Luleå University of Technology
Computer Science
Håkan Jonsson

January 25, 2017
D0010E

Lab 2

– Reaching New Levels –

1 Introduction

This assignment assumes you have completed Lab Assignment 0, the *tutorial*, where you learned about basic Java and started to use the programming environment *Eclipse* to create programs. In addition, we will need material introduced in the first few interactive video lectures in the course. You work on the assignment together with another student (your lab partner). The assignment is divided into tasks that should be completed in order.

When every task is completed, you and your partner should present your work to a lab assistant to have it graded. You do this together, and pass or fail as a group, so make sure you both know everything essential about your work.

This grading normally takes place in the lab and with the use of a lab computer. Your Java programs should be polished and written in such a way they are easy to read and understand. You are expected to answer questions about your work and reason about what you have achieved. Upon request, your are also expected to present and demonstrate your programs in *Eclipse* to, for instance, support claims of correctness. Prepare well before the grading starts.

2 The purpose

Many computer games in the genre "adventure role-playing games" have levels[1] filled with rooms that players explore in a quest for treasure and fame. To go to the next level you need to either solve a puzzle, find a stair-case in the maze of rooms, seek and destroy a vicious monster etc.

In this lab assignment we will construct a small program that creates, keeps track of, and displays a level. The program will also provide rudimentary support for moving between rooms by pressing keys on the keyboard.

The purpose of the assignment is to learn about dynamic objects that are created at run-time, recursive structures in the form of an incrementally constructed directed graph, and operations on such structures, and basic graphical user interfaces (GUI:s) where the model and the view of the model are kept separate.

3 Components

The final program will consist of five classes stored in two packages. On the course web page you can find Java files for all the classes. These are incomplete and your task is to fill in the missing parts. The five classes that you will write in this lab assignment are:

Main This class contains the mandatory method `main` to which the operating system transfers control when the program starts. This method does just two things:

1. It creates an object of type `Driver` and
2. calls the method `run()` that this driver has.

This class belongs to the package `lab2`.

Driver The driver class is responsible for creating both a level and a GUI for the level. To create a level, it first puts together a graph of rooms by creating a set of rooms and connecting them to each other by one-way corridors. This class belongs to the package `lab2`.

[1] Note that a *level* here is **not** the same as *experience level* that can be found in role-playing games.

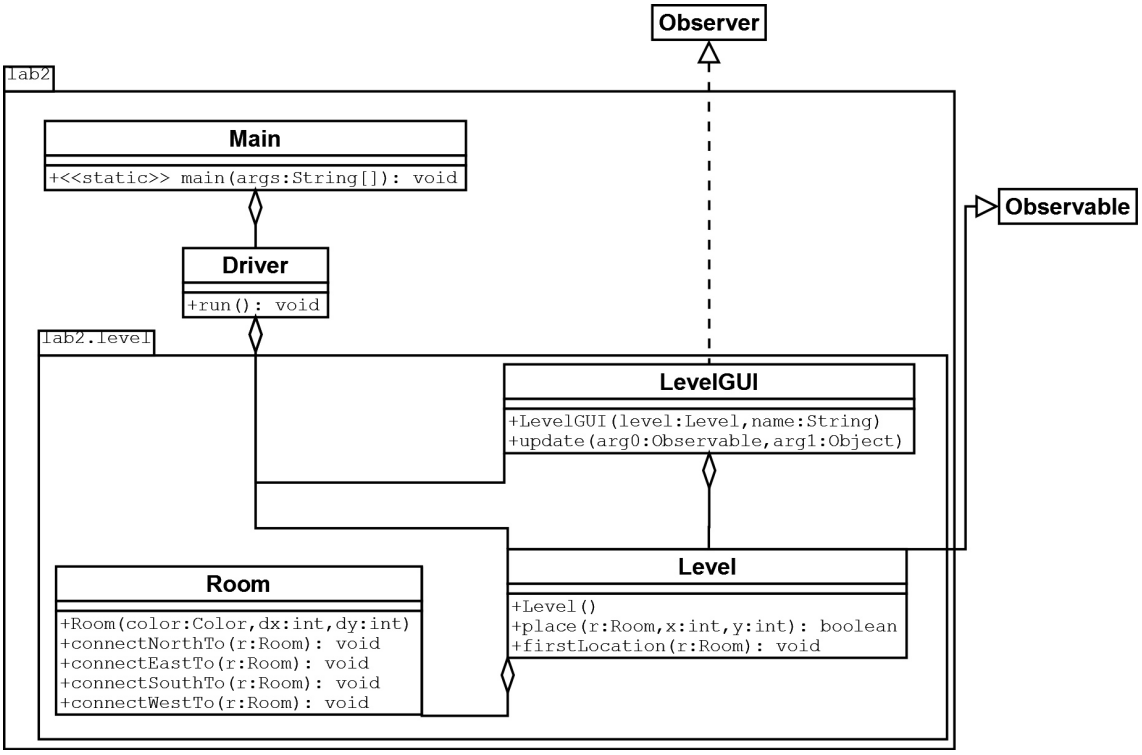

Figure 1: UML-diagram for lab 2. Private inner classes are not shown.

Room A dynamic object of this class represents a room. This class belong to the package `lab2.level`.

Level A dynamic object of this class represents a level. A level is based on a graph of rooms. This class belong to the package `lab2.level`.

LevelGUI A dynamic object of this class observes a level and visualizes it in the form of a simple 2-dimensional drawing in a separate window. This class belongs to the package `lab2.level` and contains two private inner classes:

Display A kind of area on which graphics can be painted. This is where everything that shows up on the computer screen is produced. A dynamic object of the class `LevelGUI` has a dynamic object of the class `Display`.

Listener A kind of `KeyListener`, and code that makes a level react on key strokes on a keyboard. A dynamic object of the class `Display` has a dynamic object of the class `Listener`.

Note that the class `Level` represents the level itself, while the class `LevelGUI` represents a *view* of the level together with functionality to let a user interact with the level. This difference will be explained in more detail later in this document. We will now program these classes in separate steps.

4 Main program

In Lab Assignment 0 we created plain programs with static objects only. These had static variables and static methods. In this assignment we have a number of *dynamic*[2] objects but just one (non-empty) static object: the class `Main`. This object contains the static `main`-method that always need to be present in Java programs. The only responsibility of this method is to

1. create a new dynamic object of type `Driver`, and

[2]Another name for dynamic object is *instance*.

2. to call the method `run()` the driver has.

Figure 2 shows a UML-diagram for `main` and `Driver`. All essential computations will take place in `run`, which belongs to a dynamic object of the class `Driver`, or in methods in other dynamic objects created during run-time. Likewise, all data and results computed during the program run are stored [in variables] in such dynamic objects. This is in principle how Java programs work in general: What happens essentially happen in methods of objects.

Task 1. Start by locating the provided code for lab 2. Copy the Java code in these files into classes in a new project in Eclipse before you start your programing. Note that the classes should be placed in packages that must exist before you create the classes. Make sure you preserve the given package structure.

Then finalize the class `Main` in the package `lab2`. Add a static method `main` so that it works as described above. Also, modify the class `Driver` so that the `run` method prints the string

```
"This is a print-out from the driver."
```

in the console in Eclipse (where all print-outs should be made, if not otherwise is stated).

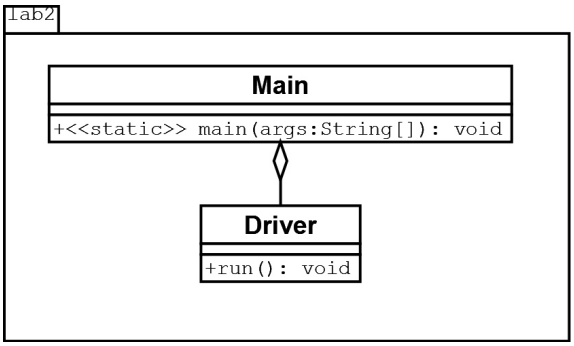

Figure 2: UML-diagram for Assignment 1.

5 Rooms

A room is a rectangular area bounded by four walls. The walls are axis-parallel and named *north*, *east*, *south*, and *west* according to the direction in which the wall is visible when viewed from within the room. The floor of a room is colored and different rooms can have floors with different colors.

In each wall there can be an opening, a *doorway*, leading to another room via a one-way corridor. When created, rooms have no doorways. Instead they provide the methods

- `connectNorthTo(Room r)`,
- `connectEastTo(Room r)`,
- `connectSouthTo(Room r)`, and
- `connectWestTo(Room r)`

each of which opens up a doorway in the wall in a given direction, and connects it to room `r`. Connecting rooms to each other via these methods creates a directed graph of rooms we call a *maze*. Nodes in a maze represent rooms. There are directed arcs between any pair of nodes that represent rooms that are connected by corridors.

The methods open up one-way connections only. This means that if there is a corridor leading from a room r_1 to another room r_2, there is not automatically a corridor going back from r_2 to r_1. Such a connection going back has to be explicitly created via a separate call to one of the methods:

```
r1.connectNorthTo(r2);
r2.connectSouthTo(r1);
```

This might seem strange and counterintuitive but makes it possible to have, for instance, teleportation where a player exits through a doorway and end up in a room far away with no immediate way back.

Note that there could be more than one doorway from a room leading to the same room. In fact, a doorway could even lead back to the same room(!) In the latter case, it is considered to not exist, since going through that doorway leads to the same result as staying in the room.

Although we assume that there are corridors that connect openings in walls to each other, corridors are not modelled internally in the program. Hence, there is no need for a class that represents corridors.

Task 2. There are two things to do here.

1. Write a class `Room` that implements a room (Fig. 3). Put the class in the package `lab2.level`.
   - The constructor should take as arguments the dimensions of the room (in number of pixels) and the color of the floor in the room.
   - You need to include, in your class, four dynamic reference variables of type `Room`, one for each wall, each of which refers to a doorway in the wall and what room that doorway leads to. To begin with, these should all be set to `null` to signal that there are no doorways.
   - You also need a dynamic variable to hold the color of the floor. In Java, the class `Color` is used to represent colors. For example, `Color.blue` can be used to represent the color blue, and so on. This class is already imported in the given class `Room`.
   - All variables should be declared with package access so that they can not be accessed from outside package `lab2.level`.
   - Change the constructor of `Room` so it prints the dimensions of the room and the color of the floor.
2. Change the method `run` in `Driver` so that it no longer prints text. Instead, first add declarations that creates (at least) three rooms with different dimensions and colors. Then, add Java code that connects them all into a maze so that it is possible to move between all rooms.
3. Run your program and verify that, when rooms are created, there are actually prints-outs that corresponds to the creation of dynamic objects. If you like, add or remove rooms and see how the print-outs change.

6 Level

A level consists of a maze. The rooms of the maze have fixed positions and there is at all time a room in which a player is located. A level has, in addition to a constructor, two public methods (Fig. 4):

place Once a level has been created, rooms of a maze can be inserted one at a time by calling the method `place(Room r, int x, int y)`. This method places the room so that its upper left corner ends up at (x, y) on an imaginary map. The coordinates are integers given in pixels ("screen coordinates").

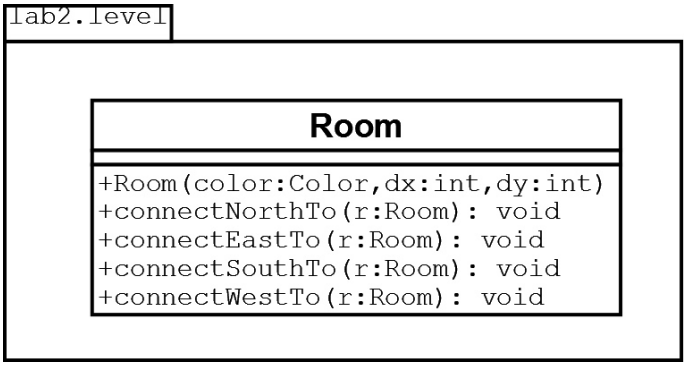

Figure 3: UML-diagram for the class `Room` in Assignment 2.

The rooms in a level may not overlap. The method `place` must check this. If a call to `place` is done with a room that overlaps an already placed room, the request to place the room is denied (not carried out) and the method returns `false` to signal this error condition[3]. Otherwise, if the room does not overlap with another room, the room is accepted, placed, and `true` is returned.

Hint: The coordinates of a room should be stored in the room itself. Change `Room` by adding variables with package scope to dynamic `Room`-objects so that `Level` can access those.

firstLocation This method takes as argument a reference to a room that has previously been placed in the level. The player is assumed to start in this room. The room where the player is located may change during the program execution and the level should keep track of the location of the player throughout its existence.

When this method has been called it should not be possible to place any more rooms on the level.

Task 3. Again, there are two tasks:

1. Write a class `Level` as described above. Make sure it belong to the package `lab2.level`.

2. Add code to the method `run` in the `Driver` class so that it creates a level and places 5-6 interconnected rooms in it. Re-use the rooms from the previous task and add some more.

   Remember to test that `place` really does not accept rooms that overlap.

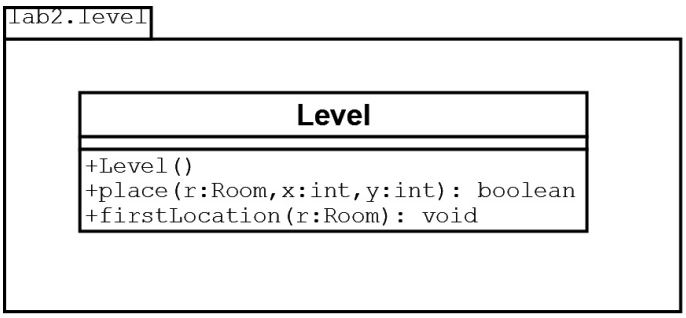

Figure 4: UML-diagram for the class `Level` in Assignment 3.

[3]Later on in the course we will signal errors like this using *exceptions* but more about them then.

7 A Graphical User Interface

Now, when we are able to create rooms and levels, we will add the ability of the program to visualize a level as a 2D-picture in a separate window. Such a visualization will be part of the *graphical user interface* (GUI) of the program. The GUI will also provide simple interaction with the user of the program so that it is possible to go from one room to another through a corridor by pressing keys on the keyboard.

It is possible to make a nice-looking GUI, with decorated rooms filled with beautiful artifacts and pictures of players and other creatures moving around. Our purpose is, however, not to study how such works of art can be produced but rather to show how GUI can be programmed in Java. Once the technique is known, the simple geometric shapes we will use (mostly colored rectangles and lines) can swiftly be substituted for more elaborate and nice looking pictures and shapes.

The program we will end up with makes a clear distinction between the *model* – in our case a level containing rooms with all their defining characteristics – and the *view* of the model – in our case a drawing of a level. This separation makes it possible to code the behavior of levels and rooms, and to create any number of them, without bothering about how they are displayed. Decoupling the model from the view also makes it easier to simultaneously have more then one view. Indeed, it even becomes rather easy to have several fundamentally different views being presented at the same time, for instance one graphical, one textural, and a view of a level based on audio.

The model is already described in the classes `Room` and `Level`. What we will add is a class `LevelGUI` that represents a graphical view of the model.

7.1 Creating an Interactive GUI

The GUI we will program has two responsibilities. First, it opens up a new window on the screen. In this window it then draws a picture of a level. Whenever the internal state of the level changes, the GUI redraws the picture to reflect the change. Moreover, the GUI reacts to keys being pressed on the keyboard and takes appropriate action, requesting a change of the internal state of the level, depending on what key is pressed.

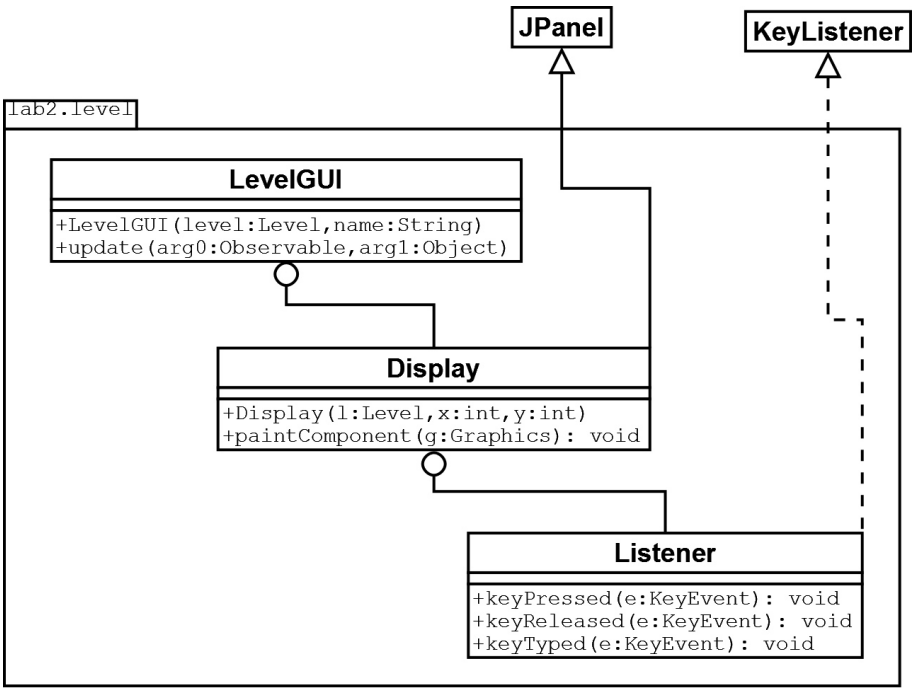

Figure 5: UML-diagram for the class `LeveGUI` in Assignment 4.

The class `LevelGUI` contains variables and methods (Fig. 5). It also has a private inner class `Display` that contains a private inner class `Listener` of its own. The `LevelGUI` constructor

1. creates a window (called a `JFrame`) that exits the program when its close button is pressed, and
2. creates an instance of the `Display` class, which it then adds it the window.

7.2 Drawing the picture

The first step is to actually draw the picture of the level. The `LevelGUI` class already contains the inner class `Display`, which will take care of displaying the level.

The method `paintComponent` is automatically called each time the display needs to be redrawn. This happens, for example, each time the window is moved on the computer screen, or when another window that previously blocked the display is removed. It is in this method we will add code to display all the rooms as well as the connections between the rooms.

Task 4. Add code to `paintComponent` in `Display` to draw a 2-dimensional view of all the rooms and corridors in the level. You are free to implement your own way of drawing the rooms and the corridors, as long as:

- Sizes and locations of rooms are correct.
- It is easy to see which room the player is currently in; one way could be to give that room a different border than all the other rooms.
- The color of the floor of each room is displayed.
- Connections between rooms are displayed. Find a way to show the direction of each connection.

Since this is probably the largest part of this assignment, do not put all the code directly in `paintComponent`. Rather, divide the code logically into (private and) more easily programmed methods, that you then call.

7.3 React to changes

When there is a change in the level we display, we want the GUI to be notified so that it can redraw the level. We achieve this via a design pattern know as the *Observer Pattern*. This pattern prescribe two kinds of entities and their relationship. The first kind is something *observable*, in this case a level. The other kind of entity is an *observer* that reacts and presents an updated view of something observable that it is observing. In our case, the view will be a 2-dimensional and colorful drawing of a level.

The functionality we get when we use the Observer pattern is that the observable entity (the level) is responsible for informing all observers as soon as an essential change in its external state has occurred. The observers, in turn, are responsible for updating their view of the observable entity when they are notified of a change.

7.3.1 Observable

We want the `Level` class to be observable. In this laboration, you can treat `Level` as if it already contains the methods `setChanged`, `notifyObservers`, and `addObserver`, even though you have not written these methods. [4]

The first, `setChanged`, should be called by the level as soon as there is a change that could effect a view of the level. The method `notifyObservers` informs all observers that there is a change to consider. The two are often called in this order and one after another. Making these calls when needed is the responsibility of the observable object.

[4] This might seem strange, but the reason for this is that `Level` *inherits* another class. We have not yet covered inheritance in this course so for now you can just think of it as if `Level` contains methods that are actually written in another class (in this case, in a class called `Observable`).

The method `addObserver` must be called by every class that wants to observe the level. To be able to do so, an observer need to have a reference to the object it likes to observe. This is the responsibility of the observer.

Task 5. Add methods that makes it possible to change the state of `Level`.

- Add methods in `Level` that enables the user of the program to move the player in any of the four possible directions if there is a corridor in that direction (don't forget to check for it!). In other words, the user should be able to change the room in which the player is located. These methods should be visible in the package, but not from outside the package.
- As soon as the player moves from one room to another, potential observers should be informed of this fact by calling the methods `setChanged` and `notifyObservers`.

It is allowed to make four methods, one for movement in each direction. However, if parts of several methods are identical, that part of the code should only exist in one place. For example, all such code could exist in a private method that is called from the other methods.

7.3.2 Observer

An observer class must contain the method `update`, which `LevelGUI` already does. When an observable object notifies its observer that it has changed, the method `update` at the observer is automatically called. Since this method is currently empty, nothing happens when our GUI is informed of a change. Your job is to change that.

Task 6. Make `LevelGUI` react to changes in the Level by redrawing the picture.

- Add code to the `LevelGUI` constructor so that it adds itself as an observer to the level.
- Add code to the update method so that the display is repainted when the level reports a change. This is achieved by calling the method `repaint` of the `Display`-object with no arguments.

  This will eventually, when the run-time environment and the operating system find it suitable, result in a call to `paintComponent` in `Display`.

7.4 Moving around

Now we have a view of the level that responds to changes in the model, but we have no way of actually moving around in the level yet. The final step of this lab is to add functionality to our GUI so that we can move around in our level using the keyboard.

The `Display` class already contains the inner class `Listener` that listens for key events. The methods `keyPressed`, `keyReleased`, and `keyTyped` are run each time a key is pressed, released, or typed. Each method takes the key event as an argument. If we want to know which key the event concerns, we can use the method `getKeyChar`. Since all the methods are currently empty, nothing happens when we try to use the keys.

Task 7. Add functionality to the `Listener` class so that the four keys `w`, `s`, `a`, and `d` on the keyboard can be used by a user to move the player to the north, south, west, and east on the level. Movement should, of course, only be carried out if there is indeed a doorway in the indicated direction.

Note that the `Listener` class already contains a reference to the level.