AALBORG UNIVERSITET

Det Teknisk-Naturvidenskabelige Basisår
Natuvidenskablig - Software
Strandvejen 12-14
Telefon 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Title:
Mini-Krak

Theme:
Network and algorithms

Project period:
P2, spring semester 2008

Project group:
B205

Participants:

Frederik Højlund

Jens-Kristian Nielsen

Søren Møller Larsen

Jesper Kjeldgaard

Kenneth Madsen

Peter Lindegaard Hansen

Supervisors:
Gitte Tjørnehøj
Søren Kerndrup

Copies of report: 10

Report page count: 75

Appendices: 1

Report finished: 26. May 2008

Synopsis:

The purpose of this project is to develop a pathfinding application. With Krak.dk as rolemodel, our program should be able to navigate inside the university building. Furthermore we will try out an agile development method and compare different methods, both linear and agile. The project and our method experiences are described in this report. As a part of the application development and research we report on the theory of graphs, and go in depth with different path finding algorithms. We made a product that enables the user to find the shortest route on a map. We also made a map editor that makes it easy to maintain system data. Our process using XP went well even though we did not utilize it completely.

Contents

Preface

This report is the result of B205's P2 project at Aalborg University in 2008 with "Mini-krak" as the main topic.

Besides describing the theory about path finding algorithms and system development methods, we have developed a working prototype of a pathfinder program to be used inside buildings. The program and its source can be found on the attached CD-ROM. All images in the report can also be found in color on this CD-ROM.

Citation references are located in the reference section at the end of the report (on page 74), just before the appendix. Words written in *italic* is described further in the terminology section (on page 73).

As a part of our study, Aalborg University have in this semester provided a course on the subject "Discrete Mathematics", which we have used large parts of in this project. Other subjects used in this project, such as Databases, UML, System Development Methods are self-study by the group members.
This report is written with the assumption that the reader has a basic knowledge of computer software and computer programming.

Approach

The report consists of four main parts.

1. Problem Analysis
2. Theory
3. System Development
4. Reflection

We start with a Problem Analysis where we find the requirements for this project. First by analyzing the different stakeholders and then formulating the problem that define the contents for the rest of the project. The Theory chapter is about the theories and methodologies we need to learn about in order to develop our product. The System Development chapter is about how we applied the theories from the previous chapter when developing our product. It also contains a product description where we introduce our product with all its features. In the Reflection chapter we reflect on our process and conclude our project by looking at it from different perspectives.

Chapter

1

Problem Analysis

1.1 Introduction

Nowadays it is getting more commonly known how to navigate with use of different pathfinders, like Krak, GPS systems and Google Maps. With this perception it is harder to be excused for being late, because of lack of proper navigation. The problem we see in this is that modern pathfinding tools only navigate on maps seen from a sky-view, meaning that buildings are visualized only by their outline and not their actual content. For most people that is enough to fulfill their needs, but for others that is not always enough, because the pathfinders are limited only to navigate to the outline of the destination. The result of this is that you are only led to the front door of a possibly complex building, but you may need to get to a specific room. This is where we see our product come in handy. We see our product as support for floor plans at the the destination. It should be a simple application that allows the user to select a destination and then find and display the shortest route to that location. All this is inspired by the complexity of the Basis department at AAU, where there seems to be a repeating problem every semester when new students arrive. We have made the university our fictional buyer to create a more realistic development process. Below is our initiating problem that we wish to analyze before writing our problem definition.

1.2 Initiating Problem

- How can we ease navigation in the Basis department at AAU using software?

1.3 Stakeholder Analysis

Here we describe the possible stakeholders for a path finding system that operates inside buildings. The stakeholder analysis is based on assumptions of the different stakeholders interests who could be involved in a project like this.

To make an optimal stakeholder analysis we have to find out which stakeholders would have influence on the project and which of these are most relevant. There are many possible stakeholders that could affect this project. Such could be other companies that produce pathfinding applications. We have chosen the following three stakeholders: The buyer, users of the system and the developers. Furthermore these three sections are divided into two parts; one part which shows the general stakeholder of such a system and an other part which is realistic suggestion of a stakeholder of our project. The buyer is chosen because he is the one who sets the requirements. The developer is the one who weight these demands and replies how the outcome should be and the users are the people that end up using the product. They were all chosen because of their potential interests in the product.

1.3.1 The Buyer

1.3.1.1 In General

Usability is important when a product is aimed at helping people. Buyers would be interested in a simple and easy-to-use product. It is essential that the system data is kept up-to-date, otherwise the system will be functional but obsolete, making the overall system of no value to the buyer.

A product is often ordered with many requirements and the developer might interpret these requirements in another way than the buyer. These misunderstandings can cost a lot of time and money. Because of this, the buyer will always be really interested in following the development and seeing frequent releases in order to correct misunderstanding as early as possible.

The buyer is interested in a product of high quality that solves the problem in the best possible way. They would not be satisfied with a non working or partially working product. They will expect the developer to be able to add features in case the product is a success. In short terms the interests can be prioritized as following:

1. Good usability.
2. Interested in seeing frequent releases during development.
3. Features to keep the data up-to-date.
4. Quality of the result produced by the program.
5. Possibility of extra features.

1.3.1.2 The University

The university would be interested in purchasing this product to optimize time consumption of students, teachers, personal and visitors of the university. As the university in total have more than 207.388 square meters of indoor area. Which indeed make their buildings complex, just by sheer size.[15] The users have a varying technical knowledge and will therefore require a simple program. Use of rooms often change after a semester and this can cause the data to be out of date. Because there

are new students every semester, it would be practical to be able to edit the data easily for the system administrators. This could also make it possible for secretaries to update the data. If the product is a success they may want to include more features which can be integrated in the product. For example integration with existing timetables so students can find classrooms by clicking on their timetable. In short terms the interests can be prioritized as following:

1. Simple program with good usability.
2. Easy maintenance of data.
3. Possibility of extra features.

1.3.2 Users of the Product

1.3.2.1 In General

First of all we have to determine the type of users for this product.
Important things include experience of using the system. Some users are inexperienced whereas other users might have used a similar product before. If it is mostly first time users, then the program has to be very simple. There must be as few complex elements as possible in order to not confuse the user. The more experienced the users become using the product, the more advanced the program could get without sacrificing usability. Furthermore, when the product is about pathfinding, users would be interested in understanding the route. In short terms the interests can be prioritized as following:

1. Easy to use.
2. Ways to read and understand the route.
3. Feature development with integration in mind.

1.3.2.2 People at the University

Most users of the product are teachers and students. Students will need direction to find common rooms like toilets, classrooms and cafeterias. These can be very hard to find when the university has several buildings and floors. Certain areas are owned by other organizations making it even more complex.

Teachers will also look for toilets and classrooms, but they will also look for the different group rooms where the students are. When supervisors have meetings with students, they will need to find a specific room where the students are. This requires a certain amount of detail of the map in the program. Integrations between timetables and the program would also be preferable so students and teachers could see what classroom they need to be in. In short terms the interests can be prioritized as following:

1. Easy to use.
2. A lot of details regarding group rooms.
3. Feature development extra features.

1.3.3 Developing company

1.3.3.1 In General

The main interest of the developing company, in such a software project, is the income that the project will bring to the company. Software projects like these are the central business in a software developing company, thus quite essential. As the main interest is to make economic profit, the company wants to make a good contract with the buyer, which is either fair for both parties, or favorable for the company. The interest can be summarized as:

1. To develop the product for a customer, and gain profit.

1.3.3.2 B205

Since we have a tight schedule we will focus on developing as simple and as fast as possible. And meanwhile still aim for high quality, both in the product and the report, as we will be graded on the quality of our overall work. Another interest is however, to learn as much as possible from developing this project. Facing the challenge of developing a new product, we aim to train our coding skills and learn how to develop a larger project in a team effort. We are interested in learning about different agile system development methods and try to use it in practice.

1.3.4 Conclusion

All the stakeholders have one thing in common. They are all interested in simple product with good usability. This reduces the work needed to develop the program and increase the usability. When talking about the development phase the main priority is the cooperation between the developers and the buyer; allowing the buyer to comment and change the program early in the development. Because of the deadline given by the university the end result might not be satisfying for any first time user; however, the prototype will show the basics of the program. The final requirements which should be considered can be prioritized like this:

1. Simple program with good usability.
2. Allow the buyer to see frequent releases in order to retain/expand the deadline
3. Possibility of adding features

1.4 Problem Formulation

1.4.1 Problem Description

The problem in this project is that modern path finding systems that can only guide us to our destination building. But they can not help the user any further than the main door.

Throughout the stakeholder analysis we have discovered what our product should contain in order to be sold to the stakeholder. For selling our product it should meet these criteria. But how can we develop a product and at once be sure it fits the users needs? We will have to examine different development methods and the different pathfinding algorithms available. We have narrowed our problem down to a problem definition.

1.4.2 Problem Definition

How can we develop a system that satisfies our users needs?

The definition above covers a wide area of expertise. The four questions below narrows that area down to a more tangible size.

- Using discrete mathematics, which algorithms can be used in a path finding system that operates inside buildings?
- How do we develop a system using an agile system development method?
- How does agile development methods affect developers, customers and planning?
- How do we develop a system that enables the user to navigate inside buildings?

1.4.3 Problem Limitation

During the course of this project, we acknowledge that it is impossible to cover all the features of a pathfinder program. The main theme of this project is discrete mathematics. Even though usability is a high priority for the stakeholders, we have chosen not to focus on it in this report. We will limit the extent of the project later, when we have described the program with all the desired features.

Chapter

2

Theory

2.1 Graph Theory

2.1.1 Introduction to Graphs

In this section we explain the mathematical structures used to model pairwise relations between objects from a certain collection, called graphs and some of the theory and terms related to it.

Sources for the following are [23] and [17]. Graph theory contains many different terms. These terms are:

- Vertex
  A vertex is a point, that can be connected by zero or many edges.
- Edge
  An edge is a connection between two vertices. This connection can have a weight which for example determines the cost of traveling from the one vertex in the edge to the other vertex. A directed edge is an edge with a specific direction between the two vertices. The expression "Multiple edges" means that two or more edges are connecting the same two vertices. A loop is an edge that goes from a vertex to the exact same vertex.
- Graph
  A graph consists of vertices and edges. There exists many types of graphs:
  - Simple graph
    A simple graph consists only of vertices and undirected edges. It does not have multiple edges or loops.
  - Simple directed graph
    This graph is basically the same as the simple graph only with the difference that the edges are directed.
  - Multi graph
    This graph is basically the same as the simple graph only with the difference that multiple edges are allowed.
  - Directed multi graph
    A directed multi graph has multiple edges and at the same time the edges are directed.
  - Pseudo graph
    Pseudo graph consists of multiple edges and loops. The edges are undirected.
  - Mixed graph
    A mixed graph consists of both directed and undirected edges. Furthermore it has multiple edges and loops.
- Tree
  A tree is a term for a graph that does not have a circuit of any kind. Furthermore a tree does not have multiple paths between any two of its vertices. Trees can be used to show the structure of: Relations between parents and children (family tree), Organizations, File systems, Networks etc.
- Root
  Defining the root of a tree means that all vertices are directed from the root, which leads to the direction of the tree is from the root to the vertices.

- m-ary Trees
  A m-ary tree is a tree with no more than m children. A full m-ary tree has exactly m children per vertex. A tree with m = 2 is named a binary tree.

- Subgraph
  A subgraph is part of a graph using some of the vertices and edges from the original graph. The graph that contains the subgraph is called the supergraph.

2.1.1.1 Shortest-Path-Tree

In this project the Shortest-Path-Tree is an important section of graph theory[26]. The tree is a subgraph and actually a subtree because it fits the description of a tree above. The definition of a Shortest-Path-Tree is, in addition to being a tree, a tree constructed so that the distances between a root vertex and all other vertices are minimal. That implies that if you travel from the root to another vertex via the tree it is the shortest path. This tree can be very useful in Shortest-Path-Problems.

2.1.2 Shortest Path Algorithms

2.1.2.1 Introduction

In this section we describe the general theory about the shortest path problem and links real world problems to graph theory.

The problem of finding the fastest or shortest route from one place to another has a role in most peoples everyday living. We might not think about it when leaving home and driving to work, or when going shopping downtown. But these actions could be converted to a shortest path problem. Of course if time or fuel efficiency is of no importance then it does not matter. But with the tight schedule of many peoples lives, these things does matter. First step is to create a map of the area and measure distances, but if this should be really efficient we need computers to handle it. To simplify the map it will be converted into a graph. A graph can be used to represent for example a map. Edges can represent segments of road and vertices can represent locations of road forks. The only thing left to do is having the computer run a graph-search algorithm on the graph. To find the shortest path between to vertices in a weighted graph the algorithm needs to find a path where the sum of the weights of the edges in the path is minimized[25]. These algorithms have of course already been invented, and in the following section we will take a look at some important graph search algorithms: The Dijkstra algorithm, the Best-First-Search algorithm and at last link the two into the A* algorithm.

2.1.2.2 Dijkstra's Algorithm

In this section we will show you how Dijkstra's shortest path algorithm works.

Edsger Dijkstra (1930 - 2002) developed in 1959 a shortest-path algorithm. This algorithm is shown in pseudo-code below:

Procedure Dijkstra(G: weighted connected simple graph, with
    all weights positive)
{G has vertices a = v_0, v_1...., v_n = z and weights w(v_i,v_j)
    where w(v_i,v_j) = ∞ if w(v_i,v_j) is not an edge in G}
for i:= 1 **to** n
    L(v_i) := ∞
L(a) := 0
S := $\emptyset$
{the labels are now initialized so that the label of a is 0 and all
    other labels are ∞, and S is the empty set}
while z $\notin$ S
begin
    u := a vertex not in S with L(u) minimal
    S := S $\cup$ {u}
    for all vertices v not in S
        if L(u) + w(u,v) < L(v) **then** L(v) := L(u) + w(u,v)
    {this adds a vertex to S with minimal label and updates the
        labels of vertices not in S}
end {L(z) = length of a shortest path from a to z}

This pseudo-code is taken from the book "Discrete mathematics and its applications" written by Kenneth H. Rosen and the book is also the source[17] for this section.

For any weighted connected simple graph the algorithm will always find the length of the shortest path from a start-vertex to a destination-vertex. It works by first applying the length of all vertices to be infinite except for the start-vertex that will have the length 0. This is because the distance from the start-vertex to the start-vertex is 0. Then the closed list, S, is set to be empty. The closed list contains all the vertices that has been examined for the shortest path from the start-vertex to the vertex. This means that when the destination-vertex is in the closed list, then the algorithm has found the shortest path from the start-vertex to the destination-vertex. Thus, the algorithm are running a loop until the destination-vertex is in the closed list.
The loop starts with finding a vertex with the lowest length between the vertices that are not in the closed list and that vertex is set to be the current-vertex. Then the current vertex is added to the closed list. Thereafter the algorithm looks for adjacent vertices to the current-vertex. If it finds a vertex adjacent to the current-vertex, then it calculates the length to that vertex from the start-vertex over the current-vertex to the adjacent vertex. If the length is shorter than the current length of the adjacent vertex, then the length of the adjacent vertex is changed. By running the loop until the destination-vertex is in the closed list, then the shortest path is found from the start-vertex to the destination-vertex. Furthermore it does not only find the shortest path from the start-vertex to the destination-vertex, but it does also find the shortest path from the start-vertex to any of the vertices in the closed list. Which in fact is a tree that shows the length of shortest path from the start-vertex to any of the vertices in the closed list. See section 2.1.1.1 about

shortest path trees.

Example of Dijkstra's Algorithm

To show exactly how Dijkstra's algorithm works we have made a small graph for finding the length of the shortest path from v_0 to v_3 as shown in figure 2.1:

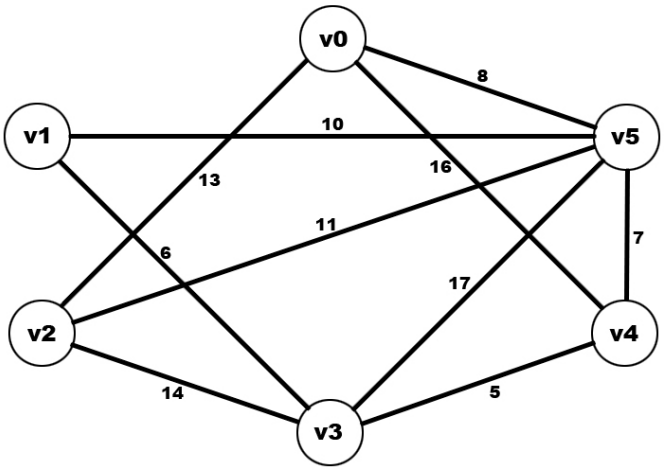

Figure 2.1: This image shows a weighted connected simple graph consisting of six vertices and ten edges with different weights.

The algorithm then apply the length to v_0 as 0 and the other lengths are set to ∞. Thereafter the closed list is set to be empty. The results of these operations is illustrated in figure 2.2:

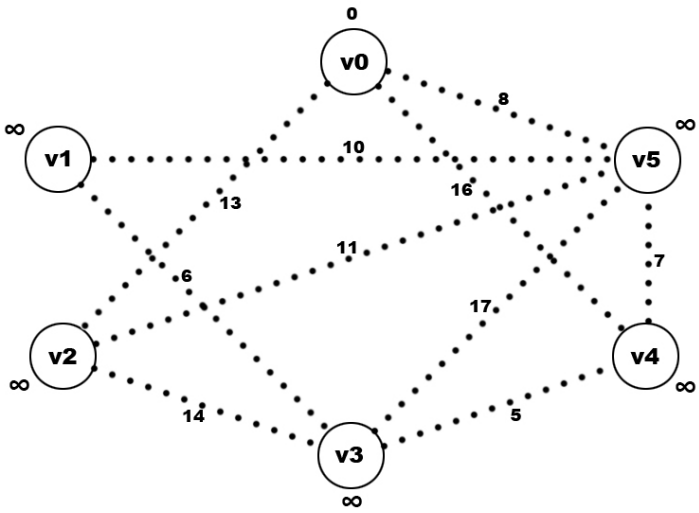

Figure 2.2: The result after the first few operations.

Then the algorithm goes through the first iteration of the while loop:

- u is set to be v_0.
- v_0 is added to S.
- The for-loop returns the following:
  - L(v_1) is still ∞ because L(v_0)+w(v_0,v_1) equals ∞, which is equal to L(v_1) so L(v_1) does not change.
  - L(v_2) is set to 13 because L(v_0)+w(v_0,v_2) equals 13, which is less than L(v_2).
  - L(v_3) is still ∞ because L(v_0)+w(v_0,v_3) equals ∞, which is equal to L(v_3) so L(v_3) does not change.
  - L(v_4) is set to 16 because L(v_0)+w(v_0,v_4) equals 16, which is less than L(v_4).
  - L(v_5) is set to 8 because L(v_0)+w(v_0,v_5) equals 8, which is less than L(v_5).

The results of the first iteration is illustrated in figure 2.3:

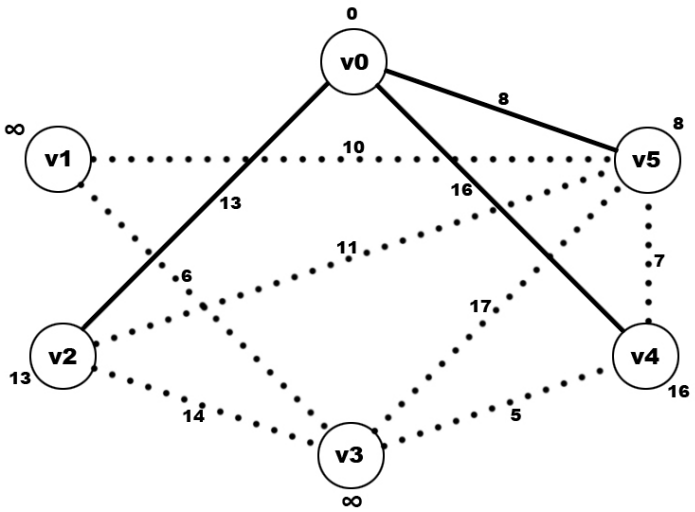

Figure 2.3: The result after the first iteration.

The second iteration:

- u is set to be v_5.
- v_5 is added to S, which now contains: v_0 and v_5.
- The for-loop returns the following:
  - L(v_1) is set to 18 because L(v_5)+w(v_5,v_1) equals 18, which is less than L(v_1).
  - L(v_2) is still 13 because L(v_5)+w(v_5,v_2) equals 19, which is more than L(v_2).
  - L(v_3) is set to 25 because L(v_5)+w(v_5,v_3) equals 25, which is less than L(v_3).

  - L(v_4) is set to 15 because L(v_5)+w(v_5,v_4) equals 15, which is less than L(v_4).

The third iteration:

- u is set to be v_2.
- v_2 is added to S, which now contains: v_0, v_5 and v_2.
- The for-loop returns the following:
  - L(v_1) is still 18 because L(v_2)+w(v_2,v_1) equals ∞, which is more than L(v_1).
  - L(v_3) is still 25 because L(v_2)+w(v_2,v_3) equals 27, which is more than L(v_3).
  - L(v_4) is still 15 because L(v_2)+w(v_2,v_4) equals ∞, which is more than L(v_4).

The fourth iteration:

- u is set to be v_4.
- v_4 is added to S, which now contains: v_0, v_5, v_2 and v_4.
- The for-loop returns the following:
  - L(v_1) is still 18 because L(v_4)+w(v_4,v_1) equals ∞, which is more than L(v_1).
  - L(v_3) is set to 20 because L(v_4)+w(v_4,v_3) equals 20, which is less than L(v_3).

The fifth iteration:

- u is set to be v_1.
- v_1 is added to S, which now contains: v_0, v_5, v_2, v_4 and v_1.
- The for-loop returns the following:
  - L(v_3) is still 20 because L(v_1)+w(v_1,v_3) equals 24, which is more than L(v_3).

At the sixth iteration v_3 is added to S which induces that there are no more vertices to check for. The length of the shortest path is therefore L(v_3) = 20. In the process of finding the length of the shortest path from v_0 to v_3, the algorithm has produced a shortest path tree which can be seen in figure 2.4 below:

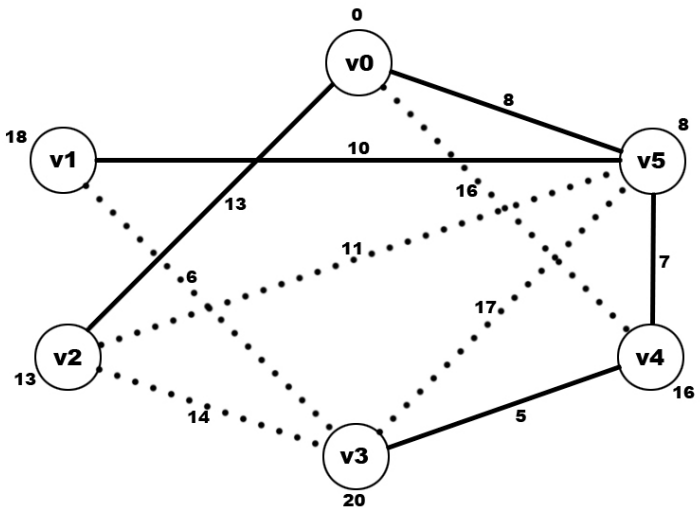

Figure 2.4: The shortest path tree the algorithm has created.

2.1.2.3 Best-First-Search Algorithm (BFS)

The source of the following is [13] and [9].
The Best-First-Search (BFS) algorithm works very similar to Dijkstra's algorithm, except that it uses an estimate instead of the cost of the path so far. The estimate is called an heuristic and is defined by the distance from a vertex to the destination. Different heuristics can be made but the simplest one will ignore all obstacles and find the distance as a straight line.
Instead of selecting the vertex closest to the start-vertex like Dijkstra does, the BFS selects the vertex closest to the destination-vertex of the ones adjecent to the start-vertex. This is both an advantage and a disadvantage. This algorithm runs much faster than the Dijkstra algorithm because it checks much fewer vertices, however the BFS algorithm is not guaranteed to find the shortest path possible. So why does BFS check fewer vertices than Dijkstra? In figure 2.5 the Dijkstra and the BFS algorithm are compared and the grey blocks are vertices that the algorithm checks. Both algorithms starts at the A-vertex and finds the shortest path to the B-vertex. It is clearly that the Dijkstra algorithm checks a lot more vertices than the BFS does. This is because Dijkstra searches "blindly" in all directions while BFS uses the heuristic to guide it towards the destination and causes it to use fewer vertex-checks. The vertices closer to the destination simply "weighs" more for the BFS algorithm causing it to search in that direction.

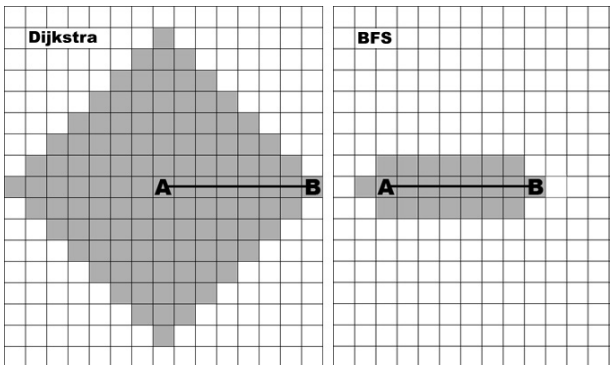

Figure 2.5: Dijkstra versus BFS

The example is a very simple one though, where the shortest path will be a straight line between the two vertices. The next example shows a situation where the BFS does not find the shortest path. In figure 2.6 the shortest path is not a straight line and the algorithm is required to "navigate" around the obstacle. The result is that while the BFS algorithm checks fewer vertices it fails to find the shortest path to vertex-B. Dijkstra again checks a lot of vertices, but unlike BFS it clearly finds the shortest route from A to B. This is the disadvantage of the BFS algorithm. It is greedy and ignores the cost of the path so far and keeps trying to head toward the destination even if it is a wrong path. This causes BFS to find a path, but not the shortest path possible.

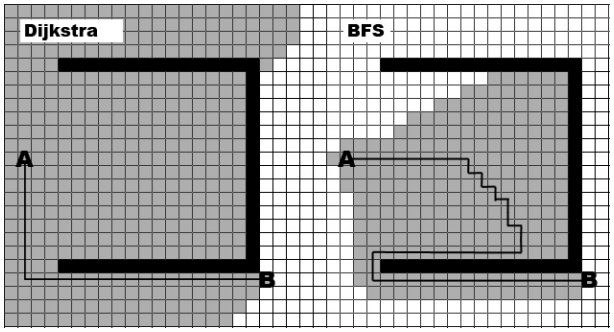

Figure 2.6: Dijkstra versus BFS

It leads to the conclusion that Dijkstra is slow and accurate while BFS is fast but inaccurate. Would it not be good to combine the accuracy of Dijkstra with the speed of BFS? This would require the algorithm to take both the path walked and the estimated distance to the destination into consideration. A Star (A*) is such an algorithm and will be described in the next section.

2.1.2.4 A Star (A*)

The source of the following is [13] and [9].
A* is a popular algorithm for pathfinding because it is flexible and can be used in many contexts ranging from games to various navigation systems. If implemented correctly it is like Dijkstra guaranteed to find the shortest possible path, but will work faster than Dijkstra. In the simple case as in 2.5, A* would work as fast as the BFS and in example two, A* would find the shortest route by checking about half of the vertices as Dijkstra does.

So how does A* work? It uses the cost of the path so far like Dijkstra and an estimate like BFS. In A* every vertex has a score F which is the sum of a G and H score. G represents the cost of the path walked so far and H is the estimated cost from the current path to the destination. Each time the algorithm checks a vertex it will favor the one with the lowest F score. This causes A* to search toward the destination, but since it keeps track of the path walked so far, it will never venture into dead ends like BFS did in 2.6.

A topic we have not touched is how these algorithms save the path they have "walked", or how the final path output is handled. The Dijkstra algorithm actually does not find a path but merely finds the length of it. It is however fairly simple to add the ability to save the path to an algorithm. A common way to handle the path walked is to add a parameter on each vertex that points to another vertex. This is called a parent vertex. The parent vertex is part of the shortest path to the current vertex. Meaning that if you come from the parent vertex, the current vertex will have the lowest possible F-score. Dijkstra actually creates a Shortest-Path-Tree (see section 2.1.1.1) and the edge from a vertex to its parent vertex represent an edge in the Shortest-Path-Tree. This means that if you run Dijkstra on the entire graph then you will end up with a complete Shortest-Path-Tree. Pick any vertex and by following the parent vertices you would end up at the start-vertex and have traveled the shortest path possible between the two vertices. Of course it would be enough to stop the algorithm once the destination is in the Shortest-Path-Tree.
A situation where the heuristic becomes rather useless is when the destination is not known exactly. For example if a graph has more than one possible destination and we want to find out which one is closest to the start vertex. Here a heuristic cannot be determined because its not known where we want the algorithm to go. In this case the Dijkstra algorithm would be optimal because it searches in all directions and the first possible destination vertex it finds would be the closest one.
For A* to work properly it is important to have a good heuristic function. If the heuristic over- or underestimates the distance to the destination it affects the algorithm in different ways. It can be generalized into five cases.

- Extreme Underestimate
  If the heuristic is zero all the time it will have no influence on the algorithm and A* would turn into Dijkstra.
- Underestimate
  If the heuristic is lower than the actual cost of moving from the current vertex to the destination, the algorithm will always find the shortest path. The lower the heuristic is the closer the algorithm gets to Dijkstra and more vertices are checked. This is because the walked path still "weights" more than the estimate and causes the algorithm to search more vertices.
- Exact Estimate
  If the heuristic always is exactly equal to the actual cost of moving from the current vertex to the destination, the algorithm will find the shortest path and not expand very far from the shortest path, making A* very fast. This is however hard to obtain in all cases, but given the right information this is the way A* star works optimally.
- Overestimate
  If the heuristic overestimates the actual cost, the algorithm can no longer guarantee to find the shortest path, though it might run faster. Because the estimate now "weights' more than the walked path the algorithm can in some cases run into dead ends like the BFS

- Extreme Overestimate
  If the heuristic is very high compared the the actual distance, the cost so far plays no role and A* turns into a BFS algorithm.

The source of the list above is [13].

This means that the heuristic can control how A* should behave. When implemented, the developer can balance A* to be slow and accurate or fast and inaccurate. Sometimes a good path is better than a perfect path, because the good path is faster to calculate. In game programming A* could be required to return a lot of paths all the time and speed would be more important than accuracy. In other cases where A* do not have to return many paths fast, a slower and more precise version would be preferred.

The following is the pseudo-code of A*. Note the added heuristic function H() and the parent function P():

Procedure AStar(G: weighted connected simple graph, with
    all weights positive)
{G has vertices a = v_0, v_1...., v_n = z and weights w(v_i,v_j)
    where w(v_i,v_j) = ∞ if w(v_i,v_j) is not an edge in G}
for i:= 1 **to** n
    L(v_i) := ∞
L(a) := 0 + H(a,z)
S := $\emptyset$
{the labels are now initialized so that the label of a is 0 and all
    other labels are ∞, and S is the empty set}
while z $\notin$ S
begin
    u := a vertex not in S with L(u) minimal
    S := S $\cup$ {u}
    for all vertices v not in S
        if L(u)+ w(u,v) + H(v,z) < L(v) **then**
            L(v) := L(u) + w(u,v) + H(v,z)
            P(v) := u
    {this adds a vertex to S with minimal F-score, updates the
        F-scores of vertices not in S and sets the parent.}
end {L(z) = length of a shortest path from a to z}

It is then possible to find the shortest path by backtracing from the destination-vertex to the start-vertex following the parent-vertices.

2.2 System Development Methods

2.2.1 Introduction

In this section we explain how a software system can be developed using different system development models[1]. How have these software developments models evolved? What are the advantages of different models and how do they differ? What is traditional and agile system development?

[1] a model is the same as a method when we use them throughout the report

The motivation for this research is to find which development model suits our project best. We start with the traditional models and then move on to the agile models. In the end of the chapter we conclude on our choice of development model.

First we will explain what a software development model is. A software development model is an optimized and structured life cycle for developing a software product. There are many different models, that each have their own approaches on how tasks are distributed, how phases are planned etc. Conventional models tend to prefer to take a very linear approach, while agile models are iterative and hereby able to change what ever is necessary at any point in the project.

2.2.2 Conventional Methods

In this section we explain different conventional system development methods. How do they work, what are their advantages and their disadvantages?

2.2.2.1 The Waterfall Model

The source for the following section is [29]. The Waterfall Model is a process for developing software, originally proposed by Winston W. Royce in an article published in 1970, however Royce never actually used the word waterfall in his article. Royce used the waterfall model in his article as an instance of a flawed way of developing software. The article discusses how the waterfall model could be developed into an iterative model, with feedback from each phase influencing following phases, which then lead to the spiral method. The Waterfall Model is the earliest method of structured system development.
The original waterfall model consists of seven phases:

1. Requirement specification
2. Design
3. Construction/Coding
4. Integration
5. Testing and debugging
6. Installation
7. Maintenance

When following the waterfall model, one must complete each step entirely and perfectly, before moving on to the next. This is why it is called the waterfall model; once you have passed a step in the waterfall, you cannot move back up.

There are numerous arguments for and against the waterfall model: A typical pro-waterfall argument is, if for instance the design is impossible to implement, then it would have been a major time saver just to have had ensured that the design was perfect, before moving on, so that you do not have to start all over again when it fails. However in practice this seems to be for multiple reason: It is very unlikely that a stage even can be perfected in a non-trivial system. Another argument against the waterfall model is that, for instance the clients requirements might change during the development. The client may want to give feedback on

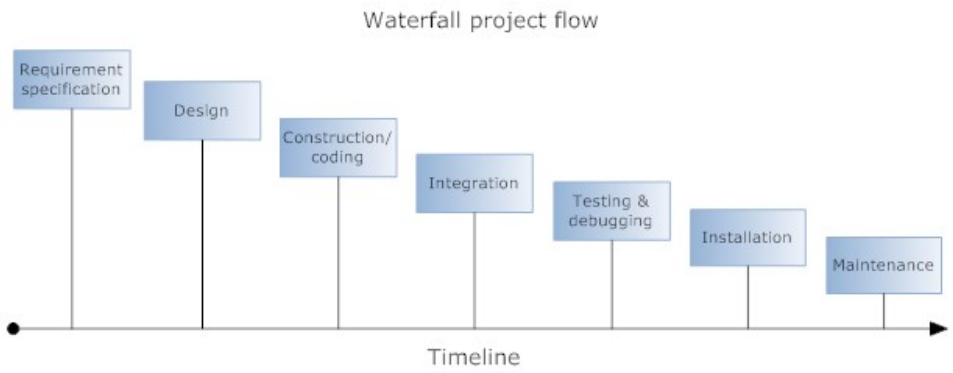

Figure 2.7: The Waterfall Model

prototypes and change his requirements.

Even though the waterfall has been used by large organizations such as NASA and the U.S. Department of Defense, it is not considered as a smart method by the most of the development community. Many consider it to be good in theory, but bad in practice.

2.2.3 Rational Unified Process (RUP)

In this section we describe one of the most widely used conventional software development methods called Rational Unified Process or RUP. How is it used and why is it popular. What makes it better and more efficient than the waterfall model?

The Rational Unified Process is an adaptable iterative software development process framework. It is heavily defined and well documented. It is intended to be customized by the development organizations and development teams, that will choose the elements of the process that are best suited for their needs. It was developed by the Rational Software Corporation in the 1980s and 1990s.

RUP is a part of IBM's Rational Method Composer or RMC product which is a software product, that allows customization of the process. It enables the users to select and deploy only the process components they need, and then publish it through their own intranet. The RMC product includes a hyperlinked knowledge base with sample *artifacts* (see page 73) and detailed descriptions for many different types of activities[24]. The concept of user stories descends from RUP, which will be explained in the section about Extreme Programming (XP).

The Rational Unified Process is based on the six key principles for business-driven development[14]:

A Adapt the process

B Balance stakeholder priorities

C Collaborate across teams

D Demonstrate value iteratively

E Elevate the level of abstraction

F Focus continuosly on quality

1. **Adapt the process**
   First the process must be adapted to the needs of the project or organization in question. RUP provides a wide array of pre-customized templates for many different types of projects. These templates can then easily be further customized. Adapting such a process also encourages the continuous improvement of an existing process in an organization.

2. **Balance stakeholder priorities**
   This widens the discussion between stakeholders and business goals. They often conflict, which needs to be balanced out between the parties involved.

3. **Collaborate across teams**
   Communication between team members, stakeholders etc. is essential. That is why test results, release management and project plans are exchanged regularly.

4. **Demonstrate value iteratively**
   Projects are delivered in increments in an iterative way. The increment, which includes the value of the past iteration, is used to measure if the project progresses. That increment is also used to encourage feedback from stakeholders about the direction of the project. This allows projects to adjust to changed situations based on the feedback.

5. **Elevate the level of abstraction**
   A major problem in software development is complexity. Working at a higher level of abstraction reduces complexity and facilitates communication. One effective approach to reducing complexity is reusing existing assets, such as software patterns, framework or open source software. And if you have a higher level of abstraction it can also allow discussions on different architectural levels. These can be accompanied by visual representations of the architecture, for example by using Unified Modeling Language (UML, see more on page 44), which is also an invention of the Rational Software Corporation.

6. **Focus continuously on quality**
   Quality checks are performed at almost a daily basis, rather than at the end of each iteration. Scripts helps when dealing with the always increasing amount of tests due to iterative development.

The RUP life-cycle
The best way to explain how a RUP process works is with the chart seen below.

- The horizontal dimension represents time and shows the dynamic aspects of the process expressed in terms of cycles, phases, iterations, and milestones.

- The vertical dimension represents core process disciplines also known as work flows, which group different software engineering activities by their nature [20].

The four phases
The process consists of four phases, which each are followed by a milestone, where the stakeholders assess the state of the project and must agree, that the tasks are completed as they should. If so, the project proceeds to the next phase.

1. The Inception Phase
   In this phase the following tasks are performed: Define the project scope, estimate cost and schedule, define risks, develop business case, prepare project environment, identify architecture. This is followed by the milestone called LCO (life cycle objectives).

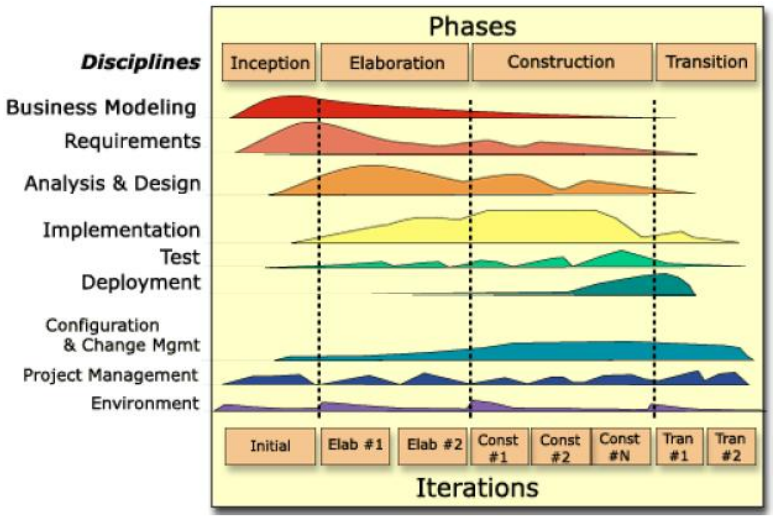

Figure 2.8: Phases, disciplines and iterations of RUP. *This image is Copyright by Rational Software Corporation, now a part of IBM.*

2. The Elaboration Phase
   In this phase the following tasks are performed: The requirements must be specified in greater detail, the architecture must be validated, the project environment must be evolved and a project team must be staffed. This is followed by the milestone called LCA (Life cycle Architecture).

3. The Construction Phase
   In this phase the following tasks are performed: The system must be modeled, built and tested, and supporting documentation is developed. The milestone at the end of this phase is called IOC (Initial Operational Capability).

4. The Transition Phase
   And finally these tasks are performed: System testing, user testing, system refactoring and system deployment. The final milestone is called PR (Product Release)

A typical RUP project spends 10% time in the inception phase, 25% time in the elaboration phase, 55% time in the construction phase and 10% time in the transition phase [18].

2.2.4 Agile System Development

This section begins by describing the origin and basics of agile system development. Then we move on to cover some selected principles from the agile manifesto. In the end we introduce the declaration of interdependence.

The idea of agile software development was first introduced in a paper by E. A. Edmonds in 1974. Here Edmonds describes a development method that was more adaptive to the project than other development methods at that time. But it was not before the 90s that this idea became a reality in the form of several different approaches like Scrum and Extreme Programming, which we will describe later in this chapter. These new lightweight development methods came as a reaction against the conventional methods, also called heavyweight methods, like the waterfall model. Pro lightweight system development key figures typified conventional

software development methods as being heavily regulated, micro-managed and too static in their form. In 2001, 17 representatives from all the different lightweight methods gathered to find some common ground for their new development methods. This gathering resulted in The Agile Manifesto, and hereby the term agile software development was created. The purpose of this manifesto is the following: "We are uncovering better ways of developing software by doing it and helping others do it. We value:[1]

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Costumer collaboration over contract negotiation.
- Responding to change over following a plan."

The introducing sentence points out that the signatories of the manifesto are still elaborating the methods by practicing them in their own work. The four bullets are contains their preference followed by an important item they see as a lower priority. They acknowledge the importance of processes and tools but state that interactions of individuals overrule any of such. The next bullet points out that working software is of greater importance than documentation. This is one of the more radical changes they present since most conventional methods often were documentation driven. The last two bullets point out that continuous costumer collaboration and welcoming change is better than making strict contracts and plans.

The Agile Manifesto consists of twelve principles that deflect the purpose from above. Whereas the purpose may see vague the principles elaborate their meaning into a more doable size. Each of the many different agile development methods implement these principles in their own way. The Manifesto is not to be seen as a total solution but more as a toolbox. It is up to the user to decide which principles can be used to best suit their needs. We will now cover some of the principles that are essential to our project [1].

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Costumers does not care much about UML diagrams and other technical documents they only care whether or not they get the working software they ordered in time. In agile terms this means they get a piece of software that proves to them that the project is evolving in the right direction and that it suits their demands.

In traditional project management achieving a plan equals project success which then equals costumer value. Costumer value in software development is when the software delivered works in the best interests of the costumer and hereby giving them a competitive advantage in their respective field. The swiftly changing demands in todays projects require that the costumer value be reevaluated often to ensure it follows the best interests of the costumer. Therefore achieving a plan that was made a while back does not automatically ensure success due to changes in costumer value.

- Simplicity–the art of maximizing the amount of work not done–is essential.

By implementing the easiest solutions possible the programmers not only saves time but also makes it easier for other developers to do maintenance and implement changes on the software. When the project leader assigns a new task to a team he should ask them "What is the simplest solution that could possibly work?". By doing this the team only needs to focus on what is needed now and not what might be needed in the future.

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

During the last three decades the software development methods have moved from delivering the final product in the end of the process to having multiple deliveries, each adding new features to the product. Even though this practice is being used widely it is not yet where it needs to be. The Manifest states that the shorter the iteration the better. It is also important to distinguish a delivery from a release. It may not be the optimal choice to put new software into production every other week due business and/or management issues. But it is still important to have frequent deliveries that allows everyone in the project to evaluate and get up to speed on the growing project.

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Whereas traditional development methods use documentation as a tool, to ensure that everyone in the project get the information they need about the product, agile methods use face-to-face communication. Even though many of the founders of the manifest are writers they acknowledge that it is not the best way to convey information amongst individuals. And because the problem in both cases is understanding and not documentation agile methods swear to use face-to-face communication over documentation. By using face-to-face communication it is not a heavy tome filled with documentation that moves around the people in the project, but instead it is the people themselves that contain the knowledge and now it is just a question of getting the right people together. Good communication in a agile method is therefore based on face-to-face communication that is achieved through different practices. These practices include pair-programming, daily and weekly costumer-team communication and all programmers are on the same location during development. We will describe these practices later in this chapter.

- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The authors of the manifest acknowledges that even though agile development methods are superior to traditional methods, they are not processes that can be followed slavishly form start to finish with perfect results. Therefore the team must refine, reflect and evaluate their practices as they go along to get the best results under their project circumstances. So instead of the management monitoring the process and work-flow constantly, they should trust their employees to improve their development processes amongst themselves both individually and in their teams.

Declaration of Interdependence

In 2005 the Declaration of Interdependence, an addendum to the Agile Manifesto, was written to help project managers achieve an agile mindset in terms of product and project management. This declaration consists of six management principles that focus on team effectiveness, creativity and accountability. These principles are mainly elaborations of principles from the Agile Manifesto but aimed more at project leaders. The two most outstanding are:

- Boost performance through group accountability for results and shared responsibility for team effectiveness.
- Unleash creativity and innovation by recognizing that individuals are the ultimate source of value, and creating an environment where they can make a difference.

By sharing responsibility for team effectiveness no single group in a project will fall behind, since there is a team spirit. The argument for the second principle is to create an environment where the individual can still thrive while beeing part of a team. The way this is achieved is by giving individuals the courage needed to unleash their creativity and innovate better software solutions.[3]

2.2.4.1 Scrum

In this section we describe Scrum, one of the most used agile system development methods.

The source of the following section is [19].

Scrum is an agile software development method, in which the focus is how the project is organized and planned, opposed to extreme programming where the focus is how to work with programming.

Scrum is based on a 30 calendar day iteration (about 1000 hours), called a sprint, which is a focused effort to reach a series of fixed goals, called the product backlog.

There are several roles in the Scrum model: The product owner, the scrum master and the scrum team. The product owner represents a customer. He is usually the customer itself, but can also be a part of a large organization. The product owner administers the product backlog, which is a current to-do list, in which all the specifications for a product are listed according to how profitable they are deemed to be. The scrum master is the leader of the scrum team. He meets with the team everyday at 15 minute meetings called daily scrums. The objective of these daily scrums is to keep the work running smoothly and to eliminate any problems that might arise. The scrum master must always provide the best possible circumstances for the scrum team to reach the goals, provided by the product backlog. After the sprint the scrum master holds a meeting called a sprint retrospective, which has the purpose of elevating the scrum teams knowledge and motivate them for the next sprint. And finally the scrum team consists of five to nine people, which performs the actual work. There are no specific roles for the team members. All team members should be able to swap task along the way if needed, however this does not mean that the individual team members cannot have areas of expertise.

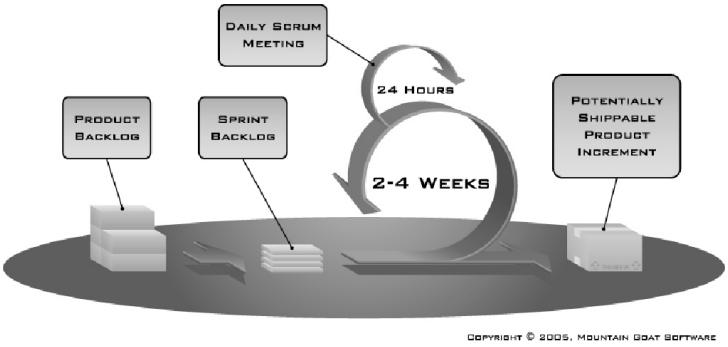

Figure 2.9: The Scrum process

The process consists of four phases:

1. **Backlog creation phase**
   The product owner composes a prioritized to-do-list of requests and specifications, such as new functionality and bug fixes. When the product backlog is finished, the product owner calls the scrum team for a meeting, to make the sprint backlog.

2. **Sprint phase (30 days)**
   The first few days are set aside to create a sprint backlog, which is a list of the highest prioritized tasks from the product backlog, that are estimated by the scrum team to be realized within the sprint period. After this the scrum team will distribute the work by it self. The better the sprint log is composed, the more smoothly the work will run. Every morning during the scrum, the daily scrums are held by the scrum master to ensure that everything is running as it should.

3. **Demonstration**
   A large informal meeting is held, called a sprint review, in which the new functionality is demonstrated for an audience, consisting of the product owner, management, customers etc. This meeting lasts about four hours.

4. **Evaluation**
   The scrum team and master has a three hour meeting called a sprint retrospective, in which they review both what went well and what should be improved in the next Sprint. This is the starting block for a new potential sprint.

And so the sprints continues until the product owner deems the project closed.

2.2.4.2 Extreme Programming (XP)

This section describes the background, values, principles and practices that make up extreme programming.

Extreme Programming (XP), one of the first agile software development methods, was a by-product of Kent Beck's work on the Chrysler Comprehensive Compensation System (C3) payroll project, on which he became project leader in 1996. The

objective of the C3 project was to replace the company's many different payroll applications with a single one. The project had run for a year without substantial progress before Kent Beck became project leader. It took about a years work on the project for Kent Beck to create a software development method that he would later refer to as extreme programming (XP). The project adopted the XP method 1997 and went into production about a year later. The system only made it to pay 10.000 people due to performance problems, but despite two more years of development there never was another release. The project was stopped in February 2000. Although XP had an unsuccessful start the method had caught on in the software engineering field and, according to IBM, XP is one of the hottest software development methods as of 2001.[22][30]

XP is based on a set of values the define what XP intends to improve and why. Out of these values we derive the basic principles that describe how such values are manifested in a project. Each principle embodies the values, and hereby the slightly vague values become more concrete. In the next section we describe these values in further detail.[6]

Values

The sources for this section are [6] and [8].

There are four values in XP. Each value aims to solve a problem that often arose using other development methods. Kent Beck compares these values to society values like rituals, punishments and rewards. He states that without these values humans would revert to their own short-term best interest and not pursue the best interests of the team/society. This makes sense in system development where the optimal setting is the team working together in unity as one organism. We now move on to cover the values in detail by describing the problem and how XP aims to solve it.

- **Communication**
  One of the most common problems in projects are often cause by somebody not talking to another somebody about something important. This kind of bad communication is not just entirely up to chance, but mostly the result of different circumstances. Such a circumstance could be the programmer not wanting to report bad news to the manager to avoid the confrontation. It could also be intercommunication between programmers and costumers that go wrong due to almost infinitely many reasons.

  XP aims to ensure a good communication flow by applying practices that cannot be done without proper communication present. Some of these practices are unit-testing, pair programming and task planning and estimation. They are all used consistently throughout the project and require that managers, programmers and costumers communicate on a regular basis.

- **Simplicity**
  The programmers should always strive to code the simplest thing that could possibly work. This keeps the programmers from wasting time on complex designs that maybe forfeit later on in the project cycle. XP makes the bet that it is better to do a simple thing today and then maybe spend a little time and money tomorrow changing it, than to make a complex solution today that maybe forfeit anyway. When a project manager assigns a task he should

ask the team "What is the simplest solution could possibly work?". By doing this the team does not have to focus on functions that might have to be implemented later on in the project. This ensures that there is minimal work hours lost on forfeit implementation. Kent Beck acknowledges that obtaining simplicity in code is one of the hardest things to accomplish. He states that by thinking ahead, and implementing things that will be needed next week or next month, the programmers are listening to their fears of the exponential cost of change curve. They fear that the longer they wait the more expensive and challenging it will get later. This fear will get suppressed as the team gains courage and experience using XP.

- **Feedback**
  When developing a system using the XP method the people in the project get constant feedback throughout its course. The team gets feedback from the system through practices like unit tests and collective code ownership. These two practices will be described further down in this section. The team also get indispensable feedback from the costumers. This is due to the short delivery cycles in XP that the programmers get a lot of feedback from the costumer and can therefore quickly change project direction and hereby avoid wasting too much time working in the wrong direction. This is one of the biggest strengths of XP and other agile development methods, being able to get constant feedback on the costumer value of the growing system.

- **Courage**
  If the team lacks the courage to follow XP they will tend to follow their own short-term best interests and not abide the values of XP. But when courage exists the programmers are not afraid to welcome changes late in a project and throw away useless code if it is no longer needed. It could be days of coding that should be discarded because of bad design, or realizing at the end of a project that the architecture is bad and the system would need a redesigning to ensure good costumer value. Throughout the life of a project there will be many situations where difficult situations will arise and put pressure on the team. But it is often in these situations that the future success of the project is achieved or lost. It is therefore essential that both the team, management and the customers have the courage to follow XP even though it may be easier at first not to.

Summary

It is important to understand that these four values all rely on each other to ensure optimal functionality and understanding. Communication ensure simplicity because of feedback from the team and costumers. Communication and feedback allows for courage during development because of unit tests and team spirit. These are just a couple of the interconnections between the values of XP, but they illustrate that when it comes to the core values of the methodology XP comes as a complete package. The part where a team can pick and choose for themselves is when it comes to which practices they wish to use.

Practices

XP consists of twelve different practices. We will now describe them all starting with the seven practices we used followed by a short description of the last five.

- **Pair Programming**
  When developing a XP project it is the optimal setting to use pair programming. Pair programming is when two developers work together in front of a single computer. One person controls the keyboard and mouse while the other person monitors and thinks over the code that is being written. Then after a while the two switch places to get a bit more variety. The first thought critics get when hearing about pair programming is that it then takes twice as much time to develop the same software as one would normally do. But that is not true. Two people working one the same computer will actually produce the same amount of code as they would separately but by working together the quality of the code produced is higher. And with increased quality comes savings later in the project because there is no need for comprehensive refactoring and adjustments.

- **User stories**
  User stories are written by the costumer and contain the things they need the system to do for them. A user story should not contain more than three sentences of text without any technologysyntax. The user stories substitutes a requirements document and there should consequently be user stories for all the things the costumer needs the system to do. The developers can help the users visualize what features their system should contain and how they see themselves use these features. All developer tasks are derived from the user stories and it is up to the costumer to add new stories or delete existing ones as the project progresses.

  Using user stories is not an actual XP practice but a way to get the requirements for the system directly from the user. We have chosen to insert the description here because we will be using the term frequently during the rest of the report.

- **Spike Solutions**
  Spike solutions serve the purpose of discovering tricky parts early in project to ensure better release planning. The spike solution should be a simple solution of what the team estimates to be tricky and/or time consuming. The code is not meant to be used in the actual system, but is most likely to be discarded.

  Spike solutions are also not an actual XP practice.

- **Planning game**
  The planning game covers both release and iteration planning. The release plan is made by estimating the time needed to implement the collected user stories. This plan is revised constantly during the project to fit the project velocity. For every iteration that does not go as planned the release plan is pushed forward. It is important that the costumer understands this way of working before the project begins. The business has to understand this way of working for it to work.

- **Testing**
  There are two kinds of testing in XP. One is unit testing which is performed by the developers, and the other is acceptance tests which is performed by the costumer.

  Unit testing - XP development uses a test driven approach. This works by first writing a test that checks if the code in question does what it is supposed to do. This could be checking if a method returns the correct data or if GUI elements are actually shown. Each unit test serves as a strict contract that the code must abide in order for the test to run successfully. All the test are assembled in a test suite that, when run, checks if all the individual tests run successfully. Writing a test before the code also helps to ensure simplicity because only the needed code will be written. The developer only needs to implement the code necessary to make the tests run successfully. This helps the developer to not think forward and start implementing more features than their task requires.

  Acceptance testing - Acceptance tests are performed regularly by the costumer to ensure that the user stories have been implemented correctly. Acceptance tests are usually taken after each iteration.

- **Coding Standards**
  To maintain easy understanding and readability of the code for all programmers in a project, the naming of classes, methods and variables should be consistent. A way of doing this could be that the programmers made a unanimous decision on a code standard to use throughout the project along side the standard formatting of the used programming language.

- **Sustainable Pace**
  Acknowledging that Pair Programming is a more intense way of developing because you always have to present out of respect for your partners time. Being present does not only mean that you have to actually be there, it also means you cannot let your mind wander off as you could if programming alone. XP's solution to this problem is to advocate following a sustainable pace through the whole project. Usually this means sticking to a 40-hour work week instead of tiring the developers out by assigning too much overtime. By limiting the work week to 40 hours the programmers have a better chance of upholding a good productive value per hour. Another thing to de-intensify the development is to take a 15 minute break after each task is completed. This allows the developers to get away from the monitor in a while and maybe discuss what went good, what went bad, what may need improvement or maybe just have a small coffee break.

- **On-site Customer**
  During a XP project the costumer assigns an employee to be available to the development team all the way through the development phase. This person is in most cases either one of the future users of the system or, if it is a commercial product, a person that has responsibility for the finished product. The team can then use this person as their knowledge base if they have any questions about the system they are developing. The optimal setting is to have the costumer on-site so that communication can be face-to-face and take

place in the developers environment.

- **Refactoring**
  To keep the source code clean and simple it is important to review it once the unit and acceptance tests pass. This is done by refactoring. When refactoring the developers remove all code that is not necessary for the tests to pass, and hereby ensure optimal simplicity of the code. Refactoring does not change the result of the code, only simplifies it. Usually the teams refactor each others code to get a more objective perspective on it's functionality.

- **Continuous Integration**
  To adapt changes in a project more rapidly XP uses continuous integration throughout the process. And to find the errors or misunderstandings that result in these changes the costumer need to be able to give feedback to the team constantly. XP solves this by following an iterative approach where each iteration should be as short as possible, usually an iteration last a week. For each iteration user stories are split into tangible tasks that are divided amongst the teams. Then after each iteration the system is shown to the costumer to get valuable feedback on the quality of the the present product.

- **Collective Code Ownership**
  In XP all members of the team has the same rights to view and edit all the source code. It would be very hard to practice pair programming and rotate teams without this practice. Collective code ownership will only work if unit tests are written for the code and the teams do not implement more changes from more than a few hours worth at a time. Unit tests help to ensure that new functionality or changes in the code does not compromise the functionality of existing code. And by implementing small changes regularly instead of big changes once a week helps to find out where and why a possible error occur. Collective code ownership also ensures knowledge is available to the entire team which gives the everyone the ability to find and correct faulty or complex code.

- **Simple Design**
  When developing with XP the developers should always strive to implement the simplest possible solution that solves the task or user story. It is important not to start thinking ahead on future functionality because this will just spoil the advantages of continuous integration. It is important to acknowledge that doing the simple thing is not always as easy as it sounds, especially if the developers are lacking courage they will tend to follow their own beliefs and risk assessments.

- **Metaphor**
  A metaphor in XP is a model or rhetorical figure which purpose is to explain the functionality and environment in which the system will be integrated. The metaphor should give the reader an idea of what the product will contain. It is possible to use a role model analysis instead of a metaphor if there exists a obvious candidate.

- **Small Releases**
  Each release should be as short as possible, and only contain the most relevant features that are prioritized through the user stories. This way the costumer gets the most relevant features at that time in the project and can hereby better estimate the priority of what should be worked on in the next iteration.

Course of a Sample XP Project

A XP project starts by collecting user stories from the costumer. These user stories are split into tasks and analyzed. Then spike solutions are made for possible tricky parts. Then a release plan based on the analyzed user stories are made to give the costumer a rough idea of the project time schedule. After the release plan is made the first iteration is planned. During the iteration planning tasks are divided amongst the teams and then the iteration is ready to begin. During the iteration the teams implement the simplest possible solution that could possible solve the tasks they have been given. This is done by first writing unit tests and then implementing the code needed to make these tests run without failure. If there is time to spare the code is refactored and optimized to ensure simplicity and performance. When the iteration is over the next planning of the next iteration commences. If some teams did not complete all their tasks then the release plan is revised to fit the current time estimate. The cycle continues like this until either all user stories have been implemented or the project is demarcated due to money/time issues. This is only a sample project cycle and there are many other factors and processes like system integration and acceptance tests that take place during development.Below is a simple illustration that shows the course of a sample XP project.

1. Collect user stories.
2. Write a system metaphor.
3. Conduct spike solutions for tricky parts.
4. Create a release plan.
5. Iteration cycle.
   (a) Iteration planning.
   (b) Choose new user story.
   (c) Implement the simplest solution that could possibly solve the task.
      i. Write unit tests.
      ii. Implement until all unit tests run without failure.
   (d) Optimization and refactoring.
   (e) Small release.
   (f) Re-estimate release plan and reorganize user stories.
6. Repeat from 5. until all user stories are implemented or development is stopped due to other reasons.
7. Product goes in to production.

2.2.5 System Development Method Summary

In this section we sum up on the different methods we have described in the earlier sections.

We have made a diagram that displays the four methods we have covered on two different axises. The main axis shows how linear or iterative the method is, and the second axis shows when the method was created.

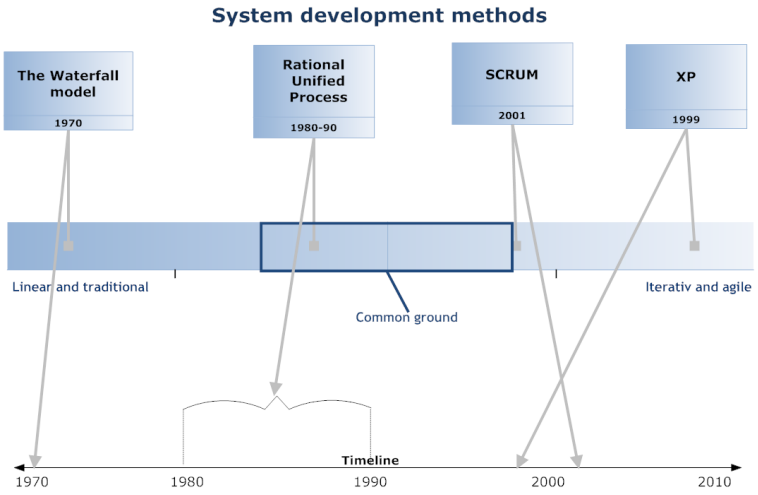

Figure 2.10: Comparison diagram

The diagram only shows the methods we have covered in this project, there are of course many other alternatives, but these were the ones that we found essential to the subject. The dates are rough estimates since they all have evolved over time into were they are currently at, but the dates we have set are when they were officially published. The main axis has a field in the middle that illustrates that the methods in the middle share many of the same characteristics. The time line shows that the new methods are very iterative, with XP as the most extreme of them. But even though Scrum is more recent that XP has moved a little over towards the middle. The main difference between XP and Scrum is the length of the iterations, where Scrum is longer than XP.

2.2.6 Choice of Development Method

In this section we compare agile and traditional methods and try to determine which is best for our project, and why so.

The source for the following is [21].

When organizations use different development methods, their final products are very different. The new agile system development methods and practices were created because of a low rate of success in the field of system development. The early adopters of the agile methods believed that their use might change that, while the followers of the more traditional methods thought that the agile methods were too chaotic and lacked the strict procedure that for instance RUP possesses.

The most important difference between agile and traditional development is that the traditional methods try to minimize the amount of change as the project progresses, through strict and direct requirements gathering, analysis and design. The intention of this is to attain higher quality results under a controlled schedule. Agile methods on the other hand embrace and depends heavily on these constant and invertible changes throughout the project. Agile methods uses change to achieve innovation through individual initiatives. The focus is on adaptation and innovation rather than prediction and control.

Even though adoption of agile development is increasing in the industry, it may not be the best approach for any given project. Modern organizations engage in a considerable variety of development project, but when is it most convenient to use an agile approach and when is it most convenient to use a conventional approach? the choice of whether traditional or agile methods is given for a project is largely dependent on five factors. Source of the following is: [21].

- The size of the systems development project and team
- The consequences of failure (i.e., criticality)
- The degree of dynamism or volatility of the environment
- The competence of personnel
- Compatibility with the prevailing culture

If a project has stable requirements, lots of legacy code and mainly involves maintenance tasks, it can be very inefficient and difficult to adhere to an agile method. Traditional development is desirable when the requirements are stable and predictable and when the project is large, critical, and complex. On the other hand, agile development is suitable when there is a high degree of uncertainty and risk in the project, arising from frequently changing requirements and the application of new or unusual technology.

2.2.6.1 Our Choice

We have chosen to use XP as our system development method based on the following reasons:

- We will be using technology that is new to us, this makes iterative development method a great choice because we quickly will discover if something does not work.
- Our product is not going to be a mission critical application, like hospital equipment software, therefore we can use a development method that involve a greater risk.
- Our costumer provides unstable system requirements.

Other more non-technical reasons for why we chose XP:

- It came as an indirect demand from our costumer, in terms of requested frequent deliveries and involvement in the development process.

- Having read a lot of material on XP and other methods we have mainly decided to use XP because it seems interesting to try out in practice. Also the fact that agile system development is in the early stages of its development makes it exciting to be a part of.
- The fact that we have been assigned a supervisor that have industry experience with using XP, makes it a great opportunity we should not miss out on.
- Pair programming seems like a good way to divide development tasks amongst the group members.

2.3 Related Technologies

2.3.1 JUnit

JUnit is a unit test written in Java [10]. Unit tests plays a very important role in performing the XP system development method. With unit tests, the source code of the project is checked for errors before the program is distributed. This is important in XP, because of the collective code ownership that enables all teams to edit the same code. If there are many developers working on the same project, some mistakes could occur like changing the existing code could go wrong. By writing a method JUnit test before writing the method, you can be sure that when other developers are editing your source code afterward, the JUnit test will display an error if something goes wrong. In this way the developers are warned if they made a change that did not work as intended. By having unit tests, it becomes easier to refactor the code because eventual new errors are detected instantaneously. When the tests are written it is also possible to edit the code later and this is where refactoring comes in handy. Refactoring is also a big part of XP, which is why JUnit is so important when performing the XP system development method.

2.3.2 ANT

ANT is short for "Another Neat Tool" and is an automated build tool mainly for Java projects [2]. It can be used for many things, such as: Compiling the project, do documentation (by using *Javadocs*), making the distribution media, create statistics about the project, run JUnit tests and more. There exists many plugins to ANT, so it can almost do anything you want. ANT uses *XML* to define the building process and thereby describing the instructions carefully. By using ANT you are guaranteed that every build is performed alike. This also ensures that the whole team compiles with the same version of Java. Another thing about ANT is that it is automated. In this way you can set it to check for errors in JUnit-test every hour, so if there suddenly occurs an error it can send a mail to the developers who made the error.

2.3.3 OOP: Object Oriented Programmering

This section describes what OOP is and how to develop software using it.

The source for this section is [7].

The concept of OOP is the use of classes and instances of these classes, which are the objects. In this section we will use a simplified example about vehicles and cars.

The use of this example should make the understanding of OOP more simple. The first piece of code is shown beneath to create an underlying basis for our explanation of OOP:

```
1  public class vehicle {
2          //The fields:
3          private int wheels;
4          private String manufacturer;
5          //The constructor:
6          public vehicle(int _wheels, String _manufacturer){
7                  wheels = _wheels;
8                  manufacturer = _manufacturer;
9          }
10         //The methods:
11         //Getters and setters:
12         public void setWheels(int _wheels){
13                 wheels = _wheels;
14         }
15         public int getWheels(){
16                 return wheels;
17         }
18         public void setManufacturer(String _manufacturer){
19                 manufacturer = _manufacturer;
20         }
21         public String getManufacturer(){
22                 return manufacturer;
23         }
24         //Specific class-methods:
25         public void Drive(){
26                 //moves the vehicle
27         }
28         public void TyreChange(){
29                 //changes a tyre
30         }
31 }
```

2.3.3.1 Classes

As you can see in the example, a class is a model of a real-life object. In our example we have written a simplified structure of a vehicle. This class has two variables, `wheels` and `manufacturer`, which characterizes a vehicle. These variables are also commonly known as fields. Setting the state of the fields is an expression that means to alter the variables in the class. When creating an instance of a class, the class uses a constructor to set the state of the fields in the class. A constructor is a method with parameters and it is called when creating an instance of an object. The constructor must, in many languages, be named the same as the class. There can also be multiple constructors inside a class which normally are one without parameters and one with parameters. A class can contain methods. There are mainly two kinds of methods in a class. The first kind is the methods which change or return the state of a field in a class. These methods are called mutators, if they change the variable. And accessors if they return the variable. Secondly is those methods which are specific for the class. In the example there are two mutators, `setWheels()` and `setManufacturer()`, and two accessors, `getWheels()` and `getManufacturer()`. The two methods `Drive()` and `TyreChange()` are specific methods for a vehicle which brings the aspect of reality to the class. A variable is either private or public. Public means that it can be read or used outside the class, private means it cannot.

2.3.3.2 Inheritance

A class is able to inherit from another class. To illustrate this we have created a class `car` which inherits from the class `vehicle`:

```
public class car extends vehicle{
        //The fields:
        private int mileageRecorder;
        //The constructor:
        public car(int _wheels,
                        String _manufacturer,
                        int _mileageRecorder
                ){
                super(int _wheels, String _manufacturer);
                mileageRecorder = _mileageRecorder;
        }
        //The methods:
        //Getters and setters:
        public getMileageRecorder(){
                return mileageRecorder;
        }
        public setMileageRecorder(int _mileageRecorder){
                mileageRecorder = _mileageRecorder;
        }
        //Specific class-methods:
        public void TurnOnEngine(){
                //Turns the engine on
        }
        public void TurnOffEngine(){
                //Turns the engine off
        }
}
```

As you can see in the example above, the structure of the class `car` is not much different from `vehicle`, though there are two lines that differ. Besides the class declaration the first line also has an extra phrase `extends vehicle`. This means that this class `car` inherits from `vehicle`. Because of the inheritance from "vehicle", the `car` class also contains the fields, constructor and methods from `vehicle`. This means that the constructor from `car` calls a method `super` which invoke the constructor from `vehicle` with the parameters it needs. `Car` has furthermore two methods `TurnOnEngine()` and `TurnOffEngine()` which are specific to the class. In OOP multiple inheritance is possible, but depends on the programming language used.

2.3.3.3 Objects: Instances of classes

As mentioned earlier, objects are instances of classes. This means that when you create an instance of a class you create an object. If you want to create an object named `Lorry_1` from the class `vehicle` then you would write:

```
Vehicle Lorry_1 = new Vehicle(6,''Scania'');
```

This creates the object `Lorry_1` and the object has six wheels and `Scania` as manufacturer. Now you can call the methods of the object by using dot notation. Dot notation consists of a dot between the name of the object and a method from the object. If `Lorry_1` punctures and we need to change the tyre, we could write:

```
Lorry_1.TyreChange();
```

Then the method `TyreChange()` is invoked and the `Lorry_1` changes tyres.
OOP is effective when the architecture of the program is schematized. This is usually done by making a UML-diagram, which is described in the following section.

2.3.4 Unified Modeling Language (UML)

The source for this section is [27].

UML is a way to describe all sorts of models using diagrams. It is used for many things including business processes, data and application structures, behavior and architecture. Currently there exist a lot of different tools to edit and view UML; typically it is saved as XML-files. It appeared from different types of already existing modeling languages back in the ending of the 80's and it have been further developed since. The diagram above is like a tree which goes down in different

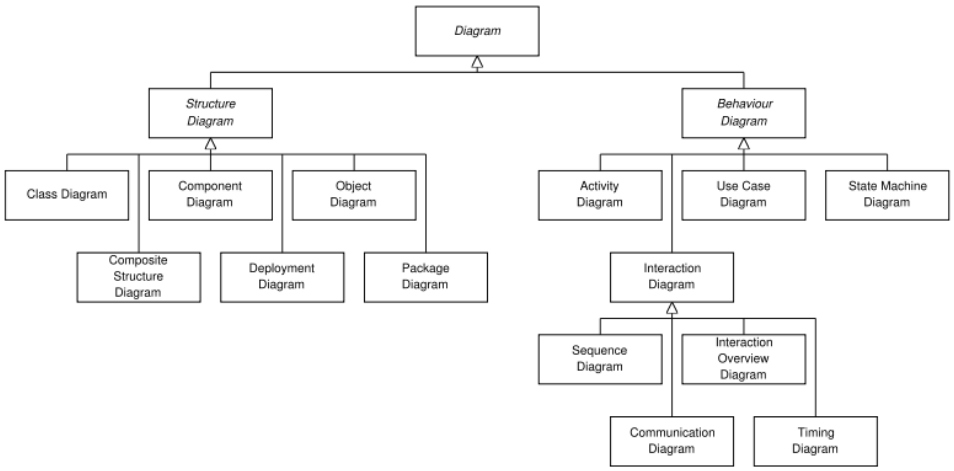

Figure 2.11: Different types of UML diagrams.

directions. These three different directions are following: The structure diagrams, which is used to explain a systems structure. These types of diagrams do not show how each object work during run-time but they explain where the different objects are placed in relation to each other.

Behavior diagrams shows what must happen in the system being modeled. This is mostly used to get the big picture of a program and does not explain any data flow or internal variables.

Interaction diagrams are like a behavior diagram with extended information. It explains the dataflow in the model, like how they call each other and what data they exchange during that.

Class diagrams, which is a diagram related to The structure diagrams are used to show our structure later on. Furthermore we will explain the interaction between a user and our program by a Sequence diagram.

Below is a brief description of how the diagrams work, which will increase the understanding of our diagrams.

2.3.4.1 Class Diagram

This diagram shows the relations between the different classes that we have created. It also shows which type of relationt and it will always be one of these 3 relations:

Aggregation which is a relation that can be used where you have 1 class which uses 0 to unlimited numbers of the other class. The relation will there fore have '1' at the one side and '0..*' at the other side. It is represented by a unfilled diamond.

Composition which is a stronger variant and only have '0..1' or '1..1' on each side. It is used mostly where you have like a car which is in composition with a carburetor '0..1' to '1..1'. This means that if there is a car it needs a carburetor. This relation is represented by a filled diamond.

Generalization indicates that one class is specialized from the other. This is used when you want to extend or maybe build further on a class. An example will be the `JFrame` class which contains a GUI for a program. It is then possible to extend the GUI and build on top of it, meaning that you will save the code of writing the base GUI your self.

2.3.4.2 Sequence diagram

This diagram shows the interactions between different objects. In our case it will be between the classes and the database. Each object will be represented as a horizontal line. A box on the line means that the object is in use and the arrows between the boxes are the relations in which the data is transferred. A dotted line is a connection to a notation, and this does only have extra information about a certain point in the diagram.

2.3.5 Databases

In this section we will describe the basics of computer databases. We cover the the following topics; relational databases, database engines and SQL.

The source used in this section is [16].

When handling large amounts of data, it is often a good idea to store it in a database. A database is a collection of information organized in such a way that a computer program quickly can obtain desired pieces of information. When using a database to store data it is possible for more than one program to change and store data at a time. There are different types of database models, but we will only cover the relational database model, since it is the most used database model.

2.3.5.1 Relational Databases

Relational databases consists of relations in the form of tables, meaning that the content of a table is related to other tables. A table consists of columns and rows. The named columns are called attributes. These attributes then have a set of values they are allowed to take which is called their domain. The domains are set by selecting a data type. Relational databases adhere to three basic rules. First, the ordering of the columns in the table does not matter. Second, each row can only contain a single value for each of its attributes. Third, there cannot be identical rows in a table. Therefore each row in a table has a unique identifier called the primary key. Below is an example of how a relational database is structured.

This picture illustrates two tables: User and Group, from a relational database. The top line of both tables contain the attributes and the other lines contain the data. The tables are interconnected in the way that Group_id in User refers to

Table 2.1: User table

Id	Name	Age	Group_id
1	Chris Tucker	37	2
2	Maria Kristensen	29	1
3	Jeffrey Stallman	46	2

Table 2.2: Group table

Id	Location	Department
1	Denmark	Sales
2	USA	Development

the Group table. This makes it easier to append data to many users, like in our example information about location and department, by just including one more attribute in the User table.

Table 2.3: User table

Attribute name	Data type
ID	Integer (automatic incremation)
Name	Char
Age	Integer
Group_id	Integer

The image above shows how the attributes in the User table are set. The first column sets the name of the attribute and the second column contains the information on which data type the data is stored as. The first row, Id, is set as a primary key and therefore the data type is set to be integers with automatic incrementation. When a new entry is made the table gives that entry a unique Id by automatically incrementing the Id from the last entry. This is done to uphold one of the rules for relational databases, which is that there cannot be identical rows in a table.

2.3.5.2 Structured Query Language (SQL)

SQL is a language used for retrieval and management of data in a relational database. SQL is a language used to interact with the SQL database. SQL can be used in two ways, interactively and pre-programmed. When using SQL interactively the user issues an immediate command, in the form of an SQL statement, to the database and gets an immediate response according to the statement. This method is mostly used by developers. The other way to use SQL is preprogrammed SQL. Here the SQL is embedded into an application written in another language.

Listing 2.1: SQL statement example

```
1 SELECT * FROM User WHERE Group_id = 1
```

This SQL statement translates into: select all the data from the table User where the attribute Group_id equals 1. The `SELECT` command returns the data that match the requirements stated in the `FROM` and `WHERE` commands. The other parameters tells which tables and attributes that are used.

Chapter

3

System Development

3.1 Process

3.1.1 Introduction

During the process some different problems and challenges arose. Different problems ranging from using a new (to us) system development method to problems implementing theory into our program. Looking back on our developing phase we had quite a few discussions about how to do different things. We chose the following problems for further discussion because we found them interesting and essential to the understanding of our process creating this project.

- Our XP process.
- Fictional customer.
- Static teams.
- Pathfinding Algorithm.

Several other problems could be described and discussed. We have the following suggestions:

- GUI map scaling.
- Third dimension and multiple floors.

3.1.2 Our XP process

In this section we describe how our XP project cycle went from start to finish. This section serves to make an overview of how our system development process went, and then in the following sections we will cover some of the practices and problems in further detail.

Our XP process differ a bit from the sample process we described in the theory chapter. The main difference is that we have not performed any acceptance tests nor have we used unit tests consistently. In the beginning of the project we discussed performing usability tests on our system and make them our acceptance tests, but prioritized further development of the system over this so we excluded that part. Not all teams have executed test driven development due to a lack of experience. Even though we did some refactoring on the code after each iteration, it did not harvest all it's benefits. This was because the teams refactored their own code and did not get the other developers opinion on it. We will now describe how our process went from start to finish. Below is an overview of our cycle.

1. Write user stories.
2. System role model analysis (KRAK).
3. Conduct spike solution for algorithm.
4. Split user stories into tasks.
5. Create a release plan.
6. 1^{st} and 2^{nd} iteration.
   (a) Iteration planning.
   (b) Choose new user story.

   (c) Implement the simplest solution that could possibly solve the problem.
   (d) Refactoring.
   (e) Small release.

7. System development process stopped.

We started our development process by writing some user stories that would serve as the requirements for a pathfinding system ordered by a fictional costumer. At first we wrote one user story for each function the system should contain but later found out that we should have a story that described how the user saw themselves use the system. With this in mind we rewrote the user stories and ended up with six stories where the first described the basic use of the system and the other five described extra features. Later in this chapter we describe the user stories in further detail.

Before we started our release planning we made a system role model analysis and a spike solution. Because our system would be pretty much like KRAK we decided not to waste too much time "reinventing the wheel", and instead make an analysis of KRAK and see how they have chosen to design their pathfinding system. The spike solution we made served the purpose of finding out exactly how hard it would be to implement a pathfinding algorithm. Both the KRAK analysis and spike solution are described in further detail later in this chapter.

After we had made our release plan we started to plan which tasks we should implement in the three iterations. Here we made the mistake of splitting the user stories into tasks. We based our decisions on what we needed to implement the first user story. The three teams got tasks for three iterations and the programming commenced. A detailed description of the development can be found in appendix A. Here we describe step-by-step what problems arose when coding and how we tackled them.

After the second iteration we decided to stop developing on the system and instead concentrate on writing the report. We had successfully implemented the core of the system that allows the user to find the shortest path between two locations. We had also completed the Map editor so that it was possible to maintain maps by editing locations and paths. And because these essential tasks were completed we decided to prioritize writing this report over adding extra features to the system.

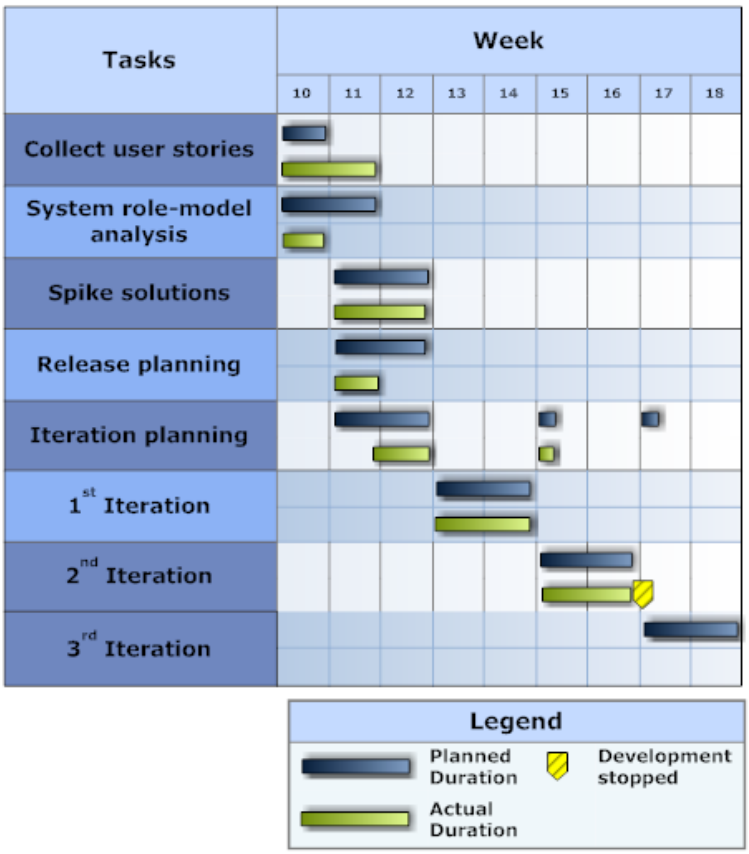

Figure 3.1: Xp project progress

Figure 3.1 shows a Gantt-chart of our XP cycle, it shows that our planning was somewhat successful. Creating the user stories took more time than we had originally planned out. This was due to a misunderstanding in what they should actually contain. We stopped development after the 2^{nd} iteration because we did not have time to complete the third iteration. We choose to focus more on writing this report than adding more features to the product.

3.1.3 The Fictional Customer

In XP there are several roles that has to be filled, and one of the most important roles is the customer. The costumer provides the initiating problem for the entire development process and he provides feedback along the way as the development progresses. As you can read in the XP section, this is essential to the entire XP concept. In fact communication and feedback are some of the core values in XP. As our prototype system was not a commercial product, we did not have a customer, so we had to imagine one for XP to work.

Our system prototype was based on the idea of making a program that would make it easier to navigate at the university, so logically our fictional buyer had to represent the university.

3.1.3.1 The Customer's User Stories

The fictional customer could not provide us with user stories, because he did not exist. However the user stories are essential for the process to work. The way we solved this was by imagining, and any other case in which the customer had to act, we took the role of the customer. We wrote six user stories, imagining that the customer wanted his system to be, in a non-technical language, that the customer would use.

3.1.4 User Stories

We made up six user stories that serve as our requirement specification. The first user story explains the basic usage of the system and how the users sees themselves using the system. The rest of the user stories cover different functions that the system should contain.

1. The user starts by choosing a starting point. The default starting point is the position of the information stand, but it should be possible to choose an alternate starting point. Afterwards he/she chooses a destination.
2. The user should be able to choose from different perspectives like speed, length, and route difficulty (eg. stairs).
3. The route should be displayed in a way that shows the route from start to finish to give a better overview for the user.
4. The route overview should be resizable on the screen allowing the user to inspect it in detail.
5. There should be a printer connected to the system so the user can print the complete route.
6. For maintenance purposes the map should be easy to edit for the administrator.

We have divided the user stories into separate tasks and then handed these out to the three different teams. These tasks are described in further detail in appendix A. We have handled the user stories slightly different than it is usually done in XP. After we had created the user stories, we divided all of them to the tasks. Then we divided those tasks to the programming pairs. If we had followed XP correctly we had taken one user story at the time, accorded to how they were prioritized. That user story would then have been divided to tasks and only if there would not be enough work for everyone, we would have moved on to the next prioritized user story. How we divided the tasks is shown in tables in section 3.1.7

3.1.5 Spike Solution

To better estimate and to cope with the whole project we needed to make a spike solution. The biggest obstacle in this project, were the pathfinding algorithm. This was a perfect candidate to spike.

3.1.5.1 Choosing Algorithm

To spike the path finding, we needed to pick an algorithm. Wikipedia[25] introduces six graph search algorithms, for finding the shortest path. Without much further investigation we decided to implement the A* algorithm, in short, it is like Dijkstra (which we learned in our *Discrete mathematics* lectures) but has an estimate (heuristic) with the direction towards the destination. Because it has an estimate it should speed up the process, because makes the algorithm avoid going in "wrong" directions. If the direction points to a dead end path, A* will go back and therefore computing unnecessary paths, but still less paths than Dijkstra, which computes in all directions. See 2.1.1 .

3.1.5.2 Implementing A* Spike

The spike should be simple, and only show a proof of concept. So the simplest possible solution that shows the concept must be chosen. Instead of developing an application that shows real maps, real points and pathways, we made a simple solution just with blocks. Here a rectangular block can represent a pathway or a wall, and a single block is chosen to represent the start block, and same for the finish point. This solution shows, that we can get a pathfinding algorithm to work, which finds a way through the blocks.

One of the great things of using this approach is that, it is easy to turn a block into a starting point (marked by the letter A) or into a wall (marked by grey blocks), and this too makes it really easy to show the found path, simply marking the blocks that make up the path, in our case white.

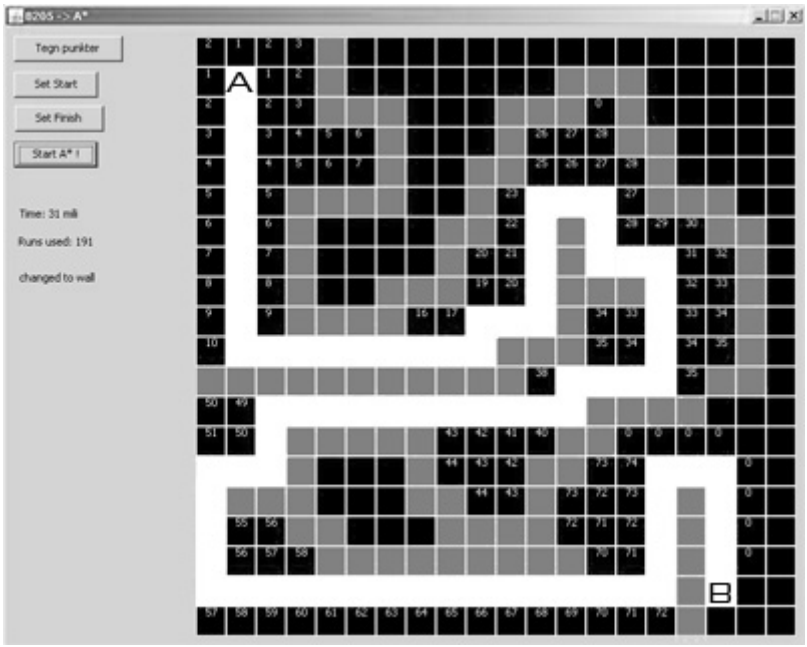

Figure 3.2: Screenshot of spike

Figure 3.2 is an example of the working application. It shows the path finding algorithm in action, finding way from start (A) to finish (B) through the maze of

walls (grey). As the figure shows, the block solution is very simple, but illustrates the concept. In the actual program the starting point is green, the destination point is red, wall is blue, open space blocks are black and pathway blocks are yellow.

3.1.6 Rolemodel Analysis

As a part of our XP cycle, we planned to perform a rolemodel analysis to learn from existing experiences, so we do not "reinvent the wheel".

We have chosen the online pathfinding service, at `http://www.krak.dk`, because it is a well known danish service, that average more than 1.7 million users per month. One reason, that Krak is well known in Denmark, is that Krak started their pathfinding service already in 1770, as a light paper "pathfinder handbook" for the 80.000 citizens of Copenhagen[4]. Krak have kept up-to-date and evolved from small paper version of Copenhagen to high-tech online pathfinding search engine, so they too have a great deal of experience in the area, which makes them perfect as a rolemodel for our project.

Krak has many similarities with our project, and furthermore, one might say that Krak is our rolemodel. Krak delivers users to our location, where it is intended that our system comes in handy, and guides the users to their final destination within our location. Therefore it is an advantage that our way of doing and structuring things, will resemble the model of Krak, so our users already are familiar with the general concepts.

3.1.6.1 Target Group

Kraks target group is roughly the same as ours, with the obvious exception that the geological aspect of course eliminates the bigger part of Kraks user base. The average user of Krak, is familiar with basic use of computers and Internet.

3.1.6.2 Start- and Destination Selection

Using Krak, one has to search for both start and destination locations, by giving an address with a city or zip code. Krak will then match this against its huge database of vertices, this works well because of the huge number of vertices, it would be impossible to simply pick from a list.

3.1.6.3 Route Visualization

Krak visualizes its route in two ways, one way is to simply display written instructions on how to navigate, by writing "Turn left after 200m on "Universityroad", and possibly displaying the estimated time for the action. The other way Krak does route visualizing is by drawing the route on a map, from the start location to the finish location.

3.1.6.4 Print

Krak offers the availability to print both its instructions and its visualized map, so the user easily can print out the route and bring the map along.

3.1.6.5 Rolemodel Conclusion

Krak has excellent searching capabilities, but when having a small location to cover, as we do, this is not necessary. We think that a more light-weight solution is better,

like a filtering list.
Krak has a good print version of the visualized route, which too is perfect for us, as people would often like to print out their directions. Krak offers two outputs, visualized route and textually described route. Textual description is great on roads, because roads has names, which hallways does not have. This then makes a textual description almost useless in our context, so we should therefore only provide a visualized output.

3.1.7 Release Planning

We planned our project aiming for at least two iterations. We also planned the 3rd and final iteration, however we did not have enough time to carry it out. Our iterations are planned, as follows.

3.1.7.1 1st Iteration

Date range for the first iteration is set for Tuesday the 25th of March to Monday the 7th of April

Table 3.1: Tasks for **First iteration**

Priority	**Team**	**Task**
1	A	Study Algorithm.
1	B	Setting up GUI.
1	C	Setting up GUI.
2	A	Construct classes.
2	B	Extend classes.
2	C	Fetch Data.
3	B	Fetch Data.
3	C	Represent Map.
4	B	Point selector.

3.1.7.2 2nd Iteration

Second iteration is set for Tuesday the 8th of April to Friday the 18th of April

Table 3.2: Tasks for **Second iteration**

Priority	**Team**	**Task**
1	A	Implement algorithm.
1	B	Call algorithm.
1	C	Add data.
2	A	Test algorithm.
2	B	Validate user input.
2	C	Delete data.
3	C	Edit data.

3.1.7.3 3rd Iteration

The third and last iteration is set for Monday the 21th of April to Friday the 2th of May

Table 3.3: Tasks for **third iteration**

Priority	**Team**	**Task**
1	A	Implement toilet extension.
1	B	Print functionality.
1	C	Validate user input.
2	B	Toilet GUI.

3.1.8 Static Teams

We had static teams during this project, because some of our tasks were very close related. For example the tasks:

- Study algorithm
- Implement algorithm

These two tasks would be very difficult to split op. This would require a lot more time and communication.

Another good side effect of having static teams is, that we get some highly specialized people working in their areas. They get the time to focus and concentrate on their subject, without worrying about other parts of the application.
One problem that this raises, is that the little communication, causes some unnecessary work. For instance, because two of our teams did not communicate properly, they did more or less the same work. The teams handling the GUI and Map Editor, have some shared work. They both needed to represent the map, with vertices and edges. But still, both teams did this work.
This might too have been affected by the fact, that the teams did not always develop at the same time, nor in the same room. This of course has the same negative effects on the inter-team communication.

3.1.9 Algorithm Progress

This section will deal with development of the path finding algorithm. We will account for our choices regarding writing an algorithm capable of fulfilling the user story of finding the shortest path between two locations.

3.1.9.1 Spike

Before planning the iterations we had to study graph theory and path finding algorithms for better estimate how long it would take to implement a path finding algorithm. Researching different algorithms we ended up focusing on A*, because it seemed perfect for our project. A* have some advantages, mainly its speed and accuracy. If implemented correctly it would be faster and still as accurate as Dijkstra. We then wrote a little spike-program testing the A* algorithm. We discovered that implementing A* was manageable in a reasonable amount of time. The spike tested A* on a flat graph like the graphs in the A* section (see section 2.1.2.4). We saw the algorithm return the shortest path every time even in situations like the one in the figure 2.6 on page 22. This convinced us to implement A* in our project.

3.1.9.2 First Iteration

To make sure our algorithm was working we decided to setup a testing environment first. We determined that a small test independent of all other parts of the program

would comply with our needs. We drew a test graph where we fairly easy could calculate the shortest paths manually. According the graph theory section (page 15) we created vertex and edge classes to represent a graph. These classes could be used to test and would also be the final classes for the project.

3.1.9.3 Second Iteration

Having set up all things needed to implement the algorithm itself, we wrote the path finding class. While testing our small test graph the algorithm seemed to work fine on the graph at first. But then we began to edit our test graph, we added and subtracted cost for some of the edges which caused the heuristic to not always work as intended. We changed the test because the plan was to add cost for an edge that for example represented a staircase, and subtract the cost for a broad straight corridor. Another thing was that we had planned to do path finding spanning over more than one floor of a building. A flat graph could be made to represent multiple floors, but we came up with test graphs where our heuristic function would overestimate and sometimes make our algorithm end up with a wrong route, longer than the shortest possible. These flaws in the heuristic function would have taken quite some time to fix. A whole new heuristic function probably had to be written if things like multiple floors and varying costs should be taken into consideration.

We came to the conclusion that we did not have enough time to implement a working heuristic function. We fixed it by removed the heuristic from the algorithm. This would turn our algorithm into Dijkstra and we concluded that it was enough to fulfill the user story of calculating the shortest path.

3.2 Product

3.2.1 System Platform

3.2.1.1 Java

Java is an object oriented and cross-platform programming language developed by Sun Microsystems, released 1995. The syntax is similar to C++. The philosophy behind Java is, "Write once, run anywhere". All Java code is compiled into bytecode (a kind of stage between source and machine code). Any Java virtual machine can compile the bytecode into executable machine code, on the operating system and architecture which it is running. This allows Java programs to be executed on a wide variety of architectures and operating systems.

In 2007 Sun Mircosystems announced that Java will be available as open source and free under the GNU General Public License (GPL).

3.2.1.2 Swing

Swing is Java's graphical user interface toolkit. The Swing libraries can be used to make everything needed to create a graphical user interface, buttons, lists, windows, etc. An advantage with Swing is, that it is platform independent (like the rest of java), a button made with Swing is programmaticly made the same, but is rendered accordingly to the native visual layout of the operating system below. This helps the developer avoid making different user interfaces for different operating systems Swing is the successor to the Java AWT toolkit. Swing were needed because AWT had some serious shortcomings, one of them where that AWT used

the native window system to display its components, which meant some components where lacking on some platforms, and therefore too lacked some of the more advanced components like Trees and Grids, because they where not common to all platforms. Because of the mistakes from AWT, Swing takes another approach, and all the components are implemented directly in Java and "emulates" the native looks of the operating system. This approach also opens the possibility to develop custom "look and feel"s. Such a graphical theme is called a "look and feel". This also gives the freedom to force an application to display like native Mac OSX on all platforms. Another important feature of Swing, is the use of "Layout managers", clever components to arrange graphical components correctly, adjust their size and positioning. This becomes important when the user resizes the window, the layout manager will adjust all the components on-the-fly [12].

3.2.1.3 SQLite

SQLite is a small self-contained SQL database engine. It is basically a file that contains both the database and the database engine. SQLite does not have a separate server running as most SQL Systems do, instead SQLite is an embedded database, meaning that the server itself is in the file. So SQLite only consists of a single file, which makes it perfect for use in desktop applications, embedded systems and mobile devices. SQLite supports almost all of the SQL92 standard features. SQLite is free, its not only open source, its in "free domain", meaning that the code has absolutely no copyright restrictions at all, and therefore can be used fully in all kinds of projects, commercial too [5].

3.2.2 Description

3.2.2.1 Architecture

We start by showing a database diagram and then use UML (see page 44) to show the structure of our program by creating a UML Class Diagram and show how our program works by creating a UML Sequence Diagram.

Database Diagram

There are some basics which must be covered before being able to understand a database diagram and this is explained in section 2.3.5.

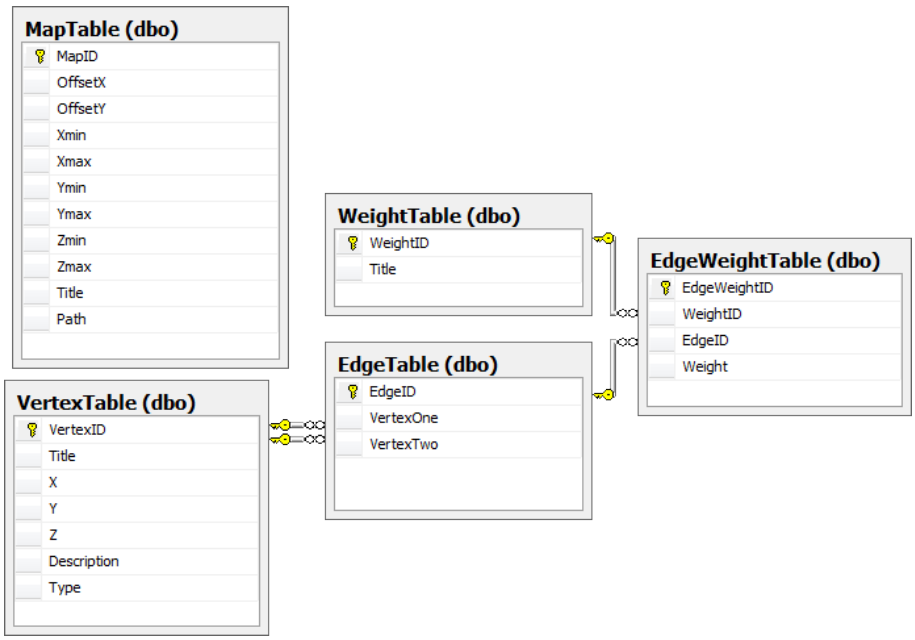

Figure 3.3: Diagram of our current database

The tables above are explained below:

VertexTable contains all vertices. X, Y and Z are used as coordinates and Title, Description are self descriptive. Type is an integer and can be: 1 for pathway, 2 for room and 3 for toilet. EdgeTable represents the edges between the vertices and consists of the vertices it connects, VertexOne and VertexTwo. One vertex can of course be connected to several other vertices.

WeightTable contains the title for each weight type. An example would be "normal", "wheelchair", which each have its own need of edges between the vertices. The idea is to have a table which controls the title (group) instead of writing the title on all the edges directly. In this way we can easily change a title and delete a whole weight group without worrying about any mis-spellings.

EdgeWeightTable contains a relation to WeightID and EdgeID. It also contains a weight. The relation makes it possible to choose which groups have a weight and even decide different weights for each group to the same edge. For instance it would be harder for physically challenged people to use stairs. In that way the edges weight could be 14 for normal users and 20 for wheelchair users to ensure the path is the fastest.

MapTable was the table we were supposed to use for placing maps below the Vertices. The idea were to create maps which overlap each other, but this is a complicated process. Our product currently only supports the build-in map.

Class Diagram

A class diagram shows the different classes and how they relate. This is both regarding inherit and include; however, we have only inherited from excisting sys-

tem classes. The two types of relation used is described in section 2.3.4.1 and the third type 'Generalization' is only used to system classes and is therefore not on the diagram.
Below are the diagrams three diagrams. The first diagram shows the overall picture of packages without any classes in it. The two others shows the classes of the program and the mapeditor. The descriptions below each diagram briefly describe the content of the diagram so it is understandable in the big picture. There is a lot of unnecessary details left behind which is necessary to make the program work; however, it will only confuse in this case.

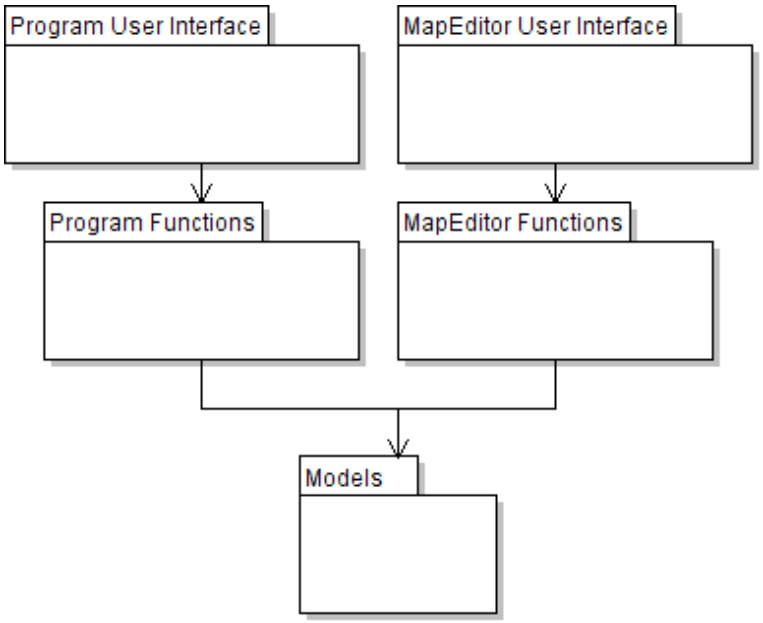

Figure 3.4: Overview of Packages

Each package represents one or more classes which is related to that package. They are created to place the classes in groups which makes it easier to read and understand the influence of the classes.

Program User Interface and **MapEditor User Interface** are the two uppermost packages. The classes within these extend from the JFrame and create the interactable interface for the user.

Program Functions and **MapEditor Functions** contains all the classes which contains functions. These classes are the ones which works as a engine for the program and calculate everything needed for the program to run.

Models contains the classes which works as a model. It means that these classes are mostly aggregated and contains no functions. They only work as a bunch of variables collected in one place. The model layer is used in both the mapeditor and the program but the classes within are only written once.

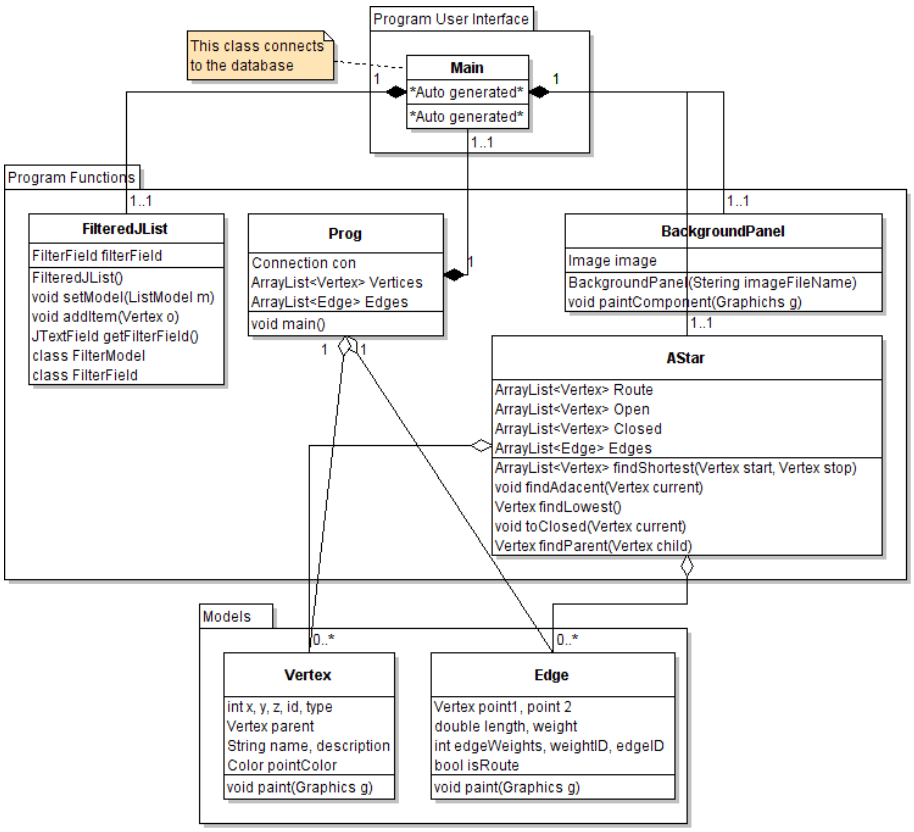

Figure 3.5: Zoom on the Program Packages with visible classes

Prog is the class which starts the program. It has an aggregation to both `Edge` and `Vertex` and got a composition to `Main`. The aggregations is used to store data in which it loads from the database and the composition is to the class which contains the GUI it displays.

Main is a class which inherits from `JFrame`, meaning that it will be displayed as a GUI. The code of this is generated from a designer view. It got a composition to both `AStar`, `FilteredJList` and `BackgroundPanel`.

FilteredJList is used for sorting lists of data. It sorts the data while the user types which gives a more dynamic flow.

BackgroundPanel contains the background picture.

Astar got a aggregation to Edge and Vertex which it uses to store information within while calculating a path. This is the class where the graph search algorithm is implemented.

Vertex and **Edge** contains a lot of variables and us used to store data within.

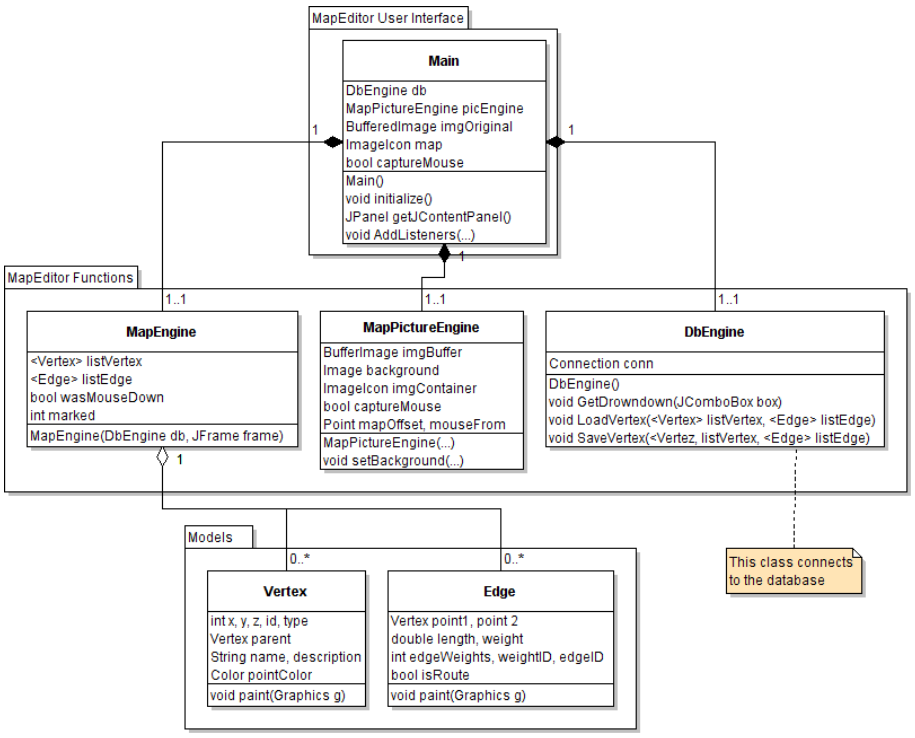

Figure 3.6: Class Diagram of MapEditor

Main has a composition to `MapEngine`, `MapPictureEngine` and `DbEngine`. All content could in theory be moved into Main but is moved out in order to get some structure and move functions like database connections out in its own class.

DbEngine controls the connection to the database and is basically a class where all activity, which must be handled between the program and the database, is stored.

MapPictureEngine controls some of the GUI and handle all the calculations between the map and the pointer. Basically this class works like a desktop which controls the registration of the mouse and displays the background image.

MapEngine is a composition of `Main` and it has an aggregation to both `Vertex` and `Edge`. These are from 0..*, meaning that this class can contain an array of both vertices and edges. This class is mainly used to control the activity on top of `MapPictureEngine`. In short terms it controls the vertices and edges.

Vertex and **Edge** contains a lot of variables and us used to store data within.

UML Sequence Diagram

A user diagram shows what is going on the interactions of the user. The following phase shows a user who starts the program, selects a location and a destination, and finds the path between it.

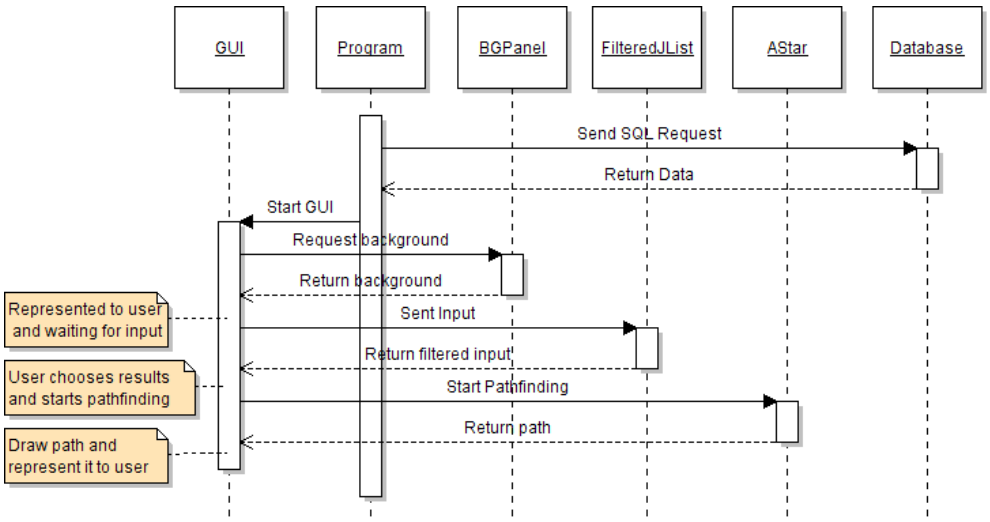

Figure 3.7: UML Sequence Diagram

Program is the class which is being initialized when the program started.

Send SQL Request sends an command specified in SQL language which asks for all the vertices and edges in the database.

Return Data Returns the data to the Program where it is stored in two arrays.

Start GUI Initializes the GUI which the user sees and loads the `BGPanel` (BackgroundPanel) to show the background of the program

Send Input and Return filtered input means that the user has started to type something and the filtering of the results below the text field has begun. The GUI uses `FilteredJList` to handle the sorting but the GUI does not use any actual path finding yet. It only reduces the list of choices in location/destination and present these to the user.

Start Pathfinding runs when the user has chosen some of the filtered results. This will include the A Star class and use this to find an actual path between the location and destination.

Return Path returns that data from the A Star and the GUI then draws it in the display.

3.2.2.2 User Interface Design

This section contains two subsections. One, which describes usability in short terms and another which will point out how we have used usability and what we might improve. The description will be briefly since the usability are not one of our major aspects. We use this section to evaluate our product because we do not have any costumer that could perform the acceptance tests required by XP.

Usability In Short Terms

Usability can in short terms be divided in five different areas: learnability, efficiency, memorability, errors and satisfaction. The source of this is [28].

Learnability is about how fast a new user can learn to use the program. Both regarding to the state where the program is usable and to the state where the user knows all the functions. The most obvious way to improve the learnability is to place the different sections correctly. When a person starts a program he will normally begin to look in the upper left corner, since that is where you start when you read. Then the eyes will follow the GUI of that point and borders are therefore preferred. The interactions, which have to be done first, will therefore logically fit in the upper left corner. Furthermore learnability can be increased by having popup descriptions to the different functions of the program.

Efficiency is not about how easy it is to use but about how effective it is to use when the user have learned the interactions of the program. This is mostly about optimizing processes which makes the typing and interacting faster.

Memorability is about how hard the program is to remember to use after a while. How fast the user can reestablish the proficiency. This is mostly done by creating logic icons that makes it easier to remember functions of a program.

Errors is about looking at what errors users make while using the program and how these can be prevented in the future. What made the user do something wrong to begin with and how can this be prevented for other users.

Satisfaction is about how pleasant and successful the use of the product is. This is mostly done by animations and graphically good looking interfaces. It could also be extended with sound which can interact with other senses.

Usability In Our Program

The program is based on three different sections; selecting section, the action section and the result section. The following drawing shows the three different parts and each part will be explained further.

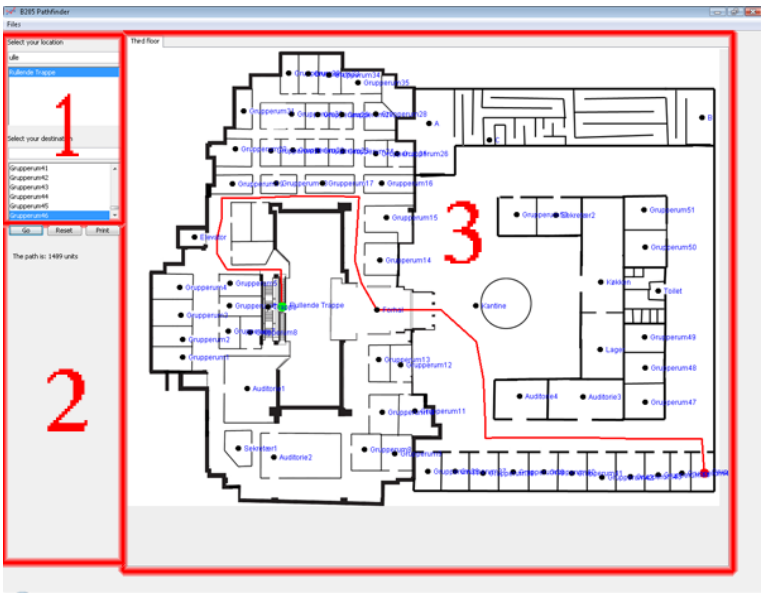

Figure 3.8: GUI divided in three sections

Part one is the selection section where the user chooses start and destination location. It is placed in the upper left corner with the idea that the user will pay attention to the section first. This is done to achieve easy learnability and prevent errors, since this is the most logically place to start interacting with a program. When the user starts typing in the first text field the list of possible selections will be filtered using the typed text. This means that the user does not have to scroll through all the possible locations to find the one he is looking for. This greatly increases the efficiency because it makes the application faster to use. It also increase the satisfaction since the program seems more dynamically when it automatically filters.

Part two is the section below the selection section and is used for the actual path finding. It consists of three buttons; One is a 'Go' button which means the program will start finding the path from the location to the destination, then display it. The next button is 'reset' which resets the location and destination selections then clears the display for a possible previous drawn route. The final button supposed to print a map of the path. The learnability in this area is not the best, since the Go do not explain what will happen and it does not indicate that it is used to find the path between the two selected locations. The efficiency could be improved by enabling pressing 'enter' in the destination field will result in the same as a 'Go' click. The memorability is pretty good since those buttons are the only ones. Furthermore the 'Go' button generates an error messages that notifies the user if a wrong or non-logic selection has been made. This is not the most satisfying area since the only way you can see something is happening is by seeing the line change on the map in section three.

Part three is the place the result of the pathfinding will be displayed. There is a floor tab, but since we only have one floor this section cannot be interacted with. This can actually be misleading since all the vertices are shown and looks like they can be clicked on. The only satisfaction here is to see that the algorithm is really working and enjoying the green dot at the location, red dot at the destination and the red path in between.

Map Editor

Our main idea was to create an editor where you could use the mouse to position the vertices and edges. Below is described how the editor is used.

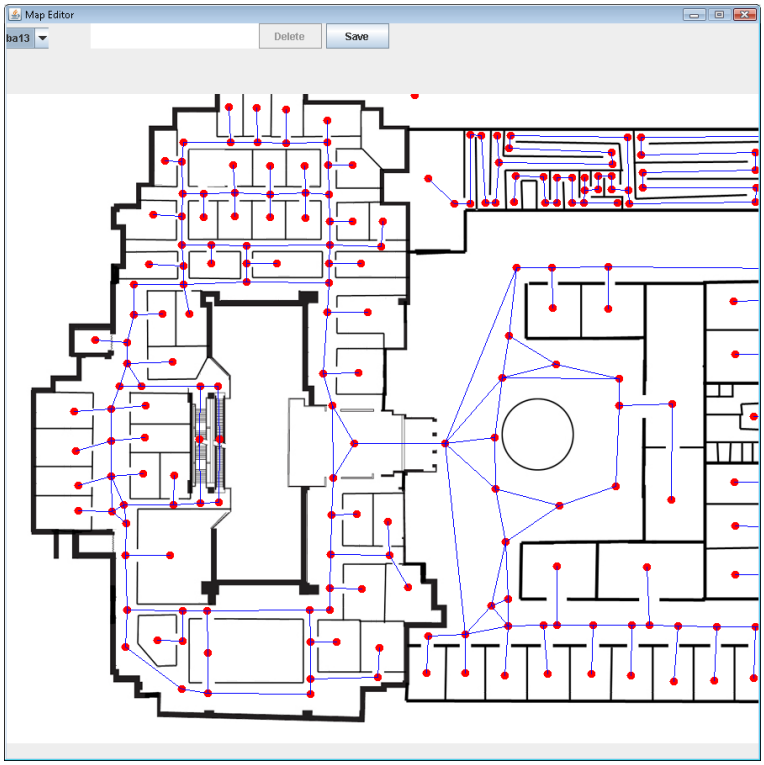

Figure 3.9: Map Editor

There is a drop down menu in the top left of the program. This contains the different maps which is in the database, but currently this only contains our main map. A text box and two buttons with the text 'Delete' and 'Save' are next to the drop down menu. The text box and 'Delete' button is for the vertices and the 'Save' button saves all the content to the database. Below is the map with a lot of red dots and blue lines between them. Each red dot represents a vertex and each line represents an edge. If a vertex is marked it will be shown as green instead of red.

Right Mouse Button handles the movement of the map. The map is in full scale which means that it is too big for the program window. The right mouse button let us hook to the map and drag it around, making it possible to drag the hidden parts of the map to the the view area.

The **Left Mouse Button** is used for many things. It can create vertices, edges and mark a vertex for editing or deleting.

How To

- Create Vertex
  If no vertex is marked, a new vertex can be created by left clicking in a empty area.
- Mark Vertex

A vertex can be marked by clicking on it. The color of the vertex will then turn green and the text box and 'Delete' button in top of the program will turn from gray to normal, indicating that they can be used.

- Unmark Vertex
  A marked vertex can be unmarked by left clicking on it again. The color of the vertex will then go back to red.
- Delete Vertex
  The 'Delete' button in top of the program will be clickable when a vertex is marked. Notice that all the related edges to the marked vertex will be deleted as well.
- Change title on a vertex
  The text box in top of the program will contain the current title of the marked vertex. Notice that the title will not be saved until the vertex is unmarked by clicking on the vertex again.
- Change the type of vertex
  The map editor only supports pathway and room at the moment. If the title is empty, the vertex will be considered a pathway. If not, the vertex will be considered a room with the given title.
- Add Edge
  The vertex that the edge begins from must be marked before it can be edited. Clicking on another vertex will add an edge to this vertex. Notice that the marked vertex will change to the vertex the edge was going to.
- Remove Edge
  If you try to add an edge between two vertices which already have an edge between them, the current edge will be deleted. An edge will also be deleted if one of the vertices it is attached to is removed.

3.2.3 Demarcation

This section is about what we excluded from our program and what compromises we took in order to get the program done within our deadline.

1. Database related issues
   (a) We made some compromises in relation to our database. In the beginning we only created one row of data in `WeightTable` and `EdgeWeightTable`, making all the edges have the same weight. This was of course not meant to be that way but a combination of misunderstandings and the fact that this was already coded into the program made it harder to correct. In case the product is developed any further, this is one of the errors which must first be corrected in order to secure the correctness of the path.
   (b) Our program does not contain a solution for handling routes spanning over multiple floors. Even though the algorithm in theory could find a path on a graph that covers multiple floors, our GUI cannot display this.
   (c) Since we only operate with one map and one floor, there is no reason to use and integrate the `MapTable`. The idea behind that table and how the map should be able to overlap each other have not been developed at all since it is not necessary for the demo to work. Again, this must be developed in order to make the program work, but this depends on item B.

2. Feature issues

   (a) Another feature that does not work is the print button. This is very visible since the print button is already there but does not do anything when you click it.

Chapter

4

Reflection

4.1 Our XP process

4.1.1 Inexperienced Developers

When developing using XP it is important that the teams use the practices agreed on and upholds the values of agile system development to gain the advantages it brings. In our case we did not have the proper experience to use unit testing consistently throughout the development phase. We found it hard to write unit tests to code that we did not know exactly how would work before we started actually writing it. The unit tests we have written all came after the functional code was written but worked out great none the less. Due to the lack of unit tests we lost vital feedback from the system that may have given us more courage when choosing between safe or more drastic solutions.

Because this was our first time using XP, or any structured system development method for that matter, we also had some difficulties adjusting to all the many new principles and practices. One of the hardest things was maintaining the courage to follow XP all the way. This resulted in the teams not always just implementing the simplest solution but instead thinking forward and making more in one iteration than originally planned.

We could have utilized XP more if we had appointed a person to serve as XP coach or project manager. A such person should guide the teams through the development phase and make sure they applied the XP practices correctly.

4.2 User Stories

4.2.1 Problem With a Fictional Customer

As one can imagine, there are quite a few problems with having a fictional customer in the XP method. As mentioned in the XP section, some of the core values in XP are feedback and communication, which was both difficult in our case. When we acted as the customer in relation to the creation of the user stories, we already had an idea of how the product was supposed to look and act. We did not really get the non-technical customers perspective on this, which could have made our user stories very different. We had a very good idea of what we thought was possible, but a real customer, might have have had other ideas, which would have had to be discussed.

By not having a customer we lacked feedback on the product during our development process. This is especially a problem since agile development thrives on customer feedback. It is the only outside influence, that corrects the developers along the way. We imagine that the product would have been very different with a real customer input. Especially in terms of usability.

4.3 Static Teams

As mentioned in section 3.1.8, our teams have been static throughout the development phase. This has some advantages and disadvantage, that we will try to discuss in this section.

The single, most important, argument for changing teams, is communication and knowledge sharing. Why does changing teams promote knowledge sharing? Simply

because it forces developers to share knowledge to the new developer too. This gets even more promoted with changing tasks in XP, when the original developers of, for instance a module, are not destined to be the same people, when a new task demands new features for this module. So it is essential that, the developers communicate and share their knowledge.

In software developing companies, rotating teams are very useful. First of all, this improves the overall knowledge of all individual developers, improving their skills in bug fixing other developers code, because they all have some knowledge of whats going on in all areas of the project. This is not, just on a project level, with lifting the overall developer skills and knowledge, this comes to good use in all future projects too. And therefore, of course, makes more value of the developers, for the company. There might be some social aspects to this too, if the developers get to know one another better. This might too be in the interest of the company.

Is there any reason not to change teams, with all the good arguments presented? The single biggest argument for this is time. It takes up time to change teams. When changing teams, members are not guarantied to work on the same areas from turn to turn, and therefore new members needs to do the needed research, to fully understand the problem and code in order to continue developing it.

When writing a university project like this, where the main goal is education, why would we not want to change teams? We decided that we did not have the time needed to do this, and because we had some tasks, that where tightly related, for instance splitting the following tasks between to different teams would require a lot of extra work, and to some extent, repeat much of the first task.

This being an university project, we need to write a report on the subject, and in this way we actually do our knowledge sharing within the report. By keeping static teams, our groups get highly specialized, this has some advantages in our case. When writing the report, we gain the high accuracy demanded. It is important that the author of a section, is very well informed on the subject, as we do not want any incorrect information in the report. On this basis, we concluded that it was best to keep static teams for our project.

We got some bad experience with static teams, because we did not communicate properly. For instance two of our teams wrote duplicated code. The GUI team and the Map editor team, had similar tasks, “Setting up GUI”. Because of bad communication both teams made their own code to show the map, vertices and edges.
This has probably also been influenced by the fact, that the teams were not forced to develop at the same time, and might even develop from different buildings. If this was improved, it could have had a high impact on our team communication.

Large and international companies have similar problems, because they often have departments in different locations, and international companies have even more problems, based on the fact that departments in different countries often speak different languages. This complicates communication even further.

In the last few years, the term "outsourcing" has become increasingly popular amongst companies, partially because of the new communication that the Internet provides, the lower salaries in educated development countries and the high employment rates in western countries [11].

4.4 Our Path Finding Algorithm

During the process of developing the path finding algorithm we ran into some challenges. The main issue was our rather inaccurate heuristic function. Getting the estimate by calculating the distance between points in a coordinate system is only useful on a planar surface where the cost equals the distance between two points. When selecting algorithm we did not take multiple floors and varying costs into consideration, because we based our decision upon the spike solution. We might have focused too much on getting a both fast and accurate algorithm. In a project like this where the algorithm only is required to find a single path and the graph have relatively few vertices and edges. Even if we had implemented an accurate heuristic function the speed increase would not have any significance. We should have focused on accuracy alone.

Failing our tests and nearing the iteration deadline we were forced to not use time on the perfect heuristic function. This way we might have avoided wasting time on a complex task with little payoff. For the algorithm to handle multiple requests at once, creating the perfect heuristic would however payoff in the long run. If we had used a traditional development method we could risk running into dead ends like this. By using XP we had the courage to cut the heuristic and keep the algorithm simple and still end up with a working product. Doing that turned our algorithm into Dijkstra.

4.5 Perspective

In this section (we pretend) the product is fully developed, with all features as described in the Product Description, see page 57.

As a university project, our product is not intended to be economically profitable. But if we would turn this project into a commercial one then we would have to consider:

- Does the product have scalability?
- What are the cost of developing a product like ours?
- What are the expansion-abilities for this project?

The scalability of the product is huge, because of the enclosed map editor. This map editor makes it more convenient to expand a building and thereby having a new floor-plan loaded into the program, or having a room divided into a number of smaller rooms or halls, so new paths occur.

We have no economical interests in this project. We did not get paid, neither did we use any commercial developing tools. For a firm developing the exact same project, the economical aspect changes. There would be salaries to the developers to consider and maybe even money spent on licenses to development tools. Furthermore there would be some sort of implementation expense in the form of computers the software is meant to be executed on. For gaining profit, the software company

would charge the buyer at least (and often more) as much as the firm itself have spent on the project, unless the firm has a special interest in developing this project.

For our program as an idea or as a proof of concept, it can be expanded in several ways. For instance the idea of a guidance inside buildings being merged together with already existing pathfinders could end up being a great innovation. We could expand our product to quickly guide people inside hospitals, where time is of the essence. Another possibility of expansion could be to use the program from a hand-held device making real time navigation and path finding possible in building.

4.6 Conclusion

We achieved the goal of creating a program that can find shortest paths inside a building. We learned how to apply graph theory to a pathfinding problem and implement a working graph search algorithm. Throughout the process of developing the program we found that the Dijkstra algorithm (A* without a heuristic) was a very suitable solution for implementing in our program. All things taken into consideration, Dijkstra fulfills our requirements and was manageable to write in the time we had. If we had more time however, a full A* algorithm with a good heuristic function would be preferred.

We tried to use XP as the development method in this project, but we did not apply the XP process to its full extent. We found pair programming to be very efficient and it helped us write code of high quality. However we had some problems with communication between groups but our planning went well, except that we had to skip the third iteration because of time trouble. The negative things we encountered may have been avoided if we had more experience, both in XP and in programming in general. Assigning an XP coach or project manager might have helped us follow the XP process more strictly and thereby utilize the strength and advantages to a fuller extent.

Agile development methods affects customers and developers in different ways. It affects the costumer in a positive way, by giving him opportunities to change the requirements, even at a late state in the project. The negative side for the customer is that he does not get a fixed deadline. For the developers the positive side is that they get to focus more on the code than documentation and active feedback from the customer during the development. This helps the developers avoid writing unnecessary code and keeps the project on the right track. We found no negative aspects for the developers when using XP aside from adjusting, understanding and learning the process, in a project like ours.

We have successfully found a possible solution for our initiating problem, which is easing navigation in the Basis department at AAU, using software.

Terminology

All definitions are from Wikipedia.org.

Artifact
The term artifact in connection with software development is largely associated with specific development methods or processes e.g., Unified Process.

System Development Model/Method
We use both model and method but it is the same.

XML
The Extensible Markup Language is a general-purpose specification for creating custom markup languages.

Discrete mathematics
Discrete mathematics, also called finite mathematics or decision mathematics, is the study of mathematical structures that are fundamentally discrete in the sense of not supporting or requiring the notion of continuity. Objects studied in finite mathematics are largely countable sets such as integers, finite graphs, and formal languages.

Bibliography

[1] The agile manifesto.
http://agilemanifesto.org/principles.html.

[2] Apache ant.
http://ant.apache.org/.

[3] Declaration of interdependence.
http://pmdoi.org/.

[4] Kraks historie.
http://www.krak.dk/Information/OmKrak/Historie.aspx.

[5] Sqlite website.
http://www.sqlite.org.

[6] K. Beck. *Extreme programming explained - Embrace change*. Addison Wesley, 2000.

[7] D. J. B. et.al. *Objects First With Java: A practical introduction using BlueJ*. 2003.

[8] M. Lange. 12 practices.
http://www.xpexchange.net/english/intro/practices.html.

[9] P. Lester. A* pathfinding for beginners.
http://www.policyalmanac.org/games/aStarTutorial.htm.

[10] O. Mentor. Junit.
http://junit.org/.

[11] . Mikael Lindvall, Dirk Muthig. *Agile Software Development in Large Organizations*.

[12] Niemeyer and Knudsen. *Learning Java*. O'Reilly, 2005.

[13] A. J. Patel. Amit's notes about path-finding.
http://theory.stanford.edu/~amitp/GameProgramming/
AStarComparison.html#S2.

[14] I. Per Kroll. Key principles for business-driven development. http://www-128.ibm.com/developerworks/rational/library/oct05/kroll/.

[15] A. Peter Nørkjær. *AAU i tal: Bruttokvadratmeter*.

[16] I. t. S. Rick F. van der Lans. *Introduction to SQL: Mastering the Relational Database Language*. 2006.

[17] K. H. Rosen. *Discrete Mathematics And Its Application*. 2007.

[18] A. Scott W. Ambler. Managers intro to rup. http://www.ambysoft.com/downloads/managersIntroToRUP.pdf.

[19] Softhouse. Scrum in five minutes. http://www.softhouse.se/Uploades/Scrum_eng_webb.pdf.

[20] R. Software. What is the rational unified process. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jan01/WhatIstheRationalUnifiedProcessJan01.pdf.

[21] S. C. W. Vinekar, Vishnu and A. Nerur, Sridhar; University of Texas. Can agile and traditional systems development approaches coexist? *Online Publication*, 2006.

[22] Wikipedia. Chrysler comprehensive compensation system. http://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System.

[23] Wikipedia. Glossary of graph theory. http://en.wikipedia.org/wiki/Glossary_of_graph_theory.

[24] Wikipedia. Ibm rational unified process. http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process.

[25] Wikipedia. Shortest path problem. http://en.wikipedia.org/wiki/Shortest_path_problem.

[26] Wikipedia. Shortest path tree. http://en.wikipedia.org/wiki/Shortest_path_tree.

[27] Wikipedia. Unified modeling language. http://en.wikipedia.org/wiki/Unified_Modeling_Language/.

[28] Wikipedia. Usability. http://en.wikipedia.org/wiki/Usability/.

[29] Wikipedia. Waterfall model. http://en.wikipedia.org/wiki/Waterfall_model.

[30] A. P. Wilson. Extreeme programming with ibm visualage. http://www.ibm.com/developerworks/library/it-aprcc01/index.html.

Appendices

Appendix

A

Process

In this section we explain how our development proceeded, while following the XP model and what the result was.

A.1 Introduction

Certain things cannot be learned just by reading about them; we believe that system development is one of these things, so we developed our prototype software system using the XP process model. To follow the XP-model certain roles had to be filled, so we pretended that the university was a buyer interested in buying a pathfinding system for their buildings and we were a software firm hired to develop this system. The process progressed as follows:

1. We began by forming the teams. As we were six people in total, we made three groups with two members each.
2. Then we created a series of user stories, as if they were written by the fictional buyer.
3. We split the assignments from those user stories into tasks, which were prioritized and divided.
4. A spike solution was made, in which we integrated a path finding-algorithm into a working sample program.
5. Then we analyzed.
6. And made a release plan.
7. And finally we started the iterative process. We completed two full iterations before ending the development.

A.1.1 Iteration Planning

To make developers adapt to the programming habits in XP, there are some useful tools to support the way of thinking XP. These tools makes the process of programming much easier by doing automated build or by testing existing code. We used

two commonly used tools for XP, which were JUnit and ANT which was described in the XP chapter.

A.1.1.1 Tasks

Team A which is responsible for the algorithm has the following tasks:

Table A.1: Tasks for **Team A**

Priority	**Task**	**Description**
1	Study algorithms	Carefully study the shortest path algorithms, and choose a good candidate to implement.
2	Construct Classes	Make classes to represent Vertices and Edges.
3	Implement algorithm	Make a implementation of the algorithm chosen. The algorithms should work on the defined classes
4	Algorithm test	The algorithm should be extensively tested with test data to ensure that the algorithm works as expected (note, this is not UnitTesting)
5	Implement toilet extension	Implement a extension to the algorithm, that finds the shortest path to a toilet (multiple endpoints).

Team B which main responsibility is the graphical user interface, for the end-user, have:

Table A.2: Tasks for **Team B**

Priority	**Task**	**Description**
1	Setting up GUI layout	Design the general GUI layout in Swing components.
2	Extend classes	Extend Team A's classes, to make them represent a graphical object. And show the vertices and edges on a map.
3	Fetch data	Fetch all data from the SQL database to a internal list, and add all the objects to the graphical map.
4	Point selector	Make a start point list selector, and a end point selector. Both should be search/filter-enabled.
5	Call algorithm	Call the algorithm with the fetched data, to produce a graphical representation of the generated pathway.
6	Validate user input	Validate all user input, to ensure that all input is valid and cant crash the application. And give user feedback if the start and end points are the same, for instance.
7	Print	Make a print functionality, so the pathway can be printed out.
8	Toilet function	Make GUI to interact with a toilet finding extension.

Team C which main responsibility is the graphical user interface, for the end-user, have:

Table A.3: Tasks for **Team C**

Priority	**Task**	**Description**
1	Setting up GUI layout	Design the general GUI layout in Swing components.
2	Fetch data	Fetch data from the database, into the classes already used by team A and B.
3	Represent map	Make a graphical representation of the map, with all the current vertices and edges.
4	Add	Make it possible to add both new vertices and edges.
5	Delete	Possibility to delete both vertices and edges.
6	Edit	It should be possible to edit existing vertices and edges, both rename and move.
7	Validate user input	Make user friendly feedback input is missing etc.

A.2 1^{st} Iteration

This section describes how the different teams solved their tasks in the first iteration.

A.2.1 Team A

There was two tasks planned for this iteration. Select an algorithm and create vertex and edge classes to contain a two dimensional graph in the program. We studied different graph-search algorithms and found one suited for this project. We focused on Dijkstra and A* (see section 2.1.2.2 and 2.1.2.4). The main goal for Team A was to find the the shortest path possible and at first we concluded that if we could implement A* and create a good heuristic function, this would be the optimal solution. In addition to finding the shortest route our program would be able to run faster.
We then started creating the Vertex and Edge classes. The idea was that every edge object should contain the two vertex objects it connects in the graph.

- Vertex
  The most important variables of the vertex class are: the G, H and F-score used to store the path walked and estimated distance to the destination, the Parent variable which holds the parent vertex object and x and y-coordinates. The coordinates would make the vertices easy to draw in the GUI and also give the heuristic function something to work with. We created a function to add the G and H score to get the F-score. This would make it easier to create the main algorithm later.

- Edge
  Important variables of the edge class are: Of course the two vertices it connects and the length of the edge that will be used to calculate the G-score. We defined the length as the mathematical distance between the two vertices using:ht of the edge that will be used to calculate the G-score. We defined the length as the mathematical distance between the two vertices using:

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

A.2.2 Team B

There where four tasks appointed to Team B for this first iteration, the first iteration we started working on where the setting up the general graphical layout of the application. This task where done mostly with use of the Eclipse plugin Visual Editor, a plugin enabling developers to drag'n'drop components in, to visually form the desired layout of the application. This would otherwise have been a quite large and tedious task, with writing all the graphical components by hand.

Another task where to extend Team A's classes, to be used in combination with Swing. Here the priority became important, because we could not have done this, if Team A have not made the classes in the first place! So because we started with the general GUI setup, this allowed Team A to work on the classes.

The most important line, for making the classes "graphical" is to extent a base swing component.

```
1 public class Edge extends JComponent {...
2 ...
3 public class Vertex extends JComponent {...
```

Then we need to overwrite the `Paint()` method, that Swing calls to paint the object. In this way, one can say that the object "paints" itself. Which comes in handy, as will see later. Here we see the `Paint()` method from the Vertex class:

```
1         public void paint(Graphics g) {
2                 if (type != PATHWAY) {
3                         super.paint(g);
4                         g.setColor(pointColor);
5                         if (pointColor == Color.black) {
6                                 setBounds(x - 5, y - 5, 15 + name.
                                      length() * 7, 15);
7                                 g.fillOval(0, 0, 10, 10);
8                                 g.setColor(Color.BLUE);
9                                 g.drawString(name, 15, 10);
10                        } else {
11                                setBounds(x - 8, y - 8, 21 + name.
                                      length() * 7, 16);
12                                g.fillOval(0, 0, 16, 16);
13                                g.setColor(Color.BLUE);
14                                g.drawString(name, 21, 10);
15                        }
16                }
17        }
```

The above class does several things; first we only paint the object if its anything else than a pathway, we do not want to paint "hidden" verties, such vertices can arise when a hallway is making a turn, then a "hidden" vertex is needed to make the curve. It would not make sense to paint those.

Line 4 sets the color of this vertex, this color is set by different commands such as `setAsFinish()`, `setAsStart()`, `setAsRoute()` and lastly `reset()`. The colors represent the vertex type.

Line 6 checks that, if the vertex is color black (then its a normal pathway vertex) then we draw it as a Oval shape with dimentions 10x10. Otherwise its a Start or End vertex, those need to be somewhat bigger, 16x16.

The `setBounds()` function is a swing function to set the swing-componets drawable size, here we take the length of the name into account, ensuring that the drawable area is large enough for the name text to fit, and small enough so the tooltip do not appear at wrong places. The name is drawn with the swing function `drawString()`.

We encountered a wierd problem, when working with the classes, at first we could simply not get them drawn at the right places - or not at all. Thes turned out to be our own (Team A) functions, `getX()` and `getY()` that overruled our `setBounds()` for positioning the component. We have then had to refactor the functions, to be named `getVertexX()` and `getVertexY()`.

Our next task where to fetch the vertex and edge data from the database. The SQLite database provides JDBC drivers. So the task is then trivial. Loading the SQLite JDBC driver and connecting to the database (the database is called "Map.sqlite"):

```
1 Class.forName("org.sqlite.JDBC");
2 conn = DriverManager.getConnection("jdbc:sqlite:Map.sqlite");
```

Example of fetching data, and saving the fetched data to new instances of our Vertex class:

```
1  Statement stmt = conn.createStatement();
2  ResultSet rs = stmt.executeQuery("SELECT * FROM VertexTable");
3  while (rs.next()) {
4          Vertex v = new Vertex(rs.getInt("VertexID"),
5                                rs.getInt("X"),
6                                rs.getInt("Y"),
7                                rs.getInt("Z"),
8                                rs.getInt("Type"),
9                                rs.getString("Title"),
10                               rs.getString("Description"));
11         Vertices.add(v);
12 ...
13 }
```

We simply create a new SQL statement, and executing the SQL statement `SELECT * FROM VertexTable`, which means that we want to fetch all vertices from the database. Then for each record in the database we create a new Vertex object, with all the fetched data given in the constructor to the new object. Then lastly on line 12, adding the new object to our internal list of vertices.

The make the point selector, we wanted to have to select boxes, and a input field at the top of each, to be used as filtering fields. Being lazy programmers, we used the class `FilteredJList` from the book Swing Hacks (O'Reilly) by Joshua Marinacci - Hack #14.

Then its easy to change the `Jlists` that we placed with Eclipse Visual Editor with `FilteredJList` because `FilteredJList` extends `Jlist`.

A.2.3 Team C

Task 1 is to create the GUI and task 3 is about representing the map. These separates a bit from task 2, fetching data from the database, so Task 1 and 3 is in one section and Task 2 is in another section.

A.2.3.1 Task 1 and Task 3

This task is about creating the UI. Basically this is about creating a frame which can contain the whole program itself and make a interactive surface for the user. We decided to create the design static, which basically means that we can place the buttons everywhere we want. Furthermore we wanted to build an engine which we could just add classes to, allowing the Map Editor to be further developed.

Main Class
We created a Main class which extends from `JFrame`, meaning that it will be opened as a form. The form will have some properties which is wise to set so we have gathered these in a void named `initialized()` which is started in our static main.

```
1 private void initialize() {
2         this.setSize(900, 900);
3         this.setResizable(false);
4         this.setContentPane(getJContentPane());
5         this.setTitle("Map Editor");
6 }
```

`setSize(900,900);` does so the form resizes to 900 * 900 pixels.

`setResizable(false);` does so the form cannot be resized and will be at the fixed size of 900 * 900 pixels.

`this.setContentPane(getJContentPane());` calls a function which returns `JPanel` which specify the borders of the frame.

setTitle("Map Editor"); sets the title of the frame. This is the title which can be seen on the top left border of the program.

MapEngine Class
Since the map can vary in sizes, we have to make a way where we can move the map. We decided to make the left mouse button handle all the actions and the right button to handle the map. We made an x, y offset so the map could be drawn different without changing the actual x, y scale. This means that if the map is drawn from (0, 0) it will still be drawn from (0, 0) while moved. The only difference is the offset which changes when holding right mouse button down and move the mouse. In this way we can edit on a map even through it is several sizes bigger than the 900 * 900pixels the forms contains of.

A.2.3.2 Task 2

This task is about connecting to the database. The program connects to the database in the same way as the connection made in `Team B`. The whole data fetch is placed in a class called `DbEngine`. When the class is initialized it connects to the database and makes sure the connection is working. Furthermore it contains three functions which each are specified to fetch data to different controls and save data. In this way all the fetching/saving from the database is written one place which makes it easy to debug and edit. The three functions are the following:

`GetDropdown(JComboBox box)` fetch a list of maps from the database and fill it into the `JComboBox`. The function is working; however, we only use one map which makes this function unnecessary for the demo to work. The `JComboBox` can is the

class of a dropdown box.

`LoadVertex(ArrayList<Vertex> listVertex, ArrayList<Edge> listEdge)` fetch all the vertices and edges from the database and fill it in to the two arrays: listVertex and listEdge.

`LoadVertex(ArrayList<Vertex> listVertex, ArrayList<Edge> listEdge)` basically deletes all the data about vertices and edges and fill the database with the new data from listVertex and listEdge.

A.3 2^{nd} Iteration

This section describes how the different teams solved their tasks in the second iteration.

A.3.1 Team A

The goal of this iteration was to create and test the main algorithm of the project. The first thing we considered was the input needed by our algorithm. We concluded that instead of a vertex-list and an edge-list it was enough to get a edgelist that contains edge-objects. Because all edge-objects contains vertex-objects, the vertex-objects can be reached through the edge-objects. First we created the first loop. A function that will take the start and destination-vertex as input and return the shortest path between these as an arrayList. The structure is similar to the pseudo-code in the Dijkstra section (2.1.2.2). The following snippet is the primary loop:

```
1  public ArrayList<Vertex> findShortest (Vertex _start, Vertex
       _destination) {
2          Open.clear();            //The lists are cleared
3                  Closed.clear();
4                  Route.clear();
5                  Open.add(_start);
6                  Vertex current = null;
7                  //The primary loop
8                  while(!Closed.contains(_destination)) {
9                          current = this.findLowest();
10                         this.toClosed(current);
11                         this.findAdjacent(current);
12                 }
13                 this.Route.add(_destination);
14                 while(!this.Route.contains(_start)) {
15                         this.Route.add(this.findParent((Vertex)(this.
                               Route.get(this.Route.size()-1))));
16                 }
17                 return this.Route;              // Returns the
                       shortest route.
18         }
```

First the function clears the `Open`, `Closed` and `Route` lists. This is because the same instance of the pathfinder object can be used more than once in the program. This way we make sure the algorithms starts out correct. We also set the current vertex variable to null so that this variable is not loaded from the previous run. The function then continues to add the start-vertex to the open list. This is where the algorithm starts the search. The function will then start the while loop and run through it until the destination has been found and added to the closed list. The while loop invokes three functions:

- current = this.findLowest()
  This line will search through the open list and set the current-vertex to be the vertex with lowest score in the open list. The first time this is run the current-vertex is the start-vertex. The findLowest() function looks as follows:

```
public Vertex findLowest(){
        Vertex lowestF = ((Vertex)(Open.get(0)));
        if (Open.size()>1) {
                for (int a = 1 ; a <= Open.size()-1 ; a++) {
                        if ((lowestF.getF() >= ((Vertex)(Open.get
                            (a))).getF())){
                                lowestF = (Vertex)Open.get(a);
                        }
                }
        }
        return lowestF;
}
```

  It starts by setting the first vertex in the open list to be the lowest. It then cycles through the list and compares all vertices to the lowest one, replacing the lowestF every time the current one is lower.

- this.toClosed(current)
  This moves the current-vertex from the open list to the closed list because it have been checked, or will be in the next line.

- this.findAdjacent(current)
  We made this function to do a few things. First of all it finds all the vertices that the current-vertex is connected to via an edge and add them to the open list. It then runs through the open list and if it finds a vertex with a higher F-score than the F-score it can get by going through the current-vertex, then the current-vertex will be set as parent for the vertex. It then updates the H, G and F-scores to fit the new path. The following is the `findAdjecent()` function. Note that about half of it is omitted. This is because every vertex can be either point1 or point2 in an edge. Therefore it is required to check for both these situations.

```
public void findAdjacent(Vertex _current){
 for(int a = 0 ; a < edges.size() ; a++) {
  if(((Edge)(edges.get(a))).getPoint1().equals(_current)) {
   if(!Closed.contains(((Edge)(edges.get(a))).getPoint2())){
    if(Open.contains(((Edge)(edges.get(a))).getPoint2())){
     if(((Edge)(edges.get(a))).getPoint2().getF() > ((Edge)(edges
         .get(a))).getPoint1().getF() + ((Edge)(edges.get(a))).
         getLength()){
      ((Edge)(edges.get(a))).getPoint2().setParent(_current);
      ((Edge)(edges.get(a))).getPoint2().setG(((Edge)(edges.get(a
          ))).getPoint1().getG() + ((Edge)(edges.get(a))).
          getLength());
      ((Edge)(edges.get(A))).getPoint2().setH(Math.sqrt(((Math.
          pow((((Edge)(edges.get(A))).getPoint2().getDX() -
          _destination.getDX     ()), 2)) + (Math.pow((((Edge)(
          edges.get(A))).getPoint2().getDY() - _destination.getDY
          ()), 2)))));
      ((Edge)(edges.get(a))).getPoint2().setF(((Edge)(edges.get(a
          ))).getPoint2().getG() + ((Edge)(edges.get(a))).
          getPoint2().getH());
     }
    } else {
     Open.add(((Edge)(edges.get(a))).getPoint2());
     ((Edge)(edges.get(a))).getPoint2().setParent(_current);
     ((Edge)(edges.get(a))).getPoint2().setG(((Edge)(edges.get(a
          ))).getPoint1().getG() + ((Edge)(edges.get(a))).
          getLength());
```

```
16        ((Edge)(edges.get(A))).getPoint2().setH(Math.sqrt(((Math.
              pow((((Edge)(edges.get(A))).getPoint2().getDX() -
              _destination.getDX     ()), 2)) + (Math.pow((((Edge)(
              edges.get(A))).getPoint2().getDY() - _destination.getDY
              ()), 2)))));
17        ((Edge)(edges.get(a))).getPoint2().setF(((Edge)(edges.get(a
              ))).getPoint2().getG() + ((Edge)(edges.get(a))).
              getPoint2().getH());
18        }
19     }
20    }
21    if ((((Edge)(edges.get(a))).getPoint2()).equals(_current)) {
22    //This code is omitted.
23    }
24   }
25  }
```

The first For loop cycles through all the edges of the graph, and if an edge contains the current vertex it continues to check with the other vertex that is connected to the edge and therefore also with the current vertex (We will call the other vertex the next vertex). The function then checks if the next vertex is in the closed list. Because if it is, it has already been checked thoroughly. If the next vertex is not in the closed list it can be two things: not in the open list or in the open list. We are now at line 6.

If the next vertex is in the open list, the function will check if the path through the current vertex is shorter than the one the next vertex already has. If the F-score through the current vertex is lower than the one the next vertex has the function will set the current vertex as parent for the next vertex and update the G,H and F-scores of the next vertex.

If the next vertex is not in the open list, the function will not check for shorter paths (Line 12). It will just add it to the open list, set the current vertex as parent and update the G,H and F scores.

Our heuristic function was as simple as the function that calculated the length of the edges.

When this is done the function continues to check if the other vertex in the edge is the current vertex. This however is a bit meaningless since the current vertex can only be either point1 or point2. As mentioned the code for this is very similar to the code described above. The only exception is that point1 and point2 are switched all the way through.

We tested our algorithm by drawing a small graph and add the corresponding vertices and edges to an empty `Arraylist`. A object was created from our `Astar` class and we invoked the function `FindShortest()` with two points. It seemed to work fine on the graph at first. But when we began to add and subtract cost for some of the edges the heuristc did not always worked that well. We did this because the plan was to add cost for an edge that for example represented a staircase, and subtract the cost for a broad straight corridor. Another thing was that we had planned to do pathfinding spanning over more than one level of a building. A flat graph could be made to represent multiple levels but we made some examples where our heuristic function would overestimate and sometimes make our algorithm end up with a wrong route, longer than the shortest possible. These flaws in the heuristic function would have taken quite some time to fix. A whole new heuristic function probably had to be written if things like multiple levels and varying costs should be taken into consideration.
We came to the conclusion that we did not have enough time to implement a working heuristic function. Instead we removed the heuristic from the algorithm, making

the H-score zero at all times. This would turn our algorithm into Dijkstra, but we concluded that Dijkstra was enough to fulfill the user story of calculating the shortest path.
Removing the heuristic function would make our algorithm a bit slower, but since the algorithm is only required to run once, we did not see this as something that would ruin our program. Dijkstra would be able to handle a graph with multiple layers and varying costs.
Using this way of system development caused us to not waste time on writing a complex but working heuristic function and still deliver working code at the end of the iteration.

A.3.2 Team B

We had two main goals for this iteration, the first was to invoke the algorithm, and display the found route on the screen.

```
1  AStar PathFinder = new AStar(Prog.Edges);
2  ArrayList<Vertex> route = new ArrayList<Vertex>();
3  route = PathFinder.findShortest(start, end);
4
5  for (int i = route.size() - 1; i > 0; i--) {
6          Vertex f1 = (Vertex) route.get(i);
7          Vertex f2 = (Vertex) route.get(i - 1);
8
9          boolean isFound = false;
10         Iterator<Edge> it = Prog.Edges.iterator();
11         while (!isFound && it.hasNext()) {
12                 Edge e = it.next();
13                 if (e.getPoint1().equals(f1) && e.getPoint2().equals(
                       f2)) {
14                     e.setAsRoute();
15                     routeLength += e.getLength();
16                     isFound = true;
17                 } else if (e.getPoint1().equals(f2) && e.getPoint2().
                       equals(f1)) {
18                     e.setAsRoute();
19                     routeLength += e.getLength();
20                     isFound = true;
21                 }
22         }
23 }
```

First we create a instance of the `AStar` pathfinding class, the constructer takes a list of Edges it should work on. Then we "start" the algorithm with the function `findShortest(start, end)`, the start and end, are the start vertex selected, and end, is the end vertex selected. This function returns all the vertices that make up the route, so the only thing left, is to paint this route on the map.

Line 5 starts a for-loop that iterates over all the vertices that make up the route. We save the current vertex in a local veriable called `f1` and the next in `f2`.
Line 11 starts a while-loop that iterates over all the Edges we have in the application. We need to do this, to find the edge that connects `f1` and `f2`. We save the current Edge in a local variable called `e`. Then we test if `f1` and `f2` are the end points for `e`, if they are, then we call `setAsRoute()` on the edge object `e` that will paint the edge visible on the screen and we set the `isFound` boolean to true, indicating that we can break out of the loop.
The other task on our iteration, where user input validation. We have placed our validation in the action handler, so when the user clicks the start button, the first thing that happens is validation, before proceeding.

```
1  getResetButton().doClick();
2
3  if(startList.getSelectedValue() == null){
4          JOptionPane.showMessageDialog(getJFrame(), "Please select
               starting point!");
5          return;
6  }
7
8  if(endList.getSelectedValue() == null){
9          JOptionPane.showMessageDialog(getJFrame(), "Please select
               destination!");
10         return;
11 }
12
13 if(endList.getSelectedValue().equals(startList.getSelectedValue())){
14         JOptionPane.showMessageDialog(getJFrame(), "You have arrived
               at your destination!");
15         return;
16 }
```

The first line resets the map for any earlier stages, clearing start- and endpoints, and end routes.

Line 3 tests that a start point is selected, this is mandatory. If not, then we make a popup message saying "Please select starting point!". We do the same for end point validation.

Line 13 tests that the start and end points are not the same, if so, theres no point in using the application!

A.3.3 Team C

We had three tasks to complete in the second iteration. Last iteration we made the GUI, made it possible to fetch data from the database and represent a map in the GUI. The next functionalities we need to implement are adding, deleting and editing vertices and edges. Our first idea of solving this problem was by creating two different methods, one that would add new vertices and edges and one that would edit or delete them. Shortly we realized that it would be easier to just have one method that could do all three actions. When we fetch data from the database we get all the vertices and edges that are linked to that particular map. And since we have all the data loaded into `ArrayLists` we can simply just delete it from the database and then save the `ArrayLists` to the database when we are done adding, editing or deleting. We will now explain how we implemented this method we call `SaveVertex`.

```
1  public void SaveVertex (ArrayList<Vertex> listVertex, ArrayList<Edge>
       listEdge) {
2          try
3          {
4                  Statement stmt = conn.createStatement();
5                  //first part
6                  stmt.executeUpdate("DELETE FROM VertexTable");
7                  for(int i = 0; i < listVertex.size();i++)
8                  {
9                          Vertex v = listVertex.get(i);
10                         int type = 1;
11                         if(v.getName().length() > 0)
12                                 type = 2;
13
14                         String sql = String.format("INSERT INTO
                               VertexTable (VertexID, Title, X, Y, Z,
```

```
                                Description, Type) VALUES
15                                          (%s,'%s',%s,%s,%s,'%s',%s)", i
                                                 + 1, v.getName(),v.getX()
                                                , v.getY(), 1, v.
                                                getDescription(), type);
16                          stmt.executeUpdate(sql);
17                  }
18                  //second part
19                  stmt.executeUpdate("DELETE FROM EdgeTable");
20
21            for(int i = 0; i < listEdge.size();i++) {
22                  Edge e = listEdge.get(i);
23                  String sql = String.format("INSERT INTO EdgeTable (
                        EdgeID, VertexOne, VertexTwo, EdgeWeightID) VALUES
                         (%s,%s,%s,%s)",
24                                              i + 1, listVertex.indexOf(e.
                                                   getPoint1()) + 1,
                                                   listVertex.indexOf(e.
                                                   getPoint2()) + 1, 1);
25            stmt.executeUpdate(sql);
26            }
27  }
```

The `SaveVertex` method takes two `ArrayLists` as arguments, one for vertices and one for edges, and returns nothing. Since we are handling database connections we have chosen to insert the entire function into a try-catch block that catches possible database errors. If such an error should occur the user will get a message containing the exact error details. The method has two parts that are basically alike, the first part handles the `ArrayList` containing the vertices and the second part the edges. Because of this we will only describe the first part.

We start by creating a statement object called stmt for the sql commands we need to perform by using the `createStatement` function from the `conn` SQLite connection object. Then we delete the vertices in the database so it is empty and then there will be no more than one instance of each vertice. When the vertice table is empty we run a for loop that inserts the vertices from the `ArrayLists` one by one. For each vertex in the `ArrayList` one sql statement is executed on the database. The SQL statement contains seven different parameters that are taken from the Vertex object v except for the Type which is set according to the length. The for loop continues until there are no more vertices in the `ArrayList` and then the second part begins. In the second part the exact same thing happens, this time is just the edges that are stored in the database.

A.4 3^{rd} Iteration

This third iteration were never actually accomplished. We ran out of time in the project, and had to start writing the project report, due to newly scheduled courses late in the semester that where not planned or scheduled before.
The tasks planned for this iteration can be seen in 3.1.7.3 on page 54.