Battleship

- Design and Implementation

Computer Science – 1st module – University of Roskilde

Project supervisor: Mads Rosendahl
Student: Søren Loft Andersen
Student id: loft@ruc.dk

Resumé

Dette projekt omhandler udvikling af et Sænke Slagskibe spil i programmeringssproget Java. Ambitionerne er at udbygge forfatterens kendskab til Java programmeringssproget, samt at opøve en vis færdighed i strukturering af programmel og formidling af dette.

I rapporten undersøges først eksisterende udgaver af spillet, hvorefter specifikationerne for dette projekt udarbejdes. Dernæst analyseres og diskuteres mere konkrete muligheder og alternativer, alt imens programmelstrukturen gradvis opbygges. Derefter gennemgås den kode der udvikles til spillet, både på et overordnet, grafisk plan (vha. skemaer og diagrammer osv.), men også på et mere kodenært niveau. Slutteligt er den udviklede kode genstand for en afprøvning, som dokumenteres med skærmbilleder af programmet.

Projektrapporten konkluderer, at der findes mange mulige måder at strukturere programmellet på, og at der er fordele og ulemper ved dem alle. Ligeledes konkluderes det, at projektet i sin helhed har været succesfuldt, i den forstand at de i indledningen formulerede mål og ambitioner er blevet opfyldt.

Abstract

This project aims to develop a Battleship game using the Java programming language. The main ambitions are to develop the author's knowledge about Java, and, to some extent, learn about designing software and communicating about it.

In the report existing versions of the Battleship game are examined, after which the specifications for this Battleship game project are developed. Then specific options and their alternatives are analysed and discussed, along with the build-up of a feasible software design for the Battleship game. Next, the actual code developed for this project is examined. This is done both graphically (with schemas, diagrams etc.) and with code examples. At last the developed software is subject to a test run, documented with screenshots from the program.

The report concludes that several possible and feasible software design options exist, and that these all have advantages and drawbacks. It is also concluded that the project as a whole has been successful, as all of the formulated goals and requirements have been met.

Contents

List of figures

List of tables

1 Introduction

The purpose of this project is to explore the basics in the Java language, including its data structures and graphical components. The purpose is *not* to define and develop new and original software or academic knowledge; hence a game of Battleship serves as a case study, for exploring the basics of software design and programming in Java. By using a simple game as a case study, the focus quickly arrives at software design, structure and code, thus skipping the art of defining *all* software features from scratch – a time consuming process.

The intended reader for this report is probably a BA student, doing a 1st module report at Computer Science at Roskilde University, or equivalent. The experienced programmer will probably only find this report interesting, when reading it from a teachers perspective. That being said, there are of course several reasons to continue reading.

Firstly, the choice of programming a game of Battleship, allows one to gain experience with both the concepts of the Java programming language (its object oriented design for instance) and the details and methods of data structures. Secondly, it also allows for experimenting with user interface design, without the risk of getting bogged down in user interface details, compared to designing more elaborate software systems. And finally, as pointed out in the opening sentence, the features required are more or less pre-defined, which might leave out important aspects of software design, but nonetheless the choice of creating a Battleship game is beneficial to a one-person, BA student project, having only a limited timeframe.

The game of Battleship is a classic game, but several stories about its origin apparently exist. At neweranet.com it is suggested that the Battleship game has its origin in Russia, invented by Russian soldiers between 1917 and 1922. It is also suggested that a predecessor of the Battleship game existed as far back as 1890[1]. On wikipedia.org the writer John Toland is referenced, suggesting that American convicts sat in their cells during the 1920's, shouting to each other their strategic moves and results thereof[2]. Undoubtedly this would have been possible, since the game can be played using only pen and paper. According to gamesmuseum.uwaterloo.ca the first commercial version of the Battleship game was produced in the 1930's, crediting French soldiers for inventing the game during the First World War. Although the history of the Battleship game is interesting, this project isn't a history assignment. The characteristics of the Battleship game are of greater interest.

The game of Battleship normally features two players playing against each other, each having two game boards, divided into squares of equal size. Each game board typically has 10 squares both horizontally and vertically. Each player also has five ships to be placed on her own, private game board, which the other player cannot see. The ships have to be placed within the board, and the players have to place all the ships on their private boards, and cannot leave any ships out. The second game board is then used by the players to record the results of the shots they in turn have to fire on each others private boards. When, and if, a player hits a ship on the other players board, the shooting player has the right to shoot again, continuing until nothing is hit on the enemy's game board. When a player is successful in sinking an enemy ship (hitting all parts of it), he receives a certain amount of points, dependant on the type of ship that was sunk. There are five ships, the largest and most valuable being the aircraft carrier. There is a battleship, the second largest ship, and a destroyer and a submarine which are equal in size and value. The smallest ship, and often the most difficult to find on a game board is the mine sweeper. The players keeps taking turns at firing shots, and the game ends, when one of the players has lost all ships on her own board. The other player then wins, since points are given when a ship is sunk.

The approach taken in this project is based on an analysis of other, existing Battleship games 'out there'. The features and ideas of these other implementations will be distilled into a specification

[1] http://neweranet.com
[2] http://www.wikipedia.org

and possible design for a new Battleship game, which will then be discussed thoroughly, carefully considering alternatives and consequences.

This discussion will lead into an actual software design for a new Battleship game, taking into account desirable design pattern considerations. More concretely, actual well documented and well commented Java code will result from this discussion and the laid out software design.

My motivation for working on the project, which is defined above, stems from the desire to better my Java understanding and coding skills. This semester (autumn 2007) I have been following my first Java course at the University of Roskilde, and I then decided that the best way to further my abilities would be to design and program a game of Battleship, all things considered. I also work on my own, and therefore need to choose a subject and project goals that are not too comprehensive, given the time and space available.

The goal of this project, then, is to develop a basic implementation of a Battleship game. The game will allow a single player to start the game, and play a round against the computer. To accomplish this, a basic user interface will be created, that displays a Battleship game board. On this board the computer will place ships when the game is started. Then the player must fire shots at this board until there are no more ships, at which point the player wins the game (she can never loose). For each action the player takes, the result is displayed in a status message. If a ship was destroyed, points are assigned to the player's total score. Of course, the underlying Java code features (classes, their methods and data structures) that supports this specification of the game will also be developed. This is also discussed in section 2.2.

1.1 The report – structure and contents

The structure of the report mirrors the project outline, which was described above. The first chapter is the chapter you're reading now.

The second chapter is about the existing Battleship games that I've come across during the projects research phase. A brief overview over these games will be presented, as will, in more elaborate form, my intentions with the project. Once the intentions are firmly established, some design possibilities and alternatives will be discussed, while developing a feasible design for a Battleship game. Also in this chapter, related to the game design, a discussion about possible Artificial Intelligence features will be had. It is of importance to notice, that when applying the term Artificial Intelligence (or AI) I refer only to certain 'automatic functions' that the Battleship game must perform. I do not refer to an intelligent system that is able to 'learn' and change behaviour based on previous experience, for example.

The third chapter seeks to describe and illustrate the actual design and code that resulted from the previous chapter. This will be done by presenting certain important code parts to the reader, and sketch the program structure and flow visually.

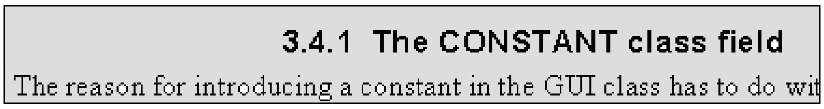
The fourth chapter concerns installation and execution of the Battleship game. The chapter covers download and installation of prerequisites and the Battleship game. Examples will be given, on how to get the program running.

The fifth chapter covers testing and a trial run of the software that was developed. Considerations regarding testing are put forward, and a full game is played, while observing features and game play in action. A series of screenshots will serve as reference.

The sixth and final chapter contains concluding remarks on the project and the process as a whole.

1.1.1 Layout and types

As the observant reader may have noticed, each chapter and/or section of the report has a headline, prefixed with a number, as the following example shows.


3.4.1 The CONSTANT class field

The reason for introducing a constant in the GUI class has to do wit

Figure 1-1 Example header

The first number indicates the chapter to which the section belongs. The second number indicates to which section the section you're reading is a subsection, and so on.

When code is reproduced in the report, a separate type is used for this. If the code is in-line `this type is used`. If presenting a code block to the reader, the special code type is also used, but this time in a figure.

```
system.out.println("Java code example - ");
system.out.println("in a figure...");
```

Figure 1-2 Example code block figure

2 Designing a Battleship game

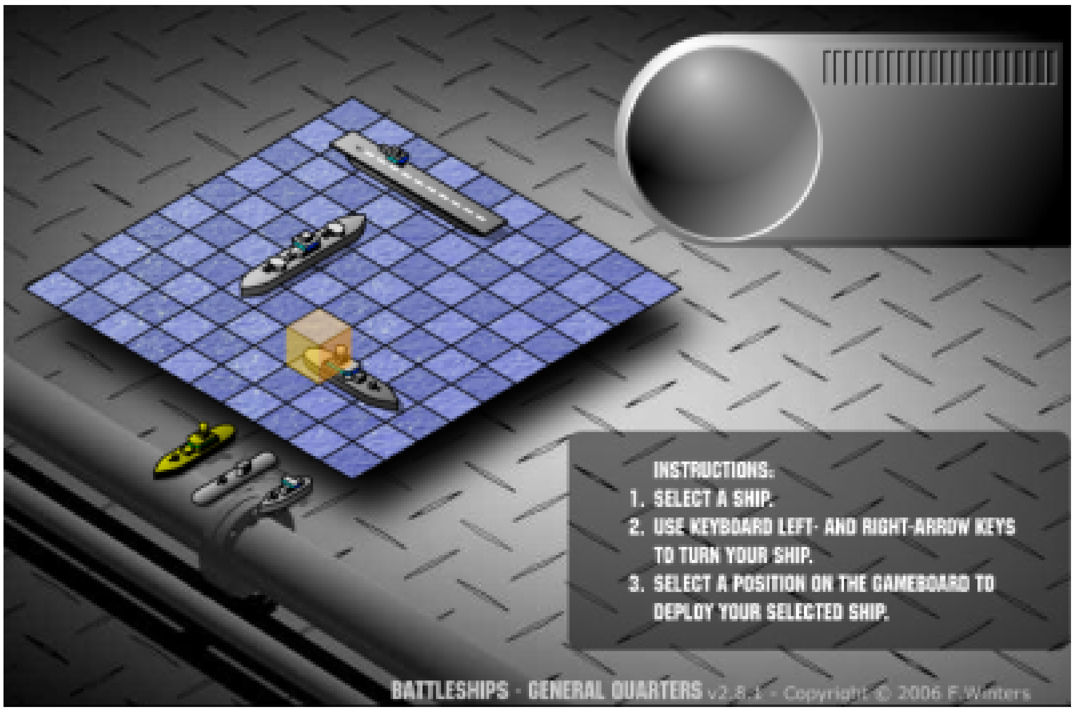
As a preliminary exercise, this chapter is about electronic versions of the Battleship game that have already been developed. Two online implementations of the game will be examined, as will briefly a standalone version that includes both server and client software. Finally, the requirements for this project's Battleship game will be formulated. Next, the possibilities for developing a sensible design will be discussed. Finally, certain implications regarding a machine player, or AI player, will be analysed.

2.1 Existing Battleship games

As indicated in the Introduction there are many ways of constructing a game of battleship as a software system. When browsing the internet, an often seen solution is to embed the game into a web page, as an applet or embedded object of some kind.

2.1.1 Browser based Flash implementation

The following screenshots shows an implementation of the game as an embedded Flash object:



INSTRUCTIONS:
1. SELECT A SHIP.
2. USE KEYBOARD LEFT- AND RIGHT-ARROW KEYS TO TURN YOUR SHIP.
3. SELECT A POSITION ON THE GAMEBOARD TO DEPLOY YOUR SELECTED SHIP.

BATTLESHIPS - GENERAL QUARTERS v2.8.1 - Copyright © 2006 F.Winters

Figure 2-1 Placing ships on the game board

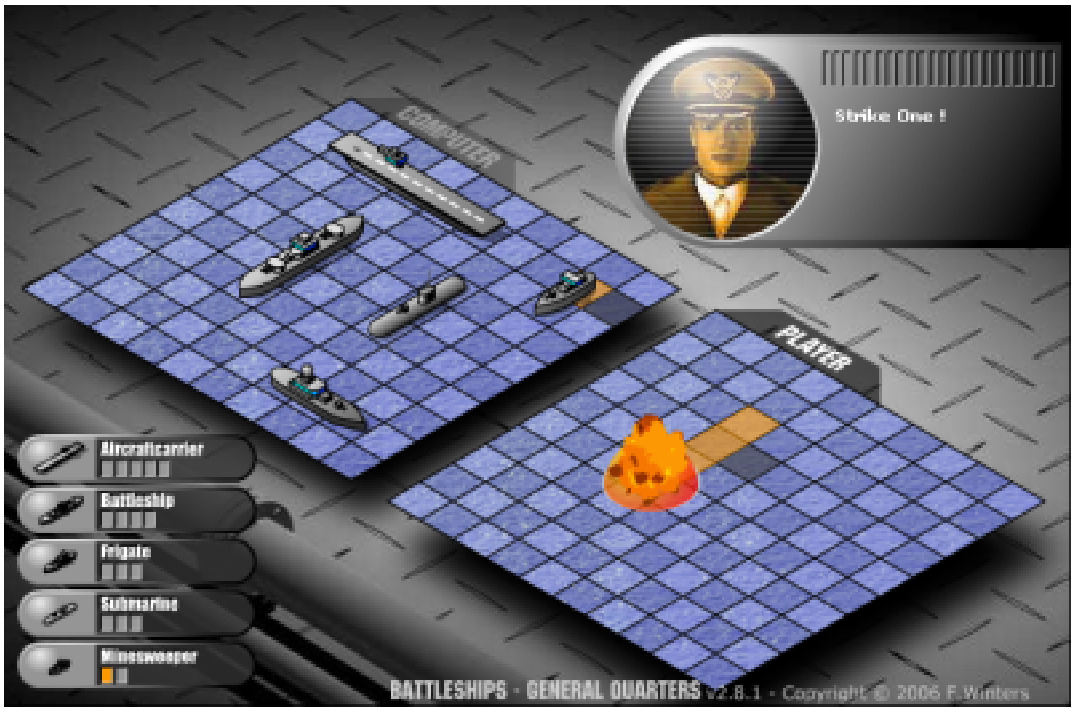


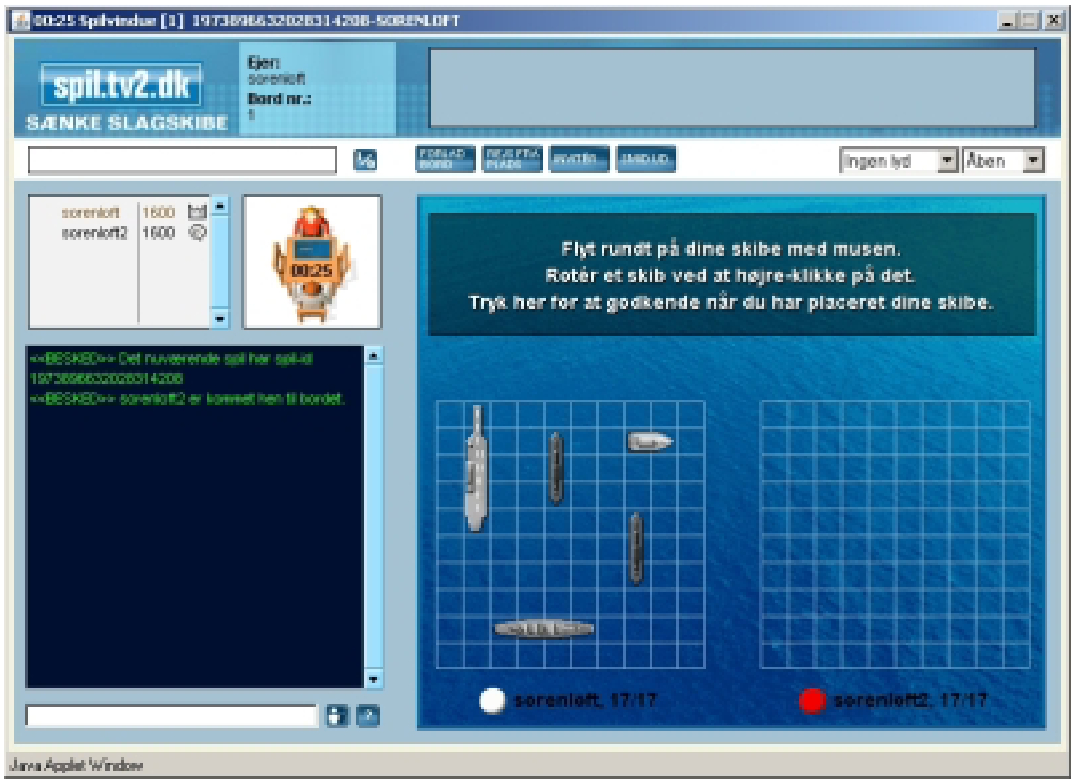
Figure 2-2 Playing against computer

These screenshots are of an elaborate and complete implementation[3], with very detailed graphics and very navy-like theme. The game play, however, is exactly as described in the Introduction, with one of the players being an 'AI player', meaning that this player is non-human, but still able to make informed decisions about where to shoot next. For example, as it can be seen in Figure 2-2, the AI player fired a successful shot, and hit the human players mine sweeper. Immediately following the successful shot, the AI player tries to fire at one of the neighbouring fields, knowing that there must be more parts of the ship that was hit previously. In the case above, the AI player did not hit anything with the second shot fired, but will remember its previous shots in the next round, and it will then fire at the other neighbouring fields, and eventually sink the human player's mine sweeper.

2.1.2 Browser based two-player implementation

It is also possible to envision other designs and implementations of the Battleship game. In the applet/object approach described above, the AI player could be replaced by another human player. The game could be designed as a two-player game, in other words. This could be a feasible implementation of the game, for an online gaming portal, such as spil.tv2.dk, where members of the gaming community can play Battleship against each other, with almost the same setup and rules as the ones outlined in the Introduction chapter. The implementation at spil.tv2.dk is based on Java, and very comprehensive regarding extra features, such as hi-score, the possibility of creating a table to sit at, excluding players from the game and more. These features are not just for the Battleship game, but part of a complete framework for the online game portal[4]. One shortcoming of the game, that distinguishes it from the Flash implementation depicted in the previous section, is the fact, that when a successful shot has been fired, the player doesn't get an extra shot. Whether this actually is a shortcoming, depends on preferences for game play, of course. Below a couple of screenshots from the TV2 implementation are shown.

[3] Source: http://www.battleships.f-active.com
[4] Source: http://spil.tv2.dk

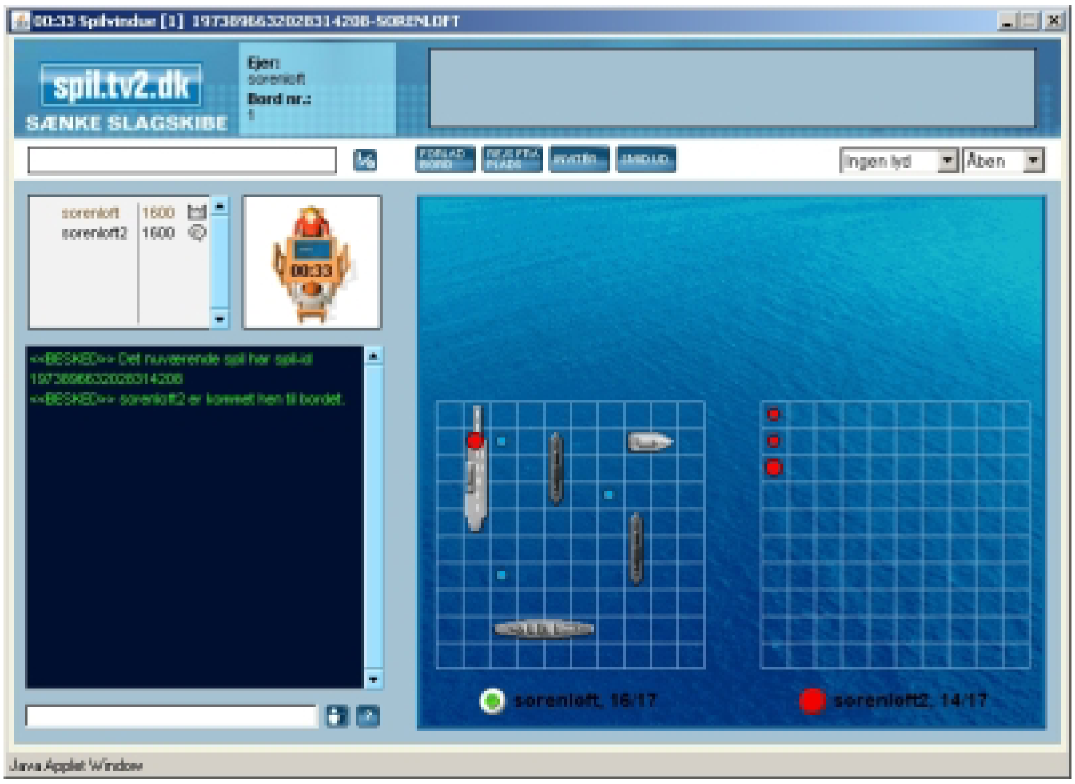


00:25 Spilvindue [1] 1973896632028314208-SORENLOFT

Flyt rundt på dine skibe med musen.
Rotér et skib ved at højre-klikke på det.
Tryk her for at godkende når du har placeret dine skibe.

<<BESKED>> Det nuværende spil har spil-id 1973896632028314208
<<BESKED>> sorenloft2 er kommet hen til bordet.

Figure 2-3 TV2 spil: Placing ships



00:33 Spilvindue [1] 1973896632028314208-SORENLOFT

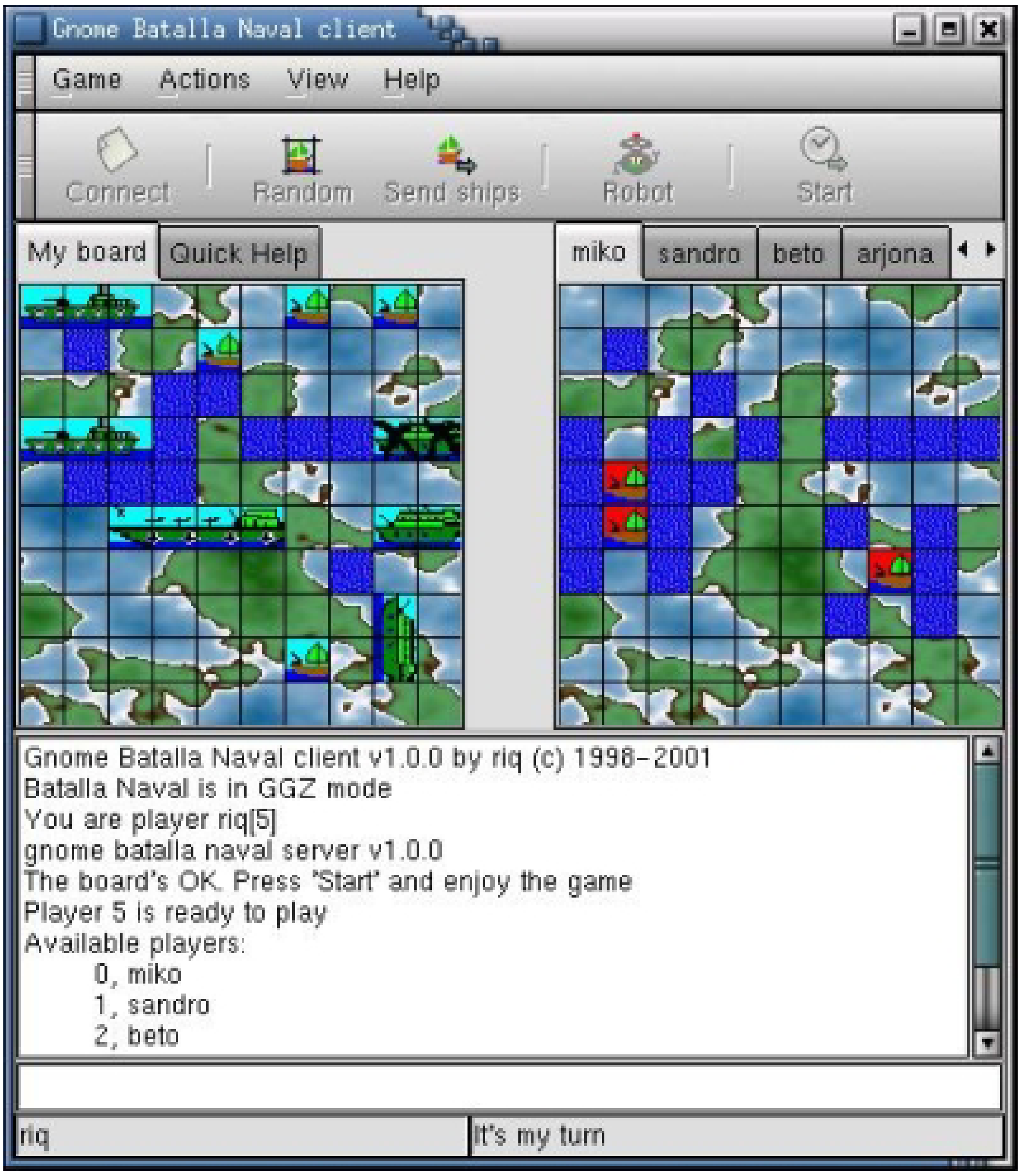
<<BESKED>> Det nuværende spil har spil-id 1973896632028314208
<<BESKED>> sorenloft2 er kommet hen til bordet.

Figure 2-4 TV2 spil: Playing the game

2.1.3 Client/server implementation

Yet another approach could be to implement the game as a standalone application with network support, to be installed independently from browsers and gaming service providers, requiring only Java Runtime (if the application was developed in Java) and network connectivity on the client. One could then consider whether it would be desirable to program the game as a 'true' peer-to-peer application, with connections only between the two clients playing the game, or the implementation should include a central server, hosting information about online gamers, their IP addresses and so on. The latter would of course make it easier to find other gamers, and the players would not have to know anything about host names or IP addresses, whereas the peer-to-peer approach would require the players to know how to connect to other players, typically by entering IP address or DNS host name. One advantage to this approach would be complete independence from any central servers and services.

A very comprehensive version of the Battleship game that illustrates some of the possibilities available is 'batalla naval'[5]. This game has 3 main components: A client, a server and a robot. It is a multiplayer game, where each of the participating players can be represented by human players or robots (AI players). Of the three components mentioned, only the client has a fairly comprehensive graphical user interface. This game is an example of a Battleship implementation, which requires a fair amount of user experience, since the current version (2002) only runs on GNU/Linux with the GNOME desktop installed. The game does exist in Debian package repositories, so installation should be relatively simple, at least on the Debian Etch operating system.



```
Gnome Batalla Naval client v1.0.0 by riq (c) 1998-2001
Batalla Naval is in GGZ mode
You are player riq[5]
gnome batalla naval server v1.0.0
The board's OK. Press 'Start' and enjoy the game
Player 5 is ready to play
Available players:
        0, miko
        1, sandro
        2, beto
```

riq   It's my turn

Figure 2-5 Batalla naval: Multiplayer client view

[5] http://batnav.sourceforge.net

2.2 Specification of this Battleship game

The ambition of creating something similar to what we have studied so far is, however, a task that requires a great deal more time than is available for this student project. But several central aspects of the game can, nonetheless, be programmed. The following is a non-technical outline, of what this student version of the Battleship game is supposed to do.

The core part of the Battleship game is the ability for players to fire at the opponent, the game being over when one's counterpart has no more ships left. Furthermore the concept of an AI player is perhaps more challenging from a programmers point of view, than the remainder of the game. Also, the concept of intelligent, autonomous systems is a widely used one, the appliance of it ranging from computer games to lunar landings. It therefore seems relevant to include at least some AI aspects in the game, with regards to AI's widespread use. Independently from the number or nature of players and the platform on which the game is implemented, the core part of the game, one could argue, is the ability to register ships and shots fired on a gaming board. To put it in another way: If we design this core part of the game, we have gone a long way in developing the game, which could then easily be extended with regards to user interface and secondary features. Based on these thoughts, the overall goal with this student project and the report you are currently reading, is to develop a core component of the game, which is able to represent a Battleship game board, and furthermore, to develop a simplistic user interface, where it is possible for a player to interact with the game board (place shots and score points). Of course this makes little sense, if the game board has no ships on it. Hence an AI player will be partly developed. This AI player will not be a 'real' player, in the sense of being able to fire shots, score points and win games, rather it will an 'invisible pseudo-player' that is able to randomly place ships on the game board, for the human player to fire shots at. In the following sections, we shall examine the implications of these ambitions in greater detail, but before doing so, this section is summarized into the following table.

Features	f-active.com	TV2	batalla naval	This project
Manual ship placing	X	X	X	
Automatic ship placing	X		X	X
AI player	X		X	
Play online (network)			X	
Play online (web)	X	X		
Single-player	X		X	X
Two-player		X	X	
Multiplayer			X	
Basic graphics				X
Rich on graphics	X	X	X	

Table 2-1 Feature comparison

2.3 Creating a design

Since the author of this report has been following a Java programming course this semester the ambition has been to gain further insight into the object oriented approach sported by the Java programming language. Of course other languages could have been used, Microsoft's C# language on the .Net platform being one possible alternative. Java was chosen however, because of the (lack of) skills of the author, and because, simply put, it is able to do the job required: It supports an object oriented approach to programming; it has support for complex data types and structures, and has a strong type system. Furthermore Java is considered 'safe' compared to other languages such as C (with the renowned susceptibility to buffer overflows). One major advantage of Java is platform independence. It is possible to run the same Java application on different operating system platforms, without putting a lot of effort into porting the code. Furthermore the Java SWING and AWT libraries make it possible to develop a graphical user interface for the Battleship application.

2.3.1 Tools

Although not directly relevant to the outcome of the coding efforts, the integrated development (IDE) chosen is Eclipse. Not because it is better than other alternatives (of which I know very few), but simply because its free, and it has the ability to assist in Java syntax and type checking, and because it is very straightforward to compile and execute code. Furthermore I find the structure of the user interface very useful, and it is easy to change your working environment and setup, according to your preferences (maybe you are working at home on a 20” display or you are on the move with a 12.1” display.

To display the user interface and its components I use the JCanvas class. This is a Java class, which simplifies the task of programming 2D-graphics, albeit being suitable for our purposes. Another auxiliary class that the Battleship game relies on is JEventQueue, which is also a simplified version of the standard Java library. Both these classes have been obtained from akira.ruc.dk/~madsr/swing.html, the homepage of this project’s supervisor.

2.3.2 Object Orientation

In using the object oriented approach and an object oriented language, such as Java, it has often been difficult for programmers with experience from other languages, to grasp the concept of ‘object orientation’. I know from myself that one tends to think in functions (equivalent to the Java methods), and do not quite see the benefit or purpose of using object orientation as a way to model the real world. Of course a complete model of almost any reality is unrealistic, but nonetheless we can extract characteristics from the real world that is useful when designing software. In other words we selectively pick the features we require in our software, and ‘assemble’ these into classes. Think of a class as a container, which can contain several objects, for which the class contains methods (functions) that defines the ‘things you can do’ with the objects contained therein.

Using the object oriented logic outlined above leads us to develop entities for our program that models the ‘real world’ – the real world in this case being our Battleship game. One can easily distinguish several abstractions over the game which could be useful. For instance there are ships, players, shots, boards, score and a fleet of ships, all of which could (or should) have varying degrees of detail associated with them. For instance one could consider the concept of a ship. One property of a ship would in the real world be colour. Since the colour of a ship plays no role in our game, we would probably not include this in our design. On the other hand, if we were to display different ships in different colours (if the ships of one player were to have another colour than the counterpart’s) we would need to include colour as a property of an object. Many such decisions would have to be made, and one will have to give careful consideration to future requirements of the software and the amount of detail needed to create a satisfying design model. We will explore this in greater detail later in this section, since we also need to take other factors into account.

2.3.3 Model-View-Controller design pattern

Another perspective on designing software that requires user interaction, is the concept (or design pattern) of Model-View-Controller (MVC), which separates the tasks of the different components of the software in a way such that three general roles can be singled out: The model, the View and the Controller. The Model contains the basic, underlying data access methods and data storage. The View represents the visual user interface and the Controller is the part of the software that listens for user input and reacts accordingly. In the original MVC design concept (see figure of MVC model below), all three elements are separated, which typically would be reflected in the final project code. When coding the Battleship game, however the View and Controller will be somewhat integrated in the sense that the use of a MVC design pattern, does not entirely separate the code relating to the View from the code relating to the Controller – this is due to Java’s implementation of event listeners. Conceptually we will still be able to think about separate elements however.

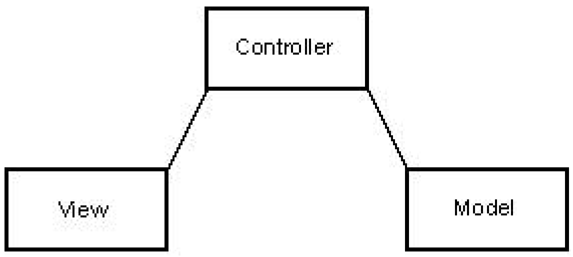


Figure 2-6 Model View Controller diagram

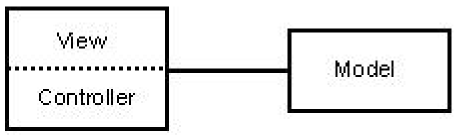


Figure 2-7 Battleship MVC concept

The benefit of using the MVC design pattern is separation of data (the Model) and the View and Controller. This separation makes it easy to update the implementation of the Model, for instance, since the Model and its underlying data structures are independent of the View and the Controller. We could then easily accommodate changes relating to for example persistent data storage or data structures. It would also be possible to use the Model as a Model for a whole different system, for example if we wanted to use the same model in an online implementation our Battleship game. If we did not apply the MVC concept to our design, this would present quite a challenge, since reprogramming most of the software for another platform would possibly be required. Thus the MVC design pattern allows us flexibility, maintainability and code reusability.

2.3.4 Battleship objects

Now, after considering design issues on a somewhat abstract level, we need to focus our attention to some of the details pertaining to the design and coding of the Battleship game. Let us start by considering the objects and classes needed in our Battleship game, and relate these to the MVC concept.

A list of possible objects was suggested above. They were: Ships, players, shots, boards, score and fleet. The question now is: Which objects make sense to include in our game, what properties should they have, and how should they be structured (in classes)?

Some of the suggested objects seem obvious to include right away. A ship is an important part of the game, since the game cannot be started (or ended) without ships. Also ships have to be placed on a game board, and the game also cannot be played without a game board. This kind of reasoning also applies to the players. We now have identified three distinct objects to include in our design and program: Ship, Board and Player. Remembering our MVC pattern, the Model element is represented by these three objects, since these three entities also represent the core parts of the Battleship game which can be briefly summarized as ‘a game where players fires shots at a board, and (possibly) hits each others ships’. The concept of a fleet was mentioned and it is another candidate for ‘objectification’. We need some way of tracking the association between ships and the game board, and for now we will decide on an object of type fleet. This is also discussed in section 2.3.6.2.

Shots, then, does not really qualify as objects, since they can be seen as an action and not an entity. If shots fired could have different characteristics, such as velocity, explosive charge and so on, it would probably make more sense to include a shot as an object in our design. The suggested object, ‘score’, is more a property of a player, than an object in its own right. There is going to be no advanced ‘score mechanics’ in this game – it’ll just be a number that changes whenever the human player destroys a ship.

2.3.5 Class design

The ‘business’ part of the game, is, as mentioned, above, the objects board, player and ship. Together these constitute the Model in the MVC design pattern. Each of these objects have several characteristics and must be able to perform certain tasks, such as keeping track of ships on the board, shots fired and the status of each ship on the board, just to mention a few.

Following Cay Hortsmann’s conventions (Horstmann 2007: 336-337) a class should represent a *single concept*. This is also the approach used in identifying the objects of the Battleship game. Following this principle, we can now state three classes for the Model of our game: Board, Ship and Player, the latter being conceptually similar to what Horstmann describes as an *actor* (Horstmann 2007: 336). In addition to these classes we need a class to represent the graphical user interface of the game; hence the class GUI is created. This class is, in design terms, a representation of the View and Controller parts, from the MVC design pattern. All graphics display and user interaction will lie here.

As was outlined in the previous section, some AI functionality is required in the game. We shall give this a closer look in section 2.4 of the report, but is bears some relevance on class design, or class *inheritance*. In future versions of the game, it would make good sense to increase the AI capabilities of the game. It would make the game more complete and more fun to play. It therefore seems wise to take this into account when designing the Player class. Also, we already have certain AI requirements; the automatic placing of ships on the game board. Therefore the AIPlayer class is created which inherits from the Player class. This way AI related functionality can be implemented here, while the AIPlayer inherits characteristics from Player, such as player name and player score.

As a way of making sure, that no errors occur when placing ships on the board, or firing shots at the board, it would make sense to define our own exceptions. This way it would be possible to ensure that the game keeps running, without seeing weird placement of ships (ships placed on top each other, for example). Also we can throw an error when attempts are made to fire a shot on a field that is non-shootable. By throwing these errors we also ensure that the game does not exit due to runtime errors. As a final note on exceptions, it would also make sense to use inheritance. Of course, to create any form of custom exception classes, inheritance would be required from the general Java ‘Exception’ class. As an exercise in the use of inheritance and exceptions, I will define a Battleship specific ‘BattleShipException’ class with subclasses ‘FieldOccupiedException’ and ‘InvalidShotException’.

2.3.6 Data structures

Since we defined the Board, Player and Ship classes as being central parts of the Battleship game, comprising the Model abstraction in the MVC design pattern, let us know consider the data structures herein more carefully. If deemed necessary auxiliary classes will be created as well.

2.3.6.1 The game board

We need a sensible way to represent the game board, the concept which we developed earlier. There are of course several different ways of representing the game board. One way is to implement a two-dimensional array, where each array index points to an integer that represents a certain field and its status. A field can have a ship on it or be empty, and it can at the same time be hit or not hit. Besides this information the two-dimensional array contains nothing else, if the array stores the status of a field only as data type integer.

This has one particular advantage. The two-dimensional array is very easy to use, and each element can be accessed simply by using the x and y coordinates of the game board as array index. Furthermore, the efficiency and scalability of this two-dimensional array data structure is very good. It takes constant time to lookup any element, independent of the number of elements stored in the array.

The disadvantage of implementing such a data structure is its simplicity. If the two-dimensional array stores only an integer, no other information than field status, can be retrieved. To determine, for example, which ship is placed on a given coordinate, one must then implement this functionality elsewhere; it cannot be read directly from the two-dimensional array, unless we choose an alternative approach by storing not integers in the array, but objects, for example of a custom type 'Field'. This way we could store the field status, along with any other necessary information, such as a reference to a ship, if any.

For simplicity, we will go along with storing the field status in a two-dimensional array, containing only integers.

2.3.6.2 The group of ships (the fleet)

Another central data structure is the group of ships placed on a board. Because we chose to implement the two-dimensional array storing only integers, we need some way of identifying the ship that could potentially lie on a board field. For this purpose we developed the concept of a fleet. Again we have several options when choosing the appropriate data structure. But since the game specification does not require us to allow for a random number of ships, efficiency and scalability is of little importance. Thus it seems sensible to use an array list in the Board class, for storing the ships associated to the game board. This, then, will be a central data structure for finding ships, when these are hit or sunk.

2.3.6.3 Coordinates

Derived from the fact that the game board's central data structure is a two-dimensional array, where access to a certain board field is achieved through x, y value pairs we need some way to deal with coordinates, that will be more agile than simply using x and y value pairs all the time. Also we need a more intuitive way than using x and y values to search for ships. Hence we introduce an object (and a class) of type coordinate. This is more due to practical coding concerns, than other design issues. As a side note, this decision is also in line with Hortsmann's recommendations, since a coordinate class represents a single concept – a coordinate.

2.3.6.4 Ship memory

Speaking of concepts, in a real world concept of a ship, one would expect that any given ship would have knowledge of its whereabouts (hopefully the crew has, anyway). Deducting from this kind of logic, we should implement some data structure in our Ship class for storing the coordinates of any instance of type Ship. But programming logic can of course be different from real world logic, and one could argue, that the game board, for each field on it, is aware of ships. This leads us back to the discussion concerning the data structure used in the Board class. Since we earlier decided to implement this by means of a two-dimensional array storing integers, we are now forced to either remake our decision, or choose an appropriate data structure to use in the Ship class, for storing coordinates. We will do the latter. Based on the fact that a Ship can only have five or less coordinates (the maximum length of a ship is five, this is equivalent to five coordinates) efficiency and scalability is of little importance, and therefore it will suffice to use an array list for storing coordinates.

2.4 AI considerations

In this section we will discuss how to implement the intelligent machine player, and also spend some time reflecting on other features that would or could be desirable to include in future versions of the game.

As described earlier, it was decided to add an AIPlayer class to the game design. This is a subclass of the Player class, and it currently should inherit characteristics such as player name and score, along with trivial methods to set and get player score and player name and so on. All these characteristics apply to the artificial player as well as the human player, since, to add to the excitement of the game, it could be a good idea to display both computer score and human player score – but in this version of the game it is not a priority.

2.4.1 Placing ships on the game board

What is a priority, on the other hand, is for the computer to be able to place ships on the board, so that the human player can start shooting and sinking ships. This functionality requirement, along with the possibility of extending AI functionally in future versions of the game, leads us to the design of an AIPlayer object and class. And the approach taken in designing this software, partly based on Hortsmann's notion of 'one class – one concept', suggests that AI related functionality should be considered part of the AIPlayer subclass, since certain operations pertain only to the machine player. One could of course consider, again in future versions of the game, to allow the human player to use parts of the AI functionality. If the game really should be expanded, one could imagine a 'quick game' menu option, where the human player is then able to utilize the automatic placing of ships on the game board, simply to get started playing right away, not spending any time placing ships manually on the board.

Regarding automatic placement of ships on the game board the simplest and easiest way to do this, would probably be to define the properties of the required ship objects in the AIPlayer subclass. In a simple method call upon execution of the game, placement of ships on the board could then be initiated.

The properties of any ship are in this version of the game fairly simple. Any given ship can have a size (length of the ship) and an orientation (it can lie either horizontally or vertically on the board). The size of the ship can then be represented as a set of coordinates, the number of which determines the length. It doesn't require a lot of programming effort to develop some routines that are able to generate a random starting coordinate for placing the first part of a ship, and then, based on a random orientation, creating a set of coordinates needed for creating and placing a ship on the board. And this procedure would then have to be followed for each ship that is to be placed on the game board. The constraints in this situation will (of course) be that no ship can have any of its parts on a game board field that's already occupied by another ship. Also all the parts of a ship must lie within the borders of the game board.

2.4.2 Other possibilities

A few other relevant observations regarding AI features can be made. One possibility would be to extend the capability of the ship placing mechanism, and add some tactical sense to the procedure. For example, measures could be taken, to ensure that each ship has a minimum distance to other ships, so that ships will not end up right next to each other, which would make it easier for the human player to sink the ships, once the group of ships was found on the board.

Possibly there would be some advantage to the machine player (i.e. it would make the game harder for the human player) if thought was given to the distribution of ships on the game board. Without going into detail here, one could calculate the probability of hitting ships when these are placed in a certain way, and apply this calculation to the placing of ships.

2.4.2.5 Intelligent distribution of shots

But the placing of ships is only one aspect of the AI functionality that a game of Battleship could have. Like a human player, the machine player would also have to fire shots at the human player's game board. The firing of shots is in itself pretty straightforward if not considering tactics or some sort of intelligence added to the action. A random set of coordinates within the borders of the game board could easily be generated, and then the shot could be placed on the board. This would be a very basic implementation of shooting capabilities, but it would probably not feel very authentic to play against such a machine – it would simply be too easy to win. Therefore it would make sense to adopt some systematic approach to firing shots. One way could be to consider the fact that any ship on the board has a length of minimum 2. Hence it makes no sense to start firing shots at the north-west corner of the board, and shooting at each and every field of the game board. Instead shots could be fired at every second field along the borders of the game board, working gradually inwards on the board after each 'round trip'. This approach is illustrated in the following figure, where the X represents a shot and the arrows represent the order in which to shoot at the game board fields.

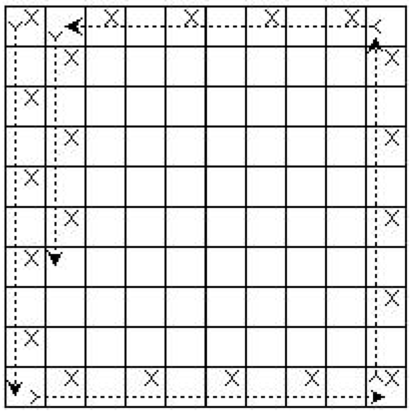


Figure 2-8 Possible distribution of AI shots

One thing is the distribution of shots on the game board; another is machine player memory and playing behaviour. The machine player must, of course, remember the previous shot, but it must also remember the *result* of the previous shot. At least if the goal is to simulate human behaviour. When a shot is fired and a ship is hit for the first time, we will know as humans, that there must be more of that ship on at least one of the neighbouring game board fields, depending on the number and type of ships left on the game board. The machine player should therefore be able to simulate this kind of inductive behaviour. After a successful first shot the machine player should then consecutively try each of the direct neighbours to the field in question. When another shot becomes successful, the machine player should then be able to induct, that it now knows the orientation of the ship it is firing at. In other words, if two shots are successful, and they are oriented horizontally, the machine player should not fire shots to either the north or south of the previously hit fields, but only to the east and/or west of the previously hit fields. Hence, after having had a second successful shot on one of the neighbouring game board fields, the machine player should continue firing in the direction suggested by the successful second shot. The machine player should then continue placing shots in this direction, until the ship is sunk, or an unsuccessful shot occurs. If an unsuccessful shot occurs, and the ship is still not sunk, the machine player should continue firing in the opposite direction from where the first successful shot occurred, until the ship is sunk. The following figure shows a section of the game board, and sketches the placing of shots.

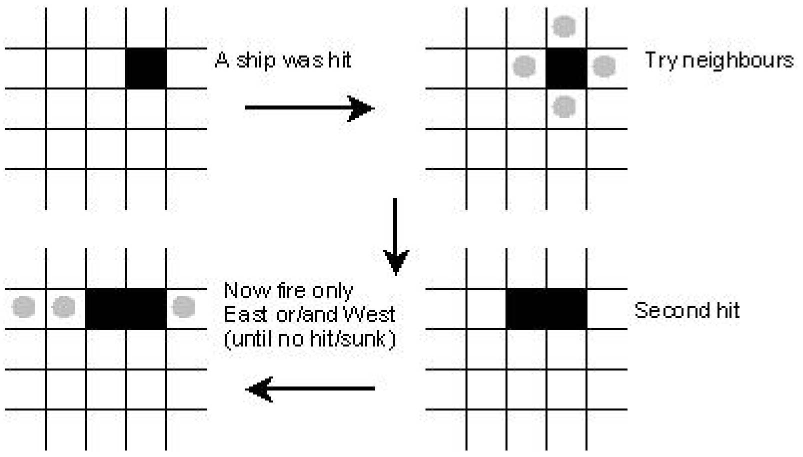


Figure 2-9 AI strategy for firing shots

2.4.2.6 Distributing shots - advanced

The paragraph above merely suggests a basic outline of how a reasonable level of intelligence could be applied to the machine player. It doesn't take into account how the machine player should react if an unsuccessful shot occurs for the second time, working in the opposite direction from the direction initially taken. If this situation occurs it must be because there are several ships adjacent to one another. Then the machine player, to behave human-like, should also be able to induct this, and behave like the basic outline described above suggests, for each of the possible ships adjacent to each other. This would certainly require a more capable machine player, since it must not only remember the last successful shot; it must also be able to pursue the sinking of all the ships in adjacency to each other, and finally it must return to the strategic firing of shots, remembering where the next shot should be played, according to whatever strategy has been laid out for firing shots. To clarify, the above considerations are sketched in the following figure.

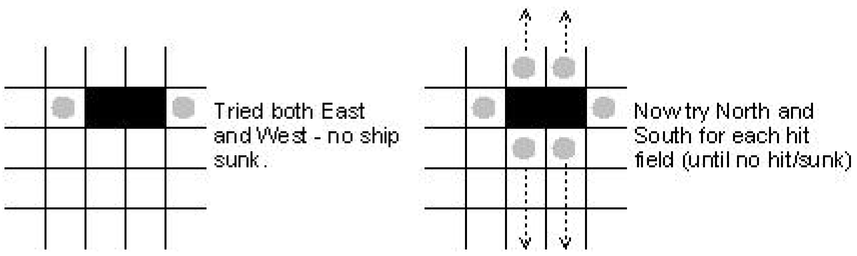

Tried both East and West - no ship sunk.
Now try North and South for each hit field (until no hit/sunk)

Figure 2-10 AI advanced strategy

2.4.2.7 User experience issues

Some final thoughts on the subject of a machine player suggest certain variations in its behaviour. If the machine always placed its ships in exactly the same way, the game would soon become trivial. Thus we need variations in ship placement. To keep the game interesting and more difficult to predict, also after the first couple of games played, variations should be introduced in the way the machine player fires shots, i.e. it should not always start firing at the same field, and then distribute the shots identically around the game board. This would make it easy for the human player to place her ships in a way that avoids detection.

Finally a couple of minor details could be considered in relation to user experience. Firstly, a small time delay probably should be introduced after the human players last unsuccessful shot and before each of the machine players shots. This should be done to simulate 'thinking' on behalf of the machine player. Secondly, it would be interesting to develop some ideas regarding an intended introduction of errors in the way the machine player behaves. The idea of a computer player that 'looses its attention' seems kind of fun, and this concept could perhaps be fostered and developed further, into a machine player skill level option.

By now it seems obvious, that really good machine player design is a student project in its own right, and hence no further details will be explored. The avid reader can look up the book "Artificial Intelligence - A Guide to Intelligent systems", by Michael Negnevitsky (2004), which is an introduction to the field of artificial intelligence. This book is also used on the KIIS computer science course at the University of Roskilde.

3 The Battleship Java code

Up till now the area of interest has been clarified in various perspectives, and some more general and abstract considerations regarding design of the Battleship program have been made. The reader has also been presented with a whole range of possible additions that could be made, in order to develop a more comprehensive solution for the Battleship game. The goal of this chapter, then, is to look into the more intricate and detailed parts of the actual code that was developed in line with the requirements stated earlier. The design process spawned several classes, all of which were described earlier. To provide an overview of the code design and structure, the following UML diagram illustrates the classes developed and the relationship between them. Furthermore, methods and class fields have been included in the diagram to provide a schematic overview of the complete project code.

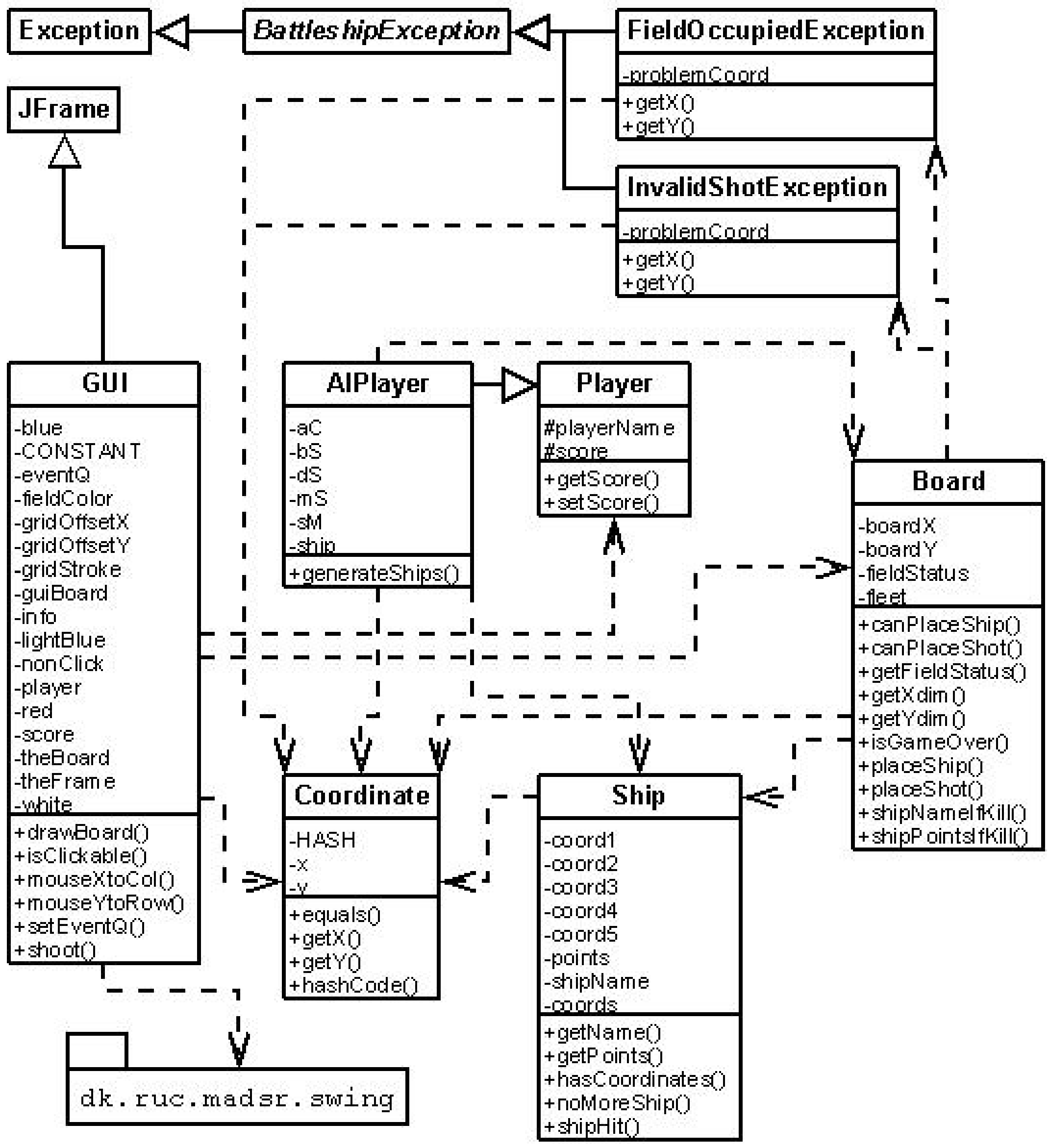


Figure 3-1 Dependencies and inheritance

Recalling the fact, that the core parts of the Battleship program are considered to be the Board, Player and Ship classes, those classes, their data structures and methods will primarily be investigated and explained here. To some extent other features of the software will be considered as well, since especially the GUI class contains methods that are best understood when accompanied by a written explanation.

3.1 The Board class

Decided upon earlier, the Board class contains a couple of vital data structures and methods that act as interfaces to these classes. This class is vital to the game, in that it provides the players (in this version only the AIPlayer) with the ability to place ships on and fire shots at the game board. It is also responsible for storing the status of each of the game board fields. Observe the following code snippet:

```
// Default no. of fields in board grid
private static int boardX = 10;
private static int boardY = 10;

// Field status array
// 0: empty, not hit
// 1: empty, but hit
// 2: not empty, not hit
// 3: not empty, but hit
private static int[][] fieldStatus;
```

Figure 3-2 fieldStatus array - Board.java

The class field, or variable, `fieldStatus`, is a two-dimensional array, storing the status of each game board field as type `int`. This is, as described earlier, a very simple and efficient solution for representing the game board. The limitation is, of course, that other objects cannot be stored in the array; hence we are forced to develop an alternative solution for associating ships with fields on the game board. Also notice, the variables `boardX` and `boardY`. These actually determine the game board dimensions and are preset with a value of 10, which is typical for the Battleship game. Alternative sizes can be supplied to the constructor of the Board class.

Since `fieldStatus` is the main data structure for representing the game board, several methods have been developed to provide access to it. These methods handle the placing of ships on, and firing of shots at, the game board.

The operation of placing a ship on the board has been split into 2 methods, as can be seen in these pieces of code:

```
public boolean canPlaceShip(Ship theShip) {
  Iterator<Coordinate> iterate = theShip.coords.iterator();
  while(iterate.hasNext()) {
   Coordinate fieldCoord = iterate.next();
   int x = fieldCoord.getX();
   int y = fieldCoord.getY();
   if(x >= boardX || y >= boardY) {
    return false;
   }
   // If ship allready in this field
   if(fieldStatus[x][y] != 0) {
    return false;
   }
  }
  return true;
 }
```

Figure 3-3 The canPlaceShip() method - Board.java

```
public void placeShip(Ship theShip)
  throws FieldOccupiedException {
  Iterator<Coordinate> iterate = theShip.coords.iterator();
  while(iterate.hasNext()) {
   Coordinate placeCoord = iterate.next();
   int x = placeCoord.getX();
   int y = placeCoord.getY();
   // If ship allready in this field
   if(fieldStatus[x][y] != 0) {
    throw new FieldOccupiedException(placeCoord, "Field
allready occupied");
   }
   else {
    // Set fields to not empty, not hit
    fieldStatus[x][y] = 2;
   }
  }
  // Add the ship to the fleet
  fleet.add(theShip);
 }
```

Figure 3-4 The placeShip method() - Board.java

The methods illustrated in the above examples demonstrate the ways in which the core data structures are accessed. Besides the `fieldStatus` array we see the use of the `ArrayList<Ship> fleet` data structure that stores all ships currently on the game board. This data structure will be examined shortly. Both the `canPlaceShip()` and `placeShip()` methods take an object of type Ship as an argument, as can be observed from the figures. These methods provide a way of making sure a ship can actually be placed on a given location – specified by the `Ship` parameter. First the method `canPlaceShip()` must be called and, depending on its return value (`true` or `false`), `placeShip()` can be called with an identical parameter. Should this procedure fail (if other parts of the Battleship program neglects to query `canPlaceShip()`), the `placeShip()` method is designed in such a way, that it handles this situation by throwing an exception, if a ship is already present on the specified game board fields. To summarize, the order in which these methods are called matters.

Other routines exist for accessing the `fieldStatus` array. The most important ones, that remains to be examined, are the `public boolean canPlaceShot(Coordinate coord)` and `public int placeShot(Coordinate coord)` methods. They are similar in structure to the methods that handle placing of ships, in the sense that the order in which they are called matters. Furthermore the `placeShot()` method throws an exception if a coordinate it receives is invalid.

An important feature in the `placeShot()` method, is illustrated by the following code segment, which belongs inside the method.

```
// We have a succesful shot, and the ship must remember that it
// has been hit
for(Ship ship : fleet) {
 if(ship.hasCoordinates(coord)) {
  ship.shipHit(coord);
 }
}
```

Figure 3-5 A ship is hit – Board.java

The code in Figure 3-5 is executed whenever a shot is successful, and it is an example of the concept developed earlier in this report, that a ship should be able to remember its own status. It is also evident here, that the `fleet` array list is used. This means, that for each of the ships stored in the array list `fleet` the coordinate on which the shot was fired is searched for. When found the ship with the coordinate in question must register that it has been hit. This is what the `shipHit()` method of the Ship class does, as will be shown shortly.

Finally, three other methods exist that are all important to be aware of. They all operate on the `fleet` data structure, as well the Ship object. Their purpose is to determine the name of the ship that was destroyed (if any), get the points for the ship that was destroyed (if any) and determine whether the game is over or not (if there are no more ships left on the game board). Their signatures are as follows:

```
public String shipNameIfKill(Coordinate theCoord)
public int shipPointsIfKill(Coordinate theCoord)
public boolean isGameOver()
```

Figure 3-6 Method signatures - Board.java

Common for all these methods, is that they must be called after each shot fired at the board, and in the order listed. This means, that the only way of finding out the name of a ship (mine sweeper, battleship etc.) is to call the `shipNameIfKill()` method which checks whether a ship was sunk as a consequence of any previous shot. If a ship indeed was sunk it returns the name of that ship, otherwise it returns an empty string. Similarly, the `shipPointsIfKill()` method returns the number of points for a ship that was destroyed. In addition to this *it then deletes the ship in question from the* `fleet` *array list*. This leads to the final method, `isGameOver()`. Since a ship that is destroyed is also deleted from the `fleet` array list, `isGameOver()` can simply check whether the `fleet` array list is empty. If it is, the game is over, and this method will return `true`.

3.2 The Ship class

Most of the code contained in this class is self-explanatory. The code also has Javadoc comments embedded, from which the issues not discussed here, can be inferred. A brief summary will, however, be provided here, after which the core data structure and methods will be described.

When examining the code in Ship.java, one will notice the presence of 5 constructors. Each constructor takes a number of Coordinate objects as parameter, and also a String representation of the name of the ship. Based on the number of parameters supplied to the constructor of this class, it is possible to set the points a player should be given for sinking a ship. If the number of Coordinate objects is 5, the type of this ship must be an aircraft carrier, so instead of supplying the name of the ship as a parameter to the constructor, it could just as easily be hard coded in the constructors of this class.

The core data structures, and methods for accessing it, are outlined in the following figure, showing field initialization, and method signatures. Coordinates are added to `coords` in the constructors.

```
// An array list for storing this ships coordinates
ArrayList<Coordinate> coords = new ArrayList<Coordinate>();

// Registers that this ship has been hit
public void shipHit(Coordinate coord)

// Checks if there are any parts left of this ship
public boolean noMoreShip()

// Tests whether this ship is placed on these coordinates
public boolean hasCoordinates(Coordinate theCoord)
```

Figure 3-7 Data structure and methods - Ship.java

As have been stated several times throughout the report each ship must remember its status. The array list `coords` stores this information, and through the mutator method `shipHit()` the status information is then manipulated.

The method hasCoordinates() utilizes the Coordinate.equals() method, which returns true if both the x and y values are equal. The hasCoordinates() method is utilized, when a shot has been fired (the Board.placeShot() method) and the status of the game board field in question indicates that a ship is present. The hasCoordinates() method is then called on each ship in the fleet array list, to search for the ship with the coordinates of the shot being fired. When this ship is found, the shipHit() method, delete the coordinates of the ship that was hit, from the array list coords. This way the noMoreShip() method can easily determine if an object of type ship has any remaining parts, since the coords array list will be empty. This idea is similar to the isGameOver() method of the Board class. The following figure illustrates the structure of these method calls in the Board, Ship and Coordinate classes. Since the GUI class represents the Controller part of the MVC design pattern, this has been included in the graphical representation.

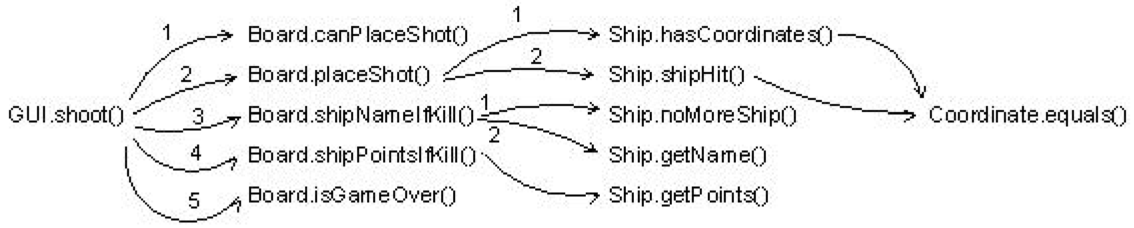


Figure 3-8 Controller: Method call sequence

3.3 The Player and AIPlayer classes

The Player class is very simple and self-explanatory. It just contains 2 class fields. A class field of type integer, to store a player's score, and a class field of type String to store a player's name. A couple of methods exist, to get and set the values of these class fields.

The AIPlayer class inherits these class fields and methods, and thereby shares these characteristics. In addition to this, the AIPlayer class contains an important method, generateShips(), that is the only AI functionality included in the game, as was discussed in the previous chapter. This method contains a block of code for each ship that is to be placed on the game board. For each ship, the code block constructs an object of type Ship, with random orientation and appropriate length. Then the canPlaceShip() method of the Board class is called to determine if a ship can be placed on the board. If it can, the placeShip() method of the Board class is called, and the code block exits, continuing to place the next ship. If the ship cannot be placed, another ship is created, until it can be placed.

3.4 The GUI class

Moving on from the Model part of the MVC design pattern, a few comments is in place for the View and Controller parts. The details of this class will not be laid out here rather a few central features will be discussed.

First and foremost, the class contains some configuration options, in the form of static class fields. The ones requiring some explanation are listed in the following figure.

```
// Constant (pixels) used to draw fields that fits screen
// resolution of 1024 * 768
// Also used when translating mouse click coordinates to row
// and column coordinates
// Used also for checking if a click on the board is valid
// 100 / CONSTANT must produce a real int number, ie. 1, 2, 4,
// 5, 10, 20, 25, 50 are valid numbers
private static final int CONSTANT = 50;

// Distance in pixels from NW corner of Center area to NW
// corner of guiBoard
```

```
private static final int gridOffsetX = 15;
private static final int gridOffsetY = 15;

// Offset in pixels that, for each field square, is non-
// clickable
private static final int nonClick = 5;
```

Figure 3-9 Configuration fields - GUI.java

The code comments, denoted by '//', explains a lot about the different class fields. Perhaps the class field `CONSTANT` is the only variable requiring further explanation.

3.4.1 The CONSTANT class field

The reason for introducing a constant in the GUI class has to do with the approach taken in drawing the game board fields on the screen. Since each of these game board fields is not represented as a separate container object, the role of the `CONSTANT` field is to draw the game board grid, with fields of appropriate size in pixels. For example, a size of 100 pixels (width and height) for each field is too large for a computer screen with a resolution of 1024 x 768 pixels. So an option to adjust the size of the game board on screen is needed.

Since grid fields are not contained within individual container objects, we cannot associate any event listeners with each field. This introduces a need for translating the screen coordinates (from mouse clicks) into coordinates that can be used for passing to the methods of the Board class. This is done by relying on calculations that use the value of the `CONSTANT` field, along with the integer data type of Java, so decimals can be ignored. The following figure gives an example of such a calculation.

```
int y = ((realy - gridOffsetY) * (100 / CONSTANT)) / 100
```

Figure 3-10 Calculation with CONSTANT class field

The 'trick' of this calculation is that by dividing the number 100 with the `CONSTANT` we take into account that a board field size of 100 x 100 pixels isn't possible. The translation from the clicked coordinates to the game board coordinates would have been easier, had this been the case; the first digit of each coordinate value could have been extracted to use as the game board coordinate. The calculation listed solves this problem, by multiplying the coordinate in question (taking into account the offset from NW corner, if present) with the relationship between the `constant` value and 100. This yields a number which by simple division by 100 can be considered a valid coordinate for a game board field, when it is of data type integer, since this data type doesn't store decimals.

3.5 The Coordinate class

This class merely serves as a tool for representing coordinates; hence it has only 2 central class fields, which are both integers, representing an x value and a y value. The only thing left to note about this class, then, is the definition of an `equals()` method, that overrides Java's default definition. The `Coordinate.equals(Coordinate coord)` method returns true if 2 objects of type Coordinate have equal `x` and `y` values. According to common Java principles (Horstmann 2007: 716), when one defines a custom `equals()` method, one must also define a compatible `hashCode()` method, and so this has been done.

3.6 Initializing the Battleship game

After considering various important aspects of the software, the following figure illustrates the complete initialization procedure when executing the game. The numbers indicate the order in which instantiation of new objects (or method calls) must occur. The dashed line indicate when one of the previously instantiated objects are passed to the constructor of another class. Class methods and fields have been left out, unless instantiation depends on them, or they are vital to the game. The following initialization is also used in Tester.java, the class containing a `main()` method for executing the game.

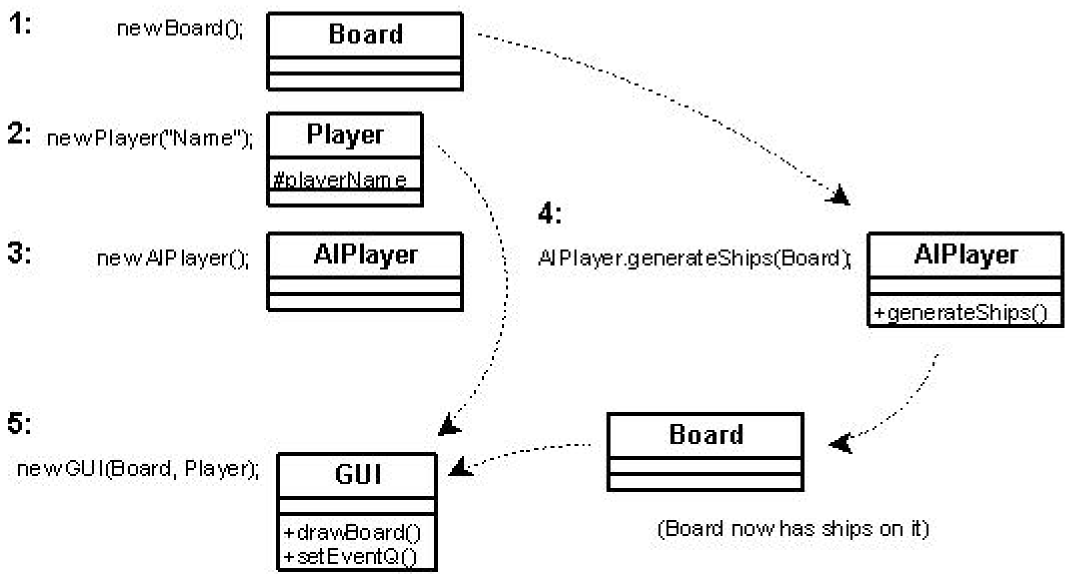


Figure 3-11 Initialization sequence

4 Install and run the Battleship game

This short chapter contains instructions on installing and running the Battleship game. The steps to take in order to get the program running, depends on the environment in which it is installed. The instructions presented here, are for installation on a Windows PC, with a recent version of Windows (XP or Vista). If installing on GNU/Linux or UNIX, there are no drive letters, and a full path is represented as something similar to: `/home/loft/src`. Also UNIX and GNU/Linux are case sensitive environments. If these issues are taken into account, the instructions in this chapter will get the Battleship program up and running on GNU/Linux and UNIX as well.

For all installation scenarios, at least version 1.5 of the Java Runtime Environment (JRE) must be installed on the computer. The latest version can be obtained from http://www.java.com. If you intend to compile the source code yourself, then install the Java Development Kit (JDK), which will also install the JRE.

Under Resources you can see where to get both source code and precompiled Java .class files. Both packages include the custom Swing package, developed by, and courtesy of, Mads Rosendahl, University of Roskilde.

4.1 Installation and execution in Eclipse

If the Battleship game is to be installed in an Integrated Development Environment (IDE), the required installation procedure of course depends on the particular IDE. For installation into an Eclipse environment, obtain the source code (see Resources below), and use Eclipse's 'Import…' function (located under the 'File' menu), and adjust your project settings to support the Battleship game. You must also import the JCanvas, JEventQueue and JBox class files developed by Mads Rosendahl of Roskilde University, into the same project as the Battleship game. Finally you must make sure, that the project has access to the default Java Runtime Environment System Library, but this is normally created by default in Eclipse. To execute and run the Battleship program from within Eclipse, one would typically have to open the class containing the `main()` method (Tester.java), and then hit Shift + Alt + X and then J.

4.2 Using precompiled binaries

If you haven't got an IDE installed on your machine, the easy way to get up and running, is to download the pre-compiled version of the game (see under Resources). This way you get a compressed bundle of Java .class files, which you will need to unzip (the unzipping functionality is built in to almost all fairly recent operating systems). Unzip the package into a known location and then open a terminal (Unix/Linux) or command prompt (Windows). Then type `cd <path-to-save-dir>` and check that this directory contains one folder: `dk`. <path-to-save-dir> is the full path, including drive letter, to the directory to which you unzipped the downloaded package. To execute the Battleship game, type the following command: `java dk.ruc.loft.battleships.Tester` and play the game.

4.3 Compiling from source

To compile the source code on your own, obtain the downloadable source code (see under Resources). Unzip the compressed file to a directory somewhere on your computer, and then open a terminal or command prompt and type `cd <path-to-save-dir>` and check that this directory contains one folder: `dk`. `<path-to-save-dir>` is the full path, including drive letter, to the directory to which you unzipped the downloaded package. Execute the following command to compile the source code: `javac dk\ruc\loft\battleships\*` and when this command finishes, execute `java dk.ruc.loft.battleships.Tester` and play the game.

4.4 Resources

Java Runtime Environment or Development Kit: http://www.java.com/
Eclipse IDE: http://www.eclipse.org/downloads/
Battleship program .class files: http://akira.ruc.dk/~loft/battleship_class.zip
Battleship program source code: http://akira.ruc.dk/~loft/battleship_source.zip

Mads Rosendahls Swing classes: http://akira.ruc.dk/~madsr/swing.html

5 Playing Battleship

Since the previous chapters have all been concerned with the design and coding of the Battleship game, this chapter is going to examine and document that the design and code actually works and is reliable. Initially, potentially critical issues will be discussed – what could go wrong and what are possible flaws in the software? Then a few remarks about adequate software testing will be made – what is adequate testing? Finally the approach taken in this chapter is described – how is the Battleship game tested?

5.1 Testing software

As have been stated earlier in this report, the core parts of the Battleship game are the Board, (AI)Player and Ship classes. From a user perspective, also the GUI class is central since this delivers the whole interface to view and control the game (which again brings to mind the Model-View-Controller concept described earlier). In this chapter the focus will be on the above mentioned classes, but in a real world scenario all classes would require thorough testing.

5.1.1 Testing core components

Section 2.3.5 (Class design) described the data structures developed for the Battleship game; the Board class is vital to the game, hence consideration must be given to possible errors that could occur here. Obvious issues of importance are the placing of ships and shots. All ships and shots should lie or be fired within board boundaries, and ships should not be able to lie on top of each other. Since a common mistake in programming, according to Horstmann (2007: 233), is off-by-one errors, one should ensure that placing ships or firing shots at the outermost fields of the game board work like they are supposed to. Throughout the programming phase of this project, I've been using console output (`system.out.println()`) to ensure that all these critical cases do not generate errors ('on-the-fly unit testing' would be an appropriate term for this). For example I deliberately tried placing ships on occupied game board fields, and outside the game board.

Other core features of interest are: The method `AIPlayer.generateShips()`, the array lists `Ship.coords` and `Board.fleet` and their accessor and mutator methods, and several other methods pertaining to the GUI class, for example the translation of mouse clicked coordinates to game board coordinates. Also the validation of mouse clicks in relation to field margins is important. The list of critical cases to test is long, and a systematic approach would be required if any guarantees about software stability would have to be issued (if such guarantees can ever be issued?). One approach would include testing all classes in isolation (unit testing), and for each feature added to a class, a new feature should be added to the tester class, a least if full test coverage is the goal. This way a whole library of test cases would be compiled over time, and comprehensive white-box testing would then be achieved.

5.1.2 Black-box testing the Battleship game

This report will not include unit testing of data structures and methods; rather an attempt will be made to simplify the testing procedure, by displaying a sequence of 6 screenshots, obtained from playing a complete round of Battleship – in essence black box testing. The ships in this game of Battleship have been placed by the AIPlayer, but a scenario has been chosen, where whole ships and parts of ships have been placed on the outskirts of the game board. Also the 6 screenshots will demonstrate the way incorrect user input will be handled (if the player clicks too close to a neighbouring field or fires at an already hit field). Furthermore the screenshots should demonstrate that the above mentioned data structures (and associated methods) work appropriately – if they did not work, the placing, hitting and sinking of ships would not succeed. Finally, by using screenshots, it is possible to convey the look and feel of the Battleship game, which is important from a user perspective.

The outlined testing procedure will of course not guarantee that the software is completely free of bugs; hopefully it will aid in convincing the reader that the software is stable and reliable.

5.2 Playing a round of Battleship

5.2.1 Starting the game

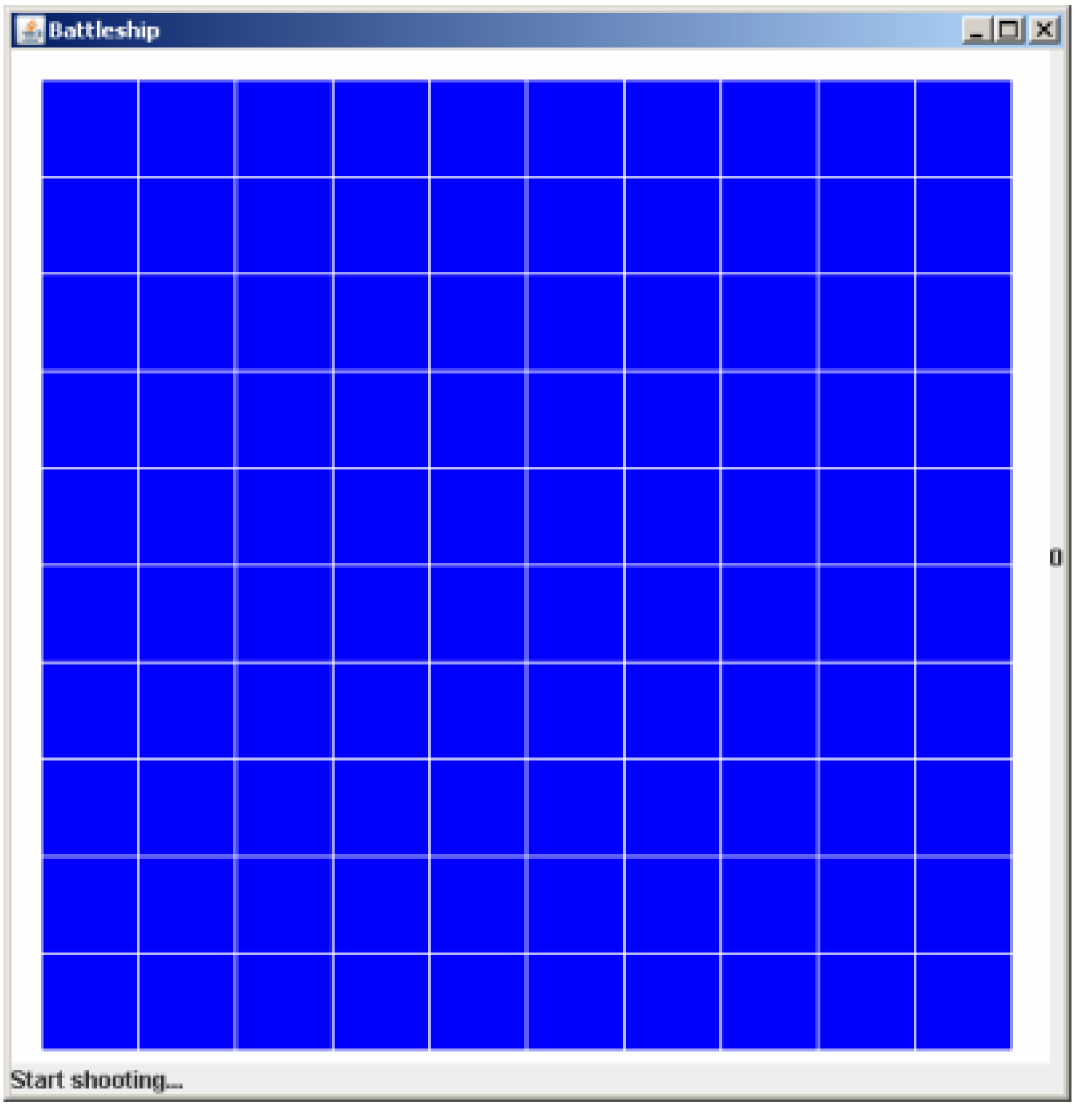
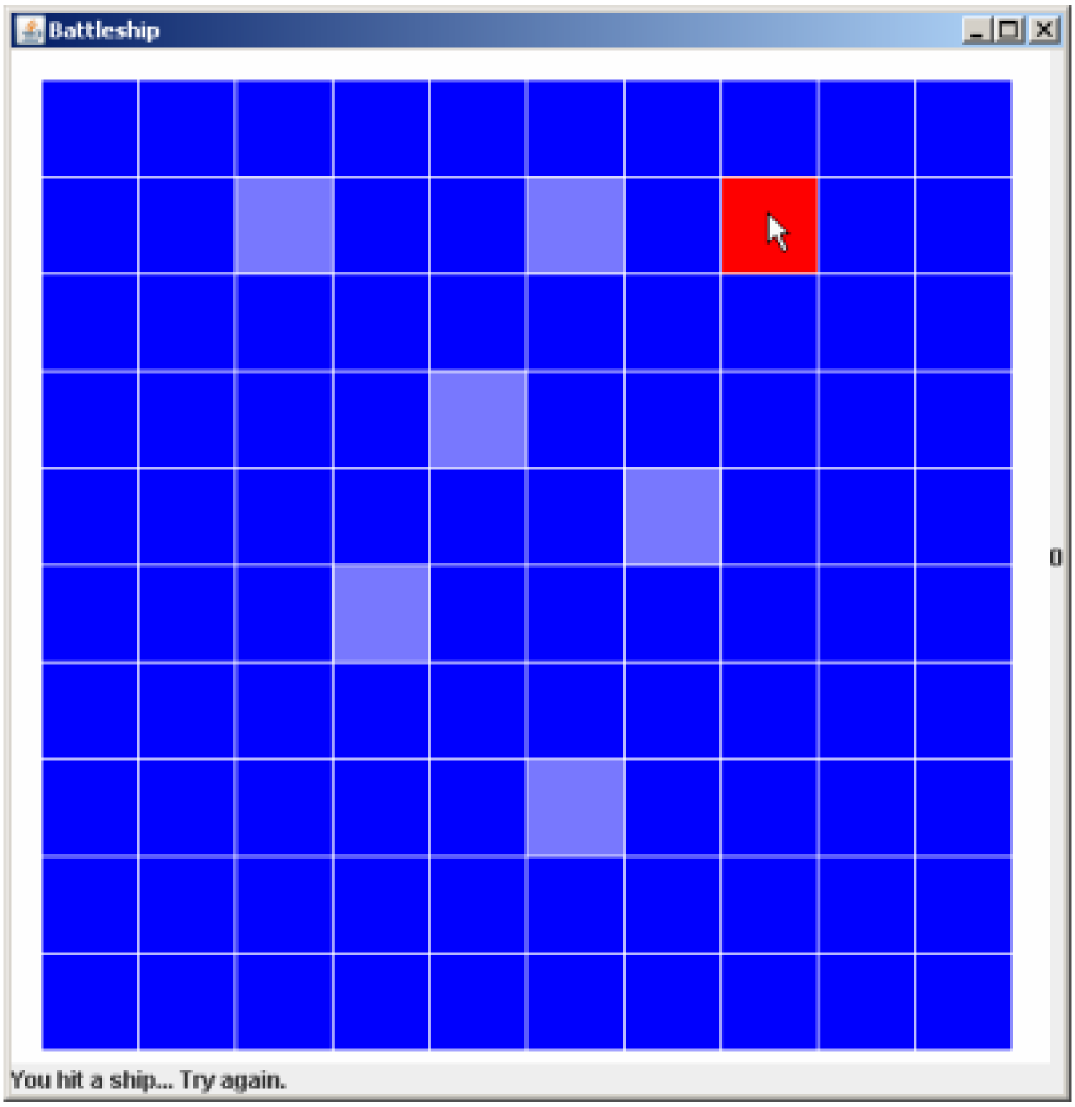


Figure 5-1 Initial screen

This screenshot shows the first screen that a player is presented with. It is the Battleship game board, which consists of 10 x 10 fields. Notice the ‘0’ on the right – this displays the players score. Also notice the text at the bottom ‘Start shooting…’ – this area relays messages to the player. Finally, notice that the game board has a small offset to the north-west corner of the window. This shows the configuration of the `gridOffsetX` and `gridOffsetY` class fields in the GUI class; in this example they hold a value of 15 (pixels) each.

5.2.2 A ship was hit

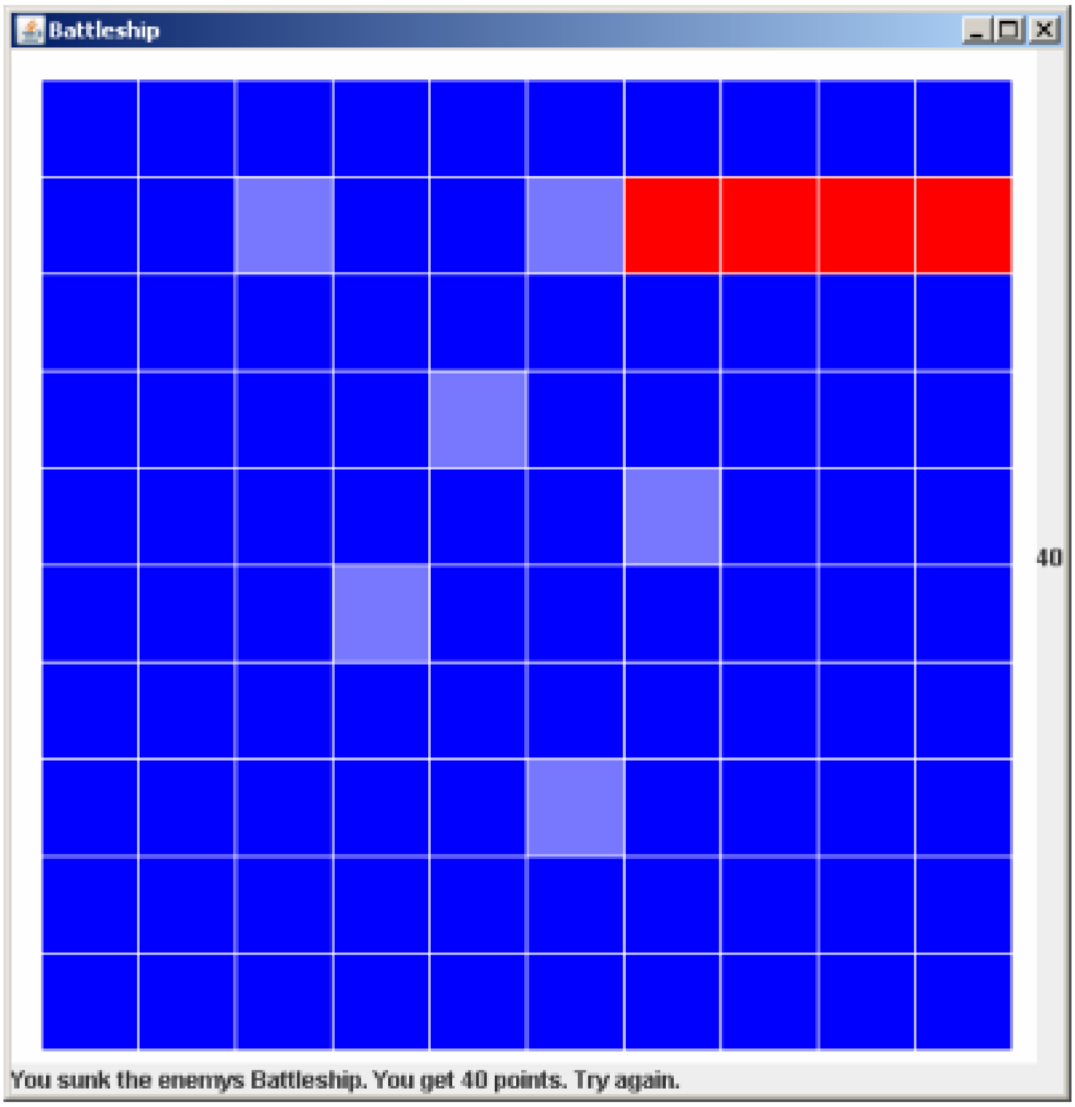


You hit a ship... Try again.

Figure 5-2 Shots fired and a ship was hit

Now the player has started playing the game. Observing the game board in Figure 5-2, 6 fields have been hit (they are brighter than the other game board fields). Upon firing the seventh shot, the player hits part of a ship. This information is relayed to the player, through the status message at the south-west corner of the window, and also through the red colour of the field upon which the ship is placed.

5.2.3 A ship was sunk

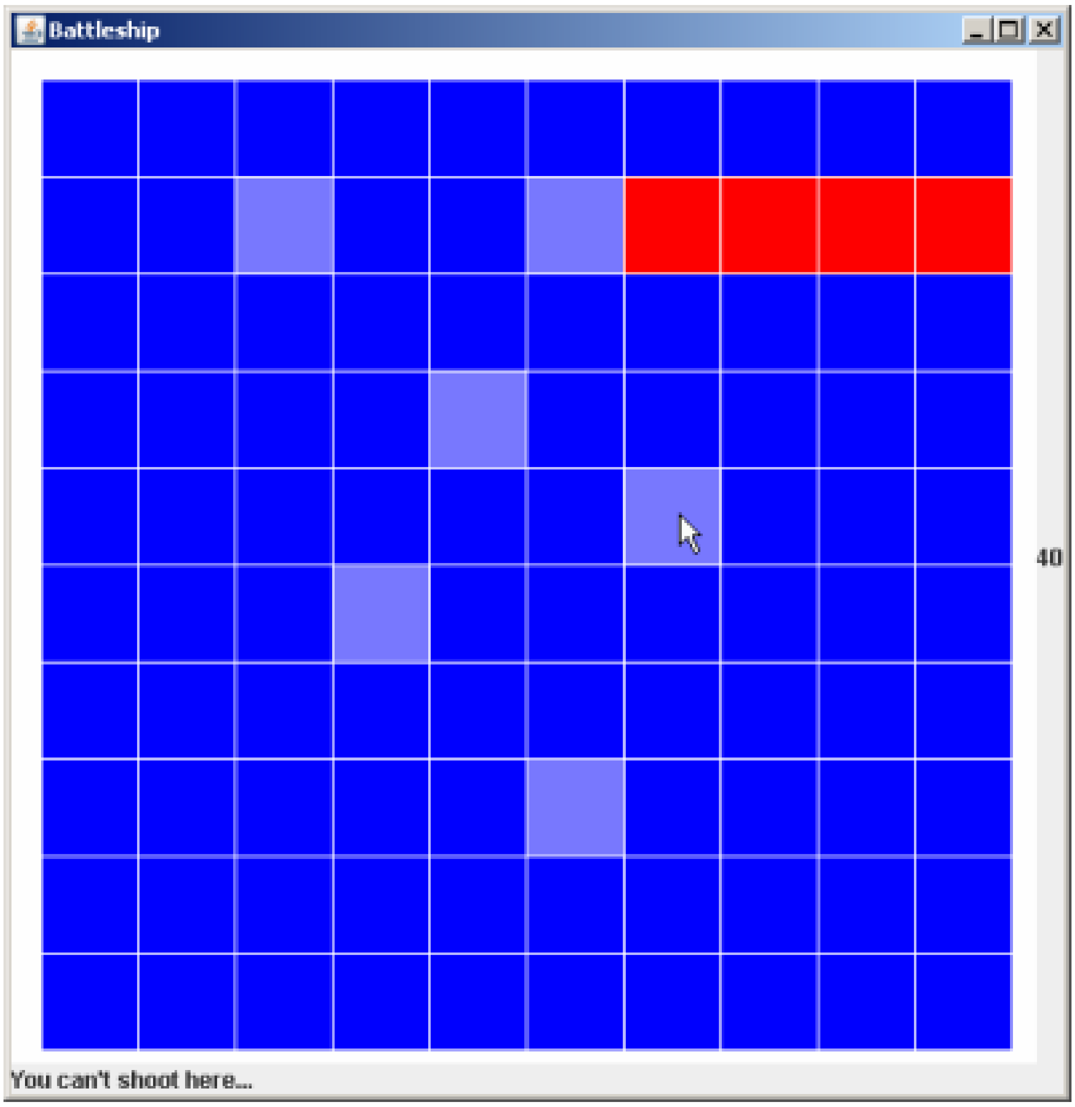


You sunk the enemys Battleship. You get 40 points. Try again.

Figure 5-3 Entire ship destroyed

In Figure 5-3 a situation occurs, where the player actually sinks a ship. A row of 4 fields are now drawn in a red colour indicating that a ship was present here. The player is also informed about her success in the status message area, and about the type of ship that she sunk. The status message also tells the player about the points assigned for sinking this particular ship, and, observing the east area of the game window, the players score has been incremented by the number of points given for the enemy's battleship, in this case 40.

5.2.4 Field already hit

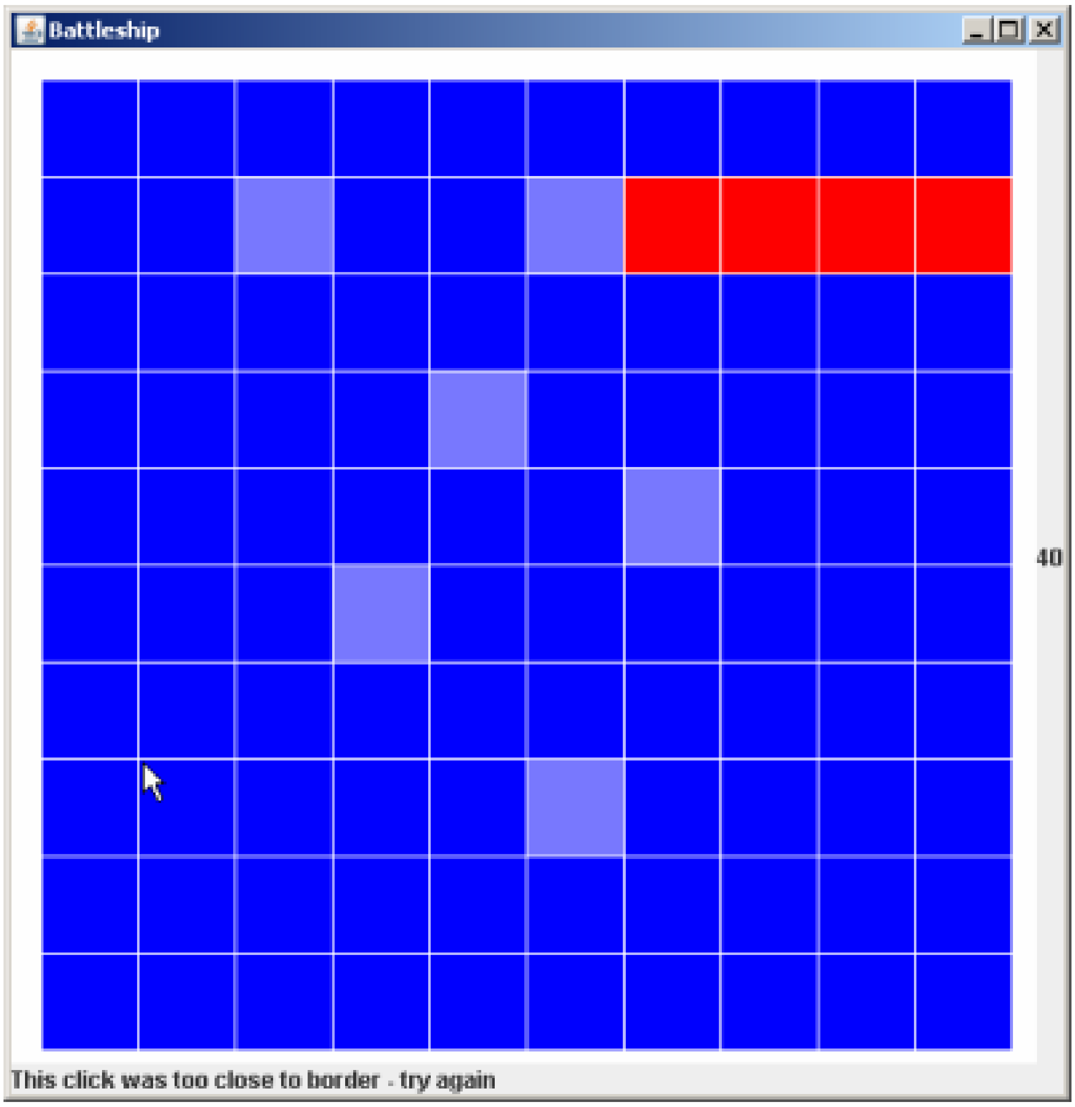


You can't shoot here...

Figure 5-4 Shooting at previously hit field

If the player accidentally fires a shot at an already hit game board field, a message will be displayed stating that it is not possible to fire at this particular area. If the player tried to fire at a field that previously held a ship, the message given would be identical.

5.2.5 Too close to field border

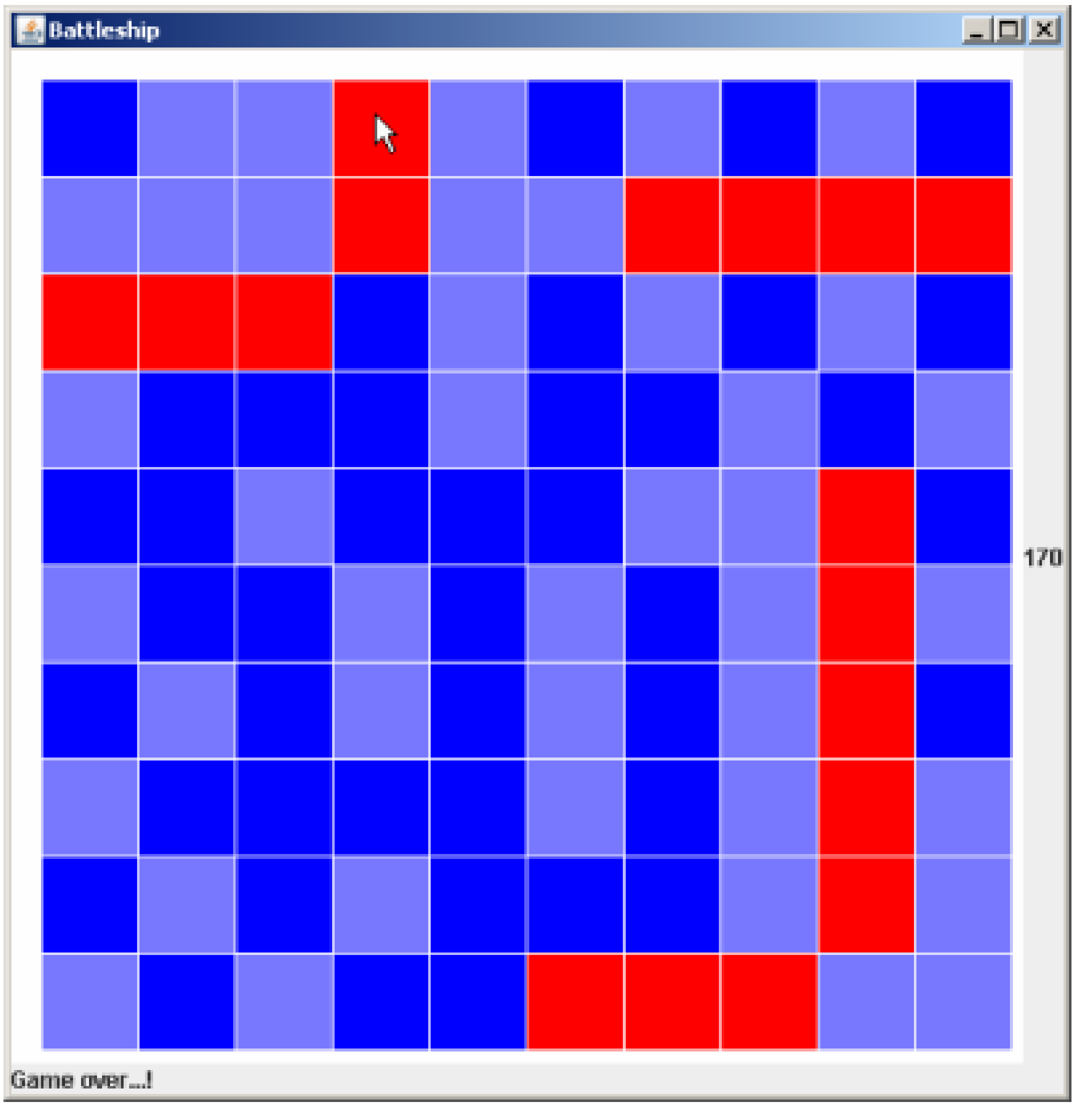


This click was too close to border - try again

Figure 5-5 Clicking within the no-click area

In this screenshot an example is given of how the game reacts when a player clicks too close to a field’s border. As always, information is displayed in the south-west area of the game window. Currently the margin value is 5 pixels; it can be configured by adjusting the value of the class field `nonClick` in the GUI class. This feature exists so the player (and the developer) can always feel sure about which field she actually clicked in.

5.2.6 All ships sunk – game over!



Game over...!

Figure 5-6 All ships are gone, and game is over...

As can be seen in the status message field, the game is now over. This information is displayed since the player has destroyed the last remaining ship on the game board. Several things are worth noticing. First of all, the score have steadily been incremented since the first ship was sunk. This can easily be verified, since each ship gives a number of points equal to its length multiplied by 10. Thus 20 + 30 + 30 + 40 + 50 equals 170, the final score. Also this screenshot illustrates the fact that the ships are placed all over the board: They can lie entirely on the outer fields of the game board, or they can be only partly located on outer fields. Ships are also oriented in both horizontal and vertical directions, and they don't lie on top of each other, or outside the game board. Also the above figure illustrates, that all 5 ships are present, as they indeed should be.

5.3 Final notes on testing

This final screenshot concludes the black-box testing procedure. The Battleship game has been successfully played from start to end; without errors. Ships have been placed, hit and sunk, points have been given, user input errors have been committed and the game is over. To conclude that the game is bug-free would be risky, but functional vital components ensure that the game and game features work, also when considering possible pitfalls as was outlined in the previous section. The black-box testing performed in this chapter, should render the fulfilment of the objectives stated in the beginning of this report probable.

6 Concluding remarks

Through examining the field of interest, Battleship games, and formulating the requirements needed for a basic implementation of such a game, a plausible software design was arrived at, decided and implemented.

The preliminary investigations concerning previous implementations of the Battleship game, revealed a fairly wide range of applications. At this point in time, of course, the obvious platform for implementing this type of game is in an online, web based context, as was described earlier. Although different flavours exist the characteristics and game play of the various implementations were very similar, if not identical, between the examples that were examined in this report. In this manner, the requirements of this Battleship game were not different from other implementations, although they might have been somewhat minimal.

Stipulated by the requirements stated in the projects preliminary phase, certain possible design alternatives were discussed. This discussion revealed that certain key decisions in the design process, will dictate other choices later on in the process, as the decision to use a certain data structure for storing the game board status showed. This illustrates the importance of working thoroughly when contemplating software designs. The discussion on design also clearly showed that there are several possible ways to design software, and that it is hard, if not impossible, to decide on a single best option. Sometimes you just have to make an informed choice, carefully weighing benefits and drawbacks.

By performing sound analysis and carefully thinking about software design, this project produced a functional and Java coded Battleship game, as was witnessed in the previous chapter of the report. The game can be installed, executed and run, with the computer automatically placing the ships on the game board, enabling a player to shoot at ships while scoring points.

The design process proved its justification, as did the Java language, since they both assisted in creating a design and code that fulfils the goals of this project.

If development of the Battleship game should continue in the future, it would certainly be interesting to look at the possibilities for implementing more AI functionality, similar to that of section 2.4.2, where several desirable options were discussed. It would also make sense, to develop a manual ship placing functionality, so the human player would be able to place ships herself. If the Battleship game featured both AI functionality and manual placing of ships, it would be a really complete solution that could be distributed on the web or run as an applet on one's homepage.

6.1 The work process

When studying at the University of Roskilde, it is very common, although not compulsory, to work on student projects in groups of 3-5 students. This way one's people skills are trained as well as ones academic skills. Also there is the advantage of having other people to discuss with, the challenges involved in working on a project. If you're having difficulties figuring something out, chances are that one of the other members of the group will have some useful ideas to contribute to solving the problem at hand. Several perspectives on the project will of course exist, probably as many perspectives as there are students in the group.

That being said, I chose to work on my own. There are many advantages to this, that I feel outweigh the benefits of collaborating with others. First of all, a lot of time can be saved. Decisions can be made when necessary, and not only when the whole group is together. Secondly, the work does not need to be planned nearly as much. There is no risk of redoing someone else's work, and random parts of the project can be started whenever it is convenient to do so. Also, on the logistical side of things, a lot of time can be saved planning when and where to meet next and the schedules of all the group members do not have to be taken into account.

I feel that this project in its coding phase, has shared a lot of characteristics with the eXtreme Programming development process, as Xiaoping Jia sees it (Jia 2002: 15-16). From the very first part of the code written, I have always had a functional software system. It has been very simple in the beginning, but the software has gradually become more and more comprehensive. So this approach is actually an iterative process, always keeping the code executable, and gradually expanding it. Between these iterations, I have frequented a piece of paper using a pencil – sketching, drawing and calculating software features. This way, all aspects of the design process have been part of each iteration. A part of the eXtreme Programming development process, that I highly cherish, is the notion of sane work hours, simply because it makes sense not to work when already exhausted. A great deal of errors will sneak into the code, and there's only one person (me) to debug the software later.

The report has been challenging to write, since this is my first Computer Science project. I have a background in Social Science and have adopted a certain way to present ideas and write student projects. Thanks to the project supervisor, Mads Rosendahl, I didn't panic completely. I hope the report explains and clarifies the ideas I've had regarding the Battleship game.

7 Literature and sources

Books:

Horstmann, Cay: Big Java, 3rd Edition, John Wiley & Sons, Inc., 2007. ISBN: 9780470105542

Jia, Xiaoping: Object Oriented Software Development using Java, 2nd Edition, Addison-Wesley, 2002. ISBN: 0201737337

Web:

Flash object Battleship implementation:
http://www.battleships.f-active.com/
- Content present on 11th of December 2007.

Two-player Battleship implementation:
http://spil.tv2.dk/spil/slagskibe
- Content present on 13th of December 2007.

Model View Controller concept:
http://en.wikipedia.org/wiki/Model_view_controller
- Content present on 11th of December 2007.

Information on the history of the Battleship game:
http://www.gamesmuseum.uwaterloo.ca/VirtualExhibits/Whitehill/Battleship/index.html
- Content present on 11th of December 2007.

Information on the history of the Battleship game:
http://neweranet.com/battleship/battleshipinfo.htm
- Content present on 11th of December 2007.

Information on the history (and rules) of the Battleship game:
http://en.wikipedia.org/wiki/Battleship_(game)
- Content present on 11th of December 2007.

Client/server multiplayer implementation ('batalla naval'):
http://batnav.sourceforge.net/batnav-en.html
- Content present on 11th of December 2007.

Other resources:

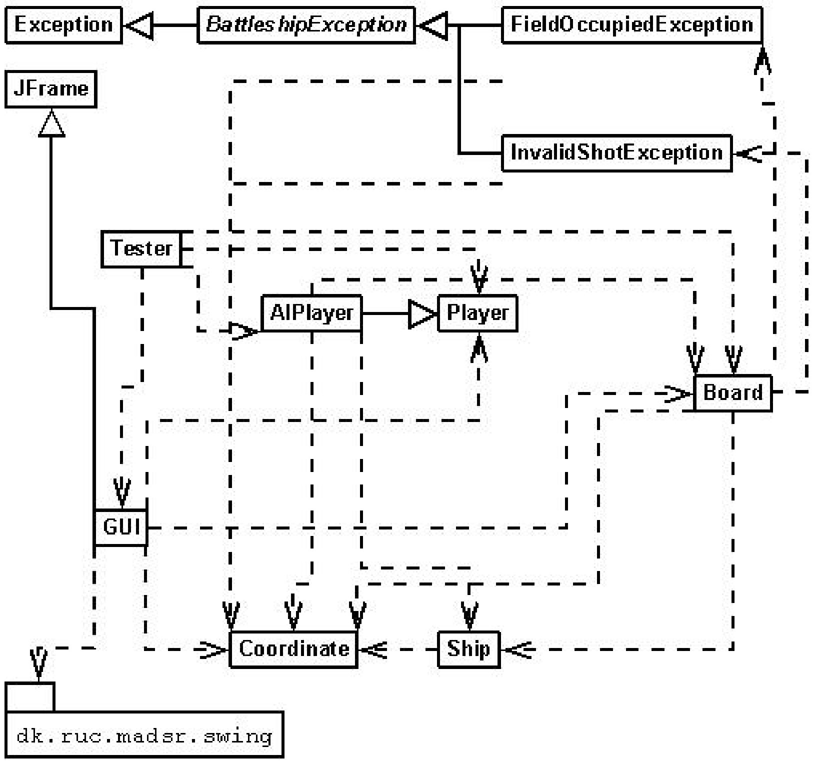
Java Runtime Environment or Development Kit: http://www.java.com/
Eclipse IDE: http://www.eclipse.org/downloads/
Battleship program .class files: http://akira.ruc.dk/~loft/battleship_class.zip
Battleship program source code: http://akira.ruc.dk/~loft/battleship_source.zip

Mads Rosendahls Swing classes: http://akira.ruc.dk/~madsr/swing.html

Appendix A – Java code

This appendix contains the Java code that was developed for the Battleship game. The appendix has been structured by means of an extra index – listing the different class files and their page numbers in *this section* of the report. This index can be seen at the bottom of this page, and on the following page.

To recap, observe the following diagram of the class structure (it was also presented in section 3). A Tester class has now been added. This class contains the `main()` method for executing the Battleship game.



It was stated earlier in the report, that the core components of the Battleship game were the Board, Ship, Player, especially AIPlayer and GUI classes. Along with the Tester class, these are listed first, as they are considered a priority.

The code presented in this appendix, is using the usual `code` typeface. Unfortunately the code has no syntax highlighting. There are a few line breaks in the code sections, as they are presented here, on A4-sized pages. I have eliminated line breaks when I felt it was distracting to read, primarily inside Javadoc comments. I have left those line breaks that occur inside methods or constructors as they are. I feel that the '`;`' ending the line suggests clearly to the reader when the line ends. Each class has its own sequence of line numbers in the left margin – this is of little importance to the reader, but should one wish to discuss the code with others, it is convenient to refer to line numbers instead of only page numbers.

Finally we have the auxiliary classes. These are the Coordinate and exception classes. These can be found on the pages listed below.

The Coordinate class – Coordinate.java:	page 19
The Exception class – BattleshipException.java:	page 20
The FieldOccupiedException class – FieldOccupiedException.java:	page 21
The InvalidShotException class – InvalidShotException.java:	page 22

1. The Tester class – Tester.java

```java
package dk.ruc.loft.battleships;

public class Tester {

  public static void main(String[] args) {
    Board theBoard = new Board();

    AIPlayer ai = new AIPlayer();

    Player player = new Player("Player");

    ai.generateShips(theBoard);

    GUI theGUI = new GUI(theBoard, player);

    theGUI.drawBoard();

    theGUI.setEventQ();

    /* Print board on console
    for(int y = 0; y < theBoard.getYdim(); y++) {
      System.out.println();
      for(int x = 0; x < theBoard.getYdim(); x++ ) {
        int tmp = theBoard.getFieldStatus(x, y);
        System.out.print("[" + tmp + "]");
      }
    }
    */
  }
}
```

2. The Board class – Board.java

```java
package dk.ruc.loft.battleships;

import java.util.Iterator;
import java.util.ArrayList;

/**
 * Class that represents the game board.
 */
public class Board {

  // Default no. of fields in board grid
  private static int boardX = 10;
  private static int boardY = 10;

  // Field status array
  // 0: empty, not hit
  // 1: empty, but hit
  // 2: not empty, not hit
  // 3: not empty, but hit
  private static int[][] fieldStatus;

  // An arraylist for storing the ships
  // Need this collection for keeping track of
  // ships their points, names and status (hit or sunk)
  ArrayList<Ship> fleet = new ArrayList<Ship>();

  /**
   * Constructs board with default size and initialize status fields
   */
  public Board() {
    // Set array size
    fieldStatus = new int[boardX][boardY];
    // Initialize all fields as empty, not hit
    for(int i = 0; i < boardX; i++) {
      for (int j = 0; j < boardY; j++) {
        fieldStatus[i][j] = 0;
      }
    }
  }

  /**
   * Constructs a board of given size and initialize status fields
   * @param the x size
   * @param the y size
   */
  public Board(int x, int y) {
    boardX = x;
    boardY = y;
    // Set array size
    fieldStatus = new int[x][y];
    // Initialize all fields as empty, not hit
    for(int i = 0; i < boardX; i++) {
      for (int j = 0; j < boardY; j++) {
        fieldStatus[i][j] = 0;
      }
    }
  }

  /**
   * Get field status
   * @param x coordinate
   * @param y coordinate
```

```
     * @return the status of x, y field
     */
    public int getFieldStatus(int x, int y) {
      return fieldStatus[x][y];
    }

    /**
     * Check if it's OK to place ship
     * @param ship object
     * @return true if ship can be placed, false if not
     */
    public boolean canPlaceShip(Ship theShip) {
      Iterator<Coordinate> iterate = theShip.coords.iterator();
      while(iterate.hasNext()) {
        Coordinate fieldCoord = iterate.next();
        int x = fieldCoord.getX();
        int y = fieldCoord.getY();
        if(x >= boardX || y >= boardY) {
          return false;
        }
        // If ship allready in this field
        if(fieldStatus[x][y] != 0) {
          return false;
        }
      }
      return true;
    }

    /**
     * Places a ship on the board
     * @param ship object
     */
    public void placeShip(Ship theShip)
      throws FieldOccupiedException {
      Iterator<Coordinate> iterate = theShip.coords.iterator();
      while(iterate.hasNext()) {
        Coordinate placeCoord = iterate.next();
        int x = placeCoord.getX();
        int y = placeCoord.getY();
        // If ship allready in this field
        if(fieldStatus[x][y] != 0) {
          throw new FieldOccupiedException(placeCoord, "Field allready occupied");
        }
        else {
          // Set fields to not empty, not hit
          fieldStatus[x][y] = 2;
        }
      }
      // Add the ship to the fleet
      fleet.add(theShip);
    }

    /**
     * Check if it's OK to place shot
     * @param coordinate to hit
     * @return true if OK to place shot, false if not
     */
    public boolean canPlaceShot(Coordinate coord) {
      int x = coord.getX();
      int y = coord.getY();
      int field = fieldStatus[x][y];
      // If field value 0 or 2, it's OK to shoot
      if(field == 0 || field ==  2) {
        return true;
```

```java
    }
    return false;
  }

  /**
   * Places shot on the board
   * @param the coordinates to hit
   * @return the result (1 or 3 - the two possible outcomes)
   */
  public int placeShot(Coordinate coord)
    throws InvalidShotException {
    int x = coord.getX();
    int y = coord.getY();
    // If this field has been hit before, throw exception
    if(fieldStatus[x][y] == 1 || fieldStatus[x][y] == 3) {
      throw new InvalidShotException(coord, "This field has allready been hit");
    }
    else if(fieldStatus[x][y] == 0) {
      fieldStatus[x][y] = 1;
      return fieldStatus[x][y];
    }
    else {
      fieldStatus[x][y] = 3;
      // We have a succesful shot, and the ship must remember that it has been
      // hit
      for(Ship ship : fleet) {
        if(ship.hasCoordinates(coord)) {
          ship.shipHit(coord);
        }
      }
      return fieldStatus[x][y];
    }
  }

  /**
   * Get board X dimension
   * @return an int representing the no of fields on X-axis
   */
  public int getXdim() {
    return boardX;
  }

  /**
   * Get board Y dimension
   * @return an int representing the no of fields on Y-axis
   */
  public int getYdim() {
    return boardY;
  }

  /**
   * Gets the name of a ship if it was destroyed. This method must be called
   * after every shot.
   * @param the field coordinate
   * @return the name of the ship it is was destroyed (if noMoreShip returns
   * true).
   * Or an empty string if ship was not destroyed.
   */
  public String shipNameIfKill(Coordinate theCoord) {
    for(Ship ship : fleet) {
      if(ship.noMoreShip()) {
        return ship.getName();
      }
    }
```

```
      return "";
    }

    /**
     * Gets the points for the ship if it was destroyed, and deletes the ship from
     * the fleet.
     * This method must be called after every shot, and after the shipNameIfKill()
     * method.
     * @param the coordinate of the ship
     * @return the points for ship, or 0 if it was not destroyed.
     */
    public int shipPointsIfKill(Coordinate theCoord) {
      for(int i = 0; i < fleet.size(); i++) {
        if(fleet.get(i).noMoreShip()) {
          int retval = fleet.get(i).getPoints();
          fleet.remove(i);
          return retval;
        }
      }
      return 0;
    }

    /**
     * Game is over if fleet arraylist is empty
     * @return true if game is over, false if not
     */
    public boolean isGameOver() {
      if(fleet.isEmpty()) return true;
      else return false;
    }
  }

```

3. The Ship class –Ship.java

```
package dk.ruc.loft.battleships;

//import java.util.Set;
//import java.util.HashSet;
import java.util.ArrayList;
/**
 * Class for representing a ship
 */
public class Ship {

  // Coordinates for the different ship parts
  private static Coordinate coord1 = null;
  private static Coordinate coord2 = null;
  private static Coordinate coord3 = null;
  private static Coordinate coord4 = null;
  private static Coordinate coord5 = null;

  // Set for storing Coordinates - iteration is then possible
  // Set<Coordinate> coords = new HashSet<Coordinate>();

  // An array list for storing this ships coordinates
  ArrayList<Coordinate> coords = new ArrayList<Coordinate>();

  // An int for the points that this ship gives when sunk
  private int points = 0;

  // Type (and name) of ship
  private String shipName = "";

  /**
   * Constructs a ship of size 2
   * @param 1st coordinate set
   * @param 2nd coordinate set
   */
  public Ship(Coordinate coord1, Coordinate coord2, String name) {
    this.coord1 = coord1;
    this.coord2 = coord2;
    coords.add(coord1);
    coords.add(coord2);
    points = 20;
    shipName = name;
  }

  /**
   * Constructs a ship of size 3
   * @param 1st coordinate set
   * @param 2nd coordinate set
   */
  public Ship(Coordinate coord1, Coordinate coord2, Coordinate coord3, String
name) {
    this.coord1 = coord1;
    this.coord2 = coord2;
    this.coord3 = coord3;
    coords.add(coord1);
    coords.add(coord2);
    coords.add(coord3);
    points = 30;
    shipName = name;
  }

  /**
   * Constructs a ship of size 4
   * @param 1st coordinate set
   * @param 2nd coordinate set
   */
  public Ship(Coordinate coord1, Coordinate coord2, Coordinate coord3,
Coordinate coord4, String name) {
    this.coord1 = coord1;
    this.coord2 = coord2;
    this.coord3 = coord3;
    this.coord4 = coord4;
    coords.add(coord1);
    coords.add(coord2);
    coords.add(coord3);
    coords.add(coord4);
    points = 40;
    shipName = name;
  }

  /**
   * Constructs a ship of size 5
   * @param 1st coordinate set
   * @param 2nd coordinate set
   */
  public Ship(Coordinate coord1, Coordinate coord2, Coordinate coord3,
Coordinate coord4, Coordinate coord5, String name) {
    this.coord1 = coord1;
    this.coord2 = coord2;
    this.coord3 = coord3;
    this.coord4 = coord4;
    this.coord5 = coord5;
    coords.add(coord1);
    coords.add(coord2);
    coords.add(coord3);
    coords.add(coord4);
    coords.add(coord5);
    points = 50;
    shipName = name;
  }

  /**
   * Registers that this ship has been hit
   * @param the coordinate that is hit
   */
  public void shipHit(Coordinate coord) {
    for(int i = 0; i < coords.size(); i++) {
      if(coords.get(i).equals(coord)) {
        System.out.println("Removed a coordinate from ships own array");
        coords.remove(i);
      }
    }
  }

  /**
   * Checks if there are any parts left of this ship
   * @return true if there are no more coordinates - hence the ship is destroyed
   */
  public boolean noMoreShip() {
    return coords.isEmpty();
  }

  /**
   * Get the points for this ship
   * @return the points given for destroying this ship
   */
  public int getPoints() {
    return points;
  }

  /**
   * Tests whether this ship is placed on these coordinates
   * @param the coordinates to test
   * @return true if ship is on these coordinates, false if not
   */
  public boolean hasCoordinates(Coordinate theCoord) {
    for(Coordinate coord : coords) {
      if(coord.equals(theCoord)) {
      System.out.println("Ship.hasCoordinates(): true");
      return true;
      }
    }
    return false;
  }

  /**
   * Gets the name of the ship
   * @return a string representing the name of the ship
   */
  public String getName() {
    System.out.println("Ship.getName(): " + shipName);
    return shipName;
  }
}

```

4. The Player class – Player.java

```
package dk.ruc.loft.battleships;

public class Player {
  // The name of the player
  protected String playerName;
  // Player's score - initially 0
  protected int score = 0;

  /**
   * Default constructor
   */
  public Player(){
  }

  /**
   * Constructs player with playername
   * @param the playername
   */
  public Player(String name){
    playerName = name;
  }

  /**
   * Adds points to the players score
   * @param points
   */
  public void setScore(int points) {
    score += points;
  }

  /**
   * Method to get score
   * @return the score for this player
   */
  public int getScore(){
    return score;
  }
}
```

5. The AIPlayer class – AIPlayer.java

```java
package dk.ruc.loft.battleships;
/**
 * Class that extends Player class
 * Main feature of this class is to generate random shots
 * and random placement of ships.
 */
import java.util.Random;

public class AIPlayer extends Player {

  // Initialize ship field to be used for creating ships
  private static Ship ship;

  // Ship names
  private static String aC = "Aircraft carrier";
  private static String bS = "Battleship";
  private static String dS = "Destroyer";
  private static String sM = "Submarine";
  private static String mS = "Mine sweeper";
  /**
   * Constructs a player with name "Computer"
   */
  public AIPlayer() {
    playerName = "Computer";
  }

  /**
   * Method that autogenerates ships
   * @param the board to use
   */
  public void generateShips(Board theBoard) {
    // Used to generate pseudo-random coordinates
    Random random = new Random();
    // The highest number to generate
    int max = theBoard.getXdim();

    // Generate AirCraftCarrier - size 5
    boolean placeACC = true;
    while(placeACC) {
      // Direction of vessel
      // 0: East-west, 1: NS (add to X)
      int dir = random.nextInt(2);
      // Generate random offset coordinates
      int x1 = random.nextInt(max);
      int y1 = random.nextInt(max);
      // Offset coordinate
      Coordinate coord1 = new Coordinate(x1, y1);
      // If dir == 0 (add to Y, X stays the same)
      if(dir == 0){
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1, y1 + 1);
        Coordinate coord3 = new Coordinate(x1, y1 + 2);
        Coordinate coord4 = new Coordinate(x1, y1 + 3);
        Coordinate coord5 = new Coordinate(x1, y1 + 4);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, coord4, coord5, aC);
      }
      // If dir == 1 (add to X, Y stays the same)
      if(dir == 1) {
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1 + 1, y1);
        Coordinate coord3 = new Coordinate(x1 + 2, y1);
        Coordinate coord4 = new Coordinate(x1 + 3, y1);
        Coordinate coord5 = new Coordinate(x1 + 4, y1);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, coord4, coord5, aC);
      }
      // Test if ship can be placed, if true, place ship
      if(theBoard.canPlaceShip(ship)) {
        try {
          theBoard.placeShip(ship);
          // We're done placing aircraft carrier
          placeACC = false;
        }
        catch (FieldOccupiedException foe) {
          System.out.println("An error occured: " + foe);
          foe.printStackTrace();
        }
      }
    }

    // Generate BattleShip - size 4
    boolean placeBS = true;
    while(placeBS) {
      // Direction of vessel
      // 0: East-west, 1: NS (add to X)
      int dir = random.nextInt(2);
      // Generate random offset coordinates
      int x1 = random.nextInt(max);
      int y1 = random.nextInt(max);
      // Offset coordinate
      Coordinate coord1 = new Coordinate(x1, y1);
      // If dir == 0 (add to Y, X stays the same)
      if(dir == 0){
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1, y1 + 1);
        Coordinate coord3 = new Coordinate(x1, y1 + 2);
        Coordinate coord4 = new Coordinate(x1, y1 + 3);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, coord4, bS);
      }
      // If dir == 1 (add to X, Y stays the same)
      if(dir == 1) {
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1 + 1, y1);
        Coordinate coord3 = new Coordinate(x1 + 2, y1);
        Coordinate coord4 = new Coordinate(x1 + 3, y1);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, coord4, bS);
      }
        // Test if ship can be placed, if true, place ship
      if(theBoard.canPlaceShip(ship)) {
        try {
        theBoard.placeShip(ship);
        // We're done placing battleship
        placeBS = false;
        }
        catch (FieldOccupiedException foe) {
          System.out.println("An error occured: " + foe);
          foe.printStackTrace();
        }
      }
    }

    // Generate Destroyer - size 3
    boolean placeDS = true;
```

```
    while(placeDS) {
      // Direction of vessel
      // 0: East-west, 1: NS (add to X)
      int dir = random.nextInt(2);
      // Generate random offset coordinates
      int x1 = random.nextInt(max);
      int y1 = random.nextInt(max);
      // Offset coordinate
      Coordinate coord1 = new Coordinate(x1, y1);
      // If dir == 0 (add to Y, X stays the same)
      if(dir == 0){
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1, y1 + 1);
        Coordinate coord3 = new Coordinate(x1, y1 + 2);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, dS);
      }
      // If dir == 1 (add to X, Y stays the same)
      if(dir == 1) {
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1 + 1, y1);
        Coordinate coord3 = new Coordinate(x1 + 2, y1);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, dS);
      }
      // Test if ship can be placed, if true, place ship
      if(theBoard.canPlaceShip(ship)) {
        try {
        theBoard.placeShip(ship);
        // We're done placing destroyer
        placeDS = false;
        }
        catch (FieldOccupiedException foe) {
          System.out.println("An error occured: " + foe);
          foe.printStackTrace();
        }
      }
    }

    // Generate Submarine - size 3
    boolean placeSM = true;
    while(placeSM) {
      // Direction of vessel
      // 0: East-west, 1: NS (add to X)
      int dir = random.nextInt(2);
      // Generate random offset coordinates
      int x1 = random.nextInt(max);
      int y1 = random.nextInt(max);
      // Offset coordinate
      Coordinate coord1 = new Coordinate(x1, y1);
      // If dir == 0 (add to Y, X stays the same)
      if(dir == 0){
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1, y1 + 1);
        Coordinate coord3 = new Coordinate(x1, y1 + 2);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, sM);
      }
      // If dir == 1 (add to X, Y stays the same)
      if(dir == 1) {
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1 + 1, y1);
        Coordinate coord3 = new Coordinate(x1 + 2, y1);
        // Make ship
        ship = new Ship(coord1, coord2, coord3, sM);
      }
      // Test if ship can be placed, if true, place ship
      if(theBoard.canPlaceShip(ship)) {
        try {
        theBoard.placeShip(ship);
        // We're done placing submarine
        placeSM = false;
        }
        catch (FieldOccupiedException foe) {
          System.out.println("An error occured: " + foe);
          foe.printStackTrace();
        }
      }
    }

    // Generate mine sweeper - size 2
    boolean placeMS = true;
    while(placeMS) {
      // Direction of vessel
      // 0: East-west, 1: NS (add to X)
      int dir = random.nextInt(2);
      // Generate random offset coordinates
      int x1 = random.nextInt(max);
      int y1 = random.nextInt(max);
      // Offset coordinate
      Coordinate coord1 = new Coordinate(x1, y1);
      // If dir == 0 (add to Y, X stays the same)
      if(dir == 0){
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1, y1 + 1);
        // Make ship
        ship = new Ship(coord1, coord2, mS);
      }
      // If dir == 1 (add to X, Y stays the same)
      if(dir == 1) {
        // Make coordinates
        Coordinate coord2 = new Coordinate(x1 + 1, y1);
        // Make ship
        ship = new Ship(coord1, coord2, mS);
      }
      // Test if ship can be placed, if true, place ship
      if(theBoard.canPlaceShip(ship)) {
        try {
        theBoard.placeShip(ship);
        // We're done placing mine sweeper
        placeMS = false;
        }
        catch (FieldOccupiedException foe) {
          System.out.println("An error occured: " + foe);
          foe.printStackTrace();
        }
      }
    }
  }
}
```

6. The GUI class – GUI.java

```java
package dk.ruc.loft.battleships;

import javax.swing.*;
import java.awt.*;
import dk.ruc.madsr.swing.*;
import java.util.*;

public class GUI extends JFrame {

  // The board to display and use
  private static Board theBoard;

  // The frame to display things in
  private static Frame theFrame = new JFrame();

  // The canvas to draw the fields on
  private static JCanvas guiBoard = new JCanvas();

  // The event queue for processing user input
  private static JEventQueue eventQ = new JEventQueue();

  // Colors to use in board display
  private static Color blue = new Color(0, 0, 255);
  private static Color lightBlue = new Color(120, 120, 255);
  private static Color red = new Color(255, 0, 0);
  private static Color white = new Color(255,255,255);
  private static Color fieldColor = blue;

  // Stroke to use for painting grid lines
  private static BasicStroke gridStroke = new BasicStroke(1);

  // Containers, labels and so on
  private static JLabel info = new JLabel("Start shooting...");
  //private static JLabel scoreHeader = new JLabel("Score: ");
  private static JLabel score = new JLabel("0");

  // Constant (pixels) used to draw fields that fits screen resolution of 1024 *
  // 768
  // Also used when translating mouse click coordinates to row and column
  // coordinates
  // Used also for checking if a click on the board is valid
  // 100 / CONSTANT must produce a real int number, ie. 1, 2, 4, 5, 10, 20, 25,
  // 50 are valid numbers
  private static final int CONSTANT = 50;

  // Distance in pixels from NW corner of Center area to NW corner of guiBoard
  private static final int gridOffsetX = 15;
  private static final int gridOffsetY = 15;

  // Offset in pixels that, for each field square, is non-clickable
  private static final int nonClick = 5;

  // The player that plays this game
  private static Player player;


  /**
   * Constructs a new GUI object with graphics and event queue
   * @param the board to use
   */
  public GUI(Board board, Player player) {
    theBoard = board;
    this.player = player;
    setSize(new Dimension(550, 565));
    setTitle("Battleship");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());
    add(guiBoard, "Center");
    add(info, "South");
    //add(scoreHeader, "West");
    add(score, "East");
    setVisible(true);
  }

  /**
   * Will generate all visual components
   */
  public void drawBoard() {
    // First delete all existing graphics
    guiBoard.clear();
    guiBoard.setPaint(white);
    guiBoard.fillRect(0, 0, 2000, 2000);
    // Next loop through each element in Board.fieldStatus[][]
    for(int i = 0; i < theBoard.getXdim(); i++) {
      for(int j = 0; j < theBoard.getYdim(); j++) {
        // Set current field properties based on current field status
        switch(theBoard.getFieldStatus(i, j)) {
          case 0:
            fieldColor = blue;
            break;
          case 1:
            fieldColor = lightBlue;
            break;
          case 2:
            fieldColor = blue;
            break;
          case 3:
            fieldColor = red;
            break;
          default:
            info.setText("Suspicious field value encountered...");
        }
        // Fill and draw rectangles
        guiBoard.setPaint(fieldColor);
        guiBoard.fillRect(i * CONSTANT + gridOffsetX, j * CONSTANT +
gridOffsetY, CONSTANT, CONSTANT);
        guiBoard.setPaint(white);
        guiBoard.setStroke(gridStroke);
        guiBoard.drawRect(i * CONSTANT + gridOffsetX, j * CONSTANT +
gridOffsetY, CONSTANT, CONSTANT);
        // Draw player info
        score.setText("".valueOf(player.getScore()));
      }
    }
  }

  public void setEventQ() {
    eventQ.listenTo(guiBoard, "fields");
    while(true) {
      EventObject event = eventQ.waitEvent();
      if(eventQ.isMouseEvent(event)) {
        int x = eventQ.getMouseX(event);
        int y = eventQ.getMouseY(event);
        if(eventQ.isMouseClicked(event)) {
          // Translate clicked coordinates to field coordinates
          int realx = mouseXtoCol(x);
```

```
            int realy = mouseYtoRow(y);
            // Check if coordinates are valid (-1 if non-valid)
            // Also check if we are within clickable boundaries
            if( realx >= 0 && realy >= 0 && isClickable(x, y)) {
              shoot(realx, realy);
              drawBoard();
            }
          }
        }
      }
    }

    /**
     * Tries to place a shot at given coordinate.
     * Also handles any messages to relay to the user.
     * @param the x coordinate picked up by event listener
     * @param the y coordinate picked up by event listener
     */
    public void shoot(int x, int y){
      int result = 0;
      String shipName = "";
      int points = 0;
      boolean gameOver = false;
      Coordinate coords = new Coordinate(x, y);
      if(theBoard.canPlaceShot(coords)) {
        try {
        result = theBoard.placeShot(coords);
        shipName = theBoard.shipNameIfKill(coords);
        points = theBoard.shipPointsIfKill(coords);
        gameOver = theBoard.isGameOver();
        }
        catch(InvalidShotException ivs) {
          info.setText("Couldn't place shot: " + ivs);
        }
        if(result == 1) {
          info.setText("You didn't hit anything.");
        }
        if(result == 3 && shipName.length() == 0 && points == 0) {
          info.setText("You hit a ship... Try again.");
        }
        if(result == 3 && shipName.length() != 0 && points != 0) {
          info.setText("You sunk the enemys " + shipName + ". You get " + points +
  " points. Try again.");
          player.setScore(points);
        }
        if(gameOver) {
          info.setText("Game over...!");
        }
      }
      else {
        info.setText("You can't shoot here...");
      }
    }

    /**
     * Translates the coordinate from getMouseX() to field coordinate (array
     * index)
     * @param the getMouseX() coordinate
     * @return the field coordinate (array index) or -1 if translated value not
     * valid
     */
    public int mouseXtoCol(int realx) {
      int x = ((realx - gridOffsetX) * (100 / CONSTANT)) / 100;
      // x must be a valid field
      if(x >= 0 && x < theBoard.getXdim()) {
        return x;
      }
      else {
        return -1;
      }
    }

    /**
     * Translates the coordinate from getMouseY() to field coordinate (array
     * index)
     * @param the getMouseY() coordinate
     * @return the field coordinate (array index) or -1 if translated value not
     * valid
     */
    public int mouseYtoRow(int realy) {
      int y = ((realy - gridOffsetY) * (100 / CONSTANT)) / 100;
      // y must be a valid field
      if(y >= 0 && y < theBoard.getYdim()) {
        return y;
      }
      else {
        return -1;
      }
    }

    /**
     * Checks whether the clicked coordinate is "clickable", ie. not too close to
     * edge of field
     * @param the x coordinate from getMouseX()
     * @param the y coordinate from getMouseY()
     * @return true or false
     */
    public boolean isClickable(int x, int y) {
      x = x - gridOffsetX;
      y = y - gridOffsetY;

      int fieldXStart = ((x * (100 / CONSTANT)) / 100) * CONSTANT;
      int fieldXEnd = fieldXStart + CONSTANT;
      int xLow = fieldXStart + nonClick;
      int xHigh = fieldXEnd - nonClick;

      int fieldYStart = ((y * (100 / CONSTANT)) / 100) * CONSTANT;
      int fieldYEnd = fieldYStart + CONSTANT;
      int yLow = fieldYStart + nonClick;
      int yHigh = fieldYEnd - nonClick;

      /* Debugging
      System.out.println("xLow: " + xLow);
      System.out.println("xHigh: " + xHigh);
      System.out.println("X is: " + x);
      System.out.println("yStart: " + yLow);
      System.out.println("yEnd: " + yHigh);
      System.out.println("Y is: " + y);
      System.out.println();
      */

      if(x > xLow && x < xHigh && y > yLow && y < yHigh) {
        return true;
      }
      else
      {
        info.setText("This click was too close to border - try again");
        return false;
```

```
        }
    }
}

```

7. The Coordinate class – Coordinate.java

```java
package dk.ruc.loft.battleships;
/**
 * This class represents a set of coordinates
 */
public class Coordinate {

  // Coordinate fields
  private int x;
  private int y;

  // Hashcode multiplier - prime number
  private static final int HASH = 31;

  /**
   * Constructs a coordinate set
   * @param the x coordinate
   * @param the y coordinate
   */
  public Coordinate(int x, int y) {
    this.x = x;
    this.y = y;
  }

  /**
   * Get value of x coordinate
   * @return the x value of this coordinate
   */
  public int getX() {
    return this.x;
  }

  /**
   * Get value of y coordinate
   * @return the y value of this coordinate
   */
  public int getY() {
    return this.y;
  }

  /**
   * Test this coordinate for equality with other coordinate. Overrides
   * java.lang.equals
   * @param the coordinate to test
   * @return true or false
   */
  public boolean equals(Coordinate coord) {
    if(this.x == coord.getX() && this.y == coord.getY()) {
      return true;
    }
    else return false;
  }

  /**
   * Generates hashCode. Not used, but if equals() is specified, so must
   * hashCode()
   * @return the hashcode value for this object
   */
  public int hashCode() {
    return x * 31 + y;
  }
}
```

8. The Exception class – BattleshipException.java

```
package dk.ruc.loft.battleships;
/**
 * Custom exception class
 */
abstract class BattleShipException extends Exception {

  /**
   * Constructs new exception
   */
  public BattleShipException() {
  }
  public BattleShipException(String msg) {
    super(msg);
  }
}
```

9. The FieldOccupiedException class – FieldOccupiedException.java

```
package dk.ruc.loft.battleships;
/**
 * Exception class for handling situations when a ship is allready
 * on the board
 */
public class FieldOccupiedException extends BattleShipException {
  Coordinate problemCoord;

  /**
   * Constructs new exception
   * @param the Coordinate with the problem
   * @param the string with the error message
   */
  public FieldOccupiedException(Coordinate coord, String msg) {
    super(msg);
    problemCoord = coord;
  }

  /**
   * Method for getting the X coordinate
   * @return the X coordinate
   */
  public int getX() {
    return problemCoord.getX();
  }

  /**
   * Method for getting the Y coordinate
   * @return the Y coordinate
   */
  public int getY() {
    return problemCoord.getY();
  }
}
```

10. The InvalidShotException class – InvalidShotException.java

```
package dk.ruc.loft.battleships;
/**
 * Exception-class for handling problematic shots fired
 */
public class InvalidShotException extends BattleShipException {

  Coordinate problemCoord;
  /**
   * Constructs new exception
   * @param the coordinates that was hit
   * @param the string with the error message
   */
  public InvalidShotException(Coordinate coord, String msg) {
    super(msg);
    problemCoord = coord;
  }

  /**
   * Method that gets the X coordinate
   * @return the x coordinate
   */
  public int getX() {
    return problemCoord.getX();
  }

  /**
   * Method that gets the Y coordinate
   * @return the y coordinate
   */
  public int getY() {
    return problemCoord.getY();
  }
}
```