CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

CENTRE RÉGIONAL ASSOCIÉ DE MIDI-PYRÉNÉES

MÉMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGÉNIEUR CNAM

SPÉCIALITÉ : Informatique

OPTION : Architecture et Ingénierie des Systèmes et des Logiciels (AISL)

par

Tomio Tetzlaff

Approche collaborative pour la qualité du code logiciel dans un contexte Agile

A Collaborative Approach to Code Quality Within an Agile Context

Soutenu le 21 octobre 2016

JURY :

PRESIDENT :
Yann Pollet, Professeur des universités, CNAM Paris

MEMBRES :
Hadj Batatia, Maître de conférences-HDR, INP Toulouse
Thierry Millan, Maître de conférences, UPS Toulouse
Jérôme Derhi, Chef de projet, Berger-Levrault
Michel Soms, Chef de projet, Berger-Levrault

Acknowledgments

I would like to express my gratitude to my mentor and advisor MCF-HDR Hadj Batatia of IPST-CNAM for his years of support in my studies of computer science, especially for hs help in the research and execution of my engineering degree memoir. Besides his invaluable advice in the research and execution of my memoir, he has been the guiding force behind l'École d'Ingénieurs du Cnam in Toulouse which has given me the opportunity to evolve from a neophyte in computer science to a competent professional. In addition, I would like to thank the faculty and contributors of IPST-CNAM for their participation in this program. A special thanks to Bertrand Raiff for his wonderfully in depth courses on networking and the fundamentals of computer science.

A shout out to all my fellow classmates. We managed to support each other through many ups and downs, and worked together to create a constructive environment for experimentation and learning. I would also like to thank Cyrille Dolez for his incite and unique way of seeing and understanding the world. I profited immensely from our discussions and debates on computer science and forays into completely unrelated topics. Thanks for tolerating me through four group projects.

I would like to thank Berger-Levrault for employing me in a work study program for the last three years and especially for the support of my managers Michel Soms and Jérôme Derhi. They provided a fruitful and supportive environment where I was allowed to hone my skills and make mistakes. I would also like to express my gratitude to Bertrand Du-Paty for providing me the time and resources necessary for my memoir.

I must express my profound gratitude to my wife for her unfettered support and continuous encouragement during the last four years. I suspect that it has been more trying for her than for me, but she has always been encouraging and patient. Finally, I must thank my children for reminding me of the importance of play.

Glossary

Agile Agile software development uses iterative development as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes fundamentally incorporate iteration and the continuous feedback that it provides to successively refine and deliver a software system [151]. 6, 11, 12, 16, 17, 18, 23, 56, 67, 77, 97

code-and-fix Development is not done through a deliberate strategy or methodology. It is often the result of schedule pressure on the software development team. Without much of a design in the way, programmers immediately begin producing code. At some point, testing begins (often late in the development cycle), and the unavoidable bugs must then be fixed - or at least, the most important ones must be fixed - before the product can be shipped [151]. 10, 12, 97

didactic method In didactic method of teaching, the teacher gives instructions to the students and the students are mostly passive listeners. It is a teacher-centered method of teaching and is content oriented. The content or knowledge of the teacher is not questioned [143]. 58

Experiential Learning Experiential learning is the process of learning through experience, and is more specifically defined as learning through reflection on doing [144]. 58, 65

Maven Maven is a build automation tool used primarily for Java projects. The word maven means“accumulator of knowledge” in Yiddish. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins [141]. 82, 84, 85, 88

n-tiers Multitier architecture (often referred to as n-tier architecture) is a client server architecture in which presentation, application processing, and data management functions are physically separated. The most widespread use of multitier architecture is the three-tier architecture [147]. 5, 85

OSGi The OSGi specification describes a modular system and a service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments. Applications or components, coming in the form of bundles for deployment, can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot; management of Java packages/classes is specified in great detail. Application life cycle management is implemented via

APIs that allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly [148]. 81, 85, 86, 87, 88

REpresentational State Transfer (REST) HTTP-based RESTful APIs are defined with the following aspects: base URL such as http://example.com/resources/, an Internet media type type that defines state transition data elements (e.g., Atom, microformats, application/vnd.collection+json) The current representation tells the client how to compose all transitions to the next application state, and the use of standard HTTP methods (OPTIONS, GET, PUT, POST, DELETE, etc) [149]. 82, 85

Scrum Scrum is an iterative and incremental agile software development framework for managing product development [150]. 6, 16, 18, 19, 20, 21, 22, 63, 64, 65, 67, 68, 69, 70, 73, 74, 76, 78, 92, 93

Spiral model In 1988, Barry Boehm published a formal software system development spiral model, which combines some key aspect of the waterfall model and rapid prototyping methodologies, in an effort to combine advantages of top-down and bottom-up concepts. It provided emphasis in a key area many felt had been neglected by other methodologies: deliberate iterative risk analysis, particularly suited to large-scale complex systems [151]. 14, 15

technical debt Code can be carefully designed to maximize the internal qualities of code or it can be hacked together in order to quickly deliver new functionality. Fast and dirty may seem efficient at first but may lead to future delays because it is not very maintainable. This is technical debt. 7, 63, 64, 78, 82, 94

timeboxing In time management, timeboxing allocates a fixed time period, called a time box, to each planned activity [153]. 16

V-model The V-model represents a development process that may be considered an extension of the waterfall model, and is an example of the more general V-model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively [154]. 6, 12, 14

Waterfall The waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through several phases [151]. 12, 13, 14, 15

Acronyms

A Abstraction. 44, 45, 46

ADP Acyclic Dependencies Principle. 40, 41, 117

AJAX Asynchronous JavaScript and XML. 6

API Application Program Interface. 50

CA Afferent Coupling. 41

CBO Coupling Between Object classes. 39, 47

CC Cyclomatic Complexity. 32, 33, 34, 37

CCP Common Closure Principle. 43, 44

CE Efferent Coupling. 41, 42

CI Continuous Integration. 75, 81, 82, 83, 84

CISQ Consortium for IT Software Quality. 26

CK Chidamber and Kemerer. 39, 43, 46, 47

CNAM Conservatoire national des arts et métiers. 5

COBOL Common Business-Oriented Language. 11

COCOMO Constructive Cost Model. 30

CRP Common Reuse Principle. 43, 44

CSS Cascading Style Sheets. 6

DAO Data Access Object. 6, 86

DIP Dependency Inversion Principle. 49, 50, 85

DIT Depth of Inheritance Tree. 47, 48

DLL Dynamic Link Library. 40

DMS Distance from the Main Sequence. 44, 45, 46, 49

DRY Don’t Repeat Yourself. 31, 69

DSDM Dynamic Systems Development Method. 16

EI Ecole d’ingénieurs. 5

ELT Experiential Learning Theory. 58

FAA U.S. Department of Transportation Federal Aviation Administration. 36

FDD Feature Driven Development. 16

FORTRAN Formula Translation. 11

GUI Graphical User Interface. 6, 86, 87

GWT Google Web Toolkit. 6

HTML Hyper Text Markup Language. 6

HTTP Hypertext Transfer Protocol. 6, 83, 84, 85, 86, 88

I Instability. 44, 45, 46

IDE Integrated Development Environment. 51, 75, 81, 85, 86, 91

IID Iterative and Incremental Development. 12, 16

IO Input Output. 87

IPST L'Institut de la promotion supérieure du travail. 5

ISO International Organization for Standardization. 26, 27, 28, 52

ISP Interface Segregation Principle. 49

JAR Java Archive. 40, 85

JDK Java Development Kit. 87

JPA Java Persistence API. 85, 86

JSON JavaScript Object Notation. 85, 86

KLOC Thousand Lines of Code. 29

LCOM Lack Of Cohesion of Methods. 43, 47, 49

LLOC Logical Lines of Code. 29, 30

LOC Physical Lines of Code. 29, 30, 32, 33, 35, 46, 54

LSP Liskov Substitution Principle. 49

MVP Model View Presenter. 87

NIM Number of Inherited Methods. 48

NM Number of Methods. 48

NMO Number of Methods Overridden. 48

NOC Number of Children. 47

OCP	Open Closed Principle. 40, 49
OMG	Object Management Group. 26
OO	Object Oriented. 11, 25, 26, 38, 39, 42, 43, 44, 46, 48, 49, 64, 85
ORM	Object-Relational Mapping. 6, 85
OS	Operating System. 9
PDSA	Plan-Do-Study-Act. 12
PO	Product Owner. 21, 93
POJO	Plain Old Java Object. 86, 87
RAD	Rapid Application Development. 16
REP	Reuse/Release Equivalence Principle. 43, 44
RFC	Response For a Class. 47
RH	Ressources Humaines. 5, 6
SAP	Stable Abstractions Principle. 40, 45, 46
SDK	Software Development Kit. 87
SDP	Stable Dependencies Principle. 40, 41, 45, 117
SIX	Specialization Index. 48
SLOC	Source Lines of Code. 29, 30, 34
SM	Scrum Master. 21, 74, 75, 78
SQALE	Software Quality Assessment based on Life Cycle Expectations. 52, 54, 55, 82
SQuaRE	Software product Quality Requirements and Evaluation. 28, 52
SRP	Single Responsibility Principle. 44, 49
SWT	Standard Widget Toolkit. 88
TDD	Test Driven Development. 18, 37
UP	Unified Process. 16
URI	Uniform Resource Identifier. 84
WADL	Web Application Description Language. 85
WMC	Weighted Methods per Class. 47
XP	Extreme Programming. 16, 18, 19, 65, 70, 73

Contents

1 Introduction

Software engineering has evolved over its history from an exclusive club of experts who built and maintained complex and expensive computers and wrote software for specific machines, to a widespread and ubiquitous activity. As machines have become more powerful, complex and less expensive, software has become an essential element in almost every corner of modern life and has overtaken hardware as the primary expense. The early days of ad-hoc software development methods led to the "software crisis" where many projects were late, over budget and did not work as expected. The need for rigorous development methodologies grew with the complexity of the software being written. These methodologies evolved over time and fall into three main categories – plan driven, iterative and agile.

Plan driven development methodologies were modeled after the techniques used in traditional engineering. These methodologies were based on the axiom that problems are completely specifiable and are centered around refining processes to maximize quality. As they mirrored traditional and proven engineering concepts, these plan-based methodologies were widely adopted. Nevertheless, problems discovered late in the process are difficult and expensive to fix which introduces an element of risk and uncertainty. Iterative methodologies were being developed in the same era, but were not as mainstream. This approach allowed software engineers to begin with a high-level design and incrementally build upon previous versions. This attenuated some of the shortcomings of plan driven methodologies. Agile methods were developed in response to rapidly changing requirements and the inherently unpredictable nature of software development. These methods are designed to respond to unforeseeable circumstances and modifications by the client. Scrum is an Agile approach that focuses on project management and organization. Self-organizing teams work together to deliver value to the client through time-boxed increments. Scrum teams rely on the expertise of the developers, who together, determine the definition of "done." Not all Scrum teams are equal and this communal definition normally includes aspects of external quality, but may or may not consider the internal qualities of software.

Software quality, both internal and external, was one of the driving factors in the development of software engineering methodologies. The scientific literature around software quality is extensive and much work has been done to objectively measure quality. Various software metrics have been developed to measure the overall quality of code and to highlight areas of the code that may require improvement. Nevertheless, these metrics are incomplete and have limitations. Coding principles, patterns and standards have thus been designed to guide developers in writing code that maximizes quality characteristics.

One of the challenges faced by Scrum teams is assuring that valuable and high quality functionality is delivered and that this code contains the characteristics of internal quality. External quality can be observed by testers and by the client; however, the internal qualities of code are not easily ascertainable. The negative effects of quality can be experienced through increasingly expensive maintenance operations and difficulties in modifying, reusing and evolving code. Ignoring internal quality leads to an increase in "technical debt" and slows development and maintenance.

This document explores a strategy for ensuring high code quality for software developed within the Scrum framework. This research was inspired by observing the implementation of the Scrum methodology in a software development company. This company recently migrated from V-model development to a Scrum methodology. The overall ambiance, implication of developers and team dynamics have improved. Even though many aspects of software quality are examined at this company, the internal qualities of code are often overlooked. This lack of code quality considerations in the development cycle instigated research on the subject.

By collaboratively integrating internal software quality considerations into their development cycle and assuming shared ownership of this quality, an Agile team can improve the quality of their code and reduce "technical debt." In the context of an Agile development methodology, code quality must be handled in a collaborative manner. Since Agile methodologies favor self-organizing teams over top down control by management, the only way of ensuring code quality is by making it an essential component of the team's production and the object of shared concern.

In this document the context and background of software development and code quality evaluation are surveyed along with various approaches to knowledge creation and learning. A modification to the Scrum framework is proposed to take into consideration the internal qualities of software throughout the development cycle. Various events and a new role are introduced to raise developer awareness of the importance of code quality and techniques are proposed to help developers internalize and share new ideas and learning. Code quality becomes an integral part of the development process and communal ownership of code and its quality is an essential aspect of this method. A support mechanism has been developed to assist Scrum teams in monitoring their contributions to quality providing increased awareness of quality issues and motivation for improving them. Details of the different components of the support system and their interactions will also be examined. A portal to oversee the evolution of code quality is also provided for the team and its managers. A trial implementation of this method will be exposed along with a proposal to apply this method to a real world enterprise application.

I

Context

Corporate Context

2

2.1. Le Groupe Berger-Levrault

Berger-Levrault is the twelfth largest software publisher in France specializing in administrative applications, forms and documents for the public sector such as hospitals, schools and city halls. This company develops professional software and regulatory content, is a service provider and acts as a hosting service provider. In 2015 Berger-Levrault had 50,000 clients, 1,400 employees and a revenue of €126 million [19].

I have been working as a software developer at Berger-Levrault since 2013 in a work study program in conjunction with IPST-CNAM. I am currently in the third year of EICNAM preparing an Engineering degree. I worked for two years on an accounting software for small to medium sized city halls called EGF-Evolution. I was recently transferred to a new project, a web based application for Human Resource management – e.sedit RH. During my time at Berger-Levrault my principal missions have been: the creation of a software factory for EGF-Evolution, developing new functionalities and maintaining EGF-Evolution and e.sedit RH.

2.2. e.sedit RH

Web Application

E.sedit RH is a dynamic web based application for managing the human resources of collectivities. It is made up of several modules based on the business needs of human resource management (Figure 2.1).

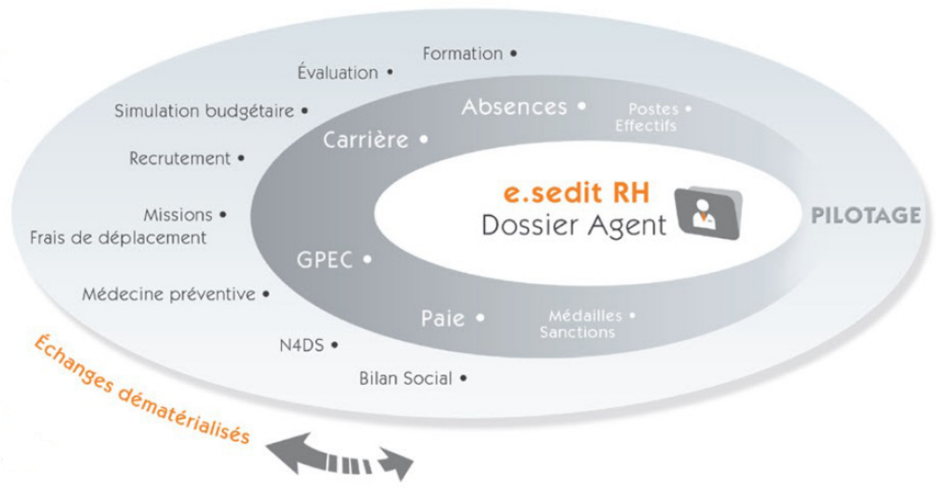


Figure 2.1: e.sedit RH [20]

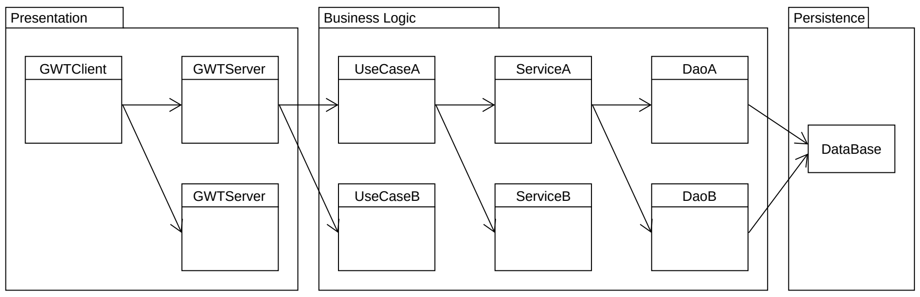


Figure 2.2: e.sedit RH Architecture

Architecture

E.sedit RH implements an N-tiers architecture with three principal tiers - Presentation, Business logic and Persistence, which are organized in layers (Figure 2.2).

The presentation tier is developed with an in house framework called WebCore2, based on the Google Web Toolkit (GWT). This framework unifies the look and feel of the application and handles access control. GWT is a framework developed by Google which provides an abstraction of web technologies such as Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript allowing developers to create Asynchronous JavaScript and XML (AJAX) based application without intimate knowledge of these technologies. With GWT the developers code in a manner similar to Java Swing with Graphical User Interface (GUI) widgets and layouts based on panels and listeners to handle events. The presentation tier is divided into two layers, client and server. The client classes are compiled by the framework into JavaScript which is subsequently interpreted by web browsers. GWT creates a different JavaScript file for each browser family in order to eliminate various browser quirks. The server layer implements Java servlets and provides an interface for the developer to handle Hypertext Transfer Protocol (HTTP) requests [63].

The business logic tier is independent of the technology used by the presentation tier. It is divided into three layers - UseCase, Service and Data Access Object (DAO). Each layer uses classes of the successive layer but does not use classes in a higher layer or within the same layer. The use case layer receives calls from the GWTServers and acts as a controller. It coordinates different services of the service layer which uses the DAO layer to retrieve and persist data. This layer interfaces with the persistence tier and uses the Object-Relational Mapping (ORM) provided by Hibernate.

The architecture of e.sedit RH was defined early in the development process and has not changed significantly over the years. Nevertheless, as new features are added and corrections implemented, the architecture has potentially eroded and become more complex through non-respect of conventions and norms, design decisions and as a result of entropy [37].

2.3. Development Methods

The development methods used at Berger-Levrault have gone through a recent evolution. When I began most projects were developed using the V-model development process, but recently Agile methods have been introduced, namely Scrum. Agile development methodologies were brought into the company in order to improve the quality of the delivered product through a continuous improvement process, enhance productivity by promoting teamwork and better manage production deadlines through shorter development and feedback cycles.

The architectural framework of e.sedit RH is well defined, follows a classic form and the coding standards are well defined. One of the difficulties encountered in this project is a lack of respect for these standards and an absence of design methods and archetypes. A plethora of developers have worked on this application over the years resulting in heterogeneous design decisions and implementations adversely effecting the overall architecture and quality of the application.

Agile methods emphasize the capabilities of the team and a continuous analysis and refinement of design [6]. E.sedit RH is an application that is deployed in hundreds of collectivities throughout France and several versions are released yearly. New functionalities and regulatory updates are continuously being developed under relatively tight deadlines. This mode of development incites developers to implement new features quickly with a focus on creating usable components; however, consideration for the internal qualities of code are often neglected, leading to maintenance and usability problems.

The developers of e.sedit RH are responsible for the design and implementation of each new feature and evolution. This freedom is gratifying for the developers and each developer has significant autonomy in his work, but this freedom requires a fairly high level of skill, competence and rigor. One of the difficulties encountered is the varying level of proficiency and experience in design and development. This heterogeneity is enriching for the team providing varying points of view and different ways of approaching the same problem. On the downside there is a lack of coherence between different modules and classes resulting in an incongruous code base.

The internal qualities of the code have degraded over time making changes and additions to the application time consuming, risky and difficult; a general lack of unit tests further increases the risk of regression. These factors have led to an erosion of the original architecture and a deterioration in quality. Working in this context of continually increasing technical debt, long maintenance cycles and difficulties in integrating new functionalities into the existing application have highlighted a need for introducing code quality management into the development process.

An Overview of Software Engineering

3

3.1. A Brief History

The history of software development is intricately tied to advances in computer hardware and the management of quality. Hardware speeds and capabilities advance rapidly and the expense of hardware continues to decline; however software development long remained a craft industry and has become the primary expense in software development. The first electronic computers were large and cumbersome machines. Operating these machines required extensive knowledge of the physical machine and expertise to keep them in a working state.

For fantastic pieces of equipment they were: in retrospect one can only wonder that those first machines worked at all... — Dijkstra 1972 [47]

Programming these machines was done by the scientists who operated them and often required physical modifications to the machine. Each of these machines was unique and programs written for them were specific to a certain computer [47]. Early programmers generally wrote simple programs to accomplish a task that they understood well, such as finding a solution for a mathematical problem [61].

Hardware technology advanced rapidly and was based on traditional engineering techniques; however, the approach to writing programs for these machines was ad hoc. As computers became more powerful, the job of "programmer" came into its own and clients could thereafter hire someone to create a computer program that responded to their particular needs [61]. The electrical engineers who developed hardware had relatively stable and well defined specifications and the nature of the development process required significant upfront planning and dealt with real world, physical constraints. On the other hand, the mathematicians and logicians who developed programs often had ambiguous specifications, changing requirements and a less industrialized development process. Programming remained a craft industry.

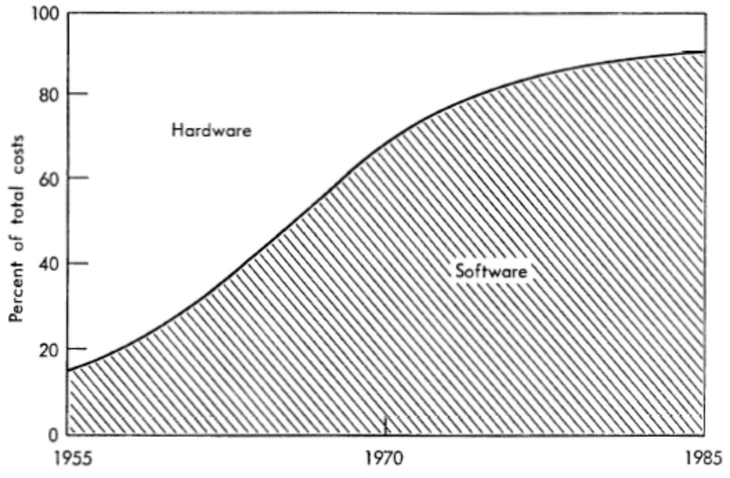


Figure 3.1: Hardware/Software cost trends [22]

As computers evolved and became more complex and capable, their users required more intricate programs with a scope that was too large for single programmer to accomplish. In the early 1960's IBM built the System/360 and the first sophisticated operating system, the OS/360. This project was a huge undertaking at the time and was completed years late and millions of dollars over budget [109]. The largest expense on this project was the operating system, not the hardware. This trend continues and today the cost of developing software greatly outweighs the cost of the hardware needed to run it (Figure 3.1). The experience of IBM and several late and over budget US Air Force projects led to what was called the "software crisis." Some of the software problems cited were [76]:

- *Unreliable software*
- *Substantial delivery delays*
- *Prohibitive in terms of modification costs*
- *Impossible to maintain*
- *Performing at an inadequate level*
- *Exceeding budget costs*

In the early days of computer programming, the principal development model was "code-and-fix" [23]. This process consisted of writing some code and then fixing any errors found. Testing, requirements and design were put off for later in the development process. This model was adequate for simple problems but as software grew more complex, the "code-and-fix" model led to complex and unmaintainable architectures and poorly designed code, resulting in problems cited above. Software development suffered from a lack of quality and insufficient production performance. Experiences in developing complex software, illustrating a general lack of control over the budget and effort required to build software, was at the origin of a new discipline – software engineering. The first references to the term "software engineering" date to 1966 [97] but is more commonly attributed to the 1968 Garmisch-Partenkirchen conference [101].

As computers evolved and became more complex and capable, their users began writing more intricate programs with a scope that was too large for single programmer to accomplish. According to Ghezzi in the book *Fundamentals of Software Engineering* [61]:

> *A programmer writes a complete program, while a software engineer writes a software component that will be combined with components written by other software engineers to build a system.*

This highlights the difference in complexity of these software systems compared to earlier programs as well as the teamwork and cooperation that was thenceforth essential to build these systems. The development team and the software engineering process has become the epicenter of software development.

As this was a new and burgeoning field, early software engineers applied well understood traditional engineering practices to software development. In order to better manage software projects a traditional engineering approach was adopted based on “structured techniques and formal design methodologies” [109]. This including project management, the use of tools, methodologies and techniques [61]. This course of action was meant to eliminate many of the problems associated with programming as a cottage industry and integrated the practice of writing code into the larger scope of a software product’s life cycle. The first development methodologies were planned, well-documented and sequential in nature and are referred to as plan-driven processes.

Software engineering has tried to apply a traditional engineering approach to software; however, the nature of software is different than traditionally manufactured products. Software is intangible and the laws of physics do not apply [29] and software cannot be perceived directly by the five human senses [30]; therefore, different models and abstract representations are necessary to visualize and conceptualize it. This abstraction increases the cognitive complexity of developing complex software systems. Software production is also labor-intensive and difficult to automate, but once software has been created, it is easy and inexpensive to reproduce. Manufacturing software can be reduced to the compilation and packaging which is largely automated. The main cost of software production is in the development, whereas in traditional engineering the principle cost is in manufacturing [82]. The traditional top-down, engineering approach has its place in software engineering, but has proved inefficient for software with ambiguous and frequently changing requirements. Certain lightweight and iterative methodologies have been proposed to overcome some of the shortcomings of traditional processes; nevertheless, the results have not revolutionized the industry.

Brooks, one of the leaders of the IBM OS/360 project noted in *No Silver Bullet* [30] that the industrialization of computer programming was not enough to solve the all the problems faced by the industry. Brooks categorized the difficulties in software development as “accidental” and “essential.” Accidental difficulties are related to problems with tools and technologies which can be fairly easily resolved, while essential difficulties concern the inherent complexity of the design of large software systems and the ambiguity of their requirements. According to Brooks, “no silver bullet” exists to magically “improve software productivity, reliability or simplicity” [30] due to the intrinsically complex nature of software.

Certain technological improvements, such as third-generation programming languages like FORTRAN or COBOL reduced the cost of programming by up to one third [76], helping to reduce “accidental” difficulties and more recent advances such as Object Oriented (OO) languages helped reduce the complexity of programming [57]. Iterative methodologies were developed to take into account ambiguous and changing requirements, and Agile, lightweight, methods have gone even further by rapidly turning “ideas into running tested code” [57] thus making some of the “essential” difficulties more manageable.

Various methodologies have been developed to help improve the efficiency of software development and no one-size fits all solution exists. Each software project has unique requirements and particularities, therefore an appropriate method of development must be chosen for each project.

	Agile	Plan-driven
Developers	Agile; knowledgeable; collaborative	Plan-oriented; adequate skills; access to external knowledge
Requirements	Emergent, rapidly changing	Largely stable
Architecture	Designed for current requirements	Designed for current and future requirements
Refactoring	Inexpensive	Expensive
Size	Smaller teams and products	Larger teams and products
Primary Objective	Rapid value	High assurance

Table 3.1: Agile and plan-driven methods [21]

3.2. Development Methodologies

Traditional development methodologies have long dominated the field of software engineering but lightweight Agile methodologies have become increasingly popular because they respond to the dynamic environment of modern business. Both have strengths and weaknesses and are each appropriate for different types of projects (Table 3.1). Hybrid approaches have since been developed for projects with a combination of characteristics that do not fall clearly into one category or another [21].

Shortly after the inadequacies of the "code-and-fix" method came to light, traditional engineering methodologies took the forefront in software development. Traditional plan-based development methodologies such as the Waterfall or V-model were created. They are sequential design methodologies based on the engineering paradigm that problems are completely specifiable and are centered around refining processes to maximize quality [102]. As they mirrored traditional and proven engineering concepts, these plan-based methodologies were widely adopted by the budding software engineering industry.

In the meantime, various individuals and companies were using Iterative and Incremental Development (IID) methodologies as early as 1957 [15]. The idea for iterative development dates at least back to the 1930's with the work of Walter Shewhart who described the cyclical development process Plan-Do-Study-Act (PDSA) in order to improve quality. In the 1940's PDSA was promoted by Edward Deming, a student of Walter Shewhart, as a focus on continuous improvement [71]. Originally this method was applied to the production of physical products and was used in software development in the 1980's [15]. Boehm's spiral model is another example of an iterative development process. Brooks, in *No Silver Bullet*, also promoted IID methods as a better alternative to the sequential methods traditionally applied to software development [30].

Later, Agile methodologies were introduced to better manage the unpredictability of software development in complex environments. Whereas traditional methods consider development as a "fully-defined" process, Agile methodologies take an em-

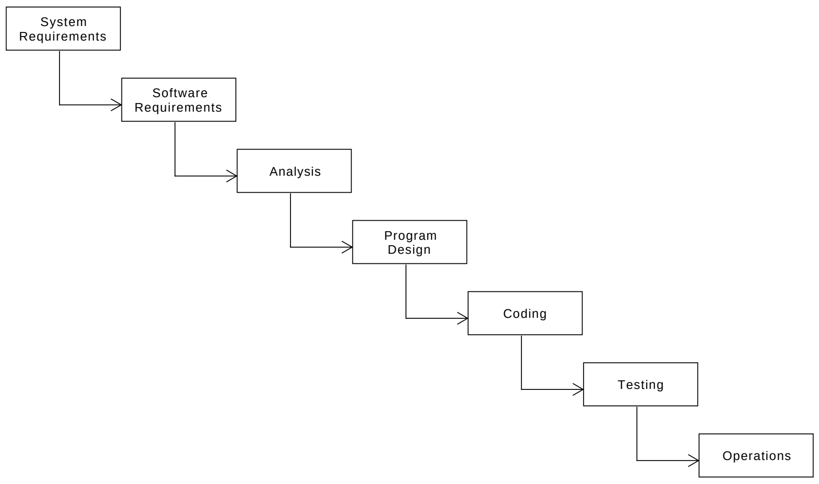


Figure 3.2: Waterfall Model [115]

pirical approach. Compared to industrial processes, much of software development is not “fully-defined” and treating it as such can lead to unpredictable results [117]. The inherent abstraction that is required in software development leads to processes that cannot be completely controlled upfront, therefore, Agile methods treat software development as an undefined process and take a “black-box” or empirical approach to counter act this unpredictability. Agile proponents claim to increase the probability of a successful project by continually adapting to change in a controlled manner by focusing on communication and the capacities of the development team [76].

Plan-based Methodologies

Waterfall

The Waterfall model was closely modeled after the engineering and manufacture of physical products. One of the first descriptions of this model was in 1970 by Winston Royce (Figure 3.2). In this model, specialists are attributed to the different phases of the project and often several completely different teams will work on the project in a sequential manner. Royce was a proponent of detailed documentation and considers it an essential component to the success of the method. For him, the documentation provides the glue between development phases and a phase is not considered finished until all the required documentation has been finished. Written documentation requires the different teams to “take an unequivocal position and provide tangible evidence of completion” [115]. This helps eliminate ambiguities and provides a mechanism for downstream and external actors to completely understand what came before and allows managers to have a comprehensive understanding of the project.

Many variations of the Waterfall model exist but follow the same basic framework. The beginning phases determine what the project will do, the secondary phases show how it will be designed, the software is coded in the subsequent phases and the final phases test the entire system [90]. This model is most appropriate for projects with

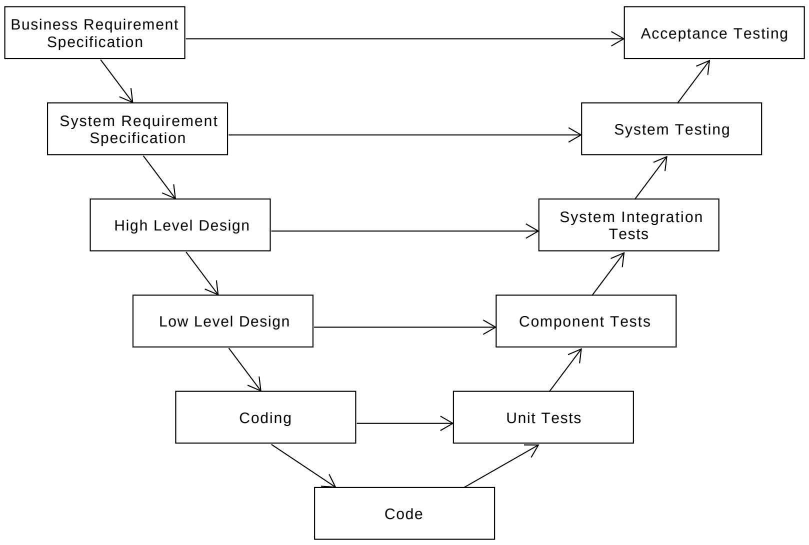


Figure 3.3: V Model [73]

clear requirements and objectives that are unlikely to change over the life of the project, such as compilers or security software [23]. This model has the advantage of maximizing the expertise of team members during each phase of the project thus conserving resources for other projects. For example, once the design phase is finished the architects and those with this valuable skill set can move on to other projects. On the flip-side this model responds poorly to changing requirements, the emphasis on written documentation can be costly to produce and maintain and the strict application of processes reduces flexibility and adaptability. The dependency on written documentation can lead to a loss of certain intricacies of the system that may not have been written down.

V-Model

The V-model was first proposed by Paul Rook [94] in the 1980's and follows a similar approach to the Waterfall model and can be considered a variant. In the V-model different test phases are defined in parallel with the development phases. For example, acceptance tests are written together with the business requirement specification and integration tests are defined at the same time as the high-level design is defined (Figure 3.3). Testing is a resource intensive activity and in the Waterfall model, this phase is executed as the last step of the process. Any architectural or design defects that may be discovered are expensive to fix because of the linear nature of the development cycle. The V-model attempts to overcome this aspect of the waterfall method by defining tests early in the process. By doing this, defects are often found earlier in the development process thus improving the changes of success [94]. Otherwise, this model has similar advantages and disadvantages to the Waterfall model.

Traditional plan driven development methodologies are most appropriate for projects where requirements are stable and well understood at the beginning of the

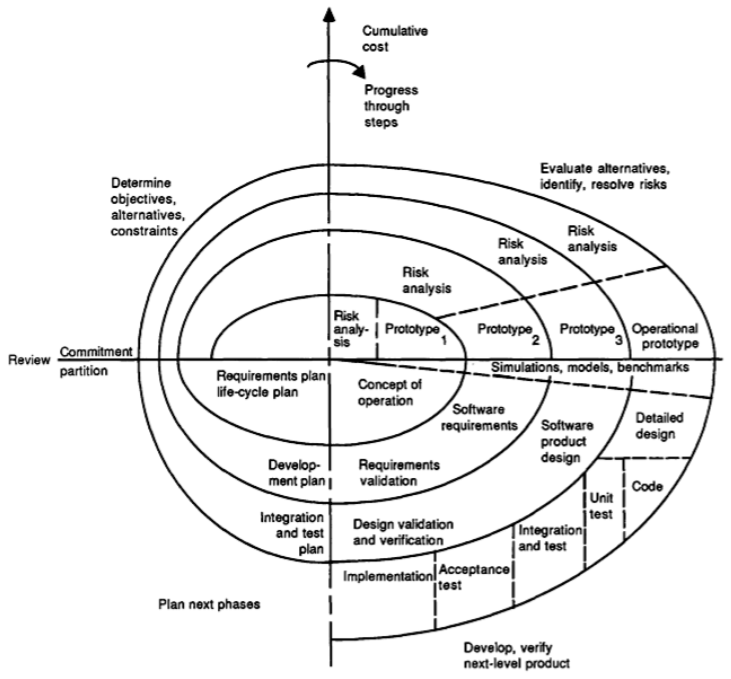


Figure 3.4: Spiral Model [23]

project; however, the sequential nature of the development leads to high risk and uncertainty. Several cyclical and prototype driven models have been developed to help overcome these insufficiencies. In the 1980's Boehm developed the Spiral model with the goal of driving the project through risk management [23].

Iterative Methodologies

Spiral Model

The Spiral model is risk based model that is made up of several cycles. Each cycle follows the same basic steps [33]:

1. Determine objectives and constraints
2. Evaluate alternatives, identify and resolve risks
3. Develop and verify deliverables
4. Plan the next iteration

The work to be done in each cycle is determined by risk; the highest risk elements of the system are treated first. The risk is evaluated by various methods such as prototyping, benchmarking, analytic modeling or other risk management strategies [23] and alternatives are proposed. High risk items are handled early in the process and lower risk elements are reserved for later development cycles, maximizing the probability successfully completing the project. In the Spiral model, the first few cycles are used to determine system objectives, run risk analyses and construct a plan for handling the system's life cycle which includes development and maintenance. Subsequent cycles are dedicated to the production and testing of the system. The cycles allow for the development team to evaluate the state of the project at regular intervals helping to alleviate some of the downfalls of the Waterfall model. Nevertheless, the Spiral model still follows the same sequential approach as the Waterfall model. Each phase, requirements, design, etcetera, is executed in sequence [117].

One of the advantages of this model is that within each cycle, different development methodologies can be applied, allowing reuse of existing expertise. Boehm claims in his original paper that projects that applied this model increased their productivity by at least 50% [23]. Nevertheless, this model requires competent developers who are able to accurately evaluate and deal with risk effectively.

Agile Methodologies

Due to the unpredictability of software development and to overcome the shortcomings of traditional methods, the 1990's saw an explosion of IID methodologies and iterative development was seen by many as a not only a viable alternative to traditional methodologies, but an improvement, yet they were still viewed by traditionalists as simple hacking. Many of these methods began to be formalized and this decade saw the birth of Scrum, Extreme Programming (XP), Rapid Application Development (RAD), Unified Process (UP), Dynamic Systems Development Method (DSDM) and Feature Driven Development (FDD) amongst others [15]. These new approaches to software development had certain commonalities such as timeboxing, incremental and adaptive development and were people oriented [1]. Two of the main differences between traditional engineering methodologies and Agile methodologies is that "Agile methods are adaptive and not predictive", and "Agile methods are people oriented rather than process oriented" [56].

Because Agile methods are designed around change, development teams are constantly reanalyzing and reworking the project to accommodate the dynamic nature of software development. Requirements in Agile methods are seen as emergent rather than preordained [21] and must be integrated as they are discovered. A single team handles all aspects of each iteration: analysis, design, implementation and testing. Agile frameworks make integrating changes more feasible and, because of the short feedback loops, less expensive. This contrasts traditional engineering methods which focus on a separation of concerns – certain teams/developers handle the beginning phases of requirement engineering, design and architecture and other less skilled programmers execute and test these plans. In traditional development all aspects of the software are determined upfront, changes downstream can be very difficult and expensive to handle, but Agile methodologies are designed to handle malleable requirements.

Traditional methods also focus on repeatable quality processes to ensure a quality product regardless of the individual experience or expertise of the team applying them. Agilists, on the other hand, rely on a competent and skilled development team to ensure quality. For this to function, Agile methodologies are designed to support the development team [56]. The emphasis on highly skilled developers marks a break from the traditional separation of engineers and workmen and is an essential aspect of Agile methodologies.

In 2001 seventeen representatives of these new development paradigms got together to discuss their similarities and to homogenize their guiding principles. The

Agile Alliance was born with the publication of the *Manifesto for Agile Software Development* [93]. This alliance was an attempt to unify and legitimize lightweight alternatives to software engineering.

This document enumerates four essential values: "individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan." The main Agile values are accompanied by twelve guiding principles:

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
4. *Business people and developers must work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
7. *Working software is the primary measure of progress.*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity—the art of maximizing the amount of work not done—is essential.*
11. *The best architectures, requirements, and designs emerge from self-organizing teams.*
12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

An essential aspect of these values is the term "over" and not "instead of." The authors of the *Manifesto for Agile Software Development* did not want to eliminate documentation, planning and modeling but rather discussed on the usefulness of these artifacts and practices in the development process. They concluded that documentation should be developed when it helps guide the development process and according to Agile principles, a document that is "just barely good enough" is the ideal document [5]. The perfect document is good enough to communicate its message and adds value, but time has not been lost perfecting an artifact. In the Agile paradigm, requirements and code evolve over time so it best to always create artifacts that are just good enough, leaving the door open for future changes and modifications.

Jim Highsmith expressed this notion in *History: The Agile Manifesto* [93]:

> *The Agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize*

the limits of planning in a turbulent environment. Those who would brand proponents of XP or SCRUM or any of the other Agile Methodologies as "hackers" are ignorant of both the methodologies and the original definition of the term hacker.

Agile methodologies are based on tried and true techniques that have been used through out the history of software development, but combined and formalized in new ways. Many agile methods exists with with an emphasis on differing aspects of software development. Two of the most popular methodologies are Extreme Programming (XP), which focuses on the engineering and technical aspects of development and Scrum which concentrates on the managerial facets of software engineering.

Extreme Programming

Extreme Programming (XP) is an Agile methodology and was developed by Kent Beck during his work on the Chrysler Comprehensive Compensation System [145]. He first published this methodology in 1999 in the book *Extreme Programming Explained.* The extreme in Extreme Programming come from taking engineering principles and pushing them to the extreme [17]:

- *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).*

Kent Beck claims that XP reduces project risk, improves responsiveness to changing requirements and improves productivity through the use of these engineering practices.

Because changing requirements are the norm in much of software development, XP is based on short release cycles and an iterative approach to development. This allows the team to react and adapt to changes throughout the development process. Key aspects of this methodology are the centrality of code, Test Driven Development, constant refactoring and pair programming within a cohesive team.

Coding is used a way of communication ideas and documenting the project. Since code is unambiguous, forming thoughts and communicating through code is an effective way of transmitting and documenting ideas. The practice of Test Driven Development (TDD) ensures that the code works as expected and provides the developers a safety net when refactoring. Architecture and design evolve through iterations and are guided by a shared metaphor. The metaphor is a figurative way of expressing the goals of the system under construction that can be understood by all. Pair programming allows two developers to concentrate on different aspects of implementing functionality. One takes a “strategic” view of the program and considers design and architecture, while the other focuses on the “tactical” considerations of how the program works. This teamwork enhances confidence in the developers, leads to higher quality code with around 15% less defects and provides a mechanism for the transfer of knowledge [39].

XP provides a framework for software development containing elements of project management but focuses on engineering techniques. If these techniques are not followed, then the team cannot claim to practice XP [17]. At the other extreme, Scrum is centered around project management and does not specify the engineering techniques that should be applied. These decisions are left up to the team.

Scrum

Scrum, one of the most popular Agile methodologies, was conceived in the 1990’s by Ken Schwaber and Jeffrey Sutherland and presented in the 1995 OOPSLA’95 Workshop *Business Object Design and Implementation* [134]. Scrum is an iterative and incremental approach to software development designed to maximize flexibility, allowing the team to adapt to environmental complexity [117]. This methodology, according to the authors, increases competitiveness and the probability of success in the face of frequent changes and the inherent unpredictability of software development. Their ideas were based on incremental object-oriented principles combined with the holistic approach to development as conceived by Hirotaka Takeuchi and Ikujiro Nonaka as described in *The New New product Development Game* [135].

Faced with a highly competitive global market, companies in Japan and the United States began applying a holistic method to product development in order to increase speed and flexibility while maintaining high quality and low cost. As in most traditional engineering processes, product development was sequential and linear with different teams specializing in various aspects of development. Takeuchi and Nonaka describe this as a “relay race” strategy where separate teams work on a project, then pass it off to another team like a baton in a relay race. They equate their new approach to a game of rugby where a single team works together to move the ball upfield as a single unit [135]. They assert that this strategy incites innovation, stimulates new types of thinking and learning and helps disrupt antiquated practices that often exist in large companies. Instead of a linear approach, the new multidisciplinary team advances through overlapping development phases. The team works closely together and all members incorporate the different aspects of the project thus increasing awareness of the various constraints and difficulties faced by other members of the team. In this holistic approach, the team members learn new skills that are not necessarily their area of expertise which incites new ideas and fosters creativity.

The team vacillates through divergent and convergent development phases. Possibilities are explored through brainstorming and experimentation in the divergent phases, and then the team integrates the best ideas into working solutions during the convergent phases. This give and take dynamic and hands on approach allows

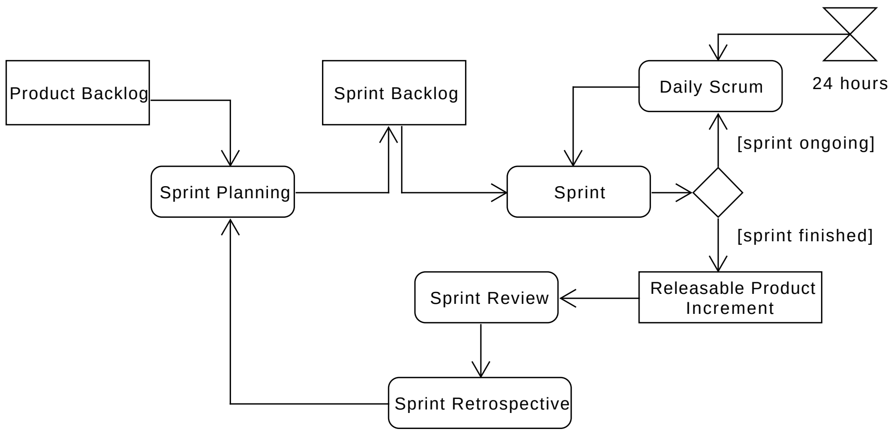


Figure 3.5: Scrum Process and Events

the team to push boundaries and come up with solutions that surpass expectations. According to the authors, successful self-organizing teams have three common characteristics: autonomy, self-transcendence and cross-fertilization. Management is responsible for creating an environment that allows teams to develop these characteristics. Management sets demanding goals and global objectives, but lets the team work uninterrupted and without strict controls, in other words, “top management acts as a venture capitalist” [135]. By allowing this type of autonomy, the teams are free to apply their creativity and know-how to the project and often come up with innovative solutions by thinking “out of the box.” This self-transcendence is the result of the cross-fertilization that takes place with a diverse team working closely together to accomplish a common goal.

The shift from experts working on one part of the project and passing it on to another group of experts to a multidisciplinary team which relies on cooperation, trial and error and hands on learning has resulted in new and revolutionary products being developed in record time [135]. Although not appropriate for all products, this approach has proven effective in the modern and rapidly changing business environment.

Schwaber and Sutherland adapted the concepts of Takeuchi and Nonaka to software development. They integrated the close knit, multi-discipline team focused on communication and individual skills and an iterative approach to answer some of the shortcomings of traditional plan-driven development.

Scrum defines a set of values, roles, responsibilities and ceremonies to manage the development process. External controls are applied to manage risk and to allow the development team to evolve their strategy and adapt to environmental factors. As such, Scrum is foremost a project management framework (Figure 3.5) that provides an approach to development within which different engineering techniques can be applied.

Scrum combines certain aspects of traditional defined processes but treats development as an empirical process based on the pillars of “transparency, inspection and adaptation” [116].

Transparency is the basis of good communication within the team. All members should be aware of what is happening in all aspects of the project. This transparency assists the team in evaluating risk, prioritizing the backlog and working effectively and efficiently which promotes trust. The team defines a shared language to describe the different artifacts and processes, freely share difficulties and obstacles that they encounter, and collectively establish their definition of "Done." This shared definition is essential so that all actors, team members and stakeholders, have the same reference point and expectations for a finished artifact.

Inspection is important to harness the data necessary to detect undesirable outcomes and to make the necessary adjustments to the project.

Adaptation is essential to an adaptive process. If undesirable variances are detected through regular inspections of the product and processes, the team must react accordingly to get the project back on course.

These pillars work together to create short feedback loops that hone the responsiveness of the team allowing them to accommodate the current situation.

The Scrum team is made up of a Product Owner (PO), the Development Team and a Scrum Master (SM). Stakeholders are also a central element of the project even if not part of the Scrum team (Figure 3.6). These roles work together to create a well balanced, self-organizing and cross-functional team [116]. The cross-functional nature of the Scrum team allows the team to develop all aspects of the project without outside intervention and to consistently provide a working version of the product.

Product Owner The Product Owner manages the Product Backlog and is responsible for maximizing delivered value. The backlog is prioritized according to the goals of the project and the PO insures that all team members are aware of and understand the items in the product backlog. Often the PO acts as a proxy for the stakeholders and has a comprehensive understanding of the business needs and is able to communicate them with the development team.

Development Team The development team is self-organizing and decides how best to deliver the promised functionalities from sprint to sprint. The team has a flat hierarchy and although different developers may have specialties, the team advances together and is collectively accountable for their results. Ideally the development team should be relatively small, otherwise communication and coordination can become an overwhelming burden. A small team is more agile and responsive and can more effectively evaluate and adapt to changing conditions.

Scrum Master The Scrum Master is the guardian of the method. He intimately understands all aspects of Scrum and assists the team in accomplishing their goals. He ensures that the team "adheres to Scrum theory, practices and rules" [116] through coaching, organizing events and removing unhelpful outside disturbances.

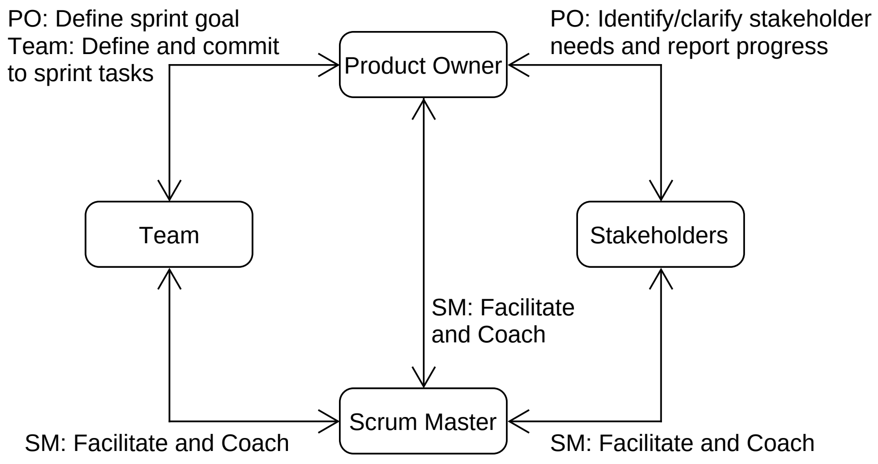


Figure 3.6: Scrum Roles [110]

Stakeholder The stakeholder is anyone who has an interest in the outcome of the project. It can be the client, management or investors. Stakeholders participate in defining the product backlog and provide the feedback necessary for an adaptive development approach.

Scrum is centered around time-boxed sprints that are between two weeks and one month in duration. Ceremonies that are designed to ensure transparency and incite inspection and adaptation at regular intervals within each sprint are: the Sprint Planning, the Daily Scrum, the Sprint Review and the Sprint Retrospective [116] (Figure 3.5). These meetings allow the team to react and adjust to the volatile nature of software development and rapidly changing requirements and technologies.

In the Sprint Planning items are moved from the Product Backlog, presented to the development team and added to the Sprint Backlog. The team decides how much can be completed within the time constraints of a sprint. These User Stories are then developed during Sprints within which daily meetings are held. The Daily Scrum is a short stand up meeting where each team member explains what was done the day before, what is planned for the current day and relate any impediments to progress. These impediments are noted and actions to resolve them are established outside of the Daily Scrum. At the end of a Sprint there is a Sprint Review where the team presents what was accomplished and what was planned but not successfully implemented. Completed work is presented to the stakeholders in a demonstration. The Sprint Retrospective is a meeting for the team where the previous sprint is discussed and analyzed. The team discusses what went well and identifies areas that need improvement. The team then agrees upon actions to remediate the problems found in order to continually improve the team's process [150].

Through regular communication and reevaluation of the status of the project, environmental changes and modifications to requirements can be integrated into future iterations. By constantly inspecting the product and adapting to external and internal modifications within the framework of transparency and quality communication, the self-organizing team can adapt to new circumstances and thus improve their odds of successfully accomplishing their goals.

***

Agile methods overcome some of the shortcomings of traditional plan-driven methods, but require a disciplined approach and high quality developers. Since communication and tacit knowledge largely replace heavy documentation and upfront planning by specialists, an Agile development team must be both diverse and highly competent. Traditional methods' formalized processes and documentation follow the ideas of "Scientific Management," as defined by Taylor, and implement a strict division of labor in an attempt to standardize developer performance. Agile methods, on the other hand, focus on communication and team-work follow more closely the ideas of Herzberg's "Job Enrichment" which attempts to motivate workers by involving them in decision making processes [75, 38]. An essential aspect of agile methods is the skill and competency of developers, but also a continued drive to improve their knowledge and abilities through close collaboration and hands on learning.

As the team matures and becomes more competent, they can redefine their definition of "done" to include stricter quality criteria [116] and consider certain non-functional quality aspects as part of this definition in order to improve the internal qualities of software.

Code Quality and Metrics

4

The Oxford dictionary defines quality as: “The standard of something as measured against other things of a similar kind; the degree of excellence of something [45],” and metric as: “A system or standard of measurement [44].” Indeed, in order to assess the quality of something a comparison must be made and metrics are the means to compare by quantifying quality.

A definition of quality in relationship to software used by Caspers Jones of Namcook Analytics is [77], “the absence of defects that would cause a software application to either fail completely or produce incorrect results.”

The dictionary definition and that of Jones are concise and straight forward; however, quality is a subjective term that can have a multitude of meanings depending on the viewpoint. David Garvin of Harvard University defined five approaches to quality as seen from different business domains [59]:

1 the transcendent approach of philosophy;
2 the product-based approach of economics;
3 the user-based approach of economics, marketing and operations management; and
4 the manufacturing-based and
5 value-based approaches of operations management.

The transcendent approach sees quality as undefinable but universally recognized. The user-based approach is subjective and depends on the preferences of the user. The manufacturing based approach is an engineering conception that measures quality as adherence to design requirements and specifications. In this approach higher quality results in lower costs by eliminating defects in the manufacturing process. The value-based approach measures the cost quality ratio, and the product-based approach defines quality by measuring and comparing certain internal factors of the product.

These different viewpoints are reflected in software quality. End-users and marketing departments are concerned with the consumer perspective focusing on the functional aspects of software quality, while the manufacturing based approach attempts to maximize the quality of software through quality processes. Software engineers, designers and developers tend to concentrate on the product-based derived structural and technical aspects of software.

The overall quality of software can therefore be seen as an interlace of two main classifications of quality: external and internal [108]. These classifications are also referred to as black box and white box, functional and non-functional [112], or static and dynamic [4].

Metrics, measurable properties that must be both reproducible and pertinent, are used for evaluating the quality of software. Different sets of metrics have evolved in order to address diverse programming paradigms. They are exploited in the improvement of development processes and methods with the goal of reducing the costs of developing and maintaining software systems [136]. The first metrics developed to measure software quality were created for procedural oriented programming. They have subsequently been shown to be general purpose and applicable to many programming languages [136]. More recently specialized OO metrics [35] have been developed to address quality issues unique to this paradigm.

Software metrics can be divided into two main categories as defined by the Object Management Group (OMG) [123] and the Consortium for IT Software Quality (CISQ) [36]: Unit Level and Technological/System Level. This paper will focus on the internal properties of software and various metrics that are used to quantify and evaluate their quality at both levels.

Unit level metrics evaluate coding practices and styles to ensure a maintainable and readable code base. They can be easily integrated into development environments and there are a plethora of automated evaluation tools, such as PMD[1], to assist organizations and teams in applying these practices to their projects.

System level metrics evaluate architectural and design factors and can be evaluated by tools that amalgamate metrics, such as Findbugs[2] for Java or the SonarQube[3] platform, which provide a rich web interface for the evaluator.

Because the industry is constantly evolving [118], no standard set of metrics have been defined [65]; nevertheless, commonly used Unit and System level metrics and their relationship with the International Organization for Standardization (ISO) 9126 quality characteristics will be surveyed along with their validation, exploitation and effectiveness in measuring and improving software quality. This report will also canvass certain OO programming principles that have a direct effect on the internal qualities of software. Code that does not respect these principles can often be detected through the use of metrics.

4.1. Quality Models

Several quality models have been created to provide a formal definition and a hierarchical breakdown of these quality domains into elements that can be measured. In the

[1]`https://pmd.github.io/`
[2]`http://findbugs.sourceforge.net/`
[3]`http://www.sonarqube.org/`

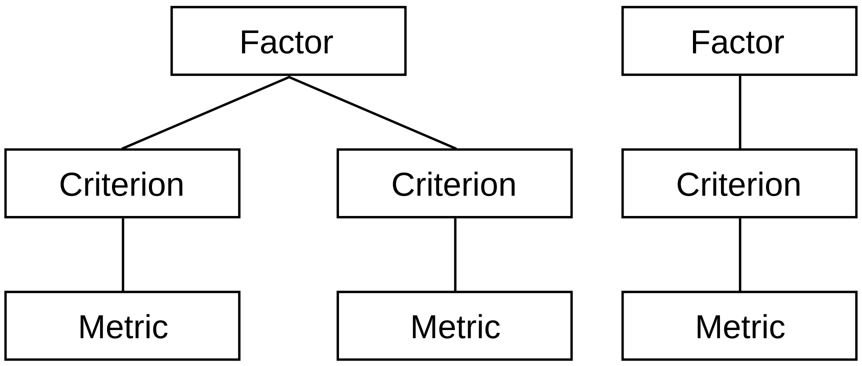


Figure 4.1: Quality Framework [32]

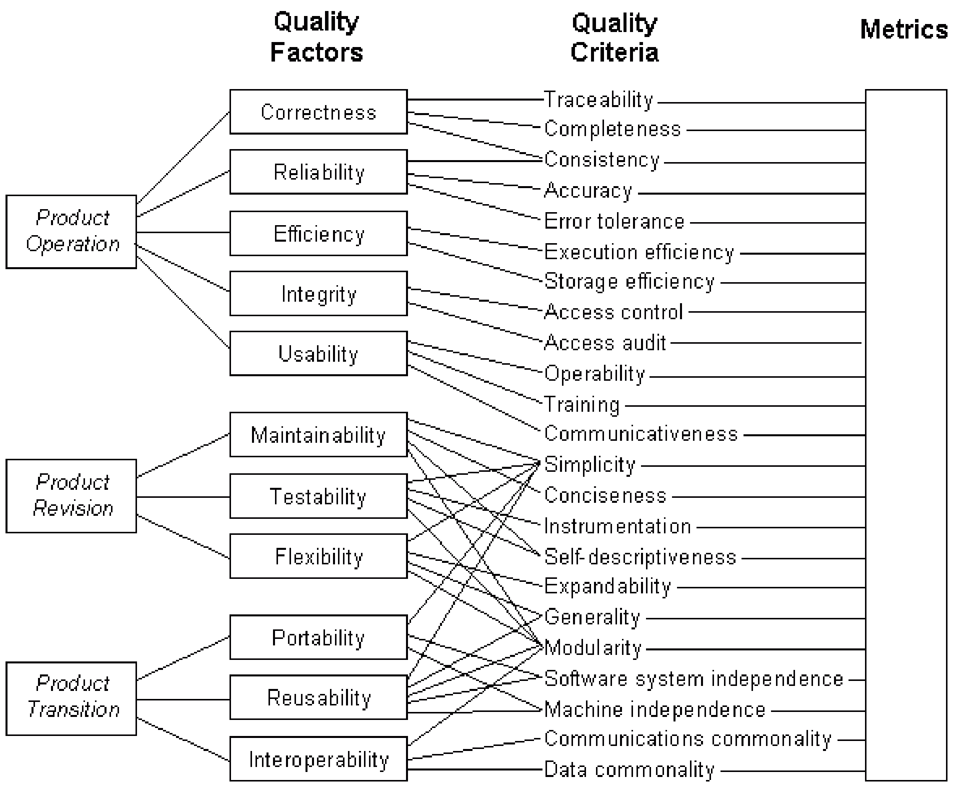


Figure 4.2: McCall Quality Model [137]

1970's McCall and Boehm each developed a quality model and in subsequent years an international standard was created with ISO 9126. These hierarchical models are all based on a similar quality framework (Figure 4.1).

McCall's model (Figure 4.2) defines three main quality factors: Product Operation, Product Revision and Product Transition [32]. These factors are further broken down into characteristics. One of McCall's contributions is the relationship between these characteristics and metrics [112] which were designed to be used not only as an evaluation of a finished product, but also during the development process.

Boehm's model (Figure 4.3) defines seven quality characteristics that are further refined into primitive constructs with metrics to measure them. Each metric is evaluated for its relationship with software quality, weighted by the effort needed to evaluate it. This model focuses on code and programming practices.

ISO 9126 [108] (Figure 4.4), developed in 1991, is a generic model that serves as an international standard that built upon the work of McCall and Boehm. It defines quality as a set of characteristics divided into sub-characteristics which are partitioned into measurable attributes. The main characteristics are Functionality, Reliability, Usability, Efficiency, Maintainability and Portability [79]. The sub-characteristics are clearly defined by the norm and specify the definition of the characteristics. The attributes of the sub-characteristics fall into two categories - external and internal. Internal qualities refer to static code and development processes; external qualities are concerned with how the product works in its environment and refers to code in operation. The development processes influence the internal qualities which in turn

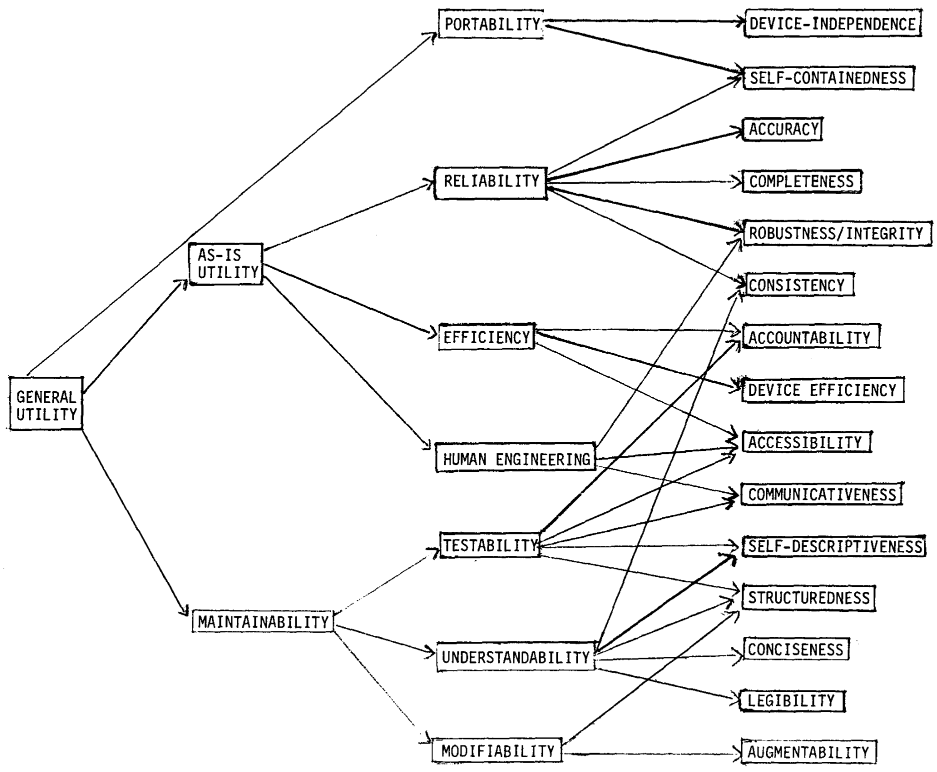


Figure 4.3: Boehm Quality Model [26]

influence the external qualities, affecting the overall quality of a product [3]. A more succinct definition of external and internal quality was given by Kent Beck [17]:

External quality is quality as measured by the customer. Internal quality is quality as measured by the programmers.

The ISO 9126 model (Figure 4.4) proposes a general structure of software quality but does not define concrete ways of linking higher level characteristics (Functionality, Reliability, Usability, Efficiency, Maintainability and Portability) and lower level metrics [100]. This standard was later refined and enhanced by combining aspects of ISO 14598, which provides a quality evaluation process [3], and became the Software product Quality Requirements and Evaluation (SQuaRE) [100] process which was eventually transformed into ISO 25010 [72]. The study of quality models continues and other hierarchical (FURPS, Dromey [108]) and non-hierarchical models (Triangle, Quality Cube [112]) have been developed to address software quality analysis.

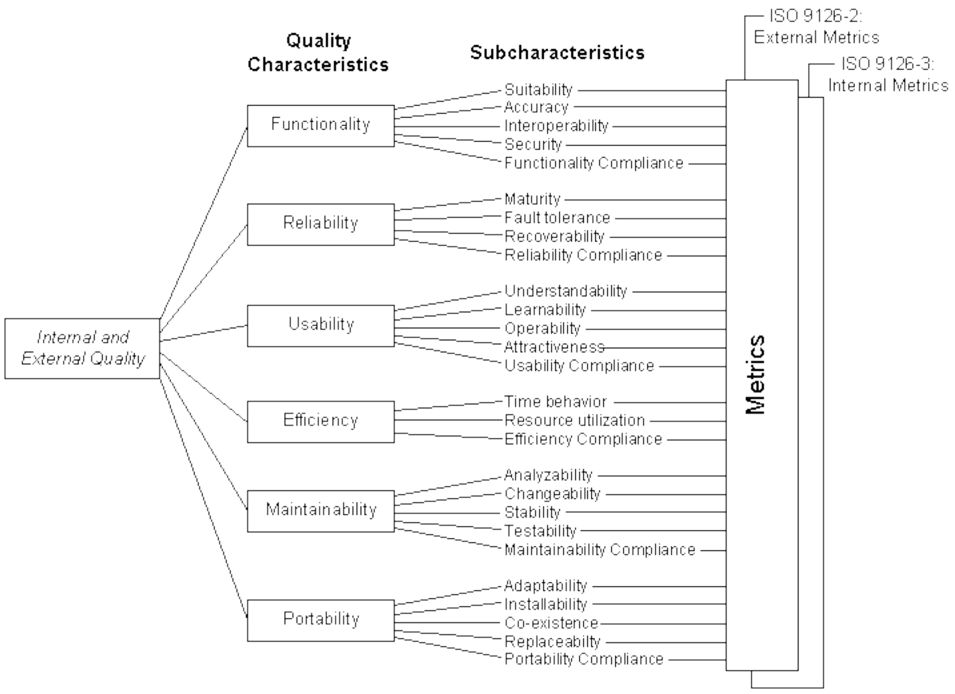


Figure 4.4: ISO/IEC 9126 Quality Model [137]

4.2. Unit Level Metrics

Source Lines of Code

The Source Lines of Code (SLOC) metric is one of the oldest software metrics [77] and is commonly used to measure the general complexity of a code base, programmer productivity and quality through the metric: number of defects per Thousand Lines of Code (KLOC) [51].

The exact method of counting lines has not been normalized and different standards exist [104]. There are two general classifications of this metric: Physical Lines of Code (LOC) or Logical Lines of Code (LLOC). Physical lines of code are the absolute number of lines in a program sometimes including blank lines and comments; logical lines of code are the number of executable statements [77]. An example of the difficulty of counting lines of code by either method is illustrated in an example from Wikipedia [152]:

One physical line of code, two logical lines of code and one comment:

```
/* How many lines of code is this? */
for (i = 0; i < 100; i++) printf("hello");
```

Five physical lines of code, two logical lines of code and one comment:

```
/* Now how many lines of code is this? */
for (i = 0; i < 100; i++)
{
  printf("hello");
}
```

Differences in coding style, such as putting the braces of conditional and iterative blocks on a new line or on the same line could represent a considerable difference in the number of physical lines of code in a large project. LLOC has the advantage that code formatting and style do not affect the line count and is therefore an better metric for comparing different projects.

The SLOC metric is also used to estimate the cost and effort required for the development of software projects through the use of cost estimation models such as the Constructive Cost Model (COCOMO). COCOMO provides a formula based on LOC and project type, while COCOMO 2.0 improves on this model by combining LLOC and other metrics such as object points and function points to more accurately measure project cost [25].

The LOC or LLOC metric have the advantage of being simple to execute; however, they have some shortcomings. LOC tends to favor low level languages where more lines of code are written to create the same functionality reducing the cost per line of code and hiding defect density [77]. The function point metric is gaining momentum in the software industry because it is a more accurate way to measure the cost of a project [13] and is programming language [13] agnostic; however, this metric is much more difficult to measure. Lines of code remains one of the most commonly used metrics to estimate overall complexity and maintainability of software projects.

Code Duplication

The duplication of code is a major problem and expense in software. An estimated 5%-10% of code in large application is duplicated or cloned code [16]. Redundant code is problematic because the same domain logic occurs at different locations in the code base. As the size of the code base increases through duplication more code must be parsed by the programmers making developing new functionality, maintaining existing code and debugging more difficult and expensive. Code duplication can be broken down into three main categories: Copy Paste, Structural and Semantic [127]:

Copy Paste duplication is the easiest to detect and several algorithms have been developed to easily identify it, for example, the Karp-Robin [12] string searching algorithm used by the source code analyzer PMD[4]. Often white space and commented code is ignored and lines or blocks of repeated code can be detected.

Structural or syntactic duplication is more difficult to detect because it concerns similar structures with possible changes in formatting, variable names and types. Baxter's AST (Abstract Syntax Tree) clone detection [16] is an effective algorithms for finding this type of duplication in source code.

Semantic duplication, independent implementations that provide the same behavior, is even more difficult to detect. Several strategies have been proposed [78] but this remains an area for further research.

Duplication occurs for several reasons such as code reuse, repeated computations, performance considerations and unintended inter-developer duplication [69]. Developers often copy and paste blocks of code that provide a certain functionality to

[4]`https://pmd.github.io/`

the one being created and adapt it to their particular need. Impatient developers or those with time constraints can fall into this trap. Code reuse duplication is often the result of a failure to respect the abstraction and reuse principle or through laziness. Repeated computations or routines can often be re-implemented by developers in many places throughout the software instead of centralizing the function. In order to increase performance, duplication can be explicitly added to the code, for example, to cache frequently used data. This type of duplication is a conscience compromise and can be prudent. Inter-developer duplication results from different members of a team coding similar functionality. This can be mitigated through intra-team communication and organization. A more insidious duplication that is often not detected is that of information repeated in comments and in the code itself. Often code is updated or modified, but the comments are not kept current leading to confusion for future developers. For this reason, code should express the low level implementation whereas comments explain the functionality at a higher level of abstraction [69].

Applying metrics throughout the development process can identify blocks of duplicated code and assist the development team in removing them, consequently improving the reuse of code. Once identified, it is essential that duplication be dealt with. Several solutions are available including refactoring common blocks of code into reusable methods or classes. Duplication mitigation is an essential aspect of programming and many principles of coding, such as Don't Repeat Yourself (DRY), are closely linked to this goal. In The Pragmatic Programmer Don't Repeat Yourself (DRY) is defined as [69]:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

When each aspect of business knowledge is isolated, code is easier to maintain and is more robust. Duplicated code leads to duplicated bugs and by removing duplication, any defects in the code can be remediated more efficiently. Having knowledge isolated in one location, maintenance becomes less burdensome because code only has to be modified once. Respecting software coding and design principles and the use of this metric can help remediate duplication in an application and tools such as refactoring can help eliminate problems once found.

Comment Density

Comment density is measured by dividing the number of lines of comments by the number of lines of code. In a study of open source projects researchers found an average of comment density of 19% [8]. The authors correlate comment density and maintainability - an exceptionally low or high percentage can indicate a problem [69]. This metric can be an indicator of quality but should be used with reservation. A study on code readability found that comments do not necessarily assist developers in understanding the code and may be even less important than an blank lines for code clarity [31]. Often well chosen names of methods and variables along with uniform formatting may be more useful than densely commented code [91].

This metric only measures the density of comments and not their pertinence, distribution or placement. Commented out blocks of code can also cause this metric to give inaccurate results because this dead code will be counted as comments. Comments should aid developers in understanding the code, but not all comments have the same value. Some can aid in maintenance tasks by making the code easier to understand; however, excessive or redundant comments can increase maintenance problems by violating the DRY principle. As mentioned in code duplication, comments can contain a repetition of knowledge or procedures and is not always kept up to date with the code. Comments should therefore focus on the "why" because the "how" is already expressed in the code itself [69]. This strategy helps eliminate duplication and associated maintenance problems.

Complexity

Complexity is a characteristic of software that influences readability and usability and has a direct impact on maintainability, evolvability and testability. Complexity is also related to the effort required for testing and to the robustness of an application. Several metrics have been developed to measure complexity, the most commonly used being McCabe's cyclomatic complexity and Halstead complexity. Source lines of code (LOC) is also used to determine an order of magnitude of program complexity. Software complexity as measured by these metrics is not to be confused with algorithmic and data structure complexity which deals with computation time and the use of resources.

Cyclomatic Complexity

McCabe's Cyclomatic Complexity (CC) was developed by Thomas McCabe in the 1970s and is one of the most frequently used complexity metrics [27]. Cyclomatic complexity is based on graph theory and measures the number of paths through a program, module or method; however, directly measuring the number of paths is impractical because it is possible to have an infinite number of paths through a graph that contains cycles. McCabe developed a method to count the number of "linearly independent paths." [95] In practice, this metric is calculated by counting the number of conditions represented by certain keywords such as: `if`, `for`, `while`, `&&`, `||`, `switch`, `return` and `?`.

The general formula for calculating cyclomatic complexity of a graph is [120]:

$$v(G) = edges - vertices + 2$$

The modified version using program language keywords is [120]:

$$v(G) = binaryDecisions + 1$$

One of the advantages of this metric is that is it easy to understand and calculate by hand or machine and provides a finer grained alternative to LOC for measuring complexity. Programmers can easily calculate the cyclomatic complexity of programs they are writing, while they are writing them, and make adjustments in order to limit the complexity. McCabe suggests an upper limit of 10 for methods [95] and this number is still suggested in modern metric analysis suites [124]. In object oriented programming, the cyclomatic complexity of a class can be measured by adding up the complexity of all its methods. This metric can highlight classes that may have too many responsibilities and therefore low cohesion (chapter 3.2). A drawback to this method is that it does not take into account the complexity of conditions. For example the two conditions below both add one to the CC even though the second is a much more complex statement.

```
if(x > 5)

if(x > b * (b - x * a sqrt(z -1)))
```

Here is an example of measuring cyclomatic complexity of a method written in Java resulting in a CC of 5 [125]:

```
public void process(Car myCar){ // +1
   if(myCar.isNotMine()){ // +1
      return; // +1
   }
   car.paint("red");
   car.changeWheel();
   while(car.hasGazol()
         && car.getDriver().isNotStressed()){ // +2
      car.drive();
   }
   return;
}
```

This metric provides a gauge for the testability of methods and expected test coverage. A method with a high CC value will be very difficult to test because of the high number of paths through the method. Untested paths can be discovered using CC [95]. The effort required to test all paths is proportional to the CC value and influences the overall programming effort required. Methods with a high CC are likely to take more effort to modify and maintain because they tend to be more difficult to read and understand. They also tend to be more fragile and difficult to refactor because of their corresponding lack of test coverage.

While cyclomatic complexity is often used, some studies [121] have shown that it is not significantly more effective than LOC for determining complexity. Because CC analyses code lexically and not structurally, some design improvements such as removing duplication can actually increase the CC of programs.

When applied to an entire program this metric does not provide much value because it simply measures the logical complexity of the entire program, but on the class or method level it can indicate problems related to coupling (chapter 4.1), encapsulation and cohesion (chapter 4.2) [126].

Halstead Complexity

Halstead complexity was developed in the 1970s as part of Halstead's Software Science which attempted to measure software quantitatively through metrics. These metrics focus on the computational complexity [27] of a program or module and the number of errors, whereas cyclomatic complexity measures logical complexity. Ease of maintenance, development and the number of expected bugs are also measured.

This metric uses the number of operators and operands in order to evaluate program length, vocabulary, volume, difficulty, effort, and the number of defects [9, 121, 86].

The elementary metrics: number of operators and operands; and unique operators and operands, are used to calculate the more complex metrics of length, volume and vocabulary. These are then used to calculate the most intricate metrics: difficulty, effort, time to develop and number of bugs [86].

Program length (N) is the sum of the number of operators (N1) and operands (N2). Program vocabulary is the sum of the number of unique operators (n1) and operands (n2). Program volume (V) is the program length times the 2-base logarithm of the program vocabulary and provides a measure of the size of functions or methods [138]. These metrics measure basic attributes of a program [111].

Program Length $N = N1 + N2$

Vocabulary Size $n = n1 + n2$

Program Volume $V = N \times log_2(n)$

Program difficulty measures the difficulty (D) to read or understand the program and is proportional to the number of unique operators. Effort (E) is the product of the difficulty and volume and is used to calculate the time needed to code the program. Effort refers to “the number of elementary mental discriminations.” [111] The time (T) is calculated by dividing effort by 18. The number 18 was derived by Halstead through experimentation, which according to his estimates, provide T in seconds. The number of bugs is calculated by taking effort to the exponent of 2/3 and dividing by 3000. This estimate has been shown to be rather conservative and more bugs are normally found [138]. These metrics are principally concerned with maintainability.

Program Difficulty $D = \frac{n1}{2} \times \frac{N2}{n2}$

Effort $E = V \times D$

Time to Implement $T = \frac{E}{18}$

Number of Bugs $B = \frac{E^{\frac{2}{3}}}{3000}$

These calculations have been judged by some to be ambiguous because the left hand and right hand sides of the calculations are based on different units of measure [111]. For example adding the number of operators and operands to get program length, or calculating time without specifying a unit of measure. An exact method for counting operators and operands was not provided by Halstead [121]; moreover, the method is language dependent making the choice of operators and operands ambiguous [86]. Nevertheless, this metric is an important part of the Maintainability Index metric which combines CC, Halstead Complexity, SLOC and comment density.

Different approaches have been developed to measure the complexity of a program, module or method. Engineers and software developers should be concerned with complexity as it is closely related to the readability, maintainability and evolvability of code. Highly complex code is also very difficult to test, often resulting in poor test coverage and an increased risk of undetected bugs.

Test Coverage

Test or code coverage, as with most aspects of quality, comes in two flavors: structural (glass box) and functional (black box). Functional coverage is based on program specifications; structural coverage concerns the internal aspects of software.

Structural Coverage

Structural code coverage is made up of several metrics, the most commonly used being statement coverage, decision and condition coverage [140].

Statement Coverage or line coverage measures whether or not a line of code has been executed by a test. The basic calculation of statement coverage is the number of lines of code tested divided by the number of lines of code LOC.

```
public String exampleMethod(int x){
   Integer a = null;
   if(x > 0){
      a = x;
   }
   return a.toString();
}
```

In the above Java example simply calling the method in a test with an `x > 0` will result in 100% coverage. This can be problematic because, for example, calling the method with an `x <= 0` will throw a `NullPointerException` despite having 100% coverage. The same is true of loops and statements containing multiple conditions. As long as they are called by a test then they are considered covered [41].

Decision Coverage or branch coverage determines if both Boolean values have been executed for each `if` or `switch` statement. This improves upon line coverage and could help catch the `NullPointerException` in the above example; however, in languages with short circuit operators this metric does not take into account multiple conditions in the same statement. An example of this from Steve Cornett in Code Coverage Analysis [41]:

```
if (condition1 && (condition2 || function1())) statement1;
else statement2;
```

Complete decision coverage could be obtained without ever calling `function1` because if `condition1` and `condition2` are `true`, `function1` will never be evaluated.

Condition Coverage tests whether both Boolean values have been tested for each condition, but not necessarily every possible combination. An example from Wikipedia demonstrates this [142]:

```
if a and b then
```

Condition coverage will be 100% by testing `a=true`, `b=false`; or `a=false`, `b=true`; however, neither of these combinations satisfy the `if`.

Other Coverage Metrics Other more rigorous metrics exist such as modified condition / decision coverage, multiple condition coverage and path coverage. These metrics require more stringent testing and are used for critical application such as by the U.S. Department of Transportation Federal Aviation Administration (FAA) for critical flight software [142], but are not commonly used in enterprise applications.

Test coverage gives an overview of how much of the code base is exercised with tests but does not evaluate the quality of the tests nor the number of defects prevented or discovered by the tests. Nevertheless, the act of testing code often reveals defects during the development process and helps to insure that the program functions correctly [140]. As stated by Dijkstra, testing is put in place to discover bugs, but cannot guarantee that software is bug-free.

Testing shows the presence, not the absence of bugs –Dijkstra (1969) [46]

Structural tests can aid programmers in developing their algorithms and provide the developers with confidence [18] that their code functions as expected and does not contain any obvious bugs.

The quality of tests plays an important role in their usefulness. It is possible, for example, to have high test coverage without really testing the code [52] by simply calling methods in a test suite. Mutation testing is one way to insure the quality of tests and to find errors in the tests themselves [7]. Certain mutations such as changing variables, modifying conditions and arithmetic operations are introduced into the code being tested to evaluate the effectiveness of tests. Mutant testing is computationally intensive and often takes a considerable amount of time to execute; nevertheless, certain techniques such as selective mutation [107] helps reduce this cost by up to 75%.

Other testing errors can reduce the quality of tests and their usefulness as quality indicators. A study by IBM in the 1970's found that about 15% of unit tests contained errors and about 20% of tests duplicated already tested functionality [77]. These tests may increase the code coverage metric, but do not provide any additional quality assurance. Other test errors such as faults of omission [89] are difficult to test because they are holes in the logic tested. Often when these tests are missing the logic is also missing from the program. Writing good tests is not any easy job [89] and although the test coverage metric provides an overview of the tested state of an application, it does not guarantee a defect free system. Nevertheless, structural tests can serve secondary purposes as regressions tests and as an aid in program design.

Structural tests normally execute very quickly and their execution can be integrated into a continuous integration process. By following this procedure, a battery of tests can be executed after every code modification, revealing unexpected regressions and validating the changes [69]. These regression tests also provide developers with the freedom to modify the code design as necessary. Altering the design without changing the functionality is referred to as refactoring [54]. When a code base is covered by unit tests developers have the liberty to refactor code and improve design. The first step in any refactoring process is to have a test suite that covers the section of code to be refactored [54]. This way, as the code is modified, the tests can be executed to verify that functionality has not been broken and regressions have not been introduced.

Advocates of test oriented methods such as TDD [18] argue that unit tests can improve the design of programs and even serve as a form of documentation. By writing tests before functional code is written, the user requirements and interaction with the program are expressed in these tests. In unit tests, program functionality is communicated succinctly and is always up to date. As mentioned in the discussion of code duplication, comments and low level documentation is not always current. Functions can change and there is no guarantee that the comments follow suit [69]. Unit tests, on the other hand, are executed code that are more likely to be accurate and reliable.

High test coverage can correlate with an improvement in certain quality metrics. For example code with a high cyclomatic complexity is arduous to maintain and evolve and is very difficult to test. By insuring a high percentage of test coverage, a reduction in CC can follow. Programs with too much coupling will also be difficult to test because a large cluster of objects will need to be created in the test set up. Having a high test coverage helps eliminate some of these negative aspects of code that are directly related to quality.

Nevertheless, testing has a certain overhead and some studies have show that the overall testing effort can amount to 50% of the cost of development [67]. Structural testing occupies a portion of that amount, but can provide many benefits: catching defects early in the development process, improving code quality and allowing for continual improvements through refactoring. It has been shown that defects in software get more expensive to fix as the development cycle progresses and by testing early and often, defects can be found in a timely manner [24].

Functional Coverage

Functional or “black box” testing tests the behavior of a program to verify its accordance to specifications. These tests are often written as scenarios with specific data, execution steps and expected results. These tests reflect the client requirements and verify the external qualities of software. Many of these tests are executed manually, but certain tools have been developed to automate this type of testing, such as Selenium for web applications [119]. These tests can be created, managed and executed with tools like Squash Test[5]. Some functional tests metrics that can be derived from these test suites are test failure rate and requirement and test coverage.

These tests can be correlated to the functional requirements of a program and the overall requirement coverage can be measured; however, the quality of these tests cannot be quantified with a metric because they depend on the quality of tests themselves [99]. While structural tests can determine if the internal functions of a program work correctly, functional tests can verify that the software satisfies client

[5]`www.squashtest.org/en`

requirements. These two types of tests work in harmony to provide a more robust and reliable product.

4.3. System Level Metrics

Coupling and Cohesion

Coupling and cohesion are two metrics that measure the complexity of a program at the architectural or module level. These metrics were originally described in the context of procedural languages and were later adapted to the OO paradigm [50]. Yourdon and Constantine defined coupling as [156]:

> *a measure of the strength of interconnection*

and cohesion of a module as:

> *how tightly bound or related its internal elements are to one another.*

In other words, coupling is the amount of interdependence between modules; cohesion is how well the elements of a module work together to provide a certain functionality [50]. In procedure oriented programming modules are procedures or subroutines, while in object oriented programming they can be methods, classes [28] or packages [66]. These metrics help to evaluate the architectural complexity of a software system.

Coupling

Traditional Coupling

Coupling, in its original form, was described as “the measure of the strength of association established by a connection of one component to another” [50]. Strong associations between modules increase the complexity of programs because they become tightly intertwined. This coupling makes understanding, modification and the reuse of components more difficult. The degree of coupling describes how much, how complex and how explicit the interrelationship of components are [50]. Many categories and measures of coupling have been described [28] in scientific literature for different programming paradigms; however, there is no official set of coupling metrics.

Traditional coupling deals with the way that modules interact with each other and object oriented programming expands this definition taking into account class, package or component interactions. Classical interaction coupling can take several forms: content, common, control, stamp or data [50].

Content coupling is when modules or methods directly access the internals of another module. The calling method must know exactly how the other method works and reacts accordingly. These two methods become tightly coupled because any change to one directly affects the other. In object oriented programming this occurs when an object method alters the internal fields of another class. This can be avoided by using encapsulation, but often private fields are accessible by other classes through access methods, opening the door for content coupling.

Common coupling is when different modules interact via global variables or data. This can cause problems because one module can alter the state of the global variable thus affecting all other modules that are using this data. Modifications to the global data is difficult because of the number of potential users.

Control coupling is when modules interact through method parameters that influence the execution flow of another method. This is often introduced by passing `boolean` operators to handle special cases. Changes to the called method may have negative repercussions on the calling method. This type of coupling can be tempered in object oriented programming through the use of polymorphism.

Stamp coupling is when data structures are passed as parameters but only part of the data is required for the execution of the method. This can cause problems when the data structure is modified. The called method may need to be modified to accommodate this change even though its execution is only related to certain pieces of data. This type of coupling can be improved by upgrading to data coupling.

Data coupling is the most desirable level of coupling. It is similar to stamp coupling except that only pertinent data is sent and therefore the entire data structure is used by the called method. Any changes to the data structure will affect the called method minimizing the need to modify it for unrelated changes.

Object Oriented Coupling

Traditional metrics are applicable to OO programming, but certain metrics specific to these languages have been developed. In object oriented languages the metric Coupling Between Object classes (CBO) was proposed by Chidamber and Kemerer (CK) [35]. This metric counts the number of times that a class uses methods or variables of another class including inheritance and polymorphism [99]. The authors contend that highly coupled classes are difficult to reuse because they are not modular. CBO measures coupling between pairs of classes but does not take into account transitive coupling where a class can be coupled with another class via an intermediate class. This metric, therefore, does not detect violations of the Law of Demeter. This programming heuristic states that [69]:

Any method of an object should call only methods belonging to:
– itself
– any parameters that were passed in to the method
– any objects it created
– any directly held component objects

Classes should only communicate with their direct neighbors and remain as independent as possible through the mechanisms of encapsulation and information hiding. For example, the chained call `a.getB().execute()`, creates hidden coupling and dependencies between classes that are not in direct contact [50]. The call that traverses the class "B" could be replaced by a method in "A" that acts as an intermediary, thus decoupling these classes. By decreasing complexity, the amount testing needed is reduced and the overall architectural design is improved [35].

Component Level Coupling

Another set of object oriented coupling metrics that counts the number of incoming and outgoing couplings at a component level was developed by Robert Martin [92]. He defines a component as a set of classes that function together as an "independently deployable binary unit," such as a Dynamic Link Library (DLL) or Java Archive (JAR). He defines three principles of component coupling: Acyclic Dependencies Principle (ADP), Stable Dependencies Principle (SDP) and Stable Abstractions Principle (SAP).

THE ACYCLIC DEPENDENCIES PRINCIPLE states that a dependency structure, seen as a directed acyclic graph, should not form cycles (Figure 4.5). This simplifies the overall architecture because inter-module dependencies are well controlled and changes to one component only affect downstream components. The software can be tested and built from the bottom up[6] because the lowest level components have the least number of dependencies. If cyclic dependencies exist, then all components are tightly coupled and can not be reused separately and any changes to one component requires retesting all components.

THE STABLE DEPENDENCIES PRINCIPLE defines the relationship between stable and unstable components. Certain components are not designed to change and can be depended on by other components. These components are considered stable. Other unstable components are designed to change frequently and should not be relied upon by other components. Stable components are difficult to change because any modification can have repercussions that ripple through the components that they have responsibilities to. SDP states that dependencies should move in the direction of stability. Figure 4.6 shows a stable component that is depended on by other components but remains independent, and an unstable component that relies on other components but is not depended on. Martin developed a metrics to measure stability: afferent and efferent coupling that is discussed below.

THE STABLE ABSTRACTIONS PRINCIPLE states that stable components should be abstract so that their functionality can be extended without modification, and unstable components should be concrete and easily modifiable. "Dependencies should run in the direction of abstraction." [92] Unlike classes which are either abstract or concrete, components can have a degree of abstractness. In Figure 4.6), the stable component should have a high level of abstraction, and the components that use is should have lower abstraction. One of the principal mechanisms for implementing this principle is the Open Closed Principle that

[6] Acyclic component A in Figure 4.5

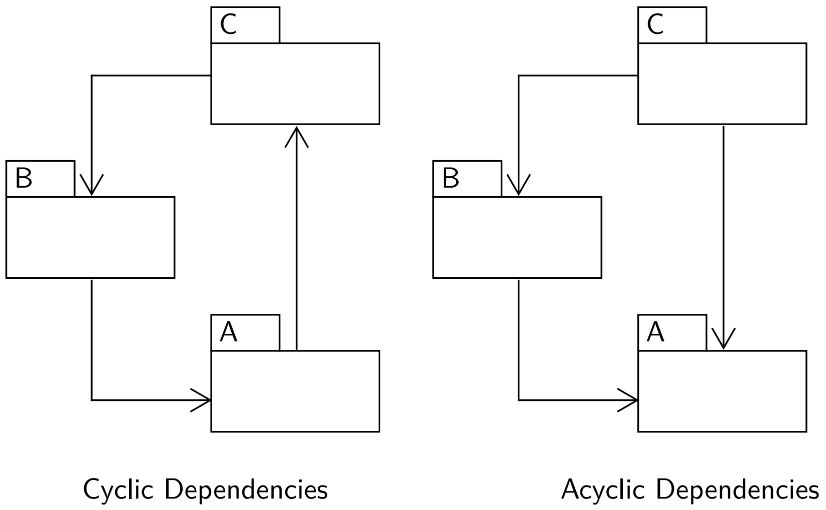


Figure 4.5: Acyclic Dependencies Principle [92]

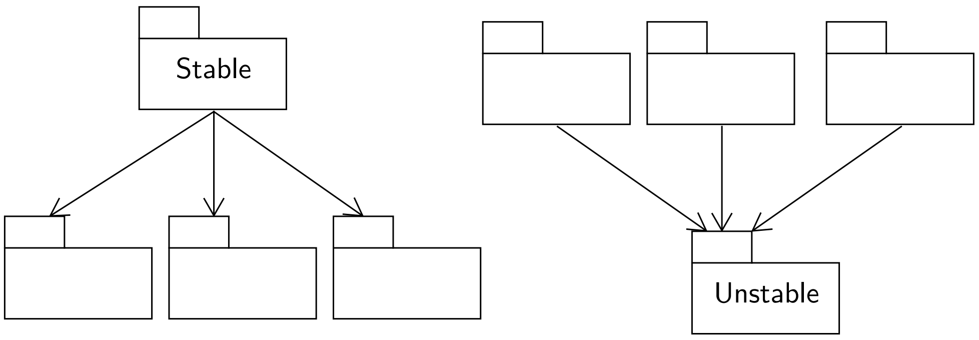


Figure 4.6: Stable Dependencies Principle [92]

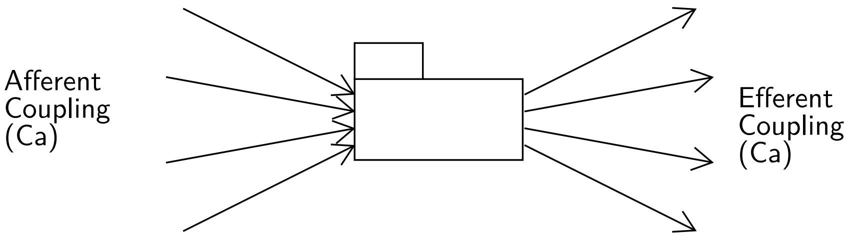


Figure 4.7: Afferent/Efferent Coupling [122]

is addressed in subsection 4.4 and can be measured by the Abstraction and Distance metrics discussed in Package Level Metrics (subsection 4.3).

Coupling can occur in two directions: import and export and thus fall into two principle categories: Afferent Coupling (CA) and Efferent Coupling (CE). Martin defined afferent coupling between components as "the number of classes outside this component that depend on classes within this component" and efferent coupling as "The number of classes inside this component that depend on classes outside this component. [92] " By classifying coupling into two categories, the total coupling of software can be measured in a straightforward way.

Afferent Coupling considers the number of "clients" that a component has. Components provide services to other components resulting in interrelationships. These dependencies can make modifying a service component difficult because any changes may affect any or all of its clients. The higher the degree of coupling, the more difficult modifications become. On the flipside, if a component is not used by any other components, it may be dead code that should be eliminated – a total absence of afferent coupling could indicate a problem [122]. This metric can be calculated by counting the number of classes that use a given component.

Efferent Coupling measures the number of "imports" – the number of other components that a component depends on. Changes to these external classes potentially affect this component in a negative way. A component that is dependent on many others may exhibit signs of low cohesion and may implement too many functionalities. This metric is measured by counting the number of other components that a given component depends on.

Cohesion

> *The cohesion metric deals with how closely related all parts of a module are. High cohesion implies that all parts of a module are closely related to whatever functionality the module provides, and hence probably easy to read and understand.* – Capers Jones [77]

Cohesion is one of the key aspects of internal software quality. Highly cohesive modules are easier to maintain and maximize the possibility for reuse. They encapsulate one area of the problem domain and therefore any changes to requirements only affect a minimum number of modules and each in an isolated way. This helps increase productivity and reduce risks [69] by improving the afore mentioned internal quality characteristics of understandability, maintainability, testability, modifiability and reusability. Cohesion is the driving principle behind many OO principles discussed in subsection 4.4.

By having a clear and delimited responsibility, highly cohesive modules or classes are easier to understand. Maintainability is positively affected because changes to requirements are isolated and making it easier to find the modules that are affected by the change. Testing these modules requires less effort because their sphere of authority is well defined and they can thus be reused in a straightforward way.

Traditional Cohesion

Several types of traditional cohesion were defined by Stevens in the 1970s [132] for procedural oriented languages running the gamut from purely coincidental, to fully functional [132, 50].

Coincidental cohesion is when different elements have nothing in common except their location.

Logical cohesion is when different elements share a common functionality such as input/output.

Temporal cohesion is when elements have logical cohesion and are performed at the same time.

Procedural cohesion is when elements are connected by the execution flow: one event leads to another.

Communicational cohesion is when elements have procedural cohesion and use the same data.

Sequential cohesion is when elements have communicational cohesion and are connected sequentially, like in an assembly line.

Functional cohesion is when all elements participate together to handle a single part of a problem domain and represents highest and most desired level of cohesion.

Object Oriented Cohesion

These definitions are expanded upon in OO programming to measure class cohesion. Classes are different than classical modules because they can contain several methods that share fields. A common metric, the Lack Of Cohesion of Methods (LCOM), for measuring class cohesion was proposed by Chidamber and Kemerer in their suite of object oriented metrics [35]. This metric measures the how the methods in a class use fields. These variables represent the state of a class; therefore, the more methods in the class that manipulate these variables, the higher the cohesion. In a class with perfect cohesion all methods use all instance variables.

The original LCOM metric has been shown to have shortcomings [34]. At a conceptual level, it is uncertain whether or not perfect cohesion, where every method uses every field, is a desirable design trait. Also, the original LCOM metric calculation does not take inherited attributes into account. Access methods (getters and setters) can artificially degrade the cohesion metric of a class because they typically only use one instance variable per method [28]. This metric was therefore revised several times. Subsequent adaptations of the original metric LCOM2, LCOM3 and LCOM4 attempt to improve upon the simplicity of the original method but are still found to be lacking in precision. The latest version by Henderson-Sellers is LCOM5 [28]. In LCOM5, the result of the calculation is in the range [0,1] where perfection cohesion results in a value of 0 and a value of 1 when each method uses one instance variable. Despite the number of revisions, the LCOM metrics do not necessarily guarantee good design.

Classes that have clearly disparate sets of instance variables exhibit low cohesion and should be broken up into separate classes with higher cohesion; however, measuring cohesion quantitatively has proved to be an arduous task because of the plethora of factors that must be taken into account. Research to find an accurate way of measuring cohesion is ongoing.

Component Level Cohesion

Classes that handle a similar functionalities are grouped together into packages or components. As with classes, components should represent a cohesive whole. These components balance the sometimes opposing qualities of reusability,maintainability and ease of development. Martin [92] defined three component level principles of cohesion: the Reuse/Release Equivalence Principle (REP), the Common Closure Principle (CCP) and the Common Reuse Principle (CRP).

Reuse/Release Equivalence Principle A reusable component provides a coherent set of functionalities that are grouped together and follow the same release cycle. Reusable and non reusable software should not be grouped together.

Common Reuse Principle states that if a component is designed to be reusable then the classes it contains must also be reusable. Classes that work together and have many dependencies on each other should be packaged together, and more importantly, classes that are not used together should not be in the same component. A components classes are used together as a whole and therefore need to be a logically cohesive unit.

Common Closure Principle states that a component should not have multiple reasons to change. Classes that change together should be packaged together. This is similar to the Single Responsibility Principle but applied at the package level.

Coupling and cohesion influence the maintainability, understandability, modifiability and testability of software [50]. These two concepts are closely related because modules that have high cohesion tend to have low coupling, and vice versa. To improve the overall quality of software, it is preferred to have highly cohesive modules that are coupled with a minimum number of external modules. In object oriented programming these two concepts are closely tied to design principles that can help developers improve the architecture and internal quality of software. Nevertheless, the notion of coupling and cohesion are difficult to measure and evaluate using metrics. Many metrics have been developed, but their shortcomings led to new variations and even to their abandonment by metric suites. These concepts are best managed through developer experience and the application of quality design and coding principles.

In OO programming classes are grouped into packages, and their certain quality properties can be analyzed with package level metrics.

Package Level Metrics

Robert Martin developed several package or component level metrics related to the component coupling and cohesion principles previously mentioned. These include Instability (I), Abstraction (A) and the Distance from the Main Sequence (DMS).

Instability

The afferent and efferent coupling metrics (subsubsection 4.3) are used to calculate the Instability of components [92].

$$I = \frac{Ce}{Ca + Ce}$$

Instability measures the ratio of efferent coupling to total coupling. The instability metric has a range of [0,1] with 0 being maximally stable and 1 being maximally unstable. A maximally stable component is depended on by other components (Ce = 0) but does not depend on other components (Ca > 0). Martin refers to this type of component as “responsible and independent” because many other components depend on it (responsible) and is unchanging (independent), but does not need to change very often because it does not depend on other components. On the other extreme, a component with an instability of 1 depends on other components (Ce > 0) but no other components depend on it (Ca = 0). This type of component is “irresponsible and dependent.” These components depend on many external classes (dependent), but no other component depend on them (irresponsible). Figure 4.6 illustrates these maximally stable and maximally unstable components.

In a software system there needs to be a blend of stable and unstable components. Dependencies must exist in software, but the negative effects of coupling can be minimized by balancing the roles of different components. The Stable Dependencies Principle is based on this definition of stability and at a component level, states that components should only depend on components that are more stable than itself. Dependency towards stability. Stability is also distributive: a component can only be considered stable if its dependencies are also stable [99].

Abstraction

Martin defined A as the ratio of the number of abstract classes (Na) in a component to the total number of classes in a component (Nc).

$$A = \frac{Na}{Nc}$$

This metric gives a value in the range of [0,1] where zero means that the component does not contain any abstract classes and one means that the component is made up exclusively of abstract classes. This metric, in combination with Instability is used to measure the Stable Abstractions Principle (SAP).

Distance from the Main Sequence

The relationship between instability and abstraction is measured by the Distance from the Main Sequence metric. The most desirable components are stable and abstract (0,1) or instable and concrete (1,0). These two points are plotted on a graph of Abstraction and Instability. The diagonal line joining these two point represents the main sequence (Figure 4.8).

Martin defines the combination of stability and abstraction that should be avoided as the “Zone of Pain” that falls near (0,0). Components in this zone have high stability

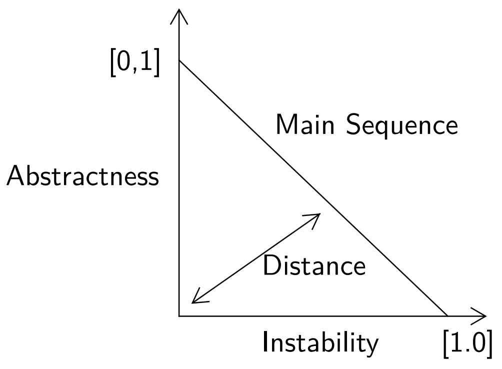


Figure 4.8: Distance from the Main Sequence [92]

but low abstraction. They are thus difficult to change and hard to extend leading to maintenance problems. On the opposite end of the spectrum highly abstract components without dependencies are in the "Zone of Uselessness" near the point (1,1).

The Instability Abstraction ratio of components in a program can be evaluated using the Distance from the Main Sequence metric.

$$DMS = \frac{|A + I - 1|}{\sqrt{2}}$$

The normalized version $DMS' = |A + I - 1|$ falls into a range of [0,1] where 0 means that a component is on the main sequence and 1 means that it is a far away as possible. This metric is used to calculate the SAP and can highlight outlier components that may be difficult to maintain and change.

Object Oriented Metric Suites

Unit level and traditional system level metrics can be applied to OO systems but they do not take into account some of the complexity of this programming paradigm. These metrics can be used to analyze class methods which correspond to functions in traditional development [114]; however, it has been questioned whether metrics such as LOC that were developed for procedural oriented programming can effectively account for the structural complexity found in object oriented programming. Another school of thought purports that because traditional metrics have been validated empirically and are well understood, they can be of value in an OO context [136].

Despite the polemic, traditional metrics are often applied to object oriented software; nonetheless, metrics that specifically address the intricacies of this style of programming have been developed. These metrics attempt to take into account encapsulation, information hiding, inheritance, polymorphism and message passing [43].

Chidamber and Kemerer Metrics

Chidamber and Kemerer developed a suite of metrics for object oriented design. They developed six new metrics based on the ontology of Bunge which deals with definitions and representations of the world and are thus directly applicable to object oriented design [35]. CK also developed a tool to collect data and analyzed the results empirically.

Weighted Methods per Class (WMC) measures the complexity of a class. The number of methods of a class and their complexity is used to calculate WMC where a Class C_1 has methods $M_1, ...M_n$ with $c_1, ..., c_n$ complexity:

$$WMC = \sum_{i=1}^{n} C_i$$

According to CK, the more complex a class, the more time and effort will be needed to develop and maintain it. Classes with a large number of methods will affect any sub-classes because the child classes inherit its parent's methods. Classes with many methods also tend to be application specific, making reuse difficult.

This metric has received some criticism because it is based on the complexity of methods, and a the method for calculating complexity was never specified; nevertheless, it can used as a general metric to calculate class size [35, 43].

Depth of Inheritance Tree (DIT) is a measure of the number of nodes in an inheritance tree from a given class to the root node. In the case of multiple inheritance the longest path is used. Classes that are deep in the inheritance tree inherit many methods which increases its complexity. Increased complexity makes it more difficult to test and predict the behavior of these classes. Classes with a shallow inheritance tree may indicate that they are not taking full advantage of the inheritance mechanism of object oriented programming. However, this metric does not take into account a wide but shallow inheritance tree, nor the use of frameworks that hinge on inheriting a base class. The former would give a low DIT value even though it inherits many methods, while the latter could artificially increase the DIT value [35, 43].

Number of Children (NOC) is the number of direct descendants of a class. This metric can indicate the amount of reuse of a class and can highlight classes with too much responsibility. A class with many children is more complex and therefore difficult to modify increasing the testing effort required [99]. NOC can help bring to light a class' influence on the overall design of software [35].

Coupling Between Object classes (CBO) counts the number of times a class uses methods or variables of another class including inheritance and polymorphism and is described under Object Oriented Coupling (subsubsection 4.3).

Response For a Class (RFC) is the number of methods that are accessible to a class. RFC is the sum of local class methods, visible inherited methods and overridden methods. This metric includes methods accessible from outside the class and measures the amount of potential communication with other classes. A high value indicates a class with high complexity that is difficult to understand and test [35]. It is unclear whether this metric takes into account nested calls: local methods that can potentially call other methods [99].

Lack Of Cohesion of Methods(LCOM) measures the how the methods in a class use fields. These variables represent the state of a class; therefore, the more methods in the class that manipulate these variables, the higher the cohesion and is described in more detail under Object Oriented Cohesion (subsubsection 4.3).

Lorenz and Kidd Metrics

Lorenz and Kidd in their book *Object-Oriented Software Metrics* [85] defined eleven OO metrics divided into three categories: size, inheritance and internal metrics.

The size metrics count the number of methods and variables of a class to determine complexity, the inheritance metrics measure the quality of the inheritance and the internal metrics analyze class characteristics [139]. A subset of these metrics will be examined.

Number of Methods (NM) measures the number of total private and public methods of a class. This is a simple counting metric that does not take into account the size or complexity of methods. Regardless, it can indicate classes that may have too much responsibility [139].

Number of Inherited Methods (NIM) measures the number of inherited methods, indicating the amount of behavior of the super class that can be reused; however, it does not measure the number of super methods that are actually used by the class.

Number of Methods Overridden (NMO) counts the number of methods that are overridden by a class. If the super method is called, a high number can indicate a high level of specialization. If the super method is not called, it may indicate that the class or the super class is poorly designed because the class is redefining its parent's behavior, not refining it.

Specialization Index (SIX) indicates the amount of specialization of a class by calculating the ratio of the number of overridden methods to the total number of methods in a class. The result is normalized based on the depth of its inheritance tree DIT.

$$SIX = \frac{NMO \times DIT}{NM + NIM}$$

This metric has not been validated empirically and has been criticized for being inaccurate [99]. It can nevertheless be used to help identify classes with potential problems that may need investigation.

4.4. Quality Conventions and Principles

SOLID

One of the major concerns in object oriented design is responding to change and managing dependencies between modules, referred to by Martin as the dependency architecture [92]. He developed principles of object oriented class design (SOLID) to address these issues, which when used correctly, can positively affect the internal qualities of software. Many of these principles are applicable at both a class and component level.

The five principles of OO class design are the Single Responsibility Principle (SRP), the Open Closed Principle (OCP), the Liskov Substitution Principle (LSP), the Interface Segregation Principle (ISP) and the Dependency Inversion Principle (DIP) [92].

SINGLE RESPONSIBILITY PRINCIPLE states that “a class should have only one reason to change.” If a class has several responsibilities, they can easily become internally coupled and change to one responsibility may undermine the class’ ability to perform the others. This principle is closely related to cohesion and classes that do not respect this principle could be spotted with the various cohesion metrics, such as LCOM.

OPEN CLOSED PRINCIPLE states that “software entities should be open for extension, but closed for modification.” Modules are open for extension if additional functionality can be added to it, and closed for modification if it is available for use by other modules but not directly modifiable [96]. This is often achieved by adding a layer of abstraction [92] to the parts of an application that are expected to change. By respecting this principle changes to requirements are addressed by adding new code and not by modifying existing code, thus limiting regressions and the number of classes that need to be changed. This improves the flexibility, reusability and maintainability of a program. This principle is the at the heart of the DMS metric. The Lorenz and Kidd inheritance metrics could also be used to assist developers in finding classes that do not respect this principle.

LISKOV SUBSTITUTION PRINCIPLE states that “subtypes must be substitutable for their base types.” This means that any client that uses a class must be able to use any derivatives of that class and still function correctly. A sign that this principle has been violated is when functions check the type of the object received and apply varying logic depending on the type. In more extreme cases a subclass can cause methods to break. By having subtypes that are transparently substitutable for their base types, the abstraction that allows OCP to work can be counted on. This reliability is essential for properly functioning object oriented design and is one of the building blocks of OO quality. There are not any metrics to evaluate this principle, but unit testing can help find violating classes. Classes should be able to pass all the unit tests for all of its parent classes, and any failures would indicate a violation of LSP [64].

INTERFACE SEGREGATION PRINCIPLE “deals with the disadvantages of fat interfaces.” Interfaces that contain multiple sets of methods that are not used by the same classes have low cohesion; some classes use some of the methods and others use a different set of methods. Even so, they all must implement all the methods. This can lead to empty implementations in the classes or default implementations in a super class causing LSP violations and increased coupling. ISP states that these interfaces should be broken up into several cohesive interfaces to be used by specific clients. A class can then choose to implement one or several of the interfaces as needed. ISP effectively reduces coupling between clients that would otherwise be tied together by a “fat” interface. Several metrics for determining interface cohesion have been developed [10] although not discussed in this report. The gist of these metrics is to measure the ratio of the number of methods used by clients to the total number of methods in the interface.

Dependency Inversion Principle states that

A) High-level modules should not depend on low-level modules. Both should depend on abstractions, and
B) Abstractions should not depend upon details. Details should depend upon abstractions.

High-level modules contain the policies and business aspects of an application and should not depend on the lower level implementation details. If they do, then any changes to the lowest levels of a program will have repercussions that affect the high-level modules. This would make the high-level modules context dependent and difficult to reuse. DIP reverses this trend by having modules depend on abstractions of their dependencies in order to separate the implementation details from the higher level Application Program Interface (API). By depending on abstractions, the lower level modules can be modified or even switched out with minimal consequences. Respecting DIP makes programs easier to change and more robust. A possible metric to help find violations of DIP is to calculate a ratio of the number of dependencies on abstractions to the total number of dependencies [64].

Applying these principles to object oriented development can improve the internal quality by reducing coupling, increasing cohesion and improving both maintainability and usability. Some of the traditional and object oriented metrics can be used to detect violations of these principles; however, the best guide is experience. To ensure the internal qualities of software, care must be taken when designing and writing code. The technical conception phase should foresee the use of these principles and the judicious application of design patterns and refactoring during the development process can improve the overall quality of software.

Coding and Design Conventions

Programs must be written for people to read, and only incidentally for machines to execute. –
Harold Abelson, Structure and Interpretation of Computer Programs [2]

Some of the internal qualities of code can be measured by using metrics, but there are also non quantitatively measurable quality factors that improve maintainability and reduce software defects. Certain rules and coding conventions can be applied to harmonize source code. Most software is the work of numerous individuals and sections of code are often manipulated by several developers [133]. When a team of developers all use a common set of coding and design conventions the overall code style is consistent, improving readability and understandability, thus making maintenance easier. Coding conventions can also help eliminate certain bugs that come from detectable coding mistakes and misreading code.

Coding conventions define certain practices such as naming patterns for variables, methods, classes and packages. These standards assist developers in finding the information they are searching for. Other syntactic conventions such as the use of whitespace, the way conditional blocks are formatted and the placement of braces help reduce difficult to spot errors. A security vulnerability in a version of Apple's iOS affecting their handling of the SSL/TLS protocol was a result of this type of error [130].

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
  ... other checks ...
fail:
  ... buffer frees (cleanups) ...
return err;
```

In this example, the `if` block does not contain braces and so if the condition is true, `goto fail` is executed, but if the condition is false, the next `goto fail` is executed. This type of bug is difficult to spot if developers do not have a consistent convention for conditional blocks. This bug is arguably easier to find when `if` blocks are always surrounded by braces.

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0){
  goto fail;
}
goto fail;
... other checks ...
fail:
... buffer frees (cleanups) ...
return err;
```

Programmatic conventions can also be defined to limit the maximum number of lines in a method or of nested conditions, improving certain quality characteristics. By limiting the number of lines in a method, cyclomatic complexity can be reduced and testability improves.

Architectural conventions help developers respect the overall design of software. Tiered architectures often have rules defining which tiers can communicate with each other, and disrespecting these rules can contribute the erosion of the architecture. Commonly used design patterns can be integrated into the conventions so that different developers use similar strategies to solve common problems. This leads to a more cohesive and understandable code base.

Several tools are available to detect code that does not respect the defined conventions such as PMD[7] and Checkstyle[8] for Java. These verification tools are often called linters. A lint "... goes through the code and picks out all the fuzz that makes programs messy and error-prone" [88]. Coding rules can also be defined in Integrated Development Environments (IDEs) such as Eclipse and automatically be applied to new and existing code.

Technical Debt

Code can be carefully designed to maximize the internal qualities of code or it can be hacked together in order to quickly deliver new functionality. Fast and dirty may seem efficient at first but may lead to future delays because it is not very maintainable. This is technical debt. Ward Cunningham developed this metaphor which compares financial debt to the delays and complications that result in poor coding practices [83].

> *Neglecting the design is like borrowing money*
> *Refactoring, it's like paying off the principal debt*
> *Developing slower because of this debt is like paying interest on the loan.*
> *Every minute spent on not-quite-right code counts as interest*

When financial debt is taken on interest must be paid, increasing the total cost. The same is true for technical debt. Time can be spent improving code quality in order to pay back some of the principal otherwise the debt keeps growing. As with financial debt, however, it can be judicious to take on some debt in order to deliver a product more quickly. Care must be taken because if the debt is not paid back, production can grind to a halt [55].

SQALE

A quality method has been developed to amalgamate the results of various metrics suites and provide an overview of the quality of software in the quantifiable form of technical debt. The Software Quality Assessment based on Life Cycle Expectations (SQALE) method calculates the time needed to repair all quality issues. These issues are based on user defined rules, which are a calculated using metrics. Metrics evaluate specific aspects of a code base and provide numeric values as a result; rules determine whether these values are good or bad based on defined parameters. The remediation cost is then calculated by multiplying the number of violations by a weighted parameter that estimates the time needed to fix the violation [68].

The Software Quality Assessment based on Life Cycle Expectations (SQALE) method's model is based on the previous quality models - McCall, ISO 9126 and SQuaRE, but provides "rules" to quantify quality characteristics. In the ISO 9126 model testability and changeability are sub-characteristics of maintainability, but in SQALE they are upgraded to first level characteristics because of their importance in the development life cycle. The model defines four concepts: the quality model, the analysis model, indices and indicators [84].

[7]`https://pmd.github.io/`

[8]`http://checkstyle.sourceforge.net/`

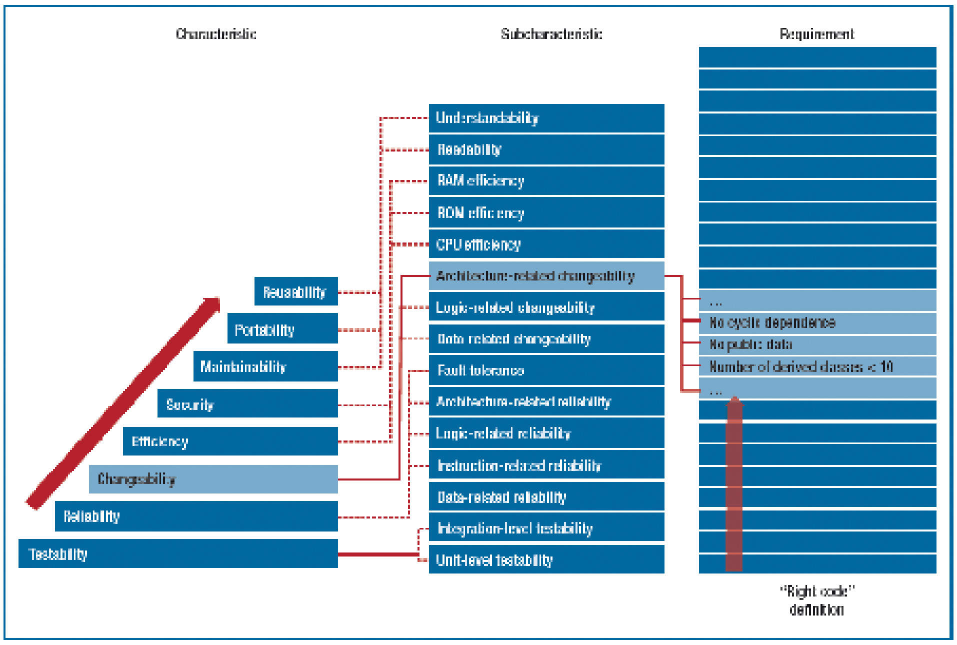


Figure 4.9: SQALE Quality Model [84]

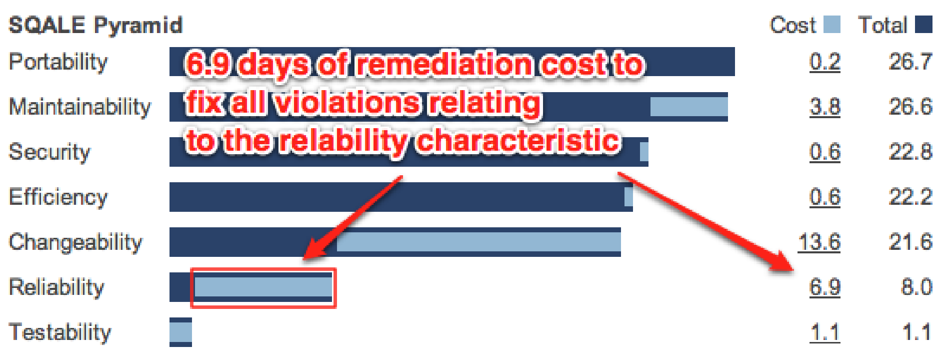
The Quality Model (Figure 4.9) defines internal quality factors that are organized in three levels: characteristics, sub-characteristics and requirements. The characteristics are Testability, Reliability, Changeability, Efficiency, Security, Maintainability, Portability and Reusability. These are broken down into sub-characteristics which are further refined into requirements. Sub-characteristics refine the top level characteristics and requirements are the rules that map to a quality metric. SQALE specifies that sub-characteristics and requirements cannot be repeated and are uniquely linked to a single quality characteristic. The sub-characteristics and requirements are user defined and reflect a project or an organization's definition of "right code." This normally takes into account coding conventions, software metrics, and architectural and design considerations. An example of this hierarchy can be seen in Table 4.1.

The characteristics are organized in a pyramid with the most important factor, testability, at the bottom. This pyramid is upside-down because the lower levels are viewed as the foundation for subsequent characteristics. As illustrated in figure 4.10, each characteristic builds upon the ones below it. The cost of Reliability is 6.9 with the total of 8.0 which takes into account the cost of Testability. If the code is not testable because it is too complex or coupled, other factors will be difficult to improve because any modifications to the code will not be tested to ensure that regressions were not introduced [131]. Technical debt should be paid back from the bottom up.

The Analysis Model normalizes the derived values using a remediation index that is applied to each software component. This index is the price that needs to be paid to correct all non-compliances. These costs are calculated based on user defined "remediation functions." The overall cost of non-compliance is

Characteristic	Sub-Characteristic	Requirement
Maintainability	Readability	There is no commented out block of instruction
Changeability	Architecture related Changeability	There is no cyclic dependency between packages
Reliability	Coverage related Reliability	All files have unit testing with at least 70% code coverage
Testability	Unit test level Testability	There is no method with a cyclomatic complexity over 12

Table 4.1: SQALE Characteristics [84]



	Cost	Total
Portability	0.2	26.7
Maintainability	3.8	26.6
Security	0.6	22.8
Efficiency	0.6	22.2
Changeability	13.6	21.6
Reliability	6.9	8.0
Testability	1.1	1.1

Figure 4.10: SonaQube SQALE pyramid [129]

Requirement	Remediation Details	Remediation Function
Code shall override both equals and hashcode	Write code and associated test	1h per occurrence
All files have at least 70% code coverage	Write additional tests	20 mn per uncovered line to achieve 70%

Table 4.2: SQALE Remediation Functions [84]

calculated by summing the debt acquired by each violation. An example of the relationship of requirements to remediation functions can be seen in Table 4.2

The Indices include an index for each of the quality characteristics. These unit level indices are calculated by summing the remediation costs related to each characteristic. The most important index is the global quality index which is a sum of the other indices and represents the total technical debt of a project.

The Indicators can be used for project analysis. The “rating indicator” is a ratio between technical debt and overall development costs based on a defined cost per LOC. Each characteristic and the overall project are rated. The SQALE pyramid graphically represents the technical debt for each characteristic and the total debt acquired.

The SQALE method can highlight problem areas in a project and help organizations develop a plan of action to overcome them. Refactoring and code improvement strategies can be developed in accordance with the “remediation priorities” defined in the method. Tools such as SonarQube amalgamate code conventions, metrics and technical debt for many programming languages. These tools provides invaluable feedback to developers and managers in their quest for improving the quality of their software (Figure 4.10). In this illustration the remediation cost of each quality characteristic can be seen with the total technical debt for each level of the pyramid and for the project as a whole.

***

Evaluating software quality is a complex domain fraught with uncertainties and ambiguous terms. Much work has been done to create formal definitions of quality in general industrial contexts. The aim of this work is to reduce production and maintenance costs and to improve product quality and customer satisfaction. The quality definitions of industry have been modified and refined for use in defining software quality and formalized through the construction of quality models.

Software quality is defined in terms of functionality and satisfying requirements. The internal non-functional qualities of code play an important role in assuring high quality software from an external and functional point of view.

Once formal definitions of quality were established, code metrics were applied to quantitatively measure and evaluate quality characteristics. As Galileo Galilei said in the 16th century [58]: “Measure what is measurable, and make measurable what is not so,” and metrics are a way to quantitatively measure quality. The use of metrics was integrated into the first models of McCall and Boehm [112, 26] and new and improved metrics continue to be created in response to new programming paradigms and environments. Metrics allow developers and managers to evaluate software in an automated and efficient way because they provide concrete measurements that can be reported and acted upon. The cost of repairing defects increases significantly as the project progresses and can be up to 100 times more expensive after a product has been shipped [24]. By using metrics throughout the development process the quality feedback loop can be shortened, consequently reducing costs.

The numbers produced by metrics, in an of themselves, do not guarantee quality, but can help highlight problem areas in a code base that may require further investigation. Metrics need to be carefully chosen by project leaders to appraise the aspects of code that they consider pertinent to their project. Unfortunately, there is no one size fits all solution and metrics need to be tailored to individual applications. Choosing combinations of metrics that are appropriate to a project and defining rules to evaluate them is a complex task. Sets of default standards have been defined in software analysis tools but must be adjusted to each project.

Once a set of metrics has been chosen and thresholds defined, they should be used to flag areas of code that need improvement, not just provide an overall value. It is more important to the project team to know which modules are deficient than to simply know that the overall code base has a certain percentage of test coverage. An important aspect of evaluating the results is following the changes over time. During the development process, programmers and management can continually evaluate the code base in order to progressively improve the overall quality. Certain functional metrics, such as the number of reported bugs and the results of functional tests, can be integrated into this process to help evaluate the external product quality. Metrics can assist teams in applying a continual improvement process in order to deliver a high quality product.

Metrics can be very beneficial to projects and assist in improving the quality characteristics of software; nevertheless, they should not be seen as a panacea. They need to be used judiciously because they can potentially have adverse effects on quality. When the product team is judged on metric values, they tend to adapt their behavior to improve the metric. For example, software testers that are judged by the number of bugs they find have a tendency to find more bugs. These testers are inclined to find superficial and easy to find bugs because the hard to find, but more critical bugs take too much time to discover [87]. A similar effect can be seen when developers are evaluated on the number of lines of code they write. This promotes code duplication, poor programming practices and degrades quality. Nonetheless, prudent use of metrics can greatly assist project teams in improving code quality.

Integrating quality models, high caliber design and coding principles and software evaluation metrics into the development process can aid in improving the internal qualities of software. This, in turn, influences the overall external qualities as seen by the end-users. Although a high degree of internal quality does not guarantee that the software answers the needs of users, it can significantly reduce the amount of defects in the shipped product and help mitigate production and maintenance costs. Metrics are a valuable tool available to software development teams that can positively influence software code quality and can be integrated into their definition of “done.”

Learning the intricacies of applying and using metrics and coding principles is a complex process. In an Agile team different members have different areas of expertise and the entire team can benefit through the sharing of everyone’s knowledge.

Knowledge Creation and Learning

5

5.1. Spiral of Knowledge

An integral part of improving developer awareness of the importance of code quality is through communication and expressing ideas to the outside world [17]. The act of learning, integrating knowledge and expressing it was referred to by Nonaka as the “Spiral of Knowledge” (Figure 5.1) which is composed of four principle steps: socialization, articulation, combination and internalization. The spiral of knowledge continually grows and was conceived as a spiral and not a circle to highlight this continuous amplification of knowledge [106]. This spiral is based on two types of knowledge – tacit and explicit [105].

> *Tacit knowledge involves intangible factors embedded in personal beliefs, experiences, and values. Explicit knowledge is systematic and easily communicated in the form of hard data or codified procedures* [70].

Tacit knowledge is gained from experience and can be seen as deep-rooted mental models and a point of view that shapes the way one interacts with the world. Tacit knowledge is the expertise of master craftsmen. By spending years honing their craft and working with masters, these craftsmen have an innate sense of what to do in different situations in order to get the best results. Bearers of this type of knowledge often have trouble transmitting their expertise because it is deeply ingrained and thus difficult to formulate in an explicit form. Explicit knowledge is “formal and systematic” [105] and is therefore easily documented and shared. The Spiral of Knowledge is the way that knowledge can be transformed from the intangible know-how of the expert craftsman to codifiable knowledge that can be understood and used by others. This knowledge is then combined with existing knowledge and internalized through hands on experience, thus completing the cycle.

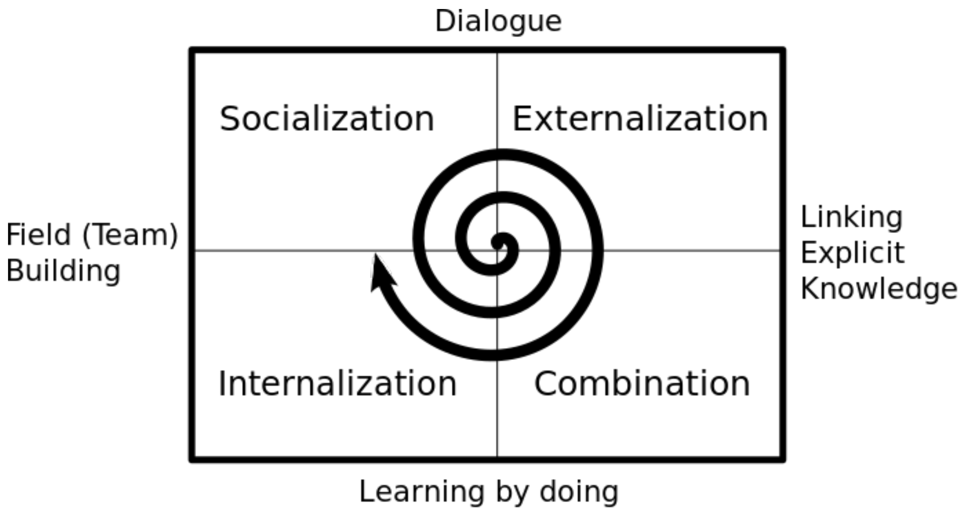


Figure 5.1: The Spiral of Knowledge [146]

This cycle navigates through various transformations: socialization, externalization, combination and internalization [105].

Socialization is the way individuals transmit tacit knowledge to others. This tacit to tacit transfer of knowledge is achieved through apprenticeship and close interaction between master and student. Technical skills are passed through imitation and practice.

Externalization is the transformation of tacit knowledge to explicit knowledge that can be shared and easily communicated to others. The process of articulating tacit knowledge is complex because it is an attempt to express something intangible. Through dialog and interaction with others, ethereal tacit knowledge can slowly become more concrete. A technique proposed by Nonaka is to move through a process of metaphor, analogy and model where ideas become increasingly more explicit. Metaphor creates a bridge between different ideas and opposing concepts. Analogy takes this one step further creating a link between logic and imagination and helps resolve contradictions between different world views. Model is a way to completely resolve these conflicts and formalize information.

Combination is the amalgamation of explicit knowledge with existing explicit knowledge in order to create more complex and comprehensive knowledge. This step relies on rationalism and logic to combine disparate knowledge into a cohesive whole [106]. Combination is often a group activity bringing together various areas of expertise and points of view.

Internalization is the integration of new knowledge by an individual through the transformation of explicit knowledge to tacit knowledge. The internalization of information is a crucial step in the retention of information and is achieved through hands on experience, action, practice and reflection. When knowledge is synthesized it can spark new ideas and the creation of new knowledge.

Nonaka asserts that redundancy is an important aspect in encouraging communication between collaborators which in turn helps the transfer of tacit knowledge and the spreading of explicit knowledge [105]. Rotating employees in different roles allows them to adopt different points of view of the same project and helps expand their horizons and knowledge base. Transparency is one of the keystones of Scrum and free access to information also helps develop redundancy within the team.

The Spiral of Knowledge provides a framework for knowledge creation and learning within an organization, and as such, does not dictate specific strategies for each step of the process. In the context of the Scrum methodology, centered around the development team with a flat hierarchical structure, traditional didactic methods are not appropriate. Within this environment, Experiential Learning and critical thinking are important aspects of the creation and internalization of knowledge.

5.2. Experiential Learning

Kolb's Experiential Learning Theory (ELT) [81] is based on the principle that:

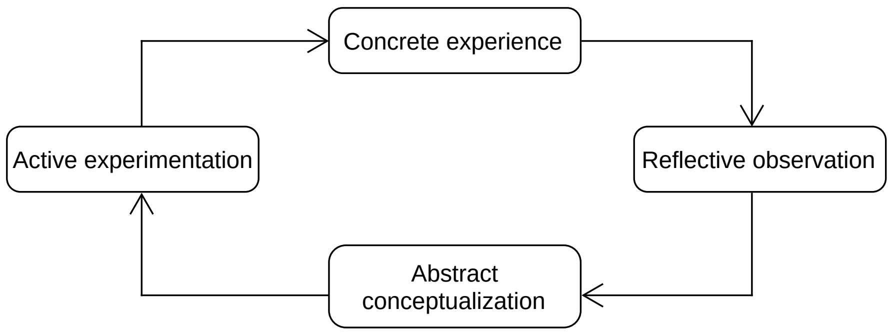


Figure 5.2: Kolb Experiential Learning Cycle [98]

Ideas are not fixed and immutable elements of thought but are formed and reformed through experience.

Experiential learning is an iterative process that involves concrete experiences, observation and reflection on those experiences, the creation of abstract concepts and then testing these new concepts (Figure 5.2) [81]. This process of learning is based on frequent feedback and continuous experimentation based on the new data.

Concrete experience It is necessary to actively engage in an activity to learn effectively. Passively learning through reading or observing is not sufficient.

Reflective observation Doing is important, but thinking analytically about the experience and evaluating what was learned is a crucial part of this cycle. Communicating with others and externalizing the experience can assist in synthesizing the experience.

Abstract conceptualisation After reflecting on experience, new models or ways of approaching the problem can be conceived. By thinking about what happened and creating hypotheses, new concepts and ideas can be formulated which enhance the subsequent round of experimentation.

Active experimentation The learner tests his hypotheses through experimentation. Learning is a process that is constructed more than transmitted and through the act of experimentation and adjustment, learning occurs.

Kolb's cycle of learning has been shown to improve learner retention compared to traditional didactic methods [144]. The act of doing and reflecting on previous actions reinforces learning and the internalization of accumulated experience. Learners are often accompanied by a facilitator who asks pertinent questions about the experience to assist the reflective and conceptualization phases. This often follows the Socratic method of asking questions and inciting critical thinking. Through this process knowledge is created by each individual "through the transformation of experience" [81] and not simply handed down from teacher to pupil.

Learners can also benefit from sharing their experiences with others. This narrative communication allows individuals to organize, interpret and give meaning to their experiences [42]. Teaching to other is an effective means of breaking down complex ideas and is a technique that can used to convert tacit knowledge into explicit knowledge. As an added benefit, the entire group can learn through shared narratives which can open the door to discussions. By seeing differing points of view, everybody benefits.

II

Collaborative Approach to Quality

6 The Problem of Ensuring Code Quality

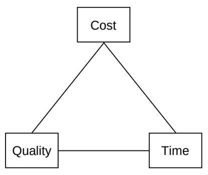


Figure 6.1: Iron Triangle [11]

Cost, time and quality are the three variables in software development that can be modulated depending on project priorities and is referred to as the Iron Triangle (Figure 6.1) [11]. In this model the priority of each variable can be set but it is impossible to have all three, therefore a balance must be found. This provides a way of controlling the outcome of the project by managing time, costs and quality. Spending more money and increasing costs makes it possible to reduce the time necessary to complete the project and to improve quality; however, the gains tend to be limited.

Not all tasks can be executed in parallel and merely adding resources does not necessarily result in shorter cycles and better quality. According to Frederick Brooks, adding resources actually increases development time [29] because new developers need to be trained. This training takes time away from experienced team members, thus slowing the development.

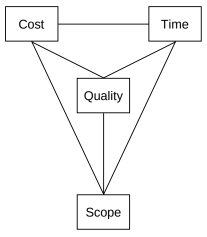


Figure 6.2: Modified Iron Triangle

By prolonging the development time, costs can be reduced without reducing quality. This constraint is often out of the control of the production team and is therefore difficult to modulate. Reducing quality can lessen overall cost and time necessary to complete a project. Degrading quality criteria can provide short term benefits but can have devastating repercussions for the project. Internal quality can be sacrificed for short term gains in development speed and cost; however, by lowering code quality and increasing technical debt (subsection 4.4) can lead to severe problems maintaining and evolving the product. Lower quality can also have a negative effect on the team's morale [17]. Finding the correct balance of these three variables is an important consideration in project management.

Kent Beck considers a fourth variable, scope, and argues that modulating it results in the most value [17] (Figure 6.2). He argues that by reducing the scope of a project, the constraints of time, cost and quality can be maintained. Since requirements are constantly evolving, modulating the scope by prioritizing and releasing the product in increments to get regular feedback from the customer, time, cost and quality can be better managed.

6.1. Code Quality Within Scrum

In the context of Agile methods, notably Scrum, the time allocated to each sprint is determined in advance and the amount of resources is set by management; however, the scope of each sprint is determined by the team. The one variable that is not systematically handled is quality – notably the internal qualities of code. This quality can be integrated into the communal definition of "done," but this is not always the case. The external or functional qualities of software are managed through functional testing by the team and validation by the client. In order to meet the deadlines of each sprint, the internal qualities of code can fall by the wayside. Scrum, a project management framework, does not explicitly handle code quality. Quality and other engineering techniques are left up to the team. Code quality can slowly degrade due to a lack of expertise in the development team, the difficulty of using code metrics effectively, the incomplete nature of commonly used metric suits and the mental burden of integrating the complex and technical notions of code quality.

One of the prerequisites of Agile methods is a highly competent and technically apt team, and if this requirement is not met, code quality often falls victim. Code quality is a complex and subjective notion closely linked to design and architectural decisions which demand expertise and experience.

In order to measure code quality in an objective manner, certain criteria have been developed through quality models, software metrics and the notion of technical debt (subsection 4.4). These tools and techniques can help guide developers to discover problem areas in the code, but they are often applied retroactively or cursively analyzed. Another shortcoming of these metrics is the difficulty in extracting meaningful information from the profusion of data that is generated from a static code analysis. Reading metric results and finding acceptable solutions to the problems that may be hidden behind the numbers is an intricate problem that requires experience and training. Metrics indicate troubled areas in code, but do not supply techniques or methods to pin point the problem or provide ways of resolving them. Without adequate knowledge and experience in this domain, metrics will not help developers to improve code quality. The preoccupations of a Scrum team of delivering a usable product in a timely fashion may not always be in consonance with investing time and resources into improving code quality.

Metrics themselves have a long history in software development and new metrics are continually being developed to overcome the shortcomings of their predecessors. Existing metrics are effective in measuring certain aspects of code quality; however, many aspects of code quality, such as cohesion and architectural quality, are difficult to measure and adequate metrics do not exist. Although difficult to measure, certain design and architectural principles, such as SOLID (subsection 4.4) for Object Oriented programming, have been developed to help developers design software that exhibits quality attributes, but again the results are difficult to measure and require experience to evaluate and execute effectively. On the other hand, some aspects of code quality such as coding conventions (subsection 4.4) are simple to measure and many tools exist to evaluate source code based on simple to define rules. The difficulty faced by development teams is to develop the expertise necessary to evaluate both the measurable aspects and more complex notions of quality that are based on principles but executed through intuition.

6.2. Contribution

The act of analyzing metrics and evaluating the results is often done in a haphazard manner leading to degraded code quality through neglect and the natural force of entropy [37]. The activity of evaluating and improving code quality is often not a central activity in the development process. It is done on an individual basis and depends on the competencies and predilections of each developer.

The improvement of the internal qualities of code must be accomplished through a collaborative approach to quality. In Scrum, the internal qualities of each delivered functionality are not explicitly handled, but could become an integral part of the definition of "done." In order to guarantee an acceptable level of code quality in a Scrum team, the entire team needs to work together with a common goal of improving code quality. As the team is self-organizing and cross-discipline, members must work together to accomplish this goal.

Analyzing metrics is complex and open to interpretation. Not all the data generated is useful and each result must be evaluated. This can be seen as uninteresting and laborious; however, the evaluation of metrics and applying appropriate solutions should become a group activity that promotes shared learning, thus making this task educative and more rewarding for the developers.

Agile methods are centered around individuals and their competencies, but even if the team is not made up exclusively of expert developers, the desire to learn and improve through doing is an essential criterion of success. Barry Boehm remarked in an article about Agile methods, that there is an "unavoidable statistic that 49.9999 percent of the world's software developers are below average" [21]. Composing a Scrum team of only expert developers is a difficult task; nevertheless, even if at the outset the team members are not all experts, by working together and exploring new realms and disciplines, the breadth and depth of knowledge of the team increases. Applying principles of Knowledge Creation theories and Experiential Learning with a focus on code quality can elevate even a mediocre team to a level where they can consistently create functional code that is reliable, usable, efficient, maintainable and modifiable.

To reach this level, the development team must have tools to measure the quality of the existing code and be able to follow its evolution over time. Each developer should have a global view of the internal qualities of the product as well as a window into their own performance. This provides a transparent way for the team to evaluate the current state of affairs and create adaptive strategies to maximize the quality of code by applying the core values of Scrum – transparency, inspection and adaptation – to code quality. By raising awareness of the benefits and added value that code quality provides and providing a framework for knowledge creation and an environment conducive to learning, the team integrates these aspects into their daily work flow and avoids treating quality as a secondary concern. The internal quality of software thus becomes a central part of the development process. Controls and events around this theme are integrated into development sprints. These additions to the Scrum framework are based on Nonaka's theory of Knowledge Creation and Kolb's theory of Experiential Learning combined with engineering techniques inspired by Extreme Programming and the Software Craftsmanship movement.

7 Quality Framework

The proposed solution to improve code quality is based on a collaborative approach. Scrum provides a general framework of teamwork and cooperation and this quality framework builds on that of Scrum. The roles and ceremonies defined in the Scrum framework focus on transparency, inspection and adaptation in the development cycle which help create a responsive team which continually improves their process. The result of a Sprint is a working increment in "useable condition" [116], which implies that the functional requirements are satisfied and the external qualities are at an acceptable level; nevertheless, aspects of internal quality are not explicitly considered (Figure 7.1). Criteria for the acceptable level of internal quality of delivered functionalities can be integrated into the team's definition of "done," but this depends exclusively on the proclivities of the team.

To overcome this shortcoming, the Collaborative Model for Code Quality has been developed and integrated into the existing Scrum framework. The new role of Quality Master and certain Quality Events are introduced to manage the internal qualities of code throughout the development cycle. This incites awareness of code quality in the team and provides a mechanism to promote the sharing of knowledge. A support system has been developed to provide the tools necessary to cover the new needs of the development team in their quest to improve code quality through a collaborative effort. This support system extends the traditional software factory to include aspects of code quality measurement and evaluation as well as support for a co-operative knowledge creation process.

Agile methodologies are people centric and to successfully integrate code quality into an Agile development process, the team of developers must be aware of the benefits of code quality and be directly involved in its improvement. All Scrum teams are not made up of expert developers with an innate sense of code quality; therefore, a key aspect of improving code quality is having developers who are open minded and eager to learn and share their knowledge with other members of the team. Each developer must make a concerted effort to assist others where possible and be willing to accept help from their colleagues. A prerequisite of a healthy team and code base is openness and communication within the team. The proposed solution presupposes

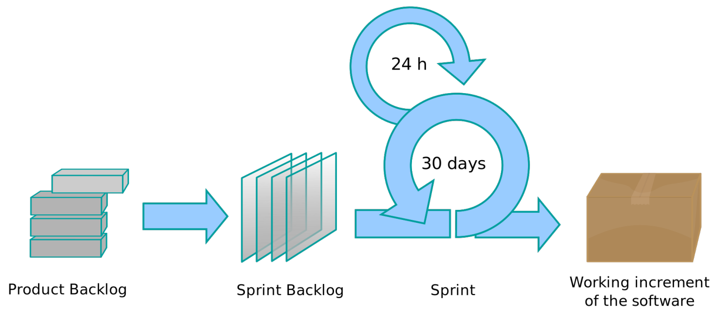


Figure 7.1: Scrum Process Overview[150]

a development team that follows a Scrum methodology and a desire to continually improve code quality.

In order to continually evaluate and improve code quality it is necessary for the development team to have regular feedback on the quality of code, both through metrics and interpersonal feedback. The overall quality of code is important, but the changes in quality over time can both motivate developers and assist them in integrating different concepts and techniques related to code quality. Creating, sharing and refining knowledge is an essential aspect of this process and this solution is based on certain principles derived from Knowledge Creation theory [106] and Experiential Learning [80].

The holistic method of Nonaka highlights the importance of autonomy, self-transcendence and cross-fertilization within the development team [135]. The Collaborative Model for Code Quality(Figure 7.2), which draws from Nonaka's Spiral of Knowledge (section 5.1), can guide organizational and team knowledge creation and management in order to improve their expertise. A new role in Scrum is introduced to guide the team through the various stages of this model. Facets of Experiential Learning are applied to different phases to help the team create and share new knowledge through experience, communication and collaboratively working towards the common goal of improving code quality. This process incites autodidacticism, knowledge discovery, awareness of code quality and the modification of the developers' coding practices.

7.1. Collaborative Model for Code Quality

Self-organizing Scrum teams are responsible for establishing a shared definition of "done." As development teams mature, aspects of quality are expected to be integrated into this definition [116]; however, this responsibility is left to the team. By making code quality a central component of each iteration, the definition of "done"

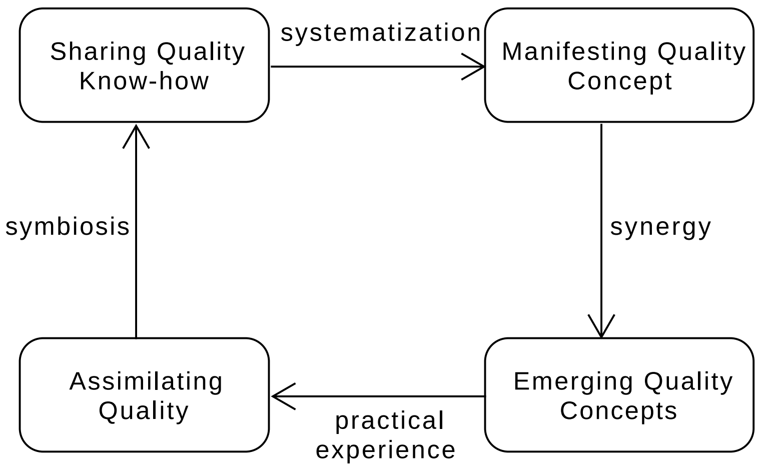


Figure 7.2: Collaborative Model for Code Quality

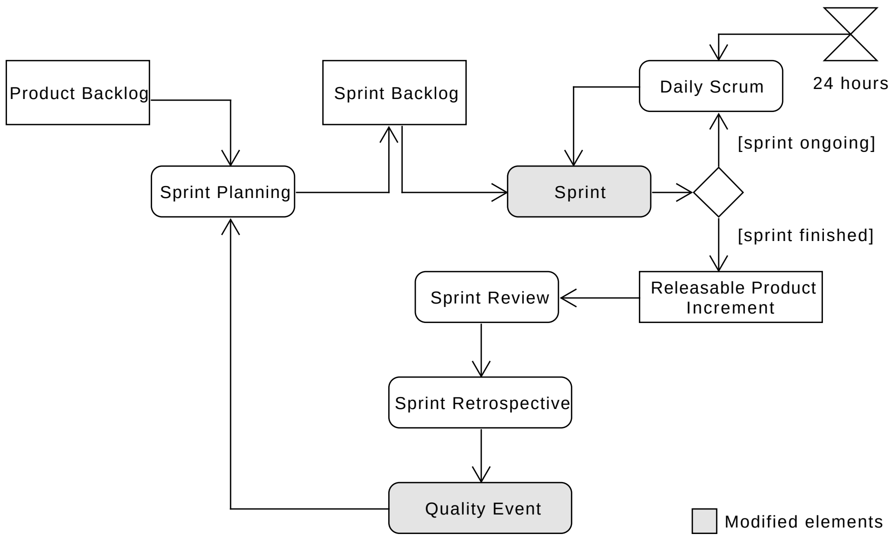


Figure 7.3: Modified Scrum Process

must evolve to include quality criteria. The criteria used to measure code quality is determined by the development team under the guidance of the Quality Master. As the team becomes more experienced and competent, this definition evolves to become more stringent. Using metrics, measurable criteria can be established and controlled using monitoring and analysis tools. By implementing collaborative coding practices, the harder to measure properties of code quality can be evaluated.

The definition of "done" should also contain strict coding standards. These standards comprise basic formatting and code structure guidelines as well as certain coding principles such as the DRY principle to avoid duplication and rules regarding cyclomatic complexity or other key metrics (section 4.4). Certain aspects of code quality that are difficult to measure with metrics can only be handled through experience, education and a communal understanding of these difficulties. The application of the Collaborative Quality Model leads the team to a collective awareness of these issues allowing them to better overcome the inherent difficulties in detecting and correcting these problems. Having a homogeneous code and knowledge base incites collective ownership of the code and thus group responsibility for the quality of the application [17].

Each developer has unique experiences, perspectives and expertise. They each have a wealth of tacit knowledge that must be transformed into explicit knowledge that can be shared with the group and assimilated by each member. To encourage the self-transcendent nature of the team and to and facilitate cross-fertilization, objectives are set, certain events and techniques are applied and controls instituted to help move the team through the phases of the Collaborative Model for Code Quality and take place at different moments in the Scrum framework.

Sharing Quality Know-how

The knowledge that each developer has internalized through study, professional experience and hands on training is deeply ingrained and often difficult for the developer to express. *Socialization* is the first step the Spiral of Knowledge (section 5.1) which facilitates the transmission of this tacit knowledge from one individual to another. The diverse areas of expertise of different developers provides a solid starting point for this process. Sharing quality know-how is the first step in transmitting tacit knowledge.

Communication is a key factor in transmitting knowledge; working closely together by doing code reviews and pair programming can facilitate knowledge transfer. These activities are also effective ways of adding redundancy to the team. Not all developers are familiar with the entire application and pair programming and code reviews can be an powerful and interactive way of learning and sharing.

Code reviews and pair programming are similar activities. Pair programming is XP's answer to code reviews: "if code reviews are good, we'll review code all the time" [17]. As such, these two techniques provide similar benefits. These activities force face to face communication, discussion on many levels and analysis of the problem at hand. In pair programming, by working as a duo, each member can focus on different aspects of coding – one focuses on implementation while other thinks more strategically. This activity encourages communication and incites knowledge creation. As an added benefit the quality of the production is also higher when developers work together [17], defects are reduced and team morale improves [39]. In contrast to pair programming where review is an ongoing process, code reviews are executed after coding has been finished. Two developers work together to analyze freshly written code and its integration into the application. Both techniques incite communication and the creation of knowledge, but through different modes of execution. These activities take place on a daily basis and are an integral part of each Sprint.

The apprenticeship model is another way of transferring tacit knowledge. Coding sessions can be proposed and organized for individual developers or groups. Instead of peers working together, a tutor or expert of a certain domain, can demonstrate skills and help others learn. This top down approach is less appropriate with a Scrum team than peer reviews or pair programming, but since developers have varying areas of expertise, they can take turns being the expert. Sharing knowledge and know-how within the team benefits everyone including the teacher. Working with someone more experienced is beneficial for the "apprentice" allowing the transfer of tacit knowledge. The act of teaching or preparing a workshop is also helpful for the mentor; teaching is an effective way of externalizing tacit knowledge. These workshops can be scheduled on a weekly or per Sprint basis and are one of the essential Quality Events within this framework.

Manifesting Quality Concepts

Transforming tacit knowledge to explicit knowledge is done through the process of *externalization*. Thinking rationally about ingrained knowledge is a complex task and some techniques that can be used are preparing and leading workshops, teaching, writing articles and group discussion. The manifestation of quality concepts is done through the transformation of ideas acquired during the previous step into exploitable and concrete notions through the systemization of knowledge.

According to a study in the journal *Memory and Cognition* the act of preparing to teach a skill to someone can enhance the learning process [103]. This study showed that when students expect to teach what they are learning, they “engage in more effective learning strategies.” This fact can be exploited by having developers regularly present what they have learned to the rest of the team. Breaking down knowledge into steps that can be followed by students is a rational process that can help formalize and systematize tacit knowledge. The act of preparing a lesson also forces individuals to think logically about their knowledge and present it in an explicit and straightforward way.

Expressing knowledge in written form is another way of teasing explicit knowledge out of tacit know-how. Writing forces authors to formalize their learning in a logical and structured manner. This effort makes them think about their knowledge from the perspective of someone who does not have the same background and experience as themselves. The act of expressing ideas in written form has the added benefit of documenting the knowledge so it can me more easily shared with others. In the spirit of transparency, this documentation is shared in a communal repository where knowledge can be accumulated, consulted and modified by the entire team.

Another way of elucidating tacit knowledge is through discussion. Sharing what has been learned through narrative communication has been shown to help consolidate ideas in the minds of the learners [42]. By expressing verbally, learners are forced to organize their thoughts. The ensuing discussion with others who have different points of view can also enrich the knowledge acquisition process and lead to the emergence of new knowledge.

By documenting individual and group knowledge, the effort of externalizing knowledge benefits the individuals, the team and the organization as a whole. A repository of accumulated knowledge and practices allows the organization to capitalize on otherwise unexploitable tacit knowledge. This repository is consulted and updated routinely in each Sprint and becomes an integral part of the developer’s toolkit.

As an added benefit, by rendering tacit knowledge explicit, new team members can more quickly integrate the team and become operational. The know-how of the team becomes available to the entire organization.

Emerging Quality Concepts

Group sessions and workshops can be introduced into the development process to aid in combining new knowledge with previously acquired knowledge to create something new. This is referred to as *combination.* New quality concepts can emerge through group discussion and brain storming. Discussion is an important aspect of this phase in order to resolve conflicting points of view and unite the group’s world view. Brain storming sessions to integrate new knowledge into the existing repository can be a way to discover new uses and combinations of knowledge that can be used in the project. Through synergy, existing knowledge can be molded and combined with new knowledge setting the groundwork necessary for the emergence of new ideas. Group discussion also helps the team increase their awareness of quality concepts and of the benefits to be gained through a steady improvement of the overall quality of software. By working together, the group accomplishes something greater than they would individually. This discussion also reinforces the externalization aspects of knowledge creation and leads the team towards incorporating quality into their daily workflow.

Assimilating Quality

The Spiral of Knowledge identifies *internalization* as the process of converting explicit knowledge into tacit knowledge. Explicit knowledge that exists in a repository remains theoretical, but through action, practice and reflection this knowledge can be synthesized and integrated by individuals [106]. Through the act of experimenting, cogitating and conceptualizing, this knowledge can be transformed into new skills and know how [81]. According to Malcolm Gladwell in his book *Outliers*, 10,000 hours of practice are necessary to master a skill [62]. Natural talent only plays a small role in success, whereas practice is the essential activity that leads to success.

In order to give developers the opportunity to assimilate what they have learned through documentation and discussion, hands-on experience and training is essential. Deliberate practice [74] is a technique that can be used to get programmers to focus on learning new development skills and improve quality. In this practice, developers do not work on production code, but focus on exercises that improve specific skills. Through repetition, skills and techniques are developed. Applying the theory of experiential learning through the cycle of experience, reflection, conceptualization and experimentation to such exercises can allow developers to effectively integrate new skills.

"Coding katas," a term coined by Dave Thomas [40], is a technique used in the *Software Craftsmanship* movement to improve skills. Kata is a term from martial arts which refers to "forms" or organized sequences of movements. Martial artists practice these katas in order to internalize techniques so they can execute them later without hesitation or conscious thought. Coding katas follow this theory and give developers the opportunity to experiment and play. Practicing experiential learning in a non critical and unintimidating environment allows developers to integrate coding concepts. The controlled environment provided by coding katas removes the obstacle of having to create working code which provides the freedom necessary for experimentation and the integration of new skills. These new skills can then by applied in the more complex and critical production environment with confidence.

Providing time for practice and organizing katas around the theme of code quality and principles is a way for developers to assimilate "textbook" knowledge and transform it into real skills. Once this know-how has been embodied by the developer, quality becomes a integral part of his reasoning about software design which manifests itself in higher quality code. Through these practices, even hard to measure aspects of code quality can be taken on. This step leads back to the "sharing of know-how," and through symbiosis developers working together both benefit from the knowledge and skills of the other.

These events do not directly provide functionality for the client, but enhance the quality of the application which inherently adds value. High quality leads to shortened maintenance times and facilitates the integration of new functionalities. The time invested in these events pays for itself through increased productivity and a close knit team that works together effectively.

7.2. Organizational Facets

Every Scrum team exists within the fabric of an organization and this cadre plays an important role in guiding the objectives of the team and creating a company culture. Management plays a crucial role in establishing a fertile and supportive environment and their actions can foster knowledge creation and learning in self-organizing and autonomous teams.

Scrum teams operate on the principle that they make important decisions as a group and work towards a common objective; nevertheless, management must oversee the results and establish checkpoints and controls to keep the project on track. Nonaka refers to an appropriate level of oversight as “subtle control” [135]. In this paradigm management is there to set global objectives and standards which guide projects but allow freedom and autonomy. A technique they use to guide the team is carefully choosing its members in order to optimize the group dynamic. Actions by management should minimize the obstacles for the self-organizing team and to incite cooperation and team unity; performance should be evaluated on a group level.

It is ultimately the responsibility of the development team to define their level of accepted quality and to determine ways of measuring it, but without a global objective of quality code, the team will not necessarily give priority to this aspect of production. One way of establishing quality as a project objective and monitoring progress is through the creation of a new role – the Quality Master.

The Quality Master is responsible for overseeing the quality of production and assisting the team in improving the overall quality of the code base. This role is

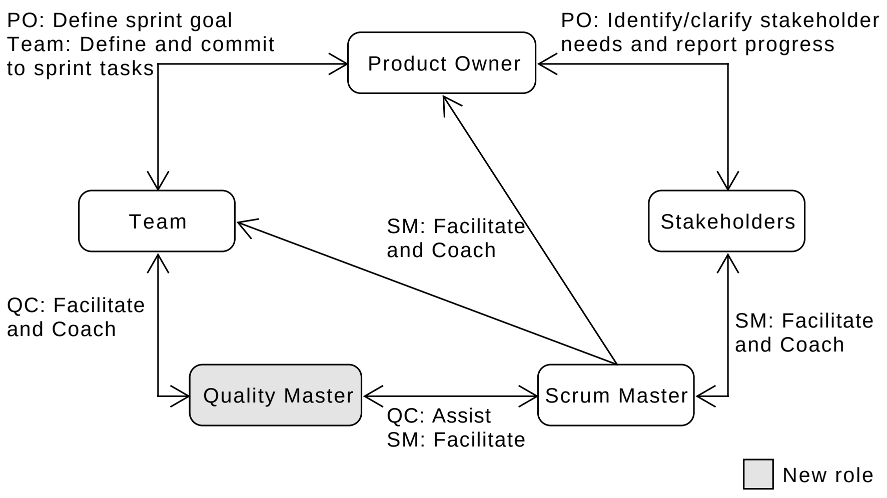


Figure 7.4: Quality Master Role

reminiscent of the Coach and Tracker as defined in Extreme Programming. The XP Coach is often a technical leader, but does not make the important architectural and design decisions for the project, instead he assists the developers in coming up with appropriate solutions. The Tracker monitors progress and evaluates the accuracy of estimations helping the team become more accurate as time goes on [17].

The Quality Master is both a facilitator and mediator. As facilitator, he promotes learning by encouraging the developers to analyze situations, think critically about problems and challenge the developers to formulate their own solutions [14]. As mediator, he acts as an interface between external parties and the development team in a role similar to that of the Scrum Master, but with a focus on the quality of the production (Figure 7.4).

The Quality Master is there to help programmers with architectural design, refactoring and other technical skills, to determine the long term quality goals of the project and to organize events and workshops around the theme of code quality. The Scrum Master is the champion of the method and ensures that the team “adheres to Scrum theory, practices and rules” [116]. He helps the team in their quest to become self-organized and efficient. The Scrum Master assists the Quality Master by integrating this new role and quality events into the Scrum framework. The Quality Master works closely with the Scrum Master and the development team, focusing on the technical and educational aspects of internal quality of the application being developed. He monitors the evolution of the code on an individual, group and application level, organizes Quality Events and assists the development team in improving code quality on a continuous basis.

The application of the Collaborative Model impacts the existing Scrum framework – new events and controls emerge that must be integrated in the development cycle (Figure 7.3). The different roles in the framework have emerging needs covered by a support system that was created to assist the team in monitoring code quality and applying aspects of knowledge creation and acquisition as an integral part of development.

7.3. Support System

The Quality Master plays an essential role as guide and facilitator. Through monitoring tools, metrics and communication with the team he can spot shortcomings in quality and organize training and discussion around these themes. The developers must also have a view into the quality of the application in order to evaluate the current state of affairs and discover shortcomings in the handling of code quality. Management must be able to oversee the evolution of quality to measure the efficacy of the development method.

The application of the Collaborative Model to the development process generates the high level requirements of improving code quality and the advancing the competencies of the developers in the realm of internal software quality. These requirements generate new user needs for the developers, the Quality Coach and management (Table A.1). Essential aspects of this solution are the ability to monitor and evaluate code quality on a group and individual basis, to access a communal repository for the sharing of knowledge and to consult statistics on quality. Requirements for the support system are extracted from these needs (Table A.2). The system must be able to generate and store code analysis results on a per commit basis, to display metric results within the development environment and provide a quality documentation repository. The development environment must also be linked to this repository for easy access. New function blocks (Table A.3) were designed to handle the new requirements and are integrated into a traditional software factory (Table A.4). This integrated system allows the team to monitor code quality over time and to document learning in their continual journey through the Collaborative Model for Code Quality.

Metrics provide an objective manner of evaluating code quality and are used to judge certain aspects of the internal quality of development. The Quality Analyzer generates these metrics through a static analysis of the source code. Quality documentation is stored in a repository which is linked to from the developers toolkit, making accumulated knowledge readily available for consultation or modification. A Continuous Integration (CI) Platform acts as the hub of this support system and coordinates the compiling, analysis and deployment of the application. A Version Control System is the repository for the source code and provides a history of modifications.

The developers have a window into the generated metrics and communal documentation via their IDE. Unifying different views and portals into the IDE, allows easy access to a wealth of information on quality and incites quality consciousness. Each developer has an easily accessible overview of the quality of their own development as well as the overall quality at a moment in time or for a chosen period. A detailed view of the metrics can also be viewed directly via the Quality Analyzer which provides a web interface. The Quality Master, Scrum Master and management can also view the results and monitor the evolution of code quality through a supervisor dashboard (Figure 7.5).

Not all aspects of code quality are measurable through metrics and are best managed by experience and know-how. This expertise is acquired through the application of the Quality Model and the knowledge of the team is stored in the Quality Documentation Repository. Group documentation is accessible to all parties and active participation in the development of this repository is necessary for the effective sharing of knowledge within the team. By linking this repository to the development tools, developers are incited to keep this documentation up to date and to regularly consult it to further their understanding of development patterns, metrics and other quality principles.

The basic workflow of this tool begins with a developers commit. Each time a developer pushes new code to the central version control, the CI Platform is notified and compiles the application. All unit tests are executed and if the build is successful, a static analysis of the code is executed by the quality analyzer. The resulting metrics are then stored and tagged with metadata in a central location that can be queried by developer workstations and the supervisor dashboard. Several views of the resulting metrics are available to all concerned parties through different interfaces.

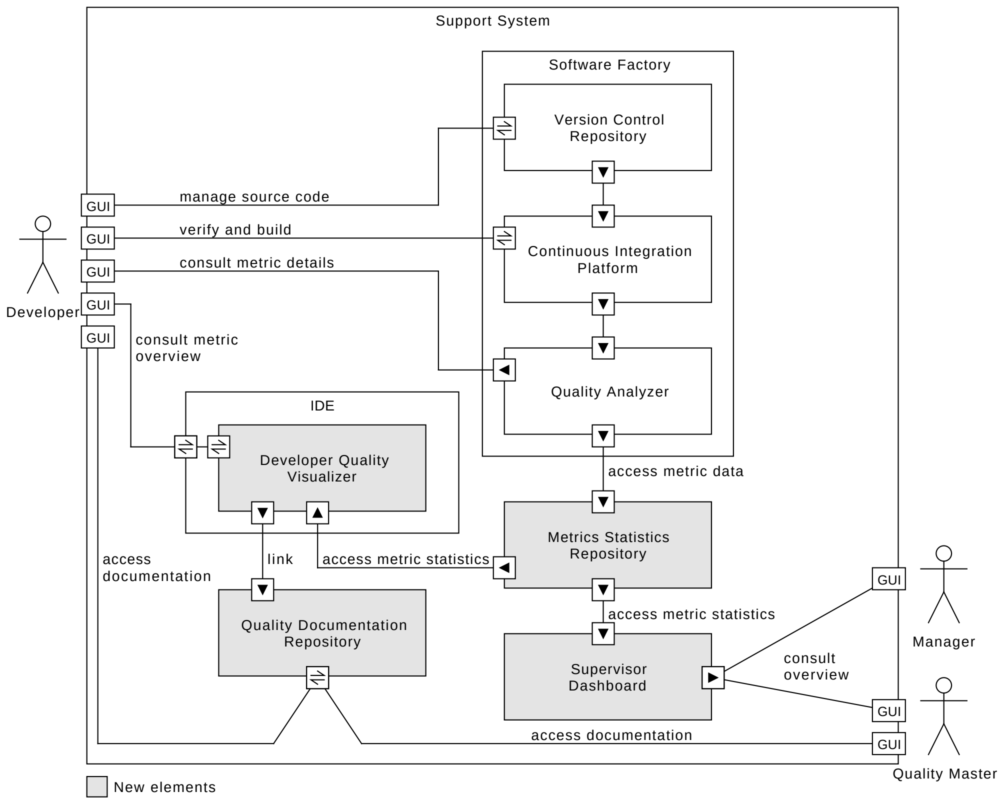


Figure 7.5: Support System Architecture

7.4. Process for Applying the Framework

A process for applying the proposed Collaborative Code Quality Framework has been developed and is based on certain preconditions.

Prerequisites

This solution presupposes a development team that follows a Scrum methodology and a desire to continually improve code quality. The executives, management and development team all must be implicated in the improvement of code quality: all members must understand the importance of code quality and the benefits it can bring.

Another key aspect of improving code quality is having developers who are open minded and eager to learn and share their knowledge with other members of the team. Each developer must make a concerted effort to assist others where possible and be willing to accept help from their colleagues. A prerequisite to a healthy team and code base is openness and communication within the team.

Modus Operandi

The introduction of any new methodology implies changing the status quo and thus requires the support of the team and management. To overcome the natural resistance to change, it is necessary to communicate the benefits and to proceed slowly and incrementally. To adopt the proposed method of collaborative code quality, the main steps are getting all concerned parties on board, designating a Quality Master, defining short and long term goals, setting up a supportive environment and evaluation tools and possibly enlisting outside expertise.

Engagement and Communication

The first step in implementing the changes proposed is to get everyone on board. It is especially necessary to gain support from management and decision makers. A study of code quality delivered through the existing development method can highlight shortcomings and areas for improvement and provide a baseline of quality. Unsatisfactory results can provide motivation for the concerned parties to test a new method that may yield better outcomes than the current methods. A pilot or test implementation could be put in place on a small project to test the feasibility of the new method and to verify the results are satisfactory.

The individual benefits for all actors and for the group as a whole should be communicated. Besides the benefits extolled in the chapter on code quality (chapter 4), this method provides individual advantages to developers, managers and executives. Developers work together to improve code quality and become technical advisors and experts on various aspects of code quality and design. Sharing this knowledge with others empowers the developers – each member of the team becomes both teacher and student in a shared quest for quality. By adopting shared ownership of quality, individual's skills are honed, autonomy grows and everyday tasks become opportunities to enhance learning. For management and executives, this approach results in a higher quality product requiring less time and expense to maintain and improved customer satisfaction. The support system allows visibility of the overall quality and its evolution.

A communication and training campaign can help all interested parties understand the benefits of undertaking a quality focused method based on collaboration. Through the use of metaphor and analogy, the global objectives of this method can be communicated in a symbolic manner that captures the imagination of the team. Metaphor is a way of using everyday language and images to express an idea which promotes communal understanding over individual interpretation. Specifics can be communicated through various forms such as documentation, meetings of various sizes, email and through formal and informal discussions. It is also necessary to provide a mechanism to receive feedback from all concerned parties and to take into account their concerns, apprehensions and ideas for improvement.

Roles

An environment that promotes personal skills development is necessary to the knowledge creation and sharing that are central to the collaborative approach to quality. Quality is hard and mistakes will be made. The team culture must promote experimentation and accept failed experiments. An Agile team embraces missteps and uses them as learning opportunities. They react quickly and develop new strategies to overcome these new obstacles.

The Scrum Master and Quality Master are charged with putting in place and maintaining a constructive environment that promotes knowledge creation and learning. As the Scrum Master is the champion of the method, the Quality Master is the proponent of quality. They work together to maximize the quality of the application being developed and create an environment conducive to talent development and knowledge creation where the developers can learn and excel. The Quality Master must have certain attributes: technical proficiency, expertise in code quality and metrics, be diplomatic, facilitative and a servant leader.

The Quality Master oversees the quality of the team's production and by analyzing metric data and doing code reviews. He must therefore understand the metrics being used and be able to interpret the data. He must also be technically proficient and understand architectural design and patterns to coach developers in writing code that exhibits quality characteristics. He must be diplomatic and ensure that team members are working together and handle conflicts adeptly. He encourages teamwork to accomplish a shared goal and facilitates the integration of quality concepts into the development workflow. Like the Scrum Master, he acts as a servant leader. Where the Scrum Master protects the team from impediments, ensures that the team follows the principles and values of Scrum and helps coordinate the method, the Quality Master provides a clear vision of quality goals and guides the team towards them. He assists team members in becoming more autonomous, confident and skillful in their coding practice.

Goals

Implementing this method can be done in stages. At the beginning short and long term quality goals are defined. Clearly defined objectives allow the team to measure the effectiveness of their interventions. The learning curve for an inexperienced team can be steep. It takes time and investment to bring the team up to speed on the meaning and interpretation of various quality metrics. Through quality building events and the application of knowledge creation theories the team can steadily increase their know-how and expertise in the realm of code quality and design.

To start a minimal set of metrics can be used. The most relevant metrics to the project should be defined by the team with the help of the Quality Master. One strategy for establishing a baseline is by looking at technical debt (subsection 4.4) and using this metric as an indicator for the overall quality of the application. Simple modifications can have a large impact on the overall technical debt. Establishing a uniform coding standard and syntax can eliminate many warnings and lead to a more consistent code base.

Once the minimal set of metrics has been sufficiently documented and assimilated by the team other metrics should be added. The acceptable threshold can also be modulated in function of the capacities of the team. As they become more proficient, these limits can be tightened.

A way of kick-starting the process is by enlisting outside expertise such as a quality expert or a method/process expert. If the team does not initially have the technical know-how to evaluate code quality and metrics, an external expert can accelerate the process. This expert can provide training on quality principles and metrics for the development team and help managers understand the benefits of improved quality. He can also assist management in understanding the quality dashboard and provide them with the knowledge necessary to effectively oversee the evolution of quality. Although this method is based on the communal creation and sharing of knowledge, it can be beneficial to the team to receive coaching and training. Workshops on different coding and design techniques can broaden the knowledge of the team and accelerate their learning and internalization of quality issues.

Events, such as code quality workshops, group coding sessions, deliberate practice and coding katas should be integrated into development sprints. The choice of events and their frequency should be determined by the team under the supervision of the Quality Master. As a collaborative method, all parties can make proposals which are discussed by the team. The facilitative and diplomatic qualities of the Quality Master are exercised in getting everyone on board and committed to these events.

Support System

A support mechanism providing a way of measuring the quality of an application and its evolution along with a documentation repository for sharing knowledge are essential factors in the implementation of this framework. The integration of a project analysis tool into the development workflow is necessary to be able to measure the internal quality of an application. Access to communal documentation is a key factor in advancing the knowledge creation spiral. Integrating these tools into the work environment incites quality awareness in the developers and promotes autodidacticism and the sharing of acquired knowledge and skill. A support system prototype was developed (chapter 8) as an example of the tools that can integrated into a traditional software factory. The effectiveness of this prototype and was test driven in a trial implementation.

8 The Support System

A support system was created to verify the feasibility of the proposed support system and to create an environment where this collaborative approach to quality could be tested. This system is based on open source software and several applications were developed to tie various off the shelf software components together (Figure 8.1). This support system provides an overview of developer and project metrics over time and integrates an easily accessible documentation repository to promote shared learning. It is based on the following technologies:

- Eclipse[1] Integrated Development Environment (IDE)
- MediaWiki[2] Wiki engine for the documentation repository
- Git[3] version control system
- Jenkins[4] Continuous Integration (CI) server
- SonarQube[5] project analysis tool

The various services are deployed using Docker[6] to ensure a reproducible environment and facilitate their migration into production. In order to provide the required functionalities, two applications were developed and integrated into this software factory: a plugin for Eclipse (Metrics Dashboard[7]) and an application that retrieves metric information on a per developer basis (Stats Collector[8]).

8.1. Software Components

The IDE that was used for this system is Eclipse because of its large presence in Enterprise software development and its extensive plugin architecture. Eclipse is based on OSGi, a modular architecture for Java which allows the creation and integration of software bundles. This open architecture allowed for the creation of a new Eclipse view so developers can consult the results of quality analyses directly in their principal work environment. Filters were created so that developers can consult metrics per developer, per application/version, per commit or by a range of dates. The value of each metric is displayed along with the difference since the previous analysis. Since developers spend much of their time within this environment, putting a metric dashboard within Eclipse encourages developers to follow the evolution of code quality. To incite knowledge creation and retention, each metric is also linked to a communal Wiki allowing easy access to documentation directly from the workspace of each developer.

A Wiki was chosen for the documentation repository because of its ease of use and its founding philosophy coincides with the collaborative nature of this

[1]http://www.eclipse.org/
[2]https://www.mediaWiki.org
[3]https://git-scm.com/
[4]https://jenkins.io/
[5]http://www.sonarqube.org/
[6]https://www.docker.com/
[7]https://github.com/mio-to/metricsdashboard
[8]https://github.com/mio-to/statscollector

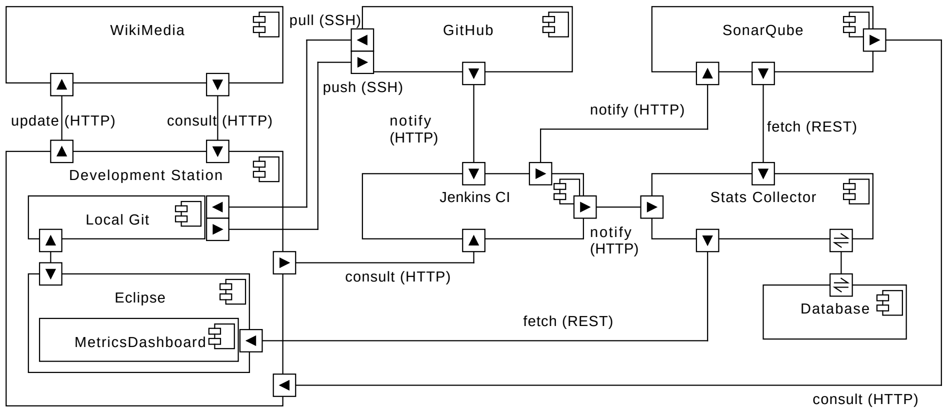


Figure 8.1: System Components

solution. Wiki software was to created to provide collaborative documentation that evolves and involves users. It is not a static repository of documentation, but a living organism that needs to be fed and cared for. This philosophy was set forth by Ward Cunningham who invented the concept and developed the first Wiki called *WikiWikiWeb* [155]. The collaborative nature of Wiki's fits with the goals of this solution and is a means of enhancing collaboration amongst the team. MediaWiki is an open source Wiki engine that was originally created for Wikipedia and has an active community and extensive documentation. Installing and configuring a MediaWiki server is straightforward and well documented. Because of the success of Wikipedia, this paradigm is familiar to most users which facilitates its integration into the team's toolbox.

Git was used for version control because of its ease of branch management and merges, the possibility to work offline, its distributed nature, the large community of developers, ample documentation and tutorials, and the strength of its open source community. Many large open source projects such as Ruby On Rails, JQuery, Debian and the Linux Kernel use Git and this has inspired a large ecosystem and with rich third party functionalities. The code for these components is hosted on GitHub[9] which provides many plugins that enhance Git and facilitate the integration of this system.

The CI server Jenkins was chosen because it is easily configurable, provides numerous plugins and scales well. The CI server is used to run unit tests, compile code and coordinate the compilation/deployment pipeline. It is used to schedule, organize and automate the build process. Bash scripts can be integrated into Jenkins further extending its functionality. Jenkins can also be deployed on several machines allowing for parallel compilations, testing and deployment. This capability could be necessary in a production environment were multiple developers are coding and pushing functionalities several times per day. The prototype that was developed uses one instance of Jenkins which is sufficient for an individual developer or a small development team.

[9]`https://github.com/`

SonarQube is an open source code quality management platform. It covers seven axes of code quality: architecture and design, comments, coding rules, duplications, unit tests, complexity and potential bugs [128]. It provides a rich web interface which allows configuring and viewing of various quality metrics and integrates the SQALE quality model for measuring technical debt. It also provides an REpresentational State Transfer (REST) API and can therefore be exploited by third party applications. The metric analysis is integrated into the build process via a plugin Maven and the results are retrieved via the RESTful API provided by the SonarQube server.

Docker is a containerization platform [48] that allows applications to be deployed in a configured environment that remains the same no matter where it is deployed. In this way it is similar to virtual machines; however, by sharing the operating system kernel of the host machine, less resources are necessary to achieve the same level of isolation and reproducibility. These containers are configurable via text files and DockerCompose is a service that allows configuring several containers and connecting them together in a single script. A DockerCompose file was used to create and link Jenkins, SonarQube and a database container. These three containers are configured to communicate with one another. Docker containers are highly configurable and can be deployed on several machines to improve performance. For this example all components were deployed on a single machine.

8.2. Constraints

Several difficulties and constraints were encountered during the conception of this solution. Some of these are linked to the functionalities of the existing applications that were to be used, some related to the existing development process and company culture.

To accomplish the goal of collecting quality statistics on a per developer basis, it is necessary to correlate the changes made by each developer with the generated metrics. SonarQube does not provide a mechanism to get per developer statistics except through a proprietary plugin that was not accessible for the development of this prototype. In order to accomplish this goal, an intermediary application, Stats Collector, was conceived to extract metric data after each Git push. In conjunction with the CI platform, SonarQube and Git it is possible to regularly extract metrics, tag them with metadata and store them in a database. By accumulating metric data over time, this database can be queried by external tools that display per developer contributions to quality.

This development of this solution brought to light the problem of simultaneous pushes by several developers. It is necessary to extract the data for each developer despite the fact that several developers can push changes at the same time. Through the use of Jenkins and GitHub, it is possible to build the project separately for each push. When GitHub receives a push command, it sends a web hook in the form of an HTTP request to Jenkins. Jenkins listens for this request and once received, clones the repository up to this push. Jenkins then proceeds to test, build and analyze the project, submit the results to SonarQube and notify Stats Collector of the build through its RESTful API.

Certain aspects of developer habits must also be addressed for the successful exploitation of this solution. On the project e.sedit RH developers use the version control system as a place to save work and to share files. Version control should ideally be used to track versions and logical changes to the application, not as an ad-hoc backup and file server. This played a role in the decision to use Git instead of SVN for version control. Git allows developers to create development branches, commit locally and only push finished work. Its distributed nature also allows developers to share files without pushing to the central repository. These "best practices" [113] have the secondary benefit of improving the integrity of the central repository as it only contains completed functionalities.

The support system prototype that was developed takes into account these difficulties and proposes certain technical and behavioral solutions.

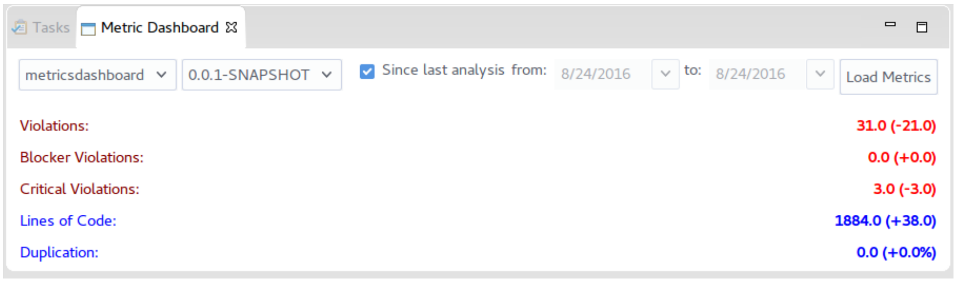
8.3. Overview

These software components work together to provide a support system for a collaborative approach to code quality (Figure 8.1). The principal window into the system for the developer is an Eclipse plugin that presents different quality metrics and their evolution over time. Developers can view their own contribution to the quality of the application and interact with the documentation repository to further their knowledge and share techniques and practices that they find useful.

As a developer creates new functionality, local changes can be committed using Git. Once a functionality has been finished, unit tests executed and the code reviewed, changes are pushed to a central Git server. GitHub provides a mechanism to notify a CI server that changes have been made via webhooks – an HTTP callback. This callback takes the form of an HTTP request to a specified Uniform Resource Identifier (URI). Jenkins provides Git and GitHub plugins that can be configured to listen for these HTTP requests and launch a build upon reception.

Once the Jenkins server receives this request, it synchronizes with the central Git repository and begins building the application. The build process is configurable and assorted compilation tools can be used. The components of this system use Maven to manage project dependencies and deployment. Many plugins are available for Maven that facilitate the generation of documentation, test coverage reports and deployment. It is also possible to execute a SonarQube analysis via a Maven plugin. This analysis is executed on the Jenkins server and a report is sent to the SonarQube server. This server analyses the report and updates its database of metric information. After completing a successful build, a custom webhook is called via a Bash script. A simple "curl" command is executed to send an HTTP GET request to the Stats Collector server containing the project name, version number, Git commit ID and developer identifier.

The Stats Collector component is a custom application that provides quality statistics via a RESTful API and extracts raw data from SonarQube via its RESTful API. When the Stats Collector receives a request informing it of a recent successful build, this component queries the SonarQube server for the most recent metric values and stores them in a database. The accumulation of metrics and the association of



☑ Since last analysis from: 8/24/2016 to: 8/24/2016

Violations:	31.0 (-21.0)
Blocker Violations:	0.0 (+0.0)
Critical Violations:	3.0 (-3.0)
Lines of Code:	1884.0 (+38.0)
Duplication:	0.0 (+0.0%)

Figure 8.2: Metrics Dashboard Screenshot

metadata allows for the Eclipse plugin, Metrics Dashboard, to query this service via its RESTful API and deliver the requested statistics to the end user.

Metrics Dashboard is an Eclipse plugin (Figure 8.2) developed following the OSGi standard. This plugin provides an Eclipse view for developers to consult the current state of code quality metrics directly in their IDE. This component also provides links to the Wiki and facilitates access to communal documentation. This plugin integrates a REST client to query the Stats Collector and parses the results. All communication between these two components is via REST over HTTP and the results are marshaled into the JavaScript Object Notation (JSON) format.

8.4. Details

Two components were developed specifically for this system – Stats Collector and Metrics Dashboard.

Stats Collector

As previously mentioned, Stats Collector is a web service client and server that accumulates metric data. It fetches the most recent metrics from a SonarQube server and stores them in a database.

Stats Collector offers a RESTful API so that this data can be used by external applications such as the Metrics Dashboard or a Web UI. The public API is defined in a Web Application Description Language (WADL) which facilitates its exploitation by third party applications. This component uses Jersey, the reference implementation of JAX-RS – the Java API for RESTful Web Services. Jersey simplifies the creation of RESTful web services, and also provides a REpresentational State Transfer (REST) client. Jersey is extensively documented, has an active community of developers and provides simple configuration and exploitation. It comes with a dependency injection framework HK2[10], and integrates a choice of JSON marshalers/unmarshalers. Jackson was chosen for its controllability and performance [49]. For database access, the ORM used is the EclipseLink[11] implementation of Java Persistence API (JPA). The build process is handled by Maven to facilitate dependency management. Several Maven plugins are used to execute a SonarQube analysis, generate unit-test coverage reports and create an executable "fat" JAR.

[10]`https://hk2.java.net/2.5.0-b07/`
[11]`http://www.eclipse.org/eclipselink/`

This component has an N-tiers architecture with three layers (Figure A.2). The inversion of control pattern is used in this component to ensure that dependency moves in the direction of abstraction as described in the section that concerns the SOLID principle of OO programming (subsection 4.4). Each layer of this architecture uses various services that are known through an interface as proscribed by the Dependency Inversion Principle. The use of this pattern as well as having classes in each layer depend only on abstractions makes the application more robust and facilitates creating test doubles. This principle improves the testability and evolvability of the application.

This component does not have a GUI but contains a resource layer which defines its public API. These resources use services of the service layer which in turn use the DAO or Web Service Client layer depending on the demand. The resources are defined as simple Plain Old Java Objects (POJOs) with annotations provided by JAX-RS. These annotations describe the path of the resource, the type of media consumed and produced, the required parameters as well as any access restriction filters that need to be applied.

The Service layer contains classes that implement Service interfaces and act as coordinators to retrieve or persist the required data by communicating with either the DAO layer or the Web Service Client layer. The DAO interfaces define the available methods which are implemented using the JPA provided by EclipseLink. The Web Service Client is based on Jersey's web service client and facilitates making HTTP requests to a RESTful API and returns simple POJOs that have been unmarshaled using Jackson. The Jersey client can also accept POJOs and marshals them to JSON strings that can be sent over HTTP.

The domain model (Figure A.3) was defined in order to add metadata such as the developer, commit ID, project and version to various metric data. The main class of this model is Analysis which represents a single SonarQube analysis. An Analysis is linked to one version of a project and each version is linked to one and only one project. Each Analysis is linked to a Developer and contains a list of Metrics. Each metric is made up of a name and a value and can be linked to a category.

This component is central to the system and acts as an accumulator of metric data. This data can be queried and exploited by third party applications such as the Metrics Dashboard which connects to this component via a REST web service.

Metrics Dashboard

Metrics Dashboard is an Eclipse plugin that displays code metrics (Figure 8.2). Several filters have been created to show overall metrics, metrics since the last analysis, per developer, project or version metrics and metrics for a range of dates. This plugin, as all Eclipse plugins, is based on the OSGi architecture and is a module or bundle that is loaded by the IDE. An Eclipse view was developed to display the metric data. This data is linked to a communal Wiki to allow developers easy access to this documentation.

This plugin follows the Eclipse plugin architecture allowing a seamless integration into Eclipse. The plugin system provides a means to add entries to drop down and

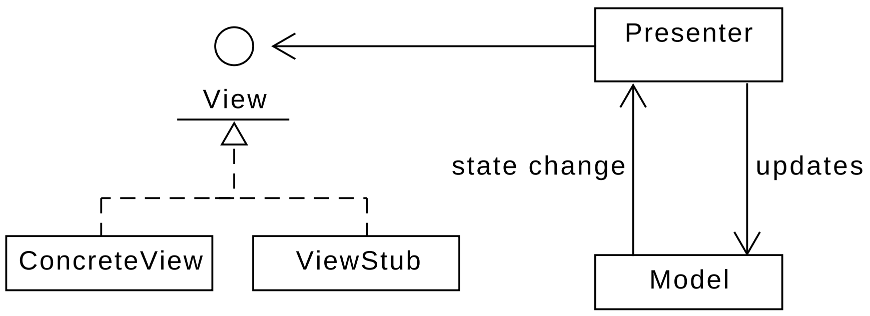


Figure 8.3: Model View Presenter

contextual menus and to the global preferences section. All dependencies, commands, menus and views are declared in the manifest.

Eclipse calls the registered handlers when menu items are clicked or the commands are otherwise triggered which allows for a single entry point for multiple views and menu items. Eclipse Software Development Kit (SDK) handlers and events are implemented to handle the filtering and fetching of metrics from the Stats Collector. It was not possible to use the Jersey web service client within an Eclipse plugin, so a web service client module was developed to handle the low level network communication. This communication exploits the `HttpURLConnection` and Java Input Output (IO) API that is part of the Java Development Kit (JDK). Request parameters are marshaled and responses are unmarshaled using GSON[12] for its ease of use and availability as an OSGi bundle.

This plugin does not contain much business logic and is essentially a front end for the Stats Collector component. All views were developed using the Model View Presenter (MVP) (Figure 8.3) pattern to maximize the cohesion and minimize the coupling between views and application logic. This separation makes it possible to isolate GUI widgets in the view and the application logic in the presenters. In this way the presenters, which are simple POJOs are testable with standard unit tests. MVP is a variant of the Model View Controller pattern applied to user interfaces. There are several variants of this pattern, and the "passive view" [53] was used to improve testability and explicitly handle user interactions. By having a passive view, there is no need to test the view itself as it is a simple collection of GUI widgets. All handlers are implemented in the presenter which updates the model and applies application logic. When the model is updated, the presenter is notified and subsequently updates the view. To further minimize coupling, the presenter only knows about the view through an interface. This facilitates writing tests for the presenter because the concrete view can be replaced with a test double.

To ensure that the presenter is not coupled with GUI widgets, the presenter uses abstractions of these widgets for the implementation of handlers. Most of these widgets have an interface such as `ISelectionProvider` with the method `addSelectionChangedListener(ISelectionChangedListener listener)`. The view in-

[12]`https://github.com/google/gson`

terface has a method `getComboView()` which returns a widget seen by the presenter as an `ISelectionProvider`. This way the presenter can add a listener to a widget without being coupled to the implementation details of the GUI. Certain widgets are not based on interfaces, and for these the GUI events are propagated to the presenter through the event bus.

This pattern was implemented in Metrics Dashboard for the handling of the search filters (Figure A.4). The `MetricViewPart` is a `ViewPart` which is an Eclipse view. This class acts as a controller and creates the `CriteriaComposite`, `CriteriaPresenter`, `BasicEventFactory` and `ResourceWSClient`. `CriteriaComposite` is a composite which contains a set of Standard Widget Toolkit (SWT) widgets and layouts. This class implements an interface, `CriteriaDisplay`, which is used by the `CriteriaPresenter`. The `EventFactory` handles the implementation of the event bus available to Eclipse plugins. The view sends certain events to the bus and the presenter is registered to listen to these events. The use of the event bus was necessary for certain SWT that are not based on interfaces. The implementation of the `ResourceWSClient` handles the communication via HTTP with the Stats Collector component.

The list of metrics is loaded when the "load metrics" button is clicked. SWT buttons do not have an interface with a method to add a listener, so the click event is sent to the presenter through the event bus. The presenter receives this event, collects the state from the model and launches the loading of metrics from the Stats Collector component and displays the results (Figure A.5). The `LoadMetricsHandler` uses the `ResourceWSClient` to load the metrics and sends them to the `DetailsFactory` which creates a line for each metric. Each line is created by instantiating an implementation of `Details` with a overwritten `setLabelText()` and `setValueText()` specific for each metric.

As with Stats Collector, this component is built using Maven; however, because of the nature of dependency management in an OSGi bundle, a set of Maven plugins specifically designed to handle OSGi metadata and dependencies was used. This suite of plugins, called Tycho[13], also facilitates the execution of JUnit tests during the build process.

These two components were integrated into this support system from the beginning of their life. This allowed the development team to follow the evolution of code metrics and quality throughout the creation of these two components. The integration of these two components with Jenkins, Git, SonarQube and a Wiki establishes a support system for a collaborative approach to code quality. These tools enhance the development process by putting a spotlight on code quality. By having a constant view of project analysis results and the personal contribution to code quality, developer awareness of quality issues improves. The communal documentation of the Wiki allows developers to synthesize quality concepts and techniques and share their learning with the rest of the team.

On a technical level this system provides a working prototype and an example of different tools that can be combined to create a support system, but it has certain fundamental shortcomings. Newer versions of SonarQube handle analysis results asynchronously and therefore it is not possible to know when each set of results will

[13]`https://eclipse.org/tycho/`

be handled. Stats Collector could extract data for several developers at the same time leading to inaccurate results. Several solutions to this problem are conceivable: use the commercial SonarQube plugin, develop an open source version of this plugin or have each developer run a local version of SonarQube. The commercial plugin is the simplest solution that provides the most comprehensive functionality but this choice was eliminated because of it was not financially feasible. Recreating this plugin was also ruled out because creating per developer metrics requires an intimate knowledge of the existing SonarQube database and structure. These internal aspects of SonarQube are subject to change at any time and therefore any plugin that relies on a specific structure will be difficult to maintain and evolve to keep up with the rapid changes in SonarQube. Reverse engineering SonarQube is also a monumental task that was not possible in the context of this prototype. The final solution is possible, but not practical. Each developer could run a local version of SonarQube, run a metrics analysis before pushing to the Git repository and Stats Collector could then query this instance to accumulate data. This requires more work on the part of developers, makes the system cumbersome for the team and consumes significant resources. None of these solutions is entirely acceptable and this remains a difficulty that needs further study.

Exploratory Study

9

In order to test the feasibility of the framework and the process of application, a trial implementation was carried out and the results of this experiment will be discussed. In view of applying this framework to the complex production environment of e.sedit RH, several modifications to their current development processes are proposed.

9.1. Trial Implementation

The proposed method of collaborative code quality was implemented in a reflective manner. During the development of the support system, the system itself was used to evaluate the code quality of its software components (chapter 8). This exercise provided a test bed for determining the usefulness of the support system within the development of an application.

Many of the prerequisites for the implementation of this method were not met. This made evaluating the efficacy of this method on code quality unfeasible. As previously described, this method is centered around a communal approach to development and to quality; however, this system was implemented by one person and therefore all considerations of group dynamics and communal learning could not be evaluated. Nevertheless, the usefulness of the support system in inciting quality improvements and individual learning through experimentation was tested.

Throughout the development of the Metrics Dashboard and Stats Collector components, they were analyzed by SonarQube and relevant metrics were available through the web interface. This interface provided feedback on technical debt, potential bugs, code smells, violation and other metrics. In the beginning, this web site was not often consulted because accessing it required switching from the context of developing to that of analyzing code. This interruption slowed development and was relegated to a task to be done after a development session or before starting a new one.

As the project matured, stats began appearing directly in the IDE. Seeing metrics improve or degrade provided motivation for immediately correcting problems and kept quality considerations in the forefront of development. Short "quality sessions" were adopted to improve the quality of previously written code based on the feedback of SonarQube, SonarLint and the Metrics Dashboard. The Eclipse plugin, Metric Dashboard, provides an overview of various metrics and violations as calculated by SonarQube with the difference since the last commit. This synopsis allowed a quick evaluation of the quality of each commit which provided a short feedback loop. Details of quality issues could be looked up using the web interface of SonarQube allowing for the correction of problems in a timely manner. Seeing quality metrics improve consistently over time was a motivating factor in continually improving code quality. The ability to easily switch the Metric Dashboard view from one project to another also facilitated following the quality of several applications without leaving the development environment.

The support system proved effective for motivating an individual developer in improving code quality, but the crux of this method concerns increasing developer awareness of quality issues, sharing knowledge with others and learning from the experience of peers. The next step in experimentation would be to implement this method in a pilot project to evaluate its efficacy in improving code quality and team cohesion. Ideally the pilot implementation would be a project with a relatively small and co-located team. A software factory would need to be put in place and a support system developed to integrate with it. The prototype developed for this thesis could be a basis for development. This would permit the assessment of the influence that this method of collaborative code quality has on the mind-set of the development team and on the code itself.

9.2. Proposition for e.sedit RH

The application, e.sedit RH, under development at Berger Levrault is not an ideal candidate for a trial implementation because of the number of developers and the fact that several teams work on the application in geographically disparate sites. Nevertheless, a proposition for integrating a quality approach is proposed.

In order to institute significant changes to the development method and processes with the goal of integrating notions of internal quality, management and decision makers must be implicated. The fact that they have supported the research and writing of this thesis demonstrates a desire to improve code quality for e.sedit RH. Workshops and training sessions with a quality expert could aid in motivating management to adopt a strategy to improve quality. Once on board, the first step in applying this method would be the introduction of a Quality Master. This new role would reinforce the importance of code quality from a managerial point of view and raise awareness of quality issues for the team. Once a Quality Master has been designated, he can help guide the team in applying an approach to quality management. He can oversee the operation of defining coding standards, documenting existing practices and patterns and set in motion the collaborative approach to quality code. Quality events should be put in place to kick start knowledge creation and sharing. A supportive and quality centric environment needs to be created.

This software already has some quality evaluation tools and a continuous integration platform that could be leveraged in putting in place a support system. A Wiki has already been put in place for the development team, but remains relatively sparse. Applying the Spiral of Knowledge and introducing quality events and practice would incite developers to document their knowledge. A modification to the implemented version of the Scrum framework and to the basic development workflow is necessary to encourage communication and the sharing of know-how within the team.

This application has a clearly defined architecture and rules for communication between layers; however, it is lacking the application of strict coding standards and principles. A step toward communal ownership and homogeneity in the code base is to define these rules and where possible integrate them into the development environment. Eclipse has a mechanism of automatic code formatting based on rules and their execution through Save Actions. After basic rules of syntax have been defined and automated, design patterns and reusable solutions applicable to commonly occurring situations should be documented and shared with the team. Training sessions

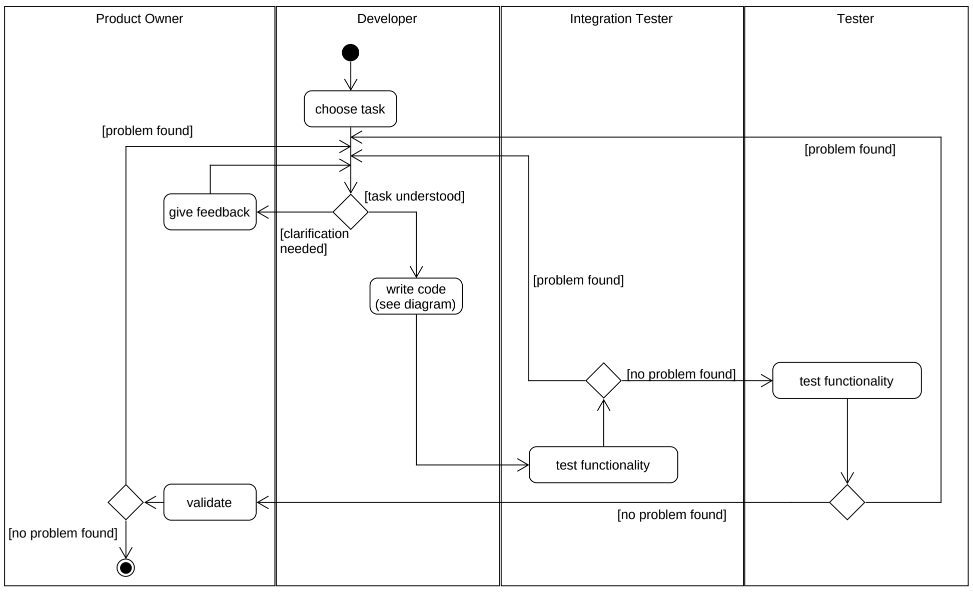


Figure 9.1: Development Workflow

and documentation could reinforce these practices. As solutions for many commonly occurring situations have already been developed, time and effort would be saved in future development through the reuse of existing solutions thus avoiding their continual reinvention. This practice would also lead to a more consistent coding style that facilitates future interventions.

Quality events, such as quality workshops and coding katas should be integrated into the development cycle. Traditional Scrum events (Sprint Planning, Sprint Review, Sprint Retrospective) are already in place and quality events could be added to the end of each sprint. If more training is necessary, these could become weekly events. The team could put aside a half day per week to focus on dedicated practice on specific quality issues and principles under the guidance of the Quality Master. These exercises allow developers to assimilate the knowledge they have gained throughout the sprint and to experiment with complex concepts in a safe and controlled environment. For example, testing newly learned design patterns in a production environment can be a risky proposition; however, by implementing the pattern in a simple test application, experience and confidence can be gained.

In e.sedit RH, the workflow for the development of a User Story within a Sprint involves the PO, developers and testers (Figure 9.1). After the Sprint Planning and breakdown of User Stories into tasks, the developers, in concert with the team, choose a task. The task is further analyzed by the developer and if clarification is needed, the developer contacts the PO. Once the task is understood, the existing code is analyzed and new code and unit tests are written (Figure B.1). The functionality is then tested by the developer. Once all the tasks of a User Story are finished, a member of the team that was not involved in the development of the story manually tests the application to ensure that the delivered functionality is in concordance with the requirements. A dedicated tester then tests the functionality. The process is repeated for the PO who decides whether the story works as expected. During any point in the process, if

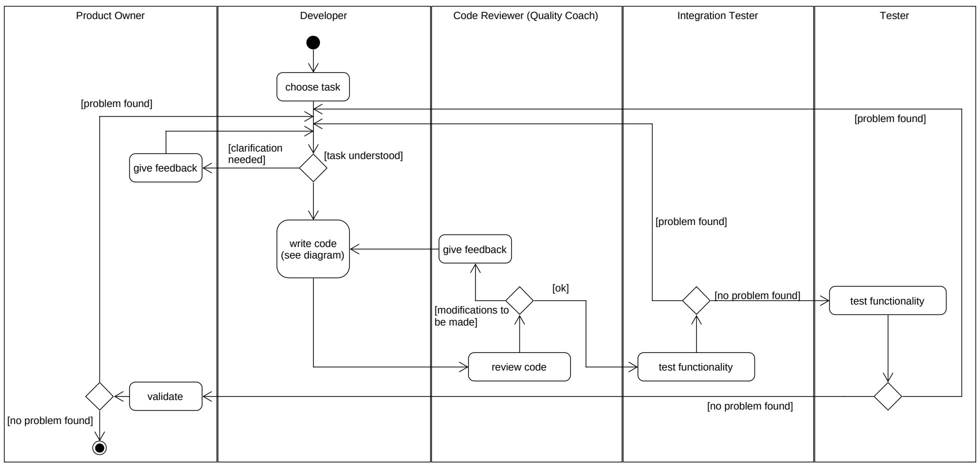


Figure 9.2: Modified Development Workflow

a problem is found, the developer begins again. A functionality is considered "done" when it is functionally correct. Notions of code quality are not considered in this definition. Each developer has the liberty to implement the functionality as he sees fit. This gives the developers autonomy in their design and this freedom is appreciated. Nevertheless, each developer has a unique manner of coding and the lack of coding norms and review process has led to a heterogeneous code base.

To reinforce the sharing of knowledge and skill in the team and shared ownership of code and quality, pair programming or code reviews could be adopted. Putting in place pair programming is a complex proposition, but implementing systematic code reviews is conceivable. A modification to the current workflow is proposed which adds the role of "code reviewer" (Figure 9.2). This role could be held by the Quality Master or by another member of the team. By reviewing each other's code team members would be forced to analyze code from an outside viewpoint and provide constructive criticism for their peers. Mandatory code reviews also cultivate redundancy in the team, promote shared ownership of code quality and improve the maintainability of code [60]. This activity strengthens code quality learning for both parties and allows developers to adopt uniform strategies for solving similar problems. Code reviews also encourage redundancy, enhance homogeneity and the shared ownership of code.

A tool for enforcing systematic code reviews could be adopted. An open source tool, *Gerrit*[1], is a platform that integrates into the development workflow and requires code reviews for every addition to the central repository (Figure B.3). This tool is placed between the central authoritative repository and the developers. All pushes go to this staging ground, are verified by an automatic build server and await code reviews. Once approved, these changes are pushed to the central repository. Different members of the team could review each other's code, or the reviews could be done by the Quality Master. Introducing a solution such as this assures that all code is seen by at least two developers and that all new additions to the code base are subject to critique and discussion. This strategy promotes the sharing of tacit knowledge.

[1]`https://www.gerritcodereview.com/`

Integrating a collaborative approach to quality for the development of e.sedit RH would require significant changes to the development workflow, tools and company culture. The proposed method could be integrated in small steps with frequent evaluations to their efficacy. Applying aspects of knowledge creation and making code quality a shared concern amongst developers and managers would lead to improvements in they quality of the application and help reduce the every growing technical debt and difficulties faced maintaining and evolving the application.

Conclusion

10

Software engineering has continually evolved. From code-and-fix to plan-driven methodologies in order to add more rigor and improve the quality of processes; from plan-driven to iterative methodologies to avoid the downward flow of defects; from iterative to Agile methodologies to better accommodate rapidly changing requirements – these different approaches all seek to improve the quality of the final product.

Agile approaches are designed to improve the reactivity of development teams allowing them to apprehend rapidly changing requirements and environments. For certain types of applications the Agile approach has proven to deliver higher value products more quickly and efficiently than traditional plan-driven methodologies; nevertheless, for certain teams that are not exclusively made up of expert developers, ensuring that the code meets high quality standards can be a challenge. As the Agile approach becomes more mainstream, this situation is becoming increasingly common.

Measuring the internal quality of software is an ongoing challenge in software engineering. Many metrics have been developed to allow the objective evaluation of code quality; however, these metrics are not exhaustive and only provide an indication of quality. Despite the fact that they deliver concrete results, extracting meaning from these numbers remains a subjective task that requires expertise and experience.

Assuring a high level of a product's internal quality in a Scrum team can be accomplished through a collaborative effort. By working together with the shared goal of improving code quality, a Scrum team can overcome the afore mentioned obstacles. Transforming quality into a shared concern incites the team to include these aspects into their definition of "done." The team members continually share and learn through exercises, workshops and documentation. Their continually increasing knowledge of metrics and quality principles allows the team to effectively manage the quality of their production. By monitoring their progress, the team is able to evaluate the results of their efforts. This approach leads to an improvement in the internal qualities of software; the quality characteristics of functionality, reliability, usability, efficiency, maintainability and portability improve.

The quality framework outlined in this document provides an approach to quality that can be implemented by a Scrum team to improve the management of quality. Through the integration of quality events and knowledge creation theories into the development cycle, the team advances through the Quality Model, enhancing the depth and breadth of their knowledge. A shared repository of this knowledge permits the transformation of the team's know-how into concrete documentation allowing the team and organization to capitalize on the combined experience of its members. A support mechanism to allow the team and management to evaluate the quality contributions of each developer and the team as a whole has been described. This is an essential part of the method as it allows continuous feedback which increases awareness of quality issues within the team and acts as a motivating factor in the continuous improvement of code quality.

Scrum is a development framework and each Scrum team implements it in a unique manner. No one size fits all solution is possible, and the same can be said of this method. The proposed modifications to the Scrum framework allow flexibility in the choice of events and controls.

Given the large number of possible implementations of this method, it is difficult to evaluate the efficacy of this solution without numerous trials. To begin, a test implementation of the support system was carried out. This system proved useful in raising awareness of quality issues and in providing motivation for improvement. Having quality metric results integrated into the development environment simplified access to this information and supplied motivation to continually improve. Nevertheless, the core of this method is collaboration and knowledge creation within a team. This aspect of the method has not been tested.

A proposal for implementing this method for the development team of an enterprise application has been written, but remains unimplemented. The complexity of this project and the size of the team make it a less than perfect test case. Ideally this method would be tested in a smaller, more easily controlled environment where the effect of the method on the internal qualities of software could be more accurately evaluated. A smaller team would also facilitate discussion and conciliation which would allow for the experimentation necessary to refine the method. As this method remains largely theoretical, feedback from trial implementations is needed to be able to evolve and improve it.

The next step in the experimentation of this method is to implement a trial. A support system would need to be put in place to measure the existing internal qualities of an application. Under the guidance of a Quality Coach the method needs to be implemented and the results evaluated on a regular basis to be able to evaluate the efficacy of the proposed solution. Observations and reactions of the development team would aid in improving the method. Short feedback loops would allow the continual modification and improvement of the method until a satisfactory version has been developed. Reimplementing the improved method on other pilot projects would permit the evaluation of the effectiveness of the collaborative approach to code quality.

III

Appendices

A Details of the Support System

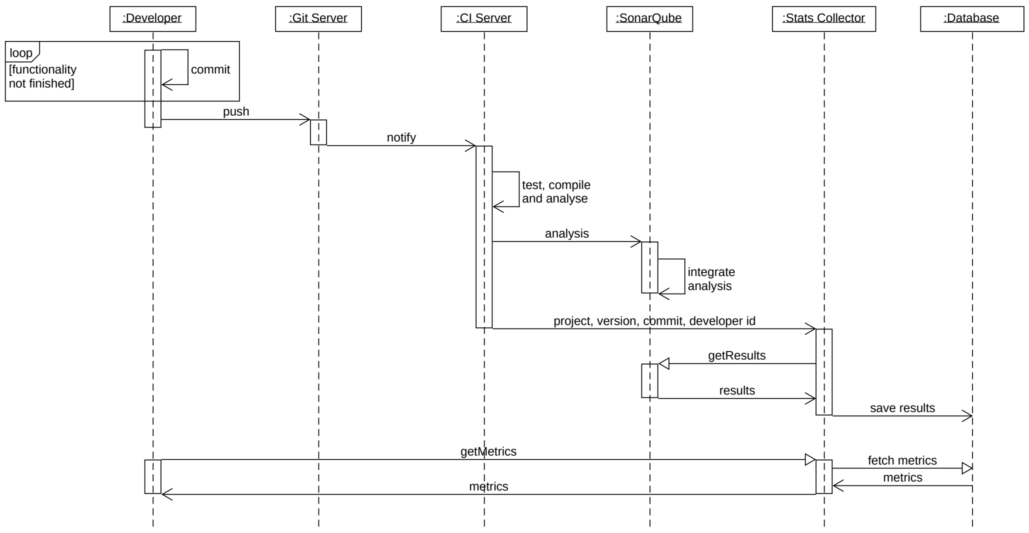


Figure A.1: Implemented System Sequence Overview

User / Need	Dev	QM	Manager
Share knowledge	X	X	
Acquire knowledge	X	X	
Chronicle learning	X	X	
Access information about quality issues	X	X	
Verify the quality of a single commit within development environment	X	X	
Follow the evolution of the overall code quality within development environment	X	X	
Follow the evolution of personal contributions within development environment	X		
Follow the evolution of quality per developer	X	X	
Consult current quality statistics		X	X
Consult quality evolution statistics		X	X

Table A.1: New User Needs

Id	Requirement
SR01	Analyze project quality on a per commit basis
SR02	Generate quality statistics per commit
SR03	Generate quality statistics per developer
SR04	Generate quality statistics per project and version
SR05	Generate quality statistics for a range of dates
SR06	Allow users to read documentation
SR07	Allow users to write documentation
SR08	Allow users to modify documentation
SR09	Connect development environment to documentation
SR10	Display quality statistics overview based on criteria

Table A.2: New Support System Requirements

Id	Function Blocks
FB01	Quality Documentation Repository
FB02	Developer Quality Visualizer
FB03	Metrics Statistics Repository
FB04	Supervisor Dashboard
FB05	Version Control Repository
FB06	Continuous Integration Platform
FB07	Quality Analyzer

Table A.3: Function Blocks

Func / Req	FB01	FB02	FB03	FB04	FB05	FB06	FB07
SR01			X		X	X	X
SR02			X				X
SR03			X				X
SR04			X				X
SR05			X				X
SR06	X						
SR07	X						
SR08	X						
SR09	X	X					
SR10		X	X	X			

Table A.4: Requirements and Function Block Mapping

A.1. Stats Collector

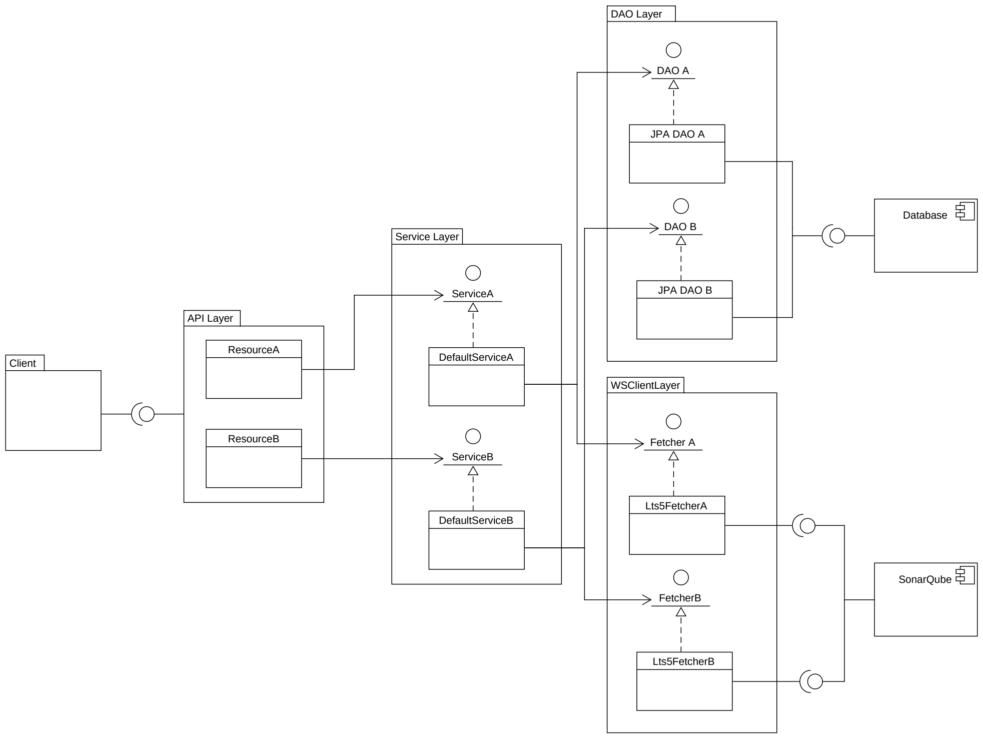


Figure A.2: Stats Collector Architecture

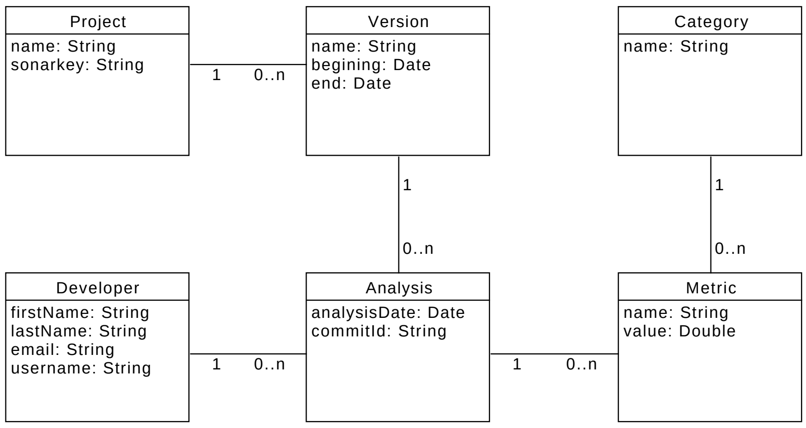


Figure A.3: Stats Collector Domain Model

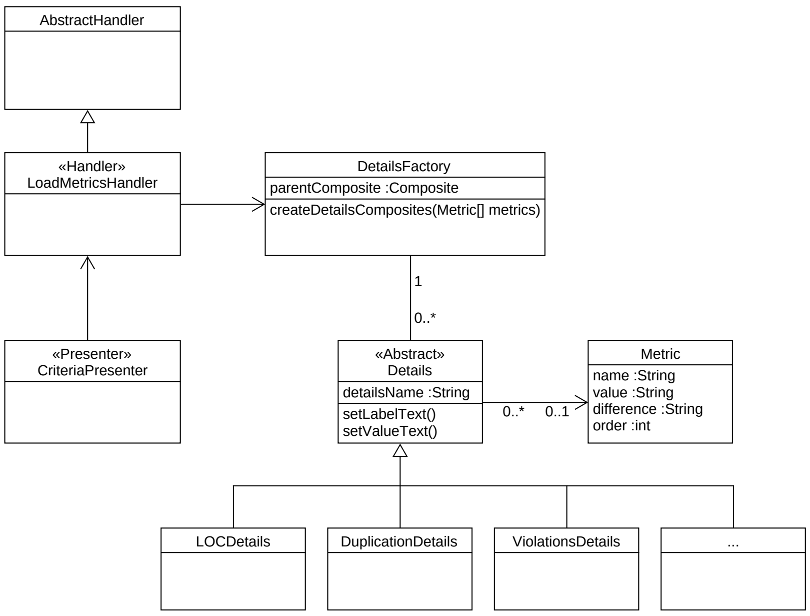


Figure A.5: Metric Details View

A.2. Metrics Dashboard

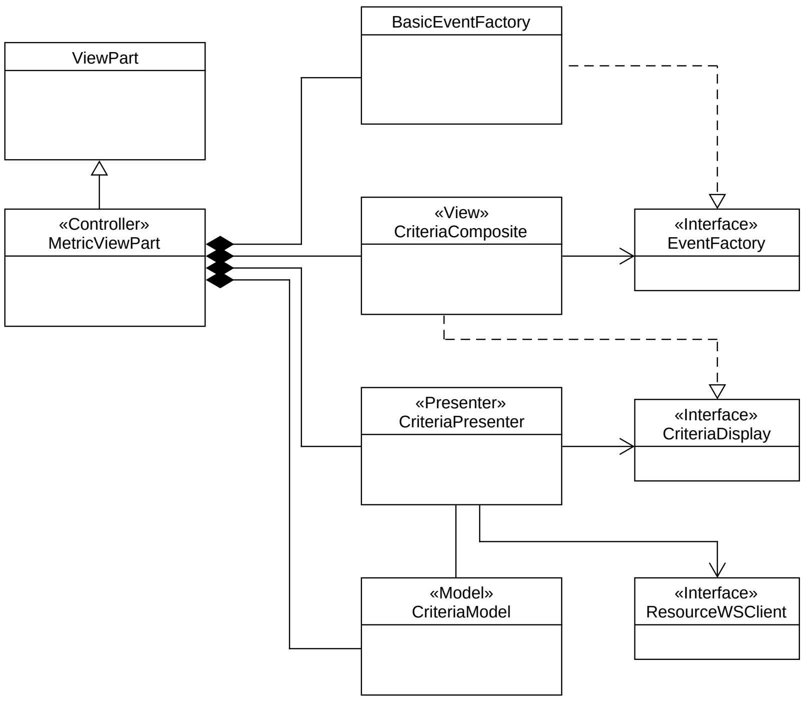


Figure A.4: MVP Implementation

Proposed Implementation Details

B

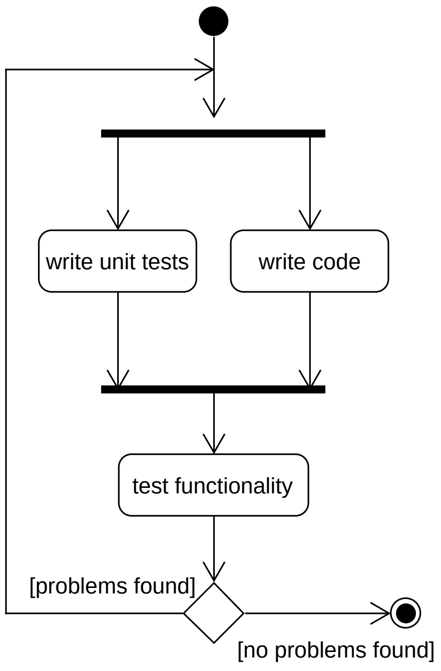


Figure B.1: Write Code Workflow

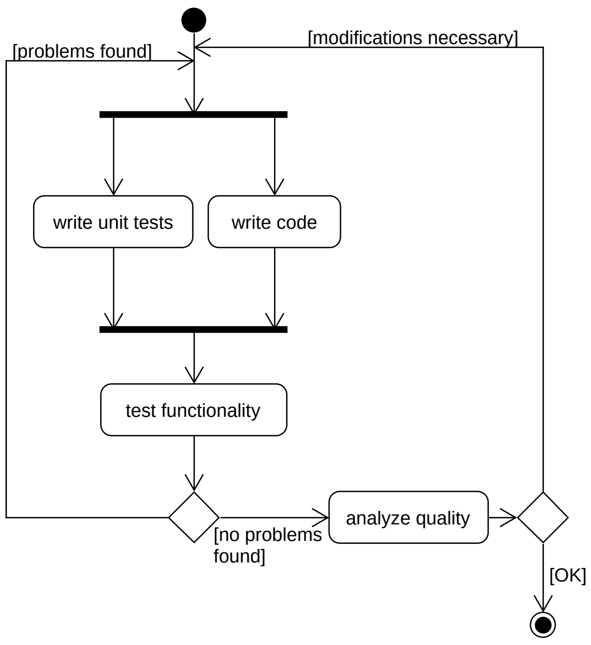


Figure B.2: Modified Write Code Workflow

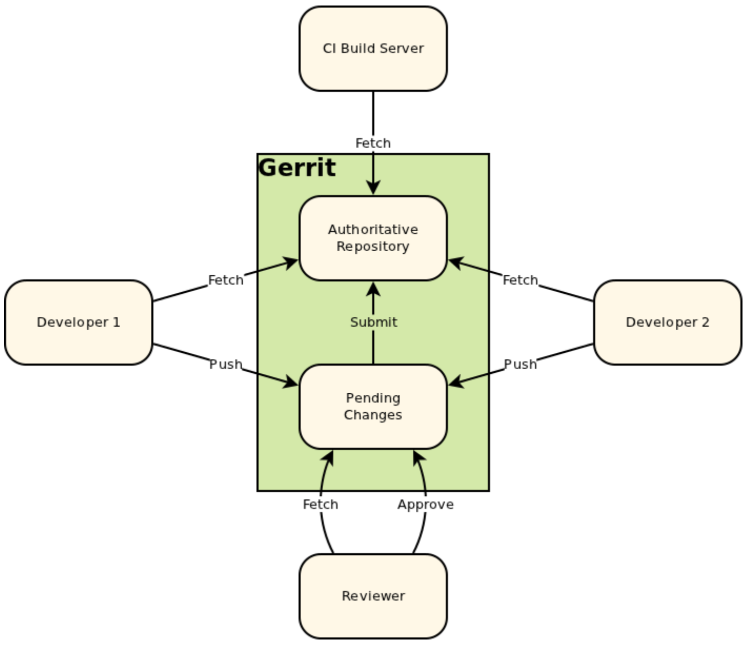


Figure B.3: Gerrit Code Review [60]

Bibliography

[1] N. Abbas, A. M. Gravell, and G. B. Wills. "Historical roots of agile methods: Where did "Agile thinking" come from?" In: *Lecture Notes in Business Information Processing* 9 LNBIP (2008), pp. 94–103.

[2] H. Abelson. *Structure and Interpretation of Computer Programs.* 2nd ed. The MIT Press, 1996, p. 683.

[3] A. Abran. "ISO / IEC SQuaRE . The second generation of standards for software product quality". In: *Development* (2003), pp. 1–11.

[4] A. Abran et al. *Guide to the software engineering body of knowledge (swebok).* IEEE Computer Society, 2004, p. 204.

[5] S. Ambler. *Just Barely Good Enough.* URL: `http://www.agilemodeling.com/essays/barelyGoodEnough.html` (visited on 01/08/2016).

[6] S. Ambler. "Quality in an Agile World". In: *SQP* 7.4 (Sept. 2005), pp. 34–40.

[7] S. Anderson. "Mutation Testing". In: *School of Informatics* (2011), pp. 1–25.

[8] O. Arafat and D. Riehle. "The comment density of open source software code". In: *Software Engineering-Companion Volume.* 31st International Conference on. IEEE, 2009, pp. 195–198.

[9] Asetechs. *KRIS Measurement : Intepreting the Halstead measures.* URL: `http://asetechs.com/NewSite2016/Products/Interpreting_Halstead_metrics.htm` (visited on 04/03/2016).

[10] D. Athanasopoulos and A. V. Zarras. "Fine-grained metrics of cohesion lack for service interfaces". In: *Proceedings - 2011 IEEE 9th International Conference on Web Services, ICWS 2011* (2011), pp. 588–595.

[11] R. Atkinson. "Project management: cost time and quality two best guesses and a phenomenon, it's time to accept other success criteria". In: *International Journal of Project Management* 17.6 (1999), pp. 337–342.

[12] R. A. Baeza-Yates. "Algorithms for string searching". In: *ACM SIGIR Forum* 23.3-4 (1989), pp. 34–58.

[13] N. Balaji, N. Shivakumar, and V. V. Ananth. "Software cost estimation using function point with non algorithmic approach". In: *Global Journal of Computer Science and Technology* 13.8 (2013).

[14] M. Banning. "Approaches to Teaching:Current Opinions and Related Research". In: *Nurse Education Today* 25.7 (2005), pp. 502–508.

[15] V. R. Basili and C. Larman. "Iterative and Incremental Development :" in: *IEEE Computer Society* June (2003), pp. 47–56.

[16] I. D. Baxter et al. "Clone detection using abstract syntax trees". In: *14th IEEE International Conference on Software Maintenance (ICSM'98)* 98 (1998), pp. 368–377.

[17] K. Beck. *Extreme Programming Explained: Embrace Change.* 2nd ed. Addison Wesley, 2004, p. 224.

[18] K. Beck. *Test Driven Development: By Example.* 1 Edition. Addison-Wesley Professional, 2002, p. 240.

[19] Berger-Levrault. *Berger-Levrault.* URL: `http://www.berger-levrault.com/` (visited on 07/20/2016).

[20] Berger-Levrault. *e sedit RH.* URL: `http://www.berger-levrault.com/solutions/fonction-publique-territoriale/gestion-ressources-humaines/rh-grandes-collectivites.html` (visited on 07/17/2016).

[21] B. Boehm. "Get Ready for Agile Methods". In: *IEEE Computer* (2002), pp. 2–7.

[22] B. Boehm. "Software and Its Impact : A Quantitative Assessment". In: *Datamation* (1971), p. 41.

[23] B. Boehm. "Spiral Model of Software Development and Enhancement." In: *Computer* 21.5 (1988), pp. 61–72.

[24] B. Boehm and V. Basili. "Software Defect Reduction Top 10 List". In: *Computer* 34.1 (2001), pp. 135–137.

[25] B. Boehm et al. "Cost models for future software life cycle processes: COCOMO 2.0". In: *Annals of Software Engineering* 1 (1995), pp. 57–94.

[26] B. W. Boehm, J. R. Brown, and M. Lipow. "Quantitative evaluation of software quality". In: *Proceedings of the 2nd international conference on Software engineering* (1976), pp. 592–605.

[27] H. P. Breivold. "Software Architecture Evolution through Evolvability Analysis Hongyu Pei Breivold". PhD thesis. Mälardalen University, 2011, p. 243.

[28] L. C. Briand, J. W. Daly, and J. Wüst. "A unified framework for cohesion measurement in object-oriented systems". In: *Empirical Software Engineering* 3.1 (1998), pp. 65–117.

[29] F. Brooks. "Mythical Man-Month". In: *The Mythical Man-Month.* Reading, MA: Addison Wesley Publishing Company, 1975, pp. 44–52.

[30] F. Brooks. *No silver bullet.* April, 1987.

[31] R. Buse and W. Weimer. "Learning a Metric for Code Readability". In: *TSE SPECIAL ISSUE ON THE ISSTA* Best Paper.TSE Special Issue (2008).

[32] J. P. Cavano and J. a. McCall. "A framework for the measurement of software quality". In: *ACM SIGSOFT Software Engineering Notes* 3.5 (1978), pp. 133–139.

[33] Centers for Medicare & Medicaid Services. "Selecting a development approach". In: *Centers for Medicare & Medicaid Services* (2008), pp. 1–10.

[34] S. M. Chandrika, E. S. Babu, and N. Srikanth. "Conceptual Cohesion of Classes in Object Oriented Systems". In: *International Journal of Computer Science and Telecommunications* 2.4 (2011), pp. 38–44.

[35] S. Chidamber and C. Kemerer. "MetricForOOD_ChidamberKemerer94.pdf". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493.

[36] CISQ. *CISQ Code Quality Standards | CISQ - Consortium for IT Software Quality.* URL: `http://it-cisq.org/standards/` (visited on 03/27/2016).

[37] R. Cloutier et al. "System Architecture Entropy". In: *Systems Engineering* (2002).

[38] A. Cockburn and J. Highsmith. "Agile Software Development:The People Factor". In: *Computer* 34.11 (2001), pp. 131–133.

[39] A. Cockburn and L. Williams. "The costs and benefits of pair programming". In: *Extreme programming examined* (2000), pp. 223–247.

[40] *Coding Katas.* URL: `http://www.codekatas.org/` (visited on 08/21/2016).

[41] S. Cornett. *Code Coverage Analysis.* URL: `http://www.bullseye.com/coverage.html` (visited on 04/04/2016).

[42] M. Cortazzi et al. "Sharing learning through narrative communication." In: *International journal of language & communication disorders / Royal College of Speech & Language Therapists* 36 Suppl (2001), pp. 252–257.

[43] S. Counsell, R. Harison, and R. Nithi. "An Overview of Object-Oriented Design Metrics". In: *Software Technology and Engineering Practice* (1997).

[44] O. Dictionary. *metric - definition of metric in English from the Oxford dictionary.* URL: `http://www.oxforddictionaries.com/definition/english/metric` (visited on 04/02/2016).

[45] O. Dictionary. *quality - definition of quality in English from the Oxford dictionary.* URL: `http://www.oxforddictionaries.com/definition/english/quality` (visited on 04/02/2016).

[46] E. Dijkstra. *Wikiquote.* URL: `https://en.wikiquote.org/wiki/Edsger_W._Dijkstra` (visited on 04/18/2016).

[47] E. W. Dijkstra. "The Humble Programmer". In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[48] Docker.com. *Docker.* URL: `https://www.docker.com/` (visited on 08/22/2016).

[49] J. Dreyfuss. *The Ultimate JSON Library.* 2015. URL: `http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/` (visited on 08/23/2016).

[50] J. Eder et al. "Coupling and Cohesion in Object-Oriented Systems". In: *Technical Report, University of Klagenfurt, Austria* 1 (1994). eprint: `9605103` (cs).

[51] N. E. Fenton and M. Neil. "Software Metrics: Successes, Failures and New Directions". In: *The Journal of Systems and Software* 8.Special Issue (1998), pp. 1–19.

[52] M. Fowler. *AssertionFreeTesting.* 2004. URL: `http://martinfowler.com/bliki/AssertionFreeTesting.html` (visited on 04/09/2016).

[53] M. Fowler. *GUI Architectures.* 2006. URL: `http://martinfowler.com/eaaDev/uiArchs.html` (visited on 08/24/2016).

[54] M. Fowler. *Refactoring: Improving the Design of Existing Code.* 1st Editio. Addison-Wesley Professional; 1999, p. 464.

[55] M. Fowler. *TechnicalDebt.* 2003. URL: `http://martinfowler.com/bliki/TechnicalDebt.html` (visited on 03/01/2016).

[56] M. Fowler. *The New Methodology.* URL: `http://www.martinfowler.com/articles/newMethodology.html` (visited on 01/08/2016).

[57] S. D. Fraser et al. "“No Silver Bullet” Reloaded – A Retrospective on “Essence and Accidents of Software Engineering” Steven". In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.* (2007), pp. 1026–1030.

[58] G. Galilei. *Galileo Galilei Quote.* URL: `http://www.brainyquote.com/quotes/quotes/g/galileogal381325.html` (visited on 04/24/2016).

[59] D. a. Garvin. "What Does “Product Quality” Really Mean?" In: *Sloan Management Review* 26.1 (1984), pp. 25–43.

[60] *Gerrit Code Review.* URL: `https://gerrit-documentation.storage.googleapis.com/Documentation/2.12.3/intro-quick.html` (visited on 08/22/2016).

[61] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* 2nd. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002, p. 604.

[62] M. Gladwell. *Outliers: The Story of Success.* Little, Brown, 2008.

[63] Google. *GWT.* URL: `http://www.gwtproject.org/overview.html` (visited on 01/01/2016).

[64] J. Gorman. "OO Design Principles And Metrics OO Design Goals". In: (2006).

[65] M. Goulão and F. Abreu. "Towards a Components Quality Model". In: *Work in Progress Session of the 28th Euromicro Conference (Euromicro 2002), Dortmund, Germany.* (2002).

[66] V. Gupta and J. K. Chhabra. "Package coupling measurement in object-oriented software". In: *Journal of computer science and technology* 24.2 (2009), pp. 273–283.

[67] M. J. Harrold. "Testing: a roadmap". In: *Future of Sofware Engineering* (2000), pp. 61–72.

[68] J. H. Hegeman. "On the Quality of Quality Models". PhD thesis. University of Twente, 2011, p. 151.

[69] A. Hunt and D. Thomas. *The Pragmatic Programmer.* Addison Wesley Longman, Inc., 2000.

[70] A. C. Inkpen. "Creating knowledge through collaboration". In: *California Management Review* 39.1 (1996), pp. 123–140.

[71] T. D. Institute. *The PDSA Cycle.* URL: `https://www.deming.org/theman/theories/pdsacycle` (visited on 01/08/2016).

[72] ISO/IEC. *ISO/IEC 9126-1:2001 - Software engineering.* URL: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749` (visited on 03/27/2016).

[73] ISTQBExamCertification. *What is V-model- advantages, disadvantages and when to use it?* URL: `http://istqbexamcertification.com/what-is-v-`

`model-advantages-disadvantages-and-when-to-use-it/` (visited on 01/01/2016).

[74] J. Jagger. *Do More Deliberate Practice.* 2011. URL: `http://jonjagger.blogspot.fr/2011/02/deliberate-practice.html` (visited on 08/19/2016).

[75] A. Janes and G. Succi. "The Dark Side of Agile Software Development". In: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software* (2012), pp. 215–227.

[76] R. W. Jensen. *Improving Software Development Productivity: Effective Leadership and Quantitative Methods in Software Management.* 2014, pp. 1–368.

[77] C. Jones. "Evaluating Software Metrics and Software Measurement Practices". In: *Namcook Analytics* (2014), pp. 1–100.

[78] E. Juergens, F. Deissenboeck, and B. Hummel. "Code Similarities Beyond Copy And Paste". In: *2010 14th European Conference on Software Maintenance and Reengineering* (2010), pp. 78–87.

[79] H. W. K. Jung, S. G. Chung, and C. Shin. "Measuring software product quality: A survey of ISO/IEC 9126". In: *IEEE Software* 21.5 (2004), pp. 88–92.

[80] A. Y. Kolb and D. A. Kolb. "Learning Styles and Learning Spaces: Enhancing Experiential Learning in Higher Education". In: *Academy of Management Learning And Education* 4.2 (2005), pp. 193–212.

[81] D. A. Kolb. "Experiential Learning: Experience as The Source of Learning and Development". In: *Prentice Hall, Inc.* 1984 (1984), pp. 20–38.

[82] T. C. Lethbridge and R. Laganiere. *Object-oriented software engineering.* McGraw-Hill New York, 2005.

[83] J. L. Letouzey. "The SQALE method for evaluating technical debt". In: *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings* (2012), pp. 31–36.

[84] J.-L. Letouzey and M. Ilkiewicz. "Managing Technical Debt with the SQALE Method". English. In: *IEEE Software* 29.6 (Nov. 2012), pp. 44–51.

[85] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide.* Prentice-Hall, Inc., 1994.

[86] V. Machinery. *The Halstead Metrics.* URL: `http://www.virtualmachinery.com/sidebar2.htm` (visited on 03/28/2016).

[87] T. V. Mahalingam. "The Last Word". In: *Dataquest* 21.23 (2003), p. 112.

[88] Y. K. Malaiya. *Automatic Test Software.* Wiley Encyclopedia of Electrical and Electronics Engineering. 2011.

[89] B. Marick. *How to misuse code coverage.* Tech. rep. Reliable Software Technologies, 1999, pp. 16–18.

[90] D. Marks. "N Cycles Methodologies". In: *N Cycles Software Solutions* (2002).

[91] R. Martin. *Clean Code.* Prentice Hall, 2014, p. 443.

[92] R. C. Martin and M. Micah. *Agile Principles, Patterns, and Practices in C#*. 1st ed. Prentice Hall, 2006, p. 768.

[93] R. C. Martin et al. "Manifesto for Agile Software Development". In: (2001).

[94] S. Mathur and S. Malik. "Advancements in the V-Model". In: *International Journal of Computer Applications* 1.12 (2010), pp. 29–34.

[95] T. J. McCabe. "A complexity measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320.

[96] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997, p. 1300.

[97] B. Meyer. *The origin of "software engineering"*. 2013. URL: `https://bertrandmeyer.com/2013/04/04/the-origin-of-software-engineering/` (visited on 07/23/2016).

[98] R. Mobbs. *David Kolb.* URL: `http://www2.le.ac.uk/departments/gradschool/training/eresources/teaching/theories/kolb` (visited on 08/17/2016).

[99] K. Mordal. "Analyse et conception d'un modèle de qualité logiciel". PhD thesis. Université Vincennes – Saint-Denis – Paris 8, 2012.

[100] K. Mordal, J. Laval, and S. Ducasse. "Modèles de mesure de la qualité des logiciels". In: *Evolution et réenovation des systèmes logiciels* (2011), pp. 1–35.

[101] P. Naur, B. Randell, and J. N. Buxton. *Software engineering: concepts and techniques: proceedings of the NATO conferences.* Petrocelli/Charter, 1976.

[102] S. Nerur, R. Mahapatra, and G. Mangalaraj. "Challenges of Migrating to Agile Methodologies". In: *COMMUNICATIONS OF THE ACM* 48.2 (2005), pp. 66–71.

[103] J. F. Nestojko et al. "Expecting to teach enhances learning and organization of knowledge in free recall of text passages". In: *Memory And Cognition* 42 (2014), pp. 1038–1048.

[104] V. Nguyen et al. "A SLOC Counting Standard". In: *COCOMO II Forum.* 2007 (2007), pp. 1–15.

[105] I. Nonaka. "The Knowledge Creating Company". In: *Harvard Business Review* 69 (1991), p96–104.

[106] I. Nonaka and R. Toyama. "The knowledge-creating theory revisited: knowledge creation as a synthesizing process". In: *Knowledge Management Research & Practice* 1.1 (2003), pp. 2–10.

[107] J. Offutt. "An experimental determination of sufficient mutant operators". In: *ACM Transactions on Software Engineering and Methodology* 5.2 (1996), pp. 99–118.

[108] M. Ortega, M. Pérez, and T. Rojas. "Construction of a systemic quality model for evaluating a software product". In: *Software Quality Journal* 11.3 (2003), pp. 219–242.

[109] G. Philipson. "A Short History of Software". In: *Management, Labour Process and Software Development.* New York: Routledge, 2005.

[110] Pmdocuments. *Agile Scrum Roles and Responsibilities.* 2012. URL: `http://www.pmdocuments.com/2012/09/15/agile-scrum-roles-and-responsibilities/` (visited on 08/18/2016).

[111] R. E. Al-Qutaish and A. Abran. “An analysis of the design and definitions of Halstead’s metrics”. In: 2005, pp. 337–352.

[112] A. Rawashdeh and B. Matalkah. “A New Software Model for Evaluating COTS Components”. In: *Journal of Computer Science* 2.4 (2006), pp. 373–381.

[113] S. Robertson. *GitBestPractices.* URL: `https://sethrobertson.github.io/GitBestPractices/` (visited on 08/09/2016).

[114] L. Rosenberg and L. Hyatt. “Software quality metrics for object-oriented environments”. In: *Crosstalk Journal, April* 10.4 (1997), pp. 1–6.

[115] D. W. W. Royce. “Managing the Development of Large Software Systems”. In: *IEEE Wescon* 26.August (1970), pp. 1–9.

[116] K. Schwaber and J. Sutherland. “The Scrum Guide”. In: *Scrum.Org and ScrumInc* July (2013), p. 17.

[117] K. Schwaber and J. Sutherland. “Scrum development process”. In: *Business Object Design and Implementation* (1997), pp. 117–134.

[118] M. Scotto et al. “A relational approach to software metrics”. In: *Proceedings of the 2004 ACM symposium on Applied computing SAC 04* (2004), p. 1536.

[119] Selenium. *Selenium - Web Browser Automation.* URL: `http://www.seleniumhq.org/` (visited on 04/23/2016).

[120] A. Serebrenik. “Software metrics”. In: *2IS55 Software Evolution.* Technische Universiteit Eindhoven, 2011.

[121] M. Shepperd and D. Ince. “A critique of three metrics”. en. In: *Journal of Systems and Software* 26.3 (Sept. 1994), pp. 197–210.

[122] P. Smacchia. *Code metrics on Coupling, Dead Code, Design flaws and Re-engineering.* 2008. URL: `http://codebetter.com/patricksmacchia/2008/02/15/code-metrics-on-coupling-dead-code-design-flaws-and-re-engineering/` (visited on 04/12/2016).

[123] R. M. Soley. “How to Deliver Resilient , Secure , Efficient , and Easily Changed IT Systems in Line with CISQ Recommendations”. In: *OMG* (2012), pp. 1–13.

[124] SonarQube. *Metrics - Complexity - SonarQube Documentation - SonarQube.* URL: `http://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity` (visited on 04/03/2016).

[125] SonarQube. *SonarQube - Cyclomatic Complexity.* URL: `https://sonar43.spring.io/rules/show/checkstyle:com.puppycrawl.tools.checkstyle.checks.metrics.CyclomaticComplexityCheck?layout=false` (visited on 04/03/2016).

[126] SonarQube. *SonarQube Discussing Cyclomatic Complexity.* URL: `http://www.sonarqube.org/discussing-cyclomatic-complexity/` (visited on 04/03/2016).

[127] SonarQube. *SonarQube Manage Duplicated Code with Sonar.* URL: `http://www.sonarqube.org/manage-duplicated-code-with-sonar/` (visited on 03/28/2016).

[128] SonarSource. *SonarQube.* URL: `http://www.sonarqube.org/` (visited on 08/22/2016).

[129] SonarSource. *Technical Debt Evaluation (SQALE) | SonarSource.* URL: `http://www.sonarsource.com/products/plugins/governance/sqale/` (visited on 04/19/2016).

[130] Sophos. *Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X!* 2014. URL: `https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/` (visited on 04/18/2016).

[131] SQALE. *SQALE | Software Quality Assessment based on Lifecycle Expectations.* URL: `http://www.sqale.org/` (visited on 04/19/2016).

[132] W. Stevens and M. G. "Structured Design". In: *IBM Systems Journal* 13.2 (1974), pp. 115–139.

[133] I. Sun Microsystems. *Code Conventions for the Java Programming Language: 1. Introduction.* 1999. URL: `http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html` (visited on 04/18/2016).

[134] J. V. Sutherland et al. "Business Object Design and Implementation". In: *OOPSLA'95 Workshop Proceedings 16 October 1995, Austin, Texas.* 1995.

[135] H. Takeuchi and I. Nonaka. "The New New Product Development Game". In: *Journal of Product Innovation Management* 3.3 (1986), pp. 205–206.

[136] D. Tegarden, S. Sheetz, and D. Monarchi. "Effectiveness of traditional software metrics for object-oriented\nsystems". In: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences* iv (1992), pp. 359–368.

[137] UMKC. *Software Quality Management.* URL: `http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/` (visited on 03/27/2016).

[138] Verifysoft. *Halstead Metrics.* URL: `http://www.verifysoft.com/en_halstead_metrics.html` (visited on 04/03/2016).

[139] M. El-Wakil. "Object-oriented design quality models a survey and comparison". In: *... on Informatics and ...* (2004), pp. 1–11.

[140] M. D. Weiser, J. D. Gannon, and P. R. McMullin. "Comparison of structural test coverage metrics". In: *IEEE Software* 2.2 (1985), p. 80.

[141] Wikipedia. *Apache Maven.* URL: `https://en.wikipedia.org/wiki/Apache_Maven` (visited on 08/22/2016).

[142] Wikipedia. *Code coverage.* en. 2015. URL: `https://en.wikipedia.org/w/index.php?title=Code_coverage` (visited on 04/04/2016).

[143] Wikipedia. *Didactic Method.* URL: `https://en.wikipedia.org/wiki/Didactic_method` (visited on 08/17/2016).

[144] Wikipedia. *Experiential Learning.* URL: `https://en.wikipedia.org/wiki/Experiential_learning` (visited on 08/17/2016).

[145] Wikipedia. *Extreme programming.* URL: `https://en.wikipedia.org/wiki/Extreme_programming` (visited on 08/22/2016).

[146] Wikipedia. *Knowledge Management.* URL: `https://en.wikipedia.org/wiki/Knowledge_management` (visited on 08/15/2016).

[147] Wikipedia. *Multitier architecture.* URL: `https://en.wikipedia.org/wiki/Multitier_architecture` (visited on 07/17/2016).

[148] Wikipedia. *OSGi.* URL: `https://en.wikipedia.org/wiki/OSGi` (visited on 08/22/2016).

[149] Wikipedia. *REST.* URL: `https://en.wikipedia.org/wiki/Representational_state_transfer` (visited on 08/22/2016).

[150] Wikipedia. *Scrum.* URL: `https://en.wikipedia.org/wiki/Scrum_(software_development)` (visited on 07/17/2016).

[151] Wikipedia. *Software development process.* URL: `https://en.wikipedia.org/wiki/Software_development_process` (visited on 01/01/2016).

[152] Wikipedia. *Source lines of code.* en. 2015. URL: `https://en.wikipedia.org/w/index.php?title=Source_lines_of_code` (visited on 04/02/2016).

[153] Wikipedia. *Time Boxing.* URL: `https://en.wikipedia.org/wiki/Timeboxing` (visited on 01/08/2016).

[154] Wikipedia. *V-Model.* URL: `https://en.wikipedia.org/wiki/V-Model_(software_development)` (visited on 07/17/2016).

[155] Wikipedia. *Wiki.* URL: `https://en.wikipedia.org/wiki/Wiki` (visited on 08/22/2016).

[156] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* 1st ed. Prentice-Hall, Inc, 1979, p. 473.

List of Figures

List of Tables

Approche collaborative pour la qualité du code logiciel dans un contexte Agile

Mémoire d'Ingénieur C.N.A.M., Toulouse 2016

Résumé

Les méthodes Agiles ont été inventées entre autre en vu d'une amélioration de la qualité des produits développés. En revanche, même si elles sont appliquées à la lettre, les équipes de développement n'étant pas universellement composées d'experts ni de développeurs conscients, ces méthodes ne portent pas toujours leurs fruits. En effet, le manque de sensibilisation aux avantages que la qualité du code peut apporter et le manque de savoir-faire dans l'évaluation des métriques et principes de qualité du code peut rapidement entraîner une dégradation des qualités internes et une érosion architecturale du logiciel en cours de production. En appliquant une nouvelle approche collaborative, une équipe ayant un savoir-faire moyen peut devenir compétente et produire du code qui manifeste les caractéristiques de qualité souhaitables. Il s'agit ici de décrire cette nouvelle approche basée sur le partage et la capitalisation des connaissances au sein d'une équipe dans un contexte Agile.

Summary

Agile development methodologies adapt to the rapidly changing environment of modern business and continuously deliver working software. Nevertheless, the use of these methodologies requires a highly competent team to produce quality code. Agile development teams are not universally made up of expert developers, and the lack of awareness of the benefits of code quality, of know-how in evaluating quality metrics and of quality principles can result in architectural erosion and a deterioration of the internal qualities of software. By applying a new collaborative approach to code quality and learning within an Agile context, a team of average abilities can produce code that exhibits first-rate quality characteristics and slowly increase their mastery of the domain. This document develops this approach which is based on the cultivation of knowledge and expertise in a collaborative environment.