CHAPTER 1

Design a large application

1.1 Objectives

The main aim of this book is to acquire skills to design a large software application. The approach taken to achieve this goal is to make a video game using software design. Each stage of this realization is punctuated by the presentation of these tools and some exercises to better understand them. The Pacman game developed throughout the book serves as an example for each introduced concept. The Pacman game is a well-known simple game: it illustrates concepts without losing the reader in issues specific to a particular video game. Note that pedagogical and fun reasons motivate the choice of making a video game instead of a more traditional application. For instance, it is possible to use notions introduced in this book to make a merchandise inventory management software. However, this kind of realization is less likely to motivate the acquisition of skills!

The most generic tools presented in this book are *Design Patterns*, applied in all stages of development. These patterns are proven recipes that solve many computer design problems without having to reinvent the wheel. Once gained, they save valuable time in design and implementation. They also solve complex problems that, without this approach, are very difficult or impossible to model, even for the greatest coders. More specific tools are also presented according to the needs of the game, such as data structures, algorithmic, display, networks... As presented in this book, design patterns improve these tools.

The method followed in this book also makes it possible to design large applications. Past a certain size or a certain level of complexity, it becomes very difficult to create an application that combines performance and scalability - while ensuring the achievement of requested features. If the problem of computer design has to be summarized to a single question, it would be this one: how to implement functionalities whose complexity exceeds human understanding? This problem has, until proven otherwise, only one solution: the division of complex problems into simple sub-problems. The presentation of this problem is the main subject of this chapter, which we motivate, present, and finally put into practice in the following sections.

The acquisition of all these tools requires some energy. The first tools are the simplest, and their presentation is very detailed to be accessible to beginners. Then, as time goes by, we see more and more complex tools. Readers with more experience will judge the first chapters too simple and find more relevant content in the following chapters. If ever a chapter seems too difficult, we recommend reading the previous chapters again and do all the exercises. All presentations and exercises have a specific pedagogical function, which the reader will only realize once he has improved its skills. Finally, if a feeling of repetition sets in, do not worry! It can happen if the new concepts presented do not seem so new, as if we constantly repeat the same pattern. Such a feeling means that you acquired the design skills and that this book achieved its main aim!

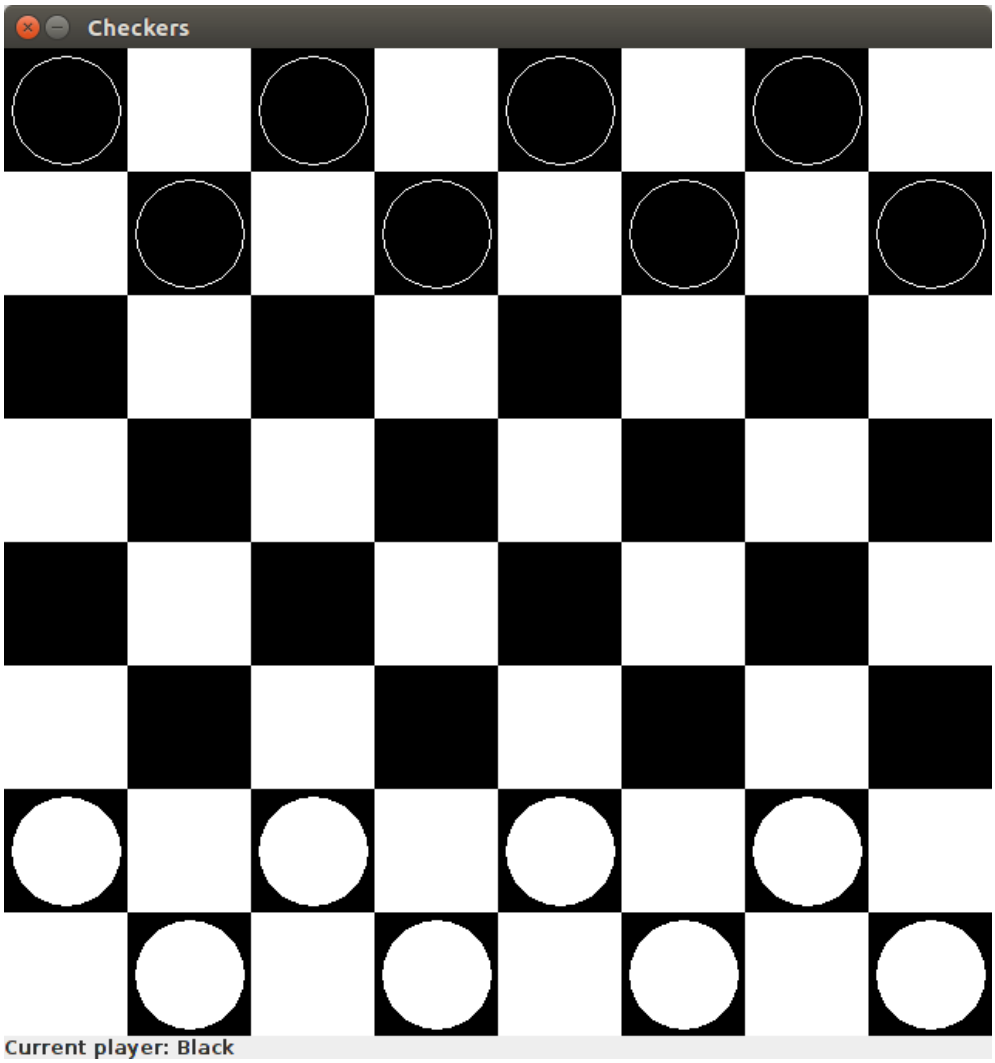
1.2 Software design: a considerable time savings

1.2.1 A simple example

Implementing a large application requires a lot of work time dedicated to software design. When you start in software development, the energy claimed for this stage is not natural. It is a normal feeling: why waste so much time when it is - a priori - much more effective to dive right into the code? People who have tried this naïve approach will testify that it is not the right one - except perhaps for very simple problems. However, it is difficult to convince oneself without having experienced this.

To reproduce the experience of implementation without conception, we consider a simple example: the checkers game. This game has eight black and eight white pieces placed on a checkerboard of eight by eight squares. Only black squares can accommodate pieces.

The initial state is as follows:



There are two types of movement: moving to an adjacent free square and capturing a piece of the other color. We do not consider the creation of king pieces in this example.

Implementing this without design, we produce the following code:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
```

```
import java.awt.event.MouseListener;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.Jlabel;

class BoardComponent extends JComponent {
    public BoardComponent() {
        setPreferredSize(new Dimension(80 * 8, 80 * 8));
    }
    @Override
    public void paint(Graphics g) {
        for (int j=0;j<8;j++) {
            for (int i=0;i<8;i++) {
                if (((i+j)%2) == 1)
                    g.setColor(Color.WHITE);
                else
                    g.setColor(Color.BLACK);
                g.fillRect(i*80, j*80, 80, 80);
                int p = board[i][j];
                if (p >= 0) {
                    if ((p%2) == 0)
                        g.setColor(Color.BLACK);
                    else
                        g.setColor(Color.WHITE);
                    g.fillOval(i*80+5, j*80+5, 70, 70);
                    g.setColor(Color.WHITE);
                    g.drawOval(i*80+5, j*80+5, 70, 70);
                }
            }
        }
    }
}

public class Checkers extends JFrame implements MouseListener {
    private int board[][];
    private int currentPlayer;
    private int selectedPiece[];
    private JLabel statusBar;

    public Checkers() {
        currentPlayer = 0;
        board = new int[8][8];
        for (int j=0;j<8;j++) {
            for (int i=0;i<8;i++) {
                board[i][j] = -1;
```

```
            }
        }
        board[0][0] = board[2][0] = 0;
        board[4][0] = board[6][0] = 0;
        board[1][1] = board[3][1] = 0;
        board[5][1] = board[7][1] = 0;
        board[0][6] = board[2][6] = 1;
        board[4][6] = board[6][6] = 1;
        board[1][7] = board[3][7] = 1;
        board[5][7] = board[7][7] = 1;

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false);
        setTitle("Checkers");
        setLayout(new BorderLayout());
        getContentPane().add(new BoardComponent(),
            BorderLayout.CENTER);
        statusBar = new JLabel("Current player");
        getContentPane().add(statusBar,BorderLayout.SOUTH);
        pack();
        redraw();
        addMouseListener(this);
    }
    public void redraw() {
        String msg = (currentPlayer==0)?"Black":"White";
        statusBar.setText("Current player: "+msg);
        repaint();
    }
    @Override
    public void mouseClicked(MouseEvent e) { }
    @Override
    public void mousePressed(MouseEvent e) {
        int i = e.getX() / 80;
        int j = e.getY() / 80;
        if (board[i][j] == currentPlayer) {
            selectedPiece = new int[2];
            selectedPiece[0] = i;
            selectedPiece[1] = j;
        }
        else {
            selectedPiece = null;
        }
    }
    @Override
    public void mouseReleased(MouseEvent e) {
```

```
        if (selectedPiece == null)
            return;
        int i = e.getX() / 80;
        int j = e.getY() / 80;
        if (board[i][j] < 0 && ((i+j)%2) == 0) {
            boolean doMove = false;
            int i0 = selectedPiece[0];
            int j0 = selectedPiece[1];
            if ( (i==i0+1 && j==j0+1) || (i==i0+1 && j==j0-1)
              || (i==i0-1 && j==j0+1) || (i==i0-1 && j==j0-1)) {
                doMove = true;
            }
            else if (i==i0+2 && j==j0+2 && board[i0+1][j0+1]
                 == 1-board[i0][j0]) {
                board[i0+1][j0+1] = -1;
                doMove = true;
            }
            else if (i==i0-2 && j==j0+2 && board[i0-1][j0+1]
                 == 1-board[i0][j0]) {
                board[i0-1][j0+1] = -1;
                doMove = true;
            }
            else if (i==i0+2 && j==j0-2 && board[i0+1][j0-1]
                 == 1-board[i0][j0]) {
                board[i0+1][j0-1] = -1;
                doMove = true;
            }
            else if (i==i0-2 && j==j0-2 && board[i0-1][j0-1]
                 == 1-board[i0][j0]) {
                board[i0-1][j0-1] = -1;
                doMove = true;
            }
            if (doMove) {
                board[i][j] = board[i0][j0];
                board[i0][j0] = -1;
                currentPlayer = 1-currentPlayer;
                redraw();
            }
        }
        selectedPiece = null;
    }
    @Override
    public void mouseEntered(MouseEvent e) { }
    @Override
    public void mouseExited(MouseEvent e) { }
```

```
    public static void main(String[] args) {
        Checkers checkers = new Checkers();
        checkers.setLocationRelativeTo(null);
        checkers.setVisible(true);
    }
}
```

The code above is complete and can be compiled and run with no other dependency than the standard Java library. The corresponding Java file is available in the Java Project provided with this book (see Section 1.3.3), in the folder examples.checkers.

→ To run it, open the Java project with Netbeans, locate the Java file "examples/checkers/Checkers.java" in the source packages. Right-click on this file and select **Run File** from the context menu.

→ To play, drag a piece of the current player.

1.2.2 Divide and conquer

This first roll is functional and allows us to play the game in its simplest version. Is the design necessary?
One could claim that it is not the case because the complexity of the problem is so low that we don't need any design. However, if we want to enrich the solution with new features, serious problems will arise.

Graphic library

The first solution uses the display features present in the standard Java library. It is relevant for office software and simple cases like the checkers game. To propose a more pleasant display, one should get closer to the graphics card of the machine. Lower level libraries like LWJGL (*Lightweight Java Game Library*) can provide such a display. To use one of these libraries, you must change the initial code in several places. More precisely, it is necessary to start with a first step of identifying the zones interacting with the graphic elements of the standard library. With no structure to find them quickly, this poor design forces the developer to review all the code. This example code is only about 150 lines, and the identification will be fast. In real cases, you have to go through thousands, tens, hundreds of thousands of lines of code. Once this identification is established, almost all identified sections will have to be rewritten completely.

This issue of graphic library change is not the most important. Indeed, it can be very interesting to be able to deploy the game on different platforms (pc, mac, console, mobile, . . .), each needing a specific graphic library. An approach without

design is to produce a code specific to each platform. It involves changes for each of these versions, resulting in a loss of time and a significant risk of errors.

Design patterns and software engineering can easily solve these problems. They make it possible to isolate the problems of display of the rest of the logic of the game. The result is a solution that can easily switch from one graphics library to another.

To conclude: it is relevant to isolate display features from the rest of the game.

Controls: User Interface, Artificial Intelligence and Networking

In the initial example, the mouse controls the pieces. It may be interesting also to propose other controls like gamepads on consoles, whose logic is very different from that of the mouse. To do this, you must identify corresponding code areas and modify them accordingly. For example, you will have to produce a method equivalent to `mouseReleased()`, surely something like `gamepadBoutonPressed()`. Furthermore, the analysis of the mouse handling method shows that two features are mixed: the management of the mouse and the rules of the game. Actually, it is at this level that movement rules and piece capture are implemented. To get a similar result with a gamepad, you need to reproduce these rules in the gamepad handling method. In the case of the checkers game, this is quite simple, but for many other games, the rules are much more complex. It results in a significant risk of errors: some actions may be possible using one device but not in the others.

In a game, it is often interesting to propose artificial intelligence (AI) to play as a human player. In this case, AI replaces the mouse or the gamepad and controls the game with a mechanics of an even more different nature. As we will see in the corresponding chapter, AI can control the game at a lower level, without going through button press mechanics or other. Adding an AI in our simple implementation of checkers is like creating an additional method, which will also need to apply the rules of the game. It increases the risk of errors again.

One last way to control a game is by networking. In this case, it is generally more interesting to only transmit on the network the commands of each player than the entire data of the game. In doing so, network packets control remote players, as if their devices were directly attached to the local machine. With a naive approach, this new control is used via new methods to implement, with the same risks that come with it.

To conclude: it is relevant to isolate the controls from the rest of the game.