OULU UNIVERSITY OF
APPLIED SCIENCES

Mikko Pietola

TECHNICAL EXCELLENCE IN AGILE SOFTWARE PROJECTS

TECHNICAL EXCELLENCE IN AGILE SOFTWARE PROJECTS

Mikko Pietola
Master's Thesis
2012
Degree Programme in Information Technology
Oulu University of Applied Sciences

ABSTRACT

OULU UNIVERSITY OF APPLIED SCIENCES ABSTRACT

Degree programme	Thesis	Number of pages
Degree Programme in Information Technology	Master's Thesis	68
Line	Date	
Master of Engineering	25.11.2012	
Commissioned by	Author	
-	Mikko Pietola	

Thesis title
Technical Excellence in Agile Software Projects

Keywords
Agile Software Development, Code reviews, Refactoring, Software Design Principles, Software Design Patters, Software Design Violations

The software industry has evolved over the past decades and it has become one of the most important industries of our time. Competition in the software industry environment is tight. The winners are often first on the market, their products deliver the requisite quality and functionality to their customers and they are able to adapt to business and technological changes rapidly. This means the leaders provide high quality software more quickly.

The agile software development methods were introduced in the last decade to revolutionize software development. The benefits of agility are promoted to be faster time to market, better responsiveness to changing customer requirements and higher software quality. However, the agile methods are mainly focusing on processes, they do not prescribe the technical aspects of software engineering for producing high quality software. The organizations which utilize agile software development processes should emphasize and encourage their personnel to develop individual skills related to practices, principles and patterns of agile software development, because often these skills are not taught at universities. This Master's thesis is a literature study to give an introduction to agile software processes, the properties of technical excellence in agile projects and an introduction to software design principles and patterns which are needed in modern software projects to build high quality software.

TABLE OF CONTENTS

1 INTRODUCTION

Agile software development is a group of light-weight software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Agile development has made its way into the software mainstream in the past years and agile process models have been touted as the methodologies of choice for continuous sustained software delivery.

Even today, over a decade after Agile Manifesto was born, many software development projects are having poor track of record in terms of budget, quality and schedule. Agile itself alone is not a silver-bullet that resolves these problems. The development projects may fail, even when they are managed with agile methods. A root cause for failing projects may be the development team itself; if the team members do not know the technical disciplines and practices required by agile methods then they cannot apply them in practice. Recently, while adapting agile methods, a great deal of efforts are put on software development processes improvement, while forgetting that the people involved in the software development deserve more attention.

The objective of this Master's thesis is to study literature to give an introduction to agile software processes, the properties of technical excellence in agile projects and an introduction to software design principles and patterns which are needed in modern software engineering projects to build high quality software.

2 AGILE SOFTWARE PROCESS MODELS

In the past, many software companies have been concerned about the variety in development time and quality of their software products. The companies built their software products using heavyweight micromanaged waterfall approaches, which included many models, frameworks, patterns and techniques that were supposed to help engineers to produce better software. For most projects the formal approaches introduced only overhead and bureaucracy waste. Hence, the software product development typically took so long to build that their formal requirements had changed long before the systems were delivered.

Lightweight software development methods started to evolve during the mid-1990s as a reaction against formal heavyweight approaches. These methods carried different names and activities, but they all aimed to address the same problem: creating reliable high quality software more quickly, while eliminating unnecessary waste and unproductive overhead. (1, p. 20; 2, p. 8.)

These development methods are:

- Scrum, Ken Schwaber, Jeff Sutherland, 1995
- Dynamic System Development Method (DSDM), Dane Faulkner, 1995
- Crystal, Alistair Cockburn, 1997
- Feature Driven Development (FDD), Peter Code, Jeff DeLuca, 1999
- Extreme Programming (XP), Kent Beck, Eric Gamma, 1999
- Pragmatic Programming, Andrew Hunt, David Thomas,1999
- Kanban, David J. Anderson, 2010

In 2001, many of the agile experts got together and collaborated to understand and agree upon the common philosophies that underlie their various methods. The result of this collaboration became the Agile Manifesto:

> *We are uncovering better ways of developing software by doing it and helping other to do it. Through this work we have come to value:*
>
> - ***Individuals and interactions*** *over processes and tools*
> - ***Working software*** *over comprehensive documentation*
> - ***Customer collaboration*** *over contract negotiation*
> - ***Responding to change*** *over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more.* (3, p. 1.)

Agile recognizes that people are unique individuals instead of replaceable resources and their highest value is in their communication and interaction. Agile requires small teams in which different roles form cross-functional units. These teams are then required to self-organize, meaning that no method or process is imposed to them. The team is trusted to get the work done in ways that they think are the best, assuming they know how to do that. The team is accountable to deliver their results. (1, pp. 22-24; 2, pp. 9-10.)

Agile understands that the best products are created when the customers are directly involved with the teams creating them. A backlog of features is constantly maintained and reprioritized by collaborating with the customer. These features are described in concise format, and more in-depth exploration and documentation starts once the team selects them for implementation. After the implementation is ready, the usefulness of features is immediately verified by the customer. (1, pp. 22-24; 2, pp. 9-13.)

In agile projects software is produced in short time frames, in time boxes or “sprints”, and delivered in many incremental releases, where each release is a potentially shippable product. This enables business owners to take control over

timing by moving release dates, depending on what features they want to make available in certain releases. Frequent product releases are meant to invite feedback from the customer and to provide new and updated features to the customer as soon as the need is detected. (1, pp. 22-24; 2, pp. 9-13.)

2.1 Scrum

Scrum is a lightweight agile project management method based on small, empowered, self-organizing teams, complete visibility and rapid adaptation. Scrum contains a set of methods and predefined roles. The management of a scrum-based process is tactically intensive, prescriptive and role-based. The main roles in scrum are: product owner, scrum master and development team. The product owner is responsible for representing the interests of customers and other key stakeholders on the team. The product owner does this by managing the product backlog, which is a prioritized list of requirements and other work to be done by the development team. The scrum master is responsible for helping development team to achieve its goals, for teaching scrum to everyone on the team, and for implementing scrum practices and rules. The scrum master is also responsible for resolving any impediments that development team has. The development team is responsible for implementing the functionality. The team members, including developers, testers and all other related personnel, are required to fully implement and deliver the functionality. The team is self-organizing, self-managing, and cross-functional. (2, pp. 41-43.)

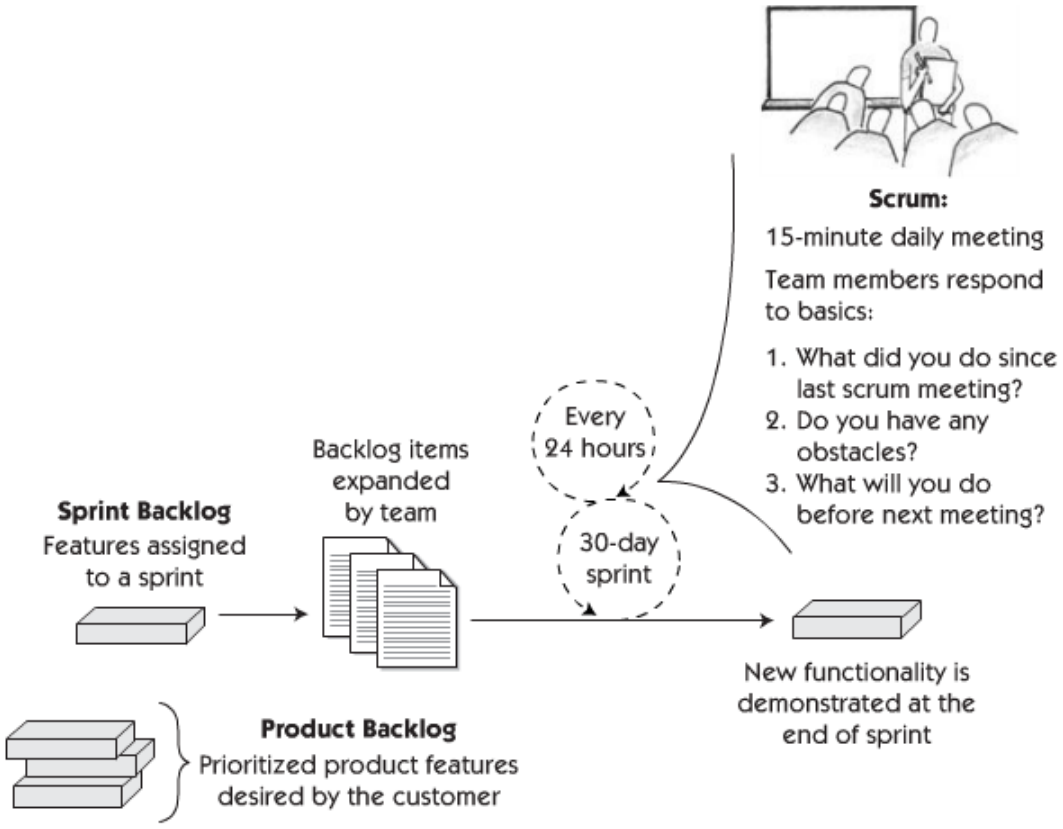


FIGURE 1. Scrum process overview (2, p. 47)

The key practices of a scrum process (Figure 1) includes the following; Cross-functional and collocated teams of eight or fewer team members develop software in sprints. The sprints are fixed length iterations, whose length is usually 2-4 weeks. Each sprint should deliver incremental, tested functionality of value to the customer. The product backlog is the source for a sprint planning meeting. Once the content of the sprint is committed in the sprint planning meeting, no additional functionality can be added, expect by the development team. This means that the work within the sprint is fixed. The scrum master mentors and steers the self-organizing and self-managing development team during the sprint. All work in the sprint is carried through sprint backlog, which includes requirements to be delivered, defects to be fixed as well as other infrastructure and design activities to be done. A daily 15-minute stand-up meeting, daily scrum, is held by the development team and is the primary communication method. A sprint review meeting is held at the end of each sprint, where the development team demonstrates new functionality for the

product owner and customers. A sprint retrospective meeting is held after each sprint, where the development team is asked two questions; what went well during the sprint and what should be improved for the next sprint. The main purpose of the sprint retrospective meeting is to improve the overall development process. (2, pp. 44-47.)

2.2 Extreme Programming

Extreme programming (XP) is a software development methodology invented by Kent Beck and it is a collection of practices, principles and values. XP shares the values espoused by the Agile Manifesto and it echoes scrum process methodology. Nowadays the principles and values are not widely adopted, while the practices are commonly used. The main purpose of XP is to organize people so that they produce higher quality software more productively.

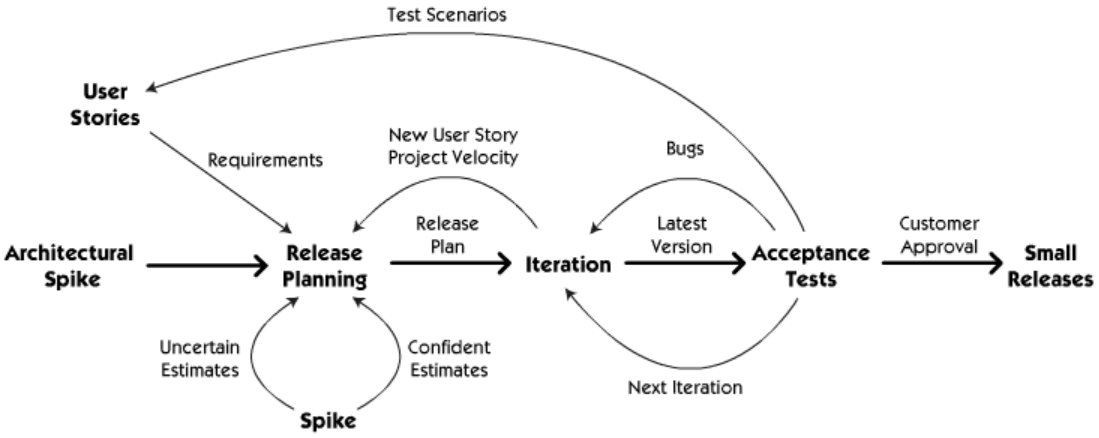


FIGURE 2. XP process model (2, p. 39)

The XP process model is shown in Figure 2. XP defines a set of practices that are common and applied to each project using the method. The key characteristics of the XP practices include the following (2, pp. 34-38; 4, p. 1.):

- *Whole Team.* In order to deliver software reliable in small releases, the software has to be defined, implemented and tested frequently. Therefore, the XP team consists of developers, quality assurance, project managers, business analysts and any other stakeholders whose skills are needed to be present in the team to deliver software. The team works together in a common and open workspace to maximize collaboration and communication. The workspace is informative; the stories under implementation are pinned to story boards (or story walls) for all to see. The story statuses and responsible developers are updated at least daily, thus everyone from the developers to the managers can assess the iteration status simply walking through the team area.
- *Planning Game.* Prior to each iteration features are broken into small stories. The stories are the unit of the functionality in XP. The stories are flexible and they can be modified during implementation or re-prioritized for a later time. The stories and architectural spikes are the main input of the release planning. The developers estimate the (cost of) stories in iteration planning and stories are then chosen to iteration by the stakeholders based on their estimated cost and business value. The sum of the selected stories work effort estimates cannot exceed the sum of estimates completed in the previous iteration.
- *Acceptance Tests.* The stories and features are defined by automated tests written by business analysts and quality assurance. When a story or feature passes the acceptance test suite, the story is said to be done.
- *Small Releases.* Software systems are released to production (or preproduction) very often. The minimum release cycle can be once per iteration and the maximum cycle is usually in quarterly cycles.
- *Continuous Integration.* In XP, the whole system is built and all changes are integrated and tested several times each day (or at least daily in the worst case). Usually, a build is triggered by a commit to version control. Developers must keep the system in a continuously deployable state. This means that build (or test) breaks are not allowed, i.e. the code must

compile and link and all the previously passing tests must pass after every commit.

- *Collective Ownership.* The code and all other work artifacts are owned by the team instead of individuals. Any team member can work and change any artifact at any time.
- *Coding Standard.* The code and all other work artifacts look as if they were written by the team. Each team member follows the team standard for format and appearance for artifacts.
- *Metaphor.* The names within code and other work artifacts are chosen to be evocative of the system being developed.
- *Sustainable Pace.* Working beyond 40 hours a week is considered to be nonproductive, because XP is intensive. Tired people are unproductive, thus the overtime must be carefully controlled and limited.
- *Pair Programming.* Code and other artifacts are produced by pairs working together on one machine. One member of the pair is responsible for the task (and keyboard) at hand while the role of the other member is to observe and help. Pair programming provides immediate feedback and peer review of the code.
- *Test Driven Development.* Developers are not allowed to write the production code before they have written a failing unit/automation test. The test first paradigm has three advantages:
    1. The developers have to understand the functionality of the test, which forces to understand the actual story (requirement) before implementing it.
    2. The test automation is built-in; the team will never have to catch up later on the test automation.
    3. The quality is assured because all code is tested code.
- *Refactoring.* XP teams invest a little in design every day. Code and other artifacts are continuously reviewed by pair programming and kept as clean as possible.

- *Simple design.* The system is always left with the simplest design that supports its current functionality, i.e. the simplest thing that could possibly work.

2.3 Software Craftsmanship

Although the Agile software development approach and processes have been adopted over the past years, still many software development projects are having a poor track of record in terms of schedule, budget and quality. The Agile Manifesto neither explicitly recognizes that all people who are involved in a development project have to be skilled, disciplined, smart and attentive, nor explicitly defines the technical practices to be used by a development team.

In other words, thinking and talking Agile does not make projects to succeed. Agile is great when the development team is great, and to have a great team, each individual must admit that they should actively change their attitudes and behaviors to improve their engineering skills in order to achieve better results. (1, pp. 196-197.)

The Software Craftsmanship is an approach which emphasizes the software development skills of the software engineers themselves. This is formed in the Manifesto for Software Craftsmanship:

> *As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping other learn the craft. Through this work we have come to value:*
>
> - *Not only working software, but also* ***well-crafted software***
> - *Not only responding to change, but also* ***steadily adding value***
> - *Not only individuals and interactions, but also* ***a community of professionals***
> - *Not only customer collaboration, but also* ***productive partnership***

> *That is, in pursuit of the items on the left we have found the items on the right to be indispensable.* (5, p. 1.)

The manifesto for Software Craftsmanship is said to both challenge and extend the original Agile Manifesto. It promotes the values of well-crafted and clean software which is produced by craftsmen software developers. The pragmatic methodology is described by Hunt et all [6] and it relates closely to software craftsmanship values. Hunt lists the following characteristics for pragmatic programmers:

- Early adopter/fast adapter
- Inquisitive
- Critical thinker
- Realistic
- Jack of all trades

Pragmatic programmers have an instinct for technologies, techniques and environments and they want to try new things out, hence their confidence is based on experience. They do not accept things as given without getting the facts first and they want to understand the nature of the underlying problems to be able to realistically evaluate how difficult things are before giving any promises. Most importantly, they care about their craft as there is no point in developing software unless doing it well. (6, pp. 28-29.)

3 AGILE TECHNICAL EXCELLENCE

Technical excellence in agile is achieved through Test-Driven Development (TDD), code reviews, Definition-of-Done (DoD), iterative development and refactoring together with a fully tested code. The software architecture is not defined up-front; instead it is allowed to emerge while developing a product. Daily builds, continuous integration and automated testing tools are in a supporting role when developing and delivering successfully high quality products. (1, pp. 22-23; 2, pp. 158-159.)

3.1 Test-Driven Development

Test-Driven Development is a software design approach where unit tests are written before the production code. TDD turns traditional software development around as it enforces software developers to write automated unit tests. TDD also requires developers to think through the requirements and code design before starting the implementation of production code. Hence, it is said to be an agile requirements' and agile design technique. TDD software development approach requires discipline because developers may find it easy to "slip" directly to writing of the production code without first writing a new test. The slippage can be avoided by Definition-of-Done which requires unit tests to be in place and by pair programming where pairs help each other to stay on track. (2, pp. 158-159.)

TDD workflow can be divided into three main states are (Figure 3):

- Writing an automated unit test.
- Running the test(s).
- Implementation of production code.

The division of writing a unit test and production code is described by the three laws of TDD (7, pp. 122-123; 8, p.1.):

1. Production code may not be written until a failing unit test is written
2. Unit test may not be written more than is sufficient to fail, and compiling is not failing
3. Production code may not be written more than is sufficient to pass the currently failing test

This means that implementation starts by writing an automated unit test together with enough production code implemented to make the test fail. Once the test is in place, the production code is implemented and improved step by step to pass the test. Once the tests pass, the next step is to start over for next unit / module.

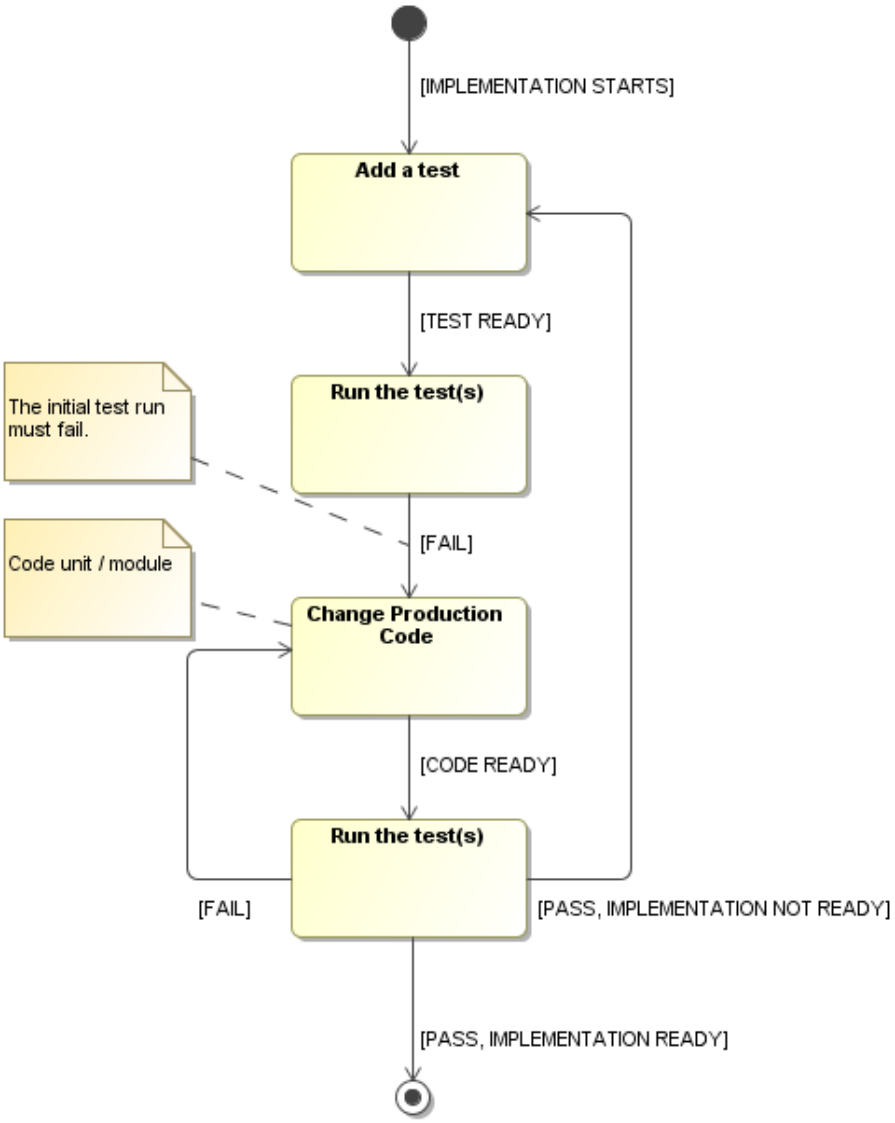


FIGURE 3. TDD workflow

3.2 Code Reviews

Code review is a process where software developers review each other's code to find out programming errors from the source code and to ensure that the source code meets the agreed quality criteria before committing the changes to the version control system. Code review is a method to find errors as early as possible in the development phase, to prevent error leakage to QA and to

product release. The most commonly used light-weight review types are (9, p. 2; 10, pp. 23-38.):

- Peer review
- Pair programming
- Email pass-around review
- Code inspection

These review types are used as to identify bugs from the source code, encourage collaboration between the developers and improve the skills of the developers by learning from peers. A code review workflow is shown in Figure 4.

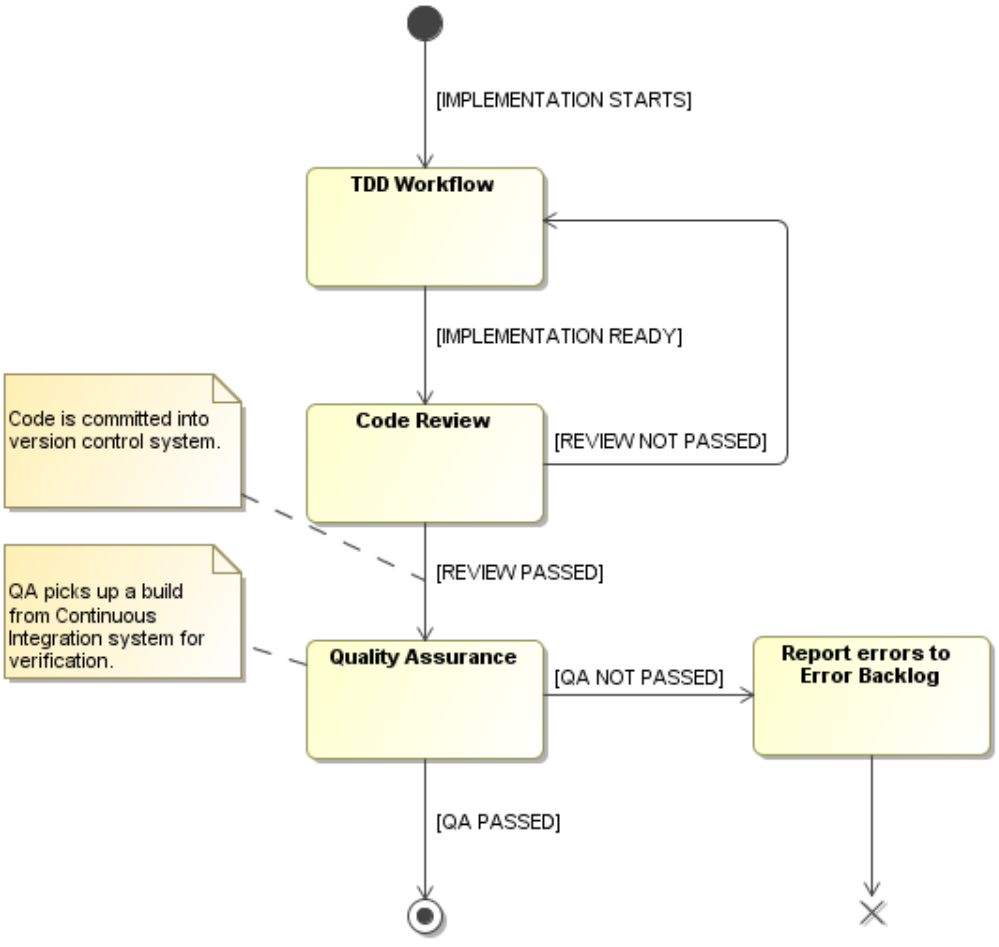


FIGURE 4. Code review workflow

The roles in the code review are Reviewer and Reviewee. The Reviewer is the developer who reviews the code and is responsible for checking the code and identifying possible problems. The Reviewer should ensure that the code under review conform the points described in the agreed Review and Definition of Done Checklists. This usually includes at least the following (10, pp. 5-6.):

- Coding standard
- Conformance to design
- Code clarity and maintainability
- Coding errors

If the problems are identified, they must be pointed out to the Reviewee to get them fixed. The Reviewer is not responsible for providing solutions for the issues found. Once the identified issues are fixed, a new review is held to ensure that defects are actually fixed. (9, pp. 28-30; 10, pp. 5-6.)

The Reviewee is the developer who wrote the code. This person is responsible to walk through the changed code. This means usually introducing the changed code together with explanation what the code is responsible for. If possible, new functionality should be demonstrated. Finally, the Reviewee is responsible that the code under inspection fulfills the points defined in Definition of Done checklist. (9, pp. 28-30; 10, pp. 5-6.)

More heavy-weight inspection processes also exist. An example is Fagan inspection where the inspection concentrates to several artifacts ranging from the requirement documentation, test plan, architectural design to code. The Fagan inspection process has multiple phases; planning, overview meeting, preparation, inspection meeting, rework and follow-up verification. Because of this, these are not usually used as part of agile processes because of their high-formality and high-cost. (10, p. 8.)

3.3 Definition of Done

The Definition of Done (DoD) is an agreement between the product owner and the team about what “done” means. The DoD is an important collection of valuable deliverables required to produce software. The collection of deliverables is measurable units of work to be done to complete a certain work package (user story, sprint or release). When DoD is defined, agreed and followed, it can be used to ensure that a work is really done. The main high level purpose of DoD is to ensure that the agile development project is delivering real value to the customer; in other words ensure the product is potentially shippable. The Definition of Done can be divided into three main categories for integrity and completeness (11, pp. 170-171; 12, p.1.):

1. User Story (or feature) DoD
    - Production level code
        - Code meets coding conventions as defined in the project code style guide
    - Code review
        - Peer-to-peer code review held or code is produced with pair-programming
    - Unit testing
        - Unit tests are implemented for all new functionality
        - Unit tests are integrated to the test automation system
        - Unit tests are in green (all tests pass)
    - Acceptance testing
        - Acceptance tests cover all new functionality
        - All acceptance criteria of the user story are met
    - Static code analysis (e.g. Lint)
        - No high warnings
        - All warnings are analyzed
    - Code integrates against the latest target environment
        - No compiler errors nor warnings

        - Integration tests of affected areas are conducted and passed
    - Code is committed into version control system
    - Documentation is updated
    - User story is ready to be demonstrated in sprint review meeting
2. Sprint DoD
    - All User Stories accepted for Sprint are done and they all meet the User Story DoD
    - All new code is integrated into the latest product baseline to form a sprint release build
        - The agreed development freeze date has been met
        - Integration testing has been done
        - Regression testing has been done
    - Non-functional testing is done
        - Branch coverage is measured to be over 70%
        - Function coverage is measured to be 100%
        - Code cyclomatic (conditional) complexity is measured and meets the agreed limits
        - No memory nor resource leaks
        - Memory consumption is measured and meets the agreed limits
        - Performance items are measured and meet the agreed limits
        - Applicable reliability items are measured and meet the agreed limits
    - No critical (or higher priority) errors exists in the error backlog
    - Sprint release note exists

3. Release DoD
    - All User Stories accepted for Release are done and they all meet the User Story DoD

- Release functional and non-functional testing is done
- No critical (or release blocking) errors exists in the error backlog
- Product is ready for release deployment
    - Formal release date defined
    - Deployment documentation exists
    - Release documentation exists

3.4 Iterative Development

The basic idea in agile methods is to develop software in small repeated cycles (sprints) and in small portions (user stories). Iteration allows the team to create a working, tested and value-delivered code in a short time box. In addition, iteration allows the development team to take advantage of what was learned during the development of the previous iteration. Learning comes from both the development and the use of the system. (2, pp. 123-124.)

All iterations have the same pattern, which gives the heartbeat of manufacturing-like routine to agile development. An iteration consists of three phases; Iteration planning, Development and Delivery as shown in Figure 5.

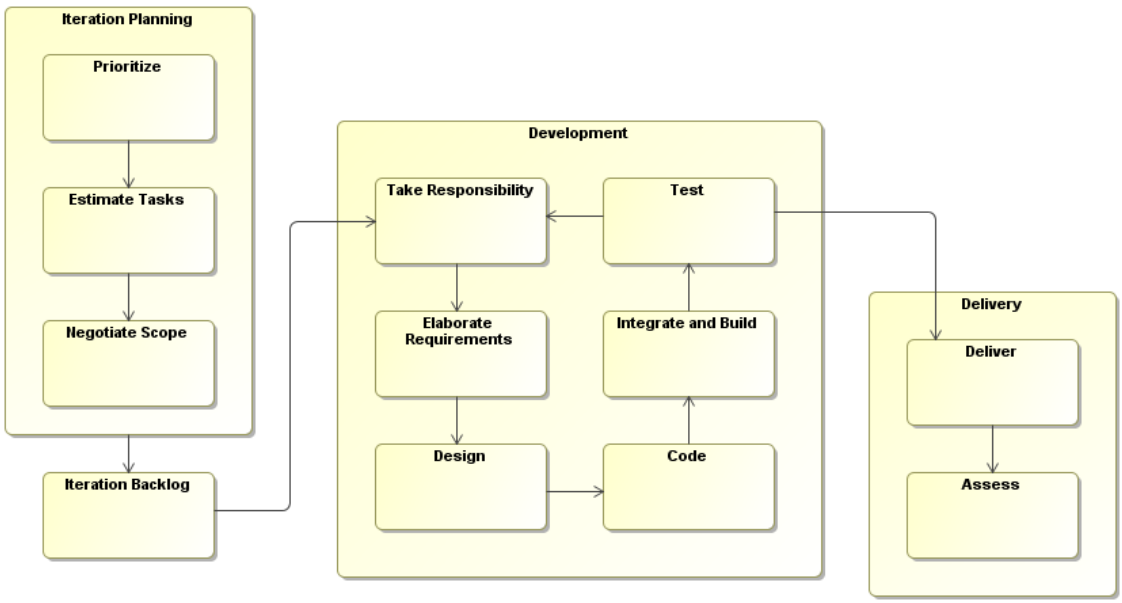


FIGURE 5. Iteration process model

Each iteration starts with iteration planning. The primary goal of the iteration planning is to define and accept a reasonable scope for the iteration. The development team is called for an iteration planning meeting where the team reviews the prioritized product backlog items and then selects the items (user stories) for the current iteration (sprint) by priority order. The development team defines and estimates the list of engineering tasks necessary to deliver the increment of work and then estimates the needed work amount to complete the tasks. The result of iteration planning is the iteration plan on which the development team commits. The plan contains following items (2, p. 127; 11, pp. 155-156.):

- An iteration backlog which contains a list of stories to work on iteration.
- An iteration theme which describes what the iteration is intended to accomplish.
- Estimated tasks for stories. Each task has a responsible developer.
- Documentation of the plan is available on visible place, usually a task/story board on development team's room/area.

The iteration execution, the Development phase, starts when the team has agreed and committed to the iteration plan. Each developer follows the same process through the iteration by repeating the development steps until there is no more work left in iteration backlog. The development phase steps include (2, pp. 129-130.):

- Take responsibility
- Develop
- Deliver
- Declare story completion

The status of development phase is usually visualized on a sprint story (or task) board as shown in Figure 6. A story board contains all the accepted stories (and the tasks which are needed to be accomplished to deliver the story) for the iteration. A story board is often divided into the following columns: Not started,

In progress, Impeded, Ready for Review, Ready for QA and Complete. The stories are moving column by column from the left side of the board to the right as the stories progress. Different tasks, such as user story, bug fix, QA, etc., have usually a dedicated post-it note color to make the recognizing of different tasks on the board easy. To promote and visualize the responsibility, the developer attaches his/hers own photo on top of the task. The project management can follow the iteration status easily by assessing the story board.

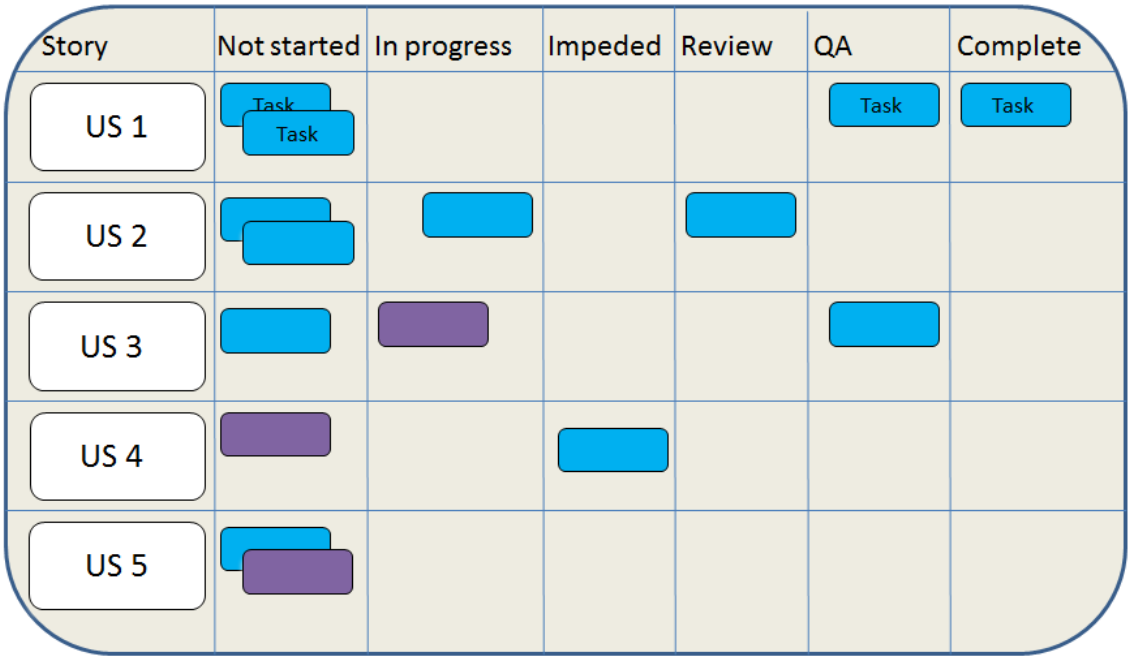


FIGURE 6. A sprint story board

Taking responsibility means that a developer without a task chooses the highest priority task from a not-started column as a task to work with. Usually, new tasks are selected on a daily standup meeting. The developers who already had a task in progress share the implementation status to other team members and move the task card accordingly on the board (it either stays in progress or moves to one of the left side columns).

When a developer has a task in progress, s/he works to resolve the task by:

- Elaborating requirements
- Designing software

- Implementing unit tests and production code
- Integrating the code into a build system

The elaborating requirements and software design are problem solving phases where the requirements are analyzed and the software architecture is designed and validated to form a software solution for requirements. The characteristics of good software design are described in Chapter 4. On the other hand, developers are often required to work with the existing software and to build new requirements into the existing modules. To work efficiently with the existing code, the developers must know how to refactor the code. Refactoring is discussed in more detail in Section 3.5, while the design principle violations (also known as code smells), which are surface indications for a refactoring need, are described in Chapter 6.

The implementation phase starts when the software design is approved and thought to be good enough to solve the problem in question. The implementation follows the TDD workflow as shown in Figure 3. Software redesign may occur after the first (prototype) version of the implemented solution is reviewed.

All of the above activities are usually repeated multiple times for each task in a story until the story implementation is ready and the quality assurance has verified the implemented behavior to match the story acceptance criteria. When the implemented solution is verified, then the backlog item is declared as complete.

The iteration outcome is assessed at the end of the sprint in a sprint review meeting where the team demonstrates the working software to stakeholders, usually the product owners, customers, and, if possible, to the end users. This allows the team to get feedback and guidance as early as possible to make the system as valuable as possible. The feedback is based on the following:

- Presentation of each completed story
- Feedback from stakeholders based on presentation

- Story acceptance based on acceptance criteria and definition of done

The unaccepted stories are reworked in the next iteration and the unfinished stories are put back to the release backlog.

3.5 Refactoring

As a software system evolves, it usually requires re-evaluation of the earlier decisions and rework of the portions of the existing code. Rewriting, reworking or re-architecting of the existing code is known as refactoring. Refactoring is a disciplined way to improve the quality of the software system properties, such as design, reliability, extensibility, modularity, reusability, maintainability, and efficiency. Refactoring does not usually target the whole software rewrite. Instead, it aims to a series of small structural modifications, supported by unit tests to make the code easier to change, and to minimize the chances of introducing bugs. A change is a refactoring only if it does not change the software system's observable behavior. (13, pp. 53-57; 14, pp. 3-8.)

When a developer writes a new code, the code is usually written with a decent quality. However, one of the problems is that the developers believe their primary goal is to get the program code "working", and when this state is reached, the developer moves on to the next task. The developer did not care to finish his/her implementation by doing successive refinements to support the maintenance / development work carried out by other developers in the future. A well-structured code is easier to understand and it communicates its purpose better. (7, pp. 200-201; 13, p. 56.)

Another problem is that the code quality degrades over time as shown in Figure 7. It is said that the code starts to rot as it is changed due to fixing a bug, adding a feature, or optimizing the system's performance or resource usage. The change may be done in a rush to support the short-term goals or the change may be done without a full comprehension of the design of the code. This causes the code to lose its structure and the original design fades away change by change. After a certain amount of time, every change takes more effort and

time, causing the overall productivity to decrease and the organization to fall behind its competition. Regular refactoring improves the design, readability and quality of software. This improves the development team's responsiveness to change, in other words refactoring improves the speed of the software development (Figure 8). (7, p. 14; 11, pp. 343-345.)

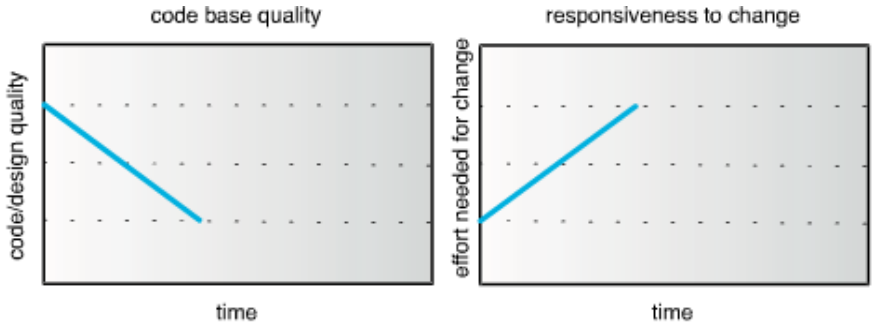


FIGURE 7. Code base quality decreases over the time (11, p. 344)

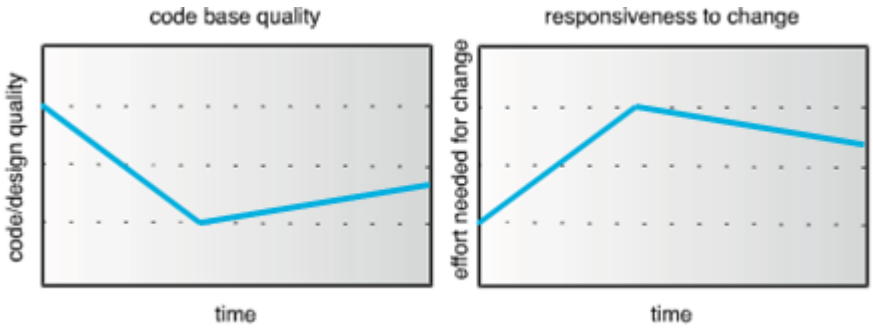


FIGURE 8. Refactoring improves code base quality (11, p. 345)

The Broken Windows Theory and the Boy Scouts rule are two metaphors which are related to refactoring. These metaphors guide the developers to keep the code base constantly clean with refactorings. The Boy Scouts rule is stated as:

> "*Leave the campground cleaner than you found it."* (7, p. 14)

This means that if all developers would check-in their code somewhat cleaner compared to it when it was checked out, then the code quality could not decrease, instead it steadily increases as the technical debt is settled in small payments. In practice this means that whenever a component is changed, the

neighboring code of the change point is evaluated and improved as shown in Figure 9. (11, p. 346.)

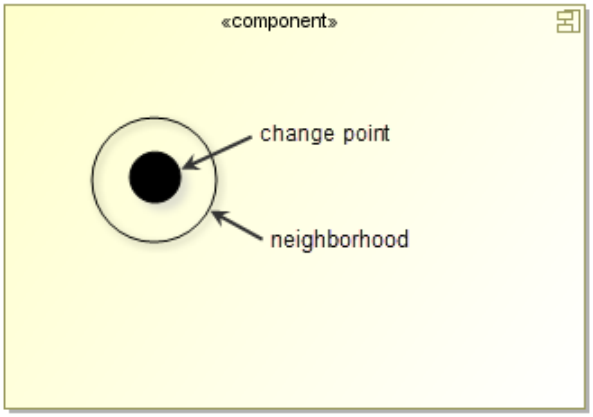


FIGURE 9. Cleaning the neighborhood of the change point

The concept of problems getting worse over time was popularized through the broken windows theory:

> *"The signs of disorderly and petty criminal behavior are triggering more disorderly and criminal behavior, thus causing the behavior to spread. By addressing all the little ways in which people make a mess of their environment, and cleaning things up frequently, it is believed that more serious crimes can be prevented."* (6, pp. 4-5.)

This means that people tend to adapt their behavior to the environment that they live in. People also copy norms and behaviors from each other. Hence, bad designs, wrong decisions or poor code should not be left unrepaired, or otherwise an impression that nobody cares about the code quality is given and any kind of changes (hacks) can be committed into the code base. Hence, broken windows accelerate the decrease of the software quality.

The refactoring process consists of three main stages: identification, refactoring and assessment. The refactoring should not be an activity that is separately scheduled; instead it should be done constantly in small steps. The most natural time for refactoring is when a new feature is implemented, a bug is fixed, or a

code is reviewed. The developers of the affected code must be notified to communicate the up-coming refactoring because it might affect their plans. (13, pp. 58-59; 14, pp. 3-5.)

The three main stages of the refactoring process are divided into the following activities:

- Identify change point, i.e. method, interface or class
- Determine refactoring
- Guarantee that applied refactoring preserves behavior
- Apply the refactoring
- Assess the effect of refactoring
- Maintain the consistency between refactored code and other software artifacts such as tests, documentation, etc.

In the identification stage the software modules that need refactoring are identified, and appropriate refactoring(s) are determined. The identification can be based on the updated target architecture design, architecture violations, or code smells. (14, pp. 18-20; 15, pp. 41-53.)

In the refactoring phase the selected refactoring is implemented. Unit tests are used to guarantee that the refactoring preserves the software module's original behavior.

Finally, the updated design and implementation should be assessed to evaluate the refactoring result; how the non-functional quality is improved in terms of size, complexity, coupling, cohesion, mobility and extensibility. The functional quality is tested with regression and performance tests.

If the identified change point does not contain tests, then the following activities should be done before refactoring the code:

- Find test points from the change point
- Break dependencies

- Write tests

This is called the legacy code change algorithm. Its purpose is to bring more of the system under test every time the system is changed to eventually bring the most of the system into the test-covered code. (14, pp. 18-20.)

The following principles should be followed when applying refactoring (15, pp. 66-68.):

- Do not try to add new functionality and refactor at the same time.
- Refactoring is applied on a working code which is already covered with unit tests. Unit tests are used to verify that refactoring preserves external behavior and to tell if refactoring has broken anything. Unit tests should be run as often as possible during refactoring process.
- Refactoring often involves making many localized changes that result in a larger-scale change. To avoid prolonged debugging sessions, take short deliberate steps instead of trying to do everything at once.

3.6 Testing

In traditional software development projects, the software system-level testing and system integration testing were activities done at the end of the project, when the requirements and coding processes were complete. This meant that large quantities of untested code were handed over to the testing organization to determine whether the code actually worked. Usually, the testing resulted in many hidden defects found in the code. This caused unpredictable amount of re-work for the development organization, and resulted to the “fix – re-deliver – re-test” cycles which in turn ultimately delayed the product release.

In an agile software development project the development teams are cross-functional. This means that there are no boundaries between the development and testing organizations, hence the testing should be a concurrent and integral part of the software development process. The main agile testing principles are as follows (2, pp. 102-109; 11, pp. 29-33.):

- All developed code is tested code (and there is no other kind). The development team does not get credit for delivering functionality that has been implemented but not tested, in other words the definition-of-done is not fulfilled.
- All tests are written within the iteration boundaries, and they are written by the developers, testers and the product owner, i.e. writing tests is a team effort.
- The manual testing work is minimized by introducing test automation. Test automation is the rule, not the exception.
- Defects are found only once. This means that once a human tester finds a bug, a test automation case is written which will check the particular case in the future.

In principle, agile promotes that all developed code is tested code. Achieving a fully tested code is a challenge for the agile teams. This forces the development team to think through how the code will be tested. The agility forces the development team to design inherently testable systems.

There are several major types of the software testing that can be performed during a development project. Usually the following four testing strategies are selected as the primary layers in an agile software development project: unit-, acceptance-, component-, system- and performance testing.

A unit test is a code that exercises the individual units of the production code. A unit is the smallest testable part of the application, usually a method, an interface or an entire class. Unit testing is low-level testing; developers have full access to the internals of the tested unit, and the unit's external dependencies are removed by introducing mock objects, method stubs and fakes. Unit testing is the foundation for all other forms of testing; if the individual software parts, methods or classes, do not work individually then they will not probably work well together in the system. The goal of the unit testing is to prove that the individual parts are correct by defining a strict, written contract the piece of the

code must satisfy. The unit tests themselves are not parts of the system under the test and therefore do not affect the test results.

A comprehensive unit test set for the software system gives the following benefits in terms of quality and productivity for the development project (2, pp. 156-159; 11, pp. 72-74.):

- The unit tests are automated tests which are integrated to continuous integration build. This means that the errors are found early and time is saved from the manual testing and the testers can concentrate more on the system level testing.
- The unit tests simplify the integration and integration testing because the individual software parts are tested before the integration.
- The unit tests can be used to measure the static and dynamic properties of the source code. The function and branch coverage are static data which can be collected when the unit tests are run. If the unit tests are deployed to the real environment, for example to a mobile device, then a dynamic analysis is also possible. This means that the unit tests can be used to reveal for example memory or resource leaks or to measure the performance characteristics of a unit. The static and dynamic analysis results help the developers to understand how a unit behaves under certain conditions.
- The unit tests give more confidence to the developers to change code. Adding new features or refactoring the existing code is easier and faster because possible regression can be seen immediately when the unit tests are run.
- The unit tests provide documentation of the source code. Each unit test specifies observable behavior of a certain method, interface or class. Hence, the developers can get the basic understanding of the code unit by studying the test case.

In the agile, acceptance testing (or functional testing) refers to the testing performed by anyone who has the ability to evaluate the new code that has been implemented during the iteration against its requirements. Usually, the testing is done either by the test team, product owner or customer. Each acceptance test represents some expected result from the system. The product owner or the customer is responsible for verifying the correctness of the acceptance test. This ensures that the implemented functionality is bug-free and the overall system meets the requirements set by the customer. (2, pp. 160-161; 11, p. 51.)

The acceptance testing is also called “black box” testing because the tests run at a level above the code and the software system behavior is evaluated by its interaction with the user. Therefore, there are multiple ways of executing the tests; manual testing, database-driven testing and automated testing. Automated test methods are often tightly coupled to the implementation details and this results in high maintenance cost of the test automation. Because of this, the acceptance testing is often done manually by the quality assurance personnel or by the customer. (2, pp. 160-161.)

The software components and subsystems are integrated in the system testing phase and the system is deployed into the staging environment (or a production like environment). The goal of the system testing is to prove that all components work well with each other. The software system behavior is verified with end-to-end testing which consists of automated component test suites and basic functional tests. The test results are analyzed and the results are saved as reference for future regression tests. (2, pp. 162-164.)

The system testing phase is often used to test other characteristics of the system. This usually includes at least performance testing, stress testing and recovery testing. The performance testing is done to find out possible bottlenecks in the system and to verify that the system meets its performance requirements under real-world conditions. The performance tests should also include resource consumption tests, i.e. the tests should measure how much the

system consumes memory, disk space, network and other processing resources under certain circumstances. Once the system testing has been completed under ideal conditions, the system should be stressed to understand when the system breaks and how it recovers (application crash, application slows down, or in worst case operating system crash) from an exceptional state. The performance test results form a basis for the future performance regression tests. (2, pp. 162-164.)

4 SOFTWARE DESIGN

In agile projects, software design and architecture are allowed to emerge and evolve as the project proceeds. Instead of investing months in building either detailed software architectural models or prototypes, the agile team focus on delivering early, value-added stories into integrated baseline. Early delivery makes possible to test the requirements and architectural assumptions. This drives a risk out by proving or disproving the assumptions of design and architectural assumptions. If the solution does not work, the team refactors, re-designs and re-implements until the solution is acceptable. In other words, every developer in the development team is responsible for the software design and architecture; hence, every software developer must know and understand the characteristics of the software design principles, and properties of the good software design.

4.1 Elements of Good Software Design

The desirable characteristics of good software design include the following properties (16, pp. 63-64):

- Loose coupling
- High cohesion
- Extensibility
- Portability

The degree how much a class depends on other classes is called coupling. Loose coupling can be achieved by separating the design into modules, or into classes in the object-oriented systems. An indication of tight coupling is a shared data and/or procedures between the modules. The loosely coupled modules are connected by their interfaces and any data they both need must be passed through an interface. The loosely coupled modules does not expose their internal behavior to outside of the interface, instead they only share their

interface. The Bridge design pattern is an example which is used to decouple its abstraction (interface) from its implementation. (16, p. 64.)

In the loosely coupled modules the changes are localized because the modules are small and self-contained and the components have well-defined responsibilities. The loosely coupled modules promote for the reuse of the software; the existing components are easier to reconfigure and reuse in a new environment. Loose coupling also has other benefits; it isolates the diseased software modules from the healthy modules. If a module is broken, it is less likely to spread the symptoms around the rest of the system, which reduces the system fragility. Furthermore, loose coupling makes it easier to design and run tests on a module as the dependencies on other modules are not impeding testing. (16, p. 64.)

High cohesion means object independency. Cohesion within a module is the degree to which a module is self-contained with regards both the data it holds and the operations that act on the data. This means that a highly cohesive class has all the data it needs defined within the class as the member variables and all the operations (methods) which are allowed to modify the data are defined within the class as well. Any object instantiated from a highly cohesive class is very independent, the object communicates with other objects through its published interface. High cohesion improves the software readability, increases the mobility and reusability while it keeps the complexity at a manageable level. (16, p. 64.)

Incremental and evolutionary process models require software extensibility, in other words the software is never “ready”. This means that customers are asking for new features release after release. The extensibility measures how easily new features can be implemented into the existing modules. When modules are simple and loosely coupled, new features are easy to implement. Modules should be designed so that the new features can be implemented by adding a new code, rather than by changing the old code which is already

working. Abstraction and polymorphism are the primary mechanism for the extensibility. (16, p. 64.)

Portability means the usability of the existing software modules in different computer environments or platforms. The main problems related to portability are related to operating systems, hardware architectures and user interfaces. The pre-requirement for portability is the generalized abstraction between the application logic and system interfaces. Portable software reduces the overall development costs if the software is to be deployed on multiple platforms. (16, p. 64.)

On the other hand, developers should also understand the characteristics of bad designs, so that they can critically evaluate the existing designs and code and to know how to avoid the properties of bad designs. The characteristics of bad designs properties are (15, p. 104; 17, p. 2.):

- Rigidity
- Fragility
- Immobility
- Viscosity

Rigidity is the tendency for software to be difficult to change. Even a simplest change causes many unexpected subsequent changes in dependent software modules. Software rigidity reduces the development team's velocity and the ability to respond to new requirements, because the team cannot make reliable work effort estimations for the required change. Hence, rigidity affects the project management; when the software is difficult to change, the managers do not allow the team to implement any non-critical changes and ultimately the management may refuse to allow any changes in the software. (15, pp. 104-105; 17, p. 2.)

Fragility closely relates to rigidity and it is the tendency of the software to break in unexpected modules every time it is changed. Many times there is no relationship between the change point and breakage point. Fragile software is

difficult to maintain because every fix makes it worse and increases the probability of breakage. A fragile software system apparently decreases the development team's credibility and reliability; the customers and managers suspect that the development team has lost control of their software. (15, p. 105; 17, p. 2.)

Immobility is the tendency for software to be difficult to reuse. The development team may discover that their software system already contains a needed module, but it cannot be reused because it is highly coupled with the surrounding software modules. An immobile software system is susceptible for code duplication. (15, p. 105; 17, p. 3.)

Viscosity is the tendency of the software to have multiple ways how a required change can be implemented into it. Some of the ways preserve the system design while others are considered as hacks. If the design preserving methods are more difficult to implement than hacks, then the viscosity of design is high, meaning that the development team can easily do the wrong decisions while it is more difficult to do the right one. (15, p. 105; 17, p. 3.)

Object Oriented Design (OOD) defines the principles and Design Patterns techniques which help the developers to build orthogonal and clean software systems. These principles and patterns are used to increase cohesion, decrease coupling, separate concerns and increase the modularity of a software system (6, pp. 34-37.). The primary design principles are described in the following sections in more detail.

4.2 The Open-Closed Principle

The definition of the open-closed principle (OCP) is:

> *"Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification"*. (18, p.1)

The software modules which are OCP conformant have the following two primary attributes (18, pp.1-2.):

- They are “open for extension”; the behavior of a module can be extended so that the module can behave in new and different ways as the requirements of the application change.
- They are “closed for modification”; the existing source code of the module is inviolate.

This means that the software modules should be written so that their behavior can be extended, without changing the source code of the module. New features should be added to the software system by adding a new code, rather than by changing the old code which is already working.

Abstraction and polymorphism are the primary mechanisms behind the open-closed principle. The abstractions are abstract base classes and the possible behaviors are presented as derivate classes. This makes it possible for a software module to manipulate an abstraction. Thus, the module is closed for modification because it depends upon the abstraction that is fixed. The behavior of the module can be extended with new derivates of the abstraction. (16, p. 117; 18, pp. 9-13.)

The example shown in Figure 10 violates the OCP principle because it uses the run time type identification (RTTI) to detect the shape type. If a new derivate type of `Shape`, for example `Triangle`, is added to the system, then the function `calculateArea` has to be modified to support the triangle area calculation.

```
100 double calculateArea(const std::list<Shape*>& list)
101 {
102     double area = 0;
103
104     list<Shape>::iterator it;
105
106     for (it = list.begin(); it != list.end(); it++) {
107         Shape* shape = *it;
108
109         Circle* circle = dynamic_cast<Circle*>(shape);
110         Rectangle* rect = dynamic_cast<Rectangle*>(shape);
111
112         if (circle)
113         {
114             double radius = circle->getRadius();
115
116             area += (PI * radius * radius);
117         }
118         else if (rect)
119         {
120             area += (rect->getWidth() * rect->getHeight());
121         }
122     }
123
124     return area;
125 }
```

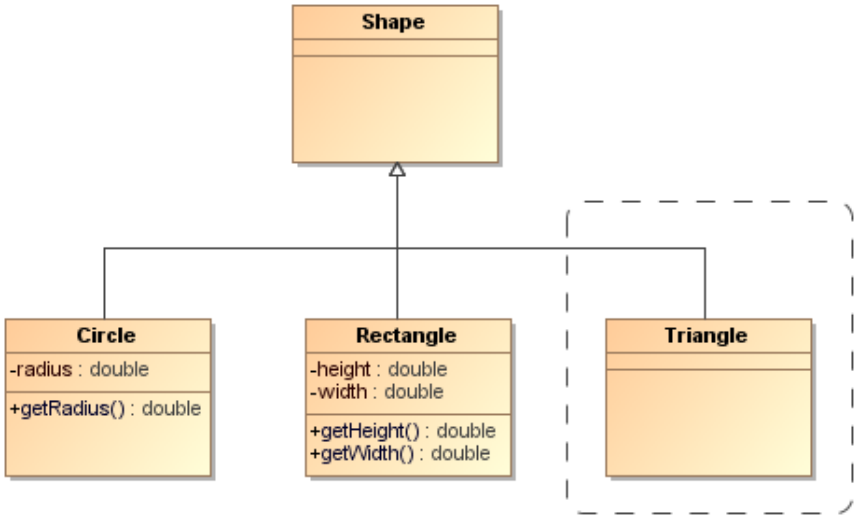


FIGURE 10. Solution violates OCP principle

Figure 11 shows an OCP conformant solution. It uses polymorphism and abstraction to make the system open for extension, while keeping the `calculateArea` intact even if a new `Shape` derivates are added to the system.

```
class Shape
{
protected:
    virtual double getArea() const = 0;
};

class Circle : public Shape
{
public:
    double getArea() const { return PI * m_radius * m_radius; }

private:
    double m_radius;
};

class Rectangle : public Shape
{
public:
    double getArea() const { return m_width * m_height; }

private:
    double m_height;
    double m_width;
};

double calculateArea(const std::list<Shape*>& list)
{
    double area = 0;

    list<Shape>::iterator it;

    for (it = list.begin(); it != list.end(); it++) {
        Shape* shape = *it;

        area += shape->getArea();
    }

    return area;
}
```

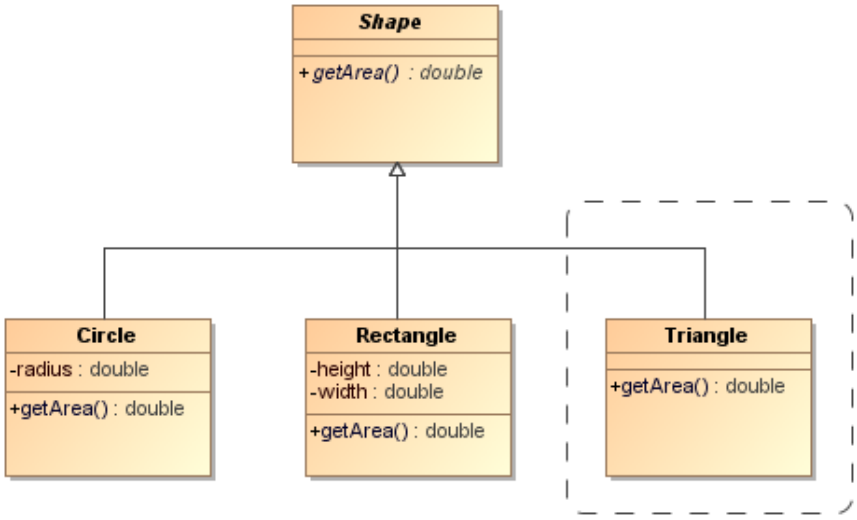


FIGURE 11. OCP conformant solution

4.3 The Liskov Substitution Principle

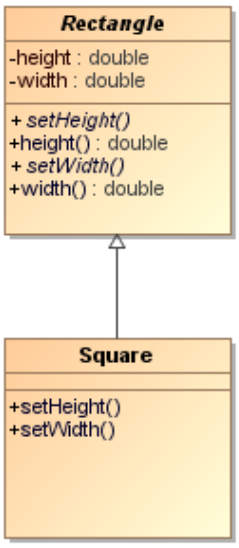
The definition of the Liskov substitution principle (LSP) is:

> *"Subtypes must be substitutable of their base types"*. (19, p. 2)

This means that the functions that use pointers or references to the base classes must be able to use objects of the derived classes without knowing it. (19, p. 2)

It is assumed that function `F` accepts as its argument a reference to base class `B`. If function `F` misbehaves when a derivate `D` of `B` is passed into it, then `D` violates the Liskov substitution principle and `D` is fragile in the presence of `F`. If function `F` is modified to behave properly with `D` by using the run time type information (RTTI), then `F` violates OCP because it has to know all possible derivates of `B` and must be modified every time when a new derivate of `B` is created (16, pp. 124-125; 19, pp. 2-7.)

LSP violation is often demonstrated with `Square` is-a `Rectangle` problem, where the inheritance design is made wrong on purpose. Figure 12 shows a design which violates the Liskov substitution principle. The example function `setDefaultDimensions()` accepts as its argument a reference to the base class `Rectangle`. If an instance of `Rectangle` is passed in, then the function behaves as expected. If an instance of `Square` is passed in instead, then the function misbehaves as the assertion fails. The reason is that the design of the class `Square` violates the Liskov substitution principle by overriding the virtual functions `Rectangle::setWidth()` and `Rectangle::setHeight()` (lines 29-30 and 35-36) and then the overriding behavior in the class `Square` forces the base class `Rectangle` to hold the mathematical width and height for square (i.e. width and height must always be kept the same for a square) which in turns causes the function `setDefaultDimensions()` to misbehave in the presence of `Square`.




```
27  void Square::setHeight(double height)
28  {
29      Rectangle::setHeight(height);
30      Rectangle::setWidth(height);
31  }
32
33  void Square::setWidth(double width)
34  {
35      Rectangle::setHeight(width);
36      Rectangle::setWidth(width);
37  }
38
39  void setDefaultDimensions(Rectangle& rect)
40  {
41      rect.setHeight(100);
42      rect.setWidth(50);
43
44      assert(rect.getHeight() == 100 && rect.getWidth() == 50);
45  }
46
47  void testSetDefaultDimensions()
48  {
49      Rectangle rect;
50      setDefaultDimensions(rect);
51
52      Square square;
53      setDefaultDimensions(square);
54  }
```

FIGURE 12. LSP violation

The LSP violations increase the software system fragility. There are no mechanical (i.e. by compiler) ways to completely avoid and detect LSP violations. Whether a class is LSP conformant depends upon the clients that it has and upon their expectations.

4.4 The Dependency Inversion Principle

The dependency inversion principle (DIP) is defined as

> *"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions"*. (20, p. 6)

The open-closed principle defines the goal of object-oriented architecture and the dependency inversion principle states the mechanism how software entities can be closed for modification while being open for extension. The dependency inversion is the strategy to design software modules so that they depend upon interfaces, abstract functions and classes, instead of concrete functions and classes. (16, p. 131; 20, pp. 6-7.)

Traversing relationships between the objects directly can lead to a combinatorial explosion of the dependency relationships. The symptoms are shown for example in the following ways: A software module is difficult or even impossible to get under unit test harness. Simple changes to one module propagate through unrelated modules in the system. The development team is afraid of changing code because they do not know what parts of the system are affected. (20, pp. 6-12.)

The principle of least knowledge, also known as Law of Demeter, is a design guideline which attempts to minimize the coupling between the modules. The principle states that any method of an object should call only methods belonging to it, any parameters that were passed into the method, any object it created, or any object it directly holds. (6, pp. 140-141.)

4.5 The Single Responsibility Principle

The single responsibility principle (SRP) is defined as:

> *"There should never be more than one reason for a class to change"*. (21, p. 1)

If there is more than one motive for changing a class, then the class has more than one responsibility.

Every large software system contains a large amount of logic and complexity. The primary goal in managing such complexity is to organize it in a well-formed way. The system should be composed of many small classes instead of large ones. Each small class encapsulates a single responsibility and collaborates, according to the Law of Demeter, with a few others to achieve the desired system behavior. (21, pp. 1-4.)

The software modules that are SRP conformant are usually highly-cohesive and loosely-coupled, meaning that readability, mobility and reusability are increased, while the complexity is kept at a manageable level.

4.6 The Interface Segregation Principle

The definition of the interface segregation principle (ISP) is:

> *"Clients should not be forced to depend upon interfaces/methods that they do not use"*. (22, p. 5)

ISP states that many client specific interfaces are better than one general purpose interface. If clients are forced to depend upon interfaces they do not use, then the clients are subject to changes to those interfaces. This results in an inadvertent coupling between all clients. In the worst case, a change in the interface has a great impact on the dependent clients and may force recompilation and redeployment of a large part of the system. (22, pp. 1-5.)

ISP does not recommend that every class that uses a service has its own special interface class that the service must inherit from. Instead, the clients should be categorized by their type, and interfaces for each type should be created.

Delegation (object form) and multiple inheritance (class form) are the patterns how fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients. The software modules that are ISP conformant are more portable, hence ISP reduces the software immobility. (22, pp. 5-7.)

4.7 The Least Knowledge Principle

The least knowledge principle (LKP), also known as Law of Demeter, is defined as:

> *"Talk only to your immediate friends".* (7, pp. 97-98)

LKP states that coupling between modules should be minimized. If *n* objects all know about each other, then a change to only one object can result in the other *n – 1* objects needing changes. LKP states that classes should collaborate indirectly with as few other classes as possible. In other words, eliminating unnecessary interactions between the modules protects everyone. On the other hand, it is impossible to implement software in which the objects are not somehow connected and collaborating. (6, pp. 140-142.)

When an object is asked for a particular service, the service should be performed on behalf of the object, not by delegating the service request on third-party object. Figure 13 shows a LKP violation on line 54 where a service request is delegated. The method `Display::showTemperature` is unnecessarily coupled to three classes – `Sensor`, `SensorData` and `OilData`. This style of coding increases dramatically the coupling between the classes. A change to any of the above three classes may affect the `Display::showTemperature` method and cause a need to change it. Thus the `Display::showTemperature` method also violates the open-closed principle.

```
52  void Display::showTemperature(const Sensor& sensor)
53  {
54      double oilTemp = sensor.getSensorData().getOilData().getTemp();
55      m_oilTempLabel->setText(oilTemp);
56
57      ...
58
59  }
```

FIGURE 13. LKP violation

Figure 14 shows a refactored version of the `Display::showTemperature` method. The method is renamed to `Display::print` and it now requires exactly the data it needs, and it does not have to care how `OilData` is acquired at the upper level of the call hierarchy.

```
87  void Display::print(const OilData& oilData)
88  {
89      double oilTemp = oilData.getTemp();
90      m_oilTempLabel->setText(oilTemp);
91
92      ...
93
94  }
```

FIGURE 14. LKP violation removed

Figure 15 shows an example of the Law of Demeter for functions. The law states that any method of an object should call only methods (6, p. 141):

1. Belonging to object itself (`Sensor::getOilData()`)
2. Of object(s) that were passed into a method as a parameter (`display.print(oilData)`)
3. Of object(s) that it encapsulates (`m_impl->getOildData()`)
4. Of object(s) that it holds directly (as local variable) (`converter.convert(oilData)`)

```
8   class Sensor
9   {
10  public:
11      Sensor();
12      virtual ~Sensor();
13
14  public:
15      void print(const Display& display) const;
16
17  private:
18      OilData& getOilData() const;
19
20  private:
21      SensorImpl* m_impl;
22  };
```

```
71  void Sensor::print(const Display& display) const
72  {
73      Converter converter;
74
75      OilData& oilData = getOilData();          1.
76
77      converter.convert(oilData);               4.
78
79      display.print(oilData);                   2.
80  }
81
82  OilData& Sensor::getOilData() const
83  {
84      m_impl->getOildData();                    3.
85  }
```

FIGURE 15. Law of Demeter for functions

LKP compliant design makes the code more adaptable and robust because it reduces the coupling between the modules. LKP reduces the size of the

response set in the calling class and hence it helps to create a more maintainable code.

4.8 The Don't Repeat Yourself Principle

The definition of don't repeat yourself principle (DRY) is:

> *"Every piece of knowledge must have a single, unambiguous, authoritative presentation within a system".* (6, p. 27)

The guideline of the DRY principle is to minimize the duplication within a system and to foster an environment where it is easier to find and reuse existing artifacts. DRY applies to all artifacts; code, unit tests, specifications, requirements, processes, etc. DRY requires that each piece of information and each behavior is expressed only once in a single place. This means that the design should be created in a way that requirements are implemented in one logical place. When a requirement changes, there is only one place to look at and implement the required change. A design conforming to the DRY principle is more reliable and hence easier to understand and maintain. (6, pp. 26-27; 7, pp. 289-290.)

4.9 Object-Oriented Design Patterns

The design patterns are high-level descriptions, templates and well-known solutions to common software engineering design problems that occur in object-oriented software design. Design patterns are not finished designs which can be applied directly into the program code. Instead, they show the relationships and interactions between the objects to solve a certain problem to help the developers to adapt the well-known solution into the design of their application. The reasons why design patterns should be used are (23, pp. 12-15; 24, pp. 80-86.):

- Make a decision whether the design is right and not just one that works.
- Shift the level of thinking to a higher perspective

- Reuse existing, high-quality solutions to commonly recurring problems to improve the code quality.
- Adopt improved design alternatives, even if the design patterns are not explicitly used.
- Improve communication within the team by using common terminology and a common viewpoint of the problem.
- Improve individual and team learning.

A comprehensive collection of the object-oriented design patterns can be found in the book Gang of Four (Gamma et. all) [23]. The book collects and documents the most common design patterns and uses the following categories and concepts to group them:

- Creational patterns (delegation)
- Structural patterns (aggregation)
- Behavioral patterns (consultation)

The creational patterns are used to deal with the mechanisms of the object creation, i.e. they abstract the object instantiation process. They aim to separate a system from how its objects are created, composed and represented to increase the system's flexibility to control its object creation. The creational patterns are further categorized into class and object creational patterns. The object creational pattern delegates a part of its object creation to another object, while the class creational pattern defers its object creation to its subclasses. The well-known creational patterns are (23, p. 94.):

- *Singleton pattern* ensures that a class has only one instance and provides a global point of access to it. The singleton instance is shared between all of the clients of the class.
- *Prototype pattern* creates new instances of the class by cloning a prototypical instance. The prototypical instance specifies the kind of object to be created.

- *Builder pattern* enables to use the same construction process to create different kind of objects. It separates the construction of a complex object from its presentation.
- *Factory Method pattern* allows a class to defer instantiation to its subclasses. It defines an interface for creating an object, but subclasses are responsible to decide which class to instantiate.
- *Abstract Factory pattern* provides an interface for creating objects without specifying their concrete classes.

The structural patterns are used to ease the software design by identifying a simple way to realize the relationships between the software entities, i.e. they are concerned with how classes are composed to form larger structures. The creational patterns are also further categorized into class and object structural patterns. The structural class patterns use inheritance to compose interfaces or implementations to form functionality, while the structural object patterns describe ways to compose objects to realize a new functionality. The object composition is more flexible than a static class composition as the object composition can be changed in run-time. Some of the useful creational patterns are (23, p. 155.):

- *Adapter pattern* is used to convert the interface of a class into another interface the clients expect, i.e. Adapter provides its own interface to the clients while it internally uses the original interface. Hence, Adapter allows the classes to work together that could not otherwise because of the incompatible interfaces.
- *Bridge pattern* is used to decouple an abstraction from its implementation so that the two can vary independently. The Bridge uses encapsulation, aggregation and possibly inheritance to separate the responsibilities into different classes.
- *Decorator pattern* is used to attach additional properties to an existing object dynamically in run-time, i.e. Decorator can be used to extend (decorate) the existing objects with the functionality of a certain object.

The Decorators provide a flexible alternative to subclassing for extending the functionality.

- *Facade pattern* is used to provide a unified interface to a set of interfaces in a subsystem, i.e. Facade provides a simplified higher-level interface that makes the sub-system easier to use for its clients by reducing dependencies and making the interface easier to understand.
- *Proxy pattern* is used to provide a surrogate or placeholder for another object to control access to it, i.e. Proxy provides a class functioning as interface to something else. The proxy usually provides an interface access to an object whose creation and initialization is expensive or the object is otherwise impossible to duplicate.

The behavioral patterns are used to describe communication and responsibilities between objects and classes, i.e. these patterns characterize the control flow that is difficult to follow in run-time. The behavioral patterns are also divided into class and object patterns. The class patterns use inheritance to distribute behavior between the classes, while the object patterns use the object composition instead of inheritance. The creational patterns let the developers to concentrate on how the objects are interconnected. This helps the developers to manage coupling between the modules. Some of the behavioral patterns include (23, p. 249.):

- *Iterator pattern* is used to access the elements of an aggregate object, such as a list, without exposing its internal representation, i.e. Iterator provides a way to take the responsibility of access and traversal out of the aggregate object. It defines an interface to access the elements of the aggregate object and behavior how the elements are traversed.
- *Observer pattern* is used to define a one-to-many relationship between the objects by registering to observe an event/state of a subject. When the subject changes, all its dependents are notified.

- *State pattern* is used to allow an object to alter its behavior dynamically in run-time when its internal state changes, i.e. the object internally appear to change its (state) class.
- *Template method pattern* is used to define the skeleton of an operation which defers a part of the operation behavior to subclasses, hence template methods make it possible to redefine a certain part of an algorithm without changing the algorithm's structure.
- *Visitor pattern* is used to present an operation to be performed on the elements of the object structure. The visitor allows defining a new operation without changing the classes of the elements on which it operates.

5 SOFTWARE DESIGN VIOLATIONS

The term code smell was originally coined by Beck and Fowler [13] and it is a widely used term in agile software projects. A code smell is a surface indication of software design principle violation and it usually corresponds to a deeper problem in a software system. From a software developer's viewpoint, the code smells are general descriptions of a bad code. Code smells can be used as heuristics to identify a bad code and help in the decision making whether refactoring is needed to certain parts of a software system. Beck and Fowler [13] provide a list of refactorings which can be used to cleanup design violations.

The following sections outline the most common design violations based on taxonomy of five groups. The taxonomy helps to better understand the design violations leading to bad code. Furthermore, the taxonomy helps to recognize the relationships between the violations.

5.1 The Bloaters

The Bloater design violations are shown in Figure 16 (25, p. 408). These smells are an indication of a code that has become so large that working with it is inefficient. These smells increase over the time when the code is changed.

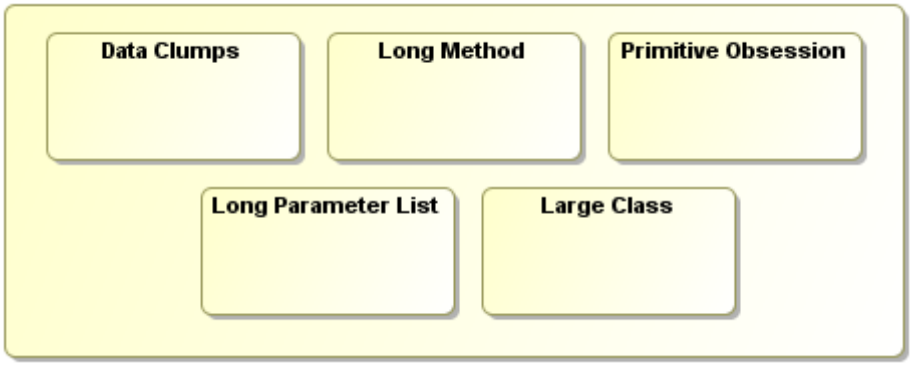


FIGURE 16. The Bloater design violations

Data Clumps refers to the tendency of using duplicated member variables in multiple classes, for instance three integers for RGB color, or duplicated

parameters in multiple methods. Data clumps can be avoided by encapsulating data into classes. This simplifies the method signatures and groups the behavior data represents, i.e. class `Color` in case of RGB data clumps. (13, p. 81.)

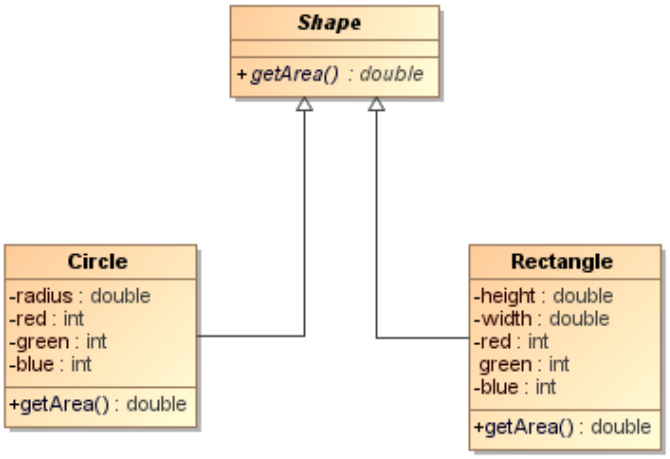


FIGURE 17. Data clumps in classes Circle and Rectangle

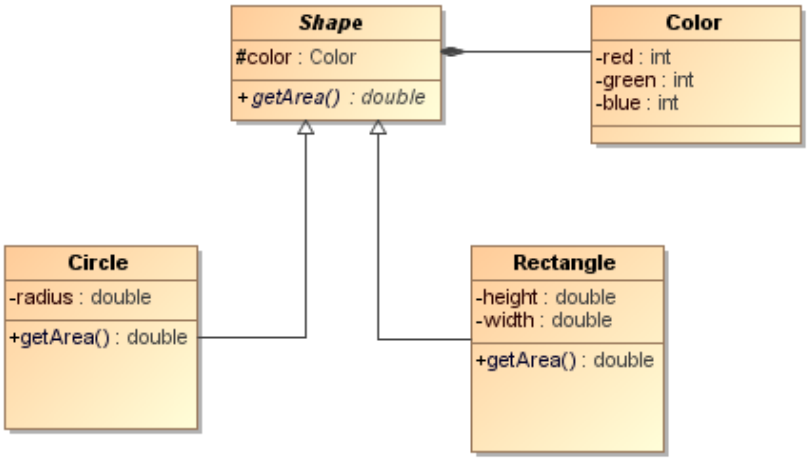


FIGURE 18. Data clumps encapsulated into class

Long Method, as its name states, refers to long methods which should be avoided because long routines are difficult to understand and maintain. To avoid long methods, developers should decompose the methods more aggressively. A heuristic that can be followed to decompose methods is: if a code comment is needed to explain the code behavior or purpose, then a new method with a descriptive name should be written instead of the comment. Short well-named

methods should be favored because their behavior is easier to understand. (13, pp. 76-77.)

Large Class design violation indicates a class which is trying to do too much. A large class often shows up either with too many member variables in a class or with too much code in a class. When a class is too large, it also has a tendency to code duplication. (13, p. 78.)

The existence of Primitive Obsession in a code indicates that there are no small classes for small entities (e.g. for telephone number). Instead, the data and functionality is wrongly added to some other class which vainly increases the size of the other class. Primitive obsession is actually a symptom that causes more design violations to occur. (13, pp. 81-82.)

Long Parameter Lists design violation is a tendency of methods to have multiple parameters. Long parameter lists should be avoided because multiple parameters may be difficult to understand and parameter lists tend to change quite often. Methods should be designed so that most of the needed data is available in objects. If the objects are passed as parameters to a method, the dependency structure should be evaluated to avoid unnecessary coupling between the objects. (13, pp. 78-79.)

5.2 The Change Preventers

Figure 19 shows the design violations in the group Change Preventers (25, p. 408). These smells describe code that is difficult to change. When the code has either Divergent Change, Shotgun Surgery or Parallel Inheritance Hierarchies design violations, it means that the code violates the rule suggested by Fowler and Beck [13] which says that the classes and possible changes should have one-to-one relationship.

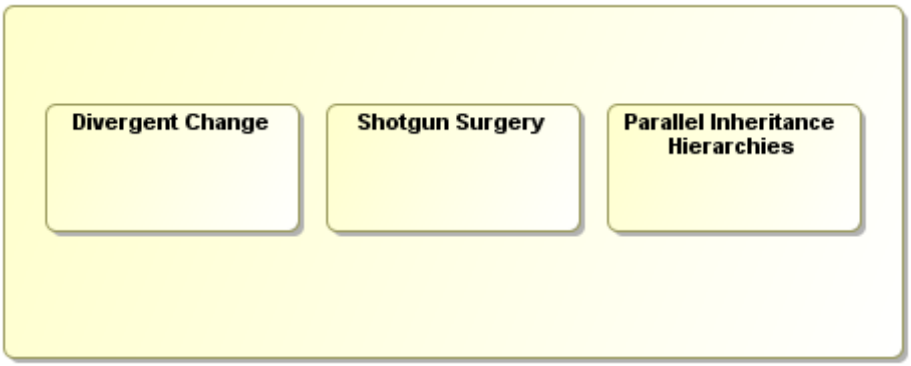


FIGURE 19. The Change Preventer design violations

Divergent Change design violation means that one class in the system is commonly changed in different ways for different reasons, for example `class A` methods `AA`, `BB` and `CC` have to be changed when a new database is added to the system, while its methods `DD` and `EE` must change when a new financial instrument is introduced. In this case, splitting the existing class to two separate classes would be better than the one class. To avoid divergent change design violation, the classes should be designed to be small enough with a single responsibility. (13, p. 79.)

Shotgun Surgery is opposite of Divergent Change. Shotgun surgery means that a change in the system always causes many changes to many different classes, for example `class A` methods `AA, AB, AC, class B` methods `BA, BB` and `class C` methods `CA, CB, CC` and `CD`. When the changes are spread over the system, it is difficult to find all the classes that require change and it is

easy to miss an important change. To avoid shotgun surgery, the classes should be designed so that a class encapsulates common behavior. (13, p. 79.)

Parallel Inheritance Hierarchies design violation means a duplicated class hierarchy and it is a special case of the shotgun surgery. In this case, when making a subclass of one class, a subclass of another class is also needed. (13, p. 83.)

5.3 The Couplers

The design violations in the Couplers group (Figure 20) are related to high coupling (25, p. 409). One of the most important design goals is to have low coupling between the software modules.

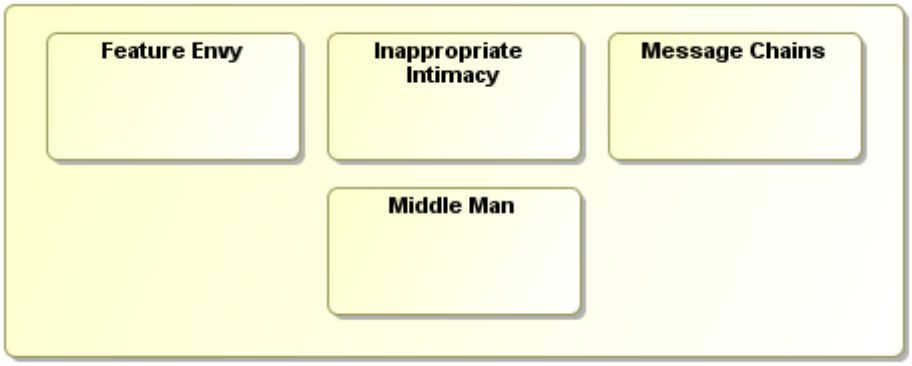


FIGURE 20. The Coupler design violations

Feature Envy design violation is a tendency of a method to be too interested in other classes. The most common envy is the data in the other class that is needed for example to calculate some value in the method. To avoid feature envy smell, the object responsibilities should be designed carefully. However, methods often use features of several classes, in this case the heuristic is to implement the method in the class which has most of the data. (13, pp. 80-81.)

The Inappropriate Intimacy violation relates to tight coupling where the objects are too interested in each other's private sections. Inheritance may lead to over-intimacy if the subclasses will always know more about their parents than their parents would like them to know. (13, p. 85.)

Message Chains is a design violation of Law of Demeter for functions. In message chains, object `A` needs data from object `D`. To access the data, object `A` has to access object `B`, and object `B` has to access object `C` and so on, i.e. `A.getB().getC().getD().getData()`. Finally, when `D` is available, `A` asks the data it needs. Navigating this way means that `A` is coupled to the structure of the navigation and any change in the intermediate relationships causes the client to have to change. (13, p. 84.)

A Middle Man is a class that is doing too much delegation instead of contributing to the functionality of the application. If encapsulation is used wrongly and the interface of the class simply delegates the requests to other classes, the design does not have a clear single responsibility. (13, p. 85.)

5.4 The Dispensables

Figure 21 shows the design violations in the Dispensable group (25, p. 409). These smells are an indication of a code which is representing something that is redundant and which should be removed. The main reason to eliminate such code is its cost; every code line costs money to maintain and understand.

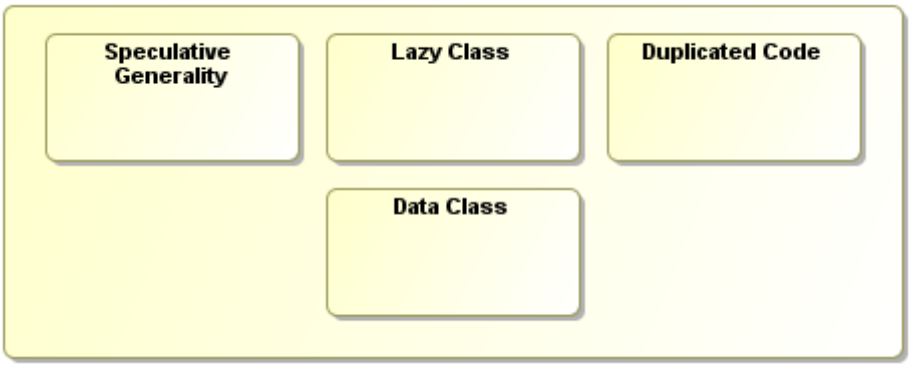


FIGURE 21. The Dispensable design violations

Speculative Generality violation refers to a code which is too general and may simply prepare for future extension. Speculative generality can be spotted when the only users of a method or class are test cases or when the code contains all

sorts of hooks and special cases to handle things that are not actually required by the existing implementation. (13, pp. 83-84.)

Duplicated Code violation is a smell which is probably the worst of the violations. This violation is not only restricted to the source code. Duplication can be found from any artifact which is produced during the development project. Potential sources for duplication are, for instance, design documentation, test specification, unit test harness, etc. In general, if the same (code) structure is shown in multiple places, then the source smells for a duplication which should be removed. (13, p. 76.)

Lazy Class violation is an indication of a class that is not doing enough, i.e., the class does not have enough responsibilities. Lazy Class is usually seen with a class inheritance hierarchy where a subclass was added to support a planned future extension but the extension was never implemented. A lazy class should be either removed or its responsibility should be increased. Data Class violation relates to Lazy Class and it describes a class that does not have anything else than data variables and getter/setter methods for the data (or the data is even made public in the worst case). Such data holders should be eliminated because they are manipulated in too much detail by other classes making them to suffer from Feature Envy violation. (13, p. 83.)

5.5 The Object-Oriented Abusers

The design violations in the group Object-Oriented Abusers are shown in Figure 22 (25, p. 408). The smells in this group are an indication of a code that does not take a full advantage of the possibilities of the object-oriented design.

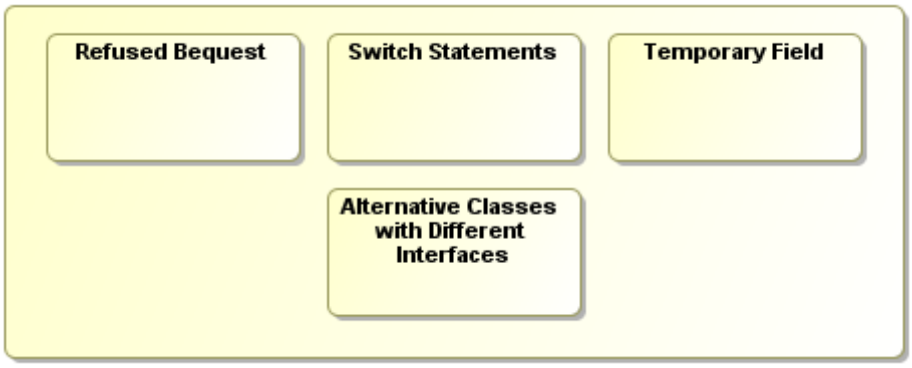


FIGURE 22. The Object-Oriented Abuser design violations

The Refused Bequest violation is an indication of improper class inheritance design. Often, the improper inheritance design means that a subclass does not accept all the properties given by its parent class. In this case, the parent class should hold only the data and methods that are common. This leads to the general advice that all superclasses should be abstract. On the other hand, a subclass may be reusing the parent class behavior while it does not want to support the interface of the parent class. This indicates that the inheritance hierarchy may be unneeded and could be replaced for example with delegation. (13, p. 87.)

The Switch Statements violation is identified for example when the object type is detected in switch statement. Usually this means that the module also violates the open-closed-principle as the function has to detect the object type. Switch statements are effectively causing code duplication in object-oriented systems. Every time when a new type is added, each switch statement has to be updated to reflect with the new type. In such case switch statements should be replaced with polymorphism if they are spread all over the system. (13, p. 82.)

Temporary Field violation means a case where the member variable of an object is set only in certain circumstances when a certain method is called. This means that the member variable should be declared in the method scope instead of in the class scope. Usually, temporary fields are shown together with a method that requires several variables, for example, for a complicated algorithm, and the temporary variables are valid only during the algorithm. (13, p. 84.)

The Alternative Classes with Different Interfaces violations indicate the lack of a common interface for the closely related classes. Hence, the smell indicates improper inheritance design. The closely related classes should be designed so that their protocols are the same to ensure the consistency. Polymorphism helps to achieve the consistency in the inheritance hierarchy. (13, pp. 85-86.)

6 DISCUSSION

According to my own experience from agile projects, the following items are commonly ignored by the development teams for some reasons:

- Refactoring is not done properly
- Unit testing is missing
- Definition-of-done is not followed
- Software design principles and patterns are not applied

Failing refactorings and the lack of unit testing are closely related. If automated unit tests are missing, then refactoring cannot be done properly as there is no guarantee that the code change done in the refactoring phase preservers the observable behavior of the system. Nevertheless, I have often seen code changes which are committed into the version control system after the code review and claimed to be “refactorings” even there is no guarantee of maintaining the observable behavior. These changes usually cause regression and hence a rework is needed later. The worst in this scenario is that often such changes are committed into the version control by a senior developer who had a feeling that nothing will be broken by the change. In the worst case the developer had such high confidence about the change that s/he did not test the change at all in the target environment.

Furthermore, the development teams are still too often missing automated testing. In the worst case the unit tests are missing entirely. Unfortunately, this means that the development team is writing their code with the Edit and Pray method. When the development is done in this mode, developers are using many efforts to carefully understand the existing code before modifying it. Once enough knowledge is gained from the change point, a modification is implemented. Then the system is run to see if the change was enabled. This is followed by further smoke tests to make sure nothing is broken in the neighboring components. In general, working with care is not something a development team should avoid, but implementing the software changes should

be effective, as working with too much care slows down the development or even stops it. When the unit tests are in place, they will provide a safety net for the developers; the working method changes from Edit and Pray to Cover and Modify. The safety net allows the developers to see easily the effects of implemented modification by running the tests. Another unit testing related flaw is that the tests are not written during the development, instead the tests are written after the implementation has been finished. I have seen trials where the development team is requested to write the unit tests to the existing software systems. In general, creating unit tests afterwards require a great deal of efforts and it might be impossible to get parts of the existing code under the test harness because the components are not designed for testability. Bringing the system under test harness afterwards is not usually the most attractive and valued task from the developer's viewpoint.

Definition-of-done is also a property of technical excellence which is not always followed by the development teams, i.e. it may be ignored in sprint review meetings, even it is one of the most important artifacts which can prove the work is really done. The definition-of-done states that the user story must pass its acceptance tests and other non-functional criteria in order to be acceptable in the sprint review. It should be clear that no user stories should be accepted in the sprint review if any of the items defined in DoD is not fulfilled. Still, many of the sprints which I have seen accepted are often missing many of the points listed in the DoD. It is clearly the product owner's fault if such stories are accepted. This behavior mainly means hiding an unfinished work from the stakeholders and creating an unnecessary burden to the development team to finish the unfinished implementation in the upcoming sprints.

The software design principles are quite often unknown to developers. Thus, the software designs are done on the ad-hoc basis which leads to a bad software architecture. These decisions might slow down the development speed of the software system. In other words, the first version might be delivered fast, but enhancements and improvements are extremely difficult to implement as the

software system does not have the characteristics of a good software design which would allow an easier extension of the system. Bad designs may have effect to willingness to have unit tests implemented; if the code constantly needs structural changes, then unit tests need changes and maintenance too. After a certain amount of time, the team might claim that having unit tests in such software system slows down the development as the maintenance of unit tests is taking too much development time. The claim is valid from the viewpoint that too much development time is burnt for unit test maintenance, but ignoring unit testing is not the solution to speed up the development. Instead, the team should concentrate on healing the design of the system in such a way that the existing code remains intact. In other words, the team should apply the design principles and patterns to their designs.

7 CONCLUSION

The thesis problem was obvious; what the main principles, practices and patterns are that agile development team should know and apply in a real world agile software development project to build software systems successfully.

Software design has changed recently as agile process models are adopted. The developers are now responsible for growing healthy software designs and architectures. To do it right, developers have to understand the characteristics of good design and the methods which support the building of high quality systems. This must be supported by the project stakeholders to require the best possible quality from the development team. Hence the project management and other stakeholders are impacted; they must also understand the agile principles and patterns in order to steer the development team work to the correct direction.

The principles, practices and patterns are widely documented in the literature, but unfortunately in many agile projects, the teams are still often lacking the skills presented in this thesis. To fill this cap in the knowledge, the skills should be taught either by universities or by employer organized trainings or self-study learning by individuals. Overall this means that the whole team should be encouraged to develop their skills in order to improve their craftsmanship.

Further research could concentrate on applying the presented principles, practices and patterns in practice to evaluate how well a team with an intermediate experience could grow healthy software in an agile project. This could be organized for example as a software design course where students execute a smallish software development project. In the end of the project each design would be evaluated and compared in terms of design and quality outcome to an up-front designed and implemented version to understand the quality differences between them.

8 REFERENCES

[1] Jurgen Appelo, *Management 3.0: Leading Agile Developers, Developing Agile Leaders*, Addison-Wesley, 2010.

[2] Dean Leffingwell, *Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007.

[3] *Manifesto for Agile Software Development,* date of data acquisition 10 Mar 2012, available at: http://agilemanifesto.org

[4] Agile vs. XP: The differences and Similarities, date of data acquisition 09 Apr 2012, available at: http://objectmentor.com/omSolutions/agile_xp_differences.html

[5] *Manifesto for Software Craftsmanship,* date of data acquisition 10 Mar 2012, available at: http://manifesto.softwarecraftsmanship.org

[6] Andrew Hunt, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.

[7] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

[8] *Introduction to Test Driven Development (TDD)*, date of data acquisition 09 Sep 2012, available at: http://www.agiledata.org/essays/tdd.html

[9] *Best Practices for Peer Code Review*, date of data acquisition 09 Sep 2012, available at http://support.smartbear.com/resources/cc/11_Best_Practices_for_Peer_Code_Review.pdf

[10] *Best Kept Secrets of Peer Code Review*, date of data acquisition 09 Sep 2012, available at http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf

[11] Craig Larman, *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum,* Addison-Wesley, 2010.

[12] *What is the Definition of Done (DoD) in Agile?*, date of data acquisition 09 Sep 2012, available at http://www.solutionsiq.com/resources/agileiq-blog/bid/64395/What-is-the-Definition-of-Done-DoD-in-Agile

[13] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[14] Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004.

[15] Martin C. Robert, *Agile Principles, Patterns, and Practices in C#,* Prentice Hall, 2006.

[16] John Dooley, *Software Development and Professional Practice*, Apress, 2011.

[17] *Design Principles and Patterns*, date of data acquisition 10 Mar 2012, available at: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[18] *The Open-Closed Principle*, date of data acquisition 10 Mar 2012, available at: http://www.objectmentor.com/resources/articles/ocp.pdf

[19] *The Liskov Substitution Principle*, date of data acquisition 10 Mar 2012, available at: http://www.objectmentor.com/resources/articles/lsp.pdf

[20] *The Dependency Inversion Principle*, date of data acquisition 10 Mar 2012, available at: http://www.objectmentor.com/resources/articles/dip.pdf

[21] *The Single Responsibility Principle*, date of data acquisition 10 Mar 2012, available at: http://www.objectmentor.com/resources/articles/srp.pdf

[22] *The Interface Segregation Principle*, date of data acquisition 10 Mar 2012, available at: http://www.objectmentor.com/resources/articles/isp.pdf

[23] Erich Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[24] Alan Shalloway, *Design Patterns Explained: A New Perspective on Object-Oriented Design,* Addison-Wesley, 2004.

[25] Mika V. Mäntylä, *Subjective evaluation of software evolvability using code smells: An empirical study*, date of data acquisition 16-Mar-2012, available at: http://www.soberit.hut.fi/~mmantyla/ESE_2006.pdf