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

Vladimir Estivill-Castro

October 8, 2020

Objectives

1. Explain design principles including of least privilege and fail-safe defaults, separation of concerns, information hiding, coupling and cohesion, and encapsulation.
2. Describe the design process for a software development project for each of the main software design methods
3. Create appropriate system models for the structure and behaviour of software products from their requirements specifications
4. Use a design paradigm to design a simple software system, and explain how system design principles have been applied in this design.
5. Select an appropriate software architecture as the design basis for a given software requirements specification, justify the selection based on its advantages over alternative architectures.
6. Create software programs that make use of appropriate design patterns.
7. Create user interface software using either event driven or call-back based designs
8. Explain the importance of Model-View controller to interface programming.
9. Discuss the properties of good software design including the nature and the role of associated documentation.
10. Create appropriate design documentation for a variety of different designs.

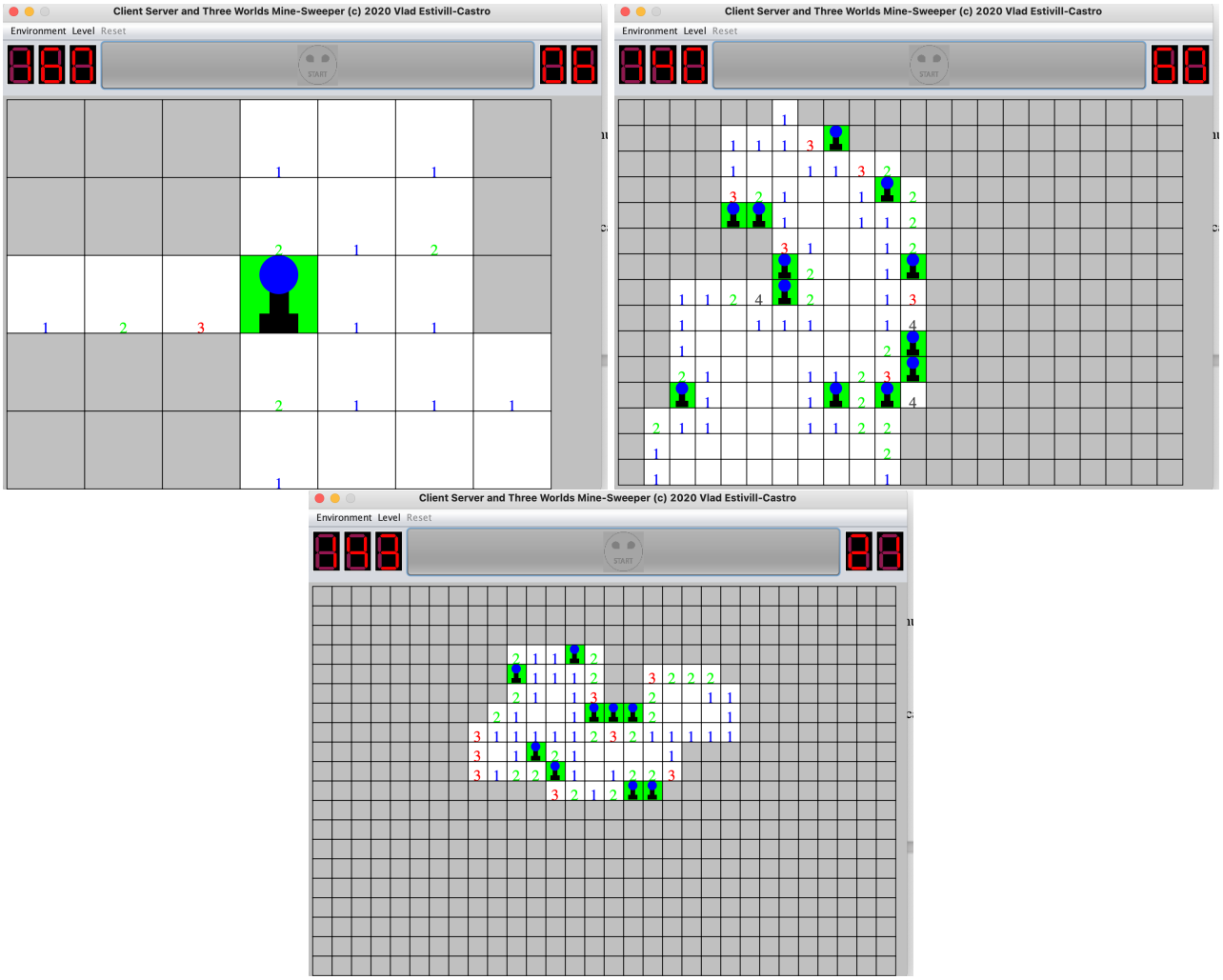


Figure 1: Different stages of the *Mine sweeper* when the environment is a square grid.

A demonstration video

A demo video is available at
echo360.org.au/media/0f90b8b0-0d0a-46be-86ae-048ee128dc37/public.

Your log of your version control software

A zip file of executing `git log > the_log` in the repository of the project is attached. We note that

```
grep Author the_log | wc -l
```

shows a total of 1044 commits during the length of the project.

Final reflective report

Fully functional implementation of Task 1

Task 1	Your first task is to implement in your favourite programming language and using your favourite programming tools a version of *Mine sweeper* where scores are the time taken for a particular dimension of the board. However, this is just one phase of the overall challenge, and you should consider designing your software architecture and software components so other tasks are also achieved.

This was completed in Milestone 1. Figure 1 shows three screen-shots of the GUI while playing traditional *Mine sweeper* which in our implementation has 3 levels (novice, competent and expert).

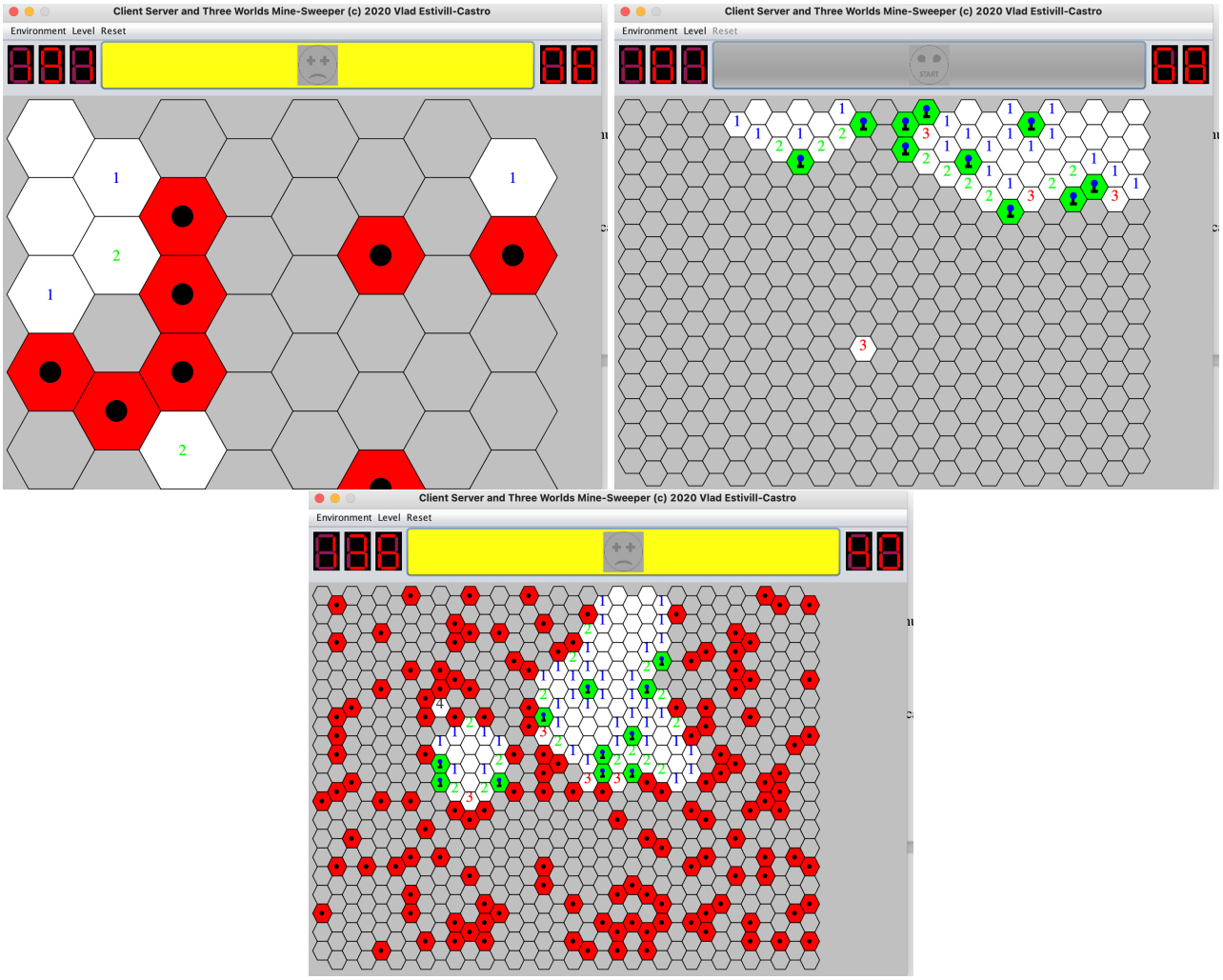


Figure 2: Different stages of the *Mine sweeper* when the environment is an hexagonal grid.

Fully functional implementation of Task 2

Task 2

> Your second task is to implement in your favourite programming language and using your favourite programming tools a version of *Mine sweeper* where scores are the time taken for a particular dimension of the board. However, this time your program should enable two version of the games, and array of square cells and also enable the user to chose an array of hexagonal cells.

This was completed as per this milestone (Milestone 3). Figure 2 shows three screen-shots of the GUI while playing hexagonal *Mine sweeper* which in our implementation also has 3 levels (novice, competent and expert).

Fully functional implementation of Task 3

Task 3

> Your third task is to extend your implementation so that the new colouring based *Mine sweeper* is also an extension of the game. Note that you are not required to represent the game in any sort of grid. The colours version of the game only requires to show the topological information (what node is connected with what other node by an edge) and whether the node is coloured or covered.

This was completed as per this milestone (Milestone 3). Figure 3 shows three screen-shots of the GUI while playing the topological version of *Mine sweeper*; which in our implementation also has 3 levels (novice, competent and expert).

Cross platform implementation of the three tasks

To facilitate this point the programming language was `java` using the most standard utilities and widgets from Java 8 and the executable is distributed as a `jar`. The videos are all shot on macOS (actually Version 11.0 Beta of

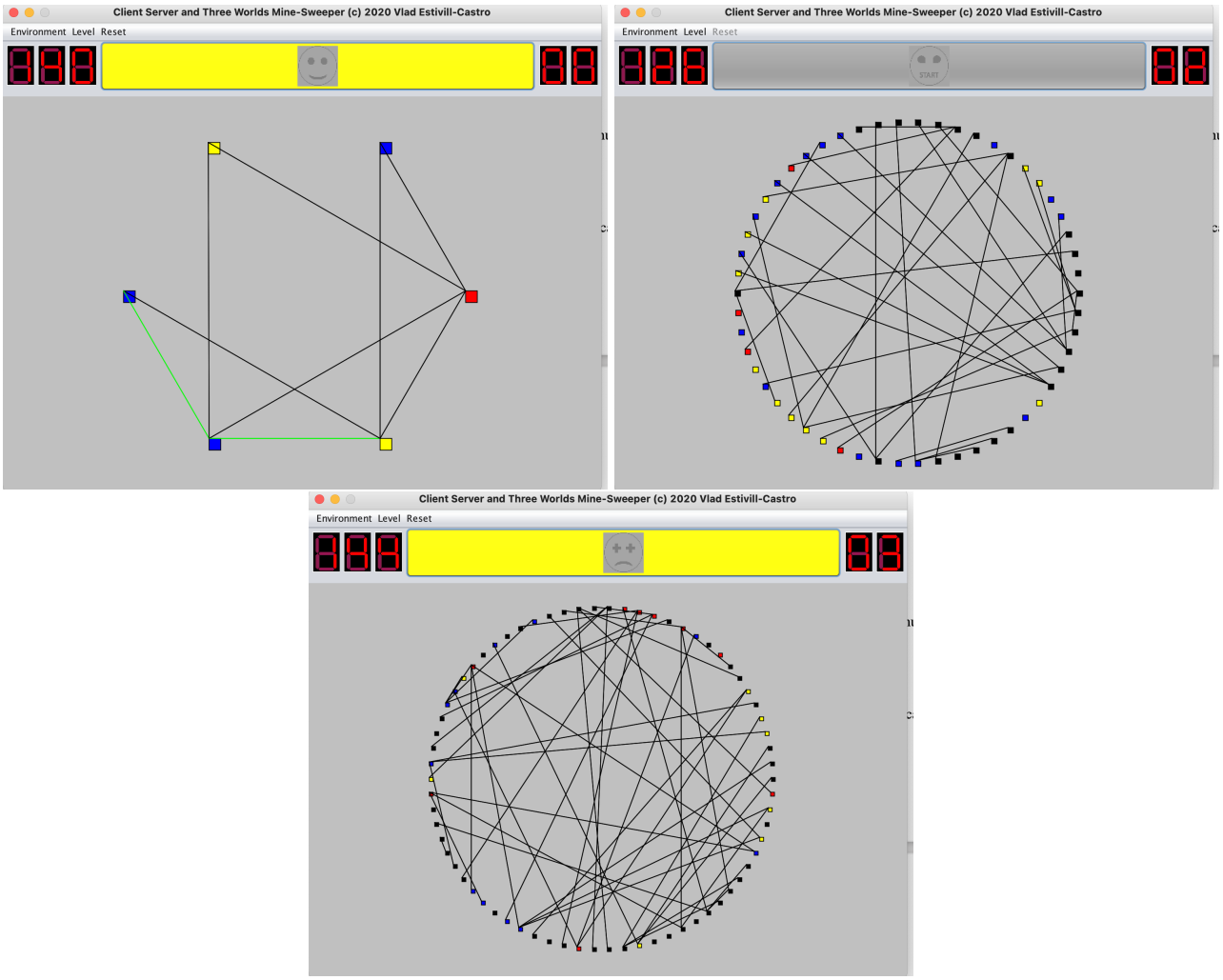


Figure 3: Different stages of the *Mine sweeper* when the environment is an 3-coloured graph.

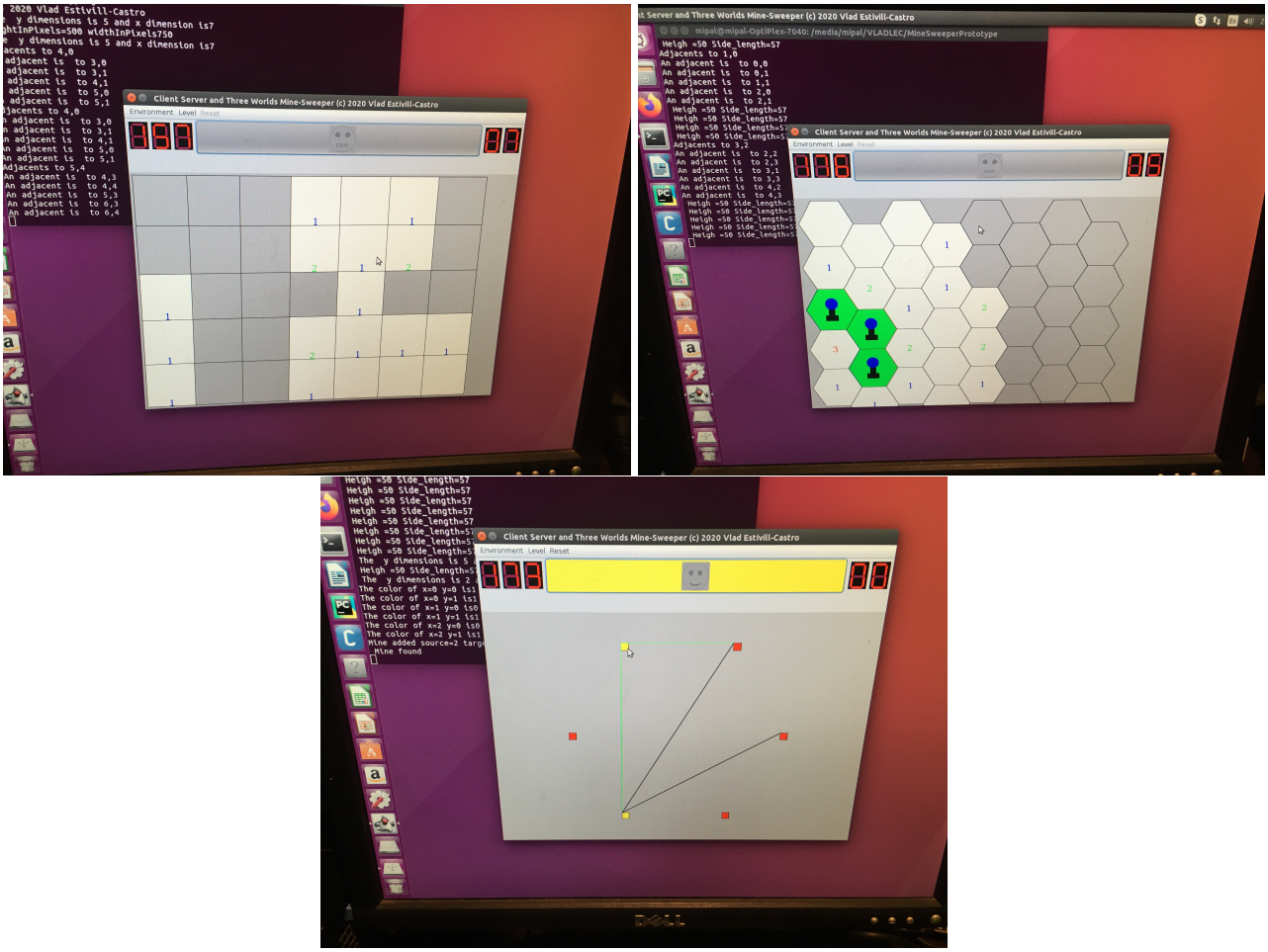


Figure 4: Different environments of the *Mine sweeper* when running on Ubuntu.

macOS Big Sur). We make the observation we had tested in several earlier versions of macOS (but now tested in one that has not even been officially released).

For completeness, the screen-shot in Figure 4 shows the execution of our software on Ubuntu 16.06. While again we tested several newer version of Ubuntu, we make the point here of showing a version so old that is not serviced any more.

Evidence of good software development practices (version control, unit testing)

For version control we used `git`. With Milestone 2 we submitted an analysis of our log till that time. Also, earlier in this documented we commented that a `zip` file accompanies this submission with a compressed version of the output of the `git log` command showing over one thousand commits.

We incorporated `JUnit` and unit testing for several of our classes. `JUnit` is compatible with our Integrated Develeopment Environment (IDE) `netbeans` (in fact, we used netbeans 8.1, netbeans 8.2 and netbeans 12.1). Figure 5 shows the results of our unit test on our model class.

It is important to mention hat some of these unit test actually launch the GUI, and a few second later operate on the model. That is why we see in the image of Figure 5 that testing this class takes over 7 seconds. Figure 19 shows some of this long test, because we launch the GUI and let a few seconds pass after the timer is started to check the time variable does count down.

An improvement would be to create some integration testing, and be able to run the application headless (without a GUI) but enabling the testing of other classes as well.

In Milestone 2 we presented a series of acceptability testing scenarios that are not reproduced here. An improvement with be to incorporate such acceptance testing scenarios in scripts for a continuous integration server. That way, more comprehensive testing could have been incorporated for many more test enabling the development without labour intensive reproduction of regression test (that is, repeating some test to make sure the introduction of a new version of the environment does not introduce faults in already working environments).

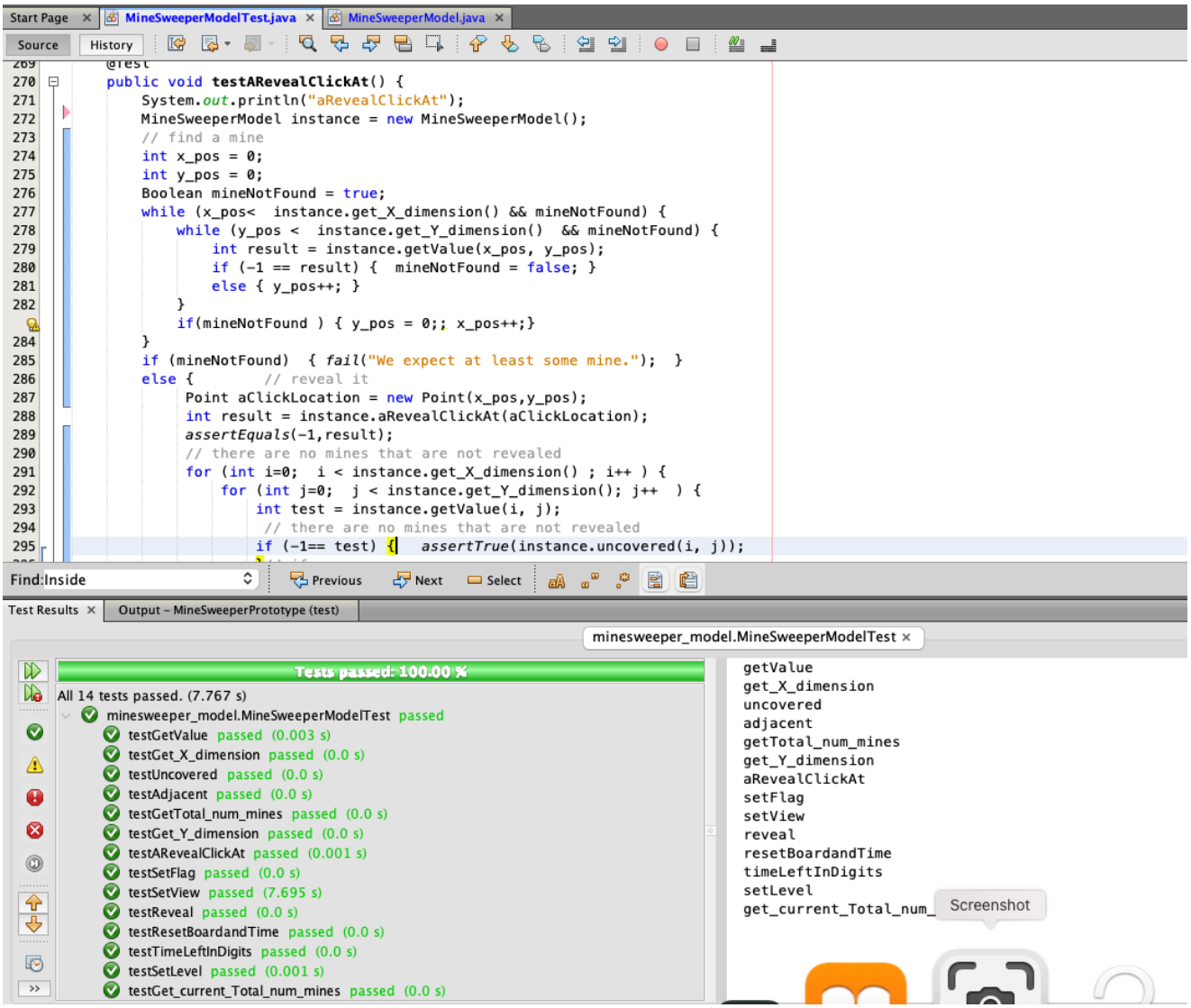


Figure 5: Results of running the the test file configured with `JUnit` for the class `minesweeper_model/MineSweeperModel`.

Illustration of Objective 1

In this project we have used an Object-Oriented design. Therefore, several elements of Object-Oriented technology are used to support design principles. We also take advantage of some of the features of the `java` programming language. In particular, we follow principles of *least privilege* by always declaring methods and variables as private in each class. The `netbeans` environment alert us if we are making use of a method or variable that is private in another class or package. Then, we can decide if the best design alternative is to make it public or to encapsulate such access by another more appropriate access approach.

Similarly, for the principle of `fail-safe defaults`, each class is provided with the minimum level of privileges and accesses to other classes. We declared almost all private variable sin our classes as `static`, not only for efficiency, but because there is only one project of that class. For instance, there is only one model. Perhaps this could have been addressed with programmatically enabling a `SINGLETON` software engineering pattern. For this simple program, we considered that to be unnecessary.

We followed the principle of *separation* of concerns with the implementation of the graph version of *Mine sweeper*.

Illustration of Objective 2

This project was developed using iterative refinement from a prototype. In fact, we developed a prototype in Milestone 1 where the focus was the functionality of the standard square-grid environment of the original *Mine sweeper* game. The prototype enabled the evaluation of `java` swing for the interface and for the event-driven programming. It enabled to evaluate cross-platform requirements and the capability of java-swing to manage the play-environments for hexagonal grid and topological grid. The prototype was developed and completed by the scheduled deadline of Milestone 1. The document submitted with Milestone 1 also enabled us to consider the structure of the design and its migration to a general architectural patters of Model-View-Controller. Evaluating the prototype was, therefore, very important to complete the other elements of the project (the other environments and tasks). We stress that getting the prototype going was very important to define requirements and to prioritise them as well (see reports and submissions for Milestone 1 and Milestone 2). The prototype also enabled the classification of responsibilities of the GUI (and later the view) and design the split to place the model module in the server. We were able to define the interface between the model and the server, even though by Milestone 3 the commands and signals between the Client and the Server expanded, they followed the same patter determined in the prototype.

The completion of the prototype was a major step in assessing the capacity to complete the tasks and the functionality of other play environments. Clearly, the inability to complete a prototype for just the square-grid environment would have been a strong indicator that the full functionality was beyond reach. Completed a prototype is good to get us going, and to organise the plan for the next releases of the project. The next potential unknown or risk was whether a client/server architecture would be possible and the other environments became less important. Nevertheless, the topological environment is much different from an extension to the square grid. In particular, the hexagonal-grid environment is not so different from the square gird, just the drawing of the GUI (game area) is what is different. It became clear that completing the hexagonal-grid environment would not be a complex task. Using a prototype turned the project in a iterative refinement approach. We just needed to add some feature or small requirement each time. This enabled also the scheduling of effort to the project since we usually could only work on the project for small snippets of time. Therefore, it was useful to plan it this way and take the iterative refinement approach.

In summary, the initial prototype was a comprehensive demonstration of the feasibility of the approach, the building tools and the plan for completion.

Some of the concerns with prototyping is that the prototype may be useless for later development and we need to build the system twice. In fact, the final version that has the 3 environments still could be considered a prototype. Many of the algorithms used are currently the simples most immediate version to achieve the functionality and not necessarily the most robust and advanced version possible. For example

1. The view uses a very simple approach to compute the embedding in the drawing area of the hexagonal grid and the graph environment. In the case of the hexagonal grid, another row of hexagons at the bottom adds a length equal to the height of the largest inscribe circle in the hexagon. This is the height of the six equilateral rectangles that make a hexagon. Adding a column of hexagon (expanding the drawing of the environment left to right) adds an amount equal to the side of the triangles. The difference is about 80% since the height of the triangle is about $cos30^\circ$ times the length of the side.

Similarly, the layout algorithm for graphs and edges was simple to place the nodes as vertices of a regular n-gon. There are much interesting algorithms to draw a graph in the plane. When the graph has only 6 or 8 nodes, the environment looks fine. However, once we work worth the COMPETENT and EXPERT level the layout is not attractive.

2. The algorithms to compute how many miens to place for a difficulty level are simple. So are the algorithms to compute the corresponding score for a certain completion time, and difficulty level.

3. The data structures for the view and the model are currently the simplest that would meet reasonable performance

4. The client/server signals could potentially be optimised for more efficient communication using sockets. They could be even be encrypted for security and privacy non-functional requirements.

Beside prototyping we applied iterative enhancement (or refinement). Our software was built as a sequence of enhancements. In Milestone 2 we incorporated the client/server modes and enabled the remote execution fo the GUI from the server, enabling one model and several views to offer competitive games. That is several players on the same environment operating concurrently.

The third enhancement was deployed with Milestone 3 where we incorporated the hexagonal nd the topological version of the game. In each enhancement some major design (or re-design) took place. For instance, from Milestone 1 to Milestone 2 a clear separation between Model-View-Controller modules was necessary to place the Model on the server side, and the View and the Controller on the client side.

In the move to Milestone 3, some software engineering design patterns were applied, in particular the *command* pattern was implemented to enhance the signals between the client end the server for the two new environments. We could say that the enhancements were also drive by Test-Driven Development since we incorporated `JUnit` as a framework for unit testing. This enabled to confirm that the enhancements were not detrimental to the functionality already achieved for the early version of the software. The iterative enhancement ensured we had a working version of the software at all times. Moreover, as we already mentioned it facilitated the overall planning of the development and facilitated the decision of what task to complete next. We typically selected the least complicated way or functionality to incorporate next. Such choice of the easiest task next may have the drawback that some re-design or re-factoring may occasionally be required and some major restructuring may become necessary. However, for this overall small piece of software that did not happen. nevertheless, there is significant improvement still potential to the current code. Among these re-design improvements are the following

1. Re-evaluate that the `model` is at the moment a monolithic class. It may be sensible to actually create a class hierarchy, factor out into a super-class aspects common to all models, and make subclasses for models according to the environment. The rational for this is that in the code we see to many test (if-statements) that check whether the environment is hexagonal or square (and to the then-part) and the else part corresponds for a rather different algorithm in the topological environment. For instance, Figure 6 shows a method where we test the type of environment and according to that a different possibilities the current number of mines is computed. This highlights that there are possible subclasses that implement differently the responsibility of keeping track of the current number of mines (as the user correctly flags them, the count goes down if it reaches zero the user wins the game). In the case of the square-grid environment and the hexagonal-grid environment, the current total of mines is computed by scanning the representation of the board (a two dimensional array of cells) and counting the mines left. In the case of the graph-environment, the representation for the model is not to have a data structure that records whether an edge is a mine or not, but just to have an integer count. As explained earlier this is the most immediate implementation that is efficient enough.

2. In the view we did implement different classes for a view for the square layout, the hexagonal layout and the graph-layout; however, there is still opportunity to factor code into a super-class. For instance, in Figure 7 we see a method in the class `PlayAreaView`, which is a class in the package `minesweeper_view` (that is in the *View*). Here, we have introduced different classes for the corresponding views. However, the code still queries the model fo the type of environment and then selects the corresponding static methods of the corresponding class. The design could be improved if the relationship between which view corresponds to which environment was establish at a higher super-class, and then the code was agnostic of the specific view type.

```java
/**
* The active mines are those cells with -1
* @return a the number of mines not flagged.
*/
public int get_current_Total_num_mines() {
    int total =0;
     if  ((GlobalModelInfo.Environments.SQUARE==environment)
        ||  (GlobalModelInfo.Environments.HEXAGONAL==environment)  )
        {
            for (int x_index=0;x_index < x_dimension; x_index++ ) {
                for (int y_index=0; y_index < y_dimension; y_index++ ) {
                 if (-1== square_board[x_index][y_index]) total++;
            }}
       }
     else { total=  current_total_of_edgeMines;
    }
    return total;
}
```

Figure 6: One method in the class `MineSweeperModel` where we can see testing of the type of environment. This could be re-designed into a class hierarchy.

```java
/**
* Obtain the click location of a player in the game area
* @return Obtain the cell id as cell coordinates
*/
private Point clickLocation (int x_on_board, int y_on_board) {
                int x_noborder = x_on_board-BORDER_IN_PIXELS;
                int y_noborder = y_on_board-BORDER_IN_PIXELS;
                Point coordinateOfTagetCell = new Point(-1,-1);
                GlobalModelInfo.Environments theEnvironment =GlobalModelInfo.Environments.SQUARE ;
                try {
                            theEnvironment = theModel.getEnvironment();
                } catch (IOException e) {
                    JOptionPane.showMessageDialog(this, "Lost conenction to Server", "Connection  error",JOptionPane.ERROR_MESSAGE);
                    System.exit(1);
                }

                if (GlobalModelInfo.Environments.SQUARE==theEnvironment) {
                            coordinateOfTagetCell = View_SquareGrid.theCoordinatesOfTargetcell(x_noborder, y_noborder );
                        }
                else if (GlobalModelInfo.Environments.HEXAGONAL==theEnvironment) {
                     coordinateOfTagetCell = View_HexagonalGrid.theCoordinatesOfTargetcell(x_noborder, y_noborder );
                }
                else if (GlobalModelInfo.Environments.TOPOLOGICAL==theEnvironment) {
                     coordinateOfTagetCell = View_TopologicalLayout.theCoordinatesOfTargetcell(x_noborder, y_noborder );
                }

                //System.out.println("<***>The cell selecyed is x="+coordinateOfTagetCell.x+" and y="+coordinateOfTagetCell.y);
                return  coordinateOfTagetCell;
}
```

Figure 7: One method in the class `PlayAreaView` where we can also see testing of the type of environment. Although this has different view classes for each, it can be improved and re-designed into a class hierarchy.

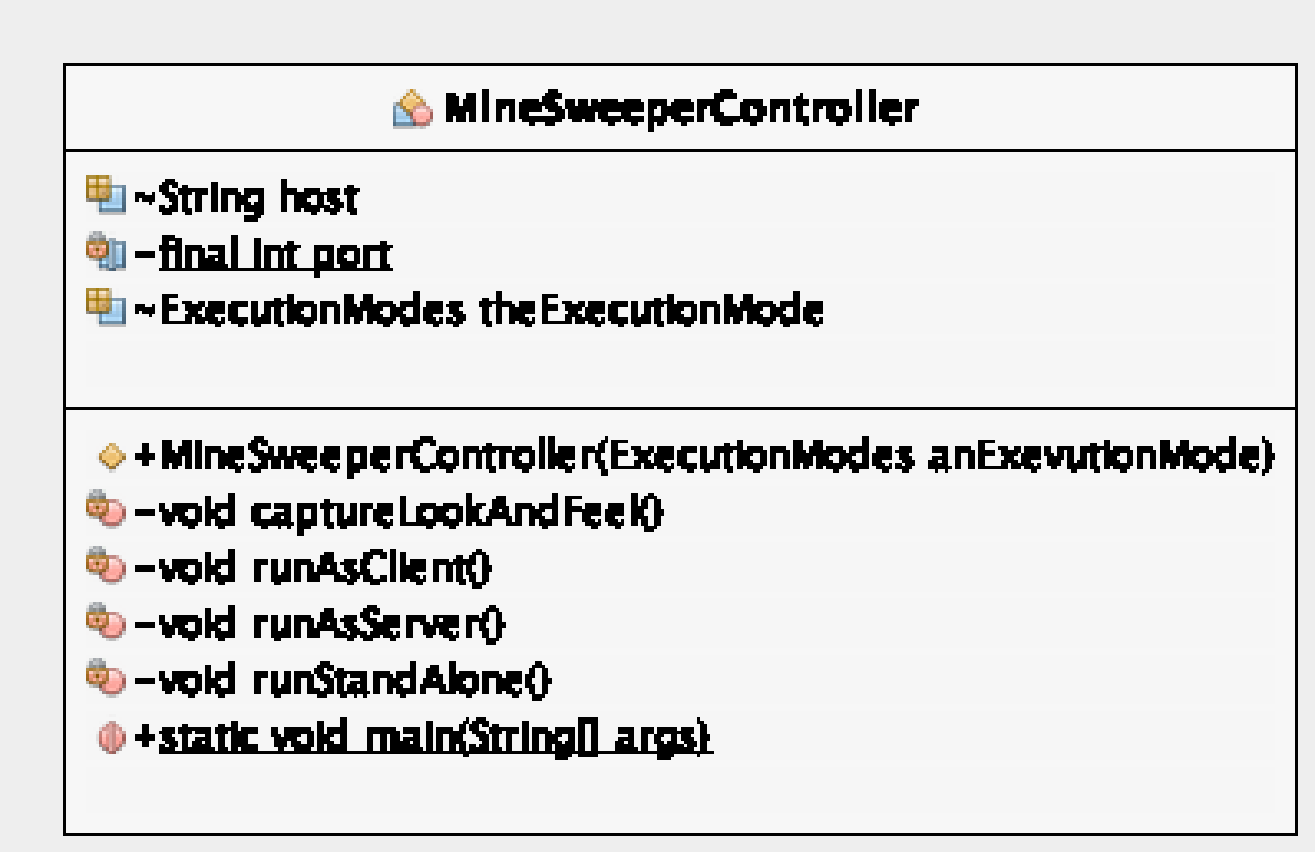


Figure 8: Our `minesweeper_controller` package has only one class. The class `MineSweeperController`.

Illustration of Objective 3

The static structure of of software can be illustrated with a class diagram. Although we were using netbeans 12 in the later stages of the project, we used netbeans 8.1 and the plug in `easyUML` to reverse engineer our code a lift the class diagram of each package. Figure 8 shows that the *Controller* in our MCV pattern is made of a single class. And in fact quite a lean class. The class has few methods and is simple because we are profiting form the vent-driven programming framework of `java swing`. We will describe the static structure of our implementation (the `java` code) using class diagram from the Unified Modeling[1] Language (UML) version 1.4 lifted with `easyUML`. With class diagrams we can see classes (or classifiers in the language of UML 2). The class diagrams will allows to observe structural features (attributes). These attributes are the "knows" responsibility of this class. The focus of attributes is mainly what the objects of these class know (where information is stored). Specific values of these attributes will imply the state of the corresponding object. Similarly, the operation in the class diagram show behavioural features (operations). The set of operations define the *do* responsibilities: what objects of the class can perform. Using `java` and `easyUML` we also observe encapsulation and information hiding as attributes and operations are tagged as private (-) or public(+).

These diagrams describe the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among them. For instance, the class diagram for the package `minesweeper_view` works around the class `MineSweeperView`. The package is large and we show it in full in Figure 9. The details of classes are hard to observe. Nevertheless, it is obvious that the class `MineSweeperView` has a large number of attributes because most of the visual GUI widgets are attached to it. While the `PlayAreaView` class adds complexity because this is the area where the game is played. A closer view of the details of this class diagram is first offered in Figure 10. Using Figure 10 we see the relationships between the class `PlayAreaView` and the class `MineSweeperView`. Although `MineSweeperView` is a container for `PlayAreaView`, we maintain a shared reference from `PlayAreaView` to `MineSweeperView` so aspects of the `MineSweeperView` can be updated as a result of interaction with the game. For instance, when the user correctly sets a flag in a mine, the display widgets in `MineSweeperView` must be updated and show one less mine is active. The relationship from `MineSweeperView` to `PlayAreaView` is the instantiation of one object of the class `PlayAreaView` to be the `JPanel` for the area where the environment is played and the game dynamics are shown.

Figure 11 shows an example of this scenario. Figure 11 is the code of the methods `setFlagWithClick` in

[1]I preserve the American spelling as this is the official name.

Figure 9: Full class diagram of the `minesweeper_view` package.

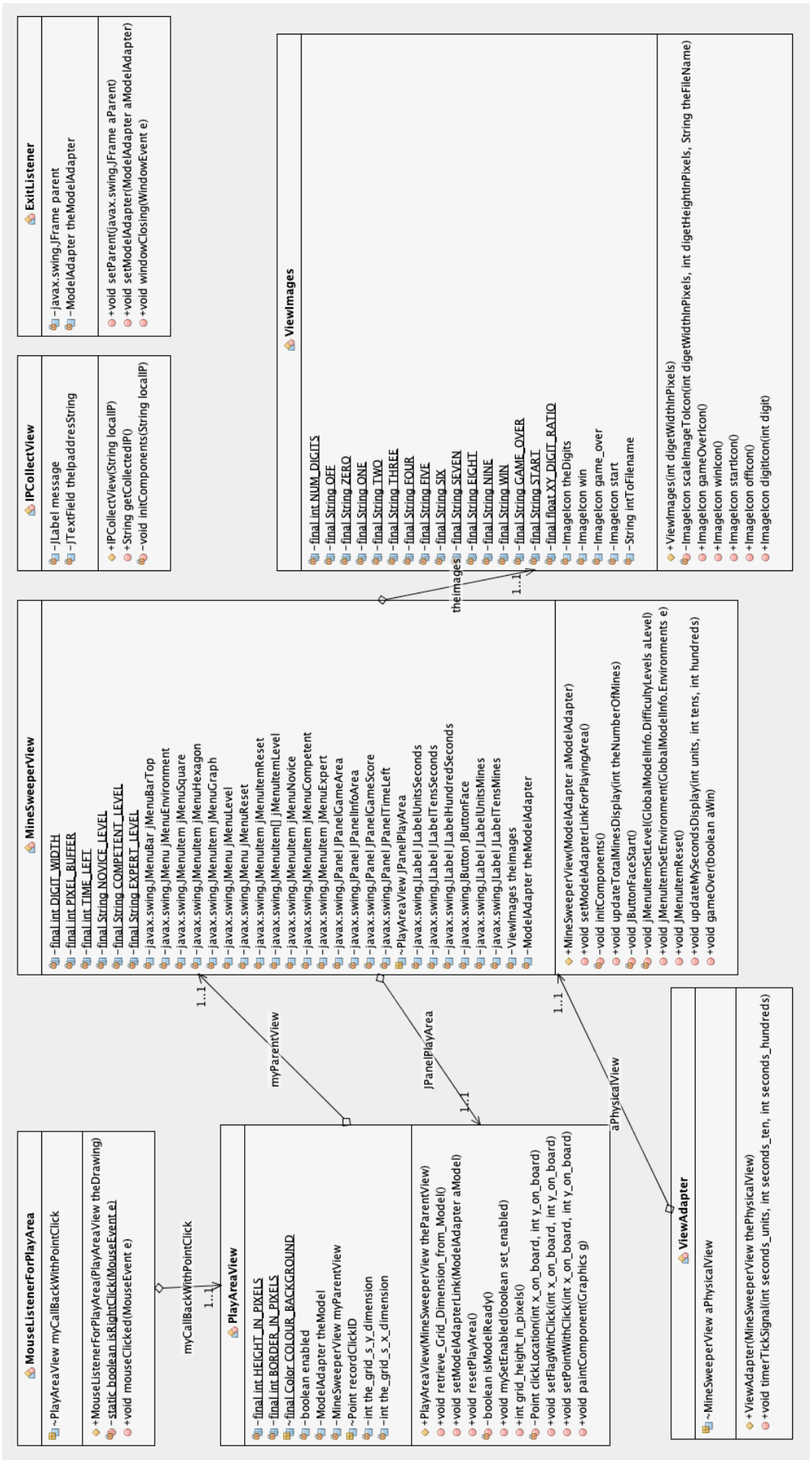


Figure 10: Section of the class diagram of the `minesweeper_view` package. This is the left part of Figure 9.

```
/**
 * A call back from the event listener, will notify the model the user's attempt
 * to flag a cell.
 * Guarded by the our enabled condition.
 * Repaints the environment to reflect changes.
 * @param x_on_board cell coordinate horizontally left to right
 * @param y_on_board cell coordinate vertically top to bottom
 */
public void setFlagWithClick(int x_on_board, int y_on_board) {
    if (enabled) {
    Point aClickLocation = clickLocation(x_on_board,  y_on_board);
    //System.out.println("A Right ckic at X:"+aClickLocation.x+" Y:"+aClickLocation.y);
    try {
        theModel.setFlag(aClickLocation);
        if (  ( GlobalModelInfo.Environments.SQUARE == theModel.getEnvironment() )
               ||  ( GlobalModelInfo.Environments.HEXAGONAL == theModel.getEnvironment() )
              )     {  // analyze correct placement of a flag in the last mine for HEXAGONAL AND SQUARE
                        if (-2==theModel.getValue(aClickLocation) )
                            {
                                // the user signals a mine correctly
                                myParentView.updateTotalMinesDisplay(theModel.get_current_Total_num_mines());
                                   if (0== theModel.get_current_Total_num_mines()) {
                                       myParentView.gameOver(true);
                                   }
                            }
                    } else { // for TOPOLOGICAL
                                myParentView.updateTotalMinesDisplay(theModel.get_current_Total_num_mines());
                                if (   (0== theModel.get_current_Total_num_mines())
                                     || theModel.allUncovered() )
                                {
                                        myParentView.gameOver(true);
                                }
                    }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this, "Lost conenction to Server", "Connection  error",JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }
    repaint();
    }
}
```

Figure 11: Code of the methods `setFlagWithClick` in the class `PlayAreaView` where the reference `myParentView` is to the only object of the class `MineSweeperView`.

the class `PlayAreaView`. We see the line of code

```
myParentView.updateTotalMinesDisplay(theModel.get_current_Total_num_mines());
```

When the user sets a flag correctly in a mine, the object of class `PlayAreaView` must use its reference to an object of the class `ModelAdapter` to retrieve the new number of active mines

```
theModel.get_current_Total_num_mines())
```

and pass this value to is reference of its parent widget (the object `myParentView` and request the widget in the object of class `MineSweeperView` to update the amount of mines displayed to the user. This is the call

```
myParentView.updateTotalMinesDisplay
```

We can illustrate the dynamic behaviour described of these sequence of calls in a sequence diagram. Thus, Figure 12 illustrates the behaviour of the code in Figure 11. In particular, we show that the parent widget of class `JFrame` (the object `MineSweeperView`) creates the object `PlayAreaView` which is a `JPanel` and will draw the environment for the game. The object `PlayAreaView` registers the call-back `setFlagWithClick` that we see in Figure 11 with a `MouseAdapter` object so that the click action on the `PlayAreaView` can provide the event (and object of the class `MouseEvent`. From the event the coordinates of the click action are retrieved and then used as the parameters to the call-back `setFlagWithClick` in Figure 11. The behaviour in the call-back `setFlagWithClick` send the signal by the model adapter to the model of the change required. In the case of a topological environment, the behaviour is slightly different. There is an homogeneity in that the method invocation for obtaining the current number of active mines is the same regardless of the environment in the call-back `setFlagWithClick` in Figure 11. However, determining if the game is over because all mines are exhausted is different. The game in the topological context may be over when all nodes are uncovered (there may be no mines in the novice level). In the grid environments it is impossible to uncover all cells and not either hit a mine or cover them all (although covering many non-mines). The other point we see in the sequence diagram from Figure 12 is the use of the reference `myParentView` from `PlayAreaView` to the tt JFrame `MineSweeperView`. This use is the request to update other widgets with the current value of active mines.

Unfortunately, the tool `easyUML` of Netbeans 8.1 does not show relationships across packages. So, in the example above of the `setFlagWithClick` in the class `PlayAreaView`, the fact that we hold a shared aggregation reference to an object of the class `ModelAdapter` is not show. It is possible to obtain the class diagram of all the

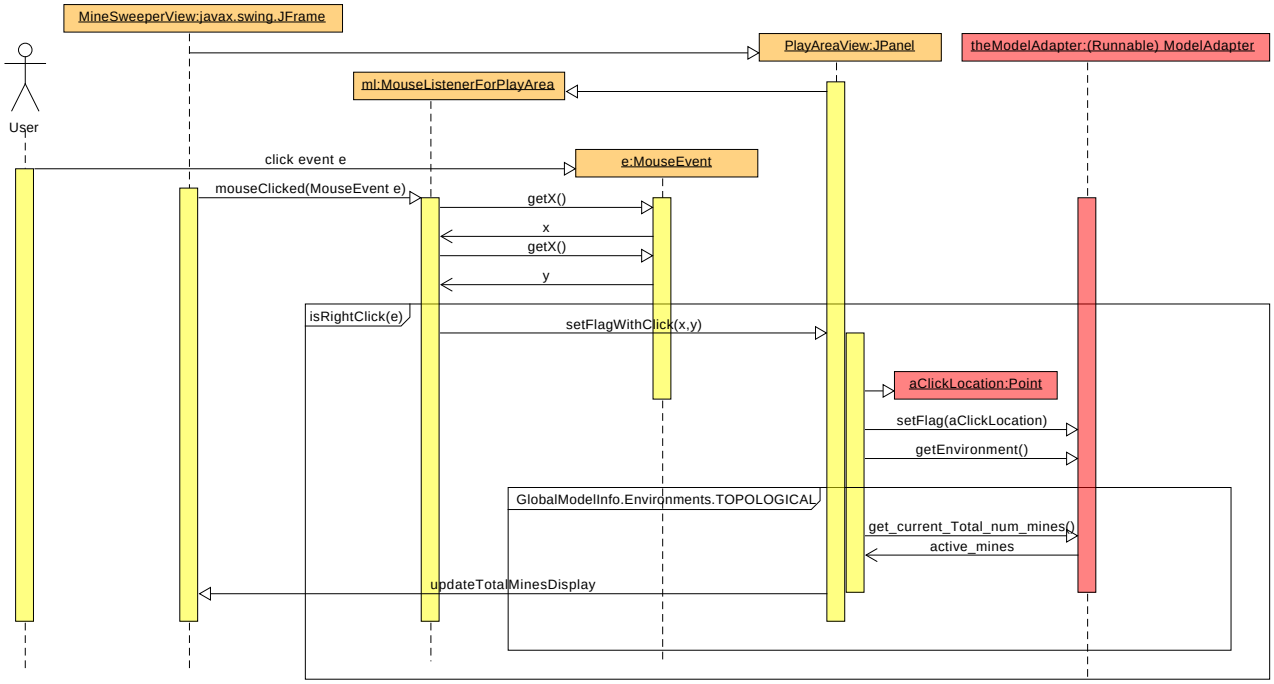


Figure 12: Sequence diagram for the method `setFlagWithClick` in the class `PlayAreaView`. This is a call-back for the event of a mouse click.

classes in the project. Figure 13 is such a structural class diagram which does show the shared aggregation reference to an object of the class `ModelAdapter`. The diagram has too much detail as it includes all attributes and all operations. Nevertheless, this diagram enables us to see some aspects already mentioned before more clearly. The class `MineSweeperModel` is large. This is a potential indication of poor cohesion. In this reflection, we have already mentioned that the model holds all the date (knows about) the environment, even though we have an environment that is wither a square grid, a hexagonal grid or a graph (exclusively). Therefore, the design needs some refactoring, there should be an abstract class with the commonalities, and subclasses specialised for the type of environment and for the different data structures that re required. We mentioned this already, for the regular girds, the retrieval of the current number of active mines during the game is a scan of the `board` data structure. In the topological environment, an explicit integer variable is used for this.

There are other visual aspects that emerge, or are confirmed, by the class diagram of all the classes provided in Figure 13. The `ModelAdapter` is a mirror of the `MineSweeperModel`. It is large because it reflects the model. However, it seems possible to use already applied *Command* software engineering pattern to rely the command from the model adapter to the model. Another aspect is what we mentioned already, the classes in the view have been separated for the different environments (a positive aspect) but not sufficient generalisation has been applied. Inspection of the code shows that there are methods that currently are the same and should be pushed up to a superclass.

Figure 14 is the class diagram for the `minesweeper_model` only. Because we see more detail in the methods and the attributes, we confirm the improvement that would be obtained if the `MineSweeperModel` would have specialisation.

Illustration of Objective 4

The core paradigm for the design is object-oriented design. There are many reasons why object-oriented design is the prevalent design paradigm of software system. The fundamental aspect is that the activity that happens on the software system is shared among performers (doers) categorized in classes. These performers receive the name of objects. All objects of the same class share the same structure and have the same profile of things they know about or operations they can perform.

Thus, the focus has been to design the classes following two major software engineering principles:

1. low coupling and
2. high cohesion.

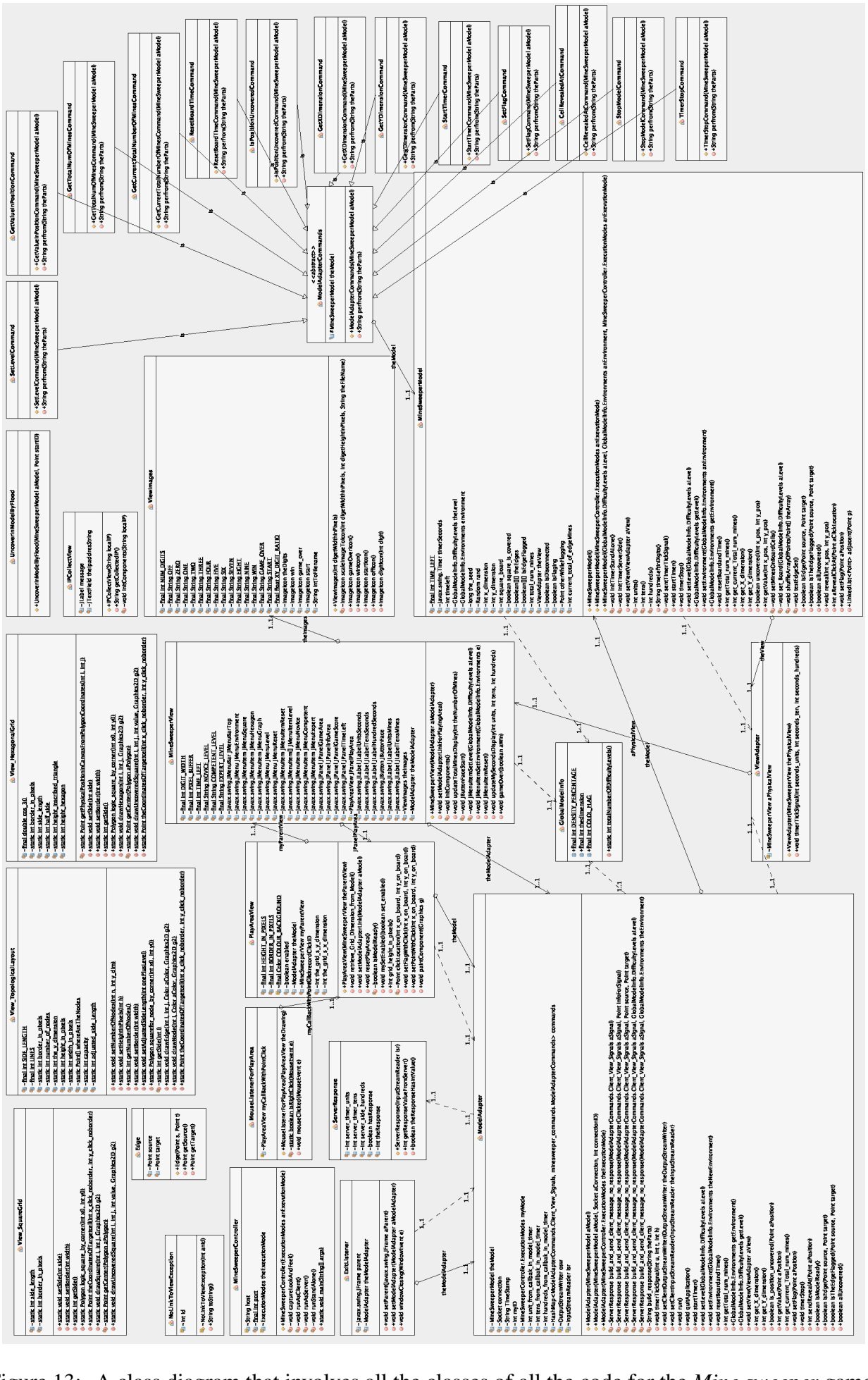


Figure 13: A class diagram that involves all the classes of all the code for the *Mine sweeper* game.

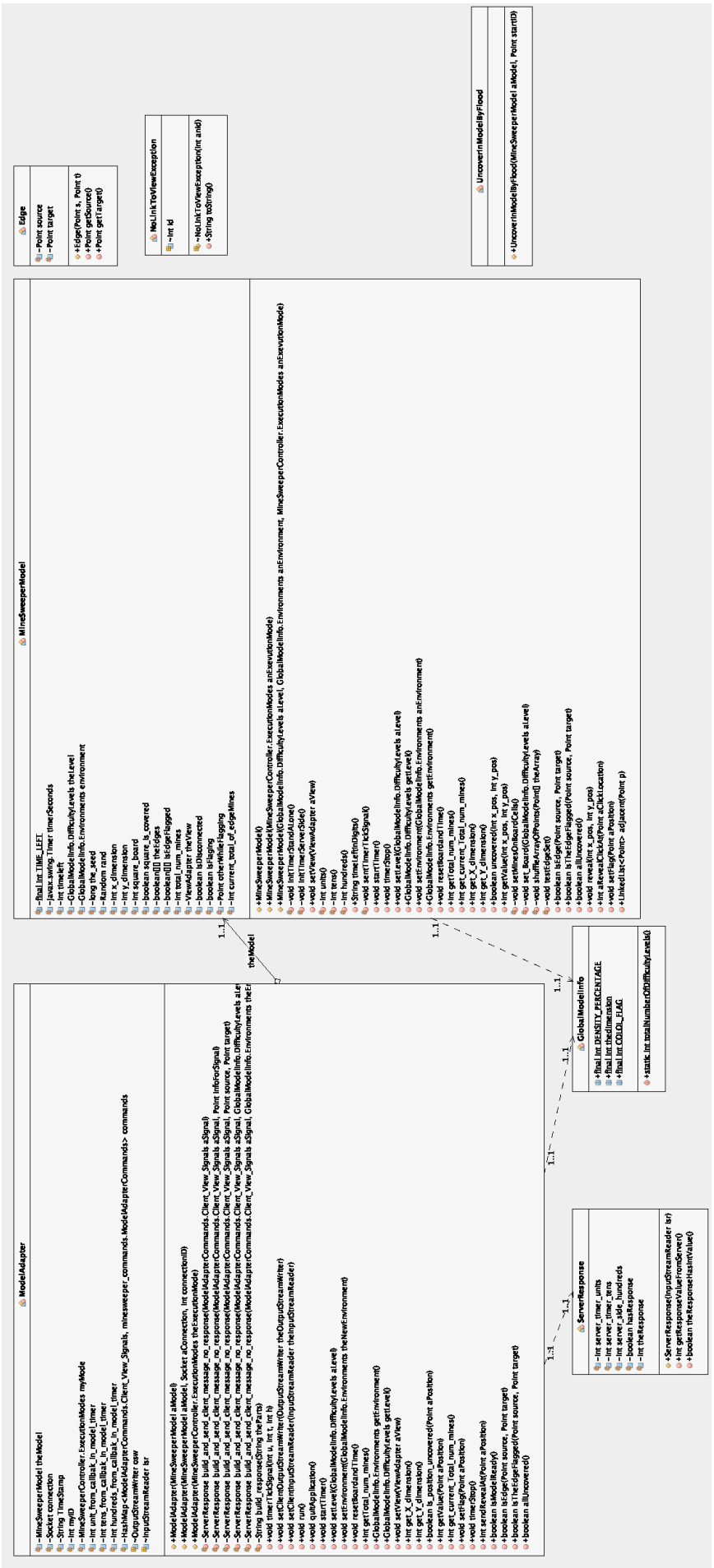


Figure 14: A class diagram that involves only the classes in the `minesweeper_model` package.

One way of follow this principles is to keep the responsibility (the job objects of a class) precise and focus. This is assisted if we apply software engineering patterns. One aspect that such design displays is a significant amount of delegation. We have already discussed some examples of the delegation to the expert who has the required knowledge. The code in the call-back `setFlagWithClick` in Figure 11 illustrates this in a positive way as follows. The code here is oblivious of whether the application runs in client/server mode or in stand-alone mode. The model adapter hides whether the model is remote or local.

However, there are also aspects to criticise, and that is why this report offers a reflection on the positives and the negatives. We already mention that the code in the call-back `setFlagWithClick` in Figure 11 needs to recuperate what type of environment is the game at this stage in order to decide if the game is over and whether the user succeeded with the challenge or they failed. An improved object-oriented design would remove the need (responsibility) from the call-back `setFlagWithClick` in Figure 11 to know the environment and just delegate this in a query to the model. In fact, the responsibility of the model is to know all about the state of the game, including whether is finished.

Another strong paradigm derived from object-oriented programming is the use of event-driven programming. This paradigm to developing software focuses in ensuring the flow of the program is determined by events. In our application the events are mainly

1. user actions (mouse clicks, key presses),

2. time triggered events (a count done of the second of an instance of the game).

Event-driven programming is the dominant paradigm used in graphical user interfaces and for a reactive system. In particular, we expect a few things to be updated as a result of an event. In the case the event in in the GUI, the GUI itself gets updated in precise ways (potentially codified as rules of the form if the vent from the user X the update we need to apply is Y). That is, there are clearly defined set of actions in response to user input. In the case that a second has gone by, the display on the timer needs to be updated to reflect the new value in the count down.

Java Swing facilitates event-driven programming because it set up the software that manages the main loop that listens for events. When one of those events is detected, the framework forwards the events by triggering a call-back function (and providing the event as a parameter). Some of this was illustrated with the call-back `setFlagWithClick` in Figure 11. However, the actual Java Swing framework functionality is outside the scope of this report. We took the framework as is. Some reviews of the Java Swing framework suggest that in itself, the classes in it are classified as a design that follows the model/view/controller software design pattern.

Another design paradigm we had in mid was *Design by Contract* and although `java` is not a programming language that directly support this, we were hoping to construct our design with contracts in mind and reflect failure to meet the contract with appropriate exception handling. Thus, initially we anticipated and designed our own classes for exception (in particular for potentially faulty request for actions on the model). However, the user can hardly produce invalid input in the games due to the constraints by the GUI. Secondly, most of the failures for communication between client/server over sockets in `java` have exception under the class `I/O exception`. Thus, the current design uses almost always the exception of the corresponding sockets classes in `java`. The code and the design would benefit from a review and re-factor from a design by contract perspective, specially for the current exception handling. In particular, the current implementation could be improved and be more resilient about the suer miss-typing an incorrect IP address for the server.

Illustration of Objective 5

The main software architecture is a `java` client/server architecture over plain sockets in `java`. However, the application can be launched as stand alone with suitable command line argument (similarly, whether is launches as a client or as the server is determined by a command line argument). As a client/server application, it works as distributed application. The responsibilities are split and partition. The server holds the model (all the information about the status of an instance of the game). The client hold the view (all the information about the GUI and its presentation, as well as capturing the events from the user). The separation or integration into stand alone is the responsibility of a `ModelAdapter` class. This class hides from the view whether the application is running as stand-alone of as a client-server. This information hiding (see in the literature the software engineering of information hiding) enables the separate design, testing and operation of the view. There is much more modularisation, and as we saw in the early discussion above regarding the presented class diagrams, the view classes are organised in a package.

The partition into the possibility of several clients and server is to address the requirement of several players simultaneously competing against the same challenge in the same environment. Client/server architectures are usually deployed with many other objectives in mind:

1. resilience to faults in one or the other as even when residing in the same computer, a failure of one does not imply the failure of the other

2. load balancing, since a client does not share any of its resources, but it requests content or service from a server.

Since often clients and servers communicate over a computer network on separate hardware, several players can launch their corresponding client and joining a simultaneous game. Using standard sockets and `java` would require minimal installation effort for the clients regardless of platform. This architectural and platform choices facilitates the execution in requirement that the application be operational in different operating systems.

We note that the partition of the view on the client and the model on a server is a common software engineering pattern usually named 2-tier architecture. In fact, Wikipedia offers the same entry for client-server model and for 2-tier architecture.

As a reflection we see the following potential improvements on the current architecture. We chose simple socket connection. That is e use the `java` class `import java.net.Socket`. This has some cyber-security implications if we enable connections to the computer running the server from clients in remote computers. The corresponding port would be open. This complicates the installation on the server machine and perhaps demands setting up firewalls and registering clients. On the other hand, the strings (the content) of the messages is in the clear. So, it is perfectly feasible to eavesdrop on the interactions between the clients and the server by a third party. An improvement could be to use a secure socket layer. That is, to re-factor the code to use `javax.net.ssl.*`

Another alternative is to migrate the application to a Web architecture over `http`. Typically there are well documented guidelines how to set a web-server securely, specially because the standard port is well-known. Following those guidelines it is easier to set the serve properly and also the corresponding firewall since gain the port is one for common use for this purpose. However, many of the classes here would need redesign. It is even very unclear if any of the code could be preserved in servlet and what of the GUI code could be preserved even if it migrates to applets.

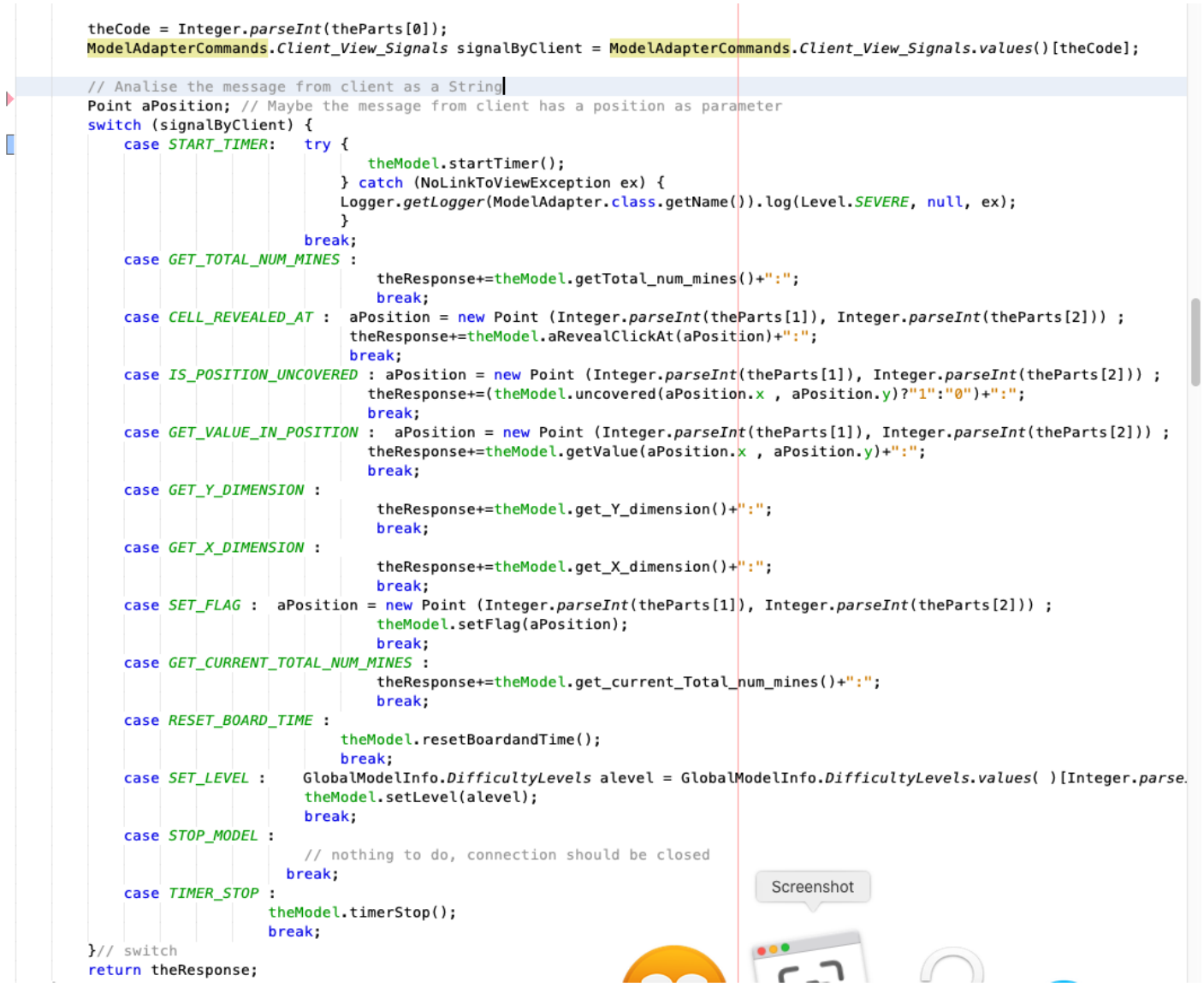
The architecture of several clients running a GUI and one server running the model has many similarities with the software architecture pattern of a repository. There are some important aspects to consider. The current architecture clones a model in the server for each connection to a client (a player). And only one player can set up a game and tear it down. This choice implies that all the issues of concurrency typically in central-repository architectures disappear. This is because in our implementation, the clients do not share any data on the status of the game. As we indicated, they all play simultaneously in their own copy.

The current implementation does not record the score of players. This part of the project remains incomplete. Nevertheless, the plan was to enable persistence in the server using `Derby` and not even with a client-server connection to the database, but enabling the database using the embedded version of `derby` (db.apache.org/derby/) This would keep the software architecture as a 2-tier client-server. That is, simpler that the alternative of a 3-tier version i.e. the database as 3rd layer).

Illustration of Objective 6

In the model solution for Milestone 2 we showed the application of at least two software patterns.

1. **Adapter**. This pattern is also known as *wrapper* and it has been documented as another name for the pattern *decorator*. We apply it to give a generic interface to our model, and hide from the front-end of our software (the view and the controller), that the model may be remote (in a server) or local (when or software runs stand alone). An illustration in this submission of this pattern is in the code of the call-back `setFlagWithClick` in Figure 11. All the references and invocations to the object `theModel` are to an object of the class `ModelAdapter`. In this way, the objects on the *View* side (such as the object of the class `PlayAreaView` which has this call-back) are oblivious (they code is agnostic as to whether the application is running stand alone or the model is remote). This `ModelAdapter` presents to our objects in the *view* the interface of an existing class (the model).



```
theCode = Integer.parseInt(theParts[0]);
ModelAdapterCommands.Client_View_Signals signalByClient = ModelAdapterCommands.Client_View_Signals.values()[theCode];

// Analise the message from client as a String
Point aPosition; // Maybe the message from client has a position as parameter
switch (signalByClient) {
    case START_TIMER:   try {
                            theModel.startTimer();
                        } catch (NoLinkToViewException ex) {
                        Logger.getLogger(ModelAdapter.class.getName()).log(Level.SEVERE, null, ex);
                        }
                    break;
    case GET_TOTAL_NUM_MINES :
                             theResponse+=theModel.getTotal_num_mines()+":";
                             break;
    case CELL_REVEALED_AT :  aPosition = new Point (Integer.parseInt(theParts[1]), Integer.parseInt(theParts[2])) ;
                            theResponse+=theModel.aRevealClickAt(aPosition)+":";
                            break;
    case IS_POSITION_UNCOVERED : aPosition = new Point (Integer.parseInt(theParts[1]), Integer.parseInt(theParts[2])) ;
                           theResponse+=(theModel.uncovered(aPosition.x , aPosition.y)?"1":"0")+":";
                           break;
    case GET_VALUE_IN_POSITION :  aPosition = new Point (Integer.parseInt(theParts[1]), Integer.parseInt(theParts[2])) ;
                           theResponse+=theModel.getValue(aPosition.x , aPosition.y)+":";
                           break;
    case GET_Y_DIMENSION :
                             theResponse+=theModel.get_Y_dimension()+":";
                             break;
    case GET_X_DIMENSION :
                             theResponse+=theModel.get_X_dimension()+":";
                             break;
    case SET_FLAG :  aPosition = new Point (Integer.parseInt(theParts[1]), Integer.parseInt(theParts[2])) ;
                             theModel.setFlag(aPosition);
                             break;
    case GET_CURRENT_TOTAL_NUM_MINES :
                             theResponse+=theModel.get_current_Total_num_mines()+":";
                             break;
    case RESET_BOARD_TIME :
                        theModel.resetBoardandTime();
                        break;
    case SET_LEVEL :    GlobalModelInfo.DifficultyLevels alevel = GlobalModelInfo.DifficultyLevels.values( )[Integer.parse.
                        theModel.setLevel(alevel);
                        break;
    case STOP_MODEL :
                        // nothing to do, connection should be closed
                      break;
    case TIMER_STOP :
                    theModel.timerStop();
                    break;
}// switch
return theResponse;
```


Figure 15: Most of the code of the methods `build_response` in the class `ModelAdapter` before the application of the command pattern.

2. **Model-View-Controller**: We separate the program logic into 3 packages. The packages have classes with 3 different responsibilities. The model is responsible for managing the information and data that represent the sate of the application (the current state of a game). The view has all the presentation information. As we mention, for the graph environment, a map is required from the position in pixels of a node, to its identifier (a pair of integers). That is the view has information of where nodes are drawn. That information does not exist in the model. The model has information about the colour of nodes and a map from pairs of nodes to whether this is an edge that exists or not. Our controller is just the set up infrastructure as the event-driven facilities of `java swing` enabling the identification of the action listener that triggers the communication from the view to the model. This is why our controller is so thin. The response to the user input and impact on the model is handled by the event-driven programming.

In this Milestone 3 reflective report, we show the implementation in our software of the software engineering pattern `Command`. Figure 15 shows a snippet of the code at the stage of development of Milestone 2. We can see the code in lengthly and error prone when one extends this to more commands. In fact several more commands were required when adding the graph environment. Figure 16 shows the enumerated type for the commands. This enumerated type is used to flatten the signals between clients and the server for socket communication. But also in the command pattern, they are used as the identifier of the command in a hash-map when implementing the *Command* software engineering patter, When a new command is required, it subclasses from the class `ModelAdapterCommands`. This sub-classing can be observed in the right-hand side of Figure 13. An example of the code for a particular command is shown in Figure 17. A specific command just needs to implement the method

```
public String perfrom(String theParts[])
```

```
public abstract class ModelAdapterCommands {

     /**
    * The enum that defines the signals the view can send in client server mode
    */
    public enum Client_View_Signals {
        START_TIMER(0),
        GET_TOTAL_NUM_MINES(1),
        CELL_REVEALED_AT(2),
        IS_POSITION_UNCOVERED(3),
        GET_VALUE_IN_POSITION(4),
        GET_Y_DIMENSION(5),
        GET_X_DIMENSION(6),
        SET_FLAG(7),
        GET_CURRENT_TOTAL_NUM_MINES(8),
        RESET_BOARD_TIME(9),
        SET_LEVEL(10),
        STOP_MODEL(11),
        TIMER_STOP(12),
        SET_ENVIRONMENT(13),
        GET_ENVIRONMENT(14),
        IS_EDGE(15),
        GET_LEVEL(16),
        IS_EDGE_FLAGGED(17),
        ALL_UNCOVERED(18);
        public int value;
        private Client_View_Signals(int value) {
                this.value = value;
        }
    }
```

Figure 16: The enumerated type that lists all the signals from the client side to the server side. This is part of the abstract class `ModelAdapterCommands`.

```
public class StartTimerCommand extends ModelAdapterCommands {
    public StartTimerCommand (MineSweeperModel aModel)
    {
        super(aModel);
    }

    public  String perfrom(String theParts[])
    {   System.out.println(StartTimerCommand.class.getName());
        try {
            theModel.startTimer();
        } catch (NoLinkToViewException ex) {
            Logger.getLogger(StartTimerCommand.class.getName()).log(Level.SEVERE, null, ex);
        }
         return "";
    }
}
```

Figure 17: A specific command derives from the abstract class `ModelAdapterCommands`.

```
/**
 * Server side response to query_command from client
 * @param process the query_command from client
 * @return
 */
private String build_response(String theParts[]) {
    String theResponse;
    // Indicate what time is in the model
    theResponse=theModel.timeLeftInDigits();

    int theCode = Integer.parseInt(theParts[0]);
    ModelAdapterCommands.Client_View_Signals signalByClient = ModelAdapterCommands.Client_View_Signals.values()[theCode];
    ModelAdapterCommands theCommand = (ModelAdapterCommands) commands.get(signalByClient);
    if (theCommand !=null) theResponse+=theCommand.perfrom(theParts);
    return theResponse;
}
```

Figure 18: The new method `build_response` of the `ModelAdapter` (after applying the *Command* pattern) and derive from the abstract class `ModelAdapterCommands`. This is very similar to the illustration of the pattern *Command* in Workshop 7.

The constructor of the `ModelAdapter` just creates the command and registers it as a possible command it may receive in a socket (if fact, we receive the enum).

```
minesweeper_commands.StartTimerCommand aStartTimerCommand = new minesweeper_commands.StartTimerCommand(aModel);
commands.put(ModelAdapterCommands.Client_View_Signals.START_TIMER,aStartTimerCommand);
```

The new method `build_response` of the `ModelAdapter` (after applying the *Command* pattern) is shown of Figure 18. We shall contrast Figure 18 where now all the work is performed in seven (7) lines of generic code with Figure 15. In particular, the code of the earlier version of `build_response` did not fir in the screen capture. Now the generic code handles even command for the following new signals.

1. SET_ENVIRONMENT(13),
2. GET_ENVIRONMENT(14),
3. IS_EDGE(15),
4. GET_LEVEL(16),
5. IS_EDGE_FLAGGED(17) and
6. ALL_UNCOVERED(18).

Illustration of Objective 7

The description of our architecture mentioned that for GUI-building we use `Java Swing`. This is an event-driven programming framework. It is rather old and meant to be replaced by Java-FX. However, under java 8 and netbeans 8.1, it even offers a GUI builder. That is, one can design/build the GUI using a drag/drop approach to the canvas and bring widgets. Java Swing is a rather lightweight Java graphical user interface (GUI) widget toolkit. Its is platform independent because itself is written in `java`. It includes a rich set of widgets. Because it is part of the Java Foundation Classes (JFC) is extremely standard and reliable. It also includes several packages for developing rich desktop applications. Java Swing has built-in controls such as trees, image buttons, tabbed panes, sliders, tool-bars, colour choosers, tables, and text areas.

Using `Java Swing` encourages event-driven programming. One declares the structure of the widgets in a containment relation that is closely correlated with the layout in the GUI. One can set properties of widgets before they become visible. One can then select what sort of events the widget should respond to, and indicate which call-back function is to be enacted as a result of the corresponding event. The event is one of the main parameters received by the call-back, but even properties of the widget can be extracted (some state or property value).

The GUI for the *Mine sweeper* environment of a square grid could be easily configured by buttons. However, there is no widget that enables hexagonal shapes, so the hexagonal-grid must be drawn. Similarly for the graph-grid. The drawing of graphs with n nodes by placing each node in a vertex of a regular n-gon ensures that (in theory) all edges are visible, no edge coincides with a segment of another, and all can be drawn as straight lines. But this simplistic embedding of a graph in the plane creates many crossings and does not scale. There are many packages in `java` for graph-drawing but we chose to keep the layout algorithm simple.

```java
/**
 * Test of setView method, of class MineSweeperModel.
 */
@Test
public void testSetView() {
    System.out.println("setView");
    MineSweeperModel instance = new MineSweeperModel();
    ModelAdapter aNullModelAdapter = null;
    MineSweeperView aPhysicalView = new MineSweeperView(aNullModelAdapter);
    ViewAdapter aView = new ViewAdapter(aPhysicalView);
    instance.setView(aView);
    try {
        instance.startTimer();
    } catch (NoLinkToViewException ex) {
        Logger.getLogger(MineSweeperModelTest.class.getName()).log(Level.SEVERE, null, ex);
    }
    try {
        Thread.sleep(3000);
    } catch (InterruptedException ex) {
        Logger.getLogger(MineSweeperModelTest.class.getName()).log(Level.SEVERE, null, ex);
    }
    instance.timerStop();
    String expResult = "2:0:1:";
    String result = instance.timeLeftInDigits();
    assertNotEquals(expResult, result);
    // another 3 seconds the timmer should have stop counting down
     try {
        Thread.sleep(3000);
    } catch (InterruptedException ex) {
        Logger.getLogger(MineSweeperModelTest.class.getName()).log(Level.SEVERE, null, ex);
    }
     String secondResult = instance.timeLeftInDigits();
     assertEquals(result, secondResult);
}
```

Figure 19: The code of a unit test that launches the GUI in order to carry some behaviour regarding the timer..

Figure 12 shows a section of the process to set up a widget. This is not the complete story. However, describing in detail the dynamics of `java Swing` is for tutorials on the framework. Suffice to say we have illustrated a call-back in in Figure 11 and the demo video shows the comamnd of the framework to use it effectively for the tasks of this milestone.

We just want to add one point here. It is possible to use `JUnit` and set tests with a GUI. Figure 19 shows a unit-test for the setting up of the GUI, setting the initial value of the timer, observe it displayed, start the timer and count down, and verify the time is, after some seconds a different value. Launching the unit testing results in the GUI becoming visible for about 3 seconds and then disapearing, and in this case, the test is passed as discussed in the earlier paragraphs on testing.

Illustration of Objective 8

We have discussed Model-View-Controller (MVC) in Milestone 2 and earlier here when we discussed software engineering patterns. MVC is probably one of the oldest and most used software patterns since what is presented to the user is precisely that, a visual representation of potentially far more information that is in the state of the application (he model). Out software was split since Milestone 2 into 3 packages to adhere to the guidelines of MVC. This is crucial to separate responsibilities

1. what and how to show, is the job of the view
2. what to change, what information to record until the next event or interaction is the job of the model,
3. how to synchronise and communicate the view and the model, when to let the view inspect if the model has changed, and when to notify the view if the model has changed are variants of a pull versus push controller.

MCV is such an important pattern that it is considered to be also a software architecture pattern In a sense, in our code it appears twice because the internal design details of the `java Swing` framework are also MVC.

Table 1: Lines of code of the files in the package `minesweeper_view`.

Lines of code	class/file
67	ExitListener.java
58	IPCollectView.java
438	MineSweeperView.java
46	MouseListenerForPlayArea.java
309	PlayAreaView.java
29	ViewAdapter.java
98	ViewImages.java
208	View_HexagonalGrid.java
147	View_SquareGrid.java
148	View_TopologicalLayout.java
1,548	total

Table 2: Lines of code of the files in the package `minesweeper_model`.

45	Edge.java
87	GlobalModelInfo.java
698	MineSweeperModel.java
663	ModelAdapter.java
22	NoLinkToViewException.java
46	ServerResponse.java
59	UncoverInModelByFlood.java
1,620	total

As explained earlier (both Milestone 2 and here), using MVC also enabled the mapping into a client/server architecture. When players are competing in the same environment and same challenge, the controller and the view are in separate clients (many), and possibly distributed. We have a single server, but in the server we clone the model, so the players play using the same environment and challenge, but only difference is who completes the correct labelling first or with a superior score. Thus, the use of MVC also enabled the distribution of the application and the competitive mode requirement.

Illustration of Objective 9

On the quality of the design

There are several aspect that correlate with good software design. Some of these are highlighted by software metrics. The following two tables show the total number of lines in the corresponding files (for the two large packages in our project). This is a raw metric because comment sin the code are included. However, they will be illustrative of several aspects. In Table 1, we see that we have 9 classes and an average of less than 200 LOC per class. The average is slightly above 150 lines of code. This suggest the classes are small, comprehensible/understandable. Their job is specific, unlikely to have lost cohesion. The class diagrams of the package as lifted (re-engineered) by `easyUML` shows only 5 associations, the class diagram (refer to Figure 9) can be drawn as a planar graph (no edge crossings). Such a low number of associations suggest very low coupling. These are indicators of a high quality design: small classes and low number of dependencies.

If we examine the `minesweeper_model` package we observe the values for LOC as per Table 2. Here we observe two outlier classes that are above 500 lines of code. We already indicated that the class `MineSweeperModel.ja` seems to have a low standard of cohesion. It is handling relatively different information and operations for relative different duties. This is because it holds all the information regardless as to whether the environment is a square-grid, a hexagonal-grid or a graph-layout. This confirm the already mentioned suggestion that the common elements among the environments from the perspective of the model should be lifted to a more general abstract class.

The current object-oriented design takes advantages of several patterns and technologies. It enables cross platform execution and there is a partition of labour among the classes that matches precise functionality (high cohesion) for the vast majority of the classes. We can conclude that there is room for improvement, but the current design can support enhancements without large disruptions to its current structure. The code is very light.

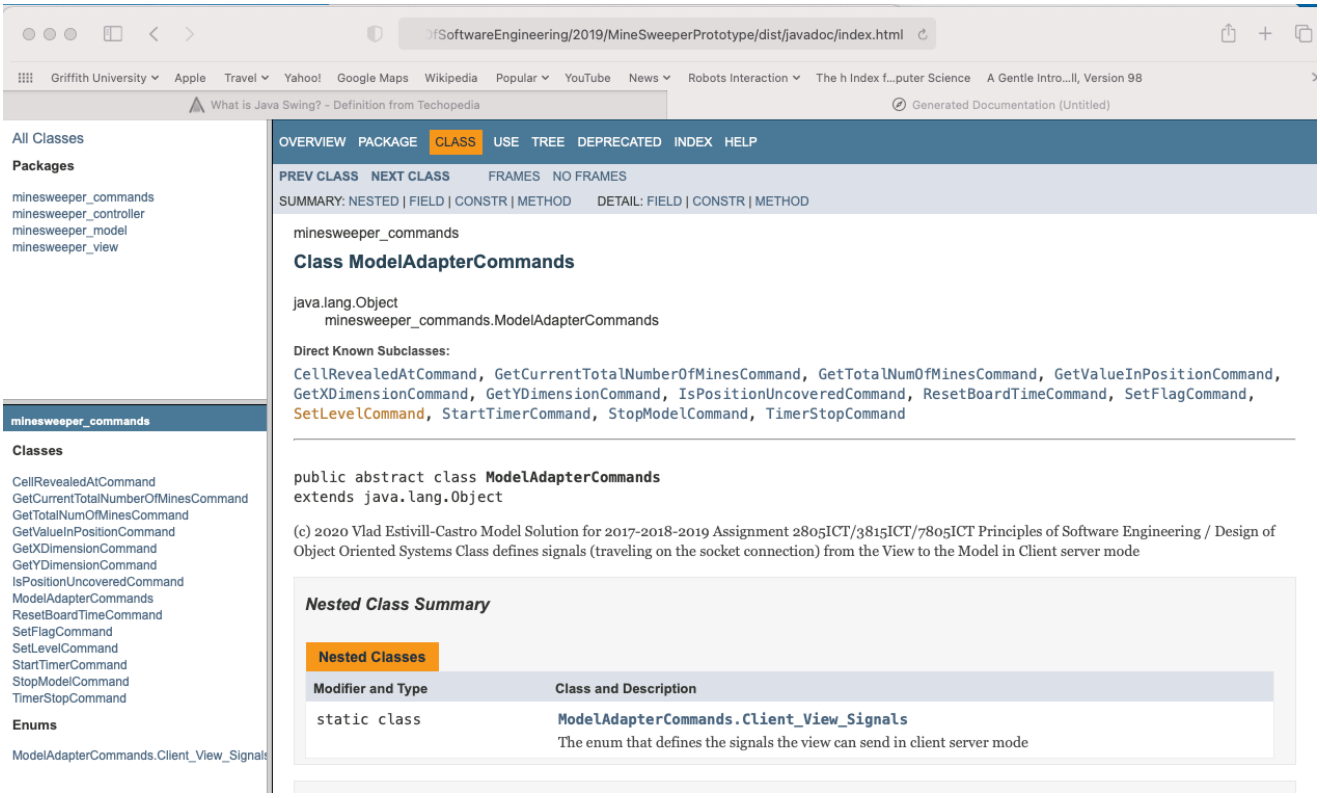


Figure 20: The top part of the page of the automatically generated documentation for the class `ModelAdapterCommands`.

The only aspect that is potentially an issue is the performance of the message passing over sockets in `java` as our own `Sting` messages. There are also `java` libraries to perform remote procedure calls or to communicate objects. We already observed that there is some lag in the current `Sting` sending, receiving and parsing. Potentially this could be a bottle neck to scalability. That is, the system appear slow in client/server mode. The user may find it not sufficiently snappy and reactive. Nevertheless, the overall design is robust in that improving this aspect affects only the `ModelAdapter`.

On the relevance of automatically generated documentation

On the other, automatically generated documentation is a reliable method to ensure the documentation reflects the behaviour in the code. It is important also to reflect on the responsibility of the method, describing the input and outputs to an individual operation. This is extended to describe the responsibilities of a class. Documentation is important because it assists in the continuous development because other developers can gain an understanding of a class functionality by looking at the in-code comments or the associated generated documentation. It also assists the developer to reflect on the design. The documentation must stipulate the and summarise the responsibility of a class. It suggest to argue for the inclusion and relevance of each operation relative to the main function/responsibility of the class. Figure 20 shows the top area of the automatic generated documentation using `Javadoc` in netbeans for the abstract class in the *Command* pattern.

Illustration of Objective 10

With our submission for Milestone 1, Milestone 2 and Milestone 3 we have demonstrated we can illustrate the structure of the design as we included several class diagrams at different level of detail.

In Milestone 1 we included Use-Case Diagram to illustrate the boundary of the system and the use-cases (the functionality) of the system. This earlier use-case diagram is reproduced in Figure 21. A quick review of the use-case diagram can be a rapid inspection for acceptance criteria. We see that all use-cases are completed except the use-case that enables users to save their score.

Dynamic behaviour has been illustrated with sequence diagrams, activity diagrams, state-charts and also communication diagrams. Figure 22 shows an activity diagram for explaining the behaviour on cells from grid. This activity diagram is insufficient to explain how edges with a mine are flagged by the user, since in this case the user must flag one node first and then the other endpoint of the edge in immediate succession. Any other click or action reverts the state away from flagging and edge and the first node marked is not uncovered and its colour revealed.

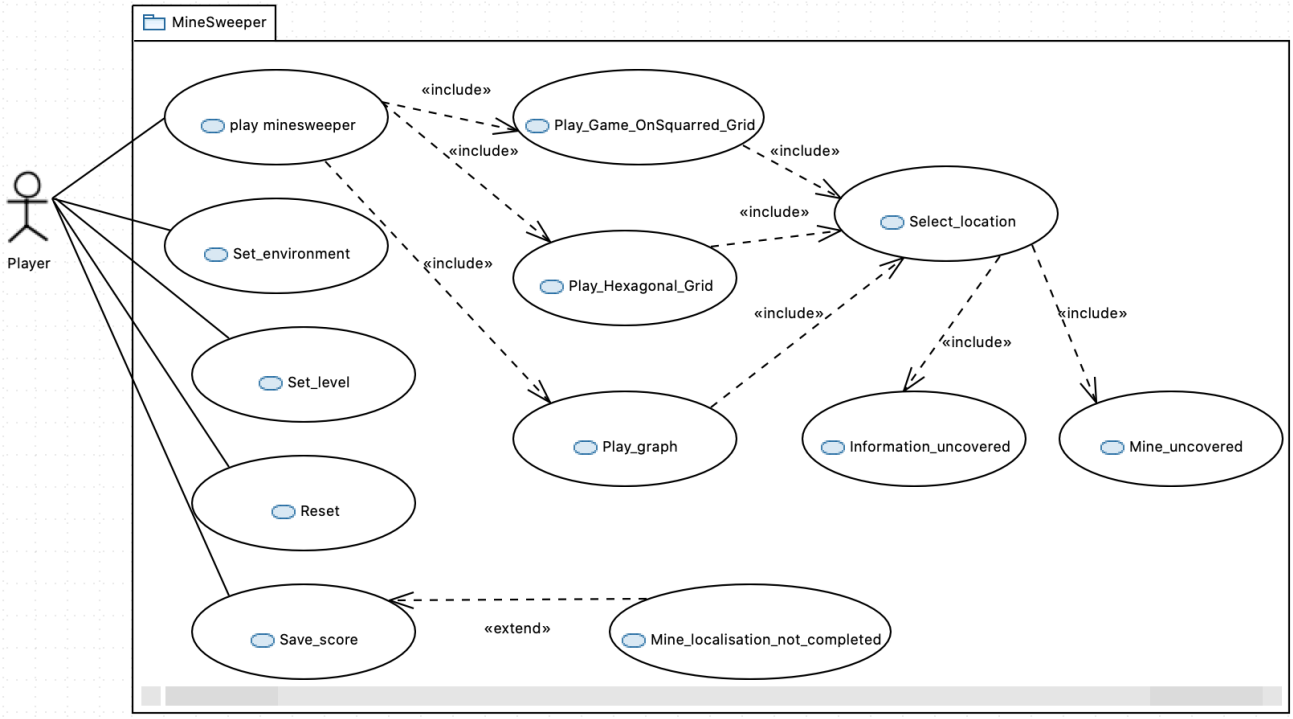


Figure 21: The use-case diagram from Milestone 1.

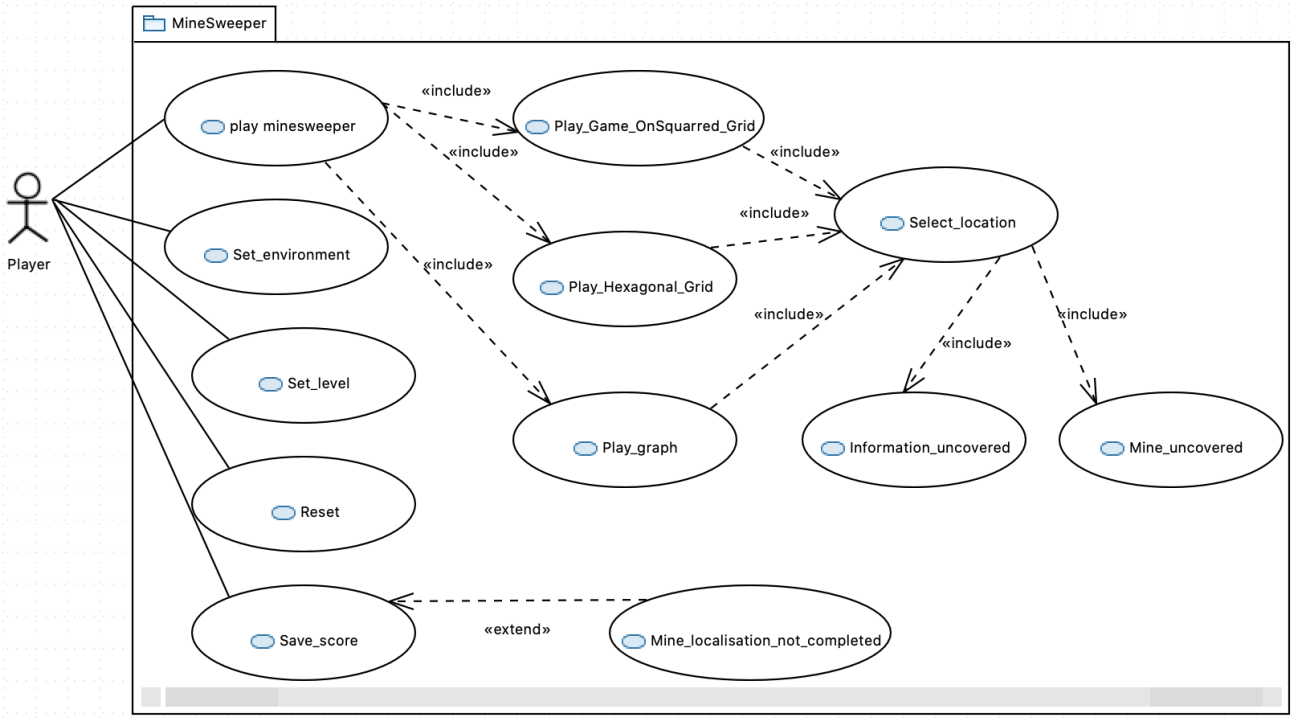


Figure 22: The activity diagram from Milestone 1.

In this submission, all UML diagrams are static diagrams (class diagrams) except for the dynamic modelling illustrated by the sequence diagram of Figure 12.

Our discussion here has shows that we can use these models (the diagrams) to reflect on properties of the design, such as the quality of achieving low cohesion and high coupling.