|  |
| --- |
|  |
| Battleships Project |
| Report |

|  |
| --- |
| dogmaan  7/27/2012 |

Table of Contents

[Executive Summary 2](#_Toc331150132)

[What the program does 2](#_Toc331150133)

[Successes and failures of the project 2](#_Toc331150134)

[Success and usefulness of the Battleships Application 2](#_Toc331150135)

[Introduction 2](#_Toc331150136)

[Requirements 3](#_Toc331150137)

[Functional requirements 3](#_Toc331150138)

[Non-functional Requirements 3](#_Toc331150139)

[Design 4](#_Toc331150140)

[Class design 4](#_Toc331150141)

[Source Code 6](#_Toc331150142)

[View Example 6](#_Toc331150143)

[Controller Examples 6](#_Toc331150144)

[Model Example 8](#_Toc331150145)

[Evaluation 9](#_Toc331150146)

[Summary and Conclusion 9](#_Toc331150147)

[Critical Self Evaluation 9](#_Toc331150148)

# Executive Summary

## What the program does

The program is a software implementation of the physical Battleships board game.

## Successes and failures of the project

The project met the vast majority of the requirements for the battleship application, the game is playable in both single player and two player.

The project took a long time to write as the design was very complex, also the actual UI design is not very accomplished or exciting.

## Success and usefulness of the Battleships Application

The application is highly usable/playable from start to finish, and also enables saving and loading in both single player and two player.

Overall the project was a success as it met all the major requirements and some additional ones too, such as a single player mode.

# Introduction

This report will cover the requirements, design, and creation of a software version of Battleships, it will also detail the methodologies, and design patterns used to create the application, and the motivations behind those decisions.

# Requirements

## Functional requirements

* Must support two players (Human or CPU).
* Playfield must be a grid.
* Each fleet must be made up of:
  + 1 Battleship that takes up 4 grid spaces
  + 2 Cruisers that take up 3 grid spaces each
  + 3 Destroyers that take up 2 grid spaces each
  + 3 Submarines which take up a single grid space each
* Each ship type may take up a different amount of spaces on the grid.
* Ships cannot be placed diagonally.
* Players must be able to place ships on the grid.
* "Play" must start when the two players have placed their ships onto the grid.
* Players must take turns to shoot at each other's ships by clicking on a grid position.
* There must be visual feedback to distinguish between a hit or a miss.
* The shot history must be displayed.
* A Player wins by hitting all positions of the opposing fleet.
* Must Have A Drop down or pop-up menu that allows the user to change the programs options.
* A Start new game option, when this option is selected the ships are randomly placed on the grid and the game is initialised.
* A Save game option which serialises the fleet and grid positions to a save file.
* A Load game option that loads the saved games text file into memory and de-serialises the relevant game data java classes.

## Non-functional Requirements

* The boxes of the grid that have ships assigned must contain the first character of the ship type, e.g. Cruiser would be CCC.
* Application must be written in Java.
* Must contain a Grid/UI rendered by the Swing UI framework.

# Design

As the requirements stipulated the use of Java and Swing UI, I researched some of the different design patterns that would best fit those constraints, coming from a .net background I would of preferred an MVVM (Model-View View Model) approach to the application, but as Java has no inbuilt equivalent to XAML and .net data binding, I decided to use the MVC (Model View Controller) approach detailed by Oracle here:

<http://www.oracle.com/technetwork/articles/javase/index-142890.html>.

The reasoning behind choosing the Model View Controller design pattern is that it helps reduce coupling between classes, encourages high cohesion, and easier testability, as it divides Logic/State, UI and communication responsibilities between several classes, the battleships MVC implementation looks like this:

Model - this holds all of the Battleships applications, state and logic.

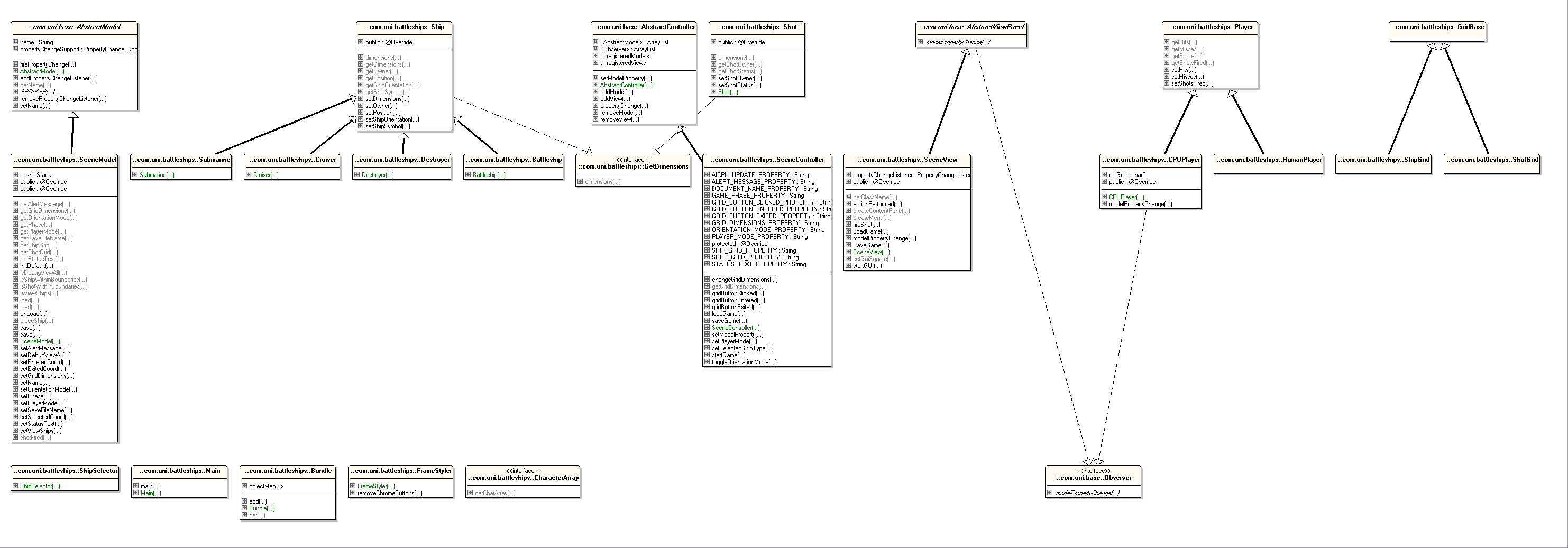
View - holds all UI components and drawing logic.

Controller - this is a container class that holds the view and model and facilitates communication between them.

Theoretically because of the use of this design pattern, and the fact that the view contains no "knowledge" of what it is that it is drawing, it would be very easy to port this application to an android device or other java implementation just by providing a new platform specific view class, and is a massive advantage to using this design pattern.

## Class design

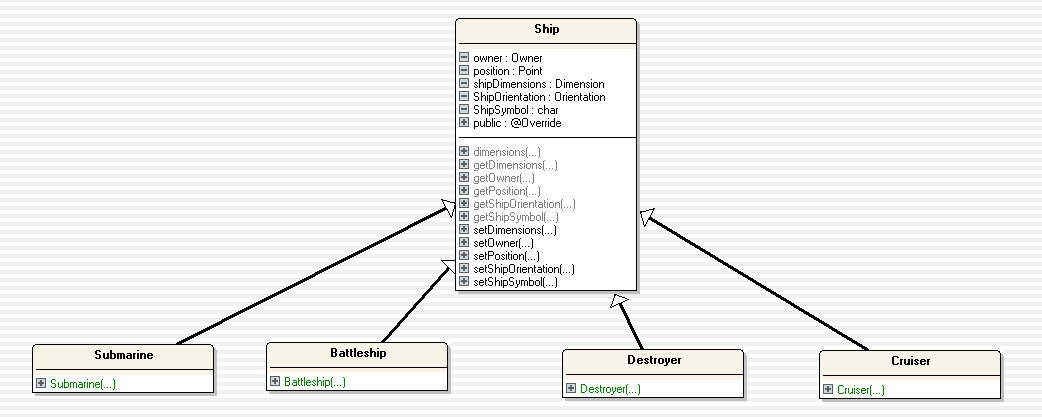
please refer to [Battleships Project Class Diagram.png] for a larger version of this image



Figure

To learn about a large portion of the Java Language and virtual machine, I made the conscious decision to artificially complicate the project, so as to cover as many areas of the java languages syntax and features as possible, hypothetically a Battleships application could be made with just one class, and one multidimensional array, but I would not have been able to demonstrate Generics, Inheritance, Polymorphism, Interfaces, Anonymous Classes, Abstract Classes, Enumerations and Serialization, without increasing the scope and complexity of the project.

For example each different ship extends a base ship implementation, that allows the application to store a stack or array of ships and each ships unique attributes in a generic <Ship> container



Figure

# Source Code

## View Example

The Scene View class contains the UI definitions, and logic on how to draw grid updates received from the Scene Model Class, the example below shows the creation of the array of buttons used to display the playfield grid, also each button has an anonymous class added to it via an action listener, this lambda style method contains a unique statement that when activated by a button press, passes the buttons coordinates to the Scene Controller class, and then onto the Scene Model class.

for (int y = 0; y < sceneController.getGridDimensions().height; ++y)

{

for (int x = 0; x < sceneController.getGridDimensions().width; ++x)

{

final int fx = x;

final int fy = y;

final int xy = x + (y \* sceneController.getGridDimensions().width);

Border bBorder;

bBorder = genButtonArrayBorders(fx);

buttonArray[xy] = new JButton("");

buttonArray[xy].setBorder(bBorder);

buttonArray[xy].setActionCommand(Integer.toString(xy));

buttonArray[xy].addActionListener(new ActionListener()

{

@Override

public void actionPerformed(ActionEvent e)

{

System.out.println("SceneView: Button x:" + fx + " y:" + fy + " pressed");

sceneController.gridButtonClicked(fx, fy);

}

});

{

## Controller Examples

Following on from the above example a message is passed to all registered views (or observer classes) via the following method

public void gridButtonClicked(int x, int y)

{

setModelProperty(GRID\_BUTTON\_CLICKED\_PROPERTY, new Point(x, y) ) ;

}

All of the possible changeable properties are stored in the Scene Controller as constants

public static final String DOCUMENT\_NAME\_PROPERTY = "Name";

public static final String GRID\_DIMENSIONS\_PROPERTY = "GridDimensions";

public static final String SHIP\_GRID\_PROPERTY = "ShipGrid";

public static final String SHOT\_GRID\_PROPERTY = "ShotGrid";

public static final String GRID\_BUTTON\_CLICKED\_PROPERTY = "SelectedCoord";

public static final String GRID\_BUTTON\_ENTERED\_PROPERTY = "EnteredCoord";

public static final String GRID\_BUTTON\_EXITED\_PROPERTY = "ExitedCoord";

public static final String PLAYER\_MODE\_PROPERTY = "PlayerMode";

public static final String GAME\_PHASE\_PROPERTY = "Phase";

public static final String STATUS\_TEXT\_PROPERTY = "StatusText";

public static final String ORIENTATION\_MODE\_PROPERTY = "OrientationMode";

public static final String ALERT\_MESSAGE\_PROPERTY = "AlertMessage";

public static final String AICPU\_UPDATE\_PROPERTY = "AICPUUpdateProperty";

To Load a saved game firstly the current Scene Model class is deleted, then a de-serialised save file is added to the list of registered models via a static load method, this returns the "re-inflated" class, next the on Load method is called, this forces the scene model to fire off property changes, and update the UI, finally the saved player mode is set and if the save file was single player a CPU player is created

public void loadGame(String name)

{

removeModel(((SceneModel)registeredModels.get(0)));

addModel(SceneModel.load());

((SceneModel)registeredModels.get(0)).onLoad();

setPlayerMode(((SceneModel)registeredModels.get(0)).getPlayerMode());

}

## Model Example

The Scene Model class contains the majority of the applications logic and properties, I decided to design the class as a kind of state machine, for example the following method's functionality changes depending on which of the 6 phases of the game the class state is in.

public void setSelectedCoord(Point point)

{

switch (getPhase())

{

case INITIAL:

break;

case SELECTION\_PLAYER1:

if(!shipStack.empty())

{

if(isWithinPlayerBoundaries(point))

{

updateSelectionPhaseViewData();

firePropertyChange(SceneController.AICPU\_UPDATE\_PROPERTY, null,

removeOppositePlayerShipsFromArray(shipGrid.getCharArray()));

break;

}

}

addShipToPosition(point);

firePropertyChange(SceneController.AICPU\_UPDATE\_PROPERTY, null,

removeOppositePlayerShipsFromArray(shipGrid.getCharArray()));

break;

case SELECTION\_PLAYER2:

if(!shipStack.empty())

{

if(isWithinPlayerBoundaries(point))

{

updateSelectionPhaseViewData();

firePropertyChange(SceneController.AICPU\_UPDATE\_PROPERTY, null,

removeOppositePlayerShipsFromArray(shipGrid.getCharArray()));

break;

}

}

addShipToPosition(point);

firePropertyChange(SceneController.AICPU\_UPDATE\_PROPERTY, null,

removeOppositePlayerShipsFromArray(shipGrid.getCharArray()));

break;

case SHOTTURN\_PLAYER1:

attemptToFireShot(point);

firePropertyChange(SceneController.AICPU\_UPDATE\_PROPERTY, null, generateCharArray());

break;

case SHOTTURN\_PLAYER2:

attemptToFireShot(point);

firePropertyChange(SceneController.AICPU\_UPDATE\_PROPERTY, null, generateCharArray());

break;

case GAMEEND:

break;

default:

break;

}

}

# Evaluation

In order to evaluate added features and to make sure I met the requirements, I kept a to do list that I ticked off whenever a new functionality was implemented and tested.

Unfortunately I did not give myself enough time to write unit tests, so the majority of the testing was repeated thread testing, and debugging via various IDE's, I proved the application worked by playing multiple games against friends, and also against the CPU, any bugs that appeared during these tests where fixed, and most UI suggestions were implemented, for example there is a status bar that keeps the user informed of the state of the game, this was asked for in testing.

Known Bugs:

The CPU player will fire its first shot before the user has been informed that it's the CPU Players turn.

BlueJ does not support some of the Exception types I used so I had to strip them, namely the InvalidStateException.

The Status Label does not have a left margin.

# Summary and Conclusion

In summary I used this project as an exercise to really learn the details of Java syntax and rules, so my personal goals were definitely fulfilled

The application itself is highly playable and functional, as a simple software recreation of battleships as the application meets its requirements and is stable I believe it has been a successful project.

# Critical Self Evaluation

The downside of over complication and generalisation in software development is that the amount of time to implement functionality increases, unfortunately this was the biggest problem I had, I found it very difficult to complete the project in the way I wanted with the time allotted, if I had simplified the design I could have concentrated on adding features rather than learning about the abstract corners and rules of the Java language I needed to use for my original design.

Having a background in C# and C++, I don't feel I have learned a lot about Java per se, the most useful things I have learned are; the basics of swing UI development, how to implement a Model View Controller design pattern in Java, how to deal with the lack of lambdas or function pointers via anonymous classes, how to handle generics in Java, and to appreciate interfaces more.

If I was to give myself a mark out of a hundred it would be 85, as I did not add clever AI functionality, I also did not add the ability to change the grid dimensions, and I did not concentrate enough on the applications presentation, it could look a lot nicer and communicate to the player better.