ABSTRACT
A literature study on software maintenance and agile development methods was used to reason about the relationship between them. First, factors influencing software maintenance were identified. Second, characteristics of agile development methods were extracted from the literature. Then, the obtained information is used to reason about the extent to which agile development methods support long-term maintainability. It was concluded that, apart from frequently changing requirements, agile development methods do not have a negative impact on the maintainability of a system. In fact, program complexity and defects are reduced.

Keywords
agile software development, software maintenance, software maintainability, agile development methods

1. INTRODUCTION
Traditionally, the software development process has often been described as a sequential process. A well-known model of sequential software development is the waterfall model [36] shown in Figure 1 [40]. A phase can be started only if all preceding phases have been completed. The sequence of all phases represents the lifespan of a software system.

![Waterfall model](image)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

In contrast, agile methods aim at delivering working software in short iterations. Each iteration, which typically lasts no longer than a month, includes all phases known to software development. However, there is less emphasis on planning and documentation, but more on simplicity and communication between team members as well as with customers. Changing requirements between iterations is expected and even encouraged. This enables being able to switch directions quickly, based on customer needs; it enables being agile.

The term agile software development was introduced in 2001 by The Agile Manifesto [15]. Since then, many papers have been published on the subject. As agile methods are still relatively new, not much has been written about the long-term effects of agile methods. More specifically, not much is known about the maintainability of software developed using agile development methods.

Software maintenance can be divided into 4 categories [17]: adaptive, corrective, perfective, and preventive maintenance. Adaptive maintenance includes all efforts aimed at changing the software to respond to a changing or changed environment. Corrective maintenance comprises tasks to correct discovered bugs, while preventive maintenance is composed of modifications aiming to prevent any faults in the future, i.e. find and fix bugs that have not yet been discovered in the practical environment. Changing the software in order to optimize its performance or maintainability is referred to as perfective maintenance. In this paper, the term maintainability refers to maintenance work related to any of the four categories unless explicitly stated otherwise (i.e., by mentioning the specific category).

My paper is organized as follows. In the next section, I will present the goal of my research and the resulting research question and sub questions. Section 4 describes the method of my research, i.e. which steps I take in order to answer the research question and sub questions. Section 5 presents my findings in literature on software maintenance, agile development methods and discusses previous research on a combination of both. In section 6 the obtained knowledge is synthesized to answer my research question and sub questions. Finally, my research is concluded in section 7.

2. PROBLEM STATEMENT
As mentioned in the introduction; little is known about the maintainability of software that is developed using agile methods. I want to explore this area by determining the relationship between agile development methods and maintainability. The focus of past empirical research has been on the direct, short-term effects of using agile methods for
software development and on analysis of the agile methodology, rather than on future implications [1, 8, 12].

According to prior research, software maintenance accounts for at least half of the total effort and money invested during the life cycle of the software system [4-5, 24, 41]. Therefore, it is essential to reason about the potential effects agile development methods have on the maintainability of the developed software. If adopting agile methods would result in a dramatic increase in maintainability efforts, for example, it may not be the best choice for software that is to be used for an extensive period of time.

This leads to the following research question:

To what extent do the characteristics of agile software development support long-term maintainability?

In order to help answer this question, three sub questions have been defined:

1. Which factors influence software maintainability?
2. How can software maintainability be measured?
3. Which characteristics should a software development method have to be considered agile?

3. METHOD OF RESEARCH

A literature review will be my method of research. Firstly, I want zoom in on software maintainability. The goal is to get a list of factors that influence the maintainability process. This will be in part based on software maintainability prediction models, empirical case studies, and software maintenance measurement metrics. During this phase, answers to sub questions 2 and 3 are constructed. I will look at all four categories of maintenance, which are described in the introduction.

When factors that impact maintainability are known, I first determine the answer to the first sub question, which defines what exactly an ‘agile software development method’ comprises. This allows me to make a generalization which describes all agile software development methods, instead of only specific ones.

In the next step I identify the relationship between each of the resulting characteristics of agile software development and the factors that influence software maintainability. Literature on agile development methods and empirical studies in the area of agile software which include experiences of industry professionals are used in this process to find these relationships.

Figure 2 visualizes my research approach. My goal is to identify the factors $F_{1,a}$ which influence software maintainability, as well as the different relationships indicated by the question marks. The result of my research is a set of research hypotheses regarding the effect of choosing agile development methods on software maintainability, which can be verified in future research.

4. LITERATURE

In this section, the results of my literature study are presented. The first subsection describes findings on software maintenance. The second section gives an overview of current literature about agile software development and its characteristics. The end of this section discusses previous work done on the topic of this paper; the relationship between agile development methods and software maintenance.

The primary sources of the literature are the Scopus bibliographic database and Google Scholar which searches through many databases including the IEEE Xplore, CiteSeerX, ACM Digital Library, and SpringerLink. In the table below the search terms per subject are listed. The method described by Webster for a structured literature review was used [43].

<table>
<thead>
<tr>
<th>Subject</th>
<th>Search terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Maintenance</td>
<td>software maintenance, software complexity, software maintenance metrics,</td>
</tr>
<tr>
<td></td>
<td>software complexity metrics, software maintenance costs, software maintenance</td>
</tr>
<tr>
<td></td>
<td>maturity model, software maintenance characteristics, software maintenance</td>
</tr>
<tr>
<td></td>
<td>quality, software maintenance factors</td>
</tr>
<tr>
<td>Agile Development Methods</td>
<td>agile development, agile software, agile programming, agile methods, agile-</td>
</tr>
<tr>
<td></td>
<td>developed empirical, xp agile case, scrum industrial case, agile methods</td>
</tr>
<tr>
<td></td>
<td>maintenance, agile methods impact, agile methods longitudinal, agile</td>
</tr>
<tr>
<td></td>
<td>development long term, agile development methods</td>
</tr>
</tbody>
</table>

4.1 Software Maintenance

Literature on software maintenance can be grouped in two categories: factors influencing the maintenance effort and metrics to measure a software system’s maintainability.

4.1.1 Factors influencing maintainability

Banker et al. identify two component activities in software maintenance, systems analysis/design and coding/testing [4]. The amount of analysis/design in a maintenance effort is measured by the amount of Function Points, a metric which is proposed by Albrecht [3] as a measure for the size of a system’s functionality. Coding/testing is measured by counting the amount of source lines of code (SLOC) added. The goal of the research was to identify environmental factors influencing the amount of labor hours required for a maintenance job. In order to find these factors, empirical data was collected from projects at a large commercial bank and this data was analyzed. The most significant environmental factor negatively influencing the amount of labor hours required turned out to be the use of a structured design and
analysis methodology. The authors say this is measured only for the short term; using a structured design may lead to benefits in the long term.

The factor which has the most positive influence on the hours spent is ‘good response time’ of the hardware used for developing. Other factors identified were ‘higher ability staff’ (positive influence) and ‘absence of application experience’ (negative influence). The factor ‘good response time’ is related to the hardware used in the maintenance process, and is of lesser importance at this time due to heavily increased processing power when compared to the time of the study.

Schaefer mentions 80% of the modifications will be made to 20% of the software modules in a system [38]. Kemerer and Slaughter wanted to verify this in their own research. With 27% of the modules receiving 80% of the total modifications, similar numbers were found [20]. Furthermore, Kemerer et al. used their empirical investigation to determine whether there are predictable maintenance patterns for the business systems of a commercial merchant.

Based on literature and empirical observations, they believe three factors influencing the amount and type of maintenance work a software module is likely to receive are functionality (strategic/important or not), development practice (automatically generated code or not), and software complexity (measured by normalizing the McCabe Cyclomatic Complexity [28] using lines of code). Apart from those factors, two control factors are introduced which are believed to not have an impact on maintenance: age and size. This assumption was proven to be wrong. The most important results of the study are significant relations between (1) high complexity and high repairs/enhancements, (2) high age and high repairs, (3) high size and high repairs, and (4) high code-generation and low repairs/enhancements.

Shen et al. add additional support for a relation between age and maintenance by noting repairing errors gets more expensive the later they are found [39]. As such, even if old modules do not contain more errors than newer ones, it still takes more effort to repair them. Niessink en van Vliet analyzed the maintenance of a large financial information system and used Albrecht’s function point methodology [3] as well as an alternative they developed themselves as a measure for the functional size [31]. They found the size of a component has a strong impact on the maintenance effort required for changing that the component.

Similarly, Ahn et al. developed a software maintenance prediction model which uses function points to estimate the required effort [2]. The effort is adjusted to the maintenance environment by using a number of value adjustment factors which relate to (1) the developers, (2) technical characteristics of the code, and (3) the maintenance environment. It was concluded the unadjusted function point model provides a good estimation of the required effort which can be further improved by the proposed factors. The authors note the influence of their factors was not as big as suspected.

A second study by Banker et al. [5] confirms software maintenance costs are significantly affected by software complexity and size. Furthermore, they mention developer skill and experience are often overlooked but should be acknowledged as an influence factor. These two factors are distinct; one does not imply the other. For example, a highly skilled developer would not be able to be very productive if he is unfamiliar with the software system.

Von Mayrhauser and Vans continue with this ‘people factor’ and explore the way programmers comprehend programs during software maintenance [42]. Five cognition models are discussed, as well an integrated model created by the authors. It is recognized documentation and manuals are oftentimes used as guidance in understanding code, but the expertise of the developer has a greater influence program comprehension. Yau et al. [46] explicitly mention ‘Understand program’ as a big part of the maintenance process, claiming the understanding comprises three parts: complexity, documentation, and self-descriptiveness. The latter refers to how well the code is understandable on its own, i.e. without supporting documentation.

A survey study by Kim and Westin among 149 electronic data processing professionals [21] confirms understandability of code is an important factor in the maintainability process. More evidence for the existence of a relationship between software complexity and maintenance costs is a survey of empirical research by Kemerer [19] where the following is concluded: “Considerable effort has gone into correlating complexity metric scores with increased effort, errors, changes or all three, and it seems clear that strong relationships do exist.”

Yau, Collofello, and MacGregor [45] analyze the ripple effect which results as a consequence of program modification. Two types of ripple effect are discussed; a logical ripple effect and a performance ripple effect. The logical ripple effect is the ‘trivial case’, e.g. changing a program variable X in some way will introduce possible errors in other parts of the program where X is used as well. Using the logical ripple effect we can create further support for the idea that higher software complexity leads to higher maintenance effort. Software which is more complex will likely contain more variables in total, hence when a complex module is changed more variables are changed. As a result, the introduced ripple effect will be larger for complex modules when compared to simpler modules.

The second type of ripple effect is the performance ripple effect. This ripple effect occurs when program changes lead to a change in performance in other parts of the program. Contrary to the logical ripple effect, the performance ripple effect is harder to detect automatically. An estimate of the complexity of program modification expressed in programmer effort is proposed, consisting of (1) effort to make the change(s), (2) effort to understand affected modules, (3) effort to examine affected modules, and (4) effort to make the necessary changes to the affected modules. As all the terms are related to programmer effort which is not explained in more detail, this estimation cannot effectively be applied before finding a reliable way to predict the required programmer effort for the various tasks. Kemerer [19] has found some limited evidence that errors caused by the ripple effect are more expensive to correct than primary errors.

In a subsequent paper [46], Yau and Collofello identify the resistance to the ripple effect they defined in [45] as one of the most important software quality attributes. This resistance is named the stability of the software, and is simply defined as the inverse of the ripple effect. The logical ripple effect and the performance ripple effect can in such a way be translated to the logical stability measure and the performance stability measure, respectively. The authors continue by giving a formula to calculate the potential logical ripple effect of a modification to a program, and an algorithm to determine it for a given program.
Kafura and Reddy [18] use seven complexity metrics to analyze three versions of an existing software system and to test whether the metrics show more-or-less the same pattern, and concluded this was the case. To be more specific, all metrics showed an increase in program complexity over its lifetime, as well as roughly the same increase in complexity. This result can be interpreted as all metrics being a valid approximation of software complexity. The metrics tested include metrics referred to before, such as the McCabe cyclomatic complexity measure, lines of code, and the logical stability measure.

4.1.2 Metrics to measure maintainability
Oman and Hagemeister [32] propose a large amount of metrics to measure a software system’s maintainability, and explain how to combine these metrics into an index for the system’s maintainability. The metrics are placed in a hierarchical tree structure. The tree is divided in three main parts: maturity attributes, source code, and supporting documentation. The leaf nodes represent measurable attributes of a software system. While the amount of metrics is impressive, it is surprising none of the metrics are related to the experience or skill of the developers of the software system. Other research has acknowledged experienced developers can decrease the maintenance effort [22].

Using the literature discussed in this section, the first and second sub question can be answered. Factors influencing the maintenance process are identified, as well as metrics which can be used to measure a system’s maintainability. The results will be presented in section 6.

4.2 Agile Software Development
4.2.1 Overview
The Agile Alliance first introduced the term agile software development through The Agile Manifesto [15]. The purpose of the manifesto is worded as: “We are uncovering better ways of developing software by doing it and helping others do it. We value:
- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.”

It is explicitly mentioned both the left and right side of each of the four core values are important, but the items on the left are valued more than the ones on the right.

Abrahamsson et al. [1] systematically reviewed existing literature on agile development approaches, with one of the three purposes being to answer the question what exactly makes a development method an agile one. Abrahamsson concludes this is the case when software development is (1) incremental (small software releases, with rapid cycles), (2) cooperative (customer and developers working constantly together with close communication), (3) straightforward (the method itself is easy to learn and to modify, well documented), and (4) adaptive (able to make last moment changes). This definition corresponds well to the core values mentioned in the Agile Manifesto.

Critical success factors that help software development projects using agile methods to succeed where identified by Chow and Cao in a survey study [8]. Hypotheses were formulated based on prior research regarding success and failure factors, all related to the four attributes of success as defined by [10, 25]; quality (deliver a good working product), scope (meeting all requirements by the customer), timeliness (delivering on time), and cost (within estimated cost and effort). After analyzing the results of their survey among 109 agile development projects, the top three critical success factors were identified as (1) delivery strategy (delivering software regularly, deliver most important features first), (2) agile software engineering practices (e.g., simple design, right amount of documentation, refactoring), and (3) team capability (manager and developer competence).

While Abrahamsson et al. described and compared all available agile methods in their work, the two most common methodologies are eXtreme Programming (XP) and Scrum [8, 12-13, 23]. These methodologies complement each other very well, with XP providing support for technical aspects and Scrum providing support for project planning and tracking [13].

4.2.2 Characteristics
One of the most important characteristics of agile development is the incremental nature; working software is delivered quickly and frequently. The typical time between increments is about 2 to 3 weeks for XP and 30 days for Scrum [16]. In each development cycle, the features to be implemented are jointly decided by the customer and the rest of the development team [30]. Boehm and Turner note that short, focused iterations make the entire (agile) development cycle much more like a maintenance phase [6].

Paecht et al. [33] mention that agile development methods are people-oriented instead of process-oriented and code-oriented instead of document-centric, when compared to more traditional methods like the well-known waterfall model [36]. Documentation beyond the code itself is discouraged, which makes the product knowledge tacit [30]. This is a result of the philosophy that the best way to communicate within a development team is face-to-face, instead of using documents [15].

Generally, agile development teams are small. The average team size is reported as only 9 by Cockburn and Highsmith [9]. This is supported by a survey study by Chow and Cao of 109 agile projects; 64 projects had less than 10 members in their development team [8].

In line with this, Dybå and Dingsøyr find evidence in their systematic review of 36 empirical studies on agile development which suggests agile methods are ‘not necessarily the best choice for large projects’ [12]. Creating a stable upfront architectural design is recommended in such cases, but is explicitly against the nature of agile development [29].

Ideally, agile developers ‘maximize the amount of work not done’ [15]. Boehm and Turner explain this philosophy by arguing agile developers design for the battle, not for the war. The motto is ‘you aren’t going to need it’, and thus a big design up front is not recommended. Designs should be stripped down to cover just what you are developing. Since change is inevitable, planning for future functions is a waste of effort [6].

This section has given an overview of agile software development and its characteristics. The obtained information provides an answer to my third sub question and is used to formulate an answer to my research question. Before that, related work found in literature is discussed in the subsequent section.
4.3 Related work
While no studies have been found which deal specifically with the maintainability of software developed using agile methods, there have been quite some longitudinal empirical studies regarding development teams using agile methods which are discussed below.

Marchenko et al. contributed to this area with an empirical study on the effects of applying Test-Driven Development (TDD) over a period of three years at a Nokia Siemens Networks team [27]. TDD is a development technique which prescribes no functionality is implemented until a test for that functionality exists; the development is hence driven by tests. TDD is an integral part of the aforementioned eXtreme Programming. The authors found no negative effects of using TDD and see improvements in code quality and explicitly mention the improved maintainability as a result of using TDD. Code quality is generally measured in many dimensions (among which are understandability, reusability and testability [35]), but sadly the authors do not specify in which dimension the code quality increased or how this increase was measured. Less overhead in code and documentation is achieved by only writing production code for functionality which is addressed by a test, which fits with the agile philosophy of only developing functionality which is required without predicting possible future needs.

Sanchez, Williams and Maximilien post similar results after having studied an IBM team which used TDD for five years [37], reporting higher quality and decreased defects. The fewer defects have contributed to the higher quality of the code so these results are not independent. The authors also mention a possible decrease in the degree software complexity increases as the software ages.

In a different case study, Fitzgerald et al. followed a development team for three years which adopted a selection of both Extreme Programming and Scrum practices and reported lower defect density and increased team communication and morale [13]. A survey study by Parons et al. found the combination of eXtreme Programming and Scrum is the most effective combination of agile methodologies [34]. While a link with maintainability is not mentioned by the authors directly, it is trivial to notice fewer bugs lead to less maintenance in the ‘corrective’ category; thus reducing overall maintenance work.

In contrast, a three year study by Li, Moe, and Dybå on a project using Scrum could not verify the lower amount of defects as a result of using agile methodologies [23]. A 17-month phase of plan-driven development was followed up by a 20-month period of Scrum and no significant differences were found in defects. An important difference with the studies mentioned above is the lack of any of the XP practices which are focused on the ‘coding’ side of development. Scrum is mainly aimed on project management. Another difference, and perhaps a more important one, is the fact the Scrum project was not started from scratch but rather continued with the work of the previous 17 months.

Capiluppi et al. analyzed evolution patterns of a system developed using eXtreme Programming for two and a half years and their main finding is the code complexity is low, while the complexity control work is relatively high [7]. Complexity was measured using the McCabe Cyclomatic Complexity metric which is mentioned in section 5.1. Complexity control work is done to reduce code complexity by refactoring. Refactoring is changing code internally, without changing its external functionality [14]. An example of refactoring is putting code fragments which are frequently used in a function to avoid code duplication.

5. RESULTS
Using the literature discussed in the previous section, the research question and sub questions are answered here. First of all, I talk about the first and second sub question regarding the factors that influence software maintainability and how to measure software maintainability. The factors identified in research as discussed in section 5.1.1 are, in no particular order: software age, software complexity, skill of developer, experience of developer, software size, and understanding of code.

The maintainability of an existing software system cannot be measured directly, as maintainability refers to how easily the system supports future error correction, feature addition and other activities described earlier. Because of this, the maintainability of a system is expressed as a set of measurable properties of the system which all affect maintainability. The research by Oman and Hagemeister provides metrics for measuring maintainability and a way to combine these measurements into a single index of maintainability [32].

Next, a definition of agile development methods is given to address the third sub question. A development method is considered agile when it is incremental, cooperative, straightforward, and adaptive as defined by Abrahamsson et al. [1]. This is backed up by the Agile Manifesto [15] as well as the other discussed papers of section 5.2. The characteristics of agile development methods as noted in section 5.2.2 and the results of section 5.3 will be used in conjunction with the answers to the other sub questions to formulate an answer to the research question.

My research question is answered by taking the factors identified in literature on software maintenance, and finding relationships between those factors and agile software development using the relevant literature. After that, I will look at some of the characteristics of agile software development which have an impact on software maintenance, and I will explain the kind of relationship between them. I will reason about the relationship between agile development methods and each of the factors below.

5.1 Size
One of the key philosophies of agile development methods is to use a simple design. This enables reacting quickly and efficiently to changes in system requirements as a result of changes in the environment. An expected effect of a simple design would be a smaller program size. However, no convincing evidence is found in literature on this matter.

The systematic review by Dybå et al [12], for example, points to studies which show an increase [11], decrease [44], and no difference [26] between product sizes for agile development projects when compared to traditional projects.

The relationship between agile development methods and size can thus not be determined reliably.

5.2 Complexity
A couple of the agile development characteristics support the assertion the complexity of software developed using agile development methods is generally lower than software developed using traditional plan-driven methods. One characteristic is the ‘simple design’-philosophy. It encourages
developers to stay away from large, complex designs which accommodate a large array of functionality, part of which will either be used in the future or perhaps not at all. All of this is supposed to be excluded from designs according to the agile philosophy. The most used complexity metric, McCabe’s cyclomatic complexity measure [28], is based on the number of unique paths in a software program. In general, supporting more (future) functionality in a design will result in additional paths through the program, e.g. by additional ‘if-then-else’ statements. Because of this, supporting less functionality will result in a lower program complexity.

A study mentioned in section 5.3 by Capiluppi et al. [7] has confirmed this and reports lower code complexity for a system developed using XP. The lower complexity is partially attributed to the constant usage of refactoring, another practice of XP.

5.3 Age
Agile development methods have no influence on the age of a system. Whether a program is used for an extensive period of time – and therefore increases in age – depends on many variables which are independent of the applied development method such as available funding for the project, achieved success in industry and so on.

5.4 Experience
Experience of the developer with the software is, just like software age, not influenced by the chosen development method. Experience is gained through simply working on a project. Even though there is no relationship with agile development methods, the factor is believed to have a big impact on the maintenance effort required, and as such cannot be left out.

5.5 Understanding
Agile development methods rely on face-to-face communication in daily meetings to share information, documentation beyond the code itself is not recommended [15, 30].

However, this focus on verbal communication has a downside: information of the product and its features exists within the heads of the developers but is not documented in a design document of some sort. Earlier research presented in section 5.1.1 mention documentation as an important part of understanding [21, 42, 46].

Members of the agile alliance argue documentation does not equal understanding, writing is an inefficient communication medium; essentially saying written communication is inferior to face-to-face communication.

The lack of documentation is not necessarily bad for understanding as long as the developers who do the maintenance are the ones participating in the meetings where the information is shared. In such cases, the daily face-to-face might be very good to share the available information among developers. Because of the increased intra-team communication, Fitzgerald et al. reported increased team morale in their study where XP and Scrum were applied for three years [13].

There are a couple of cases where the lack of documentation can become problematic for understandability, however. For example, if the developer who designed the most important part of the software leaves, his reasoning behind the design choices made is not present in documentation and maintainers responsible for extending the design might be puzzled why certain decisions were made. The same will happen if the maintenance is carried out by a separate team which will not have access to the original developers.

To conclude: the way agile development methods affect understanding is greatly influenced by the way the organization is set up and the maintenance is carried out. If the same team responsible for the development is also maintaining the product, it is unlikely there will be any issues. Instead, the frequent face-to-face might prove to be better. In contrast, if the system maintenance is outsourced problems will arise. As such, I cannot say if the relation would be positive or negative.

5.6 Skill
Just like developer experience, developer skill is not something which is affected by the development methodology. It is a property of the developer which is related to his education, talent, motivation and willingness to learn. There is no relation with (agile) development methods.

6. FACTORS INTRODUCED BY AGILE METHODS
Apart from the factors identified in studying software maintenance literature, I recognized properties of agile development methods which have an effect on maintainability.

6.1 Defects
Discussed in section 5.3, research suggests Test-Driven Development, one of the XP practices, reduces the number of defects in the developed software [13, 37]. Trivially, fewer defects in software lead to less effort repairing bugs, i.e. less work in the corrective maintenance category. It has to be noted TDD is part of XP, but can also be used in traditional methods and is not necessarily part of other agile methods. However, as XP is the most used agile development method and recommends using TDD it is a result not to be skipped.

6.2 Changing design
Many of the agile development practices are aimed at being able to quickly react to a changing environment, changing system requirements when needed. While specifically the simple design and constant refactoring are aimed at making sure changing the system design can be done quickly; it cannot be denied changing a design frequently takes effort. In particular, regularly adapting to the environment translates to maintenance work in the adaptive category.

All of the above is summarized in figure 3, where a minus sign (−) is used to indicate a negative relationship, a plus sign (+) indicates a positive relationship, and the plus-minus sign (±) is used to indicate a relationship which is neither positive nor negative. When there is no arrow at all, no relationship exists.

7. LIMITATIONS
There are some limitations of this research which have to be addressed. First of all, some of the literature on software maintenance originates from the 1980’s or early 1990’s. The field of Software Engineering has changed dramatically since that time and the factors influencing the maintenance process may have to be updated as well.

Another limitation is the heavy representation of eXtreme Programming in case studies on agile development methods. Because of this, findings backed up by those case studies
potentially do not apply to all existing agile development methods.

Lastly, some of the identified relationships are not heavily backed up by empirical research but are based on the characteristics and philosophy of agile development methods. The limitation of that approach is the fact the philosophies and characteristics are presented by authors of the agile manifesto and may not be totally impartial.

8. FUTURE WORK
This research uses literature to reason about the relationship between agile development methods and maintainability. Future work can be done in order to validate the findings presented in this paper, by setting up an experiment to specifically test the impact of development methods on maintainability. Prior empirical research has not dealt explicitly with this relationship. Instead, most empirical research has focused on other specific aspects such as programmer productivity and error count, measured mainly for the short term. It would be interesting to measure the amount of hours required for maintaining a program developed using agile methods when compared to a program developed using a traditional plan-driven approach over a long time. In order to compare the results, everything but the development method should be as similar as possible. The problem

9. CONCLUSIONS
Studying research papers on software maintenance, six factors are identified which influence the required maintenance effort for a software system: size, complexity, age, developer experience, understanding, and developer skill. Using the characteristics of agile development methods and empirical research, relationships between the factors six factors and
Agile development methods were established. Additionally, two more factors were added to the model based on the discussed effects of agile development methods in research; fewer defects and a changing design.

It is concluded there exists no relationship between agile development methods and program age, developer experience and developer skill. Previous research has addressed the relation with program size but there has been no agreement in that area, thus nothing final can be said about the relationship. Similarly, the impact on program understanding can be explained both ways.

Agile development methods do seem to decrease complexity as well as the number of defects, which has a positive impact on maintainability.

An identified negative impact on the adaptive maintenance effort is caused by what is considered to be one of agile development’s strong points; constantly adapting the requirements to a changing environment.

10. ACKNOWLEDGEMENTS
I want to thank Diederik C.F. Rothengatter for his supervision of my research, as well as for his constant and invaluable feedback. Also, I want to thank all my student colleagues for reviewing this work and providing insightful comments.

11. REFERENCES
[23] Li, J., Moe, N.B. and Dybä, T., Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. in, (2010), ACM, 1-10.


[40] Smith, P. Waterfall model.


