Reducing outdated and inconsistent code comments during software development

The comment validator program

Adam Svensson
Abstract

During software development various forms of software documentation can be produced to make the software easier to understand and maintain after the software have been developed. One of these forms of software documentation is code comments, which is a form of software documentation that is produced to make source code easier to read and maintain. Although code comments make the code easier to read and maintain, code comments can become outdated and inconsistent with their corresponding code. Outdated and inconsistent code comments increase the probability for future bugs and when these comments are encountered, developers could lose the confidence for all other comments.

In order to reduce the amount of outdated and inconsistent code comments, a program named the comment validator is presented in this study. The comment validator provides developers with the opportunity to manually validate code comments by segmenting code into three segments of code that needs to be manually validated: classes, methods and properties. The comment validator identifies when code segments have been modified after validation, therefore indicating that the code segments corresponding code comments could be outdated and inconsistent.

The comment validator were evaluated through functional testing and through a field study in order to test that the comment validator could reduce the amount of outdated and inconsistent code comments. The evaluation showed that the comment validator did remove outdated and inconsistent code comments when it were used according to the description presented in this study, therefore providing a new way to reduce the amount of outdated and inconsistent code comments in software development projects.

Keywords: Code comments, software documentation, design science
# Table of Contents

1 Introduction .................................................................................................................. 1
  1.1 Aim of this study .................................................................................................... 3
  1.2 Research method .................................................................................................. 3
    1.2.1 Research process .......................................................................................... 5
    1.2.2 Data generation and analysis ....................................................................... 6
2 Background ................................................................................................................... 8
  2.1 Code comments ...................................................................................................... 8
    2.1.2 Categories of code comments ..................................................................... 9
    2.1.3 Code comment guidelines ........................................................................ 11
    2.1.4 Specific application areas for code comments ........................................ 12
  2.2 Evolution, maintenance and updating practices of code comments .................. 13
  2.3 Programs for finding inconsistencies between code and their corresponding comments .......... 14
  2.4 Automatic documentation generators .................................................................. 15
3 The comment validator ............................................................................................... 17
  3.1 The LoginForm window ....................................................................................... 18
  3.2 The ProjectsForm window .................................................................................. 18
  3.3 The ValidationForm window ................................................................................ 21
4 Demonstration and evaluation of the comment validator ........................................... 25
  4.1 Functional testing of the comment validator ......................................................... 25
  4.2 Field study evaluation .......................................................................................... 29
    4.2.1 Development of the calculator ................................................................... 31
    4.2.2 The development process and evaluation of the calculator ....................... 32
    4.2.3 Conclusions for the evaluations .................................................................. 34
5 Discussion .................................................................................................................... 35
  5.1 Improvements for code comments ....................................................................... 35
  5.2 Comparison between the comment validator, automatic documentation generators and other programs with the same purpose .............................................. 35
  5.3 Discussion about the evaluation .......................................................................... 35
  5.4 Future improvements of the comment validator ................................................ 36
  5.5. The comment validator and software development methods .......................... 37
6 Conclusion and further research ............................................................................... 39
  6.1 Conclusion .......................................................................................................... 39
  6.2 Further research ................................................................................................... 40
References ....................................................................................................................... 41
Appendix 1: ..................................................................................................................... 45
LoginForm tests and outcomes: ................................................................. 45
ProjectsForm tests and outcomes: ............................................................. 45
ValidationForm tests and outcomes: .......................................................... 46
Appendix 2: ................................................................................................. 50
LoginForm requirements: .......................................................................... 50
ProjectsForm requirements: ..................................................................... 50
ValidationForm requirements: .................................................................. 53
List of figures

Figure 1: Block comment and line comments, written in the C# programming language. .......................... 8
Figure 2: Doc comment for the java programming language. ................................................................... 9
Figure 3: XML comment for the C# programming language. ................................................................... 9
Figure 4: Redundant comment restating the code. .................................................................................... 12
Figure 5: LoginForm window for the comment validator. ........................................................................ 18
Figure 6: ProjectsForm window for the comment validator. ...................................................................... 19
Figure 7: AddProjectNameForm window. ................................................................................................. 20
Figure 8: Choose folder for the location of the project. ............................................................................. 20
Figure 9: ValidationForm window for a project with two classes with the validation status “Not validated yet”. No methods and properties are viewed for this project. ................................................. 24
Figure 10: ValidationForm window for a project with all types of validation statuses for both classes and methods or properties. ......................................................................................... 24
Figure 11: Unmodified code for the tests 1-3. .......................................................................................... 27
Figure 12: Modified code for the tests 1-3 without any validation status change. Modifications consist of adding of empty lines, adding of extra whitespaces between words and adding and removal of comments. .......................................................................................................................... 27
Figure 13: The GUI for the calculator produced during the field study. ....................................................... 33
Figure 14: The code and the code comment for the method button20_Click. ............................................. 34
Abbreviations
SDP – Software development project
IS – Information systems
IT – Information technology
UML – unified modelling language
API – Application programming interface
HTML – Hypertext markup language
XML – Extensible markup language
IDE – Integrated development environment
DIR - Documentable item ratio heuristic
1 Introduction
During software development, different forms of software documentation is produced for the program that is being developed. Some forms of software documentation, such as system documentation, is produced to help software developers understand a program after it has been developed in order to make it easier to maintain the software and to help developers to further develop the software (Dennis, Wixom & Roth, 2010). One form of system documentation used to help software developers understand software is code comments. Code comments is a form of documentation for source code that consists of natural language annotations that is written within the source code. Code comments are produced to make the code easier to read and understand as well as to provide additional descriptions for the source code that the code itself can’t describe. Code comments have been proved to have a positive impact on code readability (Buse & Weimer, 2010; Tenny, 1988), an internal software quality that describes how easy it is to understand text. Some code comments, such as method comments, are also related to an increased “low-level program understanding” (Nurvitadhi, Leung & Cook, 2003).

Code readability is a software quality that is related to the maintainability of a software development project (SDP) (Buse & Weimer, 2010) because reading and understanding code is considered to be “the most time-consuming component of all maintenance activities” (Buse & Weimer, 2010). Some studies suggest that up to 70 % of the life-cycle of a SDP consists of maintenance work (Boehm & Basili, 2001). Because of the importance of readable and understandable code for software maintenance, it is important for software developers to improve the readability of a SDP to reduce the time spent reading and understanding the code while trying to maintain it.

Because of the relation between code comments and code readability as well as the relation between readability and maintainability, it could be assumed that code comments make code not only easier to read and understand, but also easier to maintain. This assumption is confirmed by de Souza, Anquetil & de Oliveira (2005) that found out that software maintainers consider code comments to be one of the most important software artifacts, only surpassed by the code itself. Similar results were gathered by Garousi et al. (2013).

Even if code comments is a highly usable component in SDPs for developers and maintainers, a potential problem with using code comments is that whenever code comments gets inconsistent with their corresponding code, the probability for future bugs increase according to Ibrahim et al. (2012). Other articles also mention that outdated and inconsistent code comments could result in future bugs as well as mislead developers (Jiang & Hassan, 2006; Tan et al., 2007).

Outdated and inconsistent comments could also lead to that code comments are ignored by developers. Siy & Votta (2001) stated in their study that most of the developers lost the “confidence in the reliability of the rest of the comments” and ignored all comments after an inconsistent comment has been encountered.

Various approaches have been proposed to solve the problems with outdated and inconsistent comments. Some of these approaches have been through the use of specific software programs such as the @tComment program where inconsistencies between doc comments and their corresponding code are identified (Tan et al., 2012). Another program with the same purpose is the docfacto Adam program in the docfacto toolkit (Docfacto, 2014). The docfacto Adam a
program that validates Javadoc comments to their corresponding code by identifying differences between code and its corresponding comments such as incorrect parameter tags and missing descriptions for tags. Other programs have been produced that are also able to solve the problems with outdated and inconsistent comments, even if this might not be the main purpose of these programs. One of these programs is the JavadocMiner (Khamis, Rilling & Witte, 2013) which is a program that calculates the quality of the in-line documentation for source code but also have the capability to identify outdated and inconsistent comments. The JavadocMiner uses a set of heuristics to measure both the quality of comments as well as the consistency of comments compared to their corresponding code, for example through the use of analyzing the correct amount of tags required for the comment compared to how many tags that are present.

Neither of the programs @tComment (Tan et al., 2012), docfacto Adam (Docfacto, 2014) and the JavadocMiner (Khamis, Rilling & Witte, 2013) check for the inline comments and all of these programs have limitations regarding their analysis of the content within the comments. Some researchers, such as Steidl, Hummel & Juergens (2013) oppose the use of some of these programs and considers that quantitative methods can’t identify outdated comments and other “useless” comments, such as the JavadocMiner (Khamis, Rilling & Witte, 2013) program does. Even if these programs are promising and useful, they still have several limitations, see chapter 2.3 for more information about these limitations.

Another type of programs that have been developed to solve the problems with missing, inconsistent and outdated comments are automated documentation generators with updating capabilities, such as the Comente+ (Zanoni et al., 2014). Even if this program is promising, the code comments lack the ability to capture the developers’ intention behind the code and rather produce comments that repeat the code. The automated code comment generators also lack the ability to capture particular aspects that could be contained within code comments, such as information describing why a more complex algorithm has been chosen compared to a simpler algorithm and information describing units of measurement for variables such as the comment “//in meters” for the variable “Length”.

This study will present a new way to minimize the amount of outdated and inconsistent code comments by expanding the concept of manual validations of code comments through a new computer program named the comment validator. This program will make it possible for developers to assure that the comments within a SDP are accurate. The code comment validator allows developers to manually validate the code comments within a SDP to assure that the existing code comments are describing its corresponding code correctly. This program identifies when code have been modified while the code’s corresponding comments haven’t been validated, therefore indicating possible situations where the comments could have become outdated and inconsistent with their corresponding code. This program therefore indicates which parts of the source code in a SDP that have reliable comments when the comment validator is properly used. Because of this feature both developers and maintainers could know when a comment is reliable and when the comments could be out of date in order to reduce the possibility of being misled by outdated comments.

In this study a design science instantiation (March & Smith, 1995) will be constructed showing how code comment updating practices could be improved through manual code comment validations performed with the comment validator program. This instantiation will be evaluated
for its functional properties to verify its functionality. The instantiation will also be evaluated through a SDP to show its utility and how the instantiation could be used.

This thesis is structured as following: In the first chapter of this thesis an introduction of the topic is presented as well as the aim of this study and the research method used in this study. In the second chapter a background for the project is described, describing what code comments are, how they are used in SDPs, the evolution of code comments within SDPs, how code comments are maintained and associated risks connected to outdated and inconsistent code comments, the programs available to find inconsistencies between code and their corresponding comments and descriptions of code comment generators. The third chapter describes the proposed instantiation and how it solves the issue with outdated and inconsistent comments. The fourth chapter contains information about the demonstration and evaluation of the proposed instantiation. The fifth chapter contains a discussion about the proposed instantiation, its evaluation, further research improvements as well as compare it to the current solutions available to minimize outdated and inconsistent code comments. The sixth and final chapter contains a conclusion about the work presented in this thesis.

1.1 Aim of this study
The aim of this study is to show how an instantiation can be produced to help developers to find and remove all outdated and inconsistent code comments within an SDP. This therefore leads to the research questions that this study will try to answer:

- Is it possible to produce an instantiation that helps developers to find all outdated and inconsistent code comments within a SDP?
- Is possible to produce an instantiation that could help developers to remove all outdated and inconsistent code comments within a SDP?

1.2 Research method
The research method chosen during this study were design science. Design science in the Information systems (IS) discipline is described as a “problem solving paradigm” (Hevner et al., 2004) that focuses on the development and evaluation of IT-related artifacts that are developed to solve organizational problems in order to provide utility. The artifacts produced in design science are of at least one of these four different types, according to March & Smith (1995):

- **Constructs**: An artifact based upon vocabulary concepts and definitions of it, such as the concepts of objects and classes in object oriented programming.
- **Models**: Artifacts that consists of combinations of constructs to representation. One example of this is charts for the unified modelling language (UML).
- **Methods**: Guidelines and frameworks for how to solve particular problems in IT. This involves both algorithms and methodologies, such as the agile methods.
- **Instantiations**: Artifacts that consists of a functional system that could show and demonstrate constructs, models, methods and other theories that could be applied in an IT context, such as medical expert systems that could diagnose patients.

Successful design science follows a set of guidelines, and several types of guidelines have been proposed to produce better design science research. One of the most referenced set of guidelines in the IS community is the set of guidelines proposed by Hevner et al. (2004), that this study
also follow. The guidelines proposed by Hevner et al. (2004) consists of seven guidelines. The guidelines presented by Hevner et al. (2004) are:

1. **Design as an Artifact**: A design science research project have to produce an artifact according to the four types of artifacts described by March & Smith (1995). In this study this guideline were applied because an instantiation according to March & Smith (1995) definition were developed.

2. **Problem relevance**: Each artifact proposed and produced in a design science research project should address a relevant and important business problem. The instantiation proposed in this study followed this guideline because the instantiation have the capability to remove the outdated and inconsistent code comments found within source code, therefore reducing future bugs and decreasing the risk of misleading developers through outdated and inconsistent code comments.

3. **Design Evaluation**: According to Hevner et al. (2004) “The utility, quality and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods”. In this study the proposed instantiation were evaluated through both functional testing and through a field study. In the functional testing the whole instantiation were tested according to a set of requirements for the functionality of the instantiation. The field study were performed to test the instantiation in a SDP and to test if outdated and inconsistent code comments could be removed if the instantiation were used during the software development process. The comments in the instantiation were compared with its corresponding code after development to determine the accuracy of the comments.

4. **Research contributions**: Design science research need to “provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies” according to Hevner et al. (2004). During this study this guideline were followed by presenting a new way to show how code comments could be maintained in order to minimize the amount of outdated and inconsistent code comments.

5. **Research Rigor**: Design science projects should be both constructed and evaluated according to the application of rigorous methods. This guideline were used and followed in this study by carefully following established guidelines for how to perform design science and by following an established research process in order to perform the research. These steps rendered in both a rigorous construction and evaluation of the proposed artifact.

6. **Design as a Search Process**: The guideline design as a search process is described by Hevner et al. (2004) as “The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment”. According to this principle, design is considered as an iterative process and solutions to design problems are encountered during a process of producing and evaluating design proposals until a satisfying solution is encountered. During the development of the proposed instantiation in this study, various design proposals were considered before the current solution were chosen. The design proposals were evaluated according to its properties and whether they were available at solving the research problem presented in chapter 1.1.

7. **Communication of Research**: Design science needs to be presented and communicated to multiple audiences, such as a technology oriented audience as well as a management oriented audience in order to describe its utility. This thesis is written in
such a manner that it is supposed to be understandable for both technology oriented and management oriented audiences. This is performed by describing technical concepts and how developers could benefit from using the instantiation and the benefits for management if developers within a software development team use this instantiation.

1.2.1 Research process

During this study, the guidelines proposed by Hevner et al. (2004) were implemented during the research process, a research process that were structured according to the activities described by Peffers et al. (2007). The research process when working with design science described by Peffers et al. (2007) is considered to be an iterative process consisting of six different steps. These steps, and how they were implemented in this study are:

- **Problem identification and motivation:** During this step the research problem is determined and the question why an artifact should be constructed is answered and described.

  This step were performed in the beginning of this study, during and after the literature review to define the research problem and to justify why this problem should be solved with an artifact.

- **Define the objectives for the solution:** Objectives for how a solution could solve the identified problem should be realistically defined. The objectives could be both quantitative to describe how a new solution is better than existing ones or qualitative to describe how a new artifact could solve problems previously unaddressed.

  In this study, this step were performed after the problem had been identified. The objectives in this study is composed quantitative objectives to describe how the proposed instantiation in this study performs better than existing solutions.

- **Design and development:** This step involves creating the proposed artifact and deciding what functionality the proposed artifact should contain.

  This step were performed after the objectives had been defined for the proposed instantiation. An instantiation were therefore developed according to the proposed objectives.

- **Demonstration:** The proposed artifact should be demonstrated to prove that it have the capacity to solve the identified problem. The demonstration could be performed as an “experimentation, simulation, case study, proof, or other appropriate activity” (Peffers et al., 2007).

  The demonstration of the proposed instantiation were performed as two separate steps in this study, both as a function testing in order to demonstrate the functionality of the developed instantiation and how it works under various conditions and as a field study to demonstrate the use of the instantiation in a SDP.

- **Evaluation:** During the evaluation the artifact is tested to observe whether the objectives of the artifact are met. The evaluation uses observed results from the demonstration of the artifact to test whether the objectives for the solution for the artifact are met. The evaluation could contain empirical evidence or logical proofs to evaluate the whether the artifacts objectives are met.
In this study the evaluation of the artifact were performed together with the demonstration of the artifact. Quantitative data and qualitative data were gathered during the demonstration of the artifact to prove that the objectives of the artifact were met.

- **Communication:** The research performed during a design science research project should be communicated, both the problem definitions, the artifact itself and its utility and the effectiveness of the artifact should be communicated to the appropriate audience for the design science research project.

The research project presented in this thesis is described in an suitable manner to present the instantiation that were developed during this study, why it was developed, its associated utility and how the instantiation were evaluated to assure that the proposed instantiation did solve its associated objectives. The thesis was written for both management oriented audiences as well as technology oriented audiences because both of these groups could benefit from implementing the instantiation proposed in this study in future SDPs.

### 1.2.2 Data generation and analysis

The data generated during this study comes from the function testing and the field studies that were gathered during the demonstration step in the research process and then analyzed in the evaluation step in the research process. The data generated during these studies were generated from observations and documents according to the definitions presented by Oates (2006). Quantitative data were collected from observations and qualitative data were collected from both observations and documents.

The quantitative data were collected when performing a set of tasks that were performed in order to develop software components as a part of the field study presented in chapter 4.

The generated quantitative data were consisting of information about the changes within the source code that happened during the field study, such as the amount of new methods added during each task and the amount of code segments that could contain outdated and inconsistent code comments within the SDP before the validation for each task.

The quantitative data were analyzed with a quantitative data analysis. During this analysis the gathered data were compared for each task performed to measure the observed differences.

The qualitative data collected during this study were collected from observations and documents generated during the demonstration step in the research process. The collected qualitative data were analyzed during the evaluation step. The qualitative data were analyzed with a qualitative data analysis. The qualitative data were generated during both the function testing and the field study described in chapter 4.

The data generated during the function testing were generated as field notes from observations, presented in its entirety in this thesis in appendix 2. Because the function testing were performed as a set of different requirements that needed to be fulfilled under different external conditions, the results from the tests for each requirement were documented as field notes according to what happened during these tests and how the program responded during the tests. The results from the tests for each requirement were then analyzed compared to how the program should have performed according to the requirement.
The documents data were gathered during the field study in the form of source code and the source codes associated code comments. The source code and the code comments were generated after each task during the field study had been performed. The generated data from the documents were analyzed with a qualitative data analysis in order to compare whether the code comments were accurately describing its corresponding source code.

The qualitative data that were gathered in this study were gathered in order to both prove that the proposed instantiation worked according to a set of objectives for how the instantiation should work, but also to prove that the solution worked in practice to meet the objectives that had been set for the instantiation, namely that the existing comments after validation were accurate and describing its corresponding source code correctly.
2 Background

This chapter contains information about the background behind the thesis and the theory that the thesis is based upon. This chapter is divided into four subchapters: In the first subchapter general information about code comments is presented together with information about categorizations, guidelines for how and when to apply code comments and information about specific application areas for code comments. In the second subchapter the evolution of code comments within SDPs and how code comments are maintained and updated within SDPs is presented. In the third subchapter the available programs to find inconsistencies between code comments and their corresponding code is presented. In the fourth subchapter the area of code comment generators is presented.

2.1 Code comments

Code comments is a source code documentation practice that allows a developer to add information within the source code of a program that isn’t compiled when the program is run. Comments are therefore used to be able to describe source code and also to be able to give additional information that a particular developer might consider important to describe through a comment, such as a description about why the code has been produced in a particular way (Michaelis, 2010). This type of comment could be used in order to describe why a complex algorithm have been chosen instead of a simpler algorithm, such as describing that the more complex algorithm is more effective than the simpler algorithm (McConnell, 2004). Comments could therefore contain any type of information that a developer want to include within a SDP in order to describe a code segment (Bell & Parr, 2009).

Two types of comments are commonly used, which are block comments, also sometimes described as delimited comments and line comments, also known as single line comments. A block comment is a comment that spans over several lines in the source code while a line comment is a comment that only spans over one line in the code, see Fig. 1. Line comments as such could be added at the end of a line of code, giving information about that specific line of code while block comments could be used to comment both whole sections as well as provide a comment within a line of code (Michaelis, 2010).

```csharp
// This is a line comment

/*
   This is a block comment
*/

Figure 1: Block comment and line comments, written in the C# programming language.

Modern programming languages have also added additional features through commenting structures, such as the doc comments for the Javadoc program for the java programming language and the XML documentation comments in the .NET languages. Doc comments are a type of code comments written to be used with the Javadoc program, which is an application programming interface (API) documentation generation program that uses the doc comments to generate external documentation from these doc comments. Doc comments have specific annotations they can use, such as the @param tags used to describe the parameters for a particular method. Doc comments could also be written together with the HTML markup language. The Javadoc program can extract and generate doc comments into other documentation forms, such as a website consisting of all available doc comments. This functionality have made doc comments popular when producing API libraries because it is
possible to produce well defined comments describing functionality of code, functionality that the developers might be able to access even if they can’t access the code behind the functionality (Oracle, 2012).

```java
/**
 * Calculates the square of the input variable n.
 * @author Add name and email address here
 * @version 1.0
 * @param n Input variable used to calculate the square root
 * @return The sum of n^2
 */
```

*Figure 2: Doc comment for the java programming language.*

The XML comments used in the .NET languages are a way to comment methods and classes in for the .NET languages. The XML comments are built upon the extensible markup language (XML), containing a flexible commenting standard that allows comments to contain several different types of fields with information, written as XML. The XML comments consists of two types of comments, delimited XML comments and XML single-line comments. The XML comments in the .NET languages have the possibility to generate full sized XML trees with all the XML comments contained, therefore it is possible to derive complete catalogs with information for all segments marked with XML comments. Because the comments are written as XML, this system have the possibility for developers to define their own sections that they would like to divide their comments information into (Michaelis, 2010).

```csharp
/// <summary>
/// This is an xml comment for a method for an auto generated button in c#
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

*Figure 3: XML comment for the C# programming language.*

### 2.1.2 Categories of code comments

Because the content, the purpose and the location of code comments could differ from comment to comment, several different categorizations of comments have been made in order to organize the different types of code comments that exists. One of these categorizations of code comments were presented by Steidl, Hummel & Juergens (2013) where comments were categorized depending on their location and their content. The different categories of code comments according to Steidl, Hummel & Juergens (2013) are:

- **Copyright comments**: A copyright comment is a type of comment that describes the copyright and license information for a particular file. These types of comments are usually found in the top of a file.
- **Header comments**: A header comment is a comment giving overview information about a class and could also give information about the author of the class and if the class has been peer reviewed or not.
- **Member comments**: This type of comment usually occurs before a method or a field and describes the functionality of the method or field.
- **Inline comments**: Inline comments are comments that occur within a method containing descriptive information about the implementation.
Section comments: Section comments are comments describing several methods or fields that share some similarity, such as a comment describing a set of getter and setter methods within a class.

Code comments: This type of comment contains code that have been commented out, either because the code might not be used right now but might be used later, or that a particular section of code has been commented out for debugging or testing purposes.

Task comments: Task comments are comments showing information for other developers about problems with a code snippet or information about possible improvements for a particular code snippet. Could also be named marker in the code (described below).

Other categorizations of code comments have been made based rather upon the information that a code comments contain. One of these categorizations is presented by Steve McConnell (2004), containing these six categories:

- **Repeat of the code:** This type of comment restates what the code says without telling why the code is acting in a certain way or describing the code in a more abstract way.

- **Explanation of the code:** This type of comment is used to describe and explain complex code. Steve McConnell (2004) describes that although these comments are useful for describing complex code, it often better to try to improve design of the code to make it simpler and more understandable rather that describing it.

- **Marker in the code:** A marker comment is a temporary comment that is describing a particular situation for the selected code snippet, such as an incomplete code section that should be improved before release. Although these code snippets should be improved rather than commented, Steve McConnell (2004) recommends standardization of how this type of comment is used to simplify the process of finding all these comments so that these code snippets could be improved before release. Some suggestions on how these comments could look like are:
  - // ***** Fix code section
  - // !!!!! Fix code
  - TODO: Fix code

- **Summary of the code:** This comment summarizes a few lines of code into a shorter section, therefore making it simpler for other developers to understand specific code snippets.

- **Description of the code’s intent:** This comment describes a section of code rather by describing what type of problem a code snippet is addressing, therefore describing the intent of the code that the developer is trying to address. According to Steve McConnell (2004), the summary comments and intents comments could be similar, even if this usually isn’t a large problem.

- **Information that cannot possibly be expressed by the code itself:** This category contains comments with information that the code can’t explain, such as copyright notices, version details, references to other documentation and other kinds of information that the code can’t explain.

According to Steve McConnell (2004), only comments of the three last categories are acceptable for completed code.
Comments could also be divided into two other categories depending on their purpose for other developers. Jan Skansholm (2013) describes these two categories as:

- Comments that are read by other developers for the purpose of modification of the selected code snippet.
- Comments that are read as part of a class library or as a part of an API, documenting classes and methods. These comments are only read for the purpose of giving understanding of how these methods and classes work so they could be used and called.

Depending on the on the type of comment in a system, as well as for what type of system the comment has been developed for, further categorizations have been made. Padioleau, Tan & Zhou (2009) constructed another taxonomy of comments for operating systems code based upon four categories:

- What: The content of the comment.
- Who: The audience of the comment, both who benefits from the comment as well as information describing the author of the comment.
- Where: The placement of the comment, both where the comment is located within a file as well as in what subsystem the comment where found.
- When: At what time where the comment written and how have the comment evolved over time?

Other forms of categorizations of code comments have also been performed depending of the purpose of the comment. Monperrus et al. (2012) constructed a taxonomy for API documentation based upon 23 kinds of API directives.

Another categorization that has been made has been developed according to the type of task comments that has been found in software. In a study by Ying et al. (2005) a set of categories were defined for task comments found in the integrated development environment (IDE) program Eclipse.

2.1.3 Code comment guidelines

No formal standards have been produced stating how code comments should be used and applied. Definitions for how code comments should be applied during development have been proposed by several software developers in various books. A recurring principle described by software developers in the software development literature is that comments should describe why and not how (McConnell, 2004; Goodliffe, 2006). The use of comments to explain the intention of the code and why the code exists is an area of application that even software developers such as Martin Fowler (2000) promotes, even if he considers code comments to often be superfluous.

Steve McConnell (2004) have proposed a set of recommendations for how comments could be used in an efficient manner. These recommendations cover how comments could be used to comment individual lines of code, paragraphs of code, data declarations, control structures, routines as well as classes, files and programs. The guidelines for how to comment code proposed by Steve McConnell (2004) consists of 46 different practices for how code comments should be applied and how it should be avoided. Some of the principles proposed by Steve McConnell are that variables should be commented with what type of unit they are measured by, such as the comment //In meter for the variable length, comments should be used to describe
the source of an algorithm if an external source have been used in order to develop the algorithm and endline comments used in order to describe maintenance work should be avoided.

Other software developers have constructed guidelines for how and when code comments should be managed, such as the recommendations presented by Robert C. Martin in the book Clean Code: A Handbook on Agile Software Craftsmanship (2009). The guidelines presented by Robert C. Martin consists of dividing comments into good comments and bad comments, where good comments are recommended for developers to use within a SDP while the bad comments is not recommended to be used. The guidelines proposed by Robert C. Martin consists of 8 recommended practices for commenting, such as using comments for legal information and warnings for other developers about possible risks and problems with a specific code segment. Robert C. Martin also discourages the use of 18 types of comments, such as commented-out code and comments restating the code without providing any additional useful information, see figure 4.

```java
/// <summary>
/// Returns the name of an author
/// </summary>
/// <returns>The authors' name</returns>
private string ReturnAuthorName()
{
    return AuthorName;
}
```

*Figure 4: Redundant comment restating the code.*

Other standards have been produced such as the commenting recommendations for GNU development projects (Free Software Foundation, 2015) even if these standards aren’t as exhaustive and expressive as the standard proposed by Steve McConnell (2004).

The guidelines proposed by Steve McConnell (2004) compared to the guidelines proposed by Robert C. Martin (2009) contains some differences while also containing some similarities. For example, both developers proposes that comments should be used to explain the intention of the developer. On the other hand, while Steve McConnell proposes that Javadoc comments should be used whenever possible, Robert C. Martin rejects the use of Javadoc comments in nonpublic code because “the extra formality of the Javadoc comments amounts to little more than cruft and distraction” (Martin, 2009).

An important aspect of code comments isn’t only the type of comment that is used to address a particular situation but also how the comment is formatted and how often a comment is occurring. According to Steidl, Hummel & Juergens (2013), longer comments are preferred in code. According to Capers Jones, as referenced to by Steve McConnell (2004) some studies suggest that an optimal comment density exists at roughly one comment at every 10 statements. Both fewer as well as more comments would reduce the code understandability.

### 2.1.4 Specific application areas for code comments

The use of comments have led to the development of specific types of software development practices, such as the pseudocode programming process (McConnell, 2004) where pseudocode is produced before the actual code is produced defining what the code should do and how it should do it. The pseudocode is written as comments and after development the comments are
kept to describe the corresponding code, which “eliminates most commenting effort” according to Steve McConnell (2004).

There exist other specific application areas for code comments, such as when code comments are combined with the software development strategy design by contract. Through design by contract the pre-conditions and post-conditions of a code snippet is stated before the code is developed, therefore assuring what the code should do and what results should be expected from the code snippet. When developing through design by contract, code comments are sometimes used to state this type of information, therefore not affecting the code produced.

2.2 Evolution, maintenance and updating practices of code comments

Some studies have been performed in order to understand the evolution of code comments within SDPs to understand whether the development of source code and comments co-evolve. In one of these studies, Arafat & Riechle (2009) studied the comment density in open source software code and found out that the code comment density remained similar to a large extent over time with a minor decrease over time. Although this study didn’t examine whether the code comments over time got updated and maintained or not, their study reached a similar conclusion as Jiang & Hassan’s (2006) study that examined the development of the program PostgreSQL. Jiang & Hassan (2006) came to the conclusion that the percentage of commented functions remained similar over time except for changes early in the project.

In a study by Fluri et al. (2009) the co-evolution of code and comments were studied. Their study showed that source code and comments have an even growth over time. Their study also suggested that depending on what type of entity the code is it will be commented differently. Their study suggested that comments and source code changes were related in more than 90% of the changes made in the comments with the exception of API changes, where the associated comments didn’t co-evolve even if those changes got re-documented later.

Even if the studies by Arafat & Riechle (2009), Jiang & Hassan (2006) and Fluri et al. (2009) indicate that code comments gets updated when the code changes, Tan et al. (2007) and Tan et al. (2012) identified several inconsistencies between code comments and their corresponding code within SDPs. These studies indicate that code comments doesn’t always get updated when their corresponding code changes, therefore resulting in that outdated and inconsistent code comments exists in several SDPs.

Some studies have also been performed in order to understand why comments get updated or not. Malik et al. (2008) studied why some comments becomes updated and why others doesn’t when their associated code becomes modified. Their conclusion were that the characteristic of the changes made to a code snippet were the most important variable to describe why some code comments gets updated relative to other code comments.

Other studies have been performed in order to understand whether outdated and inconsistent code comments result in a bug increase. Ibrahim et al. (2012) studied the relationship between updated code comments and software bugs. Their study suggest that inconsistent comment changes might not necessarily correlate with an increase in software bugs although their preliminary analysis suggest that some particular situations, such as inconsistent comment changes in subsystems that usually are updated increases the probability for future software bugs.
The area of how to update comments is a subject that yet has to be further defined, as stated by Ibrahim et al. (2012) in their conclusion: “More detailed, fine-grained analysis is needed to derive more concrete comment updating guidelines and to drive the development of methodologies and programs to prevent out-of-date comments”.

2.3 Programs for finding inconsistencies between code and their corresponding comments

To be able to find inconsistencies between source code and their corresponding comments, specific programs have been produced to solve this problem. One of the earliest solutions to find outdated and inconsistent code comments automatically were the iComment program produced by Tan et al. (2007). The iComment program is a program that relies on automatic identification of code comment inconsistencies through the use of natural language processing, statistic techniques, machine learning and program analysis techniques in order to identify differences and inconsistencies between code comments and their corresponding code. This program did find outdated and inconsistent comments within SDPs, although it had some limitations that could be improved in subsequent versions of the program, such as improving the accuracy of the program.

A subsequent approach to the iComment program were the @tComment program presented by Tan et al. (2012). The @tComment program introduced another way of finding inconsistencies between source code and their corresponding comments by focusing on doc comments written for the java programming language. The @tComment program “automatically analyzes the English text in Javadoc comments” (Tan et al., 2012) in order to imply probable properties for the doc comments corresponding methods. The @tComment program then “generates random tests for these methods, checks the inferred properties, and reports inconsistencies” (Tan et al., 2012). The @tComment program focuses on finding outdated and inconsistent code comments related to “null values and related exceptions” (Tan et al., 2012). Although the @tComment program did find outdated and inconsistent code comments, it has some limitations such as the focus on null values and exceptions which limits the @tComments’ ability to find as many outdated and inconsistent code comments as possible within SDPs.

Other programs that are similar to the @tComment program have been produced, such as the docfacto Adam program within the docfacto toolkit. The docfacto Adam is a doclet that validates the consistency between code and Javadoc comments while it also consists of a syntax checker (Docfacto, 2014). The docfacto Adam contains a set of customizable rules for how comments could be checked in order to find inconsistencies between doc comments and their corresponding code, such as missing descriptions for parameters, invalid parameter tags, methods without doc comments and missing descriptions for the return tags. Although the docfacto Adam is able to find structural differences between doc comments and their corresponding code, it is unable to find semantic differences written in natural language between source code and code comments such as incorrect descriptions for parameters.

Another program that were developed in order to find outdated and inconsistent code comments is the JavadocMiner. The program JavadocMiner were produced to assess both the quality and consistency of Javadoc comments compared to their associated source code (Khamis, Rilling & Witte, 2013) by using a set of quantitative quality metrics. The JavadocMiner automatically estimates the quality and consistency of code comments through the use of a set of heuristics. One of these heuristics used is the “documentable item ratio heuristic (DIR)” (Khamis, Rilling
(Witte, 2013) that calculates the ratio between the documented parts in Javadoc comment, such as the tags for the parameters and the parts that should have been documented. This metric will therefore result in a value between 0 and 1 where 0 indicates that no parts were documented and 1 indicates that all parts were documented.

The JavadocMiner program by Khamis, Rilling & Witte (2013) have been criticized by Steidl, Hummel & Juergens (2013) based upon their earlier proposal, the article “Automatic Quality Assessment of Source Code Comments: The JavadocMiner” (Khamis, Witte & Rilling, 2010). Steidl, Hummel & Jeurgens (2013) criticize whether some of the metrics used in the JavadocMiner are able to measure the meaningfulness of comments.

The @tComment, the docfacto Adam and the JavadocMiner programs focus on specific comments, namely Javadoc comments. Because these programs only focus on Javadoc comments it also means that they neglect other types of comments, such as inline comments. The neglect of inline comments could therefore result in that the inline comments for a SDP could be inconsistent with their corresponding code, therefore misleading developers and causing the introduction of bugs.

A recurring problem with the iComment program, the @tComment program, the JavadocMiner and the docfacto Adam is that all of these programs are limited in their ability to identify outdated and inconsistent code comments even if their inability is shown in various forms, such as only focusing on doc comments or a limited accuracy for identifying outdated and inconsistent code comments. These programs also lack the ability to identify outdated and inconsistent comments that provide information that can’t be compared with its corresponding code, such as comments describing units of measurements for variables and comments referring to external sources of information. For more information about these issues, see chapter 3.

2.4 Automatic documentation generators

Another alternative to reduce the amount of outdated and inconsistent code comments compared to the programs described in chapter 2.3 is by using automatic documentation generators instead of having developer written code comments. Automatic documentation generators are programs that create code comments within SDPs by constructing comments from information within the source code, such as statements and identifiers (McBurney & McMillan, 2014). One of these programs is the Comente+ presented by Zanoni et al. (2014), a program that have the capability to create and update code comments with information that has been gathered from other artifacts of a project. Even if this program is promising, it still lacks the possibility to capture the context for a particular section of code. Other programs have been produced to automatically produce comments but to also include the context of the comment to describe why a method works in a particular way, such as the program presented by McBurney & McMillan (2014). Although their program have interesting capabilities, it lacks the possibility to verify whether previous comments are consistent with their corresponding code.

The program presented by McBurney & McMillan (2014), as well as the Comente+ program contain some limitations because they only rely on the content found within the source code. These automated comments lack the possibility to contain information describing, for example, why a more complex algorithm has been chosen compared to a simpler algorithm. These programs also lack the possibility to describe the intention of the developer for a code segment when the developers’ intention differ from the actual code. Another limitation with the automated documentation generators exist when method names and variable names doesn’t
describe their intention and their functionality, which could result in problematic comments. Automated code comments could also result in comments of the type “repeat of the code” (McConnell, 2004) therefore only restating the code without giving any additional information to describe why the code exists and why it works in a particular way.
3 The comment validator

In this chapter an instantiation is presented that were constructed as a part of this thesis, an instantiation which will be able to find outdated and inconsistent comments in order to help developers to remove outdated and inconsistent comments.

Before the instantiation were developed, a set of objectives were defined to describe how an instantiation could provide a solution to the research questions defined in chapter 1.2. The objectives behind the proposed instantiation were:

- The proposed instantiation should be able to find all possible situations where code comments could be outdated and inconsistent. This could be compared to the current solutions for finding outdated and inconsistent code comments that are only able to find some of the outdated and inconsistent code comments.
- The use of an instantiation should be able to remove all outdated and inconsistent code comments within a SDP all along the whole development process as well as during the whole maintenance process. In comparison to the other solutions that also removes outdated and inconsistent code comments, those solutions are only able to remove some of the outdated and inconsistent comments that exist while the instantiation proposed in this study should be able to help developers to remove all outdated and inconsistent code comments.

To be able to come up with a solution that is able to solve the objectives for the instantiation, an instantiation were developed based upon manual validations of code comments performed by developers. Manual validations of code comments were chosen because manual validations are necessary to perform in order to guarantee the correctness of code comments when following some of the code comment guidelines mentioned in chapter 2.1.3., such as the guidelines proposed by Steve McConnell (2004). For some types of comments that are recommended according guidelines proposed by Steve McConnell (2004), such as “Comment the units of numeric data”, “Comment coded meanings”, “Comment anything that gets around an error or an undocumented feature in a language or an environment”, “Justify violations of god programming style” and “Document the source of algorithms that are used”, manual validations performed by a developer is necessary in order to ensure that their intention is correctly described. These types of comments are impossible to validate with automatic solutions because they rely on the ideas and assumptions made by developers. This could be exemplified with the guideline “Document the source of algorithms that are used” (Steve McConnell, 2004) where the wrong source of the algorithm would mislead a developer in their understanding of whether the code or the code comment is correct according to the situation. This type of comment can’t be validated with an automatic solution because no computer system would be able to tell which of the two proposed solutions that would be the correct one without any additional information, only the developer of the code for the algorithm could know whether the code or the referenced algorithm is the “correct” one in that particular situation.

Manual validations of code comments performed through an instantiation were also chosen because code comments could be validated for their correctness by the same developer who has produced them, therefore only one developer is needed to perform this process. This makes the proposed procedure simpler and less labor consuming compared to if practices used for ensuring code quality would be used in order to guarantee the correctness of code comments. Practices for ensuring code quality, such as pair programming and formal inspections, requires
more than one developer to analyze and understand the code and its associated code comments and are therefore more labor consuming to use than the comment validator if only the correctness of code comments needs to be validated.

To reduce the amount of outdated code comments through manual code comment validations, an instantiation named the comment validator were constructed. The comment validator is a program that encourages developers to manually validate the correctness of code comments found within source code. The comment validator identifies classes within a SDP and the classes’ corresponding methods and properties, all of them which needs to be manually validated by a developer in order to validate the comments within a SDP. This will then assert that the comments for a code snippet, which is either a method, property or a class, is updated and therefore describing its’ corresponding code accurately.

3.1 The LoginForm window
When starting the comment validator, the first window shown is the LoginForm window. In the LoginForm window the user of the comment validator needs to log in with a user name consisting of both a first name and last name, preferably with the users own name. The login functionality is used to later on be able to store who has validated a method, a property or a class. This is stored because if problems arise with the understanding of a particular code comment it is possible to contact the person responsible for the validation of the code comment to understand the code comment better. In the login form, the user have the possibility to store the login name so that the same name will be shown the next time the user uses the comment validator, therefore the user doesn’t have to write the same information the next time. If a user logs in without checking the store login information checkbox, the text fields first name and last name will be empty in the login form the next time the comment validator is opened.

![LoginForm](image)

Figure 5: LoginForm window for the comment validator.

3.2 The ProjectsForm window
After a user have logged in to the comment validator the ProjectsForm window is opened, a window where the user is able to add, remove and open SDPs to have the comments within an SDP validated. The ProjectsForm window consists of one table named the Projects table containing two columns named Project and Locations, the four buttons Open project, remove project, add project and log out and in the upper left corner of the ProjectsForm a line of text exists presenting the name of the user who is currently using the comment validator. The Projects table presents the projects that have been added to the comment validator where the
names of the projects are presented in the Project column of the table and the location of the projects are presented in the Location column.

The functionality presented in the ProjectsForm window is created to give the user the ability to in a simple and efficient manner manage multiple SDPs within the comment validator. This functionality is particularly useful in organizations where individual developers could be involved in multiple projects occurring at the same time, therefore this functionality will make it easier for these developers to manage the comment validations between several projects. When a project has been added to the comment validator, it is possible to choose this project to start the validation of the comments within the project.

![ProjectsForm window for the comment validator.](image)

Projects are added to the ProjectsForm through the “Add project” button. When pressing the “Add project” button, a chain of events starts in order to add a new project to the projects table in the ProjectsForm as well as to store the project within the comment validator. The first step when adding a new project occurs after the add project button have been pressed, which results in that a new window named AddProjectNameForm is showed. The AddProjectNameForm consists of one text field and one button labeled “Add project name”. In the
AddProjectNameForm, a name for the new project should be written in the text field and when a name have been written, the add project name button should be pressed to continue the procedure of adding a new project. If no name is written in the text field, a warning message occurs stating that a name needs to be added for the project. If a name for a new project already exists as a stored project, a warning message occurs stating that a project with the same name already exists.

![AddProjectNameForm window.](image)

When a name for a project have been added, the folder for the selected project needs to be chosen from a list of folders. When a folder have been selected, the user needs to press the OK button in the select folder window. If the folder is the desktop folder, a warning message occurs stating that the folder location is invalid and the procedure of adding a project is stopped. If the procedure of adding a project is stopped by pressing either the abort button in the bottom right corner or the red cross in the upper right corner, a warning message occurs stating that the location for the project is invalid and the procedure of adding a project is stopped. If another folder than the desktop folder is selected and the user proceeds the process of adding a project through the OK button, the new project will be stored in the comment validator. In the ProjectsForm window, the project will be added in the projects table and the name of the project will be presented in the Projects column and the location of the project will be presented in the locations column.

![Choose folder for the location of the project.](image)
When pressing the remove project button in the ProjectsForm window, a selected project from the projects table is removed both from the projects table as well as from its stored location within the comment validator. If no item is selected in the projects table, a warning message occurs stating that an error occurred.

When pressing the log out button, the ProjectsForm window closes and the LoginForm window opens, therefore giving a user the opportunity to change the login name.

The open project button opens a selected project in the ValidationForm. For more information about the ValidationForm, see chapter 3.3.

3.3 The ValidationForm window

When a SDP have been selected in the projects table and the selected SDP is opened through the button open project, a new window is opened named the ValidationForm. In the ValidationForm window the manual validations of code comments are performed. The ValidationForm window contains two tables, one showing the classes contained within the SDP and another that shows the methods and properties found within a class. The ValidationForm window contain a set of buttons labeled update classes, find methods and properties for selected class, validate method and validate class. The ValidationForm window also contain a line of text in the upper left corner of the window showing information about who is logged in to the comment validator and a line of text above the methods table showing information about which class the methods in the table are belonging to.

When a SDP is opened for the first time in the ValidationForm window, the comment validator creates a folder named CommentValidations within the folder of the SDP. The comment validator then adds XML-files to the CommentValidations folder for each class contained within the SDP. The XML-files contains all the variables, the methods and the properties found within a class that exists when the SDP is added to the comment validator. The methods and properties are stored in the XML-files with their whole bodies and the variables are stored with their individual, physical lines of code, word by word. Empty lines and additional whitespaces between words are removed before the variables and methods are stored. No comments are stored in the XML-files for either the classes, the variables, the methods or the properties. If a project already contains a CommentValidations folder when the project is opened, no new folder will be added and created.

In the ValidationForm window code comments are validated by validating code segments that could contain code comments. The code segments validated in the comment validator are classes, methods and properties. To validate a code segment the user should press the associated validation button in the comment validator for the code segment. By doing this for a code segment, the status of the code segment is changed, therefore indicating that the code segment has been validated by a developer for its correctness. To validate a class requires that all of the methods and properties contained within the class have been validated first, otherwise the class can’t be validated.

The code segments classes, methods and properties are identified in the ValidationForm with their validation status. Four different validation statuses exists for the code segments and these are: “Validated”, “Out of date”, “Not validated yet” and “New method/New class”. For the code segments methods and properties, the different validation statuses occur under these conditions:
If a method or a property have had their comments validated already and its current body is the same as the stored, previously validated method or property, its status will indicate that the method or property’s comments are validated and correct, therefore it will have the validation status “Validated”.

If a method or property have previously been validated, but the body of the current method or property is different from the stored, validated version, this method or property will have a validation status indicating that the comment could be out of date. These methods and properties will have the validation status “Out of date”. Some changes that doesn’t affect the functionality of the method or property, namely changes considering empty lines, additional whitespaces and comments doesn’t affect the validation status for a method or a property.

If a particular method or property existed within the SDP when the SDP were opened in the comment validator for the first time and the method or property haven’t been validated yet, the method or property will contain the validation status “Not validated yet”.

If a method or property have been added to the SDP after the SDP were opened in the comment validator the first time the method or property will have the validation status “New method”.

This process of storing and comparing validations is used to identify methods and classes where developers might forget to update the comments by indicating segments where comments could be outdated. The difference between the validation statuses “Not validated yet” and “New method/New class” are used to indicate new changes performed to a SDP after the comment validator have been established as a validation program for comments for a SDP.

Regarding the validation status for classes, the classes contain the same types of validation statuses as methods and properties with the exception that “New method” is called “New class”. The different validation statuses for classes occur under these conditions:

- For all of the classes that existed when the SDP were opened for the first time in the comment validator and that haven’t been validated yet, their validation status will be “Not validated yet”.
- For all of the classes that are added to the SDP after the SDP were opened in the comment validator for the first time and that haven’t been validated yet, their validation status will be “New class”.
- For all of the classes within the SDP that have been validated and therefore contains the same methods and properties as the stored XML-file for the class does as well as contains the same variables, their validation status will be “Validated”.
- For all of the classes that have previously been validated but either contains differences in the methods and properties for the class compared to the class’ XML-file or that contains differences in the variables, this class will have the validation status “Out of date”. Some specific conditions doesn’t affect the validation status for classes and these are: The location of variables, additional empty lines and additional whitespaces in variables and the adding and removal of comments.

Regarding the tables contained in the comment validator, the table for the classes shows all the classes within a project, their validation status, when the class where validated last as well as who validated the class last time. This information is showed in their associated columns named
class, validation status, last validated and validated by. The table containing the methods and properties for each class shows all the methods and properties within a class with their name and input parameters, the methods and properties validation status, when the methods and properties were validated the last time, who validated the method or property the last time and whether the item in the table represents a method or a property. The information for the methods and properties is presented in their associated columns named methods, validation status, last validated, validated by and method type.

The cells in the columns for validation status both in the class table and in the methods and properties table contain different background colors depending on the validation status for the represented object. For objects with the validation status “Validated”, their validation status cell will be green. For objects with the validation status “Out of date”, their validation status cell will be red. For objects with the validation status “Not validated yet”, their validation status cell will be gray. For objects with the validation status “New class” or “New method”, their validation status cell will be yellow.

In order to validate methods, properties and classes, the user have to interact with the buttons presented in the ValidationForm window. The functionality of the buttons aren’t restricted to validate classes, methods and properties, although all the buttons functionality are related to this purpose. The functionality for the buttons in the ValidationForm are:

- The update classes’ button updates the status of the classes within the SDP.
- The find method and properties button displays the methods and properties found within a selected class in the table for methods and properties. If no class has been selected when the button is pressed, a warning message occurs stating that no methods can be viewed. If the class contains no methods or properties, a warning message occurs stating that no methods or properties exists within the class.
- The validate method or property button validates a selected method or property and stores its content. If no method or property is selected when the validate method button is pressed, a warning message occurs stating that no method or property can be validated.
- The validate class button validates a class if all the methods and properties found within the class has been validated first. If the button is pressed in order to validate a class that contains methods or properties that haven’t been validated yet, a warning message occurs stating that the class contains methods or properties that haven’t been validated yet and that these methods or properties needs to be validated before the class can be validated. If no class is selected when the validate method button is pressed, a warning message occurs stating that no class can be validated.
Figure 9: ValidationForm window for a project with two classes with the validation status “Not validated yet”. No methods and properties are viewed for this project.

Figure 10: ValidationForm window for a project with all types of validation statuses for both classes and methods or properties.
4 Demonstration and evaluation of the comment validator
In this chapter the demonstration and the evaluation of the comment validator is presented.

The comment validator is demonstrated and evaluated through functional testing where the comment validator is tested according to a set of requirements and through a field study to demonstrate and evaluate how the instantiation works within a SDP.

4.1 Functional testing of the comment validator
A functional testing of the comment validator were performed to test and demonstrate that the comment validator worked according to its functionality presented in chapter 3. During the functional testing the whole application were tested after it had been developed. The functional testing were performed by testing the comment validator according to a set of requirements that were constructed describing how the comment validator should work. During the functional testing various test cases where constructed to test each requirement and to find possible errors with the comment validator.

The requirements that were tested were sorted into three categories based upon the three different windows that the comment validator consists of. For each individual requirement, a set of test cases were constructed and performed in order to evaluate whether the program could meet the requirement and to find any potential problems with the program. After the test cases for each requirement had been performed and the result of the test cases had been observed, a conclusion were drawn from the result regarding whether the requirement had been fulfilled or not.

The three forms presented in chapter 3 were tested individually during the functional testing. During the tests for the LoginForm window the following functionality were tested:

- It should be possible for a user to log in to the comment validator when the text fields for a first and a last name exists.
- It should be possible for a user to store a login name so the same name is viewed in the textboxes the next time someone opens the comment validator.

The test cases for the proposed functionality for the LoginForm passed which proved that the requirements for the LoginForm had been fulfilled.

During the tests for the ProjectsForm window the following functionality were tested:

- A user should be able to add a new project to the projects table in the comment validator according to the process described in chapter 3.2. This project should also be stored and viewed in the projects table with its name and the location of the project.
- A user should be able to remove a previously stored project from the comment validator, therefore also removing it from the projects table.
- It should be possible to open a selected SDP, therefore opening the ValidationForm window for the selected project in order to validate classes, methods and properties.
- It should be possible to log out from the ProjectsForm window, therefore returning back to the LoginForm window.

Test cases that were performed in order to test the proposed functionality for the ProjectsForm passed, therefore proving that the requirements for the ProjectsForm had been met.
During the tests for the ValidationForm window the following functionality were tested:

- When first opening the ValidationForm window, all user formatted classes within the SDP should be shown, in the class table with the classes’ corresponding information according to the description in chapter 3.3.
- It should be possible to find methods and properties for selected classes, the methods and properties for that class should be shown in the methods table according the description of this functionality in chapter 3.3.
- It should be possible to validate selected methods or properties. These methods and properties will then receive the validation status “Validated”.
- It should be possible to validate selected classes if these classes have had all of their methods and properties validated first. After a class have been validated, its validation status should change to “Validated”.
- The validation status for classes, methods and properties should change according to their descriptions in chapter 3.3.
- It should be possible to update the validation status of the classes shown within the ValidationForm window.

During the functional tests for the ValidationForm window, although most of the tests passed, one of the tests for requirement 4.1 failed to meet the requirement. Below are three of the requirements for the ValidationForm and their tests presented in order to illustrate how the tests for the requirements were performed. These tests illustrates both tests that passed and met their requirements and the test that did not meet its requirement. These particular tests were selected to illustrate the functional testing because these selected tests were the tests that best illustrated how the more complex functionality for the program were tested and the complexity of testing some of the features for the comment validator. A table containing a summary for each test and their outcome is presented in appendix 1. The complete set of the tests, requirements and how the tests were performed is presented in appendix 2.

**Requirement 4.1:** If a previously validated method gets modified without becoming re-validated, its validation status in the methods table in the ValidationForm window should change from validated to “out of date” and get a red background. Modifications that should change the validation status of the method are all character changes made to the method, both by replacing the existing characters as well as adding new characters and removing characters. A few exceptions exist that should not alter the validation status. Modifications that should not alter the validation status are adding new empty lines to the method, adding new whitespaces occurring outside parentheses and brackets (as long as each individual word could be separated through whitespaces) and when comments in any form is added or removed.

**Test:** To test this requirement, five tests were performed to see which changes made within a method that could change the status of a method from validated to out of date.

In the first test, new empty lines were added within the method. This test showed that new empty lines did not affect the validation status.

In the second test whitespaces were added to lines of code outside parentheses and brackets. These whitespaces were added at several different places within the lines of code, both before the code, within the code and after the code. This test showed that whitespace changes did not change the validation status of the method.
The third test considered the adding and removal of comments. In this test comments were both added before lines of code, at the end of lines of code and within lines of code. In this test both line comments and block comments were used and comments were not only added, but previously stored comments were removed from methods to test that comments shouldn’t affect the validation status. The results of the third test did show that both adding and removing comments didn’t affect the validation status of a method or property. Figure 11 and figure 12 illustrates code for the tests 1 -3 both before and after the modification.

```csharp
/// <summary>
/// Adds an operator to the NumbersAndOperators list if a number
/// precedes it, if no number precedes it in the NumbersAndOperations list
/// A zero is added first
/// </summary>
/// <param name="Operator">An operator of any type</param>
private void AddOperatorToNumbersAndOperations (string Operator)
{
    if (LastNumber.Count > 0 && textBox1.Text != "0")
    {
        // Adds a number to the NumbersAndOperations list
        NumbersAndOperations.Add(String.Concat(LastNumber));
        LastNumber.Clear(); // Clears the LastNumber list
        NumbersAndOperations.Add(Operator);
        textBox1.Text += "/TestComment" + Operator;
    }
    if (textBox1.Text == "0" && LastNumber.Count == 0)
    {
        NumbersAndOperations.Add("0");
        NumbersAndOperations.Add(Operator);
        textBox1.Text += Operator;
    }
}
```

*Figure 11: Unmodified code for the tests 1-3.*

```csharp
/// <summary>
/// Adds an operator to the NumbersAndOperators list if a number
/// precedes it, if no number precedes it in the NumbersAndOperations list
/// A zero is added first
/// </summary>
/// <param name="Operator">An operator of any type</param>
private void AddOperatorToNumbersAndOperations (string Operator)
{
    if (LastNumber.Count > 0 && textBox1.Text != "0")
    {
        NumbersAndOperations.Add(String.Concat(LastNumber)); // New comment
        LastNumber.Clear();
        NumbersAndOperations.Add(Operator);
        textBox1.Text += Operator;
    }
    if (textBox1.Text == "0" && LastNumber.Count == 0)
    {
        NumbersAndOperations.Add("0");
        NumbersAndOperations.Add(Operator);
        textBox1.Text += "TestComment/" + Operator;
    }
}
```

*Figure 12: Modified code for the tests 1-3 without any validation status change. Modifications consist of adding of empty lines, adding of extra whitespaces between words and adding and removal of comments.*
In the fourth test whitespaces were added within parentheses, braces, brackets and chevrons, and the changes for the validation status were observed after each whitespace had been added. This test showed that whitespace changes made within parentheses and brackets did change the validation status of the method, while whitespace changes occurring within braces or chevrons did generally not change the validation status, with the exception of whitespace changes made between two chevrons located between each other such as “=>” going to “> >”.

In the fifth test character changes were made outside parentheses, brackets, braces and chevrons and these characters changes did not consist of whitespaces, empty lines or the adding or removal of comments. In this test single character changes were made, such as changing lower-case characters to upper-case characters, adding and removing extra characters to variable names, adding whole new lines of code, removing lines of code and adding and removing of other special characters such as changing “=” to “=>”. After each character change were made, the validation status were controlled to see whether it had changed or not. After the validation status had been controlled, the changes were reverted to ensure that the validation status before a new change would be tested always were “Validated”. The result of this test showed that for all of these situations, the validation status did change from “Validated” to “Out of date”.

**Requirement status:** Not passed. Even if most of these tests did work under the conditions it should have performed, this requirement did not pass because of a minor incident with whitespace changes occurring between chevrons. Although this problem shouldn’t be a problem in most situations, it is a situation that needs to be considered and could be fixed and updated in future versions.

**Requirement 5.2:** When the validation process proceeds without any warning messages occurring for a selected class, the validation status for the selected class should change to “Validated” and the validation status cell for the selected class should get a green background.

**Test:** To test this requirement, four test were performed. In the first two tests two classes were validated that have had all of its methods and properties validated. The validation status for these classes were “Not validated yet” for the class in the first test and “New class” for the class in the second test. The results of both these tests were that after the validate class button were pressed the validation status for both classes were changed to “Validated” and the validation status cell got a green background. In the third and the fourth tests classes without any methods or properties were validated, one with the validation status “Not validated yet” and the other with the validation status “New class”. The results of the third and fourth tests were that both classes got the validation status “Validated” and their validation status cell got a green background.

**Requirement status:** Passed.

**Requirement 5.3:** When a previously validated class becomes modified, its validation status should be changed from “Validated” to “Out of date” and the validation status cell should receive a red background. These changes concern changes made both within the methods and properties of the class as well as outside the methods and properties. Some changes shouldn’t affect the status of the class and these are extra whitespace changes on a line of text as long as each individual word or set of symbols are separated by whitespaces, adding or removal of empty lines, adding or removal of comments and separating the code into regions.
**Test:** To test this requirement, a set of changes were made during a set of tests to a validated class and after each individual change the validation status were observed.

In the first test empty lines were added to a class, both within the methods of the class as well as outside the methods within the class. This test showed that the validation status remained intact even after empty lines had been added to the class file.

In the second test extra whitespaces were added to lines of code both within the methods and properties in a selected, validated class as well as in lines outside the methods and properties in the selected, validated class. The validation status of the class remained intact even after these whitespace changes to the class.

In the third test comments were added to a class and these comments where both XML-comments located before methods, line comments located above lines of code, block comments located above lines of code, line comments occurring at the end of a line of code, block comments occurring at the end of a line of code and block comments occurring within lines of code. The validation status of the class remained the same after these changes had been performed.

In the fourth test regions were added to a class through the use of the keyword #region to begin the region and #endregion to show when the region did end. This test showed that adding and removing regions to a class did not affect its validation status.

In the fifth test, the validation status were tested when new lines of code were added to a class. To test this new lines of code were added to both methods within the class and outside the methods within the class. After each new line of code had been added, the validation status of the class were observed and if the class had become out of date, the class were validated. This procedure were then repeated for each new added line of code. This test showed that for each new line of code that were added, either if it were inside a method or not, the validation status were changed for “validated” to “out of date”. This test passed.

A test were performed where individual characters in lines of code both inside and outside methods and properties in a selected class were changed in order to test whether character differences could be recognized. The character differences that was tested were: lower letter characters in variable names were changed to upper letter characters, additional characters were added to variable names to extend the variable names, sequences of non-letter, non-number characters were extended, such as > were extended to >= and characters were removed. The result of all these tests were that the validation status went from “Validated” to “Out of date” as expected.

**Requirement status:** Passed.

**4.2 Field study evaluation**

A field study evaluation were conducted to demonstrate and evaluate the use of the comment validator. The field study were conducted to test whether the comment validator were able to remove the occurrence of outdated comments and therefore reduce possible bugs and misleading comments stemming from outdated comments. Due to time limits the possibility to test whether the software could solve all these problems were hindered, therefore the evaluation were restricted to test whether the source of these problems could be tested and evaluated, namely by testing the accuracy of the code comments produced.
To be able to test whether the comment validator could produce more accurate code comments, a SDP were performed where code comments were compared to their corresponding code after the code comments had been validated. The SDP were composed of producing a calculator consisting of a set of small and simple features. Due to time limits a more complex program couldn’t be developed and because a calculator could be composed of several features, it is easy to test the comment evaluator at several times during the development process.

Before starting the development a set of different strategies were considered depending on how and at what point during the development process that the comment validator should be used. The different strategies that were considered for when the comment validator could be used to perform the validations in this project were:

1. **After a segment of code have been developed or modified.** During this validation strategy segments of code, such as methods and classes, could be continually validated in order to reduce the time spent reading and understanding code when validating code. The assumption behind this evaluation strategy is that this strategy might be able to reduce the time spent reading and understanding code and code comments because segments of code are validated directly after being developed. Because code comments are validated directly after code have been developed or modified, the developer might not have to go back to segments of code to try to understand what a segment of code does and what the actual intentions of the developer were when that particular segment of code were developed.

2. **After a task have been performed.** During this type of development the comments within a SDP are validated after some form of task have been performed. This situation could be used to simulate the use of how the comment validator could be used in an actual working environment where development projects are built up by different tasks that are performed in order to produce a complete system. In this situation comments could be validated before their corresponding code is submitted to a project and considered to be finished. This would assure that the comments submitted together with the code is accurate. Compared to the first strategy presented this strategy would reduce the amount of times that the comment validator might be used, therefore simplifying the process of how to use the comment validator. The disadvantage with this strategy is that more time might be spent reading and understanding code and code comments when validating code comments compared to the first strategy.

3. **After a period of time have passed, such as at the end of a work day.** When using this validation strategy comments are validated after a period of time have passed, such as at the end of a working day or before each weekend. This situation could be used to minimize the amount of times that the comment validator would have to be used in a project yet still retain the practice of validating comments at recurring intervals. The drawbacks of using this strategy might be that comments might need to be revalidated several times during the development of a task if the task spans over several intervals. Another drawback of only using this strategy is that if code is submitted to a software project before the end of an interval, code without validated comments might be submitted to a SDP which would therefore reduce the purpose of the comment validator. This strategy might therefore work if it is combined with one of both of the other strategies but to use it alone could reduce the purpose of using the comment validator.
During the field study evaluation performed in this thesis, the second validation strategy were used. This strategy were selected because this type of strategy simulates how the comment validator might be used in a real workplace scenario where code is submitted to a shared SDP after tasks have been performed.

Before the development of the calculator started, a set of commenting guidelines were defined to describe how the code for the calculator should be commented. The commenting guidelines were consisting of a requirement of at least one summary or intention comment for each method and property developed and at least one summary or intention comment for each class created by a developer or containing code created by a developer.

4.2.1 Development of the calculator

Before the development of the calculator started, the structure and functionality of the calculator were planned and the components that the calculator should contain were established and written down as a set of tasks. Because the development of the calculator were developed according to the second type of validator strategy in the list above, validations of code comments were performed after each task had been fulfilled and new features had been developed and added to the calculator. During these validations, all the methods and classes within the project were validated.

The proposed calculator were defined as a calculator containing a set of arithmetic operations, but with the possibility to produce arithmetic operations containing multiple arithmetic operators and parentheses. The calculator were produced as a windows forms project and all of the operations performed through the calculator were considered to be performed by pressing buttons rather than writing. The calculator did also contain a text field that showed the current operation. The functionality that the calculator would contain were produced as a list of tasks that should be performed in order to produce the calculator. The tasks and the order in which they were performed to be able to develop the calculator were:

1. **Design the interface for the calculator.** The interface should contain a text field for the numbers and the operations performed and the text field should also be able to show the result of performed operations. The interface should also contain a set of buttons for the number 0-9, a decimal dot to create decimal numbers, a delete button to delete the previously inserted numbers or operations, a button to reset the current operations, an equality button to calculate a result and buttons for the arithmetic operations addition, subtraction, multiplication and division.

2. **Create the functionality for the buttons for the numbers.** During this task the functionality for the numbered buttons should be developed so that whenever a button is pressed, the buttons number is added to the calculator’s text field.

3. **Create decimal function.** This tasks should add a dot in the text field for a number and make it possible to therefore create decimal numbers.

4. **Create reset function.** This function should reset the text field and any operations performed.

5. **Create equality function.** This function should make it possible to get the result of previously performed operation.

6. **Create addition function.** This function should make it possible to perform an addition operation between two numbers by using the addition button. This function should add a “+” operator in the text field, indicating that an addition operation has been performed.
7. **Create subtraction function.** A subtraction function should be added so that the operation subtraction could be performed on two numbers. This function should add a “-” operator in the text field, indicating a subtraction operation between two numbers. The subtraction operation should be performed and added to the set of operations whenever the subtraction button is pressed.

8. **Create multiplication function.** This function should make it possible to multiply numbers through the multiplication button. This function should be activated when pressing the multiplication button and it should add a multiplication operator in the text field as well as add it to the order of operations.

9. **Create division function.** This function should make it possible to divide numbers through the division function. This function should be activated when the division button is pressed and it should add a division operation to the text field as well as to the order of operations.

10. **Create backspace function.** The backspace function should make it possible to remove the last added number or operator.

11. **Create order of operations.** This function should manage the order of operations so that multiplication and division operations are performed before addition and subtraction according to arithmetic rules. This means that the calculation 1+2*3 should calculate the operation 2*3 first and then the addition operation is calculated between the result of 2*3 and 1.

12. **Create parenthesis function.** This function should make it possible to create parentheses to perform more complex operations. The parentheses should be handled properly which means that if any errors for the parentheses exists, such as an unbalanced amount of left and right parentheses, a warning message should occur before trying to get the result from the operations.

### 4.2.2 The development process and evaluation of the calculator

The calculator were developed according to, and in the order of the tasks presented above. After each task had been performed, its comments were validated. Most of the tasks were performed individually, but during the tasks 5 & 6 as well as for the tasks 11 & 12, these tasks were combined and developed at the same time.

After the first task, the GUI of the calculator had been developed which is presented in figure 13. The GUI remained the same during the whole development process.
Before each validation during each task, the validation status for all methods and classes were counted. During the first task, all methods and classes had the validation status “Not validated yet”. When validating classes and methods for all the other tasks, the validation status for the classes within the project were always one class with the “Validated” validation status and one class with the “Out of date” validation status. The validation status for the methods in the class with the “Out of date” validation status are presented in table 1. No properties were created during any of the tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Validated methods</th>
<th>Out of date methods</th>
<th>New methods</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>Only methods and classes with the validation status “Not validated yet” existed during the first task.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>22</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>11 &amp; 12</td>
<td>17</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Method status for all of the methods in the class with the “Out of date” validation status.

The comments produced were compared with its corresponding code after the validation had been performed after each task to see whether the comments produced were accurate or not.
The result after these comparisons were that the comments produced were correct and accurate, even if the quality for some of the comments produced were low. One of these examples are presented in the figure 14. The example shows a method named button20_Click that is calling another method AddParenthesisToNumbersAndOperations with the input parameter “)”. The called method will add a parenthesis to a list of strings named NumbersAndOperations, which in this case will add a right parenthesis to the list of strings. The problem with the comment in this case is that the comment refers to that a right parenthesis is added to the correct list while also stating that the parenthesis is added to the list by “calling that method” while not specifying what method is called. Even if this comment could be considered ambiguous it still isn’t outdated and inaccurate because it explains the correct intention for the method and summarizes the method correctly. Since a method is in fact called, the expression “that method” is most probably intended to refer to the method AddParenthesisToNumbersAndOperations which therefore shows the accuracy of the comment.

```
/// <summary>
/// Adds a right parenthesis to the NumbersAndOperations List by calling that
/// method
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button20_Click(object sender, EventArgs e)
{
    AddParenthesisToNumbersAndOperations("");
}
```

Figure 14: The code and the code comment for the method button20_Click.

4.2.3 Conclusions for the evaluations

The functional testing of the comment validator showed that most of the functionality worked, and that only one of the requirements, requirement 4.1., didn’t pass its tests. Even if this requirement weren’t fulfilled, its effects for a normal developer wouldn’t interfere with the development process, therefore the conclusion that the program works for normal situations would be correct.

The field study evaluation showed that the program did in fact solve the problems that it were supposed to solve, namely to remove the outdated and incorrect comments when developing with the comment validator. The field study evaluation also showed that even if the comment validator did show that outdated comments could be removed when developing with it, the comment validator doesn’t guarantee that the comments produced for a SDP is of high quality.
5 Discussion

In this chapter a discussion of this thesis is presented where improvements for code comments is described, a comparison between the comment validator and other similar programs, a discussion about the evaluation performed and a lastly some future improvements for the comment validator is presented.

5.1 Improvements for code comments

During the development of this thesis it was recognized that although code comments could contain a wide array of different forms of information, such as commented out code and summarizations of code segments, the notation for the comments are almost always the same which is either line comments such as // and block comments such as /* */, regardless of what type of information the comment consist of. This could be improved in the future by adding other forms of notations for comments to be able to in a better way convey what type of information that a comment consist of, such as a //# notation for commented out code. These forms of notations could also be shown in a different color in IDEs to be easier to spot and make it easier to separate different types of comments. Using different forms of notations for different types of comments could also be useful in the long run to differ between automatically generated comments and user generated comments, if automated comments and user generated comments would have different notations.

5.2 Comparison between the comment validator, automatic documentation generators and other programs with the same purpose

In chapter 1 and 2 other programs are mentioned that were also created in order to reduce the amount of outdated and inconsistent code comments found within SDPs, such as the docfacto Adam program within the docfacto toolkit (Docfacto, 2014), @tComment program (Tan et al., 2012) and the JavadocMiner (Khamis, Rilling & Witte, 2013). Even if these program have the same purpose as the comment validator, i.e. they are also created in order to remove outdated and inconsistent code comments, their functionality is different. While the comment validator relies on recurring manual validations of code comments during the whole development process the programs docfacto Adam, @tComment and the JavadocMiner are automatic solutions that finds outdated and inconsistent comments by analyzing and comparing the properties of the comments to the comments corresponding code. Even if these programs might be less time consuming to use compared to the comment validator, they are unable to find all possible outdated and inconsistent comments found within a SDP compared to the use of the comment validator which is able to remove all outdated and inconsistent code comments, as described in chapter 2 & 3. Therefore, in order to remove outdated and inconsistent code comments completely, a solution based upon the same principle as the comment validator is necessary. This doesn’t mean that the other programs mentioned couldn’t be used together with the comment validator in order to more efficiently find outdated comments and indicate where outdated and inconsistent code comments could exist, although some form of manual code comment validations are necessary to perform in order to validate the correctness of comments that describe some types of information such as units of measurements for variables.

5.3 Discussion about the evaluation

Different results from a field study evaluation for the comment validator could probably be achieved in a more professional working environment where the developers are more experienced at writing both code and comments and where the performed tasks are of higher
complexity that the tasks performed during the evaluation in this study. The evaluation methods used in this study are limited in its application to evaluate how the application would work in practice, the field study is small and even if they prove some worth, it might work differently in an another environment.

The function testing performed during this study may have missed certain situations where problems could occur when working with the comment validator, such as how different lines of code are processed, which could result in problems in an another environment. This doesn’t affect the general principle behind the comment validator and the principle of comment validations, but it could result in unexpected problems if the program were applied in another environment.

Similar results that were gathered during the field study might be able to achieve without the use of the comment validator or any other, similar program based upon manual validations. Similar results might be possible to achieve by using recurring code inspections of the code comments in order to guarantee the correctness of code comments. Although similar results might be possible to achieve without the use of the comment validator or any other similar program, the comment validator would probably be more efficient at finding potential outdated and inconsistent code comments compared to, for example, code inspections. The comment validator is probably more efficient because it notices differences in the source code and could therefore notice changes in the source code that developers might forget that they have performed. This also gives developers the possibility to only inspect comments for code that have changed rather than having to review all the comments in a SDP, therefore reducing the amount of inspections that are necessary to perform.

5.4 Future improvements of the comment validator
The program presented in this thesis could be improved in several ways because right now it have limitations in its functionality. Possible changes in the future could be to recognize whether code comments exists or not for methods, properties and classes. This improvement could also provide the possibility for methods, properties and classes without any code comments to get the validation status “Validated” automatically in order to reduce the amount of necessary validations, because there are no comments for these code segments to validate.

Another future addition could be to add the possibility to require that classes, methods and properties should contain certain comments, such as a doc comment for each class, method and property and that these comments should meet a set of requirements, such as a minimum number of words within each comment. Including this type of functionality in the comment validator would provide the possibility to enforce the use of code comments and enforce that the code comments meets a certain quality standard.

One possible application of this program or any similar program could be to integrate it with revision control systems used by developers to share code for SDPs. If the comment validator were combined with a revision control system it could enforce developers to validate their comments before code is submitted to a revision control system. This solution could result in that some developers might not inspect their comments before validating them in order to ensure that the comments are accurate. Another possible scenario is that this solution would encourage developers to inspect their comments one extra time before submitting code to a shared SDP in order to improve the accuracy of the comments within the SDP.
with the recommendation from project managers that developers should inspect their comments before submitting their code, then this solution could be even more effective.

In the current version of the comment validator, some specific code segments aren’t handled properly, such as events. In a future version of the comment validator events could be handled and managed in a similar manner as methods and properties to improve the usefulness of the comment validator.

The project right now is focused on searching through code by reading the lines of code, line by line and recreating the lines of the code, a process that right now based on the physical lines of code in the project rather than the logical lines of code. The use of physical lines of code for understanding SDPs requires that programs are structured in a very specific way in order to be properly handled by the comment validator, which could result in that long lines of logical code can’t span over several physical lines of code because the comment validator wouldn’t then be able to process these SDPs correctly. In a future version of the comment validator this functionality could be changed so that logical lines of code are used instead of physical lines of code in order to reduce the need for specific structures for classes and methods.

Errors within the software occurs now with parentheses and in some cases whenever two non-letter, non-number characters are located between each other without having any direct relationship, such as “>>” compared to “=>”. In a future version of the comment validator, all these types of errors could be fixed to improve the functionality of the comment validator.

A problem that were recognized during this study were that during the comment validations with the comment validator a lot of time were spent switching back and forth between the comment validator and the IDE that were used, namely visual studio. Another issue that were identified were that a lot of time were also spent on finding the selected code for the particular validation. In the future, the comment validator could be integrated directly into the IDE and have the ability to switch between methods, properties and classes by selecting code segments from a list in the IDE to go to and view a selected code segment. This alteration of the comment validator could therefore reduce the time spent switching between the comment validator and the IDE as well as the time spent for trying to find specific code segments.

5.5. The comment validator and software development methods
This study have focused on describing how the comment validator can be used by developers to remove outdated and inconsistent code comments. Although the comment validator probably could be used in other ways than this study proposes as well as for other purposes than this study describes, the utility of using the comment validator could be lost if it is used in differently. The utility of using the comment validator could also be lost if the comment validator would be misused in a workplace, such as if no rules and regulations are considered for how the comment validator should be used.

The comment validator would therefore need to be fitted into the software development methods used at workplaces in order to use the comment validator properly and to enforce proper use of the comment validator. If the use of the comment validator were incorporated into software development methods at workplaces, then project managers could enforce developers to inspect and validate their comments at recurring intervals and encourage developers to spend time for this procedure.
In chapter 4.2 three different strategies are described for how the comment validator could be used, and these strategies could be incorporated within software development methods to promote how and when the comment validator should be used during the software development process. If the comment validator would be incorporated into software development methods, and the use of the comment validator were promoted by project managers, the comment validator could be properly used by developers so that developers inspect and validate their comments at recurring intervals.

The comment validator could easily be combined with some of the agile software development methods that are common today when working with software development. The most popular of these agile software development methods is the method scrum (VersionOne, 2014). Scrum consist of four main components: The scrum team, the scrum events, scrum artifacts and the rules for scrum. When developing software with scrum, the scrum team performs a set of scrum events with the support of the scrum artifacts. A main principle behind scrum is that software is produced as a set of tasks that are performed during short intervals named sprints (Schwaber & Sutherland, 2012). The comment validator could be incorporated by project managers who uses scrum by enforcing developers to validate and inspect their comments after each task they perform. Project managers could also enforce developers to validate comments at the end of the sprints as a specific event in order to promote the maintenance of code comments. These types of validation strategies for how the comment validator could be combined with scrum is based on the validation strategies presented in chapter 4.2.

Another agile software development method that have been instrumental for popularizing the agile software development methods is the method extreme programming (XP). XP is an agile software development method that consists of a set of practices that ranges from how the software development process should be planned to how the workplace for the software development team should be structured. An important aspect behind XP is that functionality for the software is described as, and broken down into a set of so called user stories. The user stories describe a feature for the system that is being developed (Beck & Andres, 2005). The comment validator could be integrated into the XP development process by allowing developers to inspect and validate their comments after each user story have been developed and integrated into the upcoming system.

The comment validator could also be combined with other software development practices, such as the software development practice test-first programming (TFP), more commonly known as test-driven development (TDD). TFP is a software development practice that focuses on writing unit tests for the code for proposed features before the code for the feature is written. After the tests have been written, the tests are run to test that they fail. When a feature have been developed for the tests, the tests are run again to test whether the code for the feature did behave as it was supposed to do (Beck & Andres, 2005). The comment validator could be combined with this practice by allowing developers to inspect and validate comments whenever tests are run, thereby combining the TFP practice with comment validations. The programs available for running unit tests could in the future be combined with the comment validator so that whenever unit tests are run in order test whether the code for the SDP works correctly, the comment validator is run in order to find code comments that could be outdated and inconsistent to their corresponding code.
6 Conclusion and further research
In this chapter the conclusion of this study is presented and further research that could be performed in the same area of this study.

6.1 Conclusion
Outdated and inconsistent code comments could introduce bugs and mislead developers. In order to reduce the amount of outdated and inconsistent code comments within software development projects, the aim of this study was to show how an instantiation could be produced that helps developers to reduce the amount of outdated and inconsistent code comments within software development projects. To reach the aim of this study, an instantiation were developed named the comment validator.

The comment validator helps developers to reduce the amount of outdated and inconsistent code comments within software development projects through the use of recurring manual validations of code comments performed by software developers. The comment validator segments code into three segments: classes, methods and properties. The segments identified by the comment validator are then manually validated by developers in order to assure that the code segments corresponding code comments are accurate. The comment validator have the ability to recognize when the code changes for validated code segments in order to identify all situations where code comments could be outdated and inconsistent.

The comment validator is able identify all code changes that affects the behavior of a code segment, which gives the comment validator the ability to recognize all situations where code comments could be outdated and inconsistent to its corresponding code. This functionality does not tell whether the code comments for code segments are outdated and inconsistent but rather indicate that the code comments for code segments could be outdated and inconsistent. This functionality gives the comment validator the ability to identify more situations where the code comments for code segments could be outdated and inconsistent compared to other programs produced with the same purpose, such as the programs @tComment (Tan et al., 2012), the JavadocMiner (Khamis, Rilling & Witte, 2013) and docfacto Adam (Docfacto, 2014) because these programs are only able to find some situations where code comments are and could be outdated and inconsistent.

The comment validator were evaluated both through functional testing and through a field study in order to test whether the program were able to reach the aim of this study. The functional testing showed that the comment validator worked well in most situations and does identify situations where code comments could be outdated and inconsistent. Although a few minor defects were detected for the comment validator during the functional testing, these defects wouldn’t interfere with the development in most situations. The field study showed that the comment validator works according to its proposed functionality in an actual software development scenario by removing outdated and inconsistent code comments. Because the result of the field study showed that the comment validator were able to remove all outdated and inconsistent comments in a SDP, the aim of the study were reached.

Although the comment validator is a fully functional program, the comment validator could be improved in a set of various ways in order to produce an even more efficient and effective program for manual validations. One of these possible improvements for the comment validator
could be to integrate the comment validator with revision control systems in order to enforce that developers validate their comments before submitting them to a SDP.

Even if the comment validator is promising for its capability to identify all possible situation where code comments could be outdated, the comment validator still relies on how developers use it. If developers doesn’t use the comment validator according to the description in this thesis, such as validating comments without inspecting the correctness of those comments, the positive effects of using the program will be missed and outdated and inconsistent comments could still remain in a SDP.

6.2 Further research

Further research could be performed in order to evaluate the comment validator, or any other similar program based upon manual code comment validations, through longitudinal studies for larger projects. These studies could be performed in order to evaluate the long term effectiveness of manual validations of code comments as well as the potential drawbacks of performing manual comment validations in SDPs.

Further research in the area of code comment updating practices could be to try to combine manual code comment validations with automatic code comment validations. These different forms of code comment validations could be combined in order to try to reduce the amount of manual validations of code comments and to provide guidance for developers to improve their comments. Further research could test if and how these different forms of code comment validations could be combined in order to make code comment validations more effective and efficient.

Further research could also use the comment validator or any other similar program to try to better understand how outdated and inconsistent code comments affect software development projects. Because the use of the comment validator within SDPs provides the capability for developers and researchers to find and reduce all potential outdated and inconsistent code comments, these studies could use these capabilities of the comment validator to find all possible outdated and inconsistent comments to test how they affect other parts of the SDP, such as the introduction of bugs.
References


Nurvitadhi, E., Leung, W.W. & Cook, C. (2003). Do class comments aid Java program understanding?. *33rd Annual Frontiers in Education* (FIE ’03) (pp. T3C13-T3C17). Boulder, USA 5-8 Nov. Available at:


Siy, H. & Votta, L. (2001). Does the modern code inspection have value?. Proceedings of the IEEE International Conference on Software Maintenance (ICSM ’01) (pp. 281-289). Florence, Italy 7-9 Nov. Available at: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=972741&newsearch=true&queryText=%20Does%20the%20modern%20code%20inspection%20have%20value [2015-08-26]


Appendix 1:
LoginForm tests and outcomes:

<table>
<thead>
<tr>
<th>Req. Nr.</th>
<th>Test</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Studying the graphical user interface (GUI) of the LoginForm.</td>
<td>The login form contained the correct content</td>
</tr>
<tr>
<td>2</td>
<td>A login attempt were performed with no names added to the text fields before trying to log in.</td>
<td>Login attempt failed</td>
</tr>
<tr>
<td></td>
<td>A login attempt were performed with only a first name added in the first text field before trying to log in.</td>
<td>Login attempt failed</td>
</tr>
<tr>
<td></td>
<td>A login attempt were performed with only a last name added in the second text field before trying to log in.</td>
<td>Login attempt failed</td>
</tr>
<tr>
<td></td>
<td>A login attempt were performed with a first name added in the first text field and a last name added in the second text field before trying to log in.</td>
<td>Login attempt succeeded</td>
</tr>
<tr>
<td>3</td>
<td>A login attempt were performed with the checkbox checked and after the login attempt had been performed, the program were restarted to observe if the login name were stored.</td>
<td>The same name used in the login attempt were made visible after the restart of the program.</td>
</tr>
<tr>
<td></td>
<td>A login attempt were performed when the checkbox where unchecked. The program were restarted after the login attempt to see if the name were stored.</td>
<td>No names stored in the text fields when the program were restarted.</td>
</tr>
</tbody>
</table>

Table 2: Tests performed for the LoginForm window with their outcomes presented for each requirement.

ProjectsForm tests and outcomes:

<table>
<thead>
<tr>
<th>Req. Nr.</th>
<th>Test</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The projects form were opened and its GUI analyzed.</td>
<td>The GUI contained the correct content</td>
</tr>
<tr>
<td>1.1</td>
<td>The add project button were pressed to see if a new form would occur.</td>
<td>A new window occurred named add projects form containing a text field and a button.</td>
</tr>
<tr>
<td>1.2</td>
<td>The button in the add projects window were pressed while no name were added in the text field.</td>
<td>A warning message occurred when the button were pressed stating that a name for the project needed to be added.</td>
</tr>
<tr>
<td></td>
<td>A name were added before the button where pressed in the add projects window.</td>
<td>The form closed and the process of adding a project proceeded.</td>
</tr>
<tr>
<td>1.3</td>
<td>No folder were selected (i.e. the desktop folder) before the OK button were pressed in the select folder window.</td>
<td>A warning message occurred stating that no project could be added because the</td>
</tr>
</tbody>
</table>
The process of adding a project is cancelled by pressing the cancel button in the select folder window. The process were cancelled by pressing the red button in the top right corner in the select folder window. A folder were selected that were not the desktop folder and then the OK button is pressed in the select folder window. The process proceeded and completed the process of adding a project.

1.4 A project were added to the comment validator and the project table were analyzed to see whether the new project had been added there or not. The comment validator were then restarted to see if the result had been stored or not. The project had been added to the projects table, its name were shown in the Projects column and the location of the projects folder were shown in the location column. The added project were still visible in the projects column after the program restarted.

2 A project were removed from the projects table. The program were then restarted. The project were removed from the table and after the program were restarted it was still removed.

3 The log out button were pressed. The projects form were closed and the login form were made accessible.

4 A project were added where the selected folder for the project was the starting folder for a C# project. The project were opened correctly. A project were added and opened where the chosen folder were a folder containing multiple C# projects. A warning message stating that an error occurred.

The open project button were pressed without any project selected. A warning message stating that an error occurred.

<p>| Table 3: Tests performed for the ProjectsForm window with their outcomes presented for each requirement. |
|---|---|---|
| Req. Nr. | Test | Outcome |
| 1 | A project were added to the comment validator and opened for the first time. The folder of the project were inspected manually to see if the CommentValidations folder had been created. The CommentValidations folder were inspected to see if it contained XML-files for all the correct classes in the project. | Warning messages occurred stating that some files couldn’t be added to the project. The CommentValidations folder had been created at the correct place. The CommentValidations folder did contain the correct XML-files for the classes from the selected project. |
| 2 | The GUI were studied when a project had been opened to test if it contained all of the components from the requirement. | The GUI contained all of the buttons required as well as the two tables. |
| 2.1 | A project were opened to see if the class table contained all of the classes within the project and if table contained the correct columns. | The class table contained all of the proposed columns as well as all of the classes within the project with their names mentioned in the class column. |
| 2.2 | A project were opened for the first time to test if all classes had the validation status “Not validated yet”. | All classes in the class table had the validation status “Not validated yet” and the cells for their validation status had a grey background color. |
| 2.3 | A previously added and opened project were modified to have a set of new classes added. The validation status for the classes were then inspected in the ValidationForm window. | The new classes in the class table had the validation status “New class” and their validation status cell had a yellow background color. |
| 2.4 | A project with classes that contained the validation statuses the “New class” and the “Not validated yet” were opened to see if the “last validated” and “validated by” columns were empty for all classes. | The columns “last validated” and “validated by” in the class table were empty for all classes. |
| 2.5 | The methods table within the validation form were analyzed after a project had been opened to see if it contained all the correct columns, a line of text above the table describing that no methods could be viewed for any class and that it contained no methods. | The methods table contained all of the columns described in the requirement, the methods table were empty when the program were started and there were a line of text above the methods table describing that no methods for any class were viewed right now. |
| 3 | The button find methods and properties were pressed without any class selected. | A warning message occurred stating that no methods could be viewed. |
| 3.1 | The find methods and properties button were pressed for a selected class without any methods or properties. | A warning message occurred stating that the class didn’t contain any methods. |
| 3 | The button find methods and properties button were pressed for a selected class with both methods and properties. | The names of the methods and their parameters were shown in the methods column and in the method type were shown in the method type column. |
| 3.1 | A number of classes had their methods and properties shown through the find methods and properties button to test if the line of text above the methods table changed. | The line of text describing the current class changed for each class to properly describe the current class. |
| 3.2 | A previously added project without any validated methods were opened and had the methods and properties shown for one of its classes to test if the validation status for the methods and properties were “Not validated yet”. The selected class were then modified and new methods were added to the class to see if the new methods had the validation status “New method”. | The validation status for all of the methods and properties in the selected class were “Not validated yet” and their cells had a grey background. The new methods had the validation status “New method” and their cells had a yellow background. |
| 3.3 | The columns “Validated by” and “Last validated” in the methods table were studied for a class without validated methods and properties to see if they were empty. | The columns “Validated by” and “Last validated” were empty for the methods and properties. |</p>
<table>
<thead>
<tr>
<th>4</th>
<th>The validate method button were pressed without having selected any method.</th>
<th>A warning message occurred stating that a method couldn’t be validated.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The validate method button were pressed for a selected method were the validation were supposed to work.</td>
<td>The validation status changed of the method, the validators name were added to the validated by column and the date of the validation were added to the last validated column, all on the same row where the selected method were situated.</td>
</tr>
<tr>
<td>4.1</td>
<td>New empty lines were added within the method to test how it affected the validation status of the method.</td>
<td>This test showed that new empty lines did not affect the validation status.</td>
</tr>
<tr>
<td></td>
<td>Whitespaces were added to lines of code outside parentheses and brackets. Whitespaces were added both before the code, within the code and after the code to test how it affected the validation status for a method.</td>
<td>Whitespace changes did not change the validation status of the method.</td>
</tr>
<tr>
<td></td>
<td>Line comments and block comments were added before lines of code, at the end of lines of code and within lines of code. Old comments were removed from methods. After this the validation status of the method were observed.</td>
<td>The results of the third test did show that both adding and removing comments didn’t affect the validation status of a method or property.</td>
</tr>
<tr>
<td></td>
<td>Whitespaces were added within parentheses, braces, brackets and chevrons, and the changes for the validation status were observed after each whitespace had been added.</td>
<td>Whitespace changes made within parentheses and brackets did change the validation status of the method, whitespace changes occurring within braces or chevrons did not change the validation status, except for whitespace changes made between two chevrons located between each other such as “&gt;&gt;” going to “&gt; &gt;”.</td>
</tr>
<tr>
<td></td>
<td>Character changes were made outside parentheses, brackets, braces and chevrons. The validation status were controlled after each change. The changes were reverted after each change had been controlled.</td>
<td>The result of this test showed that for all of these situations, the validation status did change from “Validated” to “Out of date”.</td>
</tr>
<tr>
<td>5.1</td>
<td>The validate class button were tried on five classes, one that had all of its methods validated, one that contained validated methods and out of date methods, one that contained validated and new methods, one that contained validated and not validated yet methods and one class that only contained “Not validated yet” and “New method” methods.</td>
<td>For the class who have had all of its methods validated, the validation process proceeded without any problems. For all of the other classes, a warning message occurred stating that all of the current methods and/or properties in the current class needs to be validated before the class could be validated.</td>
</tr>
<tr>
<td>5.2</td>
<td>Two classes were validated that have had all of its methods and properties validated. The validation status for these classes were “Not validated yet” and “New class”.</td>
<td>The validation status for both classes were changed to “Validated” and the validation status cell got a green background.</td>
</tr>
<tr>
<td>Classes without any methods or properties were validated, one with the validation status “Not validated yet” and the other with the validation status “New class”.</td>
<td>Both classes got the validation status “Validated” and their validation status cell got a green background.</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Empty lines were added to a class, both within the methods of the class as well as outside the methods within the class. The validation status were then observed.</td>
<td>The validation status remained intact even after empty lines had been added to the class file.</td>
</tr>
<tr>
<td>Whitespaces were added to lines of code both within the methods and properties in a selected, validated class as well as in lines outside the methods and properties in the class. The validation status were then observed.</td>
<td>The validation status of the class remained intact even after these whitespace changes to the class.</td>
<td></td>
</tr>
<tr>
<td>Comments were added to a class, both XML-comments, line comments and block comments. These comments were located at various places within the class. The validation status were then observed.</td>
<td>The validation status of the class remained the same after these changes had been performed.</td>
<td></td>
</tr>
<tr>
<td>Regions were added to a class through the keywords #region and #endregion. The validation status were then observed.</td>
<td>Adding and removing regions to a class did not affect its validation status.</td>
<td></td>
</tr>
<tr>
<td>Lines of code were added both to methods within the class and outside the methods within the class. After each new line of code had been added, the validation status of the class were observed. This procedure were then repeated for each new added line of code.</td>
<td>For each new line of code that were added, either if it were inside a method or not, the validation status were changed from “validated” to “out of date”. This test passed.</td>
<td></td>
</tr>
<tr>
<td>Characters in lines of code both inside and outside methods and properties were changed.</td>
<td>The result of all these tests were that the validation status went from “Validated” to “Out of date” as expected.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A project were opened in the comment validator while the content within a set of classes with the validation status “Validated” were modified. The button update classes were pressed to observe any validation status changes.</td>
<td>The classes’ validation status were changed from “Validated” to “Out of date” when the update class button were pressed.</td>
</tr>
</tbody>
</table>

*Table 4: Tests performed for the ValidationForm window with their outcomes presented for each requirement.*
Appendix 2:

LoginForm requirements:

1. The login form should contain two text fields, one checkbox and a button. It should be possible to write in both of the text fields, check and uncheck the checkbox and press the button.

   **Test:** By studying the graphical user interface (GUI) of the login form it is possible to see that it contains two text fields, one checkbox and a button. It is also possible to write in the text fields, to check and uncheck the checkbox as well as press the button.

   **Requirement status:** Passed.

2. The login form should enforce that developers have logged in to the instantiation with a first name and a last name before the developer are able to start validating comments for various projects. If not both a first name and a last name are contained when the login attempt is attempted a warning message should occur stating that the login attempt failed.

   **Test:** To test this requirement, four scenarios where performed: In the first scenario no names were added to the text fields before trying to log in, in the second scenario only a first name were added in the first text field before trying to log in, in the third scenario only a last name were added in the second text field before trying to log in and in the fourth scenario a first name were added in the first text field and a last name were added in the second text field before trying to log in. In this test, the login attempt failed during the first three scenarios and resulted in a warning message stating that a login name needed to be added. During the fourth scenario, the login attempt succeeded.

   **Requirement status:** Passed.

3. It should be possible for a developer to store the login names used so that the same name occurs in the text fields the next time the program is opened and this functionality should be performed by checking the checkbox before a login attempt.

   **Test:** To test whether this requirement were met, two scenarios where performed: In the first scenario a login attempt were performed with the checkbox checked and after the login attempt had been performed, the program were restarted. In the second scenario a login attempt were performed when the checkbox where unchecked and after the login attempt had succeeded, the program were restarted. In the first scenario, the same name used in the login attempt and made visible after the restart of the program. In the second scenario, there were no names stored in the text fields when the program were restarted.

   **Requirement status:** Passed.

ProjectsForm requirements:

1. When opening the projects form, the developer should be able to see four buttons and a table. The buttons should be named open project, remove project, add project and log out, and the three first of these should be located in a row next to each other while the fourth button, the log out button should be available in the top right corner of the projects form. The table in the projects form should be located beneath the row buttons and should contain two columns named Project and Location.
Test: To test this, the projects form were opened and its GUI analyzed. The GUI contained the four named buttons at their correct places as well as a table with the two correct columns.

Requirement status: Passed.

1.1. When adding a project, the developer first needs to push the add project button. After pushing the add project button a new form should occur containing a text field and a button.

Test: To test this requirement, the add project button were pressed to see if a new form would occur. The test showed that a new form occurred named add projects form and that form contained a text field and a button.

Requirement status: Passed.

1.2. In the new form, a name for the project should be written in the text field in the form and if no name is written for the project the developer should be unable to continue the process of adding a new project when pressing the button on the form and a warning message should occur.

Test: To test this requirement, two scenarios were performed, one where no name were added in the text field before the button where pressed and another scenario where a name were added before the button where pressed. In the first scenario a warning message occurred when the button were pressed stating that a name for the project needed to be added. When the button were pressed in the second scenario the form closed and the process of adding a project proceeded.

Requirement status: Passed.

1.3. After the developer have written a name for the project and gone through to the next step, the developer needs to select a folder from a list of folders for the location where the added project exists and then press the OK button to complete the process of adding a project. If no folder is chosen, or the default folder is chosen, which in this case should be the desktop folder, this should result in a warning message indicating that no project could be added because the location of the project were invalid.

Test: To test this requirement, four scenarios were constructed. In the first scenario, no folder is selected (i.e. the desktop folder) before the OK button is pressed to complete the process of adding a project. In the second scenario the process of adding a project is cancelled by pressing the cancel button. In the third scenario, the process is cancelled by pressing the red button in the top right corner of the choose map form. In the fourth scenario a folder is selected that is not the desktop folder and then the OK button is pressed. In all of the first three scenarios a warning message occurred stating that no project could be added because the location were invalid and then the process of adding a project were stopped. In the fourth scenario the process proceeded and completed the process of adding a project.

Requirement status: Passed.

1.4. When a project has been added to the comment validator, the project should be stored internally within the program and the stored project should be made visible in the stored projects table in the projects form. The name of the project should then be shown in the projects column and the location of the project should be shown in the folder location...
column. Because the project should be stored, it should be accessible and made visible in the projects table even after the program is restarted.

**Test:** To test this requirement a project were added to the comment validator according to the procedure presented in the requirements 2.1-2.3. After the project had been added, the project table were analyzed to see whether the new project had been added there or not. The analysis of the projects table showed that the new project had been added to the projects table and that its name were shown in the Projects column and that the location of the projects folder were shown in the location column. The comment validator were then restarted to see if the result had been stored or not. After the comment validator had been restarted, the added project were still visible in the projects column.

**Requirement status:** Passed.

2. Projects should be possible to remove when pushing the remove project button. By doing this the project should be removed from the project list in the projects form as well as from its stored location, therefore it should be removed and unavailable even after the program has been restarted.

**Test:** To test this requirement a scenario were performed were an added project were removed from the projects list by selecting a project and then pressing the remove project button. The program were then restarted to see if the project had been removed from its stored location. The results of this test showed that the project were removed from the table after the remove project button had been pressed and after the program were restarted it was still removed.

**Requirement status:** Passed.

3. It should be possible for a developer to log out from the projects form, by doing this the developer should be returned to the login form.

**Test:** To test this requirement, a test were performed were a successful login attempt were performed and then the log out button were pressed to log out which resulted in that the projects form where closed and the login form were made accessible.

**Requirement status:** Passed.

4. It should be possible for a developer to open a project by selecting a project from the projects list and then pressing the open project button. If no project is chosen or any other error occurs while trying to open a project, a warning message should occur. By pressing the open project button for a selected and valid project should result in that the validation form should open with information about the selected project, for more information about the validation form requirements, see the section below about the validation form requirements.

**Test:** To test this requirement, a three different scenarios were performed. In the first scenario a project were added where the selected folder for the project was the starting folder for a C# project. This scenario led to that the project were opened correctly. In the second scenario an incorrect project were added where the chosen folder were a folder containing multiple C# projects. This scenario led to a warning message stating that an error occurred. In the third scenario the button were pressed without any project selected. This scenario led to a warning message stating that an error occurred.

**Requirement status:** Passed.
ValidationForm requirements

1. When a project is opened for the first time, the program should check for whether the project contains a folder named CommentValidations. The CommentValidations folder should exist in the same folder as the Program.cs class found within the projects folder. If the CommentValidations folder does not exist, the program should create this folder in the same folder as the Program.cs class file, and within the folder XML-files should be created for each user generated class file and the Program.cs file within the project, class files such as the AssemblyInfo.cs and files ending with Designer.cs and DatabaseEngine.cs shouldn’t be created as XML-files. If an XML-file for a class file can’t be created, a warning message should occur with information about for which class that an XML-file couldn’t be created.

Test: To test this requirement, a project were added to the program through the projects form and then the project were opened through the open project button. The project were opened for the first time so that no CommentValidations folder did exist within the project folder. When opening the project, several warning messages folder exist within the project folder. When opening the project, several warning messages occurred stating which files that couldn’t be added to the project, resulting in an individual warning message for each of these files. After the project had been opened, the folder of the project were inspected manually to see if the CommentValidations folder had been created and if it had been created at the correct position. The result of this inspection showed that the CommentValidations folder had been created at the correct place. Lastly, the CommentValidations folder were inspected to see if it contained the correct XML-files for each of the appropriate classes in the project. An inspection of the CommentValidations folder showed that it did contain the correct XML-files for the classes from the selected project.

Requirement status: Passed.

2. When a selected project has been opened through the projects form the validation form should be opened. This form should contain four buttons and two tables. The buttons should be named update classes, find methods and properties for selected class, validate method and validate class. The two tables should be marked with class and methods in their corresponding table, a name that should occur in their first column.

Test: To test this requirement, the GUI were studied when a project had been opened to test if it contained all of the components from the requirement. The test showed that the GUI contained all of the buttons required as well as the two tables.

Requirement status: Passed.

2.1. The class table should also contain columns named Class, validation status, last validated as well as validated by. The class column should contain the classes’ name. The class table should show the classes contained within the selected project.

Test: To test this requirement, a project were opened to see if the class table contained all of the classes found within the project and also whether the table contained the correct columns. The test showed that the class table contained all of the proposed columns as well as all of the classes within the project with their names mentioned in the class column.

Requirement status: Passed.

2.2. For all the classes that is existing when a project is added for the first time in the comment validator their validation status shown in the validation status column should
be “Not validated yet”. The validation status cell for each class with this validation status should be marked with a grey background color.

**Test:** To test this requirement, a project were opened for the first time. For the classes shown in the class table in the validation form, all of them had the validation status “Not validated yet” and the cells for their validation status had a grey background color.

**Requirement status:** Passed.

2.3. For all classes that are created within a project after the project has been added and opened in the comment validator their validation status should be “new class” until they have been validated. Their validation status cell should be marked with a yellow background color.

**Test:** To test this requirement, a previously added and opened project were modified to have a set of new classes added. When the project were opened again in the comment validator the class table in the validation form showed the new classes with the validation status “New class” and their validation status cell had a yellow background color.

**Requirement status:** Passed.

2.4. For classes with the status “New Class” and “Not validated yet”, no information should be shown in the columns “last validated” and “validated by” in the class table.

**Test:** To test this requirement, the same project that were used and modified in requirement 2.5 were opened in the comment validator because this project contained classes with both the “New class” and the “Not validated yet” validation status. When this project were opened in the validation form both of the columns “last validated” and “validated by” in the class table were empty for all of the classes within the table.

**Requirement status:** Passed.

2.5. The second table, the methods table should contain five different columns named methods, validation status, validated by, last validated and method type. Initially when the validation form is opened, the methods table should be empty and above it a line of text should exist containing information describing that no methods for any class is viewed right now.

**Test:** To test this requirement, the GUI of the validation form were analyzed after a project had been opened. The analysis of the GUI showed that the methods table did contain all of the columns described in the requirement, the methods table were also empty when the program were started and there were a line of text above the methods table describing that no methods for any class were viewed right now.

**Requirement status:** Passed.

3. The find methods and properties button should show the methods and properties for a selected class in the methods table. The methods in the methods table should have their names and their parameters presented in the Methods column (i.e. it should be presented as Test1(int Int1, int Int2)) and whether a method is in fact a method or a property should be presented in the method type column. If no methods or properties are found or any other
error would occur while trying to find the methods and properties for a class a warning message should occur.

**Test:** To test this requirement, a three scenarios were tested. In the first scenario the button were pressed without having any class selected. This led to a warning message stating that no methods could be viewed. In the second scenario a class without any methods or properties were selected. This scenario led to a warning message stating that the class didn’t contain any methods. In the third scenario a class with both methods and properties were selected. This led to a scenario where all of the names of the methods and their parameters were shown in the methods column in the methods table and in the method type column it was shown whether the method were a method or a property.

**Requirement status:** Passed.

3.1. The line of text above the methods table should contain information about which class the current methods in the methods table belong to, even if the class doesn’t contain any methods or properties.

**Test:** To test this requirement, a number of classes had their methods and properties shown through the find methods and properties button. For each time the button were pressed for a valid and selected class the line of text were changed so that it did describe the proper class for which the methods should contain.

**Requirement status:** Passed.

3.2. For each method or property that existed within the project when the project were opened for the first time should have the validation status “Not validated yet” and the validation status cell for those methods or properties should have a grey background. For each new method or property that is added to the project after the project has been opened for the first time their validation status should be “New method” and their validation status cell should have a yellow background.

**Test:** To test this requirement, a project who had previously been added and opened in the comment validator, but who had not have had any methods validated yet, were opened and one of its classes had its methods and properties shown. The validation status for all of the methods and properties in the selected class were “Not validated yet” and their cells had a grey background. The selected class were then modified and new methods were added to the class. When the project were opened again in the comment validator and had the methods and properties shown for the class who had previously been modified the new methods were shown in the methods table and their validation status were “New method” and their cells had a yellow background.

**Requirement status:** Passed.

3.3. For all methods and properties that have the validation status “Not validated yet” or “New method”, their cells in the columns “Validated by” and “Last validated” should be empty.

**Test:** To test this requirement, the same project used in the test for requirement 3.2 were opened and the methods and properties were shown for the same class that were modified
in 3.2. For all of the methods and properties in this class their cells in the columns “Validated by” and “Last validated” were empty.

**Requirement status:** Passed.

4. The button validate method should, when activated, validate a selected method from the methods table by changing its validation status and store the method and its internal structure. By validating a method through the validate method button should change the status of the method, therefore the status in the validation status column in the methods table should be changed to validated and get a green background. After a method has been stored, the name of the person who were logged in and validated that method should be shown in the validated by column and the date of when the method were validated should be shown in the last validated column. If a method can’t be validated, or any other error occurs when trying to validate a method, a warning message should occur.

**Test:** To test this requirement, a two tests were performed. In the first test, the validate method button were pressed without having selected any method. This test resulted in a warning message stating that a method couldn’t be validated. In the second test, the validate method button were pressed for a selected method were the validation were supposed to work. This second test did change the validation status of the method to validated, added the validators name to the validated by column and added the date of when the validation had been done to the last validated column and all of these changes were made on the same row where the selected method were situated.

**Requirement status:** Passed.

4.1. If a previously validated method gets modified without becoming re-validated, its validation status in the methods table in the validation form should change from validated to “out of date” and get a red background. Modifications that should change the validation status of the method are all character changes made to the method, both by replacing the existing characters as well as adding new characters and removing characters. A few exceptions exist that should not alter the validation status. Modifications that should not alter the validation status are adding new empty lines to the method, adding new whitespaces occurring outside parentheses and brackets (as long as each individual word could be separated through whitespaces) and when comments in any form is added or removed.

**Test:** To test this requirement, five tests were performed to see which changes made within a method that could change the status of a method from validated to out of date.

In the first test, new empty lines were added within the method. This test showed that new empty lines did not affect the validation status.

In the second test whitespaces were added to lines of code outside parentheses and brackets. These whitespaces were added at several different places within the lines of code, both before the code, within the code and after the code. This test showed that whitespace changes did not change the validation status of the method.

The third test considered the adding and removal of comments. In this test comments were both added before lines of code, at the end of lines of code and within lines of code. In this test both line comments and block comments were used and comments were not only added,
but previously stored comments were removed from methods to test that comments shouldn’t affect the validation status. The results of the third test did show that both adding and removing comments didn’t affect the validation status of a method or property.

In the fourth test whitespaces were added within parentheses, braces, brackets and chevrons, and the changes for the validation status were observed after each whitespace had been added. This test showed that whitespace changes made within parentheses and brackets did change the validation status of the method, while whitespace changes occurring within braces or chevrons did generally not change the validation status, with the exception of whitespace changes made between two chevrons located between each other such as “>>” going to “> >”.

In the fifth test character changes were made outside parentheses, braces, brackets and chevrons and these characters changes did not consist of whitespaces, empty lines or the adding or removal of comments. In this test single character changes were made, such as changing lower-case characters to upper-case characters, adding and removing extra characters to variable names, adding whole new lines of code, removing of lines of code and adding and removing of other special characters such as changing “=” to “=>” and the opposite. After each character change were made, the validation status were controlled to see whether it had changed or not. After the validation status had been controlled, the changes were reverted to ensure that the validation status before a new change would be tested always were “Validated”. The result of this test showed that for all of these situations, the validation status did change from “Validated” to “Out of date”.

**Requirement status:** Not passed. Even if most of these tests did work under the conditions it should have performed, this requirement did not pass because of a minor incident with whitespace changes occurring between chevrons. Although this problem shouldn’t be a problem in most situations, it is a situation that needs to be considered and could be fixed and updated in future versions.

5. When pressing the validate class button, a selected class should be tried to be validated. To validate a class is composed of a set of different sub-requirements that needs to be fulfilled in order for the class to be validated.

5.1. For a class to be able to be validated, the class needs to have had all of its methods and/or properties validated before and if the stored class does contain methods and/or properties with a different validation status that “Validated”, a warning message should occur.

**Test:** To test this requirement, the validate class button were tried on five different classes, one that had all of its methods validated, one that contained validated methods and out of date methods, one that contained validated and new methods, one that contained validated and not validated yet methods and one class that only contained “Not validated yet” and “New method” methods. For the class who have had all of its methods validated, the validation process proceeded without any problems. For all of the other classes, a warning message occurred stating that all of the current methods and/or properties in the current class needs to be validated before the class could be validated.

**Requirement status:** Passed.
5.2. When the validation process proceeds without any warning messages occurring for a selected class, the validation status for the selected class should change to “Validated” and the validation status cell for the selected class should get a green background.

**Test:** To test this requirement, four test were performed. In the first two tests two classes were validated that have had all of its methods and properties validated. The validation status for these classes were “Not validated yet” for the class in the first test and “New class” for the class in the second test. The results of both these tests were that after the validate class button were pressed the validation status for both classes were changed to “Validated” and the validation status cell got a green background. In the third and the fourth tests classes without any methods or properties were validated, one with the validation status “Not validated yet” and the other with the validation status “New class”. The results of the third and fourth tests were that both classes got the validation status “Validated” and their validation status cell got a green background.

**Requirement status:** Passed.

5.3. When a previously validated class becomes modified, its status should be changed from “Validated” to “Out of date” and the validation status cell should receive a red background. These changes concern changes both within the methods and properties of a class as well as outside the methods and properties. Some changes shouldn’t affect the status of the class and these are extra whitespace changes on a line of text as long as each individual word or set of symbols are separated by whitespaces, adding or removal of empty lines, adding or removal of comments and separation the code into regions.

**Test:** To test this requirement, a set of changes were made during a set of tests to a validated class and after each individual change the validation status were observed.

In the first test empty lines were added to a class, both within the methods of the class as well as outside the methods within the class. This test showed that the validation status remained intact even after empty lines had been added to the class file.

In the second test extra whitespaces were added to lines of code both within the methods and properties in a selected, validated class as well as in lines outside the methods and properties in the selected, validated class. The validation status of the class remained the same even after these whitespace changes to the class.

In the third test comments were added to a class and these comments were extra whitespace changes on a line of text as long as each individual word or set of symbols are separated by whitespaces, adding or removal of empty lines, adding or removal of comments and separation the code into regions. The validation status of the class remained the same after these changes had been performed.

In the fourth test regions were added to a class through the use of the keyword #region to begin the region and #endregion to show when the region did end. This test showed that adding and removing regions to a class did not affect its validation status.

A test were performed to see whether big differences within classes were recognized, such as adding new lines of code to a class. To test this new lines of code were added to both
methods within the class and outside the methods within the class. After each new line of code had been added, the validation status of the class were observed and if the class had become out of date, the class were validated. This procedure were then repeated for each new added line of code. This test showed that for each new line of code that were added, either if it were inside a method or not, the validation status were changed for “validated” to “out of date”. This test passed.

A test were performed where individual characters in lines of code both inside and outside methods and properties in a selected class were changed in order to test whether character differences could be recognized. The character differences that was tested were: lower letter characters in variable names were changed to upper letter characters, additional characters were added to variable names to extend the variable names, sequences of non-letter, non-number characters were extended, such as > were extended to >= and characters were removed. The result of all these tests were that the validation status went from “Validated” to “Out of date” as expected.

**Requirement status:** Passed.

6. The update classes’ button should update the status of different classes and whether any differences have been made, and if it is impossible to update the status, a warning message should be given.

**Test:** To test this requirement a project were opened in the comment validator while the content within a set of classes with the validation status “Validated” were modified. The button update classes were then pressed to see whether these classes validation status had been changed. The results of this test were that the classes’ validation status were changed from “Validated” to “Out of date” when the update class button were pressed.

**Requirement status:** Passed.