ABSTRACT
Changing requirements is a problem that, to a certain degree, can be eliminated by the use of simple prototypes. By utilising simple tools to test ideas and solutions, we can improve our understanding of individual components as well as of whole systems. We can also create advanced prototypes which simulate the behaviour of system components and devices. Such simulators do not only help us avoid dependencies towards external devices, they also enable automatic testing of such systems without the need of handling the device. This paper tries to apply the concept of prototyping used in Interaction Design to the Software Engineering design process. The paper also covers agile methods and describes how simulators can be used in order to enable work with non-existing devices. These techniques can help developers predict and handle changes in the requirements in an efficient way.

Keywords: Prototyping, Software Engineering, Requirements, Simulation, Agile Development, System Architecture.

1. INTRODUCTION
One common problem in software engineering is that the requirements change; often because they were badly understood, and therefore badly designed from the beginning. By using prototypes to explore solutions to problems, we can improve our understanding of the system, and therefore formulate the requirements in a better way. But prototypes and models are not always enough to fully understand all the requirements in a project. Sometimes a custom device is part of the project. If such a device is available to the developer team throughout the project, things might work out just fine. But what happens when such devices are unavailable? The device might even be in development at the time the project that depends on them is started. In such circumstances simulators can be used, as a form of active prototypes, to enable development of a system compatible with the device.

There are areas within software engineering that requires skills and knowledge that generally is not considered belonging to the traditional engineering skills. Usability is a good example. It is very easy to say that a computer program should be easy to use, but at the same time it is hard to define what makes a computer program user friendly. Interaction designers solve problems like these by creating prototypes of the Graphical User Interface (GUI) and then performing tests on people who are likely to use the software in the future. If the test persons understand the GUI prototypes, they will understand the software GUI too. Some of these methods can be adapted into other areas of software engineering, which would help software engineers to better understand what they are working on.

1.1 Problem definition
This paper will explore the possibilities to adapt the prototyping process used by interaction designers into other areas of software engineering; in order to achieve a greater level of understanding for what needs to be done. It will also examine alternative solutions for external devices, etc., which are hard to prototype in a project. This paper also covers the basics of agile development, using Scrum as an example. Agile methods encourage changes to the requirements throughout the whole project, and therefore benefit greatly from the use of extensive prototyping in order to understand the changes later in the project. The focus has been set on smaller development teams and smaller companies, which are not as likely to have the required funds to hire an expert to help them solve the problem.

1.2 Summary of Contributions
This paper summarises how prototypes and simulations can be used in order to better understand and interact with projects where the requirements are likely to change.

1.3 Related Work
Prototyping is not something new in software engineering. A lot of studies have been performed over the last 30 years. Most of those studies cover one or more of the following areas:

- Computer Enhanced Prototyping Systems
- Code generating prototypes
- Rapid prototyping
While these areas are common, it is hard to find information on how we can use low-fidelity (lo-fi) prototypes to understand system architecture and other parts of the system. It is not uncommon that lo-fi prototypes are mentioned, but it is uncommon that they are mentioned in other circumstances than GUI design and interaction design. The publications in the field of small-scale device simulation are also limited. While there is a lot of information on how industry-size simulators are developed and used in software projects, the use of less advanced simulators seems to have been overlooked, or deemed less important.

1.4 Organisation of the paper

The following section of the paper gives an overview of how prototypes work and the benefit of using them. Section 3 covers the use of simulators and the possibility to work with devices which are unavailable. Section 4 is about agile methods, and uses Scrum as an example. Section 5 presents an example of how prototypes can be used to test system architecture. Section 6 summarises the paper and the conclusions are presented in Section 7.

2. PROTOTYPES

A prototype is more than a simplification of something: it is a tool. Even a very simple prototype can help the developer team to improve their communication, easily test new ideas and presenting ideas to customers [3], [9]. The improved communication is based on the ability to name parts of the project in a more natural manner. Examples can also be shown on the prototype instead of being described in words. A prototype also helps the developer team to get a better mutual understanding of the workflow in the product, as well as the project itself. This is of course a big step forward compared to each developer having their own opinion on how everything is supposed to work.

2.1 Different kinds of prototypes

A prototype can be constructed in many different ways. Lo-fi prototypes are often constructed out of paper or cardboard, while a high-fidelity (hi-fi) prototype might be a complex computer program which simulates a larger system. In fact a prototype can be a computer program with similar functionality to the program being developed. In these cases, the prototype might lack security features or be programmed in a quick, but inefficient way. There are several different concepts when developing prototypes. The lo-fi prototypes are often considered throw away -prototypes [9].

This means that the prototype is used to evaluate something and is then thrown away. A throw away prototype should never be implemented into the final product [9], [10]. In rare cases, a lo-fi prototype might be improved over time and upgraded to a hi-fi prototype. A hi-fi prototype is much more complex than the lo-fi version. This means that it takes longer time to develop and that it has a more refined appearance than the lo-fi version. It might also contain errors in functionality and so on, being more complex. Hi-fi prototypes should be developed with care, or they might cause problems instead of solving them [9].

In some cases, a hi-fi prototype can be converted into the final project (evolutionary prototyping). An example could be a GUI built in a design tool. Even if it does not have any real functionality in the beginning, it is possible to use it as a hi-fi prototype to test the customers understanding of the GUI. Later in the project, all the GUI components can be connected to the system backend, which makes it an integrated part of the project. Evolutionary prototypes should be seriously tested for errors before implementation into the project [9].

2.2 The basic concept of prototypes

Prototypes made early in the project should be made in a very simple way, with as little effort as necessary; as this encourages testing of new ideas and change [9]. If too much time and effort have been put into a prototype, the team might feel that additional changes will make their previous work obsolete; which in turn limits the team’s motivation to find new and better solutions [9].

Another reason to keep it simple is that a prototype which lacks a few features helps the team understand what really is missing [9]. Therefore, parts which are not specified in the requirements should be left out in the beginning. A sloppy design of something that is not in use (as an example, a low priority requirement), might still give the team the idea that the feature exists and works. It is much better to leave such a feature out, until the time comes to design that feature specifically. Later in the project, prototypes often contain more elements and are therefore more complex than the early prototypes. As long as the elements are added over time and tested properly, it is natural that the prototype grows as the project progresses; the important thing is to not add everything at once.

2.3 Different uses

In interaction design, a lot of the prototypes focus on user interfaces and controls; as these are the natural points of connection between the user and the device or system. Such prototypes can in many cases be drawn on paper or cardboard. While these prototypes are created to understand problems of interactivity, prototypes can also be used to test other aspects of products. As an example, a block of wood could help the team to better understand the mobility of a handheld device. If the prototype is too big to fit in a pocket, too heavy, et cetera, the product will suffer from the same problem. While this is not a software engineering issue itself, it is important for the developers to fully understand what product they are working on.

2.4 Interacting with prototypes

The active use of prototypes varies depending on which phase the project is currently in. In the beginning, the prototype is mostly used by the developer team itself as they try to get the basics of the system worked out. In later stages of the project, the prototype can be used in order to test usability and to allow people writing manuals to begin understanding the system prior to its release [9], [10]. It is often a good idea to let one person handle the prototype during simulations and tests. If several persons handle the prototype simultaneously, the focus might shift from the prototype to the interaction between the people handling it. When the prototype is being used as a communication tool in the developer team, everyone should be allowed to show their examples in turn and explain their solutions.
2.5 Project cost
Some might say that prototyping takes time, and therefore cost money. In a way, this is true. But in order to see the real cost one must see the project as a whole. It is true that prototyping takes more time in the beginning of the project, but it also saves time during the later parts of the project [5]. Prototyping also enhances the overall quality of many projects, as well as it inspires creativity and improves communication within the developer team [9].

3. SIMULATIONS
While prototypes generally are a good tool for understanding requirements, there are areas where lo-fi prototyping is hard to do. An example could be a device containing hardware sensors, which sends data to the system. Such a device could of course be prototyped, but a lo-fi prototype would not be very useful in itself. In cases where a device sends data multiple times per second, simulating its behaviour becomes a task too intense for the development team to handle manually.

In such situations it is sometimes beneficial to create a simulator which mimics the most important behaviour of the device, or the devices’ output, and continue to work with that. Sometimes a device is easy to use, which means that the simulator would be more complex than the device itself. While this might seem like a good reason not to develop a simulator, there are a number of scenarios where a simulator still could be used. As long as the device is available and fully working, the development team can continue their work on the project. If the device will be available in two months, or has been sent away for repairs, it instantly becomes a problem. Other problems that might occur are that the developer team only has access to 3 devices, but need at least 8 devices in order for all project groups to work in parallel.

3.1 Simulation of devices
One way to enable development of systems dependent on external devices is to develop a simulator for the device. This does not only enable the development team to continue working, it also enables the team to verify their design before the device is available [2]. In some cases some form of graphical representation might suffice (for simulating output to the device). In other cases, where input into the system is needed, previously collected sensor data could be passed into the system at suitable intervals. Test data must of course be presented in a realistic way, so that the device simulator is fully compatible with the device itself. Simulators can be both simple and complex. A full simulation of an advanced device requires a lot of work, while a simple simulator that sends random integers to a system is very simple. It is important to set a goal for the simulator; it should only do what is needed in order to continue working on the project [4].

3.2 Simulation of non-available devices
Sometimes a project is dependent on a device which is not yet available. While simulation of such devices is harder, and involves a small risk of future incompatibility, it is still possible. Problems might occur if the manufacturer of the device decides to change the devices Application Programming Interface (API), the output format of sensors, et cetera. If a good level of understanding and communication can be established between the device manufacturer and the development team, such co-operative work might benefit both parties in the end. The device manufacturer will get feedback on their APIs and the development team will be able to continue work on their project without having to wait for external delays.

3.3 Simulation of non-existing devices
Simulating a device which is not yet fully designed is of course risky. There are many variables that could change over time and the device could even be cancelled. Again, a good communication with the device’s manufacturer is the key to making it possible. In this case, it is very unlikely that there is an API to use in the project. Should there be one; the situation would be more like simulating the non-available device than a non-existant one. In situations such as this, where we need something we cannot have, we can develop a prototype of the device in order to try to understand it better. Similar products should also be examined closely, if they exist. As we study and learn the devices functionality and details, we can design an API for the device. We then implement a simulator that emulates the way the device works. As long as the research and prototyping have been done properly, this should not be too hard; it is just like prototyping the other parts of the system.

The API and simulator allow the development team to continue working on the project, even though the device is still not available. When the device is available, it might be similar to the simulator, but it might also differ a lot from the design that has been developed. This is, of course, always a risk; especially if the development team have failed to do any decent research about similar devices. In order to use the device, any differences between the simulator and actual APIs must be eliminated. Should the differences be small, it would likely be a good idea to modify the implementation so that it uses the API in a good and efficient way.

It is, however, more likely that the differences between the simulator and device APIs are substantial. Modifying the project to use the device API directly could generate a lot of work, and also make all the implementation of the simulator completely wasted time. The solution lies in an adaptor [7], which has the same interface(s) as the simulator does. The adapter translates all the method calls and values between the device API and the system; which might seem a tedious task, but in the end it might save the developer team a lot of time. Of course, this approach might not be a good solution for all kinds of systems, and the result might vary a lot.

3.4 Other benefits of simulation
Simulation can in many cases be preferable to using a real device. One of the most obvious occasions is when the number of devices is limited, so that not all developers can run their code at the same time. Another important occasion is when the system, the device or both is showing strange behaviour during test runs. A broken sensor could generate malicious data any time, which in turn could generate strange errors. Of course a system should verify that sensor data is valid before using it in other parts of the system, but during the development process bugs occur.
A simulator could deliver data which is recorded from actual sensors and then manually verified to be valid. By using such test cases, the development team knows that any malicious values encountered are a result of the internal corruption of data. It also helps debugging, since calculations can be made manually and compared to the ones in the software when unexpected results occur. Also, simulators performing instructions from a system can be programmed to display debug information when a bad instruction is received; which is much more helpful than the device crashing or simply ignoring the instruction [1]. It is worth mentioning that even if we get every detail right when we implement our simulator, there might still be differences compared to the real device. Things like onboard timers might be a fraction of a second off, sensors might display a reading which is off by half a percent, et cetera. In such cases the simulator might actually help us notice the difference we would not see otherwise.

4. AGILE METHODS

Software engineers can choose from a wide array of different project methods and even design their own, should none of the existing suit their needs. In a classical method, such as the waterfall method, a lot of the design and requirements are analysed in the beginning of the project and then written down in a project design document. Common problems which can occur in the waterfall method include, but is in no way limited to, misunderstood requirements, badly formulated requirements, the customer orders a product that can do "X, Y and Z" but really needs a product which can do "X" in an efficient way, et cetera. The problem is that a lot of work is put into the design of such a system before the customer has the opportunity to realise that something is amiss. If the implementation has begun, the project must be stopped and all the requirements must be evaluated and possibly reformulated. Such stops in productivity increase project cost and risks decreasing creativity in the developer team.

While the waterfall method is still suitable for projects where the requirements do not change, agile methods are becoming more and more common today [8], [10]. One of the reasons that agile methods are becoming so popular is that they are much more dynamic and allows the developer team to adapt to the customers ideas during the whole duration of the project [6], [8]. The project method Scrum is an example of an agile method, which has been successfully used by many companies. While many companies say that they do use Scrum, they might still do things very differently. Scrum is not a set of rules of how to handle a project; instead Scrum should be seen as a set of guidelines and suggestions for how things can be done [6].

In Scrum, all the requirements are converted into User Stories, and stored in a Backlog. (Each user story is a description of a feature that adds business value to the customer.) Then the customer and the Scrum Master agree on how long each Sprint (iteration) should be, which user stories should be included in the sprint, which priorities the user stories have and so on. Then the sprint starts. At this point, the scrum teams (typically small groups of 3-8 people [6]) check out their user stories from the backlog and start transforming them into tasks. Of course all the team members must have a decent understanding of what project they are working on; but at the same time, they can focus on their own user stories instead of a complete design document. It is also important to note that once a sprint has been started, the user stories cannot be changed for the duration of the sprint.

Each scrum team is typically working independently with their own task, solving problems themselves, in a way they feel comfortable with. The status of every team is reported every day at the "daily scrum" meeting, which usually lasts around fifteen minutes. If any questions arise that the team cannot answer themselves, they ask the scrum master. If the scrum master does not know, he or she asks the customer. When a sprint is finished, the teams meet up in a sprint review meeting and summarise the new functionality, which is also shown to the customer. This way, the customer can see the project grow and keep track of any changes. If the team has misunderstood the customer in any way, it will be obvious at the next delivery (iterations usually are a couple weeks long). Compared to the waterfall method, where such errors might exist for a longer period of time and even propagate through the system, Scrum helps the developer team feel that they are doing things right.

4.1 The agile problem

While the waterfall method lacks good ways of handling changes in the project, agile methods encourage changes in the requirements even late in the project. This might of course result in changes of previously completed user stories, but since a change of something would simply be entered into the backlog as a new user story, it would not affect the whole development team. A lot of changes can be made to the requirements during the project. Even if the changes are small, the result might propagate throughout the whole system in the end. While this might seem like a bad thing, it is important to remember that Scrum has functionality to handle change.

In order to minimise the generation of new items in the backlog (due to redesign of user stories), prototyping becomes an even more important skill than in the traditional cases. The ability to quickly create prototypes in which the results of changes can be explored and evaluated can help scrum teams to organise their work in a much better way. It is especially important that the Scrum master is able to utilise a prototype to give clear instructions when questions arise or misunderstandings occur. Good prototyping skills might also produce additional user stories in the backlog [6], [8], which helps the scrum master to plan the project more efficiently.

The combination of agile methods and rapid prototyping helps the development team to work with big systems without having to be afraid that changes might affect future development of the system. In fact, the combination of agile methods and good prototyping skills encourages testing of new ideas and creativity. This can in turn raise the quality level of the project, since the developer team is devoted to their work.
5. SYSTEM ARCHITECTURE PROTOTYPING

System architecture can sometimes be quite complex. While an architectural design might seem good in a Universal Modelling Language (UML) diagram, it might turn out that a lot of things have been overlooked later during development. In order to test and find out such weaknesses in the architecture, we can create a lo-fi prototype and simulate the messages being sent and received by the different components in the system. While this can be done in a number of ways, one simple way of doing it is to write the name of all the components in the system on post-its (one component name on each post-it) and then put them up on a big whiteboard. Then the relations can be drawn between all the components, after which the simulation can start. It might be helpful to start each simulated scenario by assuming that a user has performed an action in a GUI, and then track all the messages being passed on in the system. Someone should be responsible for following the messages that are sent between the components on the whiteboard. The other people in the group should listen and reflect on the behaviour of the system. If messages get passed around a lot between components, without any good reason, the architecture might need some adjustments. Since it is very easy to move a few post-its and draw some new relations between them, this is a very quick (and dirty) way of testing the system architecture. For more detailed simulations, the more traditional message sending simulations such as the swim lanes might be used.

6. SUMMARY

One common problem in software engineering is that requirements often change during the development phase; especially when agile methods are being used. If the project leader fails to see the need for change in time, or chooses to ignore questions that arise early on in the development process, a lot of time can be spent on implementing requirements which do not work well together later in the project timeline [3], [10].

In order to better understand such problems when they arise, prototypes could be used to test ideas and solutions without having to spend time implementing the real solution [8], [9], [10]. While prototyping is an important skill in general, it is even more so in agile environments where changes to the requirements generally are something that is encouraged, even at later stages of the project.

Devices, data sources, API's, and et cetera are also areas where it is important that the project leader keeps track of changes and updates. There are always risks involved when simulation of non-available or non-existing devices is being done. If such a simulator is used continuously throughout the project, without being updated when the device it emulates changes, the result can be a disastrous waste of time and money in the end. It is also important to remember that tools such as prototypes and simulators are just tools; they do not solve problems automatically. If they are used in a bad way, they might hurt the project rather than help it [11]. Software engineers are usually clever and talented people, but not everyone believes in the power of simple tools such as lo-fi prototypes. People who like solid design patterns, where everything is done in a predefined and ordered way, might have a hard time accepting that a "few pieces of paper" can solve any real problems. In their case, this might be true; since prototyping requires a positive and creative mind in order to work properly.

7. CONCLUSION

There are simple but powerful tools available for improving our understanding of projects we work with. Prototypes can be used to model whole systems, parts of systems, components and the inner mechanics of systems. Such prototypes can be produced quickly and at a very low cost, which makes them very adaptive tools suitable for testing lots of ideas during brainstorming sessions or customer meetings. While too much prototyping is bad, experimentation should be encouraged in order to find good solutions to problems. Some even say that "not prototyping" should not be an option when it comes to software development [11]. It is important to let developers share their ideas, both for the sake of the project, but also for the developer’s creativity itself. If the developers create simple prototypes as examples of their solutions, other developers will learn from their examples as well.

Simulators can be quite complex hi-fi prototypes of devices and systems, which in turn helps the developer team to understand and utilise something that even might not exist, into a project. Future work could include studies of how small Scrum teams can add a prototyping step to all of their user stories and how they would handle a situation where several simulators were needed.

8. REFERENCES