Principles of Software Engineering
2805ICT
3815ICT
7805ICT
Milestone Two
(MODEL SOLUTION)
Weight 10%

Vladimir Estivill-Castro

September 13, 2020

1 Objectives

1. Practice reporting the progress on a software development project.
2. Describe a testing plan and testing performed within the current progress.
3. Use the UML as a tool to describe and specify software designs, and be able to attach semantic meaning to UML constructs by establishing relationships with the code the artifact models.
4. Discuss software design patterns meaningfully.
5. Review the potential evolutions that a software project may have regarding software architecture and software design paradigms.
6. Model dynamic behaviour of software systems using suitable UML notation.
7. Describe the progress concerning user interface design.
8. Review metrics regarding progress by reviewing historical date on version control system.
9. Discuss the properties of good software design including the nature and the role of associated documentation.
10. Create associated descriptions to document UML diagrams.

2 Progress Report

2.1 Requirements

Please refer to Milestone 1 model solutions for a list of functional requirements. The focus of the development has been the implementation of the **client/server** mode. In what follows, the priorities have been revised to reflect the new priorities and the urgency to complete the functional requirement by using Red. An old priority is shown in parenthesis.

The table below corresponds to the partial list of requirements shown in the model solution for Milestone 1 (the model solution for Milestone 1 is not a complete lust of requirements, it just a list illustrative of functional requirements). However, since the focus is the expansion to a client/server version of the code, many requirements about

he behaviour of the server and the behaviour of the client were not detailed in the model solution for Milestone 1. Thus, although these requirements about client/server behaviour have been completed, they are not reflected by the table below.

Summary of Level of Completion			
Identifier	**Priority**	**Completion**	**Comments**
F-R 1	5	(P) Completed	The (GUI) currently works only for the square grid case. Pending the hexagonal GUI and the graph GUI.
F-R 2	5	Completed	The system offers the menu option and the call-back in enabled; resulting in 3 difficulty levels. Then density of the levels is roughly the same.
F-R 3	5	Incomplete	The system has menu-item options for the player to select different environment. But only the square-grid is functional. The hexagonal grid, and a graph layout are missing.
F-R 4	5	Incomplete	The count down of the timer works only in stand-alone mode. MyRedIn client/server mode the count down of the timer is not yet implemented. The game dose not end if the timer expires.
F-R 5	5	Completed	Working in stand-alone and client server mode.
F-R 6	5	Complete	The start button work to start the challenge.
F-R 7	5	Complete	The game finishes when all mines are located or when the user steps (uncovers) a mine.
F-R 8	5	Complete	The system displays the initial level of mines. Every time the user flags a mine correctly, the display decreases the number of mines to be identified. The mine display does not decrease for a wrongly flagged location for a mine.
F-R 9	5	(P) Complete	The system offers an opportunity to reset the game and select a new challenge after the previous game is over. In client/server mode, the server accepts only one connection, this needs to be extended.
F-R 10	5	Complete	The system provides visual feedback as to whether the game was completed by all mines being identified or because of stepping on a mine.
F-R 11	3 (4)	Incomplete	The score of a player for a level needs to account for minimising time and minimising wrong placement of flags.
F-R 12	5	Completed	Star-button and flagging or uncovering works.
F-R 13	5 (1)	Incomplete	The server should accept several connections against the same mode so the game offers a multi-player option where two players receive the same environment and they race against each other to locate all mines.
F-R 14	3	Completed	The system uses a as random seed the milliseconds in the system's date.
F-R 15	3	Incomplete	The size of the window for the game is constant.
F-R 16	2	Incomplete	There are no options to configure elements of the environment, such as the colours of numbers, or icons in the game.
F-R 17	2	Incomplete	There are no options to configure musical and sound feedback.
F-R 18	4	(P) Complete	The zero flooding algorithm works in the square version but is written generically. The flooding algorithm should work as is for the hexadecimal environment.
F-R 19	3	Incomplete	No database facility has been implemented to record users and their scores. A local database should be implemented on the Model side.
F-R 20	5	(P) Completed	Works for the square-grid environment. Incomplete for the hexadecimal-grid environment.
F-R 21	5	Complete	The exit in the GUI ends the game. Even in client/server mode, this closes the client connection. A new connection is not received by the server if the previous is closed.
F-R 22	2(1)	Incomplete	There are no options to configure all parameters of the game, such as the size (and layout) of the environment and the density of mines.
F-R 23	1	Incomplete	There is no *undo* option yet.

2.2 Product Use Cases

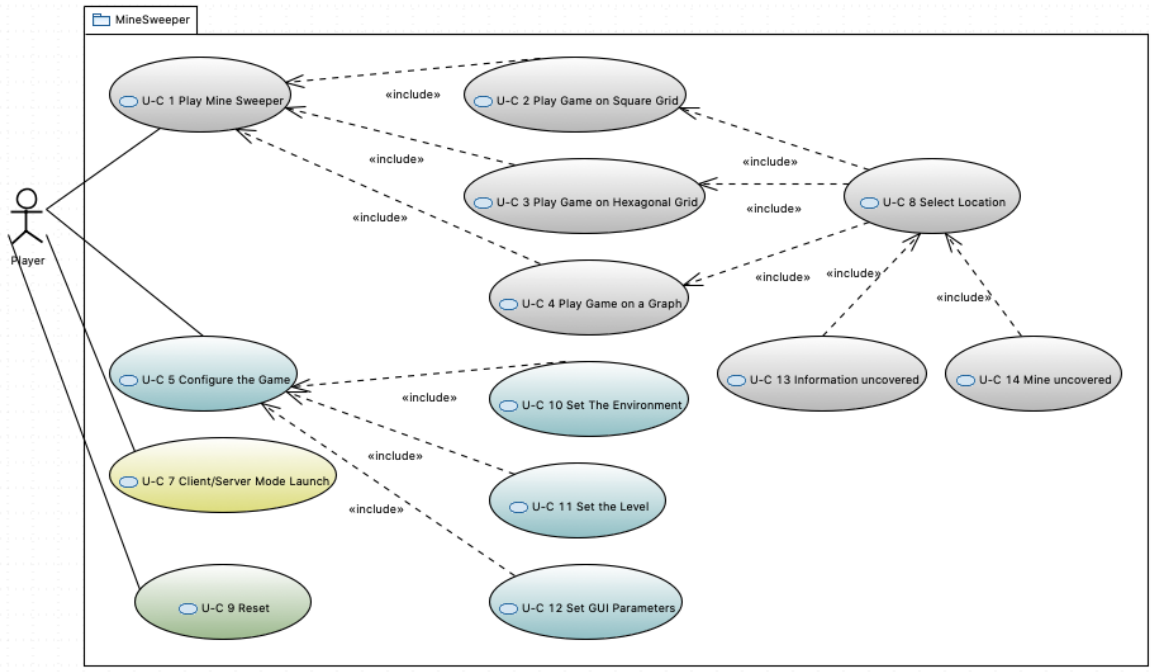

Figure 1: Reviewed use-case diagram.

Table 1 lists the uses cases. As a result of the development of the prototype and its extension to its current version, this list of use cases has been reviews. The reviewed use case diagram appears in Figure 1.

Use-Case Requirements Matrix				
Identifier	**Use Case**	Status	**Short Description**	**Requirements involved**
U-C 1	Play Mine Sweeper	Incomplete	<<includes>> many sub-use-cases. Several use-cases not completed.	F-R 1, F-R 2, F-R 4, F-R 5, F-R 6, F-R 7, F-R 8, F-R 9, F-R 10, F-R 12, F-R 13, F-R 18, F-R 20, F-R 21, F-R 23
U-C 2	Play Game on Square Grid included in U-C 1	Completed	A single player can complete games in a squared grid both in stand alone and client server mode. Some details on some functional requirements missing (multi-player options). The undo facility is not implemented	F-R 1, F-R 2, F-R 4, F-R 5, F-R 6, F-R 7, F-R 8, F-R 9, F-R 10, F-R 12, F-R 13, F-R 18, F-R 20, F-R 21, F-R 23
U-C 3	Play Game on Hexagonal Grid included in U-C 1	Incomplete	Currently there is no drawing capability for an hexagonal environment. Flooding in hexagonal environment may be delicate (6 neighbours to a hegaxon).	F-R 1, F-R 2, F-R 3, F-R 4, F-R 5, F-R 6, F-R 7, F-R 8, F-R 9, F-R 10, F-R 12, F-R 13, F-R 18, F-R 20, F-R 21, F-R 23
U-C 4	Play Game on a Graph included in U-C 1	Incomplete	Currently there is no drawing capability for a topological environment.	F-R 1, F-R 2, F-R 3, F-R 4, F-R 5, F-R 6, F-R 7, F-R 8, F-R 9, F-R 10, F-R 12, F-R 13, F-R 20, F-R 21, F-R 23
U-C 5	Configure the game	Incomplete	Currently there are no options to configure the environment (sound/music, application size, icons, environment, mine density). <<includes>> many sub-use-cases.	F-R 2 F-R 3, F-R 16, F-R 17, F-R 22
U-C 6	Save a player score	Incomplete	Currently there is no facility to save a player score (and thus no facility to retrieve it).	F-R 11, F-R 19
U-C 7	Client/Server Mode Launch	(P) Completed	Currently the game can be launched stand alone, as a server or as a client. Only one connection accepted and only the first time.	F-R 13

U-C 8	Select Location included in U-C 2, U-C 3 U-C 4	(P) Completed	Clicking options work and tested in MacOS. Still to be tested for Linux and Windows. Consider add requirement for game to be operational with keyboard only (arrow keys, short-cuts and no mouse)	F-R 12, F-R 20
U-C 9	Reset	(P) Completed	Currently rest game status to initial default status. Possibly refine functional requirements to reset specific setting of the game to defaults.	F-R 9
U-C 10	Set the Environment included in U-C 5,	Incomplete	Currently there option to configure between square, hexagonal and graph environment exists but has no effect.	F-R 3,
U-C 11	Set the Level included in U-C 5,	(P) Completed	The option for 3 difficulty levels works for the square grid.	F-R 2, F-R 6, F-R 8 F-R 11, F-R 19
U-C 12	Set the GUI parameters included in U-C 5,	Incomplete	Nothing about the GUI can be configured by the user.	F-R 15, F-R 17, F-R 22
U-C 13	Information uncovered included in U-C 8,	(P) Completed	A click on the GUI is detected, distinguished between left and shift-click and routed properly in stand alone or client-server mode. Status of the model is reflected in GUI in stand-alone and client-serve mode.	F-R 12, F-R 18,
U-C 14	Mine uncovered included in U-C 8,	(P) Completed	A click on the GUI that causes the uncovering a mine terminates the game. In client-server mode the client connection is terminated. No sound when game is over.	F-R 7, F-R 12, F-R 12

Table 1: List of use cases.

2.3 Summary of software architecture

The software architecture is based on two fundamental architectural software patterns.

1. *Model-View-Controller*: Here, we split the set of classes of the software into three packages. One completely responsible for all Graphical User Interface (GUI) items, which is the *View*. That is, the *View* displays the game to the user, the buttons, icons and numerical displays. It also captures all commands (mostly user clicks). One package for all aspects representing the status of the game, that is the *Model*. This package is responsible for recording the level, size of grid, and status of cells (whether they hold a a mine, whether they have been uncovered). The *Controller* launches the *View* and the *Model* and sets up the connection between the *Model* and the *View*. The GUI is programmed using an event-driven approach using `Java` Swing. The call-backs in the view are forwarded to the *Controller*. In some cases, responses from a status change in the *Model* result in an update invocation to *View* elements.

2. *Client-Server*: The program can run as a monolithic software in stand-alone mode, or as separate programs, a *Server* and a *Client* (the intention is to host multiple clients). This architectural pattern would easily enable the set-up of competitions in multi-player mode, where several players solve simultaneously a copy of the same mine sweeper puzzle. We are using plain sockets (ad sending character streams) in `Java` although more sophisticated libraries exit to communicate with object classes across a TCP/IP link.

The main tool for the development is the IDE NetBeans 8.2. We are also using `git` as a version control. One aspect that requires improvement is that no testing framework has been used. The following table summaries the main points regarding the infrastructure.

Development tool	**Most Commonly Used**
Main programming language(s) used	`Java`
IDE used	NetBeans 8.2
Version Control	`git`
Modelling tools	Papyrus, UMLet
GUI framework	`Java` Swing
Networking	TCP/IP and plain sockets
Persistence	`derby` embedded database

2.4 Summary of design

2.4.1 Static issues

The goal of the design is to separate the responsibility and functionality for drawing and updating the view/GUI in a package (module) for a *View*. We want to separate the visual presentation of the Mine Sweeper game from the data structures that keel the state of the game the (*Model*). This separation should not only result in more cohesive classes and minimise coupling between classes, but should enable to place the model in a server and replicate views in different clients. We also should be able to retain this modules for a stand alone version of the design.

The *Model-View-Controller* architecture is reflected in that the software consists of three `Java` packages.

1. `package minesweeper_view`

2. `package minesweeper_model`

3. `package minesweeper_controller`

The lifting to a class diagram using the `easyUML Create Class Diagram` from the current version of the code results in the diagram in Figure 2. Unfortunately, the `easyUML Create Class Diagram` does not include the package information; thus, we redraw the class diagram using Papyrus™.

The classes of the `package minesweeper_view` (preserving as much as possible the placement from Figure 2) appear in Figure 3. Several classes in the view derive from `Java` swing elements. For instance:

1. ExitListener extends WindowAdapter

2. IPCollectView extends javax.swing.JPanel

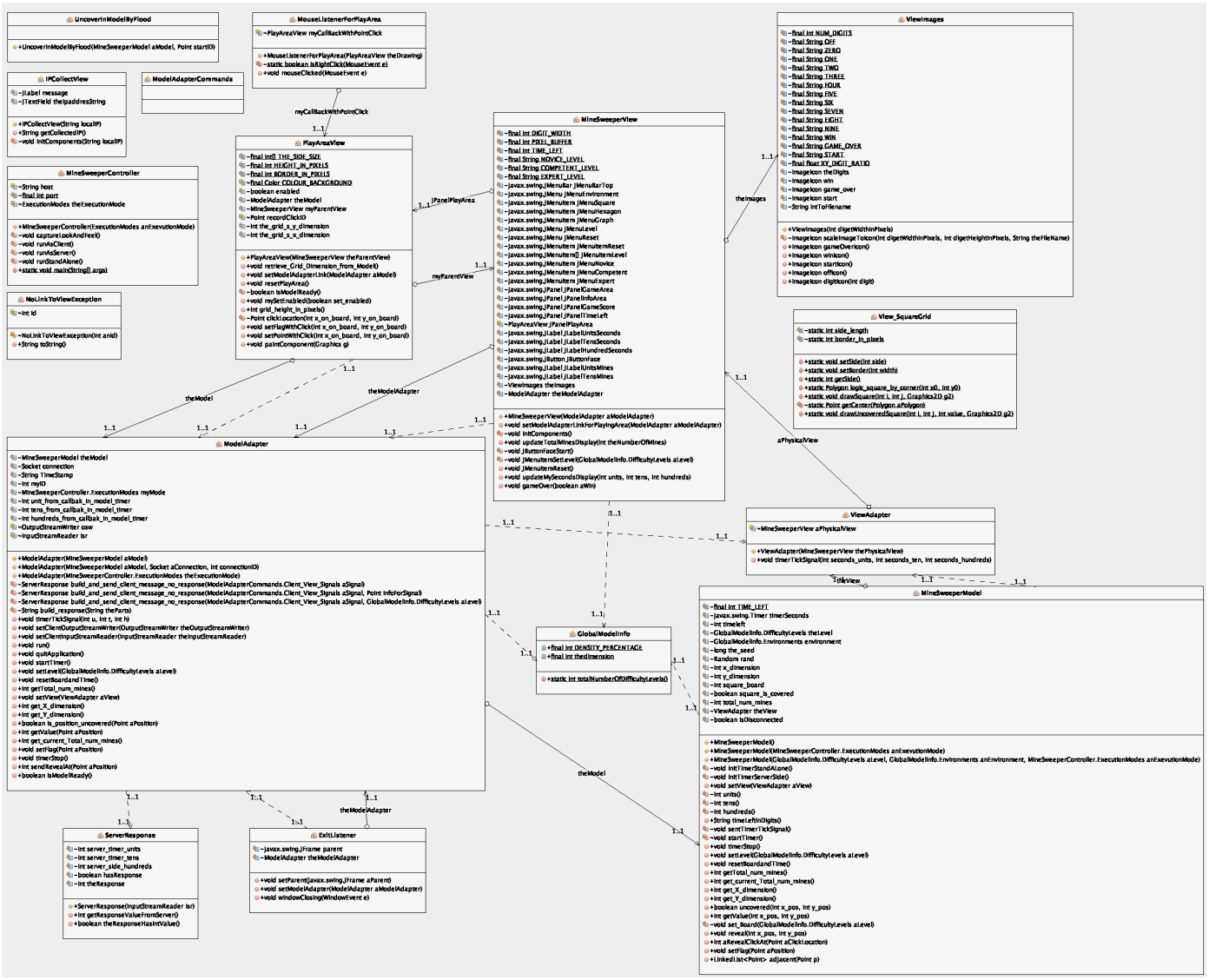

Figure 2: Class diagram from the current version of the code lifted using `easyUML Create Class Diagram` in Netbeans 8.2.

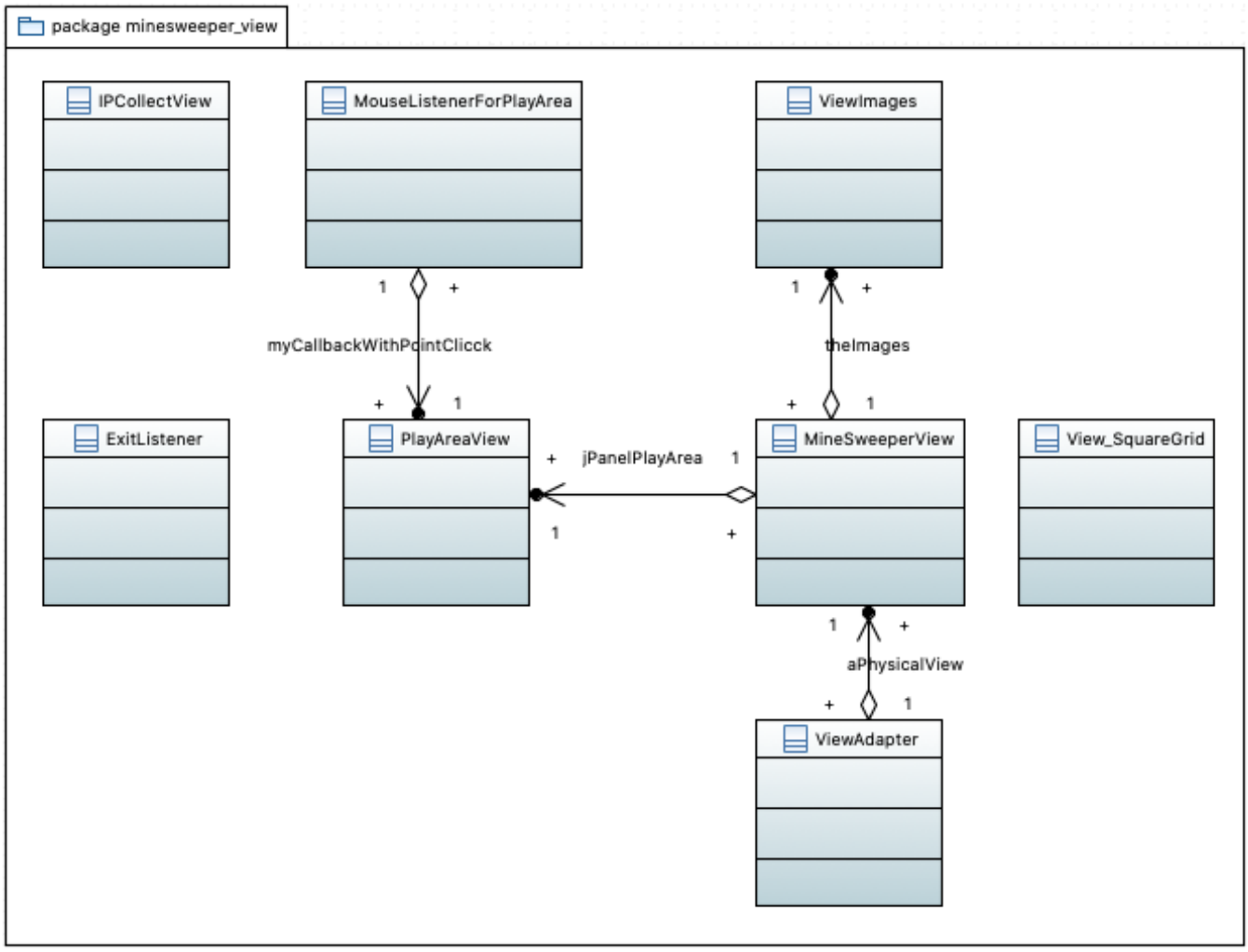

Figure 3: The classes of the `package minesweeper_view` preserving as much as possible the placement from Figure 2.

3. MineSweeperView extends javax.swing.JFrame

4. MouseListenerForPlayArea extends MouseAdapter

5. PlayAreaView extends JPanel

This is because these classes in the *View* result in displayable widgets in the GUI. More details of the *View* classes can be seen in the class diagram of Figure 4.

The classes of the package `package minesweeper_model` (preserving as much as possible the placement from Figure 2) appear in Figure 5. Some of this classes implement other `Java` classes. For example

1. ModelAdapter implements Runnable. This is because the model adapter in the server side (when in server-side mode) is expected to be a thread that handles a connection.

Other classes extend some other `Java` classes.

1. NoLinkToViewException extends Exception. The original idea was to have our own type of exception for errors in the client/server set-up, messaging and tear-down. However, the `Java` classes for reading out of the buffered streams of sockets would raise an I/O exception. So at the moment, several parts of the code raise the I/O exception rather than our own self designed exception.

To have an idea of the size in terms of properties of the classes in the package for the *Model* we draw the package with details for the classes in Figure 6.

The `package minesweeper_controller` has only one class, the class `MineSweeperController`. Figure 7 illustrates that there is only one class in the package.

A quick overview of the static modelling reveals (particularly Figure **??**) that there are many dependencies between classes across the 3 packages. The design has not yet fully separated the packages. The design attempts to

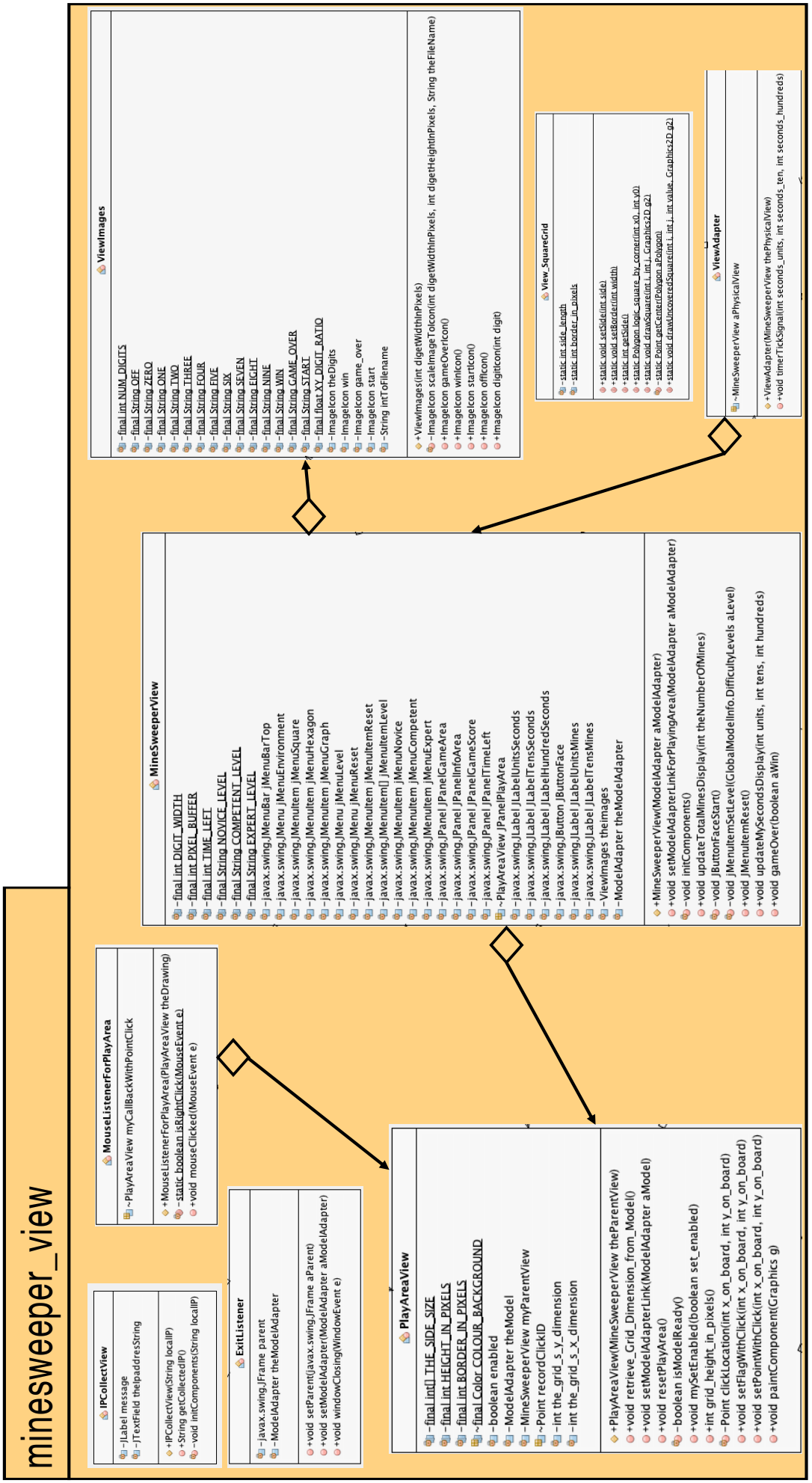

Figure 4: The detail of classes of the `package minesweeper_view` preserving as much as possible the placement from Figure 2 and the information about each class.

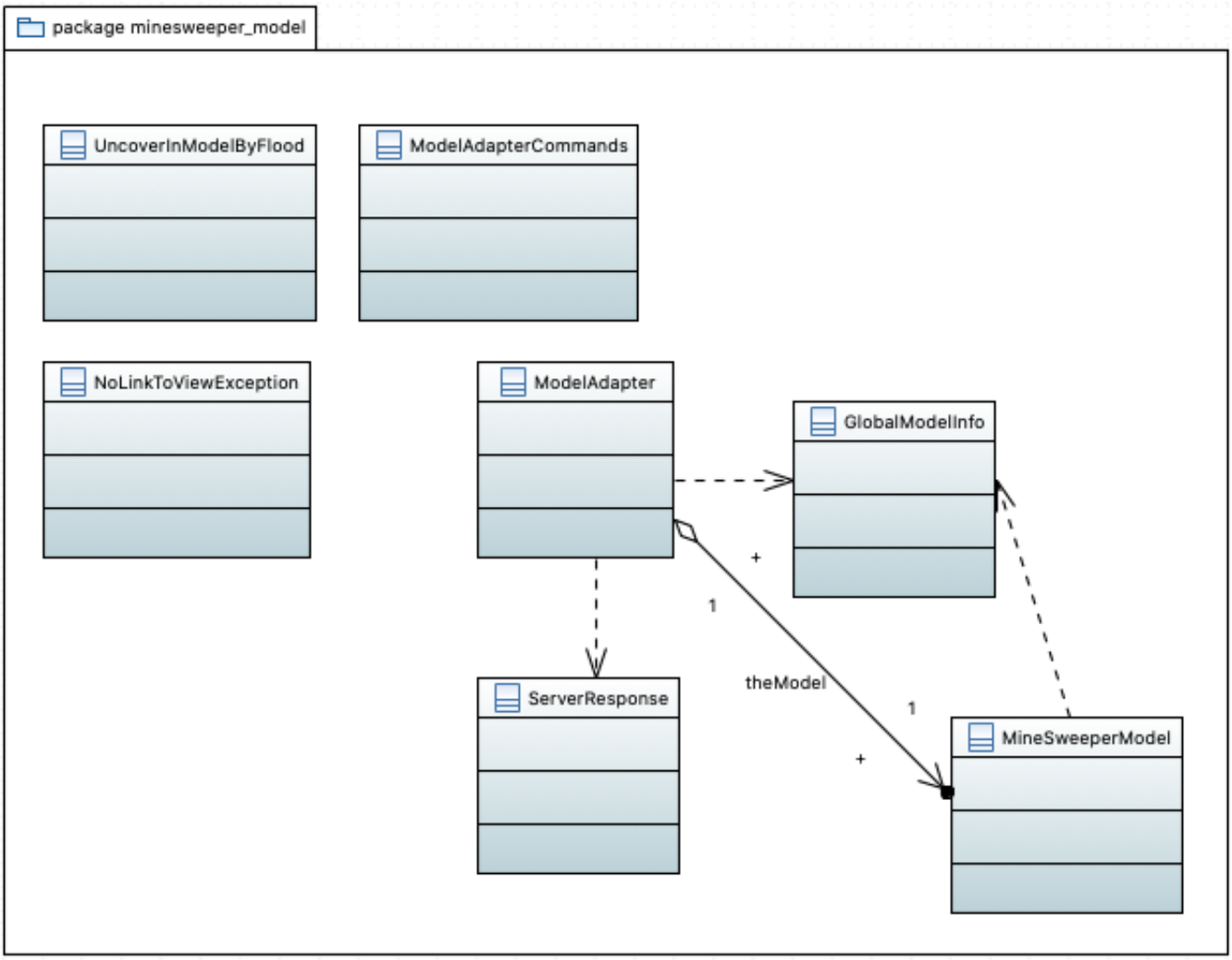

Figure 5: The classes of the `package minesweeper_model` preserving as much as possible the placement from Figure 2.

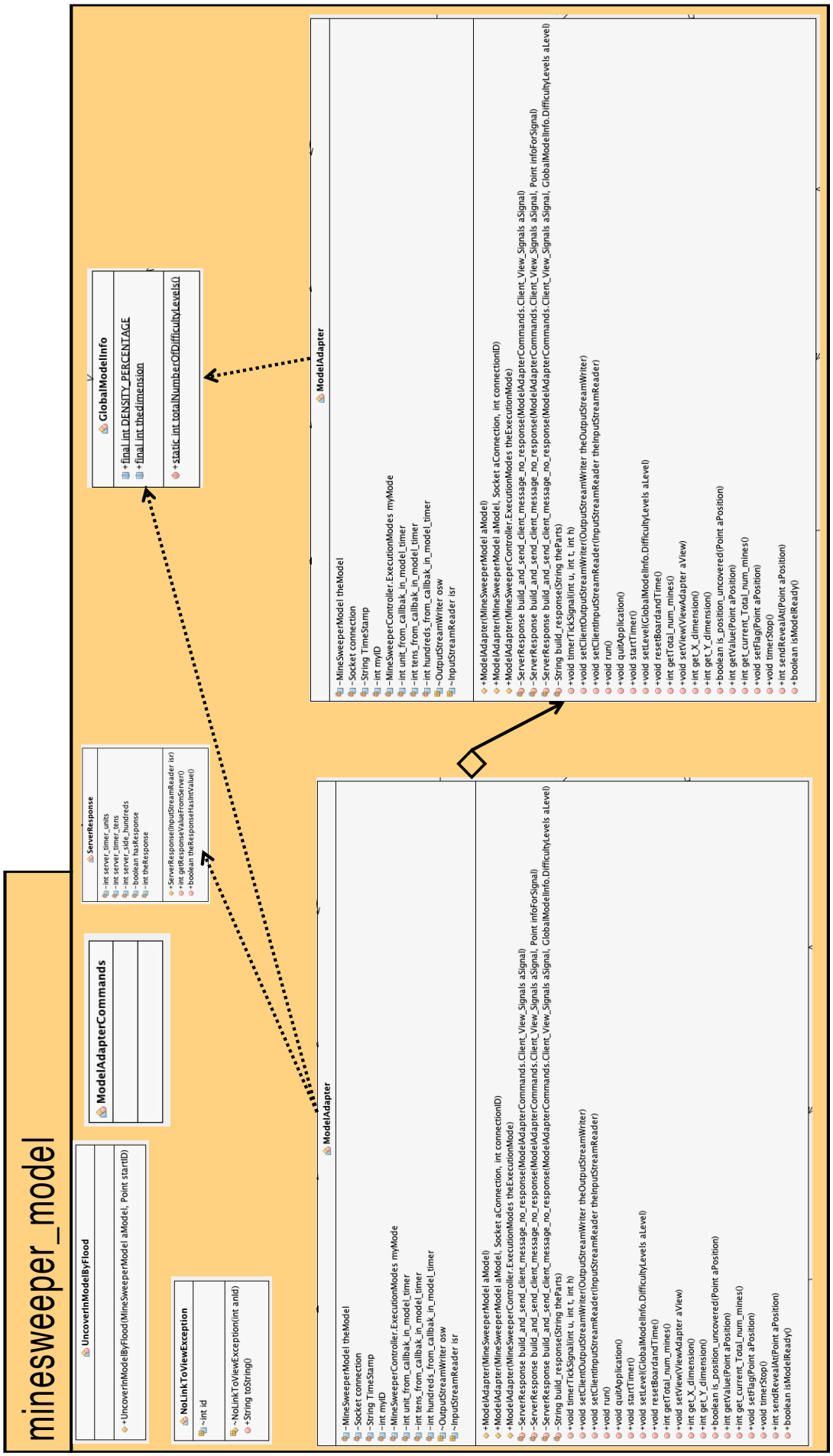

Figure 6: Details of the classes of the `package minesweeper_model` preserving as much as possible the placement from Figure 2.

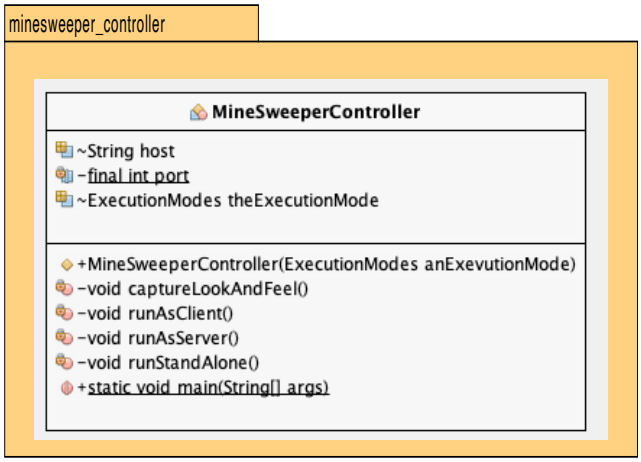

Figure 7: The `package minesweeper_controller` has only one class, the class `MineSweeperController`.

```
jMenuLevel.setText("Level");

jMenuItemLevel = new javax.swing.JMenuItem[GlobalModelInfo.totalNumberOfDifficultyLevels()];
for (GlobalModelInfo.DifficultyLevels aLevelIndex: GlobalModelInfo.DifficultyLevels.values()) {
    jMenuItemLevel[aLevelIndex.value]=jMenuNovice = new javax.swing.JMenuItem();
    jMenuItemLevel[aLevelIndex.value].setText(aLevelIndex.toString());
    jMenuItemLevel[aLevelIndex.value].addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jMenuItemSetLevel(aLevelIndex);
            }
        });
    jMenuLevel.add(jMenuItemLevel[aLevelIndex.value]);
} // For difficulty levels
```

Figure 8: Setting up the call-back method to the player's selection of a difficulty level.

use the software engineering pattern `Adapter` to shield the *View* and make it agnostic as to whether the *Model* is local or remote. Similarly, we have made and *ModelAdapter* attempting to isolate the *Model* as to whether it is part of a stand alone implementation or sitting on the server. However, there are several issues in which this isolation has not been completed yet.

2.4.2 Dynamic issues

One particular issue is the counting down of the timer. The ideal suggestion is that the timer is part of the *Model*, in particular, in the multi-player setting and client/server mode, a centralised timer seems to be the most accurate reference to judge the performance of a player. However, the display for the timer requires the GUI to receive the value of the timer in the model without an explicit command by a user.

Most of the interactions by a player on the GUI result in the following dynamic behaviour.

1. The user clicks on a widget on the GUI, typically a button or the panel selecting to uncover a cell.

2. The framework from `Java` swing listeners catches the event and launches a call-back. For instance, Figure 8 shows the code where the class `MineSweeperView` (which represents the *View* and extends from the `Java`-swing widget class `javax.swing.JFrame`) defines a menu of the available difficulty levels and registers the method `jMenuItemSetLevel` as the call-back for players selection of a level of difficulty.

```java
/**
 * Event handler
 * @param aLevel the user choice of level
 */
private void jMenuItemSetLevel( GlobalModelInfo.DifficultyLevels aLevel ) {
    try {
        theModelAdapter.setLevel(aLevel);
    } catch (IOException ex) {
        Logger.getLogger(MineSweeperView.class.getName()).log(Level.SEVERE, null, ex);
        JOptionPane.showMessageDialog(this, "Lost conenction to Server", "Connection  error",JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }
    JPanelPlayArea.repaint();
}
```

Figure 9: Call-back that implements the forwarding to the model that the user is re-setting the level of difficulty.

```java
/**
 * Forward message to appropriate model to set the level of the game
 * @param aLevel
 */
public void setLevel(GlobalModelInfo.DifficultyLevels  aLevel) throws IOException {
    if(  (MineSweeperController.ExecutionModes.STANDALONE == myMode)
    || ( MineSweeperController.ExecutionModes.SERVER == myMode)
          )        {
          theModel.setLevel(aLevel);
    }
    else
          if( MineSweeperController.ExecutionModes.CLIENT == myMode) {
              ServerResponse theResponse =  build_and_send_client_message_no_response(ModelAdapterCommands.Client_View_Signals.SET_LEVEL,aLevel);
          }
}
```

Figure 10: The facade for the `setLevel` method of the *Model* provided by the `ModelAdapter`.

3. The action that the player performed usually implies something has to change in the model, a cell is uncovered, a different setting needs to be recorded, or the puzzle must be reset, or the game starts and the timer shall kick down. In our running example, the level of difficulty is changed. The corresponding call-back is the method `jMenuItemSetLevel` also in the class `MineSweeperView` and Figure 9 present the corresponding code. We see that this code (Figure 9) results in the corresponding call to the `Model Adapter`. The idea is that the `jMenuItemSetLevel` method is part of the `MineSweeperView` (which represent the *View*) is agnostic as to whether we are in stand-alone mode or in client-server mode. We note that is not completely the case, The client/serve connection may fail and an exception may be raised (only in client/server mode, but not stand alone). However, `Java` statically request the handling on an exception. In this case, the client (the *View* terminates if the connection to the server is lost.

4. In normal circumstances, the `ModelAdapter` forwards the action to the *Model* (represented by the class `MineSweeperModel`. In our running example, the method `setLevel` of the class `ModelAdapter` performs the forwarding. The forwarding is illustrated in Figure 10. The `ModelAdapter` tests whether is a `ModelAdapter` in stand-alone mode, or in the server-side. In that case, it just forwards the call to a local instance of the *Model*. However, if it is in client mode, it needs to forward the action using the socket connection.

5. Thus, in client-mode the `ModelAdapter` needs to build a command to send trough the socket and receives back an object of the class `ServerResponse`. Figure 11 shows the code implementing

   `build_and_send_client_message_no_response`.

   Although as a client-side `ModelAdapter` in the example of setting the level of difficulty there is no expected response, we always collect a response from the server as acknowledgement of the sent action. We highlight that although we have created some classes for the protocol of the actions across the socket connection, this could be redesigned and improved using the software engineering pattern `Command` [**?**]

6. Building a message action requires to use the class `ModelAdapterCommands` which aims to eventually become the software engineering pattern `Command` [**?**]. Other classes are involved, the object `osw` is the `OutputStreamWriter` to actually write a string into the socket stream. Respectively, the object `isr` of the class `InputStreamReader` allows us to read the stream back from the server.

7. On the server side, there is a corresponding `Runnable ModelAdapter` started by the `ModelController` who accepted the connection. The `Runnable ModelAdapter` is executing a loop in its `run` method, a fragment of which is shown in Figure 12. Here, the action message from the client side is parsed, tested if it is the message stop and if not, the method `build_response` is invoked.

```
/**
 * Signal the server we want to set a new level for the game
 * @param aSignal code for the command
 * @param aLevel the desired new level the model shall be
 * @return
 * @throws IOException
 */
private ServerResponse build_and_send_client_message_no_response
        (ModelAdapterCommands.Client_View_Signals aSignal, GlobalModelInfo.DifficultyLevels  aLevel) throws IOException {
            int messageAsIntegerCode = aSignal.value;
            String process  = ""+messageAsIntegerCode+":"+aLevel.value+":";
            process += (char) 13;
            /** Write across the socket connection and flush the buffer */
            //Sending a new message with parameters with code
            osw.write(process);
            osw.flush();
            ServerResponse theResponse = new ServerResponse(isr);
            return theResponse;
}
```

Figure 11: The building of a message action (a command) for the socket and the retrieval of the response.

```
catch (Exception e){}

// Check if Client asks to stop
String clientMessage = process.toString();
String theParts[] = clientMessage.split(":");
int theCode = Integer.parseInt(theParts[0]);
ModelAdapterCommands.Client_View_Signals signalByClient = ModelAdapterCommands.Client_View_Signals.values()[theCode];
stop_signal_from_client = (ModelAdapterCommands.Client_View_Signals.STOP_MODEL== signalByClient);

if (!stop_signal_from_client) {
        // We build a response accoringly
  String returnCode = build_response(theParts) + (char) 13;
  BufferedOutputStream os = new BufferedOutputStream(connection.getOutputStream());
  OutputStreamWriter outgoing_OSW = new OutputStreamWriter(os, "US-ASCII");
  outgoing_OSW.write(returnCode);
  outgoing_OSW.flush();
}

} while (! stop_signal_from_client);
```

Figure 12: Code that shows the behaviour of the server-side `ModelAdapter`.

8. The server-side `ModelAdapter`, in its method `build_response` puts a prefix with the time in the *Model* (the server side model) and also switches to the facade method for the action. Figure 13 shows the fraction of code in the method `build_response` that invokes the action of setting the level locally. This is a call to the adapter facade of Figure 10 but now in server-side. Thus, now the methods is invoked on the model.

9. Then, the *Model* receives the request and updates its value for the level of difficulty.

Figure 14 illustrates diagrammatically the execution described above with a sequence diagram. Those objects in orange are on the server side and their lifelines are in yellow. Those objects in the server side are in red.

However, the first issue we must address relative to the stand-alone version of Milestone 1 is the set-up in the client/server version. The *View* can not draw the GUI until a connection to the server has been established, because the data on the size of the environment (level of difficulty) now resides on the *Model* on the server side.

To assist our discussion we present the communication diagram in Figure 15 which represents the code of the method `runAsClient()` of the object `theController` of the class `MineSweeperController`. We also assist our description with the following pseudo-code. We want to emphasise that the *Controller* must provide *View* with a reference to the client-side `ModelAdapter`. However, the *View* must initialise all the widgets (components of the GUI) including the panel to draw the environment of the game. Then, it has to establish the socket connection to the server side. Pass the corresponding output and input channels to the `ModelAdapter` so commands can be pushed down the socket). And finally, complete the setup of the *View* by setting up the default initial state of the game, making the *View* visible and signalling that is be redrawn.

```
1:  minesweeper_model.ModelAdapter theModelAdapter = new minesweeper_model.ModelAdapter(ExecutionModes.CLIENT);
2:  MineSweeperView theView = new MineSweeperView(theModelAdapter);
    case RESET_BOARD_TIME :
                    theModel.resetBoardandTime();
                    break;
    case SET_LEVEL :    GlobalModelInfo.DifficultyLevels alevel = GlobalModelInfo.DifficultyLevels.values( )[Integer.parseInt(theParts[1])];
                    theModel.setLevel(alevel);
                    break;
    case STOP_MODEL :
                    // nothing to do, connection should be closed
                  break;
    case TIMER_STOP :
                  theModel.timerStop();
```

Figure 13: Fragment of code of the `build_response` for the server-side `ModelAdapter` showing the selection of the action (command) and invoking the facade method of Figure 10

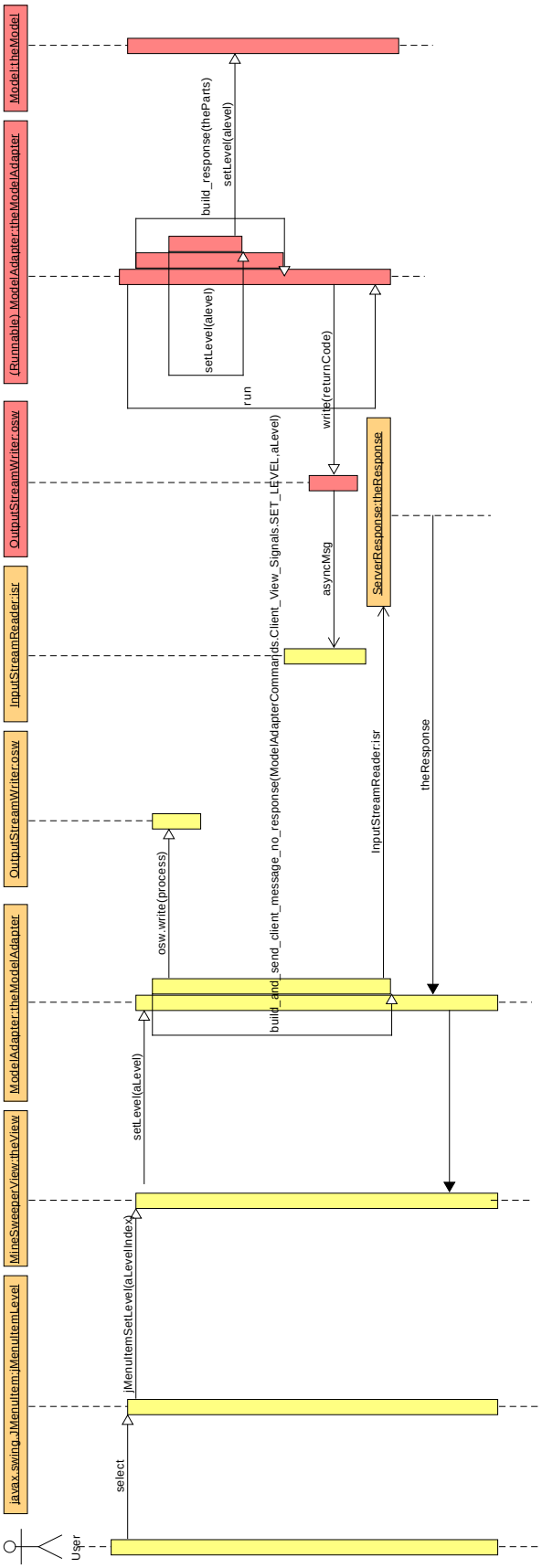

Figure 14: Sequence diagram illustrating the forwarding of an action in the GUI of the client (yellow) to the model of the server (red).

```
3:  theView.setModelAdapterLinkForPlayingArea();
4:  ViewAdapter theViewAdapter = new ViewAdapter(theView);
5:  InetAddress inetAddress = InetAddress.  getLocalHost();
6:  localIpAddress = inetAddress.getHostAddress();
7:  IPCollectView collectIP = new IPCollectView(""+localIpAddress);
8:  result = JOptionPane.showConfirmDialog(null, collectIP);
9:  server_IP=collectIP.getCollectedIP();
10:  Socket connection = new Socket(server_IP, port);
11:  BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream());
12:  OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
13:  theModelAdapter.setClientOutputStreamWriter(osw);
14:  BufferedInputStream bis = new BufferedInputStream(connection.getInputStream());
15:  InputStreamReader isr = new InputStreamReader(bis, "US-ASCII");
16:  theModelAdapter.setClientInputStreamReader(isr);
17:  theModelAdapter.setView(theViewAdapter);
18:  theView.JMenuItemReset();
19:  theView.setVisible(true);
20:  theView.repaint();
```

This pseudo-code is simpler than the actual code, because the actual code is surrounded of many exception handling cases. The pseudo-code and the communication diagram illustrate the linear scenario.

The goal of this set up is that the *View* will use the `ModelAdapter` agnostically as to whether it is operating stand-alone or in client server mode. That is, the *View* has no knowledge of whether all is local or whether there is a connection to a remote server. Step 1 in the pseudo-code is the constructor is the creation of the `ModelAdapter` (so we use dotted arrow in Figure 15). Step 2 is the constructor call for the *View*, and here the *View* set sup as many widgets as possible. Step 3 is a separate step to connect the panel for the game environment with the `ModelAdapter`. Step 5 and Step 6 extract a local IP address in case the server is in the same computer, so that the dialog from Step 8 has a default value. The retrieved IP address is in Step 9 is used in Step 10 to construct a socket. And output stream and an input stream reader are constructed from the connection until in Step 16 the input stream reader is provided to the `ModelAdapter` to collect responses from the server. Step 18 set the view in the default initial state (the effect of a reset of the game). Step 19 changes the property of the *View* that is a `JFrame` to visible, and we explicitly repaint it in Step 20 (otherwise an event needs to happen to it to be shown for the first time and this cannot be produce by the player as it is not showing yet).

The remaining issue is the counting down of the timer on the client side (on the GUI). The current design places the timer and the counting down in the *Model*. This decision is because how many second are left is information about the state of the game. If we were to have several clients competing for the same puzzle, it is easy to judge the central time for the score of the simultaneously competing players. At the moment, the value of the time is part of the message sent back from the server side to the client side every time there is a GUI event. For instance, every time a action (such as uncovering a cell) happens in the client, the response of the server, besides the outcome of whether the cell has a bomb or not, the current time on the server is included in the response. This should update the value displayed in the GUI on the client side. However, this is not sufficient. If the player does not interact for longer than second with the GUI, the timer value is not updated. So, the current version is missing a timer on the client side that pings the server regularly for the value of the time at the server. This mechanism is also useful for testing the liveliness of the connection when the player does not interact.

The activity diagram in Figure 16 shows the decision process at the start of the program for the launching as a stand-alone game, a client side version or a server side version.

Figure 17 shows how we plan to make the first connection to run as master connection. That means the game will start for all connections when the first connection presses the start button. Connections will be managed in a dictionary with the number of the connection as an ID. If some clients finish early, they will need to wait for the master connection to reset the game to record their score. Only when all other connections have terminated can the master connection reset the game. Secondary connections can be established only once there is a master connection. Resetting the game terminates all connections.

2.5 Level of sophistication

The current version has not raised the level of sophistication. As illustrated by the incomplete functional requirements, the current version does not record the scores of players at all. The plan remains to use `derby` which is a relational data base that can run in embedded format. For stand-alone executions of the game, the database would

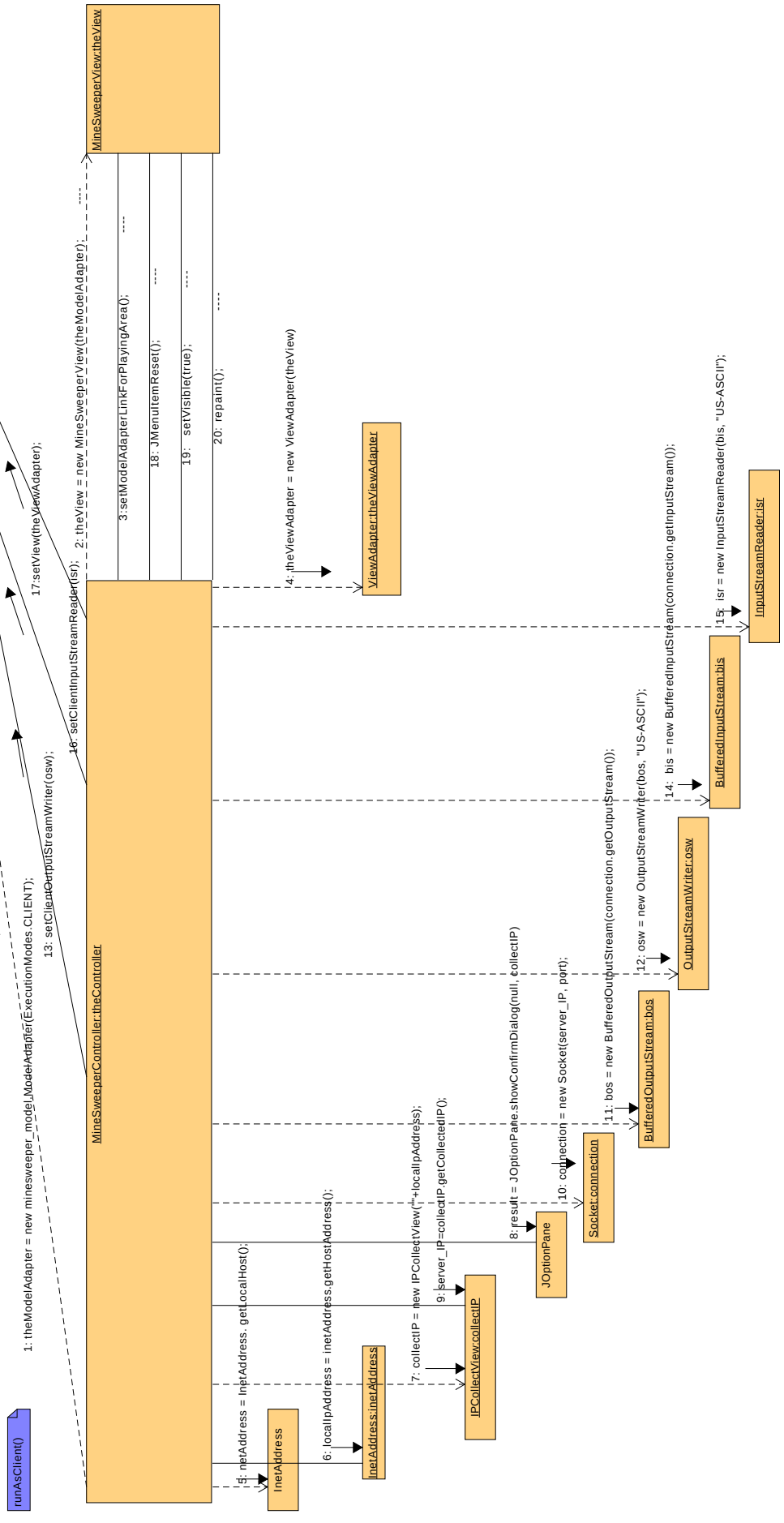

Figure 15: Communication diagram of the controller set up on the client side, in particular for the method `runAsClient()` of the object `theController` of the class `MineSweeperController`.

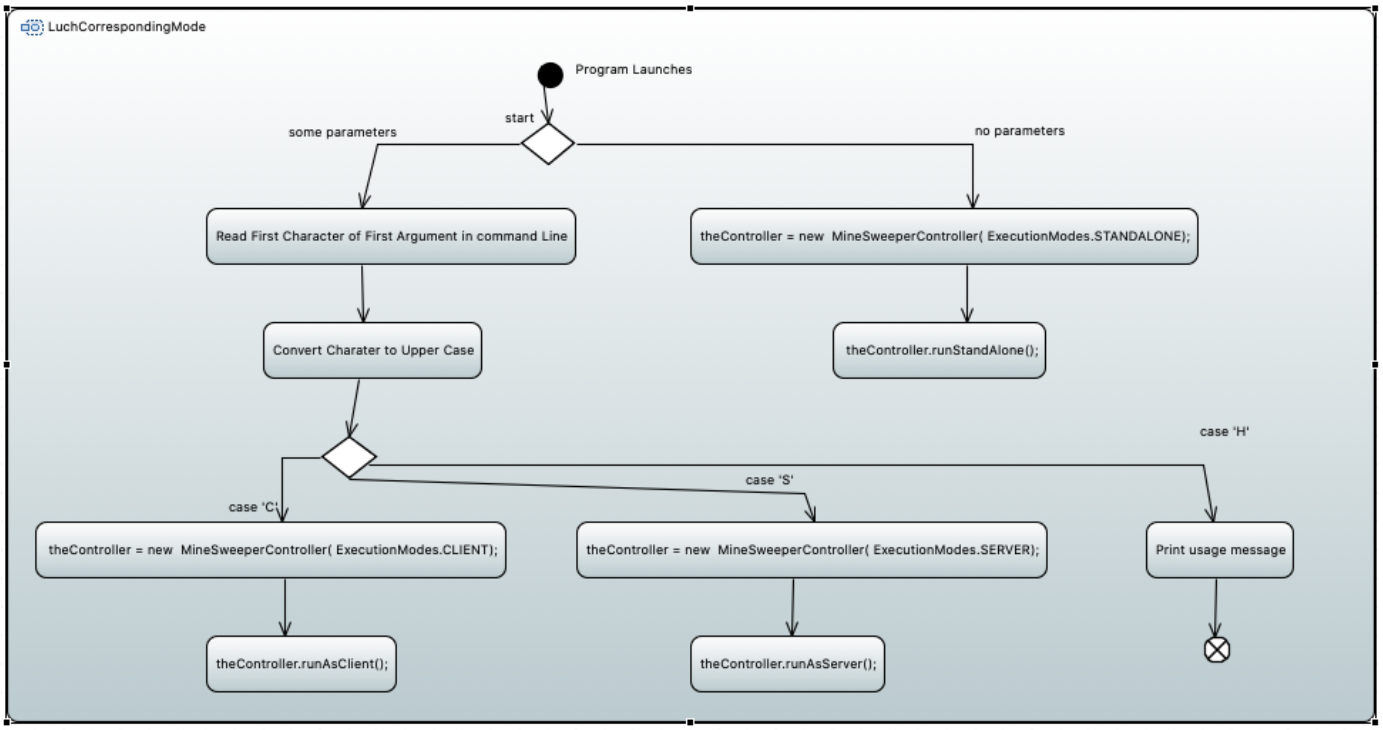

Figure 16: The decision process for launching the program in server mode, client mode or stand-alone mode.

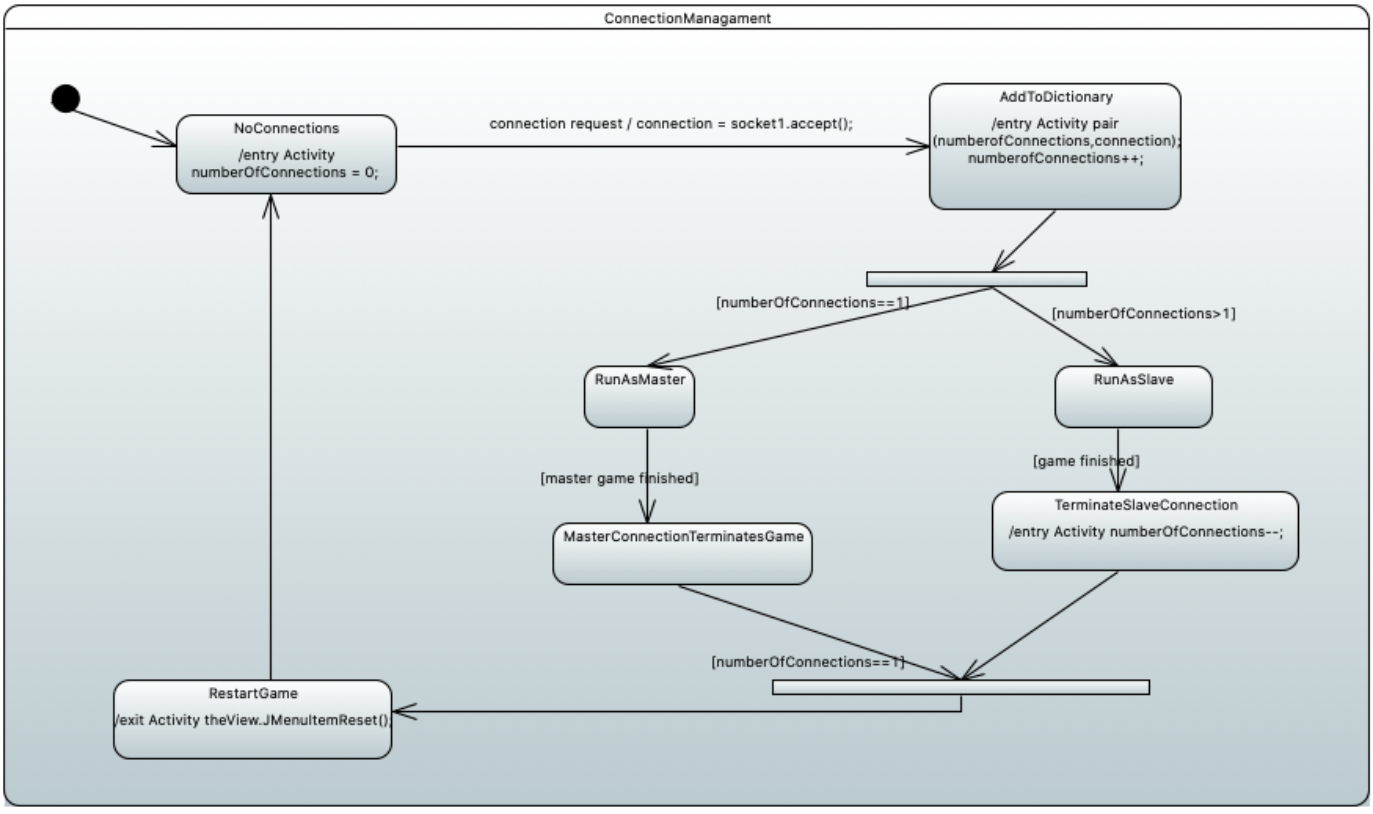

Figure 17: Managing connection on the server side. The first connection shall wait for all others to be able to change the state to a situation where the game can be reset by the first connection.

run in stand-alone format, locally recording the scores. A relational schema would need to be defined to keel data about `players` (as one entity (their name, and last name at least) and of the `game setting` (environment, density of mines, level of difficulty) and finally, the corresponding many-to-many association between `players` and `game setting` should record some attributes such as `time of completion`, `score`, `date` and `time`.

For the client/server version, the embedded `derby` would be located on the server side only.

The plan is to progress from a version with no encryption to a version with encryption. The encryption also would have two phases. One where there is an administrator password and a second one where each player records a password. The sophistication of requirements includes the next level by which users can recuperate access when forgetting a password by using some alternative mechanism.

2.6 Testing

No formal tools for testing have been used. The project would have benefited from some unit-testing frameworks to to ensure the introduction of new functionality does not regress with respect to previous functionality. At the moment all testing is manual. See tables below. The tests are repeated systematicly going over the ID of the test and recording the outcome.

In the future, a series of further test will be required when the environment is advanced to hexagons and graphs. Similarly, new tests would need to be designed when we introduce persistence to check that scores for players are properly recorded.

The series of test sessions regularly conducted is listed below. A record of the testing is described by the following tables. Table 2 is a record of the testing for the stand alone version. Table 3 is a record of the testing for the client/server version.

PROJECT:	Mine Sweeper Game			
MODULE:	Stand Alone Functionality: square grid environment			
REQUIREMENT:	F-R 1, F-R 2, F-R 4, F-R 5, F-R 6, F-R 7 F-R 8 F-R 9 F-R 10			
TEST CASE ID:	Test 01			
TEST OBJECTIVE	Test the GUI and capacity to play traditional Mine Sweeper			
TEST DATE and TIME	9AM: September 1st, 2020			
Step No	**Steps**	**Data**	**Expected Results**	**Actual Results**
1	Launch game, GUI starts and can be closed	No argument in command line	Application terminates	As expected
2	Launch game, select menu item for difficulty level	Click on different levels	Application redraws with more cells as higher difficulty level	As expected
3	Launch game, select button to start the game	Click on large yellow button	Application shows time counting down	As expected
4	Launch game, select button to start the game	Click on large yellow button	Application shows total number of mines	As expected
5	Launch game, select button to start the game	Click on large yellow button, attempt to change the level of the game	Application shows level options disabled	As expected

6	Launch game, start the game, hit a mine	Click on large yellow button, uncover cells aiming to find a mine	Application ends game and shows mine locations	As expected
7	Launch game, start the game, flag cells repeatedly	Click on large yellow button, flag cells until mien counter decreases, then uncover cells	Application ends game and shows mine locations, verify flags on cells without mine had no effect; flags on mines decremented mine counter	As expected
8	Launch game, start the game, flag cells repeatedly	Click on large yellow button, flag cells until mien counter decreases, then uncover cells, restart the game	Application ends game and shows mine locations, verify game can be reset, and Test 01-7 can be repeated	As expected
10	Launch game, start the game, flag cells repeatedly	Click on large yellow button, flag cells until mien counter decreases, then uncover cells, restart the game	Application ends game and shows sad face	As expected
11	Comment code and fix random seed, Launch game, start the game, complete the game	Click on large yellow button, uncover cells aiming to complete the game, record position of the cells, restart the game and place flags in known cell positions	Application ends game and shows happy face	As expected

12	Repeat Test-01-11	Click on large yellow button, uncover cells aiming to complete the game, record position of the cells, restart the game and place flags in known cell positions	Application ends game and timer halts	As expected

Table 2: Record of testing performed for the stand alone version.

PROJECT:		Mine Sweeper Game		
MODULE:		Client Functionality: square grid environment		
REQUIREMENT:		F-R 1, F-R 2, F-R 4, F-R 5, F-R 6, F-R 7 F-R 8 F-R 9 F-R 10		
TEST CASE ID:		Test 02		
TEST OBJECTIVE		Test the GUI and capacity to play traditional Mine Sweeper from a separate client than the server holding the data (model)		
TEST DATE and TIME		10AM: September 1st, 2020		
Step No	**Steps**	**Data**	**Expected Results**	**Actual Results**
1	Launch the server, launch the client, repeat tests TEST-01-01 to Test01-12	Argument "Server" in command line for server, argument "Client" for client	See Test-01 suite	Test 01-3 and Test 01-12 fail All others pass
2	Launch the server, launch the client, terminate a game with client, launch second client	Argument "Server" in command line for server, argument "Client" for client	See output in server accepting second connection	As expected
3	Launch the server, launch the client, terminate a game with client, launch second client, play game with client	Argument "Server" in command line for server, argument "Client" for client	Second client carries Test-02-01 tests	All behaviour fails

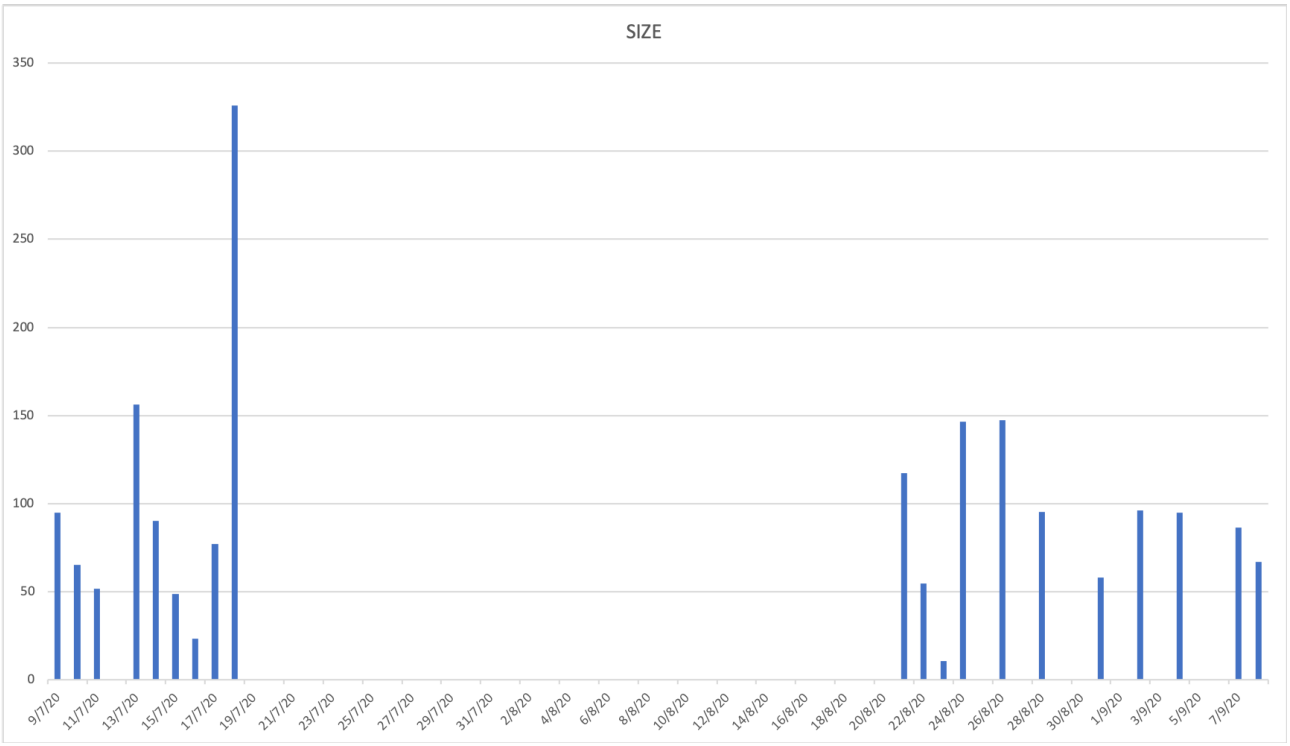

Figure 18: Commit size against date.

Table 3: Record of testing performed for the client/server version.

3 Analysis of effort

We have been using `git` as our version control mechanism. We can extract the date and time of out commits with the command

```
git log .  > commits.log
```

A typical entry looks like

```
commit 190fa3d15b73f462024a9540962ddfa77f521c08
Author:  Vlad <v.estivill-castro@griffith.edu.au>
Date:  Fri Sep 4 16:06:00 2020 +1000

A detailed view added
```

We can extract and approximate size of the commit with

```
git diff 190fa3d15b73f462024a9540962ddfa77f521c08 259cc73fcbdf0aba777bfd0a76ebd5079cc4d804 | wc -l
```

Using this we constructed a table with our 90 commits as rows. The MS-excel includes the date (day, month, tear) and time (hours, minutes,seconds) and day of the week (Monday to Sunday) as columns. The last column is the computed size of the commit. Figure 19 shows a plot of the size of the commit against the date. We can see that the commits have a separation between the first milestone of the project and the second milestone.

However, the data of the commits can be analysed in many ways. Figure **??** shows the commits and their time. We can see that there are some days fully dedicated to the development, while other days only a few hours were committed. Most of the work is on Fridays, with also some load on weekends. There is almost not commits on a Monday or a Tuesday. This shows the developer was busy with many other simultaneous projects and activities.

We can see that the programming of Milestone 1 took approxiamtedly 11 hours. Spread over 9 days. The further development of Milestone 2 took 12 hours spread over 11 days

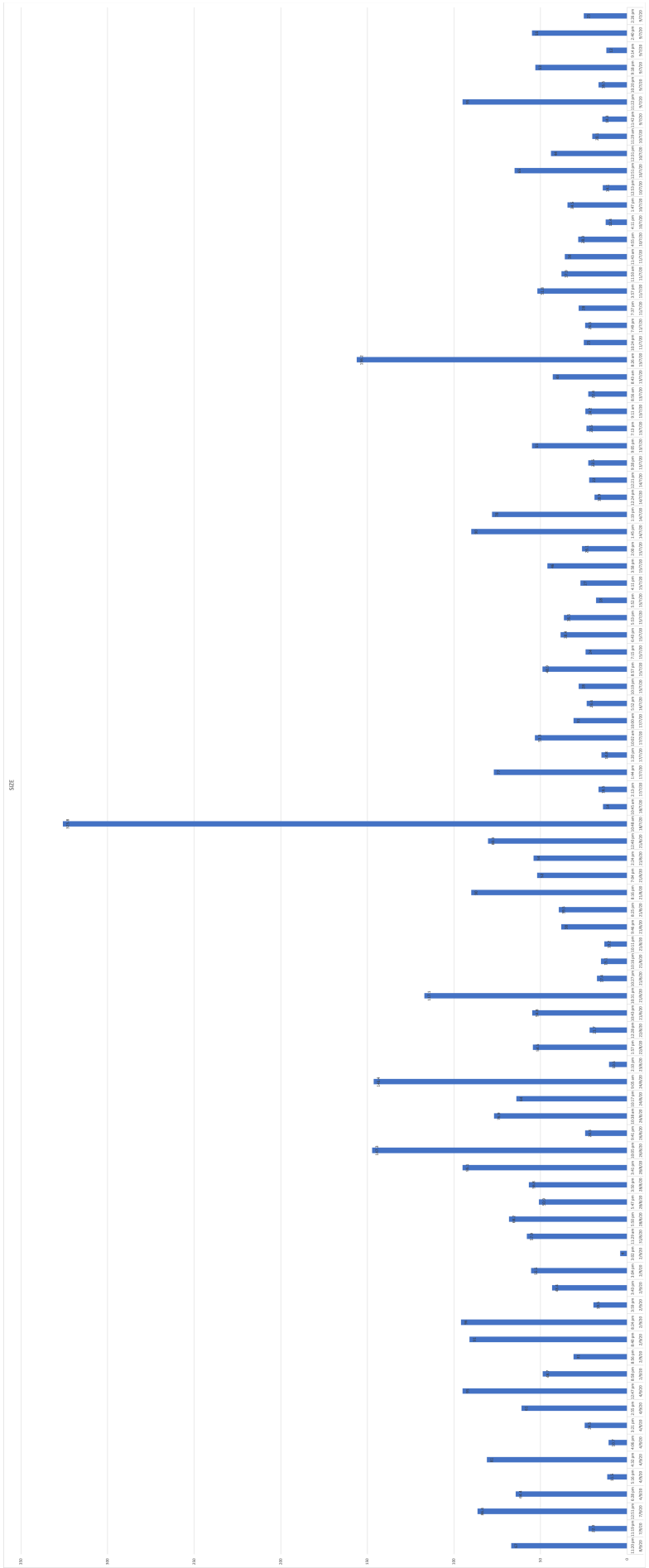

Figure 19: Commit size against time and date.

4 Putting UML to use.

4.1 Design

The static structure of the entire design is illustrated by Figure 2. To present the design in a more accessible manner Figure 3 focuses on the static relation of the classes in *View*. These classes are encapsulated in a package. To see internal details of the classes, we redraw the package for the *View* in Figure 4.

The package for the controller has only one class and this is illustrated in Figure 7.

For the *Model*, we have a global overview of this package in Figure 5. A diagram that shows *Model* classes with their properties (to indicate the complexity of the classes) is provided by Figure 6.

4.2 Associated documentation

4.2.1 UML artefacts

1. For Dynamic Modelling
   (a) Interaction diagrams describing the behaviour between objects
      i. At least one sequence diagram with the set of participating objects arranged in sequences columns and time flowing from top to bottom.
      See Figure 14 for the use of a sequence diagram.
      ii. At least one collaboration diagram, perhaps showing the sequence of events with numbering labels.
      IN UML 2, collaboration diagrams have been renamed as *Communication Diagrams*. In Figure 15 we included a communication diagram to show how the controller sets up the socket connection in the client side, and set up the view to know about the model adapter (who has input and output buffers down the socket connection).
   (b) State-based diagrams.
      i. At least one state machine that describes the flow of states for a given class in response to the outside stimuli.
      A state-machine diagram was shown in Figure 17.
      ii. At least one activity diagram, where all the states are actions states (sometimes called a Moore Automaton).
      An activity diagram was presented in Figure 16.
2. For Structural (static) Modelling
   (a) A class diagram.
   Class diagrams appear in Figure 2, Figure 3, Figure 4, Figure 7, Figure 5, and Figure 6.
3. For the use of Software Patterns.
   (a) Describe at least one implementation of a software pattern illustrated by some existing code in the current version of the project or prototype.
   The current version of the prototype demonstrates two software engineering patterns.
      i. *Model-View-Controller* (MVC) where responsibilities for presentation, control, and state-information are separated. Although not completed, the design is there to have several views on the same model. (A starting point on this topic is the corresponding Wikipedia page for the software engineering patterns).
      ii. *Adapter* (also known as wrapper, an alternative naming shared with the decorator pattern) that allows us to offer a interface to the *Model* hiding whether the *Model* is local or remote. (A starting point on the Adapter software pattern is the corresponding Wikipedia page for the Adapter pattern). Figure 20 display the automaticly generated documentation for our new class `ModelAdapter`.

   The current version demonstrates the anti-pattern of the software engineering pattern *Command*. The classification for the *Command* design pattern is as **behavioural**. Here, we should define an abstract class (which would be named *Command*). The abstract class has an abstract method `execute()` (or

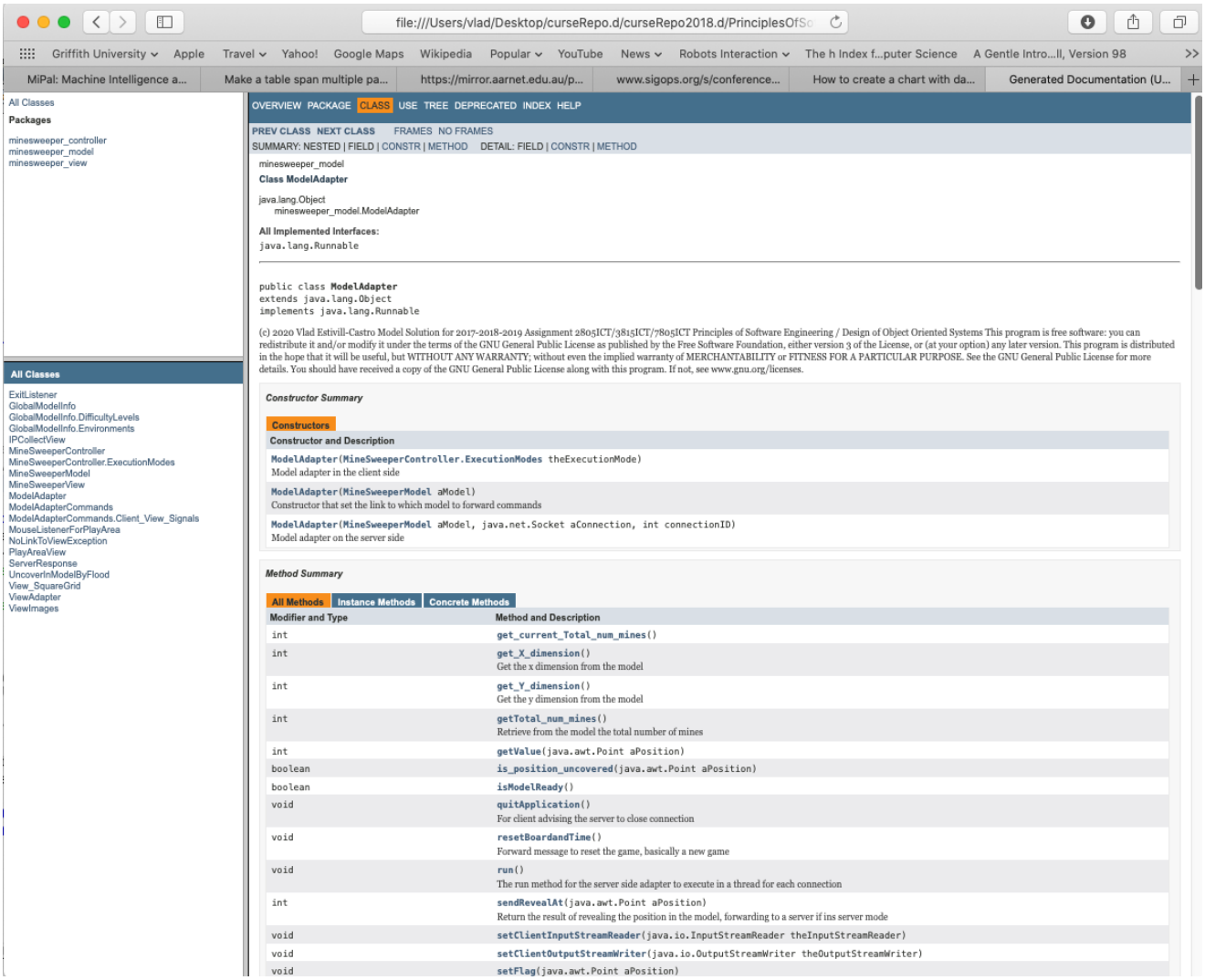

Figure 20: Automaticly generated page for the class `ModelAdapter` introduced in this milestone and illustrative of the Software engineering pattern /emphAdapter.

`perform()`, indicating all objects of subclasses of *Command* have all the information required to run, execute, perform the required command (or action) for execution when such methods is called.

The current version demonstrates two software engineering architectural patterns. These type of patterns emerge from software engineering patterns that are applied to more global and fundamental structures of a software system.

i. *Client/Server*: (A starting point on the client/server architecture is the corresponding Encyclopedia Britannica page for Client-server architecture).
ii. *Model/View/Controller*: (A brief presentation of MVC as software architecture is the corresponding Blog by Dennis Vera).

NOTE: A video running the current state of the code appears in the Collaborate Ultra live presentation of Workshop 07.