Project 2
Software for use

Group 4
Björn Lindelöf
Fabian Bergström
Mattias Gloor
Paul Strömkvist
Index

Introduction 4

Description of the project 4
Description of the method 4
Rules and principles 5

In practice 6

Timeline 6
Collaborative Requirements Dialog 7
Domain Modeling 7

Task Modeling 8
User Role Model 8
Essential Use Cases 8
Joint Essential Modeling 9
Framing session 9
Modeling sessions 9
Review session 10

Interface Modeling 11
Interface Content Model 11
Context Navigation Map 11

Implementation Modeling 12
Prototypes and Prototyping 12
Creation of the Implementation Model 13
Introduction

Description of the project
We are tasked with planning a project for implementing an administrative software package. The application shall support organizations in planning their resources, keeping track of their finances, customers, suppliers, and product life cycle. The system should be general enough so that most companies and organizations can use it, regardless of size or line of business.

We will develop the system using a usage-centered approach, although ensuring usability in such a general product can be hard. The approach means users will be involved at key points in the process to help us understand the problem and to evaluate the interface. Such a general system also needs to cater to several different clients, and maybe none can be included in the requirements phase since we are developing not for one in particular. We will nevertheless need to meet with several user representatives having all kinds of jobs, considering the broad spectrum of work our product will have to support.

There are several other project groups that all got the same assignment, but all groups use different books, describing different methods. The goal of the assignment is to make us see our selves how all methods got there advantages and disadvantages compared to other methods.

By reading the reports from the other groups we get a different angle on how a task like this can be solved. This will apart from writing this report be a central part of the learning experience. After the hand in of this report a seminar will take place where we all discuss each others solutions and get an even greater insight in these differences.

Description of the method
According to the book usage-centered design is not really a method, it’s more a collection of coordinated activities that contribute to software usability. Through this process designers can try to understand better the actual usage of the software they are going to create.

Usage-centered design is founded on a set of basic guidelines that help designers make reasonable decisions and lead them to create highly usable systems. These guidelines contribute to the goals of designing systems that are: learnable, rememberable, efficient, reliable and satisfying to use.
They include usability rules and design principles.

**Rules and principles**

There are five rules to communicate and six design principles. All the process is based on the respect of those rules and principles. A designer should always keep them in mind and try to apply them literally. The following list is a guide about how to communicate effectively with the users, because for them the UI is the system itself, they don't know what there is hidden behind the actual interface.

Usability rules are: Access, efficacy, progression, support and context.

User interface design principles are: Structure, simplicity, visibility, feedback, tolerance and reuse.

There's no need to pursue all the activities in the book, and there is no predefined order. The book suggests that we make as much as possible in parallel and divide tasks in groups, as shown in Appendix A.

Usage-centered design is a streamlined and iterative process that can be scaled to suit projects of varying size and scope. Its activities can be flexibly rearranged to suit various objectives or constraints. The only three activities that have a predefined order are Domain Modeling, Collaborative Requirement Dialog and Task Modeling, that have to be performed at the beginning of the process, also shown in Appendix A.

Another very important thing to assess is that this methodology focus more on usage than on the users themselves. But even if it is usage-centered also the users play a crucial role in the definition of requirements in order to develop usable software. End users are the primary source of information to guide usage-centered design and their involvement are marked with a star in the activity model above.

Use cases are very important as the creation process is totally based on them. Each use case describes an interaction that is complete with focus on the goal or task the user aims to achieve.
In practice

Timeline
The first step of the project is to get a grip of the situation in which the system will be deployed called Domain Modeling. This step take place at the same time as the Collaborative Requirements Dialog where we decide on the requirements of the software. As our general software will have to span several business domains this step will consume more time than a software for just a single domain. We estimate the time for these steps to be a couple of weeks for a group of ten people investigating at least five business domains.

From here a lot of work can be done in parallel. Work on the Task Modeling, Object Structure design, Operational Contextualization, Standards and Style Definition, and Help System and Documentation will all start at this point. However, only the Task Modeling can be estimated with a timeframe since all of the other activities will proceed throughout the entire project. We estimate our task modeling to take quite a bit of time since there will be a multitude of different users, and creation of use cases are time consuming. This task easily takes a couple of months in total, but since this is an iterative process, we initially spend one month on creating use cases for the most essential parts.

Interface content modeling is more hands on than the task modeling and will probably take just one or two weeks for one months worth of use cases. The same thing goes for the Implementation modeling. After those two activities are completed it is time to involve the users in the process again. Depending on the result of the first usability inspection we can either move forward to the actual coding or iterate back to ensure an acceptable level of usability.

Using concentric construction and an object oriented language the coding could be quick as we can feed the developers with precise instructions on how things should work and fit together. The code should be done in more or less a month.

Finally the second user inspection is conducted and the result will give us the answer if the product is ready for deployment or if additional work needs to be done. If that's the case we iterate back.
Collaborative Requirements Dialog

Defining requirements is necessarily a dialog. Users know about the work, developers know about the capabilities of applications, and management knows about the business goals that need to be met. The dialogue is a process of mutual exploration and negotiation of the capabilities of the system.

One of the first things that have to be done in the project is having a meeting together with user representatives in order to define the requirements for the system. Since a general software package such as this doesn’t have any specific customers, none can be included in the meeting. However, domain experts could probably help with general business goals in the domain. Maybe prospective customers could be involved as well to analyze some real business requirements.

In this meeting, questions about what the primary purpose for the system is should be discussed and brainstormed. Investigating closer the work of users and what is needed for the system is also needed. There are four categories of information that are sought when defining requirements: function, form, criteria and constraints. Functions are the capabilities of the system, form is the realization and appearance of functions, criteria are desired system properties, constraints are limitations on possible or acceptable solutions.

The meeting should produce something called essential requirements, which means that the objectives, intentions, and goals of things are the important matters, not actions.

As much information as possible should be gathered, unwanted information can always be ignored later and some things may be surprisingly cheap to do. After the meeting, there has to be a time at which no new requirements are added until the next iteration. If requirements are not controlled, the system could suffer from feature creep and bloat.

Domain Modeling

It’s a representation of all the concepts and constructs that are interrelated in the application domain. They can be expressed in two different forms: or as an entity relationship model, or as domain class model (more common nowadays). Basically its purpose is to originate the vocabulary that will be used in the system and its operations or it can also been as a conceptual model that describe all the different entities (classes, methods, etc) involved into the system and the relationships that exists between those entities. Once it has been created it can be easily reused for further applications.
With the UML method the domain modeling is represented by class diagram, a static structure diagram delineating the whole structure of a system, their classes and all the different relationships between the classes.

Being this project written for completely different domains like hospitals, universities, industries, etc. and also the final users will come from different backgrounds there will be several Domain models all of them described with a different use case. And it will not be easy try to satisfy all the personal exigences so the purpose for such project would be to be as general as possible. And that is the opposite definition of an user-centered design.

Task Modeling
The first thing that should be modeled once the essential requirements for the system are available is the users' tasks. Task models describe the developers' understanding of the users' work, and can be inspected together with users to reach consensus about the work.

User Role Model
In order to better understand what work tasks have to be modeled, an understanding of what kinds of users the system will have is needed. The usage centered design process describes a model of users called the user role model. A user role model can be more or less formal, and describes the different roles users have in the system by stating their needs, interests, expectations, behaviors and goals in the system or in relation to each-other.

A structured role model is a list of profiles for each role, the profiles describe for example who can take the role, how proficient they are in the system, how their interaction patterns look, what information they work with in the system, usability criteria the role may have, and specific functions they need.

The relations between different user roles can be represented in a user role map. By creating the user role map, weaknesses in the model can be discovered, such as roles that are essentially identical. This information can be used to go back and refine the user role model.

Essential Use Cases
Once a user role model is developed, the tasks of the users can be described using something called essential use cases. Essential use cases are a more abstract form of traditional use cases where the focus is on the intention of the users instead of on the actions.

Essential use cases are related to each other in several ways. These relations can be described in a use case map. Use cases are shown on the map with notation for showing which use cases are
extensions of others, which use cases are composited of others and which use cases are related in a more general way.

**Joint Essential Modeling**

The user roles and use cases should be defined together with users in an activity called Joint Essential Modeling (JEM). JEM is carried out in a series of sessions: a framing session, modeling sessions, and a review session. Before the framing session, as much information as possible should be collected by the developers by reading about the domain and looking at competing products.

**Framing session**

In the framing session, the framework for the subsequent modeling sessions will be established. Deliverables from the framing session are: a draft statement of essential purpose of the system, a preliminary list of candidate user roles, and a list of participants for the subsequent modeling sessions.

First, JEM is explained to the users along with the purpose of the framing session. Together with users, a draft for the essential purpose of the system is then created.

Next, a candidate list of user roles is brainstormed together with the users and how many of the candidate roles are represented among the users in the meeting is checked. Unrepresented roles are noted so that they can be represented in subsequent sessions, suggestions for users representing these roles could be noted as well.

If it is discovered during the framing session that data is missing, or questions are encountered that cannot be answered, the problems are noted down so that they can be dealt with before the next session.

**Modeling sessions**

Now that a draft essential purpose and candidate user roles are done, they can be used to start modeling. In order to keep meetings focused there can be two separate meetings for modeling roles and tasks.

The role modeling session will deliver: a final statement of essential purpose, a user role model, and a user role map. Since there will probably be quite a lot of user roles, relating them in a user role map can help make sure their work is supported properly and also identify redundant roles so that the role model can be refined.

At the beginning of the meeting, the JEM method is reviewed again to inform users what we are doing and why. User role models will also be explained so that everyone are on the same, or at least a similar, page.
The candidate list of user roles from the framing session is reviewed to identify similar roles, specializing roles and composited roles to create a user role map. The meeting is then summed up and a task modeling session is scheduled with all participants again.

The task modeling session also starts off with a review of JEM to remind everyone. The task modeling gives the following deliverables: a list of use cases with essential purpose identified, essential use case narratives, a use case map, and a list of focal use cases. The use case map is optional in the usage centered approach, but in a system as complicated as this it should prove useful when later reviewing the task model and subsequently when organizing the interface.

Hopefully, the users have little problem thinking abstractly about their intentions when working. If they do not, they can explain their work more concretely and then that can be abstracted and generalized into essential use case narratives jointly by developers and users.

When the different use cases are described they can be related according to classification, composition, and affinity in a use case map.

Before the session is done, the focal use cases should be identified. The focal use cases are the ones deemed most central, important, or representative use cases of the system.

If new information arises that mandates a change in the user role model, it can be updated accordingly in this meeting. The task modeling session is then wrapped up with a summary of the meeting.

If there are a lot of different user roles then maybe task modeling can be done in parallel with different groups of users in parallel meetings.

**Review session**

After the modeling session, a review session is needed for auditing the models and allocating use cases to different iterations or versions of the application.

When auditing, the models are cross checked to make sure that every user role is supported by some use case and that every use case is supporting some user role. Redundancies in the models should be found and eliminated. The models should be constantly refined and simplified.

The session then moves on to allocating different use cases to different iterations of development or different versions of the software. It should also be decided if anything can be omitted completely from the software. The expected frequency of each use case has to be estimated before prioritizing them, as well as business importance and the difficulty implementing them.
**Interface Modeling**

When the task modeling is done, it can be fed forward to base the interface on. The interface can be modeled without the help of users, users will be involved later when inspecting.

**Interface Content Model**

The interface content model describes the different interaction spaces that users need to perform their tasks. Each interaction space describes the information users need for carrying out a use case, as well as the different tools and operations they need to use on this information. The content model may be best described by using papers, pencils, and sticky notes, as when they are later reviewed by users, users do not mistake them for being finished. Paper prototypes are also more abstract and therefore more robust to changes in requirements later.

The developers begin by assuming all use cases need an interaction space of their own. So one by one, use cases should be gone through and interaction spaces for them modeled. First, the kind of information the user needs available in each use case is identified, and notes representing them are sticked on a paper representing the space. Names for the different contents are written on their notes. The tools and operations users need for the use case to operate on the available information is then added to the space and names are written on the operation notes as well.

The model is refined, simplified, and abstracted. Some interaction spaces may end up being quite similar because their use cases are closely related, these spaces can sometimes be combined and the use case map can help us foresee this.

**Context Navigation Map**

The different interaction spaces in the interface content model have to, in the final application, be interconnected somehow. This interconnection of interaction spaces is modeled in the context navigation map. The map represents how one can move between different interaction spaces in the system.

A context navigation map can be more or less complete, but one for each use case is probably enough. On the map are different symbols representing different spaces, all with their names so that they can be identified. Arrows are drawn between the spaces denoting how users can move between them by different actions.

The context navigation map can help to identify overly complex chains of context switches for users, as well as patterns that will emerge that can maybe guide the developers in further refining the content model and combine or divide some spaces to make the map simpler.
**Implementation Modeling**

An implementation model shows how different parts of a system fits together. It is from this model that the real system then is built. The implementation model is created from an active and high fidelity prototype.

**Prototypes and Prototyping**

There are six kinds of prototypes to consider during development.

*Active Prototypes*

An active prototype has some interaction with the user, but not necessarily full functionality. It can be functionalities like a drop down menu that actually drops down, pop ups or a whole part of the software that actually works with full functionality.

*Passive Prototypes*

Passive prototypes are just dead pictures. It can be anything between a few lines on a napkin, or a very precise and high resolution computer generated picture like a screenshot from an Active Prototype.

*High Fidelity Prototypes*

High fidelity prototypes very closely resembles the final products.

*Low Fidelity Prototypes*

Only vaguely resembles the final product.

*Horizontal Prototypes*

A thin layer of the product showing just an interface and some pop ups and menus. Expressive appearance and silent behavior.

*Vertical Prototypes*

This prototype can be called use-case-centered since it got just some functionality implemented. Basically, it’s an use case, with enough implemented functionality and design for it to work.

Deep-and-wide is a hybrid between horizontal and vertical. It got a full interface, with some use cases implemented. Good for demonstration and presentation.

In developing a general administrative tool, all these types are needed, but not in every step. In the beginning of the project there’s little need for the active prototypes. The first step is to produce passive prototypes of the overall look of the software, the main window. These are low fidelity, and evolves to high fidelity through development of all modules of the system. There might be a certain need for a high fidelity prototype as well, since these are better fitted for progress reports, but the lo-fi should be the main focus.
When the project reaches the point where it’s time for some presentation and advertisement, there’s great need for a deep-and-wide prototype. This doesn’t need a lot of functionality at first, potential customer just has to get a grip of the overall layout of the system. How is different modules displayed? How is the interface adapted for different purposes and modules?

**Creation of the Implementation Model**

It’s easier to use an implementation model than a prototype when programming software. The implementation model more closely resembles the final product. To complete the implementation model there’s three general classes of issues that need to be resolved.

**Contexts**

It’s about when to show what. Not all windows or dialogues need OK-buttons or a minimize button. Also, a pop-up message doesn’t need a menu or status bar.

**Components**

Functions closely related can be grouped together, but in the first steps each function is represented in a separate widget that later are merged into what can be called components.

**Composition**

Components of the user interface that commonly is used together or associated to each other should be grouped together. This step is done no matter the screen size of the system, but if the interface gets to large, then the optimization is done to make it fit.

**Designing the UI**

Using the content model the implementation model is created, which specifies the layout of the UI and defines the interaction between the system and the user. Every abstract interaction space will be converted into an interaction context (screen, window, dialogue box…) and every abstract component will be converted into a visual component (button, text box, list…).

There are a number of rules that should be followed when designing a usable UI. To list them all is beyond the scope of this report but some of the most central rules/guidelines for designing the UI are the following.

The visual components should be logically arranged/grouped according to a logical flow through each interaction context consistent with the most frequent use cases (task), at expense of the usability of less frequent use cases if needed. Grouping should be logical but visual components should also be grouped according to which are needed to complete a specific task.
Another important concept is visual object persistence. It should always be as clear as possible to the user as to what happens with the UI objects. Users think of objects as persistent and ask themselves “where did it go” if it suddenly disappears from the visual UI. E.g. minimizing a window by animating the process is better than just making it disappear and then spawn a new object at another place.

The interface should aim for being guessable. That is, the users best guess on what something will do is probably right. Such interfaces are called Instructive interfaces. This design goal is often reached by building on themes or variations already familiar with the user. To support learning by trial and error, perfect reversibility is needed (Ctrl-Z or Cancel button for everything) and consistency in behavior is important as well.

Through its application of abstract models and emphasis on the fundamental goals of the users, user-centered design challenges the designer do devise creative solutions to users interface problems. Sometimes standard visual components are less than optimal and custom visual components are needed. To come up with suitable custom visual components or a completely new layout a process for innovation is needed.

**Iterative innovation** starts with a period where you let ideas flow followed by a period of constrain where you test your ideas against the harsh realities of the real world. By alternating between positive thinking and negative thinking throughout the project time, ending with a polish period, you will hopefully end up with something that is both innovative and usable.

**Both-and design**
A common barrier to creative ideas is the tendency to think in terms of irresolvable opposites. At first glance two ideas may seem to conflict but if you really put your belief in that all ideas can be integrated or combined effectively you will force yourself into thinking creatively.

**Usability Inspection**
As shown in Appendix A the usability inspection is carried out two times in the usage-centered design activity model. This could also be done iteratively several time until the product reach a satisfactory usability level, or as far as the budget allows.

There are several different ways for conducting an usability inspection. The most common are the following:
Heuristic evaluation
It is done by following a set of general rules or heuristics for good users interface design, it is arguably the best known usability inspection technique.

* First pass: the user interface or design is reviewed in order to understand its overall structure and the general flow of interaction.

* Second pass: the evaluator examines the individual interaction context and their contents, assessing them against the criteria of the chosen heuristic.

Cognitive walk through
This inspection technique is based on a cognitive theory of skill acquisition. The purpose is to identify usability problems by stepping through tasks scenarios with before defined success criteria. It’s important to have them written down and to achieve them completely, not only partially. It would be better to have a counterbalanced order, so that the learnability of the new system is not unbiased for some tasks. Every task should has the same chances to be successfully accomplished, no matter if it’s easier or more complex. It will also give us some information about the barriers the testers find in learning using the software.

Pluralistic Usability Walk through
End users, developers and usability experts all together and all expected to play the role of final users.

   Everyone goes through a task scenario like in the previous method but this time without discussing between themselves. They are just allowed to write down a kind of storyboard their personal reflections. Developers should act like users as much as possible, trying to be critics in what they have done. The final discussion between all the participants is the constructive part of this inspection method, everyone expose its thoughts and ideas, and again developers should not try to protect their job, or feel attacked by the others, instead take every suggestion or critique as a motivation to improve their work.

Collaborative Usability Inspection
By including users and usability specialist in inspection and repeatedly applying the principles of usability, software developers build skills and a steady growing found of knowledge about good interface design. It’s a mix between heuristic evaluation and pluralistic usability walk through, but it is designed to be faster and more efficient. Once again, identifying defects is not a sign of failure but a sign of success.

   The inspection team should work together towards the common goal of producing a better, and more usable, product.

Usability Test
Instead of an usability inspection an usability test can be done, for this an active prototype is required. Before conducting such a test it’s advisable to write down a test specification where the goals, research
questions, details about the task to accomplish, and the success criteria are specified. The participants characteristics should also be defined.

Once the test specification is ready it's time to conduct a pilot test to check if everything is okay.

The results of such testing will be numbers, often percentage, that can then be used to identify usability problems and give suggestions to the developers and designers. These numbers are also very useful for the management of the employing company.

There is a particularity for this kind of project, because in order to have a clear usability idea the usability test should be performed from at least one representative group from every different domain. Even the specific tasks could be different according to the core business of the costumers, even if some of them will be the same for every different working domain. It doesn't mean that it will take more time, if there are enough usability experts and labs it's possible to perform it in parallel, but if the resources are limited then the time consecrated to such activity could really increase. But seen the size of this kind of project it’s possible to assume that there will be enough resources.

The most suitable approach for this kind of project is the usability test. It's possible to get more feedback and a clearer idea of the users real needs and their interaction with the software. Maybe is not the cheapest or the less time consuming, but it’s complete and cover every aspect.

Operational Contextualization

Operational Modeling
It’s important to take in consideration the context in which the system is used. When doing operational modeling the focus point is on how the system operates in different situations.

In this phase developers has to visit users and monitor their usage of the system. Imagine a bank employee that transfers money for a customer. If an error occurs this will alert the customer as well, this might stress the banker to work faster and thus make more mistakes resulting in more inconveniences for the customer. A typical example of a bad situation for sound messages.

Environment Profile
You can’t make an usage centered design if you don’t know the environment the system will be used in. Therefor you need to profile the environment and look for aspects that will affect the design of the system.

Device Constraints Profile is about the physical device and its interfaces. There might be limitations on both input and output. Sounds might not be tolerated, or it’s not possible to make an keyboard work
or fit at scene. It can also be that a keyboard is required, but it can’t be a regular keyboard. In hot environments special material might be necessary, and the keys got to be larger so users can use it with protective gloves on their hands.

Operational Risk Profile is about consequences of errors. What happens if the input is invalid? There is a system failure? Or lock up?

Environmental Adaption
This part of a project is more about the environment, and less about the context, even if these two aren’t completely separable. Depending on environment different types of systems has to be built considering reliability, mobility, quality, durability, etcetera.

Software developers need to consider colors, lighting, fonts, sounds and so on. Software is dependent upon hardware as well, for example there might not be a separate keyboard or mouse available. If the environment is very light, maybe a subject of direct sunlight, the colors can’t be to light. In the other direction, inside a mine or other dark place, it’s not comfortable or healthy to look at a light screen too long.

Standards and Style Definition
To ensure a coherent look and feel of the software, standards and style guides are needed. But by setting standards at the beginning of the project the risk is to possess standards that simply don’t work or are hard to enforce since they don't stem from any sort of consensus in the organization. Standards should instead always derive from experience and evolve iteratively.

Help Systems and Documentation
The best way to write the documentation is for the technician and the document writer to work together. The technician providing the writer with expertise help, and then reviewing the final document making sure the technical parts are correct.

Help Cases
Help cases are a subversion of use cases. They are use cases that describe how users will look for help while using the system. Following are examples of help cases.

Seeking Identification: Holding mouse over link or button and the system tells what it’s used for. Can be implemented with a tool tip message or shown in the status bar.
Seeking Instruction: Help with performing a task. A step-by-step guide through a task.
Could be a wizard or algorithm for completing the task.

Help cases are a very good way to organize the help system. Each help cases is represented by one post in the help system. This way the same task can be answered in more than one post, making it easier for the user to find the right post. If the user want to insert a special character, then the help should be found both through both *Insert* and *Special character* in the help system.

Access and Presentation Techniques
Another important aspect to think about is where to present help messages. Where does users look for help?

The best place to display short messages, like in the Seeking Identification case, are at the cursor or pointer. This is where the user most likely got their attention. For more serious messages like errors or warnings it might be better to use pop ups to alert the user.

Access to help
All ways of activating and accessing help have their advantages, which is the best depends on the context and the kind of system in use. Help can be system or user activated, static or automatic. What type is used can be predetermined, or switch depending on different modes. More advanced user probably wants to decide what help they want, and not want.

Help can be invoked by passing a button, pausing over a button, or by pressing a hot-key identifying all or selected buttons.

Wizards
Wizards are typically time consuming and inefficient. But they are very good in helping users with complicated and complex tasks, that’s not frequently executed.

Wizards can also be implemented as tutors. Here the actual interface is used and the help system displays tool tips at the next button or function to be used, with a description of what to do. Or all buttons and functions can get tool tips, that’s numbered in sequential order.

Object Structure Design
An object oriented languages is well suited for supporting a big project like this. Object orientation lends itself to code reuse and good separation between the different parts of the software. This will be extra important for this project as the final software will have to be modular and many parts are similar. Objects should be grouped according to the Model View Controller model. This grouping results in three types of Object stereotypes; (View) Interface objects, (Model) entity/domain objects, Control objects
(Controller). In practice partitioning of the use cases into object stereotypes should be done in the order of (1) Interface objects, entity objects (anything dealing with information storage or handling) and lastly control objects.

Object orientation enables separation of the interface and the supporting internals using control objects as the bridge in between. An advantage of this approach is the separation of data from presentation and that it limits the impact of a code change in one part of the program to the other parts. This also allows for code that is easy to maintain and reuse through classes/objects is an effective route to achieve consistency in the user interface.

**Accelerated/Iterative development**

**Concentric Construction**

Concentric construction is all about letting the use cases form the development of the software. The relative priority of use cases should have been established (e.g. through Joint Essential Modeling) earlier in the development process. The development effort should be concentrated at the most important use cases and each iteration of the software should deliver another implementation of a use case. Focusing on implementing use cases instead of specific features helps keeping focus on the user and prevents implementation of features with no clear purpose to the end user.

This approach to development also makes it easier to make different versions of the same software. Iterative development processes almost forces creation of code that are modular, as use cases are implemented separately.

**Architectural Iteration**

As development continues and more code is written to support more use cases the complexity of the code base grows. At some point the assumptions that was made in the beginning of the development process on what was the best technical solution to support the use cases will have become obsolete. It then becomes harder to implement new use cases in a technically sound way, or even impossible to implement use cases in a desirable way as a technically obsolete platform put obstacles in the way. A simple solution to the problem is to start over with a new architecture even if it would be better to avoid this bump in the road altogether.

The solution for avoiding the complexity problem is an iterative approach not only for implementing use cases, but for the supporting architecture as well, called Architectural Iteration. This approach take a look at the basic architecture at each iteration and try to foresee the the needs of the use cases in the next iteration and make the appropriate changes to the supporting architecture. Such changes could be
a change in some internal messaging format, a partitioning of the code into more suitable objects or maybe a change in the storage system.

Done correctly, this approach will keep the architecture up to date and will prevent or at least prolong the need for a major technical overhaul every couple of years. This way the maintenance of the supporting code is spread out over time which often is a good thing as development on code that actually serves the users needs can be done continuously as well.
Conclusions

Usage-Centered System Design in this project
Usage-centered design delivers software that are usable in the targeted domain. However good this idea is, it’s very hard, or rather time consuming, to follow this approach in development of too general software.

This is mainly because there’s a lot of field studies that have to be done, as well as user involvement. Including users always takes extra time since it’s rare that users know exactly what they want from the start, and developers has to do extra work to make users understand what to do in each step. The advantage of having a common technical platform to share, to support different domains in the general software, decreases as the relative amount of work that has to be done shifts more towards tailored solutions (usage-centered) that won’t be reusable throughout the software.

The large amount of potential users are both for good and for bad. The bad part is of course the increased complexity of having to consider users from many different domains. The good is that at least some part of those users will use the software in a similar enough situation so that we can reuse parts of our software in different contexts. This will bring cost savings as well as added efficiency in further development. These cost savings as well as the broader appeal of our software will most likely expand our market so that we will make more money. This money could then be spent on increasing development effort of the software, and thus decrease the negative effects of having to cater to so many different kinds of users. The big question is if the economic advantages of making a software for a larger market will be enough to outweigh the disadvantages of making the software general.
Appendix A

FIGURE 2-2 Usage-centered design activity model.
FIGURE 2-1 Essential models, logical relationships.