
<html>
    <head>
        <!-- header-information -->
    </head>
    <body>
        <?php include "header.php"; ?>
        <!-- Content-information -->
        <?php include "banner.php"; ?>
    </body>
</html>
```

The big disadvantage is that every file has its own HTML part. Further some parts are located in *header.php* and some in *banner.php*. In bad designs the header even opens a tag and the banner closes it. To avoid having multiple HTML data, spread out over several files, another solution was used. Instead of pulling the HTML parts, header and content information were pushed into one single HTML file, the template. This file was the HTML frame with special keywords which were replaced by the corresponding PHP files. Listing 6.5 shows the abstract source code of every file.

Listing 6.5: Loading and processing of the HTML template

```
<?php
  $html_obj = new HtmlPage(); // HtmlPage loads the template and
    replaces keywords
  ... // Setup header information. Title, extra styles...

  print $html_obj->render_html_code_until_main_content();
?>
      <!-- Content in HTML -->
<?php
   print $html_obj->render_trailing_html_code_after_content();
?>
```

With this approach a web designer has a single file where the whole HTML frame is included. It is easier to edit one physical HTML file than one logical HTML file split in multiple files.
It facilitates to add content but only the developer himself, who has the required knowledge, could do it.

Thus it was looked for a new way to make the collaboration easier and get a real wiki[8]. CERN's IT services provide the wiki implementation *Twiki*. All content was transferred to Twiki, see figure 6.5.

The current OMC wiki[9] has multiple advantages since the service is

[8] wiki – a website that allows collaborative editing of its content and structure by its users. http://www.oxforddictionaries.com/definition/american_english/wiki, visited on 21st Jan 2014.

[9] https://twiki.cern.ch/twiki/bin/view/BEABP/OMC, visited on 21st Jan 2014.

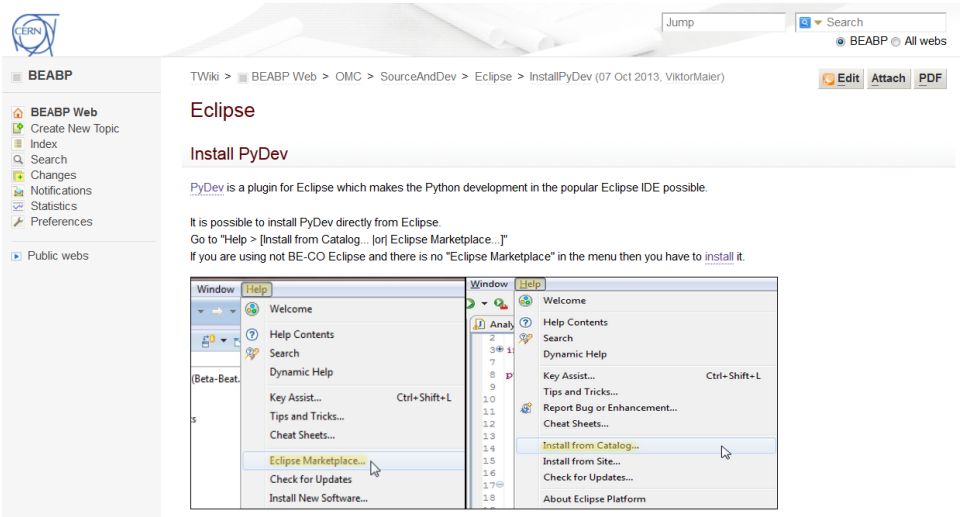

Figure 6.5: Current OMC wiki hosted by CERN's IT service Twiki

provided directly by CERN's IT services. It is without charge for CERN projects. Further it works with the CERN Login service, so authentication and authorization takes place by using CERN accounts.
Twiki offers a plenty of functionality. Every user interaction is possible through a web browser. There is no need for additional software. It is easy to add or edit pages by using the WYSIWYG editor, see figure 6.6. Advanced users can also use Twiki's markup language or HTML code to create the pages.
The revision control of Twiki keeps the history and takes the scare of editing pages. Additionally users can add and manage file attachments.

6.3 Improvement of the Beta-Beat-GUI

The Beta-Beat-GUI supports the correction process of the optics by displaying results in charts and invoking scripts from Beta-Beat.src, see appendix A.2. The Beta-Beat-GUI was developed entirely by a physicists. The most part was refactored and improved by a former technical student, also computer scientist, but there were still some few screens mostly untouched. These screens were refactored and some new features were implemented.

6.3.1 CorrectionSelection and KnobDialog

The both dialogs were completely rewritten.
The user could choose between four correction techniques in the Correction-Selection. These four techniques, four Python scripts in Beta-Beat.src, have

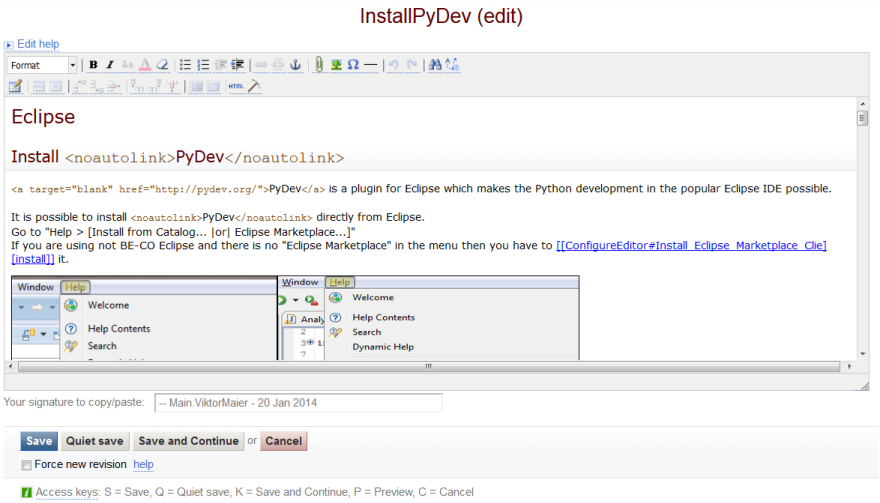

Figure 6.6: Twiki's WYSIWYG editor

some common and some different input arguments. It was possible to enter values which were not used by the correction script. In few cases this was avoided by hiding input fields. Figure 6.7 shows on the left the old and on the right the new dialog. The old dialog hid the field for choosing one or two beams but the label was still present. The new dialog separates common and different fields. The user chooses first the correction technique and the dialog shows only necessary arguments.

The KnobDialog had features which were planned but not implemented. Thus the dialog led to wrong assumptions and empty screens, see figure 6.8. The screen was restructured and only the working features were kept, see figure 6.9. This screen uses the LHC Software Architecture (LSA) API to create or delete knobs[10]. Java Remote Method Invocation (RMI) API is used to communicate with the server. It took several seconds to open the KnobDialog because it is necessary to load remote services and the data from the server. This time was saved with preparing and loading data in the background before opening the KnobDialog.

6.3.2 Frequency Chart

Drive, the C++ and Fortran program, uses the discrete Fourier transform to calculate the containing frequencies and their portions, phases and amplitudes from the BPM signal. The frequencies, also called spectrum, were not displayed in the GUI.
The visualization of the spectrum was implemented as an required feature.

[10] A Knob consists of the magnet names and calculated correction strengths.

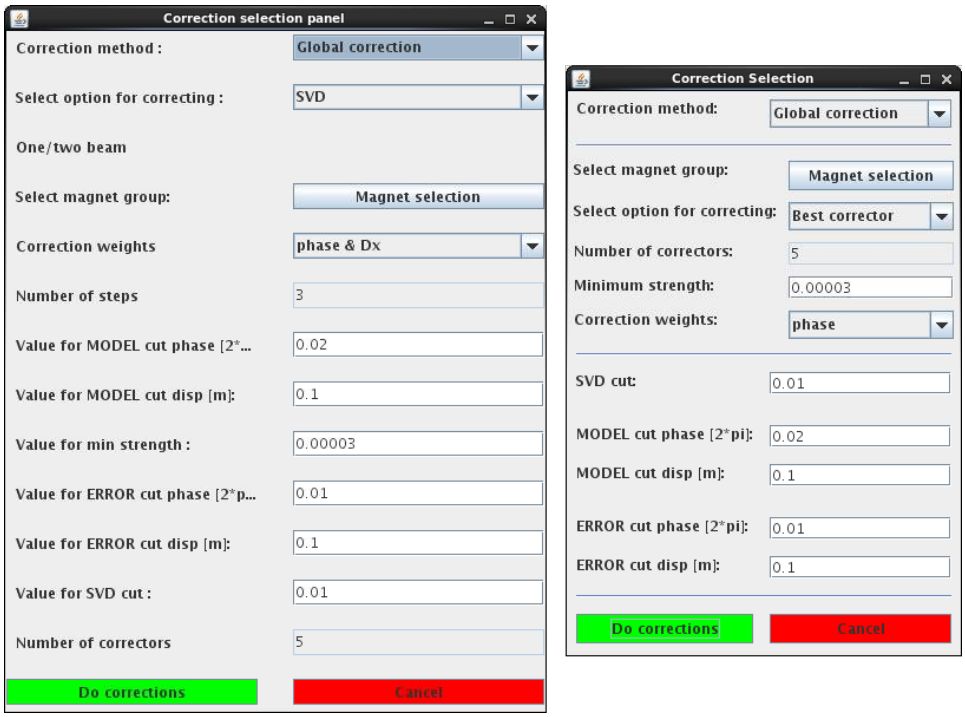

Figure 6.7: The old(left) and the new(right) CorrectionSelection screen

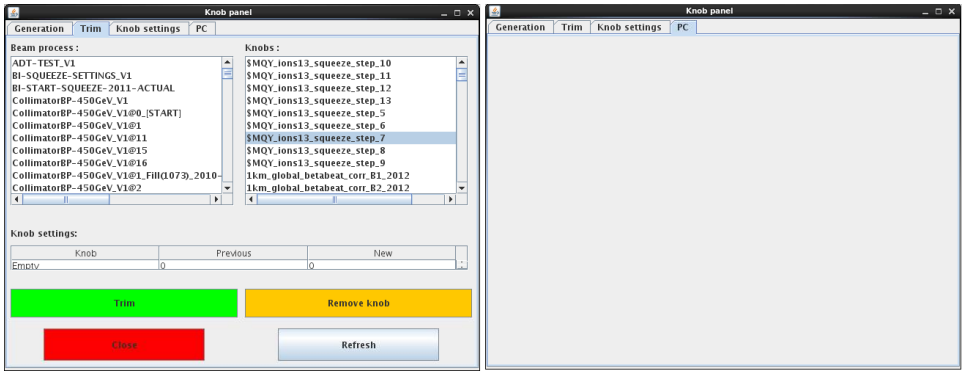

Figure 6.8: The old KnobDialog. The trim function and the PC tab were not implemented

Figure 6.10 shows the outcome.
The input data for the chart was read from TFS files (basically structured text files) which were produced by Drive. The chart was placed into the Analysis-Panel and created by using the JDataViewer package[11].

[11] JDataViewer is a software package, developed at CERN, to easily display data, see http://proj-jdve.web.cern.ch/proj-jdve/, visited on 30th Jan 2014.

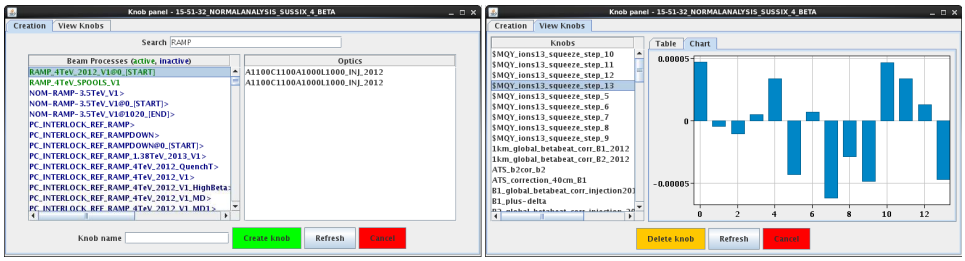

Figure 6.9: The refactored version of the KnobDialog

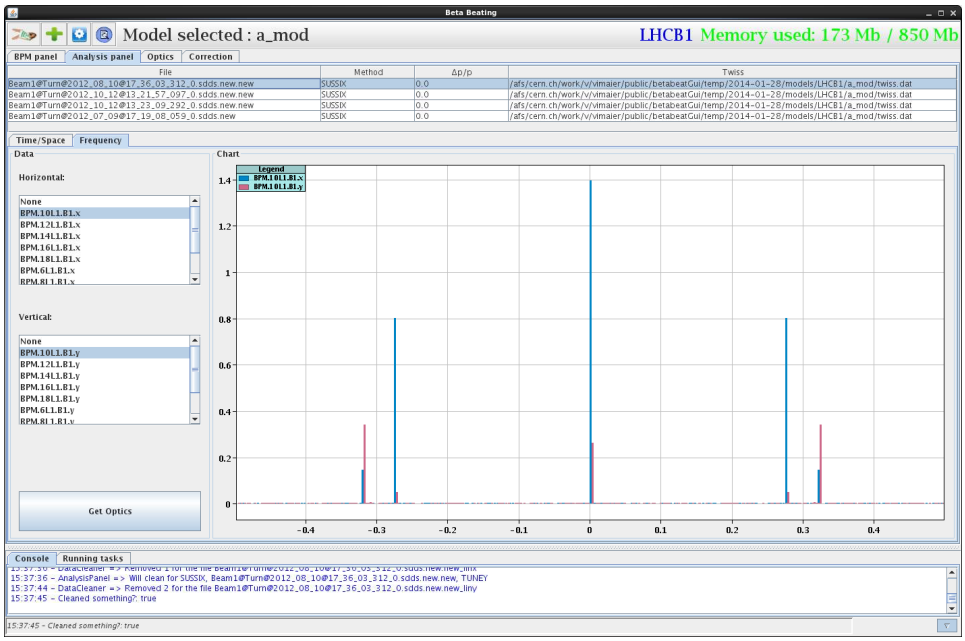

Figure 6.10: Implemented Frequency Chart

6.3.3 Action vs. Tune Chart

A further feature is implemented, the Action vs. Tune Chart. The action is calculated by GetLLM and the tune can be obtained from the output of Drive. The requirement consisted of plotting the action value against the tune value for several input files. Figure 6.11 shows the result.
The chart in the figure has three different points with horizontal and vertical error bars. The user can plot a regression line from the displayed points. The method of least squares[Yan08, p. 189] was used to implement the regression line. The user can also save the currently plotted points to a file. The chart was implemented as an additional tab in the Optics-Panel.

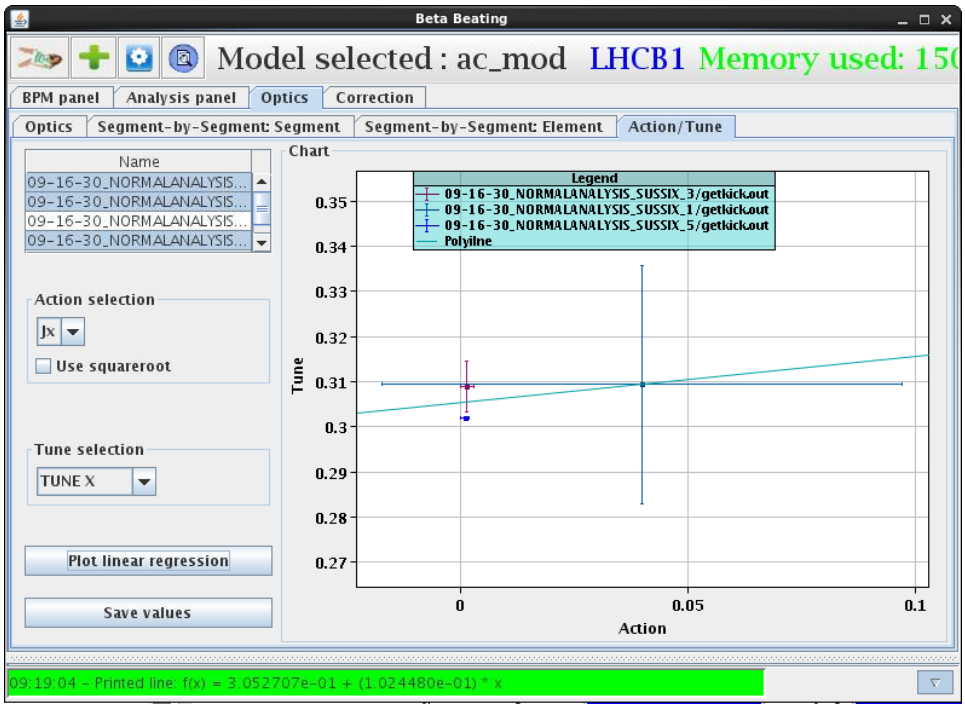

Figure 6.11: Action vs. Tune Chart

6.4 Practices to improve the Software Quality

6.4.1 Automated Tests

Several automated (unit) tests were created in the course of the refactoring. All tests have been added to our test server[12], which executes them every night. Further it checks every hour whether there are new commits in the repository and runs the tests. In case a test fails, the author of the commit receives a report via email.

6.4.2 GitHub Issue Tracker

GitHub is used to store a central copy of Beta-Beat.src. GitHub provides also an issue tracker with many features, like git commit referencing or automatic email notification. The tracker was not used. Problems were solved via email. The disadvantage is that it is not visible for all members and emails disappear with time. Closed issues remain in the issue tracker.

[12]We use CDash, an open source, web-based software testing server, http://www.cdash.org/, visited on 31st Jan 2014.

6.4.3 Coding Lectures and Best Practices Guide

To improve the Software Quality in the long term, the new code has to be better and therefore the creators have to be trained. Thus coding lectures were given to the physicists to improve their skills and to give some advice. The topics were, among others, writing clean code, doing reviews, pair programming and TDD. The slides and sources can be found in appendix C.
In addition a best practices guide was written, containing the helpful recommendations to improve software quality. The guide can be found in appendix B on p.77.

6.5 Continuous Quality Measuring

To track the quality improvement, continuous quality measuring, based on an own assessment model, see 2, was implemented by using Continuous Quality Assessment Toolkit (ConQAT)[13].
This section defines the assessment model, gives a short evaluation of SonarQube[14] and ConQAT and explains the implementation in ConQAT.

6.5.1 OMC Assessment Model

After defining a definition model by using ISO/IEC 25010:2011, see section 2.1, an assessment model for the OMC team was created. This model is restrained to the quality factor *maintainability*. If this factor has a high quality, code modifications are much easier and these are necessary to improve other factors. For example, to improve the performance, the source code has to be read and changed and it is easier to make changes if the source code has a high quality.
The assessment model consists of following metric definitions which will be measured continuously.
The quality Q of the OMC team is defined as

$$Q = (\bar{P}, S) \tag{6.1}$$

with

- the number of relative PyLint[15] issue points $\bar{P}$ and
- the function size in S.

Relative PyLint Issue Points $\bar{P}$
PyLint produces different categories of issues. It will be distinguished be-

[13] https://www.cqse.eu/en/products/conqat/overview/, visited on 7th Feb 2014.
[14] http://www.sonarqube.org/, visited on 7th Feb 2014.
[15] PyLint is a static analysis tool for Python, http://www.pylint.org/., visited on 8th Feb 2014

tween errors and other issue categories(warnings, refactoring and convention). The following aggregation operator is used to calculate the issue points:

$$P = A(i, e) = i + 5e \tag{6.2}$$

- i is the number of found issues other than error
- e is the number of found errors

The factor 5 is based on the calculation of PyLint's code rate:[16]

```
10.0 - ((float(5 * error + warning + refactor + convention) /
    statement) * 10)
```

The relative issue points $\bar{P}$ gives the issue points relative to hundredth PLOC:

$$\bar{P} = \frac{P}{\text{PLOC} * 1/100} \tag{6.3}$$

The goal is to have no PyLint issues, $\bar{P} = 0$. However, it is not easy to remove all issues in the software base. The value should fall slowly with new implementations and code changes but not increase. There are two methods to decrease $\bar{P}$. One method decreases P, achieved by Refactoring, and the other method increases PLOC, achieved by new implementations without PyLint issues.

Function Size S

The Function Size S is the tuple

$$S = (|G|, |Y|, |R|) \tag{6.4}$$

with the numbers of *green* $|G|$, *yellow* $|Y|$ and *red* $|R|$ functions. The color indicates the size range of a function where *green* is the preferred range, *yellow* is the acceptable range and *red* is the range which should be avoided. The function size f =*PLOC of function body* is categorized as

$$c(f) = \begin{cases} green & \text{if } f \leq 20 \\ yellow & \text{if } 20 < f \leq 100 \\ red & \text{if } f > 100 \end{cases} \tag{6.5}$$

The values are based on Martins statements, he writes that functions *should not be 100 lines long* and *should hardly ever be 20 lines long*[Mar09, p. 34].

G is the set with all *green* function sizes, Y consists of all *yellow* function sizes and R holds all *red* function sizes:

$$\forall f \in G : c(f) = green \tag{6.6}$$

[16] http://docs.pylint.org/faq.html#pylint-gave-my-code-a-negative-rating-out-of-ten-that-can-t-be-right, visited on 19th Feb 2014.

$$\forall f \in Y : c(f) = yellow \tag{6.7}$$

$$\forall f \in R : c(f) = red \tag{6.8}$$

If F is the set with all function sizes $\{f_1, f_2, \ldots, f_n\}$ of all functions in the project, then G, Y and R are disjoint subsets:

$$G \cup Y \cup R = F \tag{6.9}$$

$$G \cap Y \cap R = \emptyset \tag{6.10}$$

The aim is to avoid the production of big-size functions and to reduce the numbers $|Y|$ and $|R|$. This can be achieved by extracting sections in big functions. It is also not practical to reduce the numbers to zero but they should not increase.

These are few metrics but they point at the major problems in the code, big functions and the violation of conventions.
Classes were not emphasized since the OMC team rarely uses them. Available methods of classes are treated as functions.

6.5.2 SonarQube vs. ConQAT

Both, SonarQube and ConQAT, have been considered to implement the assessment model and measure the quality continuously.

SonarQube is a open source platform to measure and track code quality. Figure 6.12 shows an overview of the platform.

The platform consists basically of the analyzers, the database and a own web server. The analyzers examine the source code and save the results into the database. The web server loads the results from the database and displays it in form of HTML pages.
It is highly customizable and extendable via plugins. A Python plugin exists as well. This plugin uses PyLint, the static analysis tool for Python, to detect and display issues. Thus an assessment model would be already prescribed.

ConQAT is a toolkit to analyze software quality. It runs on source code and produces static HTML output which can be displayed by every common web server. An analysis is based on a ConQAT configuration file and a top-level block which describe what will be measured and displayed in the output. An Eclipse plugin can be used to create such a configuration file, see figure 6.13. ConQAT uses a *pipes and filter*[18] approach to create configurations or assessment models. Configurations consists of

[18]Pipes and filter is a architectural design pattern, see http://www.cs.olemiss.edu/~hcc/csci581oo/notes/pipes.html, visited on 21st Feb 2014.

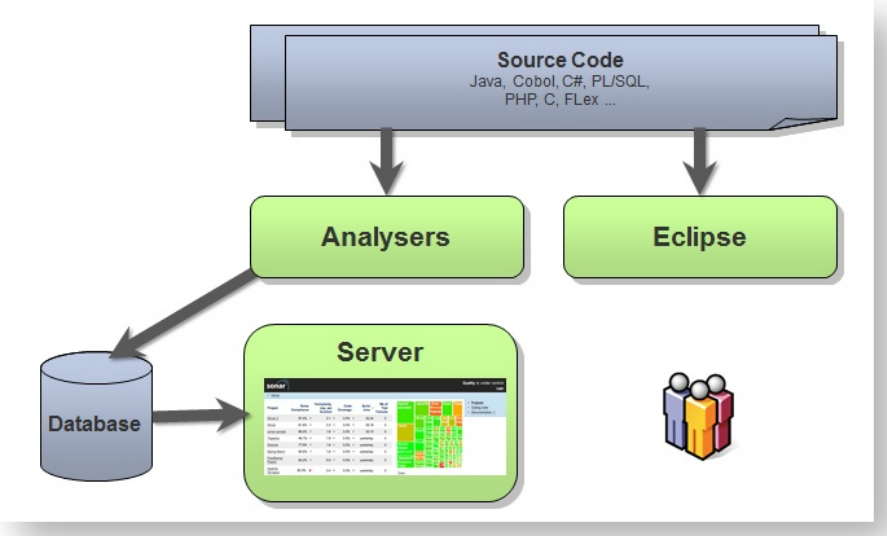

Figure 6.12: Overview of the SonarQube platform[17]

- processors,
- blocks and
- transitions between these elements.

Blocks are compound of processors and transitions and used to deliver functionality based on several processors on a more abstract level. The programming languages Java, C++ and C# are well supported. For Python, own processors have to be written.

The decision was made in favor of ConQAT. To support Python, own implementations are necessary but the integration of ConQAT is much easier. SonarQube needs a own web server and thus it's not easy to integrate it in the existing infrastructure at CERN.

6.5.3 Implementation of Continuous Quality Measuring

The created ConQAT implementation measures and tracks PLOC, function sizes and PyLint issue points. The created dashboard shows a chronological overview of the metrics, see figure 6.14. The Java source code[19] and the reference manual[con] were used for the implementation. The extensions, written in Java, are oriented towards existing processors.

[18] Figure taken from http://docs.codehaus.org/display/SONAR/Installing, visited on 9th Feb 2014.

[19] http://www.conqat.org, visited on 17th Feb 2014.

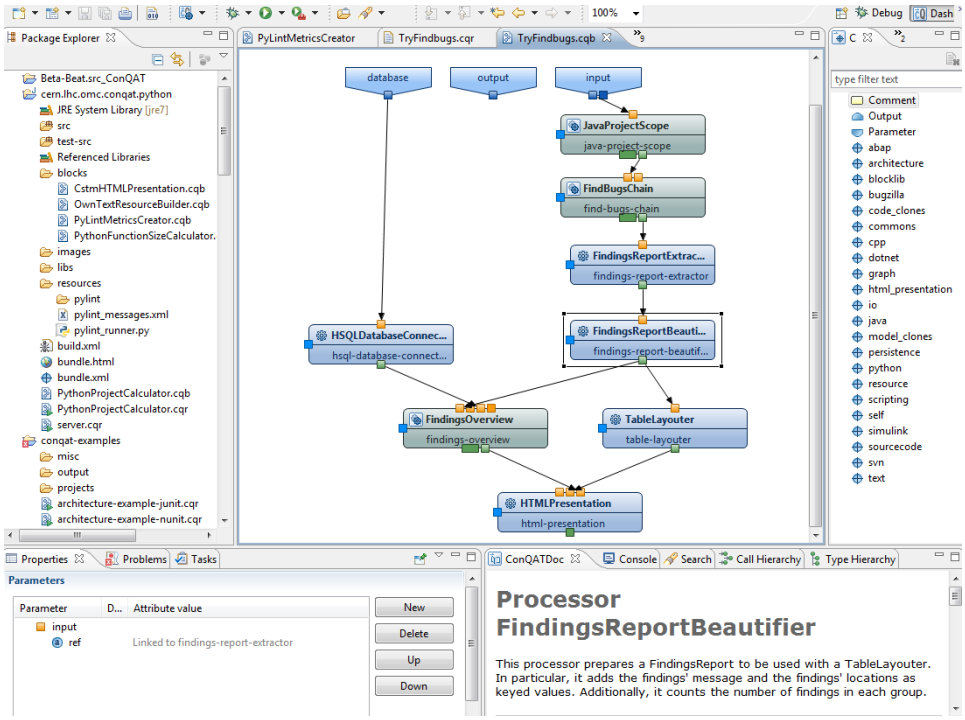

Figure 6.13: ConQAT Eclipse Plugin

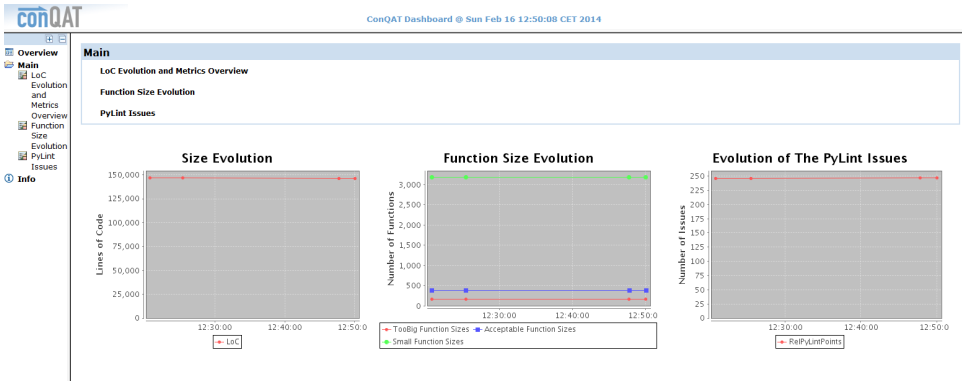

Figure 6.14: Overview page of the created ConQAT dashboard[20]

The ConQAT block *PythonProjectCalculator.cqb* is the main configuration to calculate the metrics. It consists of the following blocks:

1. PyLintMetricsCreator
2. PythonFunctionSizeCalculator
3. CstmHTMLPresentation

In ConQAT, the configuration file can become quickly confusing and the blocks abstract the logic behind.

PyLintMetricsCreator
: This block calculates all metrics which are necessary to obtain the relative issue points $\bar{P}$. It executes the self-written processors *PyLintExecutor*, *PyLintReportReader* and *PyLintIssueCounter* to determine for every module[21] the PyLint errors and other issues. Built-in *SumAggregators* and *Calculators* are used to calculate the relative issue points from equation 6.3.

PythonFunctionSizeCalculator
: This blocks calculates the function size S. The self-written processor *FunctionSizeCounter* determines the numbers of *green*, *yellow* and *red* functions for every module. *SumAggregators* sum the numbers for every directory up to the root of the project.

CstmHTMLPresentation
: This block creates the customized output, static HTML pages by using existing processors and blocks for HTML representation. The overview page, see figure 6.14 shows trend charts of the values PLOC, $\bar{P}$ and S. Enlarged charts and tables with all calculated metrics can be found in the sub pages *LoC Evolution and Metrics Overview*, *Function Size Evolution* and *PyLint Issues*. A file-based HSQLDB database and existing processors are used to store and track PLOC, $\bar{P}$ and S.

The sources can be found in appendix C. **Continuous Software Quality Measuring**

The tool *acrontab*[22] and git hooks[23] are used to measure continuously software quality. The following cron job, running on the server cluster *lxplus* at CERN, invokes every hour a *git pull* on a special repository to synchronize the repository with our centralized repository on GitHub.

```
1 # Min Hour DayOfMon Mon WeekDay Host Command
2 0 * * * * lxplus cd /afs/cern.ch/work/v/vimaier/public/conqat/
      Beta-Beat.src && git pull >> /afs/cern.ch/work/v/vimaier/
      public/conqat/acrontab_job_git_pull.log 2>&1
```

The output is logged in a file. In case there are new changes (git commits), *git merge* is automatically called and the git hook *post-merge* is invoked. This

[20] http://beta-beating.web.cern.ch/Beta-beating/quality/, visited on 17th Feb 2014.

[21] In Python, a module is called a source code file with the ending *.py*. It can contain several classes and functions or even nothing.

[22] *acrontab* is a job scheduler, based on the Unix tool cron, see http://cnlart.web.cern.ch/cnlart/221/node28.html, visited on 17th Feb 2014.

[23] Git Hooks is a way to invoke custom scripts in case of special events, see http://git-scm.com/book/en/Customizing-Git-Git-Hooks, visited on 17th Feb 2014.

hook executes the ConQAT top-level block *PythonProjectCalculator.cqb* to create the HTML output.
This client hook is time-based, not event triggered, but a server-side hook, which can be fired for every push, on external GitHub servers is harder to integrate in the existing IT infrastructure at CERN.

Occurred Problems and Obstacles
The lack of documentation and other sources of information, like an online community, slowed the implementation down. A reference manual exists, but it is out of date with the current sources and discusses only roughly the extension of ConQAT. At least the existing source code is in a very good shape, although it took time to understand the abstract design of relevant parts in the software.
For the continuous measuring, the implementation had to be ported from a Windows computer to the Linux servers. Since the whole implementation is based on Java, it was assumed to be easy. Symbolic links let the existing ConQAT processors fail badly.[24] For example, one directory contains the symbolic link *pymad.py* which points to the Python module *pymad-0.4.py*.[25] The processor *FileSystemScope* follows, on the Linux server, the symbolic link and tries to add twice *pymad-0.4.py* in the same directory. This let the processor crash. The relative path to the symbolic link was given as an input parameter to *FileSystemScope* to exclude this file.
Another obstacle was the old PyLint version on the servers which do not provide a necessary command line argument to format its output. The PyLint sources of the new version were put directly in the *resources* folder of the own ConQAT project. This ensures that always a valid version is called. Another option was to request an update but this would take some days.

6.6 Comparison of Beta-Beat.src

Table 6.2 shows metrics from a ConQAT analysis for a current Beta-Beat.src and a eleven months old one. More than a half of PLOC were reduced. This was achieved by cleaning the repository from obsolete and not used scripts. All other values decreased as well.

[24] git stores symbolic links on Windows as plain text files. The content is the pointer of the link. Thus this problem did not occur before
[25] This is also an example of the obsolete versioning system of the OMC team.

Type	April 2013	February 2014
PLOC	331,312	141,195
SLOC	206,690	88,266
PyLint warnings/issues i	468,721	157,502
PyLint errors e	10,959	8,029
Relative PyLint points $\bar{P}$	158	140
Small functions $\|G\|$	2,959	2,939
Acceptable functions $\|Y\|$	1,544	754
Big functions $\|R\|$	527	121

Table 6.2: This table shows metrics from the ConQAT analysis for the old and the current Beta-Beat.src

Chapter 7

Conclusion

Multiple improvements were achieved within the thesis. By refactoring the main programs and removing obsolete source code the software base was cleaned and the execution is more robust. Thus the programs don't crash for every small error and deliver more results. This is very helpful for the analysis of the optics. Cleaner code facilitates further changes and corrections to the programs. It will be easier for the physicists to adapt the code to changes from LS1 (long shutdown 1).
Many techniques from professional software development were applied to create a more professional development environment. The created and automated tests will take the scare of code modifications for future improvements. In case a modification will let the tests fail, an email will be sent to the author of the commit. The version control software git ensures, that it is easy to checkout a valid version and to track code changes to easily identify the cause. This will be very useful if the team is in course of corrections in the CERN Control Centre (CCC)[1]. If the current version is not working, then a previous version can be tested easily. Further the issue tracker will keep any kind of issues so that they will not be forgotten. Newcomers in the team are going to find their way with the help of the OMC wiki much faster. But also current members use the wiki as a common knowledge base and reference text.
Besides of improving the existing source code and creating a more professional software development environment, the physicists, who will write future source code, were sensitized to develop more professionally. To use the Eclipse IDE with static analysis tools, instead of text editors, was encouraged in several meetings, the coding lectures and the best practices guide. In addition common mistakes in the structure and the readability of the code were pointed out. Differences between existing, bad code and clean code and how to achieve the latter were shown. Newcomers have a good introduction

[1] http://public.web.cern.ch/public/en/spotlight/SpotlightCCC-en.html, visited on 15th Jan 2014.

to the principles of clean code with the best practices guide. However, first future developments will show if the physicists will apply what they have learned.
Therefor a mighty tool was implemented by using ConQAT, the continuous measuring of software quality. It will reveal if future implementations increase the quality. In the case of becoming worse, appropriate and aimed steps can be initiated to improve the added source code.
However, there are also fields which could not be established. Chapter 3 showed how important software development models are. It has been a field of research for many decades and still is one. A structured process in the OMC team would lead to much better software quality. One possible and tailored model was proposed in the best practices guide. If the members would took it into account, the productivity and quality would benefit. Testing, chapter 4, is also very important to produce and ensure high quality products. Despite of the automated tests, which are very helpful, no other techniques were populated. The TDD approach and informal reviews were strongly emphasized and presented at several occasions. The physicists need more training to practice TDD. But the improved environment, especially the combination of git, GitHub and the issue tracker, is a good foundation to perform informal reviews.
The ConQAT realization is just a small step into continuous quality control. Of course, the OMC team is not a software development team with the goal to satisfy customers. But there are many techniques which can be applied to the OMC team as well. As future work, the quality assessment model and the ConQAT implementation could be extended. For example, internal dependencies of functions could be analyzed to see if the module is in accordance with the *Stepdwon Rule*, see appendx B.1.4 on p. 82. External dependencies could be counted since they affect the maintainability in a negative way. Test coverage, which measures the amount of code, covered by automated tests, is a metric which helps to create further directed and specific test cases. A further interesting task is the implementation of a delta analysis[GD], where metrics of a baseline will be compared with a subsequent version. This could also be combined with an automatic email notification. This instrument would help to ensure better software quality of future developments.
For all other similar teams, the best practices guide serves as a small guideline to improve the software development.

Appendix A

OMC software

This is a rough description of the existing software base. Only the main applications are included.
The existing software consists mainly of two different repositories, Beta-Beat.src and lhc-app-beta-beating. The former is a Git repository with the programs for the scientific calculations. The latter is a SVN repository and contains the code of the Beta-Beat-GUI.

A.1 Beta-Beat.src

The Git repository Beta-Beat.src contains mainly Python scripts to measure and correct the optics. Further there is a combination of C++ and Fortran code, Drive/SUSSIX. Figure A.1 shows an overview of a correction process and used programs.

The turn-by-turn-data serves as input file, is formatted in Self Describing Data Sets (SDDS) and contains BPMs with the measured positions of a beam with several turns. The script "SVD clean" cleans the data by using the singular value decomposition and removing noise and bad measured BPMs.
The cleaned TBT-data will be processed by Drive, ac C++ program. Drive is a wrapper for the Fortran code. Drive prepares the data and invokes SUSSIX, a frequency analysis written in Fortran. The result, frequencies, phases and amplitudes for all BPMs, is written in TFS-files.
Using the output from Drive and a model, GetLLM calculates a large collection of optics functions and parameters at the BPMs. This model is created by MAD and represents the 'perfect' reference for the beam optics.
The correction scripts use the output of GetLLM and *fullresponse* to calculate correction strengths for the magnets. The fullresponse consists of multiple models with incrementally increasing magnet strengths.
Figure A.2 shows the abstract work flow using Git.

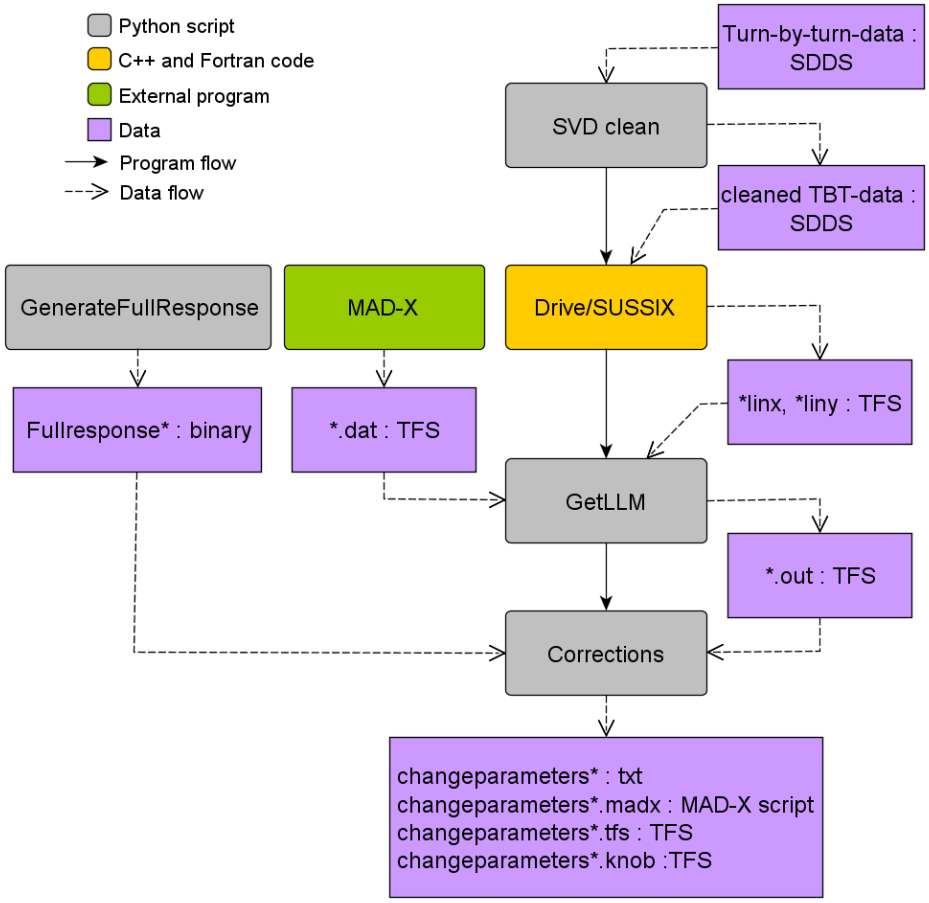

Figure A.1: An overview of the involved programs of the optics correction

A.2 Beta-Beat-GUI

This application is stored in the SVN repository lhc-app-beta-beating. It needs to be in a central SVN repository because the application is build and released by using *Commonbuild*[1].
The Beta-Beat-GUI supports the correction process. It loads the data to visualize it and invokes the scripts from Beta-Beat.src. Figure A.3 provides an overview of the interactions. Figures A.4 to A.8 show the panels of the software.

[1]Commonbuild is a build and release management tool developed at CERN to satisfy the specific needs. See https://wikis.cern.ch/display/DVTLS/Commonbuild+User+Manual, visited on 30th Jan 2014.

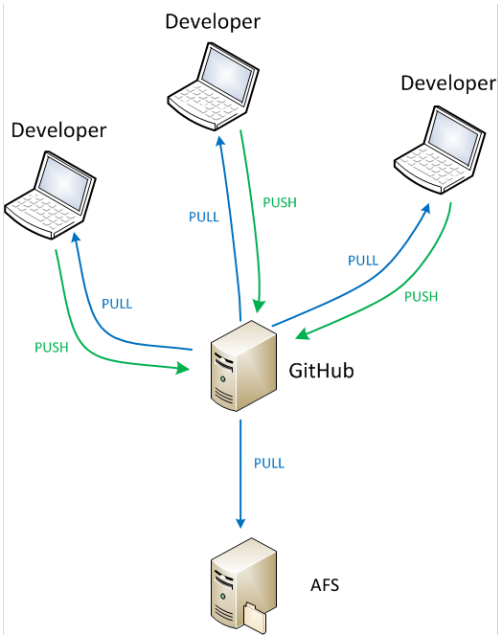

Figure A.2: The abstract work flow of the Beta-Beat.src repository

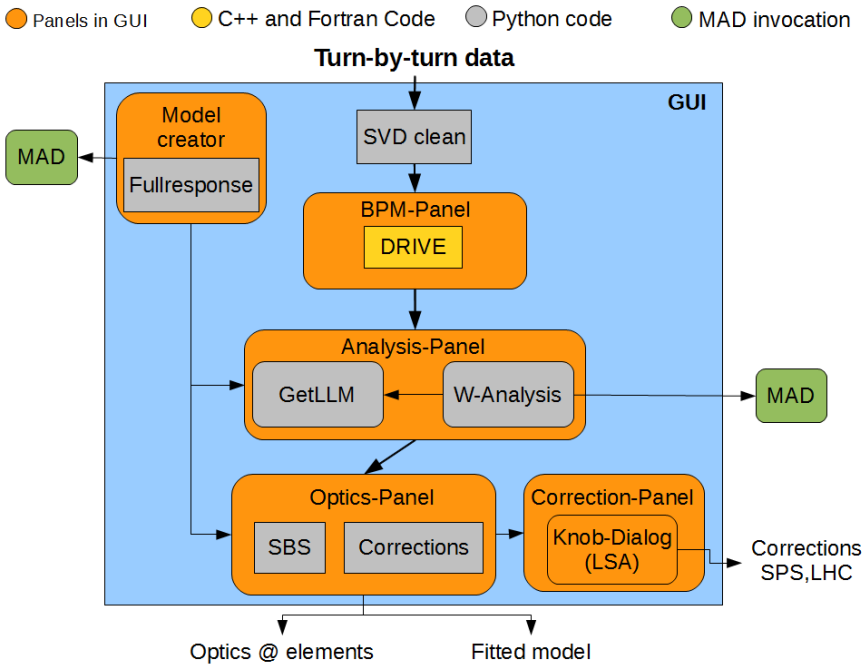

Figure A.3: This overview shows the interaction between the Beta-Beat-GUI and the correction process with external programs

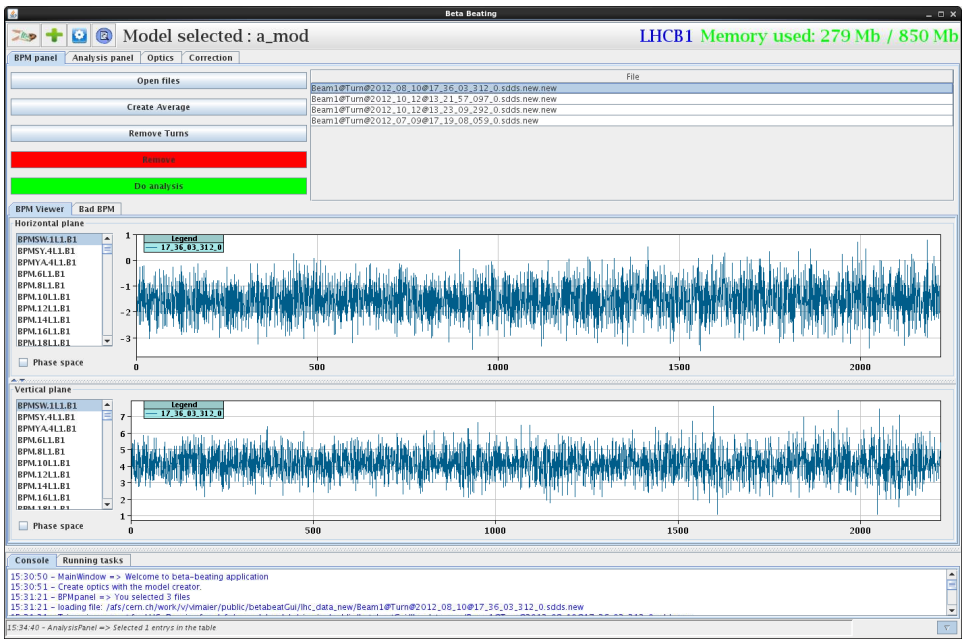

Figure A.4: The BPM-Panel loads the turn-by-turn-data and displays it separately for the horizontal and vertical plane.

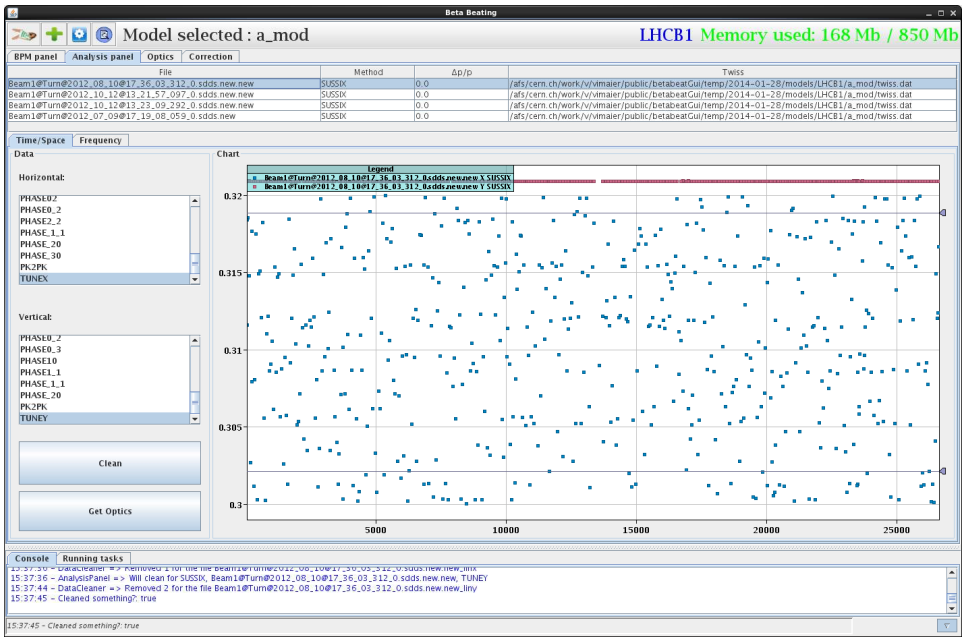

Figure A.5: The Analysis-Panel displays the output of the harmonic analysis(SUSSIX).

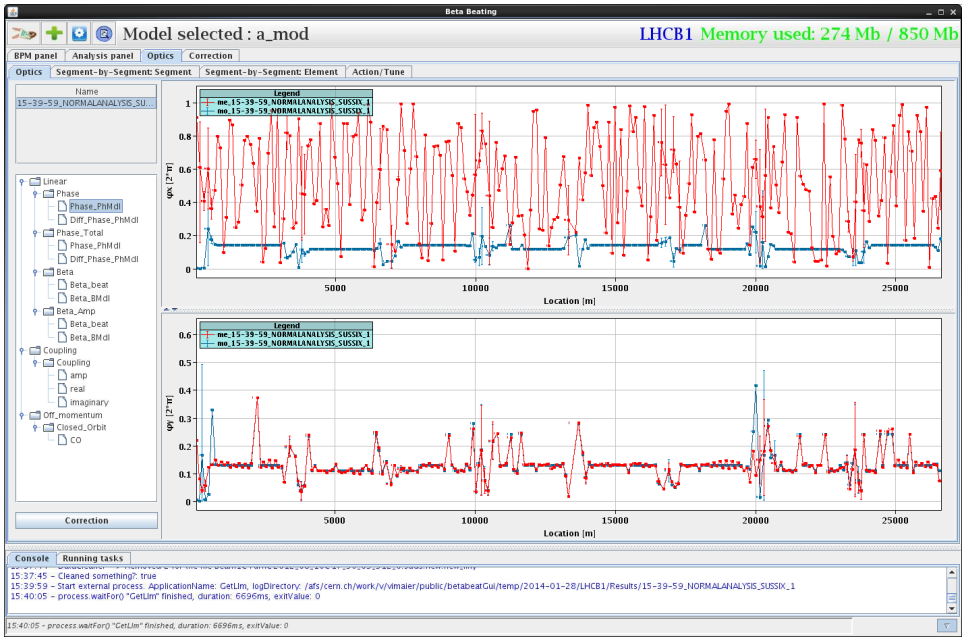

Figure A.6: The Optics-Panel shows the optics functions and parameters produced by GetLLM.

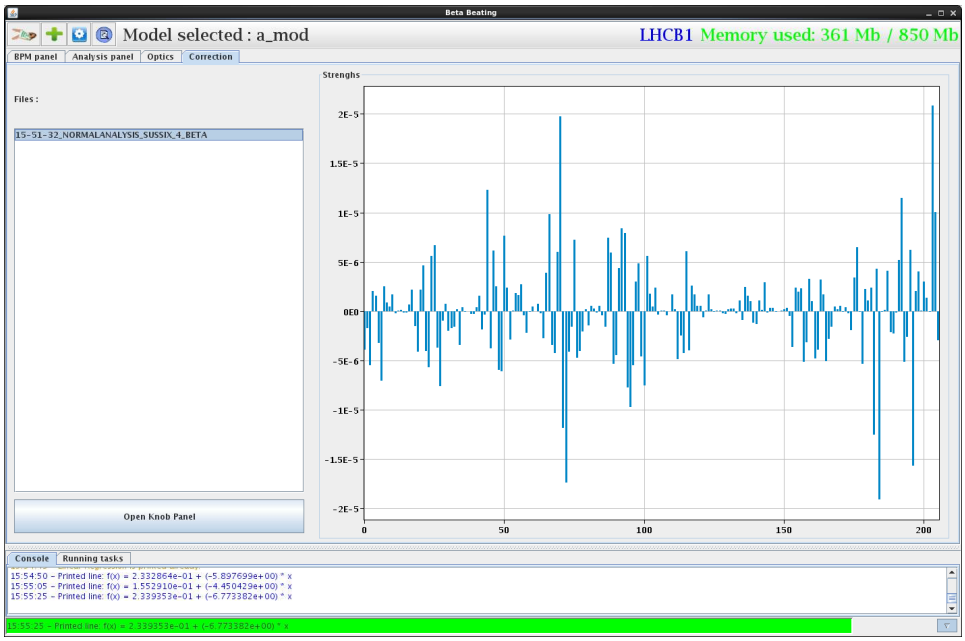

Figure A.7: The Correction-Panel visualizes the computed correction strengths for the magnets.

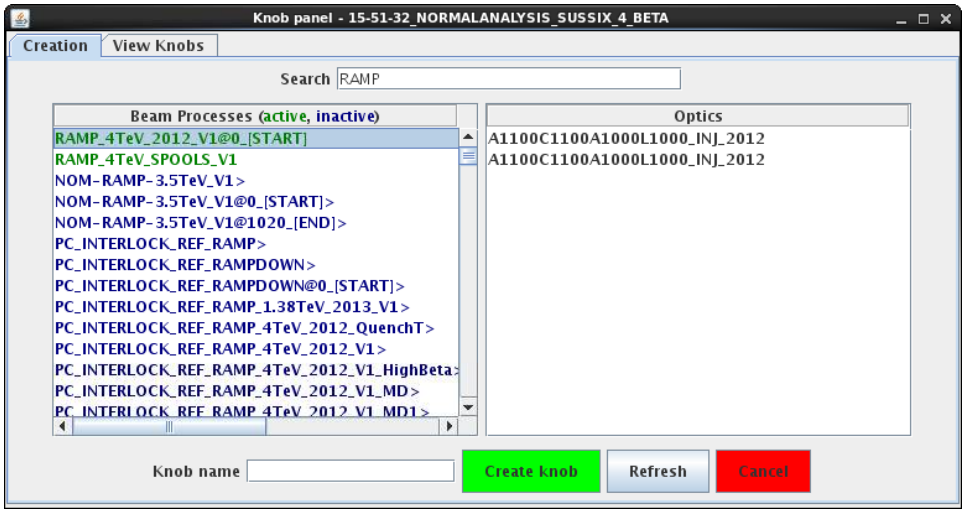

Figure A.8: The Knob-Dialog is used to send the correction strengths of the magnets to LSA

Appendix B

Best Practices Guide

This guide represents a short collection of practices which can be easily adopted by everybody to improve the quality of software.

B.1 Clean Code

Clean Code has many characteristics like easy to read, elegant, beauty and much more. Following are two facts about clean code from [Mar09]:

- *Clean code always looks like it was written by someone who cares.*(Michael Feathers)
- *You know you are working on clean code when each routine you read turns out to be pretty much what you expected.*(Ward Cunningham)

The snippet beneath, found in GetLLM, is obviously not in accordance with the facts:

```
1  def function(x):
2      fterm=4*abs(x[0])
3      main=2*sinh(fterm)
4      single=sinh(fterm)
5      first=cosh(fterm)*sin(x[1])
6      second=cosh(fterm)*sin(x[1]+2*phi12_nlinear)
7      fun1=float((main*(single+first))-bb[0])
8      fun2=float((main*(single+second))-bb[1])
9      fun=[fun1,fun2]
10
11     return fun
```

It does not seem that the author cared about the code and the function will be rather a surprise than what one would expect. There are simple rules which help to avoid such code. One might think the following recommendations are quite natural but the fact is that all the rules were violated and thus software with bad quality was written.
The important fact should be mentioned that it is possible to write good

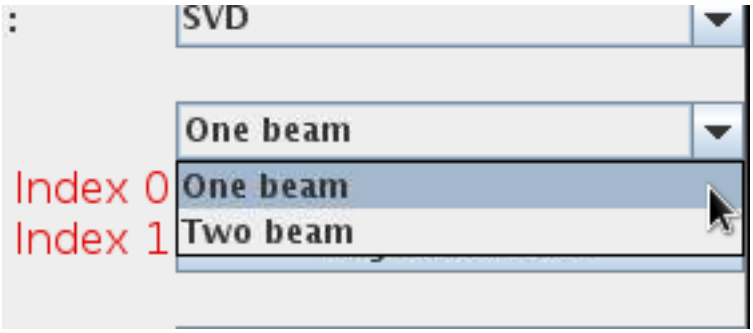

Figure B.1: Very bad design: The index of the entries is passed to the correction scripts to determine number of beams

code from the beginning but it is almost impossible to achieve very clean and well designed code right the first time. The latter is gained by iterative improvements.

B.1.1 Names

Names or identifier of variables, functions, methods, classes and package or module names should always reveal the intent of the programmer. A developer should constantly take time to choose the best possible identifier. To give names to variables only because they need to be referenced in the program is a very bad attitude. Identifiers are a part of the documentation as well.
Names with a single letter can be used and should be used in mathematical equations where the identifier is well known, for example c for the speed of light. However, the scope of single letter identifier should be strictly limited to a small local area like a function with 10 PLOC.
The following example shows how important it is to use intention revealing names:

```
1 if options.JustOneBeam=='1':
2     ...
```

A reader with knowledge about digital technique would immediately conclude if *JustOneBeam* equals 1 then the logic means there is only one beam.[1] This is wrong. In this example the value and the meaning are not in accordance. The fact is that if the condition evaluates to true then two beams should be used. The number 1 represents the index in a combo box, see figure B.1.

It was already a bad design decision to pass the index but a possible solution for this case could be:

[1] In digital technique a 1 represents true and 0 represents false.

```
use_two_beams = options.JustOneBeam == '1'
...
if use_two_beams:
    ...
```

The identifier of functions should not be chosen arbitrary. The best possible name should be given and the function should do exactly what the name says.

```
def BPMfinder(IP,model,measured):
    ...
```

```
def get_closest_bpm_names_to_ip(ip_num,  model,  measured):
    ...
```

The latter version describes more precisely what the function does. The tools of the can be used to easily rename names in whole projects.
Another big problem is the usage of *magic numbers*, number literals[2] placed directly in code:

```
for j in xrange(34):
    s += (t[j]*4)/5
```

```
for j in xrange(NUMBER_OF_TASKS):
        real_task_days = task_estimates[j] *
    real_days_per_ideal_day
        real_task_weeks = real_task_days / WORK_DAYS_PER_WEEK
        sum_of_task_weeks += real_task_weeks
```

The latter one is much longer but it expresses the intention of the calculation. It is much more difficult to understand and change the former version.
It should be avoided to give misleading information. In an example from [Idr13, p. 125] the following equation to produce a square waved signal

$$f(t) = \sum_{k=1}^{\infty} \frac{4 \sin\left((2k-1)t\right)}{(2k-1)\pi} \tag{B.1}$$

was implemented with the solution:

```
t = np.linspace(-np.pi, np.pi, 201)
k = np.arange(1, float(sys.argv[1]))
k = 2 * k - 1
f = np.zeros_like(t)

for i in range(len(t)):
    f[i] = np.sum(np.sin(k * t[i])/k)

f = (4 / np.pi) * f
```

[2] in computer science, a literal is a notation for representing a fixed value in source code.

To improve the algorithm, $2k - 1$ was calculated in advance and replaced with k. This is not a mistake but the fact that the same name was used for different things, is misleading and should be avoided. The simple solution is to give another identifier.

B.1.2 Functions

The most important rule about functions is to keep them small. They should only do one thing and name should exactly express it. If functions are long they probably do more than one thing and it is difficult to keep an overview. Often developer separate their big functions into regions and comment every region:

```
1 ...
2 #-------- START Phase
3 ...  # A lot of code
4 #-------- START Beta
5 ...
6 #-------- START Orbit
7 ...
```

It clutters up the code and shows unnecessary details which might not be of current interest. The following shows the extraction of the code:

```
1 ...
2 calculate_phase()
3 calculate_beta()
4 calculate_orbit()
5 ...
```

A reader has an overview of the code and can examine the details if necessary.
Further functions can be used to bury down unnecessary details and decisions which are not relevant for the other logic.
Cloned or duplicated code is a smell for bad code and should be always avoided. Functions can be used to extract the cloned code.
The amount of the parameters should be kept small, ideally zero. Functions with more than three parameters should be avoided. A method to decrease the number of parameter is type rich programming, see B.1.8.

B.1.3 Comments

Comments should be avoided where possible. Introductory programming courses emphasize to leave comments to be able to read the code easily after some months of absence. This approach should be avoided. The code should tell the programmer the logic and the intention by using useful identifier and short functions. Comments are excuses for bad code. The time on writing comments should be spent on cleaning the code.
The most comments can be omitted and replaced by sensible identifier:

```
def f2h(...):
    """ converts from f-term to h-term """
    ...
```

```
def convert_from_f_to_h_term(...):
    ...
```

The following listing contains a comment for a region. The code can easily be extracted.

```
#-- bet and alf at the left BPM
if plane == 'H':
    betl = mad_twiss.BETX[mad_twiss.indx[bpml]]
    betdl = mad_ac.BETX[mad_ac.indx[bpml]]
    alfl = mad_twiss.ALFX[mad_twiss.indx[bpml]]
    alfdl = mad_ac.ALFX[mad_ac.indx[bpml]]
if plane == 'V':
    betl = mad_twiss.BETY[mad_twiss.indx[bpml]]
    betdl = mad_ac.BETY[mad_ac.indx[bpml]]
    alfl = mad_twiss.ALFY[mad_twiss.indx[bpml]]
    alfdl = mad_ac.ALFY[mad_ac.indx[bpml]]
```

```
betl, alfl, betdl, alfdl = get_alfa_and_beta_from_bpm(mad_twiss,
    mad_ac, bpml, plane)
```

Further, comments which deliver no additional value clutter the code. In the following example, the comment has the same informative content as the function name and hence it is useless.

```
def calc_beta_from_amplitude(mad_twiss, list_of_files, plane):
    """ Calculates beta from amplitude """
    ...
```

Change logs should be avoided as well. They are artifacts and adds noise to the document. A professional Version Control System (VCS) should be used to store the source code. An additional task of the VCS is to save the history. It creates the change log automatically, and mostly much better than the developer. Thus long change logs should completely be omitted.

```
# Version-up history:
# V1.0, 11/Feb/2008 by Masa. Aiba
# V1.1, 18/Feb/2008:
# - Debugging, add model phase and tunes to output
# - add function to obtain DY
# - add chromatic parameter (phase for non zero DPP)
# V1.2, 22/Feb/2008:
# - test version for beta with all BPM
# V1.3, 29/Feb/2008:
# - beta from phases is improved, averaging beta1, 2 and 3
# V1.31, 12/Mar/2008:
```

```
12 # - debugged on alpha3
13 ...
```

Of course there are also legal comments. The *What* is described by the code itself and comments can explain the *Why* if it is not obvious.

```
1 # This part due to the format difference of phasef2 (from the
      model)
2 ...
```

```
1 datetime.datetime.today().strftime("%d. %B %Y, %H:%M:%S")
2 # e.g.: 17. July 2013, 12:28:56
```

B.1.4 The Stepdown Rule

Usually a file will be read from top to bottom. That is also the way, how source code should be structured. Martin calls it *the Stepdown Rule*[Mar09]. The source code file should be structured like a scientific paper or a newspaper article. It begins with an abstract which gives an overview of what is going on. This is the main function and should be around 5-30 SLOC. The further the reader proceed the more information will be revealed. The helper functions should appear in the same order as they are used in the previous function. The following example demonstrates it.

```
1 def main():
2     ...
3     func_a()
4     ...
5     func_b()
6     ..
7
```

```
 8 def func_a():
 9     ...
10     func_a1()
11     func_a2()
12
13 def func_a1():
14     ...
15
16 def func_a2():
17     ...
18
19 def func_b():
20     ...
```

B.1.5 The Boy Scout Rule

The compliance of the *Boy Scout Rule* helps to improve the software quality incrementally[Mar09]:

> *Leave the campground cleaner than you found it.*

This simple rule can be applied to software development. Checked-in code should be always be a little bit better than the checked-out code. This can be accomplished by renaming a bad identifier, extracting some code from a big function, insert docstrings and so on. The developer should not be afraid to break the code by such simple changes since the automated tests will ensure the correct behavior. *What is if there are no tests?*, is the wrong question. The question should be *why there are no automated tests?*

B.1.6 Red-Green-Refactor (TDD mantra)

The TDD mantra describes a process of developing software. There are three fundamental steps in this approach, see also figure B.2:

1. **Red** – Create small unit tests and ensure that they fail.[3]
2. **Green** – Write the minimal amount on production code to pass the tests.
3. **Refactor** – Improve the written code and assure that tests keep *green*.

It needs discipline but the result is improved software quality and the written automated test will be a great help in the future.

To get input and output data a software for high level numerical computations can be used, for instance Scilab[4]. Here is a script which implements equation B.1:

[3] The tests should fail to assure that they are not wrongly implemented and thus pass from the beginning.

[4] http://www.scilab.org/, visited on 1st Feb 2014.

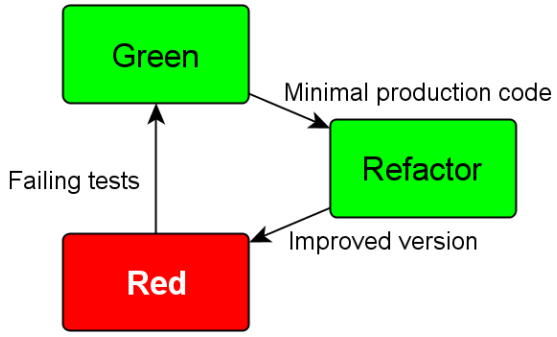

Figure B.2: The TDD mantra (Red-Green-Refactor)

```
num_points = 5
t_list = linspace(-%pi,%pi,num_points)
res = ones(num_points)

for i=1:num_points
    s = 0;
    t = t_list(i);
    for k = 1:99
        s = s + 4*sin((2*k-1)*t)/((2*k-1)*%pi);
    end
    res(i) = s;
end
```

The gained input and output is used to create a unit test.

```
def test_square_wave(self):
    assumed_results = np.array([-1.650777599717162E-14,
    -1.0032151693727007, 0.0, 1.0032151693727007,
    1.650777599717162E-14])

    num_of_osc = 99
    num_of_points = 5
    list_of_t = np.linspace(-np.pi, np.pi, num_of_points)

    calc_result = calculate_square_wave_for(list_of_t, num_of_osc
    )

    are_same_results = np.allclose(assumed_results, calc_result)
    self.assertTrue(are_same_results, "Calculated values are
    wrong")
```

A unit test serves not only as test. It is further also a usage example and represents documentation.

B.1.7 Clean Build

The aim should always be to produce software without any compiler warnings or errors. Further static analysis tools should be used to detect easily problems and bug patterns. The most warnings are reasonable but there are also some which are annoying. For instance the following code raises a warning because there are no white space around the power operator(**).

```
1 y = PI + 2**x
```

This makes no sense because the structuring of the numbers, identifiers and operators helps to read the equation easier.
On the other side all good static analysis tools are configurable to tailor the tool to the specific needs of the project.

B.1.8 Type Rich Programming

Rich type programming describes the usage of high level data types, classes and objects, instead of primitive data types, like floats and strings. A big problem in computations in physics is the correct usage of units.[5] Primitive data types have only a value but this value has to be interpreted. The following snippet[6]

```
1 void increaseSpeed(double speed);
```

The developer has to track the unit of the double value. There are not information about whether *speed* is in the unit meters per second or kilometers per hour. This can be avoided by using rich types.

```
1 void increase(Speed speed);
```

Speed itself knows which unit it has. This avoids a lot of failures.
Further classes can be used to combine multiple logically connected variables to a single one to keep the overview. Static analysis tools show warnings if a number of local variables is exceeded. Too many local variables exacerbate readability and thus should be avoided. This method can also be used to decrease the number of parameters in a function. Another possibility, which avoids creating classes for every case, is to use dictionaries. This is not a neat solution but much better than to save results in lists and return them.

```
1 def BPMfinder(IP,model,measured):
2     ...
3     try:
4         measured[0][bpml][0]
5     except:
```

[5] An example is the Mars Climate Orbiter, see http://en.wikipedia.org/wiki/Mars_Climate_Orbiter, visited on 1st Feb 2014.

[6] The C++ snippet from Bjarne Stroustrup was chosen to point out the data types. The variables in Python don't have a type. See http://ecn.channel9.msdn.com/events/GoingNative12/GN12Cpp11Style.pdf, visited on 31st Jan 2014.

```
6       ...
```

What is behind *measured[0][bpml][0]*? The developer knew it (at least a couple of hours after the implementation). The code does not show the intention. A reader has to find out what the data structure means to understand the logic. The cause is a complex return value in a function and a composition of local variables.

```
1 measured=[betax,betay]
2 # betax and betay are dictionaries(hash maps) with key/value
      pairs(k->v)
3 # k: the name of the BPM
4 # v: a list with three floats [a, b, c]
```

The developer has to track the knowledge about the content of the variables. A foreign reader, or even the developer himself, will have problems to decrypt subsequent code. Classes or dictionaries could be used to avoid it.

```
1 measured = {"beta_x": betax, "beta_y": betay}
2 ...
3 def BPMfinder(IP,model,measured):
4     ...
5     if bpml in measured["beta_x"]:
6     ...
```

B.2 Review

A software review describes the process of checking the source code(or another document) by a person. The reviewing person is not the author of the document. A review can be very formal with planned meetings and roles, such as moderator, author, reviewer, stakeholder etc., or informal without any structured process. The latter one can be adopted easily at CERN. If the code is hosted by GitHub or Jira and Fisheye, the integrated tools can be used to check the code or code changes and insert comments. Consequently the tools notify all relevant people, such as the author or watchers of the project.
It is very easy but therefor the produced code has to be published by committing and pushing the code. It should be avoided to hide code and releasing it first an a *perfect* state.
The result of reviews are less bugs since unbiased person checks the code. However, it is unlikely to find all bugs with reviews. A further advantage is the better quality. The reviewer could give advice how to improve the design or which spot of the code is in particular complex.By reviewing, knowledge will be shared automatically. So code will not be dependent from a single person.
Even if the reviewer is a much more experienced programmer, he should be pleasant with the comments. The feedback should be motivational but

not frustrating. The feedback shall also help to educate the author and to improve his knowledge.
On the other side the author has to take off his pride. The produced software is the precious of a software developer. Criticism could make the author angry but he has to act professional and accept the feedback to learn from it and become better.

B.3 Pair Programming

Pair programming describes the activity to develop a program in a pair. This includes one screen, one mouse and one keyboard but two developers which develop together. It is not pair programming if one programs and the other watches. Pair programming is a dialog between two people trying to simultaneously program (and analyze and design and test) and understand together how to program better.
Pair programming is also not a tutoring session. It will look like one in the beginning if one is much more experienced but very fast, after some weeks, the less experienced will give more and more input and get the keyboard more frequently. Basically it is a review all the time. While one is busy typing, the other is thinking at a more strategic and higher level. This avoids also code hiding and brings collective ownership. It should be used with newcomers to learn the work flow and practices in a team much faster. Pair programming encourages the face to face communication between team members and thus shares knowledge.
As a result it produces software with less bugs, a better design and higher quality.

B.4 Tools

After decades of developing software, there are a lot of tools which make the developers daily life much easier.

IDE
The main tool is an IDE. A developer should use and, very important, know the features of his IDE. Hot keys to edit text are especially useful. For the beginning a hot key *cheat sheet* can be used. A good IDE is configurable and can be adjusted to the need of the user. It needs some effort to learn the features of an IDE but the benefit is considerable. Further it supports *recognition*[7], which is useful if software will not be developed daily.

[7] It is easier to recognize information (GUI) than to recall it from memory without a cue (command-line interface).

Static Analysis Tools
A simple static analysis tool should always be already integrated in an IDE. However, these are only the basics of the static analysis. There are much more with mighty features, which look for bug patterns, memory leaks, code convention, and other bad quality smells. A developer should know the available static analysis tools for his domain.[8]

Automated Unit Tests
Every modern programming language has support for automated unit tests. These tests ensure intended behavior of the software. These tests can be run continuously on test servers with basically no costs. For refactoring and maintaining they are worth a mint.[9]

The work of a developer should be integrated as soon as possible, preferred daily, to avoid the integration hell[10] and code hiding. Thus problems can be detected earlier and the code is not hidden and can be reviewed easily. Server can build continuously the code and run the automated tests. Especially for builds and tests which takes minutes and more, this is very helpful and saves time.[11][12]

Version Control System (VCS)
A VCS is a mandatory tool if software is developed in a team. The tool has to be learned and it needs effort and practice but after mastering a tool it will produce almost no overhead. It is not only used to store the code, save different revisions and avoid overriding each others code. Further it can be used to backup and synchronize source code or any other files. LaTeX source files can be synchronized between home and office computer. If the hard disk, which stored the work of the last months, gets broken it is no problem since other repositories represent a backup.[13] Further it happens that changes cause tests to fail. It could take hours to hunt down the bug. In this case, the checkout of the last correct version and the rewriting of the changes could save the time which would be spent on debugging.

Wiki
A wiki is a good place to share knowledge. Especially newcomers will be very thankful to find a well maintained wiki with all information about the team.

[8] http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis, visited on 1st Feb 2014.

[9] http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks, visited on 1st Feb 2014.

[10] http://c2.com/xp/IntegrationHell.html, visited on 1st Feb 2014.

[11] http://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software, visited on 1st Feb 2014.

[12] CDash, http://www.cdash.org/, visited on 1st Feb 2014.

[13] This is true for distributed VCSs, in centralized VCSs only the server represents the backup.

A good wiki provide graphical text editors, so no additional knowledge is necessary. Further changes can be made immediately without risk because a revision control system behind a wiki is standard. CERN hosts a central Twiki.[14]

Issue Tracker
An issue tracker is a great enhancement which provides the possibility to make your work visible. An issue(bug, TODO, improvement, review etc.) can be discussed easier than an email conversation. Further the most issue tracker provides the ability to reference issues with code comments and vice versa. Issues get not lost and can be a good explanation and documentation for code changes. Everybody in the team can contribute to a discussion of an issue. An issue tracker shares knowledge and can be included easily in a development process. CERN uses the commercial software Jira which offers a lot of features.[15]

B.5 Software Development Process

A high-quality process leads to high-quality products. A software development process is the implementation of a model(Waterfall, Scrum, XP etc.). A structured development process is essential for software development teams to produce software with high quality. However, physicists primary task is not to develop software and it is mostly not the daily work but even a slightly structured process can increase enormously the quality.
It should be strictly avoided to start to code immediately. Time, to think about the design, the possible implementation, should be taken. The result could be pseudo code, UML diagrams or abstract descriptions and sketches. Then issues could be created and the design could be reviewed easily by another person. In this phase of the development the effort of a change is negligible compared to an already implemented solution. Figure B.3 shows a possible development process which can be easily adopted by everyone.

The first step is to open an issue and state the requirements(input/output) or planned changes, preferable with a reason. Based on the requirements automated tests can be written. Remember, it is much easier to write tests before implementation than after. The next step is to create an design, which can be reviewed easily by another person.
Only now comes the implementation phase, preferred the TDD approach. The implementation should be reviewed again to find bugs, share knowledge and improve the quality. After these activites the issue can be closed.

[14] https://twiki.cern.ch/twiki/bin/view/DefaultWeb/WebHome, visited on 1st Feb 2014.
[15] http://information-technology.web.cern.ch/services/JIRA-service, visited on 1st Feb 2014.

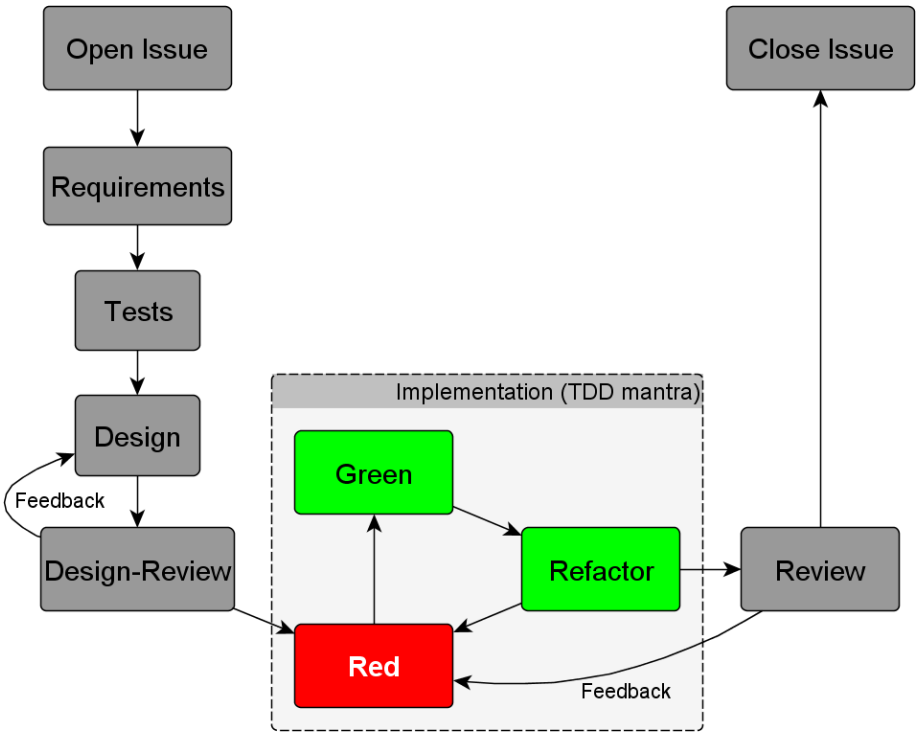

Figure B.3: An easy to adopt and tailored development process

It serves as documentation for the future.

B.6 Algorithm Improvement

Bad design in algorithms can lead to unnecessary calculations and horrible runtimes. There are several ways to improve the efficiency of an algorithm.

Numerical Libraries
The first step of implementing an algorithm should be to check existing numerical libraries. The library version is preferable to an own implementation. Famous libraries are well used and tested. The chance of an existing bug is nominal. Further the implementation behind uses probably the most efficient realization.[16]
Python is very popular in science because it is easy to use and has broad support. However, it is highly dynamic and offers no arrays. Thus it makes

[16] http://en.wikipedia.org/wiki/List_of_numerical_libraries, visited on 2nd Feb 2014.

it very slow. To minimize this disadvantage numerical libraries should be used for computations. NumPy[17] is a package for scientific computing with Python. It uses C bindings to boost the performance.
Python lists and dictionaries are slow and should be replaced by NumPy arrays. Further loops in Python should be avoided for calculations. Instead NumPy arrays and function should be used for calculations. Behind the scenes, there are again C bindings to increase the speed. Following is an naive implementation of equation B.1.

```
1 result = []
2
3 for t in list_of_t:
4     s = 0
5     for k in xrange(1, num_of_osc + 1):
6         s += (4*numpy.sin((2*k-1)*t)) / ((2*k-1)*numpy.pi)
7     result.append(s)
```

Slow data structures and Python loops are used above. Below is the improved version which uses NumPy arrays and functions.

```
1 summations = numpy.zeros(len(list_of_t))
2 k = numpy.arange(1, num_of_osc + 1)
3 two_k_minus_one = 2*k - 1
4
5 for i, t in enumerate(list_of_t):
6     summations[i] = numpy.sum((numpy.sin(two_k_minus_one*t)) /
      two_k_minus_one)
7
8 result = (4/numpy.pi) * summations
```

The ten thousandfold execution of both version gave the following times.

```
1 Start...
2 First version: 40.3803190072
3 Improved version: 1.60396869664
4 Finished...
```

The improved version of the algorithm runs 25 times faster.
The programming languages C and C++ are very efficient but they have to be mastered. The developer has to consider memory allocation, pointers, references data structures and more.
Python and efficient libraries are a good agreement between simplicity and efficiency.

Data Structures

Efficient and compact data structures lead to improved performance by avoiding cache misses[18]. As already mentioned NumPy arrays should be

[17] http://www.numpy.org/, visited on 2nd Feb 2014.
[18] A cache miss takes place if the processor has to wait for fetching data because it is not in the cache.

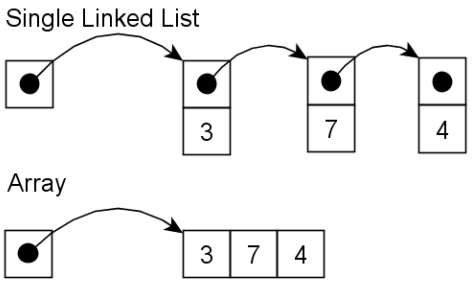

Figure B.4: The elements of a list are connected with pointers. The elements of an array are located in a coherent piece of memory

used since they are located together in the memory. Python lists are connected with pointers. Figure B.4 shows the difference.

Bottleneck

It is economically not possible to improve large algorithms completely. Hence the effort should be put into the bottlenecks of an algorithm. A bottleneck in an algorithm is the spot which consumes the most computation time and slows down the over all efficiency. They can be detected by profiler (performance analysis) tools[19]. These tools can measure the number of invocations of a function and their execution time. The built-in libraries *cProfile* and *profile* can be used for Python code.[20]

Mathematical Equations

Algorithms can also easily be improved by transforming mathematical equations with the goal to minimize the number of operations.
The scalars 4 and π from equation B.1 can be extracted:

$$f(t) = \sum_{k=1}^{\infty} \frac{4 \sin\left((2k-1)t\right)}{(2k-1)\pi} = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{\sin\left((2k-1)t\right)}{2k-1} \tag{B.2}$$

Given that n is the number of iterations(summations) used in the implementation, then one division was added but $2n-1$ multiplications were saved. Further equal expressions can be substituted. The equation above can be improved by substituting $2k-1$ with b. This will save n multiplications and n subtractions:

$$f(t) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{\sin\left(bt\right)}{b} \text{ where } b = 2k-1 \tag{B.3}$$

[19] en.wikipedia.org/wiki/List_of_performance_analysis_tools, visited on 2nd Feb 2014.
[20] http://docs.python.org/2/library/profile.html, visited on 2nd Feb 2014.

Before implementing an equation it should be checked whether it is possible to decrease the number of operations.

Parallelism
Before trying to make an algorithm parallel it should be ensured to have the best serial version of the algorithm.
Python – The Global Interpreter Lock (GIL) prevents the execution of parallel threads in Python. The *multiprocessing*[21] package can be used to create several processes. Threads exist within a process and share the same context. Processes have their own context and thus are more expensive to create.
Another possibility to use indirect parallel algorithms is to write the parallel algorithm in C or C++ and bind it to Python code.
OpenMP – Since it is easy to bind C and C++ code, bottlenecks can be implemented in C or C++ and bound to Python. The implicit threading library *OpenMP*(Open Multi-Processing) can be used to make algorithms parallel with prepocessor pragmas. See the C code snippet below:

```
1 #pragma omp parallel for
2 for (i = 0; i < N; i++) {
3  a[i]= 2 * i;
4 }
```

The forking and joining of threads is done by OpenMP and the developer does not have to explicitly take care of this. The following tutorial is a good start to learn OpenMP: https://computing.llnl.gov/tutorials/openMP/, visited on 2nd Feb 2014.
GPU – A further possibility to improve an algorithm is to swap the computations to the graphics card (graphics processing unit). This can be done by using the frameworks *CUDA*[22] or *OpenCL*[23]. Interfaces for Python exist also, *PyCUDA*[24] and *PyOpenCL*[25]

[21] http://docs.python.org/2/library/multiprocessing.html, visited on 2nd Feb 2014.
[22] https://developer.nvidia.com/what-cuda, visited on 2nd Feb 2013.
[23] http://www.khronos.org/opencl/, visited on 2nd Feb 2014.
[24] http://wiki.tiker.net/PyCuda, visited on 2nd Feb 2014.
[25] http://wiki.tiker.net/PyOpenCL, visited on 2nd Feb 2014.

Appendix C

Overview CD-ROM

The CD-ROM contains:

Beta-Beat.src – This directory contains a current copy of the OMC software base.

coding_lectures – The sources of the given lectures for the phycisists are inside this directoy.

conqat – This directory contains the workspace of the ConQAT realization. Additionally there is an example output of the HTML pages.

lhc-app-beta-beating – Here is the Eclipse Java project of the Beta-Beat-GUI.

omc_wiki – This is the git repository of the old OMC wiki.

surveys – Here one can find the surveys of the OMC team and the BE-ABP group.

thesis – This represents the source folder for the thesis.

vima0001_ba_thesis.pdf – The pdf file of the final thesis.

Glossary

AFS The Andrew File System is a network protocol for distributed network file systems.

API An application programming interface (API) is a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service..

BPM The BPMs (Beam Position Monitor) are the eyes of every accelerator. They identify the position of the beam inside the accelerator.

ConQAT Continuous Quality Assessment Toolkit (ConQAT) is a toolkit for rapid development and execution of software quality analyses, see https://www.cqse.eu/en/products/conqat/overview/.

ConQAT The Goal Question Metric (GQM) is a method to determine metrics. The base are (quality) goals which have to be achieved. The goals will be characterized with questions and metrics are associated with the questions.

GIL The Global Interpreter Lock (GIL) is a mechanism to prevent the parallel execution of threads in Python. Only One Thread can be executed at the same time.

HTML HyperText Markup Language (HTML) is a markup language to create websites and other information which can be displayed in a web browser..

IDE An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.

LSA The LHC Software Architecture (LSA) project provides homogeneous application software to operate the Super Proton Synchrotron (SPS) accelerator, its transfer lines, and the Large Hadron Collider (LHC). The system is entirely written in Java and it is using the Spring framework, an opensource lightweight container for Java platform, taking

advantage of dependency injection (DI), aspect oriented programming (AOP) and provided services like transactions or remote access.

MAD The Methodical Accelerator Design (MAD) is a software, created at CERN, to describe and simulate particle accelerators. It is used to obtain the model and the fullresponse to optimize the beam optics, see http://mad.web.cern.ch/mad/ visited on 20 Dec 2013..

PLOC Physical Lines Of Code correspond to the number of lines in the source file.

RMI Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines*, possibly on different hosts(http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, visited on 31st Jan 2014).

SDDS The SDDS (Self Describing Data Sets) ASCII file format is used to store input data for SVD clean and Drive. It consists of comment lines, beginning with #, and table data lines.

SLOC Source Lines Of Code are the number of PLOC without blank lines and comment lines.

TFS The TFS(Table File System) format is a text file which contains descriptor lines, comment lines and a table. Descriptor lines start with an *at*-character followed by the name, the type and the value. Comment lines are preceded with a '#'. The table consists of a column names line(*), a column datatypes line($) and data lines.

UML Unified Modeling Language (UML) is a standardized modeling language for the design of software applications in the field of software engineering.

VCS Version Control System (VCS) is a software that allows to manage changes of documents, programs, images and other information that is stored in form of computer files.

WYSIWYG Denotes the representation of text on-screen in a form exactly corresponding to its appearance on a printout, see http://www.oxforddictionaries.com/definition/english/WYSIWYG?q=WYSIWYG, visited on 21st Jan 2014. .

Bibliography

[And10] Anderson, D.J.: *Kanban*. Blue Hole Press, 2010, ISBN 9780984521401. http://books.google.de/books?id=RJ0VUkfUWZkC.

[BCK03] Bass, Len, Paul Clements, and Rick Kazman: *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, 2003, ISBN 0321154959. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321154959.

[BCR94] Basili, Victor R., Gianluigi Caldiera, and H. Dieter Rombach: *The goal question metric approach*. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[Bec99] Beck, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, first edition, 1999, ISBN 0201616416.

[Bec02] Beck: *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002, ISBN 0321146530.

[CKK01] Clements, Paul, Rick Kazman, and Mark Klein: *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001, ISBN 978-0-201-70482-2.

[con] *ConQAT User Guide*. https://www.cqse.eu/download/conqat/conqat-book-2013.10.pdf, visited on 17th Feb 2014.

[DeM09] DeMarco, Tom: *Software engineering: An idea whose time has come and gone?* IEEE Software, 26(4):96, 95, 2009, ISSN 0740-7459. http://www.computer.org/portal/web/csdl/doi/10.1109/MS.2009.101.

[EM07] Everett, G.D. and R. McLeod: *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley, 2007, ISBN 9780470146347. http://books.google.fr/books?id=z8UdPmvkBHEC.

[GD] Goede, Nils and Florian Deissenboeck: *Delta analysis*. Volume 32 of *Softwaretechnik-Trends*, May. http://pi.informatik.uni-siegen.de/stt/32_2/01_Fachgruppenberichte/SRE_TAV/27-goede.pdf, visited on 19th Feb 2014.

[Idr13] Idris, Ivan: *NumPy Beginner's Guide - Second Edition*. Packt Publishing, 2nd edition, 2013, ISBN 1782166084, 9781782166085.

[KQK11] Khan, Asif Irshad, Rizwan Jameel Qurashi, and Usman Ali Khan: *A comprehensive study of commonly practiced heavy and light weight software methodologies*. CoRR, abs/1111.3001, 2011. http://dblp.uni-trier.de/db/journals/corr/corr1111.html#abs-1111-3001.

[Lew05] Lewallen, Raymond: *Software development life cycle models*, July 2005. http://codebetter.com/raymondlewallen/2005/07/13/software-development-life-cycle-models/, visited on 15th Jan 2014.

[Mar09] Martin, Robert C.: *Clean Code: A handbook of agile software craftsmanship*. Prentice Hall, 2009.

[MSB12] Myers, Glenford J., Corey Sandler, and Tom Badgett: *The art of software testing*. John Wiley & Sons, Hoboken, N.J., 2012, ISBN 9781118031964 1118031962 9781118133132 1118133137 9781118133149 1118133145. http://www.amazon.de/Art-Software-Testing-Glenford-Myers/dp/1118031962/ref=sr_1_1?s=books-intl-de&ie=UTF8&qid=1330095168&sr=1-1.

[Pin08] Pinto, Pedro Nuno: *Software improvement process at cern coap*, 2008. http://repositorio-aberto.up.pt/bitstream/10216/58453/1/000129565.pdf, visited on 15th Jan 2014.

[Pol06] Pollice, Gary: *Writing clean code*, 2006. http://www.ibm.com/developerworks/rational/library/nov06/pollice/#authorN10017, visited on 21st Feb 2014.

[Roy70] Royce, Walker W.: *Managing the development of large software systems: concepts and techniques*. Proc. IEEE WESTCON, Los Angeles, pages 1–9, August 1970. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf, Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 19 7, pp. 328–338.

[Sco10] Scotland, Karl: *Aspects of kanban*, 2010. http://www.methodsandtools.com/archive/archive.php?id=104, visited on 17th Jan 2014.

[Sha11] Shalloway, Alan: *Demystifying kanban*, 2011. http://www.netobjectives.com/files/resources/articles/Demystifying-Kanban.pdf, visited on 17th Jan 2014.

[Wag13] Wagner, Stefan: *Software Product Quality Control*. Springer, 2013, ISBN 978-3-642-38570-4.

[Yan08] Yang, X.S.: *Introduction to Computational Mathematics*. World Scientific Pub., 2008, ISBN 9789812818171. http://books.google.de/books?id=T_i0o_55MNwC.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und wesentlichen Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche kenntlich gemacht habe.

CERN, am 25. Februar 2014

Viktor Maier