Quality Technique: Prototyping

Sebastian Stein
sest04@student.bth.se

Kai Petersen
kape04@student.bth.se

Jan Dielewicz
jadi04@student.bth.se

Johan Andersson
frv00jan@student.bth.se

Abstract

Prototyping is a technique already known since several years for example in production industry. During software development it is also possible to develop prototypes to evaluate or identify possible solutions. Complete prototype based software development models exist.

In this article we present the prototyping technique. We discuss the various different types of prototypes. We show that prototypes can be used to prevent defects and also to make use of software metrics early in development. Connections to quality systems are presented as well. At the end we discuss project we participated in and how they could have been improved applying different prototyping techniques.

1. Introduction

Jain et al. [13] promise improved software quality, increased software productivity, reduced costs, reduced risk, shorter schedules and improved customer satisfaction through the use of prototyping. That sounds as if prototyping is the "silver bullet" [5] everybody in software industry is looking for.

Prototyping is a technique to make the matter of interest (or at least parts of it) visible at a very early stage of development. In general, prototyping is not limited to a specific area of business, although the use and the intention for using it varies.

The underlying idea of the prototyping concept is closely connected to the awareness that in software development there is a necessity for active user involvement [15]. The prototype in this context is used as a discussion basis. From this point of view, prototyping (especially the evolutionary prototyping) could be regarded as the ancestor of the agile software development methods (e. g. [11]).

In software development, prototypes are used for different purposes and in different ways. Very often prototypes are used as communication facilitating tools. The visualisation of something gives the opportunity to make a more or less abstract topic more tangible. This is the reason why prototypes in requirements engineering are successfully used for elicitation, validation and feasibility investigations [23, 16]. For design purposes, prototypes can be used in an exploratory way to find out whether a proposed new solution would work or to find a suitable way to design the user interface [23, 17].

In the following sections we show that prototyping can be used during all phases of software development. The effort needed to build a prototype varies greatly as well. This leads to the conclusion that prototyping is independent of the kind of software project. It can be applied in market-driven projects as well as in single customer projects [23].

1.1. Outline of this Article

In the following section we discuss the various different prototyping approaches in detail. After the strengths and weaknesses as well as reason why to use prototypes are presented. Closely connected to this is the question how to motivate organisations, developers, customers, and managers to use prototypes in software development. In a following section we discuss which defects can prevented by the use of prototypes. A discussion about the relation between prototyping and quality systems like CMMI and ISO 9000:2000 is presented as well. In another section we show how software metrics and prototypes can be used together. At the end we discuss 4 example projects we participated in where a prototype would have been beneficial.

2. Variants and Concepts of Prototyping

2.1. Overview

In literature many different names for different types of prototypes and prototyping concepts exist: throw-away prototyping, rapid prototyping, evolutionary prototyping, exploratory prototyping, experimental prototyping, interface

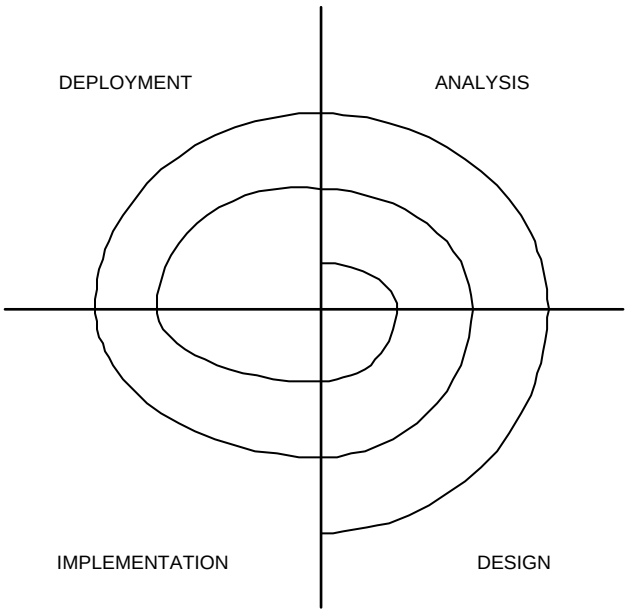

Figure 2. Spiral model [3]

prototyping, etc. A reasonable classification for the prototyping concept is presented by [23]. Sommerville distinguishes between throw-away and evolutionary prototyping. The criterion for this classification is the fact whether the prototype becomes part of the end-product or not.

For prototype implementation a large variety of different media are in use: pen and paper, whiteboard, visual tools (painting programs), executable prototypes, etc. [24]. Often the possibility to actually use the prototype is related to the fidelity of the prototype. In that sense a pen and paper prototype is a low-fidelity prototype whereas an executable program is a high-fidelity prototype [22].

In figure 1 on page 3 the prototypes distinguished according to their purpose are illustrated. Each of those prototypes is described in more detail in the following subsections. We also discuss the concepts of high/low-fidelity prototypes and vertical/horizontal prototypes.

2.2. Evolutionary Prototyping

Evolutionary prototypes [23] are developed iteratively and hence they have a strong relation to classical evolutionary process models like the spiral model [3]. A classical spiral model repeats the software development phases analysis, design, and implementation until the result is satisfactory. The spiral model is shown in figure 2 on page 2. At the end of the incremental process, as it is the case in the context of evolutionary prototyping, a complete product is deployed. Therefore we interpret evolutionary prototyping as a software process model rather then an artifact.

When developing software using the evolutionary prototyping approach the user is given the opportunity to comment the prototype in each iteration. Hence the prototype is refined iteratively according to the comments of the user. This is done until the product has achieved a satisfactory state. Figure 3 on page 3 illustrates the evolutionary prototyping process [23].

As stated by Sommerville [23] the process model is only suitable for small and medium sized systems, because otherwise several problems occur. Management problems occur because prototypes evolve very quickly and hence it is hard to define accurate milestones and do proper planning. Furthermore managers often lack of experience in managing evolutionary prototyping. Other problems are related to maintenance, because of continuous change in the software structure and software architecture. Like in management a lot of experience in maintaining evolutionary prototype systems is needed. Moreover contractual problems may occur, because there is no detailed specification in the beginning fixing what should be delivered at the end. Another thread in this context is that change requests on the part of the customer can not be controlled. Hence it is hard for the development company to determine the effort.

Because of this Sommerville [23] proposes an incremental process model very similar to the classical spiral model suggested by Boehm [3]. First an overall software architecture is established. This architecture is implemented incrementally. When components and interfaces are considered as complete they are not changed anymore and get into the final system. Later comments and complaints on the part of the customer regarding those components can be taken into consideration in later deliveries. For the process, documents are produced including e. g. plans and architectural designs. Because this process is planned and documented it is easier to manage. Also contractual problems are solved to some extent, because the process is based on defined system deliverables as illustrated in figure 4 on page 3.

2.3. Throw-away Prototype

Contradictory to evolutionary prototyping, throw-away prototypes are not used in the end-item delivered to the customer. Instead, their purpose is to find solutions for problems. As stated by Sommerville [23] and Kotonya et al. [16] throw-away prototypes aim at eliciting requirements by showing possible solutions. Hereby the focus lies on requirements which are hard to understand [16]. When the throw-away prototype is developed the customer can try out the proposed solution. This is a good baseline for further discussions about solutions for identified problems.

Sommerville [23] proposes a process for throw-away prototyping. It starts with outlining the elicited requirements. After that a prototype of the system is developed and

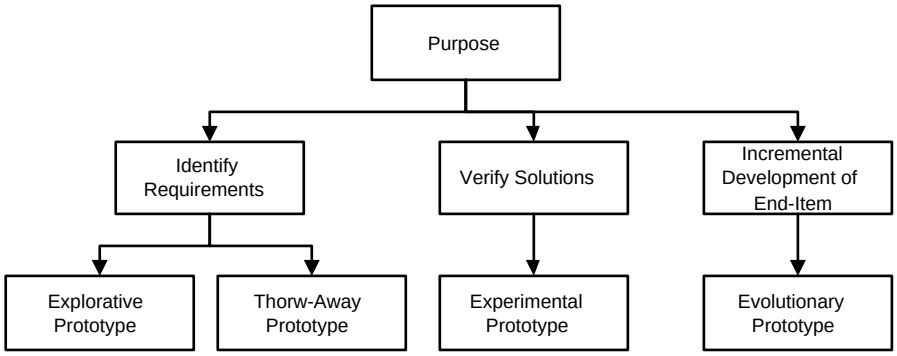

Figure 1. Structure of prototype variants

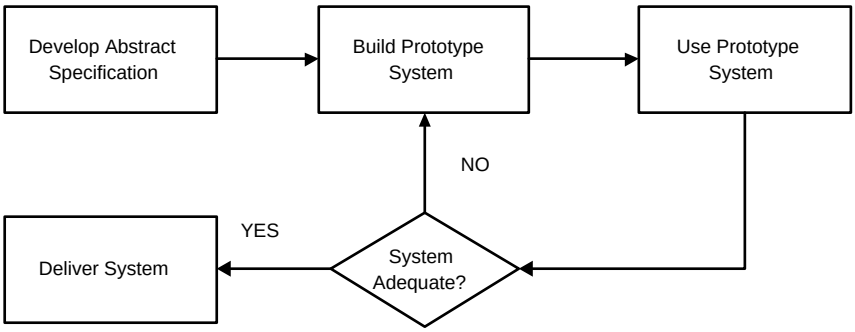

Figure 3. Evolutionary prototyping process model [23]

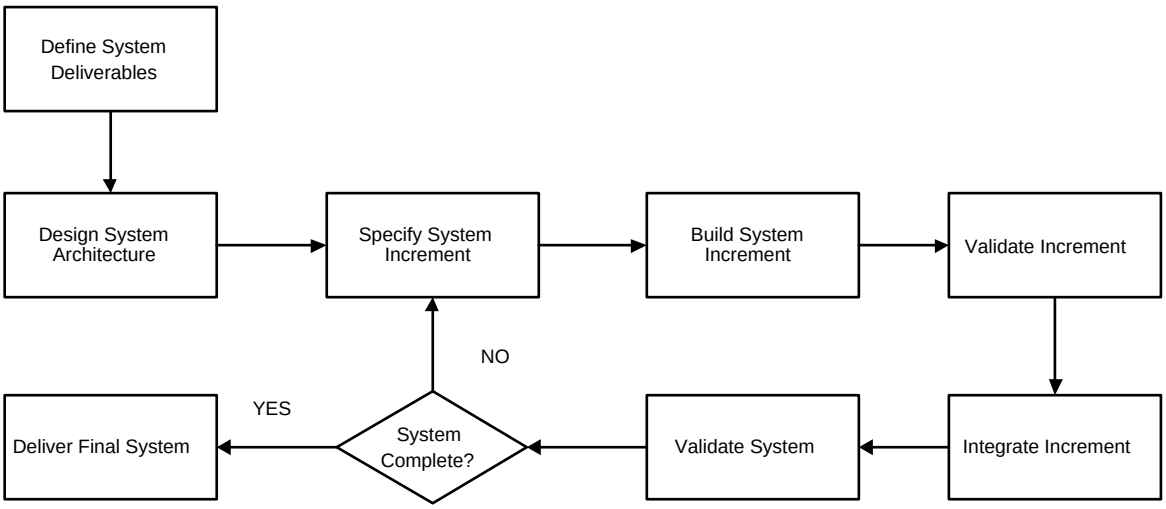

Figure 4. Incremental prototyping process [23]

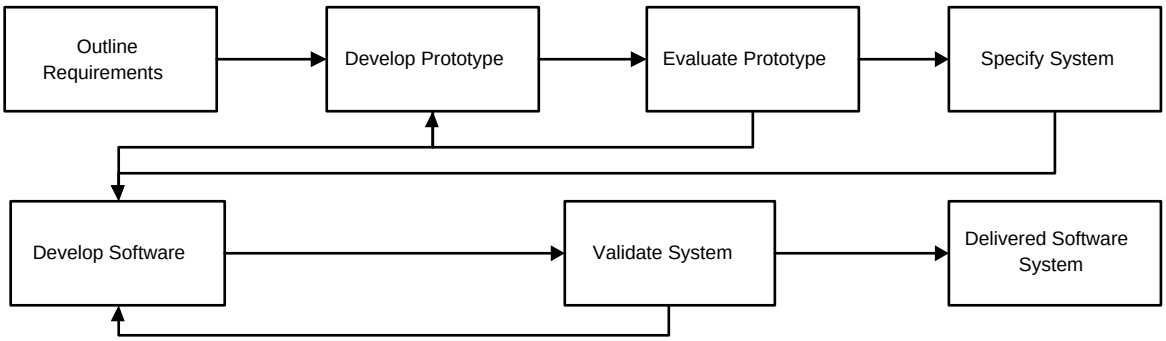

Figure 5. Throw-away prototyping process [23]

shown to the customer. Customer and developers discuss issues concerning the prototype and clarify and identify requirements. This is an iterative process, because the prototype should be modified until it meets the customers expectations, in other words: we have understood the customer properly. Based on this better understanding of requirements the system is specified and the software is developed. In general the prototype is thrown away, but sometimes, as shown in figure 5 on page 4, some parts of the prototype are reused, for example the outlined interfaces.

2.4. Explorative and experimental Prototype

As illustrated in figure 1 on page 3 explorative prototypes have the goal to clarify and analyse requirements. The requirements are unclear like it is the case for throw-away prototypes. Therefore we conclude that explorative prototypes are very similar to throw-away prototypes. However, the difference is that here the prototype should not be thrown away at the end.

Contradictory to explorative prototyping, experimental prototyping aims at clarifying technical issues. This means for example that the developers try different implementation alternatives to validate possible solutions.

2.5. Rapid Prototyping

According to Sommerville [23] rapid prototyping is aiming at delivering releases of products as fast as possible. Hereby the time-to-market is the most important aspect, more important than for example reliability or performance. To develop a prototype very fast tool support is needed. For this purpose often rapid application development (RAD) environments[1] are used. Those environments provide for example user interface builders where one can easily build an user interface by dragging and dropping user interface elements like buttons and edit fields on a form. Such environments provide many reusable components. It can be said that the more reusable components such a RAD environment provides, the more it is suited for rapid prototyping.

Furthermore Sommerville [23] points out that the selection of a programming language depends on the application domain the prototype is developed for. For example, Borland's Delphi is suited for interactive systems, because interfaces and events can be implemented very easily. Contradictory to this, logical problems should be solved by using a programming language like Prolog.

2.6. Vertical vs. horizontal Prototype

Horizontal prototypes show a wide range of features without many details. This can be for example all visual parts of the system [9]. Hereby interactions with the system can be tried out, but there is no data handling or data processing. In this context Sommerville [23] talks about interface prototyping. He states that the end-user has to be involved because he is the person who should be able to handle the interface of the final product. This can be done by evolutionary prototyping, e. g. the prototype is developed iteratively together with the user. Examples for interface prototypes are for example forms created with reusable component libraries like RAD environments. Further HTML is a good approach to develop draft interfaces for web-development projects.

On the other hand, using vertical prototypes small parts of the system are simulated in detail. Such prototypes can become a part of the final end-item [9].

We think that the horizontal prototyping is more suited for users of the future system. They have to get an overview of the whole system. Vertical prototypes are developed to evaluate technical problems like interfaces and database abstraction layers.

2.7. High-Fidelity vs. Low-Fidelity Prototype

Two ways to perform prototyping are low-fidelity and high-fidelity prototypes [21]. Low-fidelity prototypes can

[1] e. g. Microsoft Visual Studio .NET or Borland's Delphi

not be used in the end-product, because they are not detailed enough. Thus they can be considered as throw-away prototypes. In the following we present different types of low-fidelity prototypes.

Storyboard [16]: This approach is similar to a storyboard for a movie where a series of pictures illustrates the different stages of a scene following an order. This can also be done for a program where it becomes transparent how the user will interact with the system. Hereby people start thinking about how the system could be used and they may notice important and missing things. Furthermore questions according to simplicity of dialogs, reading pattern, order and grouping of elements, and the number of elements can be answered.

Paper Prototype [16]: Those prototypes are also referred to as mock-ups. Hereby a user interface is designed on a clipboard representing the screen. One possibility to create these prototypes is to draw a picture on the clipboard or to fix different dialogs (papers with various colors) on the clipboard. Also, as mentioned before, such prototypes can be created with RAD environments. Furthermore graphic drawing programs can be used for this kind of prototype.

Wizard-of-Oz Prototypes [16]: Those prototypes have an interface so that the user can interact with the system. But there is no implementation for data processing. Instead, the feedback is done by a real person, the so called wizard. Hereby a lot of implementation costs are saved, but still it is possible to simulate user interaction with the system.

On the other hand, high-fidelity prototypes are more expensive and implementation is harder [21]. High-fidelity prototypes are a draft implementation of the final product representing the whole system or just parts of it. Those prototypes can be realised for example by rapid prototyping where a basic implementation of the system is made. Moreover high-fidelity prototypes can for example be used for presentations and reviews to validate the system because they already cover a broad set of features which should be included in the final application as well. This kind of prototypes can also be used to test quality attributes like usability.

3. Strengths and Weaknesses

3.1. Strengths and Reasons for using Prototyping

The concept of prototyping allows a very broad use in software development (compare section 2). Because of this prototyping is a very flexible method and can be applied for a large variety of aspects that need clarification.

As already mentioned, and this is a major strength of prototyping, the capability of visualisation the abstract elements of software helps to improve the understanding of the development matter at a very early stage of development. Therefore, prototyping became a standard method in requirements elicitation and requirements validation [16]. Here, prototyping can be used to find out whether all involved people have developed the same understanding of the requirements. This ensures at a very early stage that the project will result in a valid system. In addition, developers, analysts and users will have the possibility to discuss the system in question further and therefore are able to discover additional requirements or refine already elicited requirements to a sufficient level. This all helps to reduce the risk that the system will not suite the customers needs. In the end, this is not only a cost factor, but also essential for the overall success.

By using the prototype, the customer gets very early an impression of how the hired software vendor is going to implement the requirements. This comes along with two aspects: First of all this is a kind of an early acceptance test, at least for those parts that are represented by prototypes. In these cases, the customer has the opportunity to approve or not to approve the suggested solutions. This is especially true for the interface prototyping. The example in section 8.2 shows, what may happen when this aspect is neglected. Especially when a completely new kind of application is built, several usability concepts may be new. This leads to the second beneficial aspect, the user training. Playing with the prototype at an early stage will increase the understanding of the users as well as it will help to identify the most serious problems with new interface concepts. Those problems then can be addressed in a proactive way by developing the user manual or the user-training program. The usability and the product related services are a very important quality factor. In some cases it might even be possible to use the developed prototype for training purposes. This means training of the future users can start earlier which supports the deployment of a new system.

Although some authors (e. g. [8]) state over and over again that building the source code is not the problem of software development, this is sometimes in fact a tough task. In those cases, prototyping helps to figure out how a specific problem can be solved. This is valid for design purposes as well as for implementation aspects. The prototype in this context gives the opportunity to try out new and unproven things, which provides the opportunity for developers to learn new methods of development or technologies. In the project that is matter of the example in section 8.2 we tried out a lot of so called design patterns, like the concepts of observer and singleton, during the development of the experimental prototypes. This helped us to find easy to program and well working solutions for the

challenges during the development. Because of this, the design reached a higher level of maturity. During the ongoing project, we decided to implement a multi-language support for the system. It should have been possible to change the language during runtime. Because of the superior design, it was possible to implement this new requirement relatively easy. This is an example of how prototyping can improve not only the design quality, but also even the maintainability. In commercial projects, this helps to save money and reduce risks for failure. This is especially true, when the same people do the development of the prototypes and the final system. The developers then have the chance to make the serious mistakes during the prototyping and not during the development of the final system.

3.2. Weaknesses

While developing prototypes, there is the risk to end up in a vicious circle of refinement. That means that analysts, users and developers loose the focus of what the prototype was meant for and start to look in details too much. When this happens, the risk not to meet the schedule and the costs increases.

The next weakness of prototyping is connected to the problem before. When a lot of effort is spent in the development of a prototype, it is not easy (because of emotion and of the resources spent for it) to throw this prototype away and start from scratch. This means, the prototype, which originally was meant as a throw-away prototype becomes an evolutionary prototype. One could state that this is no problem, when the prototype works. However, when the prototype was meant as a throw-away prototype, most probably not everything is implemented at a sufficient level of quality and therefore it is better to throw it away. This problem especially occurs when the prototype is developed with the same tools as the final product. To avoid keeping something that better should have been thrown away, one could decide to build the prototype on a lower level of fidelity or use a different technology that is not applicable for the final solution. This of course would reduce the learning benefit. In the example described in section 8.2, one motivation for using prototyping was not only to learn about the best design or specific design patterns, but also about the programming language, the integrated software development environment and additional tools like the concurrent versioning system in advance of the real product development. When one decides to change the tools, this learning effect is gone.

Arthur Lowell Jay [14] spotted that a prototype might challenge the customers patience, since the product is already visible but the developer tells the customer it is not yet ready for use. This again is especially true for prototypes of high fidelity. Possible solutions and the connected problems are the same as just discussed.

3.3. Conclusions

The weaknesses just mentioned show that experience as well as education and training are necessary for prototyping just as it is for all other techniques. The necessity increases with increasing fidelity of the prototype. Therefore, choosing the right level of fidelity is an important aspect when dealing with prototypes.

We have to admit that on the first view prototyping seems to be a very costly method for development. The project must include extra resources for prototyping. This could be regarded as the major disadvantage. On the other hand, as shown by the examples in section 8, prototyping can save money as well. This is not only true because the final system will better fit to the customer's expectations, but also because the developers can directly use their experience from the prototype development for the development of the final system. That means that at least parts of the extra time used for prototyping is going to be saved during the development of the final system.

4. Motivation of Use

In this section we discuss how to motivate organisations, developers, customers, and managers to use prototypes during software development. This includes a discussion of reasons for resistance.

Management might say that prototypes cost too much and should therefore not be done. The customer might think that a project will be delivered later because of the extra time needed for prototype development. Building and reviewing a prototype might cause extra effort for the customer, because he is more involved in the development. This will cause extra costs. Developers might ask why they should develop something which is thrown away later or which will at least not be used in the developed way completely. They might think they are doing useless work.

However, those claims are weak. Everybody will agree that it is useful to first test something out before investing much money. For example before someone buys a car he has at least a test drive and he reads car reviews in technical magazines. Prototypes are used in all other industry domains. An architect makes a small model of a building, prototypes of cars and components are produced before mass production is started.

To motivate people to use prototypes one has to communicate the advantages of prototypes. In a pilot project people must be educated in prototyping techniques. At the beginning only small prototypes like explorative prototypes should be used so that everybody can see the advantages immediately. Developers will recognise that they already used prototypes without knowing that experimenting with possible solutions can be considered to be a prototype. A system

developed using prototypes will more likely meet the requirements of the customer and may have less defects. This means a lower number of change requests can be expected, which will reduce the work load of the developers. This should be communicated to them as well. One can also use reward programs to stimulate the usage of prototypes [12].

Of course the customer will have extra effort if he participates in prototyping sessions. However, the customer also gains influence on the project and the chance increases that the final product will meet his actual requirements.

Using prototypes means reducing risk. A reduced risk level always means higher costs. It is in the end a management decision how much one likes to pay for risk reduction. However, using prototypes can reduce the risk heavily as we show in our examples in section 8. Prototypes might also help to make better estimates. After evaluating a possible solution one can better estimate how much time and money is needed to implement the final product. Furthermore, customers will be more satisfied with the software vendor, because of meeting the actual requirements of the customer by better requirements elicitation and validation through the use of prototyping. The customer might choose again the software vendor for another project, which is the aim of managers.

5. Prototyping to prevent Defects

There are a number of different problems that can be avoided with the use of prototypes. To use the word prototype is a bit to general since there are a number of different prototyping techniques [16]. Each of the different prototyping techniques is useful to apply to prevent one or a small number of different problems. Prototyping is not a silver bullet; it cannot be used to prevent all types of defects. It can however be used to prevent a number of different problems that can occur during the development of an application.

One type of defects that prototypes are good at preventing is the problems that can arise with the use of wrong requirements. A large number of the defects in a product can generally be related to flawed or misunderstood requirements [16]. With a prototype it is possible to get the stakeholders view upon the correctness of a requirement to make sure that it is understood correctly [23]. The use of prototypes often works to the engineers advantage as it makes it easier for them to formulate their points and helps them to explain solutions. The obvious example here is the use of different prototyping techniques to build a graphical user interface that the stakeholders then can evaluate to certify that it is what they want.

It is also possible to evaluate if a requirement is feasible, e. g. if it is possible to implement it at all [16, 20]. This helps avoiding situations where a requirement cannot be implemented and an alternate solution has to be found. If this is done before the implementation phase it prevents possible problems from entering the application.

Problems with the design of a software product can also be made evident by the use of a prototype [20]. By implementing the whole or at least a small part of a design it can be made evident if there are any flaws within the design that have to be corrected. When this is done at an early stage it is possible to avoid integrating flawed solutions into an application, preventing them from causing problems at a later stage.

Design of a database can be hard and avoiding to alter it at later stages during development because of flaws in the database design is important. Prototypes are in this case excellent since they allow for the databases to be implemented and tested before the wrong database design can cause problems [6].

One common source of problems in an application today is interfaces between different components [4]. To discover a flawed interface in a product at a later stage of development can be very costly. Prototypes provide a good way to certify interface correctness at an early stage of development. A prototype allows for the different interfaces to be implemented and evaluated. Should one or more problems occur with the interface it can be corrected without actually having to alter the affected components of the application, which would have been the case if implementation had already begun.

6. Prototyping and ISO 9000/CMMI

ISO 9000:2000 and the Capability Maturity Model Integration (CMMI) are both quality systems. They describe the characteristics of processes leading to high quality products. It can be said that those models describe *how* such a process should look like and not *what* exactly should be done. It is up to the company to define a process according to those quality systems. This means a company has to decide if the prototyping techniques helps implementing a process according to those quality systems.

ISO 9000:2000 is a generic quality system, not aiming at a specific domain. To support the usage of this standard in the software development domain the TickIT [18] guide exists. This guide shows how ISO 9000:2000 can be implemented for a software process. The CMMI [19] directly aims at the software development domain. It already contains several examples where prototyping can be applied to create a software process leading to high quality software products.

Prototyping is proposed by CMMI in the process area requirements development [19] as a technique for requirements elicitation. For example prototypes can be used to analyse the balance between stakeholder needs and environmental constraints.

In the process area technical solution [19] prototyping should be used to define possible solutions for requirements. For example a possible solution gets implemented and analysed to see, if the solution will fulfill the requirements. It is pointed out in this process area that the selection of a specific prototyping technique always depends on the solution as well. For a small problem rapid prototyping might be a good way to go. In contrast, for a multi-year project a much bigger prototype might be needed as well. This also includes evaluating possible software architectures and designs by the use of prototypes.

Prototyping should also be used to validate requirements. This means the user can see how the software engineer understood his requirements by reviewing the prototype developed so far. Wallace et. al [25] point out that validation and verification activities should be done all over the software development process. This is also the intention of CMMI and TickIT. This means prototyping can be applied at every stage in software development. However, it might not be always the most suited technique for this purpose. A software engineer has to decide in the given situation which technique he is going to use.

CMMI [19] mentions prototyping in the process area verification and in the process area risk management as a possible technique for risk mitigation. For example prototyping can be used to check the feasibility of a proposed solution. This also includes to use prototyping in the decision analysis and resolution process area. Here a prototype is developed to show possible impacts of a decision. This supports decision making. All those actions can help saving a lot of money, because wrong solutions are not implemented or are at least discovered early to be no good solutions at all.

Before a software product gets released it must be integrated with all other software systems in the system environment. To facilitate this task, CMMI [19] suggests to use prototyping for this purpose as well. Integration efforts should be started early by simulating the environment and the system components through prototypes. This helps to identify possible problems for integration and to develop an integration strategy.

Furthermore, prototypes can be used to teach the usage of the new system to the future users early. This is a core interest of the TickIT [18] guide.

ISO 9000:2000 emphasises the customer focus and involvement of people [2, see e. g.]. Prototyping is a valid technique to ensure the involvement of the people. It helps to understand the real needs of the customer, which is the aim of customer focus. Customer focus also includes to have good communication with the customer. Prototyping can play an important role in this context, because software engineer and customer have to closely work together to identify possible problems in a proposed solution.

In summary it can be said that prototyping is a valid technique in CMMI as well as in ISO 9000:2000. However, one has to keep in mind that it is just one technique and that just relying on prototyping is not sufficient.

7. Metrics

In this section we present some examples how to combine prototypes and software metrics.

A high-fidelity prototype produced for example by rapid prototyping can be used for any software metric, because such a prototype is very similar to the final product.

A prototype can be used to measure the associations between e. g. classes or modules. Using software metrics like fan-out and fan-in [10] show, if the code is well structured and the principles of high cohesion but loose coupling [7] are done correctly. If the metrics show problems the developers should rethink the chosen software architecture and transform it to a better one. The transformed architecture can be evaluated again using a prototype.

One can develop a prototype with the core functionality. This means only about 10% of the final code size, without any user protection and error handling, and a very basic user interface. Based on this prototype one can estimate the costs and development time needed for the complete product.

Sommerville [23] suggests to use a low-fidelity prototype to evaluate the user interface at an early development state. He argues that at this point of development an evaluation based on metrics is not possible yet, but still the low-fidelity prototype like a paper mock-up will show important results to the developers and the customer.

Another idea to combine prototyping and software metrics is to measure the usability of an user interface. If a test user gets quickly familiar with the use of the prototype one can conclude that the interface is well designed. It can be measured for example how long a test user needs to get familiar in handling 80% of the functionality.

Those small examples show that it is possible and reasonable to combine prototypes and software metrics. However, before things are measured one has to first define the question to be answered by the measurement. Having several software metrics based on prototypes without using them is wasted effort.

8. Examples

In this we are presenting 4 examples of past projects we participated in and where prototyping was used or should have been used. For each example we briefly describe the project and present how a prototype would have improved the project. A cost-benefit analysis is presented for each example as well.

8.1. Example 1: Oversized Functionality

A mistake that happened to one of us is connected with misinterpretation of requirements. The project was concerned with a web-based customer-care-tool. The requirement specification stated something like “It shall be possible to enter and maintain several contact persons per customer and per locality. One of these contact persons shall be distinguishable as the main contact-person. The main contact person may change from time to time.” This requirement was interpreted in the following way: For each customer there may be one or n locations. For each location there may be 1 or m contact persons. For each location there may be 0 or 1 main contact persons. Since all other data connected to this system were stored in a common relational database-system, the database design was extended to that level of detail. The web-form displaying the customer details was extended with a list of all locations. On the next level (another linked web-form) the details for the location was displayed as well as a list of all contact persons linked to this location. The main contact person was displayed in all details. Of course, there were functions implemented to be able to change the main contact-person, to “move” contact persons from one location to another and to link one contact person to multiple locations and so on. Of course, we very much focused on checks like ensuring that there is only one main contact-person per location. Functions to add, change, and delete contact persons were implemented anyway.

Although the developers thought they built a good solution, the customer was not satisfied, because the solution contained more than one web-form for displaying the information. When discussing what was wrong and how to solve this problem, we found out that the solution was far too far oversized for the needs of the customer: 99% of the customers customers only had one location and only one contact person. What the customer really wanted was only a simple text field to add his own qualifier like this: “main contact, receiver of invoices, likes soccer (Bayern München)”. We had to implement some changes: All information had to be displayed on one web-form. Since the extra form for each location was missing now, the relationship which contact belonged to which location had to be displayed in the list of contacts as well. Finally, we had to add a text field for special remarks, which were used for personal conversation at phone, e.g. the latest results in soccer. Of course we did not throw away the extra functionality, but to a very large extend it was useless.

Obviously, the requirement was not specified directly by the customer, but by someone else. For the developers the requirements specification seemed to be clear or at least clear enough. They did not expect such a deviation between specification and the real needs and therefore were not curious. Nonetheless, the requirement specification was wrong, or in other words contained a defect.

In this special case, prototyping definitely would have avoided the problem. This is not so much true for the design and implementation stage but especially for the requirements validation stage. Even only the use of pen and paper to draw an interface prototype (as described in [1]) would have prevented this defect: The pen and paper prototype would have displayed the different data to be displayed, the customer data, the location list, the second level with the location details and the contact list, and so on. In other words we would have had presented our understanding of what was written in the requirements specification, the customer most probably would immediately have noticed that our understanding of the requirement does not match his real needs. The use of pen and paper would have made it possible for the customer then to draw a picture of his expectations without any detailed knowledge of database design or programming.

The costs of this defect are built by the extra effort to implement the unnecessary functionality, the discussion to analyse the problem and the additional time to fix the defect by changing the original solution. The extra time to build the unnecessary functionality was about 25 person-hours. The discussion involved five people and took about 1 hour (5 person-hours). Changing the web-forms took another 6 person-hours. All in all this is an additional effort of 36 person hours.

Since we did not actually perform the prototyping, it is hard to say how much time it would have taken. We have to estimate the effort. Most probably three persons (the customer, one analyst and one developer would have participated in a rather short session. We think it would not have taken more time than the analysing discussion that we had afterwards. That is 1 hour. Therefore, the prototyping would have taken three person-hours only. Overall, it would have saved 33 person-hours. The usual payment per hour in that time was about 80 EUR an hour (not including VAT). That makes a sum of 2640 EUR (about 24.000 SEK). On the first view, 2.640 EUR does not seem to be a large amount of money. When comparing this to the overall project budget of about 35.000 EUR it is quite a number (about 7,5%).

The problem presented here is of very low complexity. Therefore, a very easy and so to say lightweight prototyping can be applied. That means that no one really would have had to learn anything from scratch. A pen and paper prototype for interface design is very much straight forword. As shown, the benefit would have been large enough to justify the effort that the prototyping would have caused.

8.2. Example 2: Lacking Usability

Topic of this project presented next was to develop software as assistance for a harbour master to administrate a harbour during tall ships events. It was run as a students project as part of a course in software engineering. The project, as well as its product, was called "Professional Port Organizer" (PPO). Since the project was very non-specific (no market, no customer) all activities and decisions were very much related to our own ideas of how such a product could look like. When conducting the first acceptance tests with untrained users we received several comments, which indicated that we had problems with the usability of our program. "Daffy game" was the least serious comment we received. During the design, we very much focused on issues we regarded as the most problematic things, e.g. object model and data persistence, 2D representation of the harbour and the ships. Therefore, we extensively used explorative prototypes to check which approach would lead to a good result. Unfortunately, we missed to find good solutions for positioning the ships and changing the shape of the harbour. Additionally the searching for ships also was not straight forward, as the users mentioned. Of course, we thought all these aspects were solved in a good way. Actually, we thought we found even excellent solutions. The reason for this was our ongoing involvement in the project and the program we developed or very own understanding of the program and made lots of the functionality invisible, e. g. using Ctrl-key and Shift-key together with the left or right mouse button and such things. Since we were the ones implementing theses solutions, we did not have any problems in using the software. However, to be honest we understand that users during our acceptance test really disliked our solutions, since the solutions are not very intuitive and far away, what one could call a windows-standard. On the other hand, this program itself also was far away of common office tools or anything else, which is often used. Especially a graphical 2D interface is not very common in usual software products. Therefore the program only was usable after an intensive training.

A prototype for this problem would have had to focus on the graphical 2D representation of the harbour and the ships. The type of the prototype, which could have improved the result, is an interface prototype. In this special case it would have been used in an exploratory way, e. g. the prototype would have been used to figure out which kind of solution for the 2D representation would have led to a good acceptance. In this context, the prototyping can be regarded as an early part of the acceptances test, at least for the interface. This is of special importance, since there were no comparable products available. Just as the other prototypes for technical solutions this prototype would have shown, which way of implementation would work out best. The implementation of the 2D interface was the result of a large programming effort compared to the complete project. Because of this, most probably building the prototype with the same tools and technologies would have caused far too high costs. Therefore, an interface builder (compare to [1]) would have been a good tool for the prototype-development in this case. Perhaps even a static picture of the situation (harbour with ships) would have been sufficient.

During a prototyping session we would have had to find out what the potential users would expect to happen, for example when they use the mouse and perform a left click on a ship, or how they want to move a ship from one position to another. Therefore, a number of questions and scenarios (according to the developed use cases) would have had to be developed. Since the use cases were developed anyway, we estimate the effort for the preparation of the prototyping session with about four person-hours. The prototyping session itself would have been conducted in two or three groups of four users with one developer and for about 1,5 hours (in total 15 person-hours). Processing the results, we estimate with another 8 person-hours. In total, this makes 27 person hours. This would have been the additional investment.

The costs of the lacking usability is a bit hard to estimate, because we never sold or used the program for any purpose. When we assume, that we wanted to launch this program we would have had to work over the 2D interface. The complete development effort spent in the 2D interface is about 95 person-hours. As a rough estimation, we think half of it had to be re-done. The benefit therefore would have been in saving about 24 person-hours. Compared to the complete project (680 person-hours) this is about 3,5% of the overall project.

8.3. Example 3: Hardware Incapability

Hardware incapability was one of the problems that occurred during the development of an Instant Messaging client for the SymbianOS platform. The application had to be run on both the Sony Eriksson P800 telephone and the emulation software that was used to simulate the SymbianOS on a standard PC. The development of the application went well in most regard but there were one setback that was most memorable.

During the development a problem started to occur, we were not able to make the application work over a Bluetooth connection. This was one of the two types of connections we were using for the application, with the second type being GPRS. The reason for having Bluetooth integrated was that it allowed the application to access Bluetooth hubs connected to the Internet. The idea was that to route the traffic through the Bluetooth hubs and access firm Internet connections. This would allow for the Instant Messaging service to be used "without" cost when in close proximity to a Blue-

tooth hub (e.g. a hub in the office where you are working).

During the design phase we included both of the different types of connections, something that also was stated as a requirement for the product. At an early stage we recognised that the network access was one of the harder parts of the implementation and assigned one person the development of this part as a full time occupation. This was done at the beginning of the first development phase even though this functionality was not required until the third delivery.

The problem that we ran into was that we had to be able to access certain low level functions of the hardware that we were working against. These functions were necessary if we were to be able to access the Bluetooth connection in the telephones. Since the operating system works over a large number of devices, the information of how to access these functions were present and the emulation software could handle the different access calls that was made to the operating system.

Since the development was done towards an embedded system we were not able to retrieve as much debugging data as we really needed. Since the emulation of the hardware worked fine it was a matter of tracking down the reason for not being able to establish a Bluetooth connection on the embedded system. Other problems occurred; the system was nice enough to fall back to a GPRS connection when Bluetooth did not work. This made it even harder since we always had to double check to make sure that the right type of connection had been established.

To test the network code we did not develop a prototype but instead we used the latest stable version of the application. What we needed for testing the network code were an application that allowed for sending and receiving information over a network connection as well as switching between the two different types of connections. Since this was exactly what our application did we thought that it would suffice to utilize it.

This is prototyping comes in; in retrospect it would have been better to develop a standalone application for the testing and evaluation of the network code. The use of the ordinary application as a base for the evaluation of a solution is not always the best idea, especially when the ordinary application is still under development.

The better approach would have been to develop a basic standalone application that mimicked the actual application without all the “fancy” stuff present (GUI, database etc.). This application would have been thrown away when a viable solution had been found and the network code that we needed could then be integrated into the real application.

This would have allowed the network programmer to evaluate his code without the influence and interference of the other people working on the application. This would also have allowed for an easier way to track down problems. The use of a more “controlled” environment would have allowed for fewer errors to distract the network programmer that instead had to compensate for problems that occurred in other parts of the application during the development.

Another problem that arose with the use of the “ordinary” application as a testing ground was that it limited the number of changes that were possible to apply to the software. What was most hindering was that since the embedded system did make it harder to retrieve debug information, we were in a need of better information “printouts” regarding what was happening in the background. This could only be achieved to a smaller extent without the risk of causing problem within unrelated parts of the program.

Since this project was a school project there was no actual cost in money. The cost came in the form of lost hours that were spent on trying to develop a solution to which no actual solution existed. Since the application only had to work on the present hardware it did not “break” the requirements when it was not fulfilled. What we lost in time was about 200 hours work for one person. This was the time that was spent on trying to figure out why the different solutions did not work as they should.

8.4. Example 4: Network Protocol Evaluation

This example is taken from the same project as the previously mentioned. During the development we had an issue that, even though not being a defect as such, still could have benefited from being developed using a prototype. The Instant Messaging application was supposed to be able to handle any number of different Instant Messaging protocols. The customer stated as a requirement that we should at least have support for two different protocols of our choice.

These protocols were handled by a plug-in system that allowed for the use of multiple protocols at the same time. For each of the protocols a parser to handle the syntax had to be implemented. The plug-in system worked against those parsers when they were processing the incoming and outgoing information. The problem we had was that we could not begin the development of the parsers before the plug-in system was finished.

The main problem was that we had to establish the interface between the parsers and the plug-in system. The interface was defined during the design phase and thereby we had a base to stand on when developing the prototype. This would have allowed us to work on the parsers in parallel to the development of the application and allowing us to incorporate them into the system when the plug-in system was working.

What we wanted to achieve here was to evaluate different protocols and how to best implement the parsers for each of them. To be able to do this at an earlier stage we would have had to built a small piece of software that emulated the networking functions of the application. This would have

allowed us to create different parsers that could be evaluated without having the finished product. Simply put, this solution would have allowed us to create parsers at an earlier stage.

It is hard to tell the actual gain of using prototypes for the purpose described. We did manage to implement and deliver two protocols as the requirements stated. It is possible thought that earlier start of their development would have allowed for an additional protocol parser to have been delivered with the final product.

In the project the parsers began development after the second half of the project time. During this time there were still problems with the application, problems that sometimes led to shorter standstills of the development. This was also coupled with the fact that a number of the different parts of the program had to be verified using the hardware e. g. it had to be tested on the P800 telephones. At the most we only had access to two telephones during the development and this in particular affected the implementation of the parsers since they had to be run on the hardware.

This caused small delays throughout the development of the parsers and therefore hindered the progress. Since as explained earlier this was a student project there was no actual loss of money due to these delays. The cost instead comes in the form of man hours. Without actual facts only a small reasoning can be done about this.

An assumption could be made that the delays cost us one man-hour for each day. The development of the two protocol parsers took for one person a total of four weeks, two weeks for each of the protocols. This means that not using the suggested alternate solution cost us a total of 20 hours during the development.

Unfortunately the building of the software used for evaluation of the different parsers would also have cost us time. If it would have taken more than 20 man-hours is hard to tell without actually doing so. Aside from the cost in development time another benefit might have arisen that would have approved the cost of the additional work for the prototyping. Each time that the prototype developers had to use the hardware there was also a risk of creating a delay for another set of developers in the project.

9. Conclusions

This article has shown that there are many different prototyping techniques available. Those techniques range from low-fidelity prototypes like paper mock-ups of user interfaces to complete software development processes like rapid prototyping. Since there are so many different techniques available, prototyping can be used in any kind of project. However, not every prototyping technique might be applicable in any project.

As shown in this article, prototyping can prevent different defects and improve the quality of the produced software system. Of course prototyping can not prevent any kind of defect. Quality systems like CMMI and ISO 9000:2000 suggest the usage of prototyping where it is appropriate. Also, it is possible to combine prototyping and software metrics as shown in this article.

Even thus the benefits of prototyping seem to be obvious, people and organisations must be convinced to use this technique. We showed several approaches to communicate the benefits and advantages within an organisation to motivate the people to use prototyping.

Furthermore we presented different examples of projects we participated in. We showed how those projects would have been improved by applying prototyping techniques and we also presented a basic cost-benefit analysis for each example.

In summary it can be said prototyping is a valuable technique, which should be used in every project if possible. However, software development organisations have to keep in mind that relying on just a single defect prevention technique is not good either.

References

[1] D. Baumer, W. Bischofberger, H. Lichter, and H. Zullighoven. User interface prototyping-concepts, tools, and experiences. In *Proceedings of the 18th International Conference on Software Engineering*, pages 532–541. IEEE Computer Society Press, 1996.

[2] B. Bergman and B. Klefsjö. *Quality: from Customer Needs to Customer Satisfaction*. Studentlitteratur, Lund, 2003.

[3] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[4] J. Bosch. *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, London, 2000.

[5] F. P. Brooks. *The Mythical Man-Month*. Addison Wesley Longman Inc., New York, 15th edition, 2001.

[6] T. Connelly. *Database Systems*. Addison-Wesley Limited, 1999.

[7] L. L. Constantine. On criteria for module interfaces. *IEEE Trans SE*, 16(12):1440, December 1990.

[8] T. DeMarco and T. Lister. *Peopleware: productive projects and teams*. Dorset House Publishing Co., New York, 2nd edition, 1999.

[9] C. Floyd. *A systematic look at prototyping*, pages 1–18. Springer-Verlag, Berlin, 1984.

[10] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Software Engineering*, 7:510–518, 1981.

[11] J. Highsmith. *Agile software development ecosystems*. Addison-Wesley, Boston, 2002.

[12] W. S. Humphrey. *Managing technical people: innovation, teamwork, and the software process*. Addison-Wesley, Boston, 6th edition, 1997.

[13] A. K. Jain and P. D. Ting. Software quality via rapid prototyping. In *Global Telecommunications Conference*, pages 642–646. IEEE, 1989.

[14] A. L. Jay. Quick & dirty. *Computerworld*, 26(50):109–112, 1992.

[15] J. Johnson, K. D. Boucher, K. Connors, and J. Robinson. Collaborating on project success. *Software Magazine*, February/March, 2001.

[16] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, Chichester, 2004.

[17] H. Lichter, M. Schneider-Hufschmidt, and H. Züllighoven. Prototyping in industrial software projects – bridging the gap between theory and practice. In *Proceedings of the 15th international conference on Software Engineering*, pages 221–229, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[18] British Standards Institute. The TickIT Guide: Using ISO 9001:2000 for software quality management system, January 2001 (5.0). http://www.tickit.org/.

[19] CMMI Product Team. Capability maturity model integration version 1.1 staged representation, 2002. CMU/SEI-2002-TR-029.

[20] S. L. Pfleeger. *Software engineering theory and practice*. Prentice Hall Inc., 2nd edition, 2001.

[21] J. Preece, H. Sharp, and Y. Rogers. *Interaction Design beyond human computer interaction*. John Wiley & Sons Inc., 2002.

[22] M. Rettig. Prototyping for tiny fingers. *Communications of the ACM*, 37(4):21–27, April 1994.

[23] I. Sommerville. *Software Engineering*. Pearson, Boston, 7th edition, 2004.

[24] M. Thompson and N. Wishbow. Prototyping: tools and techniques: improving software and documentation quality through rapid prototyping. In *Proceedings of the 10th annual international conference on Systems documentation*, pages 191–199, New York, NY, USA, 1992. ACM Press.

[25] D. R. Wallace and R. U. Fujii. Software verification and validation: an overview. *Software*, 6(3):10–17, May 1989.