SOFT3202

Adam Ghanem

March 25, 2022

Contents

1 Tutorial 4 EXPORT

We can finally start getting into some design topics, and get ready to deconstruct we some previous teachings of object-oriented programming.

1.1 Why do we use Design Principles/ Patterns?

One of the main questions that have not only been asked of me, but ones I have asked myself is why do this in the first place. Why begrudge oursevles to the change our disposition towards design towards one filled with patterns, SOLID, GRASP and whatnot. The reason why we learn these principles is not just to drop them into our programs and call it a day, but rather to achieve our goals for software development:

- **Maintainability** - *how easy is our code to manage and find errors in?*
- **Extensibility** - *how easy is our code to extend and add features to?*
- **Modularity** - *how easy is it for our code to be reused into many functions?*
- **Usability**
- **Security**

1.2 Design Principles

For each of these, try to remember what each principle roughly stands for. Remember there is no use of rote-learning these principles, understanding them is much more practical and beneficial to you.

1.2.1 SOLID

1. Single Responsibility Principle:

   From the notorious textbook [?], we have that:

   > A class should have only one reason to change.

   ```
   interface Game
   {
    // these happen to store/load game
    public void load(String level);
    public void save();

    // These happen while running
    public char play();
    public void changeDifficulty();
   }
   ```

   In this scenario here we can probably say this is a good way to manage our code. Can we separate out the methods of this `Game` class into separate classes. Well we have `load` and `save` dealing with the data of the game and storing/loading some state of it. Whereas `play` and `changeDifficulty` are concerned with changing the state of the game and behaviour. We can split these up into two separate classes: one that manages loading and one that manages game interactions.

2. Open/Closed Principle

   Again from [?]:

   > Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

   The reason is that disobeying this principle leads to a lot of **Rigidity** down the line forcing us to refactor large amounts of code. In this principle code should be **Open for Extension** but **Closed for modification**.

   - Open for extension: means we can extend the module if requirements change
   - Closed for modification: we do not have to touch the original code of the module.

   We accomplish this through interfaces and abstraction in general. We need to add a new game mode to our `Game` class, simple, we just extend the `GameMode` interface and add a new mode to inject through.

   So take a look at this example:

```
public class Game {

  // What are the issues with this method
  public void playGameMode(String mode) {
    switch(mode) {
    case "single-player":
      playSingle();
      break;
    case "two-player":
      playDouble();
      break;
    case "sandbox":
      playSandbox();
      break;
    case "creative":
      playCreative();
      break;
    case "tower defence":
      playTowerDefence();
      break;
    default:
      playSingle();
      return;
    }
  }


  // The improved version.
    public void playGameMode(GameMode mode) {
      mode.play();
    }
}
```

The issue here is that if we want to add a new game mode then we have to modify the original code within `Game`. I have already provided a hint to the solution, how would we change this such that we can obey the OCP principle.

3. Liskov Substitutability

   This one looks a bit difficult but generally the sentiment behind it is quite simple to follow:

   > The subtypes of a class must be able to have the ability to substitute in and take the place of their parent classes.

   It is created by the distinguished computer scientist, Barbara Liskov, of whom the principle is named after. Her explanation on what she desired from this principle is the clearer than anyway I could possibly attempt to put it:

   > If for each object of type S there is an object of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when it is substituted for S (which is a subtype of T). Barbara Liskov, 1988

4. Interface Segragation

   This is quite a basic but very powerful design principle, in that we must avoid the issue of non-cohesive interfaces which contain too many unrelated responsibilities. Instead we should focus on dividing and separating our interfaces each into groups of similar methods. Again from [?]:

   > Clients should not be forced to depend on methods that they do not use.

   Going back to the `Game` example say we have:

```
public class Game extends VirtualRealityApplication {
  public void play();
  public void stop()
}


public class VirtualRealityGame extends Game {
  public void loadMap(String map) {
    // do something
  }
}

// Say we have a VR engine class that takes in a VR application and applies the tr
public class VirtualRealityEngine {

  // It takes in a VR Application to load and transform
  public void loadScene(VirtualRealityApplication vrd) {
    // do something
  }
}

// This is the client class that is fed to the VR Engine
public class VirtualRealityApplication {
}
```

   Now based on this implementation, I can simply make `Game` extend `VirtualRealityApplication`, however, this is a suboptimal solution.

The issues behind doing is because not every Game is `VirtualRealityApplication` and hence you are just creeping in extra useless functionality. Instead we can have our `VirtualRealityApplication` be extended by `VirtualRealityGame` and broken up as interfaces.

```
public interface Game {
  public void play();
  public void stop();
}

// This is the client class that is fed to the VR Engine
public interface VirtualRealityApplication {
}


public class VirtualRealityGame implements Game, VirtualRealityApplication {
  public void loadMap(String map) {
  }
}

// Say we have a VR engine class that takes in a VR application and applies the tr
public class VirtualRealityEngine {
  // It takes in a VR Application to load and transform
  public void loadScene(VirtualRealityApplication vrd) {
    // do something
  }
}
```

5. Dependency Inversion

   The whole goal of this principle is to eliminate our transitive dependencies (classes being dependent or impacted by other dependency changes). From [**?**]:

   (a) High-level modules should not depend on low-level modules. Both should depend on abstractions.
   (b) Abstractions should not depend on details. Details should depend on abstractions.

   If $A \to B \to C$ but changes in C cause major issues in class A then we call this a transitive dependency. This is related to the Hollywood principle which we will uncover later on. But generally we want to split these up through abstractions with interfaces. To avoid this we should depend on three major guidelines [?]:

- No variable should hold a pointer or reference to a concrete class.
  - e.g. NO `Kitty c = new Kitty()` instead if Animal is interface then `Animal a = new Kitty();`
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.

Since this may be a bit difficult to understand I will use a proper example from [?]:

```
public class Button {

  // Uses Lamp
  public Lamp lamp;

  public void poll() {
    if (lamp.isOn()) {
      lamp.turnOff();
    } else {
      lamp.turnOn();
    }
  }
}

public class Lamp {

  public void turnOff() {}
  public void turnOn() {}

}
```

As we can see here, `Button` is dependent upon `Lamp` and therefore is affected by the changes within `Lamp`. Before the refactored version is displayed, can you think of the changes you would need to make in order to improve this?

One possible solution involves creating an abstraction between these two classes in the form of a `SwitchableDevice`:

```
public class Button {
  protected SwitchableDevice sd;

  public void poll() {
    if (sd.isOn()) {
      sd.turnOff();
    } else {
      sd.turnOn();
    }
  }
}

public interface SwitchableDevice {
  public void turnOff();
  public void turnOn();
  public boolean isOn();
}

public class Lamp implements SwitchableDevice {
}
```

This design now allows button to control any device which is switchable, allowing for a lot of **flexiblity** and improves **extensibility** down the line. Moreover, nothing really owns the `SwitchableDevice` interface and `Button` no longer depends on a concrete class, `Lamp`.

1.2.2 GRASP

These will be a bit more verbose to list and I will add to these as I go along. The GRASP or *General Responsibility Assignment Software Patterns and Principles.* These consist of:

1. Information Expert
2. Creator
3. Controller

4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

1.2.3 KISS

This is one of my favourites and the one that has stuck with me forever:

KEEP IT SIMPLE STUPID

I don't think this even needs an explanation and I see this from variable names, method names, classes, number of methods or the extra lines that act as a tautologies etc. Don't overthink usually if you can't find a use for it don't introduce it (this is particularly important for the next principle).

1.2.4 YAGNI

Another favourite:

You ain't gon need it

You know that feeling when you start writing up a software project and you start fantasising over all the possible additions you are going to make. How this simple MineSweeper game will somehow turn into a VR Dark Souls-like RPG. The truth is your aren't going to need all the extra methods you are thinking of and rather you should focus on current **requirements** or **capabilities**. Even if you may think there could be uses for such a game in the future but the addition of it now will overcomplicate things, think of Occam's Razor.

1.2.5 DRY

Most developers would have had this yelled at them some point in their lives:

Don't Repeat Yourself

A bit along the lines of being modular but we humans are lazy. And although contrasting the famous adage, "Money can't buy happiness', if we can reuse existing code or even better write modular code, it will definitely make us happier humans down the line.

1.2.6 SSOT

I always go with the shorter version of this:

Single Source of Truth

This principle extends upon the **DRY** principle in that we can have different verions of things but there still needs to be a single representation of this solution within the system. In view of this, if we are developing an App, and we need to write the same exact code in both `Swift` and in `Java` then we are possibly betraying the DRY and SSOT principle and rather can find a language that works across both devices, or some domain specific language.

1.2.7 Connasence

Connasence is an extension of two principles covered above - Coupling and Cohesion. This principles allows us to visualise and determine how coupled a dependency is and thus find ways to improve and decouple it. For two modules to be **connascent** then changing one module forces you to change another in order to maintain correctness.

1.2.8 Law of Demeter

If you have done some form of relational databases and logic this would make a bit more sense to you. If we have multiple classes:

- Cow $\rightarrow$ Grass $\rightarrow$ Sun
- where $A \rightarrow B$ means A is dependent upon B.

  How much `Cow` depends on `Sun` refers back to the Law of Demeter. The idea is that we want `Sun` class to be able to change, add functionality, etc without effecting how `Cow` works or breaking `Cow` (we are strictly assuming the laws of physics do not apply here and the Sun cannot disappear or wipe out Earth).

This means we do not have `Cow` directly calling `Sun` in anyway:

```
public class Cow {

  public void moo() {
    this.grassHeap.getSun().produceSunlight();
  }
  public void eat() {
    grassHeap.kill();
  }

}
```

The class `Grass` acts as our middle-man and if we were to axe out `Sun` or change it, `Grass` should manage it.

1.2.9 Composition over Inhaaaaance

This is something that is an should be referenced very early on in your object oriented path, the *has-a* versus the *is-a* relationships. The general idea is that instead of forcing inheritance on our design we should rather make use of dependency injections and composition. The reason being in that **inheritance** in a way forces **rigidity** in our code base, changing one section can cascade down our tree.

The C2 discussion that was recommended is actually offers a fantastic read on this topic.

1.2.10 Hollywood Principle or Inversion of Control

Could possibly be renamed into the **FAANG Interview Principle**:

> Don't call us, we'll call you

This relates to the Dependency Inversion principle in that we want the program to be controlled and sequenced by a parent class, a controller in a sense.

1.3 Designing Software

1.3.1 God Method

In this section we will be refactoring the provided God method, something that you will or have unfortunately encountered many times throughout your software development careers:

```
package week5;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class Main {

    public static void main(String[] args) {
        BufferedReader reader;
        ArrayList<String> roomStrings = new ArrayList<>();
        ArrayList<String> itemStrings = new ArrayList<>();

        List<List<String>> playerInventory = new ArrayList<>();
        List<String> allowedDirections = new ArrayList<>(Arrays.asList("NORTH", "SOUTH"
        int playerRoom;

        try {
            reader = new BufferedReader(new FileReader("rooms.dat"));
            String state = "none";
            while (reader.ready()) {
                String nextLine = reader.readLine();
                if (nextLine.charAt(0) == '#') {
                    state = nextLine.substring(1);
                    continue;
                }
                switch (state) {
                    case "ROOMS" -> roomStrings.add(nextLine);
                    case "ITEMS" -> itemStrings.add(nextLine);
                    default -> throw new LevelFileException("Invalid or missing level 
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }

        Map<Integer, List<List<String>>> roomItems = new HashMap<>();
        Map<Integer, Map<String, Integer>> roomExits = new HashMap<>();
        for (String roomString: roomStrings) {
            String[] tokens = roomString.split("\\|");
            int id = Integer.parseInt(tokens[0]);
            if (roomExits.containsKey(id)) throw new LevelFileException("Duplicate roor
            if (roomItems.containsKey(id)) throw new LevelFileException("Duplicate roor

            Map<String, Integer> exits = new HashMap<>();
            List<List<String>> items = new ArrayList<>();
            roomExits.put(id, exits);
            roomItems.put(id, items);

            String exitString = tokens[1];

            String[] exitTokens = exitString.split(" ");
            if (!"EXITS".equals(exitTokens[0])) throw new LevelFileException("Malformed

            for (int i = 1; i < exitTokens.length; ++i) {
                String[] exitDirectionTokens = exitTokens[i].split("=");
                exits.put(exitDirectionTokens[0], Integer.parseInt(exitDirectionTokens
            }
        }
        for (Map<String, Integer> exits: roomExits.values()) {
            for (int exitTarget: exits.values()) {
                if (!roomExits.containsKey(exitTarget)) throw new LevelFileException("I
                if (!roomItems.containsKey(exitTarget)) throw new LevelFileException("I
            }
        }

        for (String itemString: itemStrings) {
            String[] tokens = itemString.split("\\|");
            int roomID = Integer.parseInt(tokens[0]);
            String name = tokens[1];
            String[] keywords = tokens[2].toLowerCase().split(" ");

            if (!roomItems.containsKey(roomID)) throw new LevelFileException("Invalid i

            List<String> item = new ArrayList<>();
            item.add(name);
            item.addAll(Arrays.asList(keywords));

            roomItems.get(roomID).add(item);
        }
        playerRoom = 1;
```

```
System.out.println("Starting game");
System.out.println("Commands are:");
System.out.println("\tlook");
System.out.println("\tmove <direction>");
System.out.println("\tget <item>");
System.out.println("\tinventory");
System.out.println("\tquit");

Scanner scanner = new Scanner(System.in);

String input = "";


while (!"quit".equals(input)) {
    input = scanner.nextLine();
    String[] commandTokens = input.split(" ");

    switch (commandTokens[0]) {
        case "look":
            StringBuilder sb = new StringBuilder();

            sb.append("You see an empty room.\n");
            if (roomItems.get(playerRoom).size() > 0) {
                sb.append("You can see some things here:\n");
                for (List<String> item: roomItems.get(playerRoom)) {
                    sb.append("\t");
                    sb.append(item.get(0));
                    sb.append("\n");
                }
            }
            sb.append("\nExits:");
            if (roomExits.get(playerRoom).size() == 0) {
                sb.append(" NONE");
            }
            for (String direction: roomExits.get(playerRoom).keySet()) {
                sb.append(" ");
                sb.append(direction);
            }
            System.out.println(sb);
            break;
        case "move":
            if (commandTokens.length < 2) {
                System.out.println("Move in what direction?");
                break;
            }
            String direction = commandTokens[1].toUpperCase();
            if(!allowedDirections.contains(direction)) {
                System.out.println("Direction " + commandTokens[1] + " not fou
                break;
            }
            int targetRoomID;
            try {
                targetRoomID = roomExits.get(playerRoom).get(direction);
            } catch (NullPointerException ignored) {
                System.out.println("No exit in that direction. Try 'look'ing t
                break;
            }

            playerRoom = targetRoomID;
            System.out.println("You exit to the " + commandTokens[1]);
            break;
        case "get":
            if (commandTokens.length < 2) {
                System.out.println("Get what item?");
                break;
            }
            String[] keywords = Arrays.copyOfRange(commandTokens, 1, commandTo

            List<List<String>> matches = new ArrayList<>();

            for (List<String> item: roomItems.get(playerRoom)) {
                boolean matchingItem = true;
                for (String term: keywords) {
                    if (!item.contains(term.toLowerCase())) {
                        matchingItem = false;
                        break;
                    }
                }
        if (matchingItem) {
            matches.add(item);
        }
}

String searched = String.join(" ", keywords);
if (matches.size() == 0) {
    char firstChar = Character.toLowerCase(searched.charAt(0));

    boolean vowel = 'a' == firstChar ||'e' == firstChar ||'i' == fi

    System.out.println("You can't see " + (vowel?"an ":"a ") + sear
    break;
}
if (matches.size() == 1) {
    roomItems.get(playerRoom).remove(matches.get(0));
    playerInventory.add(matches.get(0));
    System.out.println("You got a " + matches.get(0).get(0));
    break;
}
// more than 1 match
int matchIndex = 0;
System.out.println("More than 1 matching " + searched + " found. Pl
for (int i = 0; i < matches.size(); ++i) {
    System.out.println("\t[" + (i+1) + "] " + matches.get(0).get(0)
}
while (matchIndex == 0) {
    String inputIndex = scanner.nextLine();
    if ("c".equals(inputIndex)) {
        matchIndex = -1;
        continue;
    }
    try {
        matchIndex = Integer.parseInt(inputIndex);
    } catch (NumberFormatException ignored) {
        System.out.println("Input not recognised - enter a number,
    }

    if (matchIndex < 1 || matchIndex > matches.size()) {
        matchIndex = 0;
```

```
                        System.out.println("Number not found, please try again.");
                    }
                }

                if (-1 == matchIndex) {
                    System.out.println("Cancelling");
                    break;
                }

                List<String> matchedItem = matches.get(--matchIndex);

                roomItems.get(playerRoom).remove(matchedItem);
                playerInventory.add(matchedItem);
                System.out.println("You got a " + matchedItem.get(0));
                break;
            case "inventory":
                if (playerInventory.size() == 0) System.out.println("There is noth

                sb = new StringBuilder();

                sb.append("You are carrying the following items:\n");
                for (List<String> item: playerInventory) {
                    sb.append("\t");
                    sb.append(item.get(0));
                    sb.append("\n");
                }

                System.out.println(sb);
                break;
            case "quit":
                break;
            default:
                System.out.println("Command not recognised.");
                break;
        }
    }
}

private static class LevelFileException extends RuntimeException {
    public LevelFileException(String msg) {
        super(msg);
    }
}
```

What I want to cover is something that was completely at odds with the way I was initially taught Object Oriented Programming. It is the idea in that we model every class and their associated behaviours/dependencies based of real world examples. Although it may have worked for smaller assignments, following this mantra across to larger projects can be quite disastrous and cause significant rigidity in design. It is pivotal that you understand and separate the difference of a real world domain and the implementation of it within software, although in some cases we can have similar mappings. An animal in a class isn't an animal in the real world and should not be made to exactly behave and act as one. Hence we should aim to:

- dynamically allow code representations to flourish
- understand how the code representations of our classes should behave

1.3.2 Steps

The strategy that you could follow to solve this God method could be like this:

1. Spend hours coming up with a UML design
2. Crying
3. Tear up previous designs
4. Finally land on a great design
5. Implement design
6. Why is the Sun coming up?

Sometimes rushing straight to the final design can be an issue. For smaller codebases like this, it could be easier, but if we hit large projects with 100,000+ lines of code, it is a non-trivial dilemma. Hence how we want to attack this, is by doing it procedurally:

1. Read the code and highlight large semi-cohesive sections or blocks
2. Abstract these blocks into methods

3. Refactor and check validity of code
4. Repeat 1-3 until you have decently seperated these in the one class.
5. Start delegating high level abstractions and implementations that are responsible for cohesive blocks
6. Step 3 again
7. Start applying design principles as necessary (can be done in step 5)

Obviously this is a very rough process, but you can see how much easier it is, to not only start, but to get to a final design that is designed reasonably well.

1.3.3 Procedural Development

This process is key within the Agile process, we develop the behaviour of our code representation and as we move along the structure will emerge out of it as we add more and more features. In essence we will start with a poorer example or structure initially then hopefully refactor multiple times to get to a better design. The majority of posts ranting about the issues of object oriented design target developers/code that stem from a poor understanding of how to do object oriented design. We as developers must create procedural code and refactor designs **but first we must understand the code we are refactoring**.We don't try and miraculously end up with a perfect solution right away, we chip away as we land on better and better solutions. Just like in algorithm design, rather than attempting to get the optimal solution right away it may be more efficient to start with a naive or suboptimal algorithm then slowly chip away and optimise parts of it.

Whenever refactoring or updating a working solution however, we must always make sure to test and also run our regression tests.