THE UNIVERSITY OF SYDNEY

SOFT2201/COMP9201 Tutorial 6

Behavioural Design Patterns 1

Behavioural Design Patterns 1

Behavioural patterns allows you to encode behaviour within objects to be executed at run time. Behavioural patterns like *State* and *Strategy* patterns allow the programmer to utilise input at runtime to change the object's behaviours.

Strategy Pattern

One thing to note is that behavioural patterns encapsulate information required to act in different ways. For strategy pattern, selecting an algorithm (or method) based on run-time instructions, this information is commonly encoded within the invoking method.

Question 1: Ranking Selection

We have been given a set of lap times from each race car in the race. We want to present different information based on different requests. We want to display three kinds of data: **Fastest Lap Time**, **Fastest Race Time** and **Slowest Race Time**,

You can make reasonable assumptions regarding the Racecar class and DisplayMethod enum (the current code successfully compiles and runs). Consider the following code and think about how its current version provides extensibility (identify the SOLID principle that most applies here and why). What would the Strategy pattern look like if we applied it here?

```
public class Race {

    private List<Racecar> cars = new ArrayList<>();

    public List<Double> displayData(DisplayMethod method) {
        List<Double> times = new ArrayList<>();

        if(method == DisplayMethod.FastestLapTime) {
            for(Racecar car : cars) {
                times.add(car.fastestLapTime());
            }
```

```
            Collections.sort(times);
        } else if(method == DisplayMethod.FastestRaceTime) {
            for(Racecar car : cars) {
                times.add(car.raceTime());
            }
            Collections.sort(times);

        } else if(method == DisplayMethod.SlowestRaceTime) {
            for(Racecar car : cars) {
                times.add(car.raceTime());
            }
            Collections.sort(times);
            Collections.reverse(times);
        }
        return times;
    }
}
```


Question 2: Strategy Ball

You have done the implementation of bouncy ball in your weekly exercise 4. In this question, you will notice that somebody has extended your code to handle a few bouncing balls. The extended codebase titled `StrategyBalls.zip` can be found on canvas. You need to refactor this codebase to use the **Strategy** pattern.

The strategies you need to implement will need to be settable on each `Ball` by `BallPit` in the manner of your choosing (you are told which colour ball to apply what strategy to, but you must not hard code this in the Balls themselves). The strategies each involve the Balls changing their own movement - however they are still bound by the `BallPit` rules - they cannot go past the edge, and if another ball hits them they will bounce using the energy-conserving algorithm in `BallPit` (your strategies don't need to check this, the BallPit code should enforce it).

The Balls each have the ability in their `think()` method to accelerate by 1.0 per second per axis (that is, they can add or subtract anywhere from 0 to 0.017 each from their own `xVel` and `yVel` per call to `think()`). *Note that negative velocities are normal.*

The strategies you must implement are as follows:

- Black ball: Accelerates as fast as it can either NorthWest, NorthEast, SouthWest, or SouthEast depending on which corner it is closest to
- Blue ball: Tries to stop when it bounces off another ball, keeps going in all other cases (including bouncing off a wall)
- Red ball: Does nothing (note this doesn't mean stop, it means keep moving the same speed and direction it is moving)

One of the key questions you can discuss with your tutor during tutorials is 'how will the Balls learn about their environment?'. There are a few ways to do this, some representing good design, and some representing bad design - choose wisely! You can start by implementing the Strategy pattern with the Red ball strategy applied to all balls, then work on adding the Black and Blue strategies.

State Pattern

In regards to `Strategy` pattern, the selection or retrieval of the behavioural objects is known by a single object maintaining links to them. However, State pattern encodes state transition on each state object which will affect the behaviour of the object maintaining that state.

Question 3: Switchboard Design

Your company `Telstro` is redesigning their old switchboard to operate on a completely software based system. The switchboard accepts events from phone ports which will then require a connection between the ports to be constructed. Assuming the current design matches the code in the next question, draw out your redesign in UML **before** diving straight into the code!

Question 4: Switchboard Operator

After submitting the design, `Telstro` has contracted you to overhaul their current software system.


```
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class Switchboard {
    private Map<String, PhonePort> ports;

    public Switchboard(int n) {
        this.ports = new HashMap<>();
        for(int i = 0; i < n; i++) {
            this.ports.put("" + i, new PhonePort("" + i));
        }
    }

    public void connect(String from, String to) {
        PhonePort source = ports.get(from);
        PhonePort destination = ports.get(to);
        PhoneLine line = new PhoneLine(source, destination);
        source.connect(line);
        destination.connect(line);
    }
```

```
    public void disconnect(String address) {
        PhonePort source = ports.get(address);
        source.disconnect();
    }

    public void hold(String address) {
        PhonePort source = ports.get(address);
        source.hold();
    }

    public void resume(String address) {
        PhonePort source = ports.get(address);
        source.resume();
    }

    public static void main(String[] args) {
        Switchboard switchboard = new Switchboard(10);
        Scanner input = new Scanner(System.in);
        while(input.hasNextLine()) {
            String[] words = input.nextLine().split(" ");
            if(words.length == 3) {
                if(words[0].equals("CONNECT")) {
                    switchboard.connect(words[1], words[2]);
                } else if(words[0].equals("DISCONNECT")) {
                    switchboard.disconnect(words[1]);
                }
            } else if(words.length == 2) {
                if(words[0].equals("HOLD")) {
                    switchboard.hold(words[1]);
                } else if(words[0].equals("RESUME")) {
                    switchboard.resume(words[1]);
                }
            }
        }
    }
}

public class PhonePort {

    private String address;
    private PhoneLine line;
    private PortState state;

    private static enum PortState {
        Waiting, Busy, Holding
    }
```

```
    public PhonePort(String address) {
        this.address = address;
        line = null;
        state = PortState.Waiting;
    }

    public void connect(PhoneLine line) {
        if(state == PortState.Waiting) {
            state = PortState.Busy;
            this.line = line;
        }
    }

    public void disconnect() {
        if(state == PortState.Busy) {
            state = PortState.Waiting;
            line = null;
        }
    }

    public void hold() {
        if(state == PortState.Busy) {
            state = PortState.Holding;
        }
    }

    public void resume() {
        if(state == PortState.Holding) {
            state = PortState.Busy;
        }
    }
}

public class PhoneLine {
    private PhonePort from;
    private PhonePort to;
    public PhoneLine(PhonePort from, PhonePort to) {
        this.from = from;
        this.to = to;
    }

    public PhonePort source() { return from; }
    public PhonePort destination() { return to; }
}
```

Extra Practice Question: Tic Tac Toe

This is not a mandatory tutorial question, instead it is an extra question for students who are looking for more questions for practice. Feel free to discuss your idea on this question with your tutor during tutorial.

You are going to refactor the design of a tic tac toe game. The current version of the code is available on Canvas, which is in the process of implementing a single player. You will need to redesign the application to incorporate an AI player and allow for different difficulty settings (or *strategies*). **Before** diving straight into the code, draw out your redesign using UML. Assume two strategies are available: *Easy* and *Normal*.

Given the existing code base of the tic tac toe game, implement the strategy pattern as well as each strategy. Each strategy is selectable by the player for the AI to employ.

An **Easy** strategy is where the AI randomly places their symbol on an available board piece.

A **Normal** strategy will attempt to place their symbol randomly, unless the computer can win in the next turn. If the AI knows that they can win in the next turn, it will attempt to do so.

(If you feel up to a challenge, implement a **Hard** strategy that leverages one or two more difficult strategies)