Mid - Mid Notes

MID-SWE_4501_Design_Patterns.pdf

- Strategy Pattern
- Factory Method Pattern
- Decorator Pattern
- Template Method Pattern
- Composite Pattern
- Adapter Pattern
- Singleton Pattern

Finals - Final Notes

Facade Pattern

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method `encode(filename, format)`. After creating such a class and connecting it with the video conversion library, you'll have your first facade.

Pros and Cons

- You can isolate your code from the complexity of a subsystem.
- A facade can become a god object coupled to all classes of an app.

Relations with Other Patterns

- **Facade** defines a new interface for existing objects, whereas **Adapter** tries to make the existing interface usable. *Adapter* usually wraps just one object, while *Facade* works with an entire subsystem of objects.

- **Abstract Factory** can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.

- **Flyweight** shows how to make lots of little objects, whereas **Facade** shows how to make a single object that represents an entire subsystem.

- **Facade** and **Mediator** have similar jobs: they try to organize collaboration between lots of tightly coupled classes.

    - ***Facade*** defines a simplified interface to a subsystem of objects, but it doesn’t introduce any new functionality. The subsystem itself is unaware of the facade. Objects within the subsystem can communicate directly.
    - ***Mediator*** centralizes communication between components of the system. The components only know about the mediator object and don’t communicate directly.
- A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.

- **Facade** is similar to **Proxy** in that both buffer a complex entity and initialize it on its own. Unlike *Facade*, *Proxy* has the same interface as its service object, which makes them interchangeable.

Example

/COMPLEXSYSTEM/Codec.java

```
public interface Codec {
    String getType();
}
```

/COMPLEXSYSTEM/OggCodec.java

```
public class OggCodec implements Codec {
    @Override
    public String getType() {
        return "ogg";
    }
}
```

/COMPLEXSYSTEM/MPEG4Codec.java

```
public class MPEG4Codec implements Codec {
     @Override
    public String getType() {
        return "mp4";
    }
}
```

/COMPLEXSYSTEM/VideoFile.java

```
public class VideoFile {
    private String name;
    private String format;

    public VideoFile(String name) {
        this.name = name;
        this.format = name.substring(name.indexOf(".") + 1);
    }

    public String getFormat() {
        return format;
    }
    public String getName() {
        return name;
    }
}
```

/COMPLEXSYSTEM/CodecFactory.java

```
public class CodecFactory {
    public static Codec getCodec(String format) {
        if (format.equals("mp4")) {
            return new MPEG4Codec();
        } else {
            return new OggCodec();
        }
    }
}
```

/FACADE/VideoConversionFacade.java

```
public class VideoConversionFacade {
   public boolean convertVideo(String filename, String targetFormat) {

        VideoFile videoFile = new VideoFile(filename);
        System.out.println("Loading video file: " + filename);

        // Get the source codec based on the video format
        Codec sourceCodec = CodecFactory.getCodec(videoFile.getFormat());
        System.out.println("Source format: " + sourceCodec.getType());

        // Get the target codec based on the target format
        Codec destinationCodec = CodecFactory.getCodec(targetFormat);
        System.out.println("Target format: " + destinationCodec.getType());

        // Logic for video conversion & audio mixing

        System.out.println("VideoConversionFacade: Conversion completed successfully!");
        return true;
    }
}
```

/App.java

```
public class App {
    public static void main(String[] args) throws Exception {
       // Client code only interacts with the Facade
        VideoConversionFacade converter = new VideoConversionFacade();
        converter.convertVideo("birthday-video.mp4", "ogg");
        converter.convertVideo("lecture.ogg", "mp4");
    }
}
```

Flyweight Pattern

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Intrinsic vs Extrinsic

Let’s take an example of a Word processor.

A Word processor works with **Character** objects. Each character (like 'a', 'b', 'c', etc.) has some information, such as its **content** (the letter itself), and also things like **font, style, and location** on the page.

Now imagine you're working on different documents. They might all use the same letters from 'a' to 'z', but each one might show those letters in different fonts and styles. If we create a new character object every time just to change the font or style, that would take up a lot of memory.

Instead, we can do something smarter.

We **separate the letter itself** (like 'a', 'b', etc.) from its **appearance** (font, style, etc.). This way:

- The letter (like 'a') is the **intrinsic state** – it stays the same and can be shared.
- The font, style, and location are the **extrinsic state** – they can be applied separately, depending on the document.

Solution

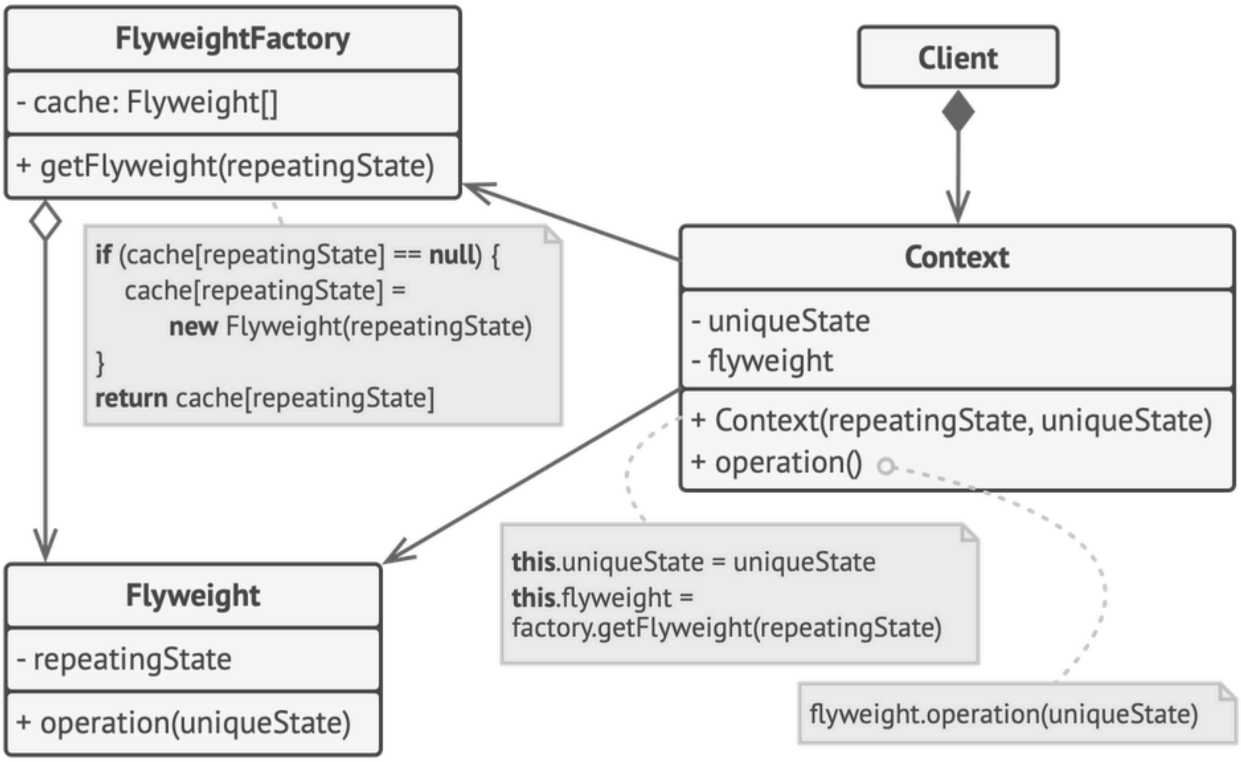
Since there are only a few different letters (just 26 in this case), we can reuse the same character objects and just apply different styles. This saves a lot of memory because we don’t need to create a new object for every styled character – we reuse the core letter and change how it looks.

Flyweight and immutability

Since the same flyweight object can be used in different contexts, you have to make sure that its state can’t be modified. A flyweight should initialize its state just once, via constructor parameters. It shouldn’t expose any setters or public fields to other objects.

Relations with Other Patterns

- You can implement shared leaf nodes of the **Composite** tree as **Flyweights** to save some RAM.
- **Flyweight** shows how to make lots of little objects, whereas **Facade** shows how to make a single object that represents an entire subsystem.
- **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object.


FlyweightFactory
- cache: Flyweight[]
+ getFlyweight(repeatingState)
if (cache[repeatingState] == null) {
cache[repeatingState] =
new Flyweight(repeatingState)
}
return cache[repeatingState]
Client
Context
- uniqueState
- flyweight
+ Context(repeatingState, uniqueState)
+ operation()
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
flyweight.operation(uniqueState)
Flyweight
- repeatingState
+ operation(uniqueState)

Example

//trees/Tree.java

```java
public class Tree {
    // Contains state unique for each tree
   private int x;
    private int y;
    private TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }


    public void display() {
      // Display logic
    }
}
```

//trees/TreeType.java

```java
public class TreeType {
    // Contains state shared by several trees
    private String name;
    private String color;
    private String otherTreeData;
    public TreeType(String name, String color, String otherTreeData) {
        this.name = name;
        this.color = color;
        this.otherTreeData = otherTreeData;
    }
//// Getters for view only, no setters
   public String getName() {
        return name;
    }

    public String getColor() {
        return color;
    }
}
```

//factory/TreeFactory.java

```
public class TreeFactory {
    // This map stores our flyweight objects (the TreeTypes)
    private static Map<String, TreeType> treeTypes = new HashMap<>();

   public static TreeType getTreeType(String name, String color, String data) {
        String key = name + "_" + color;
        treeTypes.put(key, new TreeType(name, color, data));
        return treeTypes.get(key);
    }
}
```

//forest/Forest.java

```
public class Forest {
    private List<Tree> trees = new ArrayList<>();

    public void plantTree(int x, int y, String name, String color, String data) {
        // Use the factory to get a TreeType (either existing or new)
        TreeType type = TreeFactory.getTreeType(name, color, data);

        // Create a new Tree with its unique position and shared TreeType
        Tree tree = new Tree(x, y, type);
        trees.add(tree);
    }
}
```

//App.java

```
public class App {
    static int CANVAS_SIZE = 500;
    static int TREES_TO_DRAW = 1000000;

    public static void main(String[] args) throws Exception {
        Forest forest = new Forest();

        // Plant summer and autumn oak trees
        for (int i = 0; i < TREES_TO_DRAW / 2; i++) {
            forest.plantTree(
                    random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Summer Oak", "green", "Oak texture stub");

            forest.plantTree(
                    random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Autumn Oak", "orange", "Autumn Oak texture stub");
        }
    }
}
```

Proxy Pattern

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Solution

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients.

Relations with Other Patterns

- With **Adapter** you access an existing object via a different interface. With **Proxy**, the interface stays the same. With **Decorator** you access the object via an enhanced interface.

- **Facade** is similar to **Proxy** in that both buffer a complex entity and initialize it on its own. Unlike *Facade*, *Proxy* has the same interface as its service object, which makes them interchangeable.

- **Decorator** and **Proxy** difference is that a *Proxy* usually manages the life cycle of its service object on its own, whereas the composition of *Decorators* is always controlled by the client.

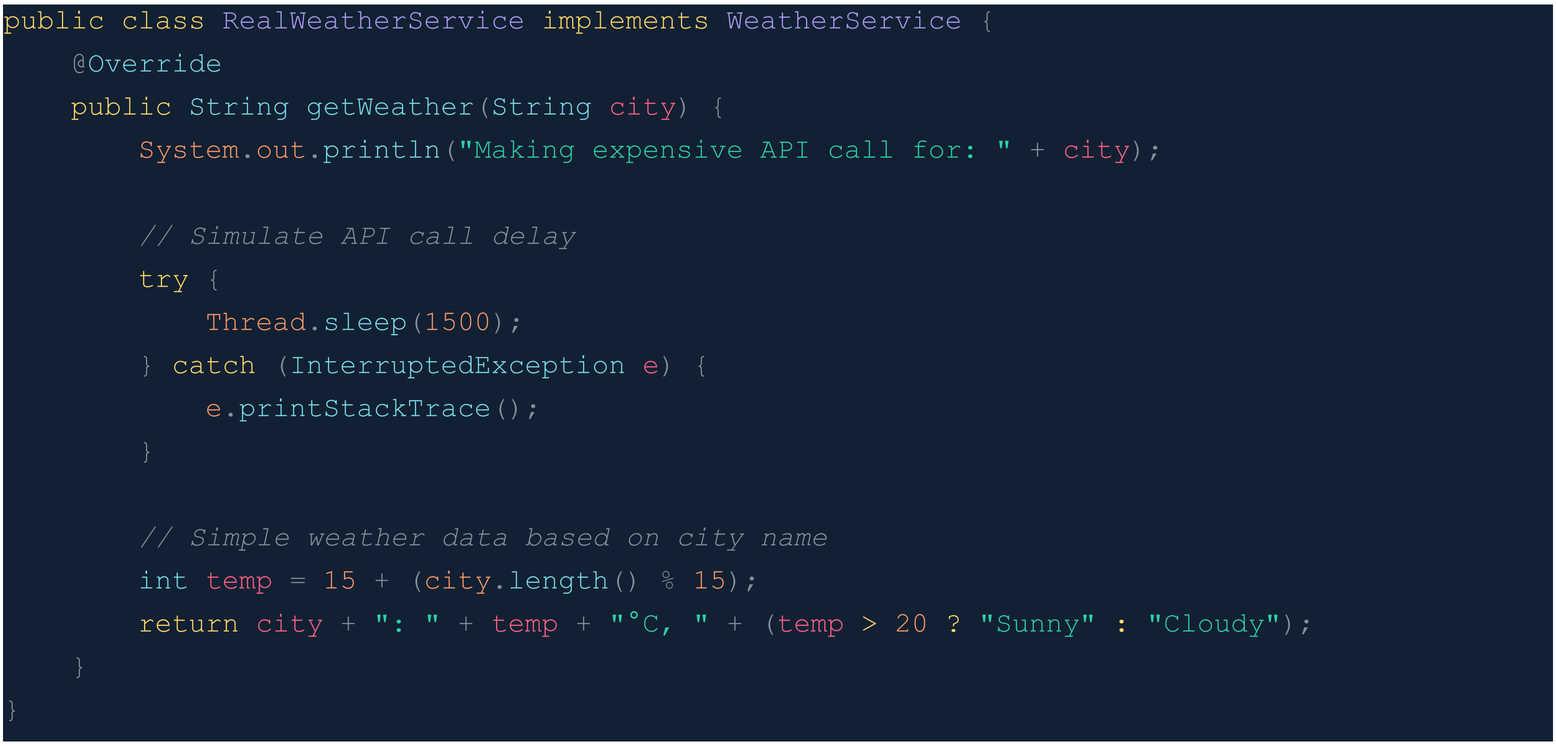
Example

Caching Proxy

//WeatherService/WeatherService.java

```
public interface WeatherService {
    String getWeather(String city);
}
```

//WeatherService/RealWeatherService.java

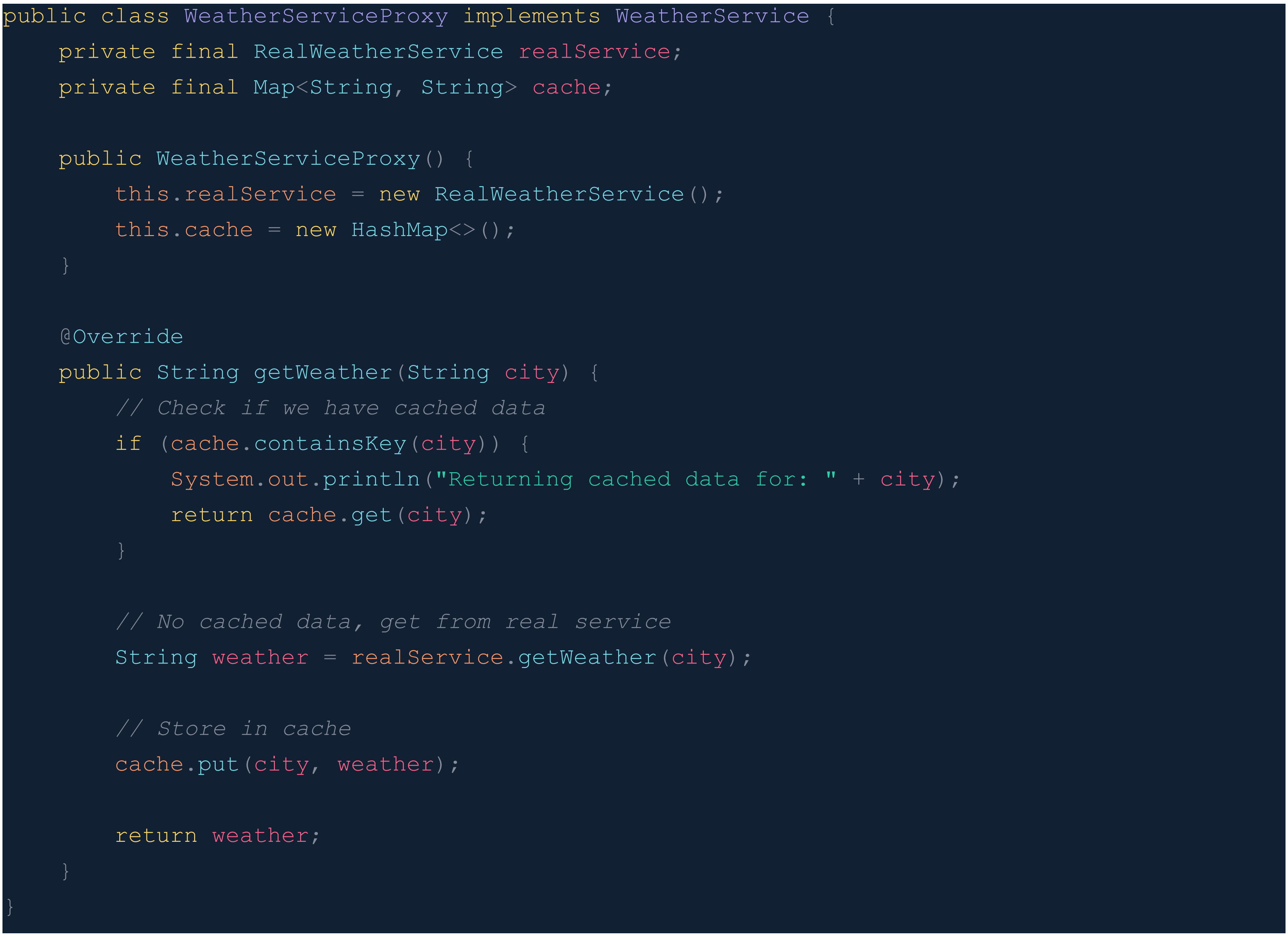


```java
public class RealWeatherService implements WeatherService {
    @Override
    public String getWeather(String city) {
        System.out.println("Making expensive API call for: " + city);

        // Simulate API call delay
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Simple weather data based on city name
        int temp = 15 + (city.length() % 15);
        return city + ": " + temp + "°C, " + (temp > 20 ? "Sunny" : "Cloudy");
    }
}
```

//proxy/WeatherServiceProxy.java



```java
public class WeatherServiceProxy implements WeatherService {
    private final RealWeatherService realService;
    private final Map<String, String> cache;

    public WeatherServiceProxy() {
        this.realService = new RealWeatherService();
        this.cache = new HashMap<>();
    }

    @Override
    public String getWeather(String city) {
        // Check if we have cached data
        if (cache.containsKey(city)) {
            System.out.println("Returning cached data for: " + city);
            return cache.get(city);
        }

        // No cached data, get from real service
        String weather = realService.getWeather(city);

        // Store in cache
        cache.put(city, weather);

        return weather;
    }
}
```

//App.java

```java
public class WeatherApp {
    public static void main(String[] args) throws Exception {
        WeatherServiceProxy weatherProxy = new WeatherServiceProxy();

        // First requests - will call real service
        System.out.println("--- First request for Paris ---");
        System.out.println(weatherProxy.getWeather("Paris"));

        System.out.println("\n--- First request for Tokyo ---");
        System.out.println(weatherProxy.getWeather("Tokyo"));

        // Show current cache status
        weatherProxy.showCache();

        // Second requests - should use cache
        System.out.println("--- Second request for Paris ---");
        System.out.println(weatherProxy.getWeather("Paris"));

        System.out.println("\n--- First request for New York ---");
        System.out.println(weatherProxy.getWeather("New York"));

        System.out.println("\n--- Second request for Tokyo ---");
        System.out.println(weatherProxy.getWeather("Tokyo"));
    }
}
```

This above example is for the Caching Proxy implementation. We also have the Protection proxy implementation **here**. But, I will just add the code below, knowing - there are people too **fuking lazy** to open the github link. **FUK YOU**

Protection Proxy

//internet/Internet.java

```java
public interface Internet {
    void connectTo(String serverHost) throws Exception;
}
```

//internet/RealInternet.java

```java
public class RealInternet implements Internet {
    @Override
    public void connectTo(String serverHost) {
        System.out.println("Connecting to " + serverHost);
    }
}
```

//proxy/InternetProxy.java

```java
public class InternetProxy implements Internet {
    private Internet realInternet;
    private static final String[] RESTRICTED_SITES = {
            "restricted.com",
            "malicious.org",
            "blocked.net"
    };

    public InternetProxy() {
        this.realInternet = new RealInternet();
    }

    @Override
    public void connectTo(String serverHost) throws Exception {
        // Check access before connecting
        for (String site : RESTRICTED_SITES) {
            if (serverHost.contains(site)) {
                throw new Exception("Access Denied: Connection to " + serverHost + " is blocked");
            }
        }

        // If access is allowed, delegate to the real object
        System.out.println("Proxy: Access check passed");
        realInternet.connectTo(serverHost);
    }
}
```

//App.java

```java
public class InternetApp {
    public static void main(String[] args) throws Exception {
        Internet internet = new InternetProxy();

        try {
            internet.connectTo("google.com");
            internet.connectTo("restricted.com");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        try {
            internet.connectTo("example.com/malicious.org");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Command Pattern

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as method arguments, delay or queue the request's execution, and support undoable operations.

Relations with Other Patterns

- **Command**, **Mediator** and **Observer** address various ways of connecting senders and receivers of requests:
    - *Command* establishes unidirectional connections between senders and receivers.
    - *Mediator* eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object.
    - *Observer* lets receivers dynamically subscribe to and unsubscribe from receiving requests.
- You can use **Command** and **Memento** together when implementing “**undo**”. In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed.

- **Command** and **Strategy** may look similar because you can use both to parameterize an object with some action. However, they have very different intentions.
    - You can use *Command* to convert any operation into an object. The operation’s parameters become fields of that object. The conversion lets you defer execution of the operation, queue it, store the history of commands, send commands to remote services, etc.
    - On the other hand, *Strategy* usually describes different ways of doing the same thing, letting you swap these algorithms within a single context class.

- **Prototype** can help when you need to **save copies** of **Commands** into history.

Example

//Commands/Command.java

```java
public abstract class Command {
    protected Editor editor;
    protected String backup;

    public Command(Editor editor) {
        this.editor = editor;
    }

    protected void backup() {
        backup = editor.getText();
    }

    public void undo() {
        editor.setText(backup);
    }
    public abstract boolean execute();
}
```

// Commands/TypeCommand.java

```
public class TypeCommand extends Command {
    private String newText;

    public TypeCommand(Editor editor, String newText) {
        super(editor);
        this.newText = newText;
    }

    @Override
    public boolean execute() {
        backup();
        editor.setText(editor.getText() + newText);
        return true;
    }
}
```

//Commands/CopyCommand.java

```
public class CopyCommand extends Command {
    public CopyCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        String selectedText = editor.selectTextPortion();
        if (selectedText == null || selectedText.isEmpty()) {
            return false;
        }

        editor.setClipboard(selectedText);
        return false;
    }
}
```

//Commands/CommandHistory.java

```
public class CommandHistory {
    private Stack<Command> history = new Stack<>();

    public void push(Command command) {
        history.push(command);
    }

    public Command pop() {
        return history.isEmpty() ? null : history.pop();
```

```java
    }

    public boolean isEmpty() {
        return history.isEmpty();
    }
}
```

//editor/Editor.java

```java
public class Editor {
    private String text = "";
    private String clipboard = "";
    private CommandHistory history = new CommandHistory();

    public void start() { //Initialization logic }

    private void executeCommand(Command command) {
        if (command.execute()) {
            history.push(command);
            System.out.println("Command executed.");
        } else {
            System.out.println("Command could not be executed.");
        }
    }

    private void undo() {
        if (history.isEmpty()) {
            System.out.println("Nothing to undo.");
            return;
        }

        Command command = history.pop();
        if (command != null) {
            command.undo();
            System.out.println("Undo completed.");
        }
    }
}
```

//App.java

```java
public class App {
    public static void main(String[] args) throws Exception {
        Editor editor = new Editor();
        editor.start();
    }
}
```

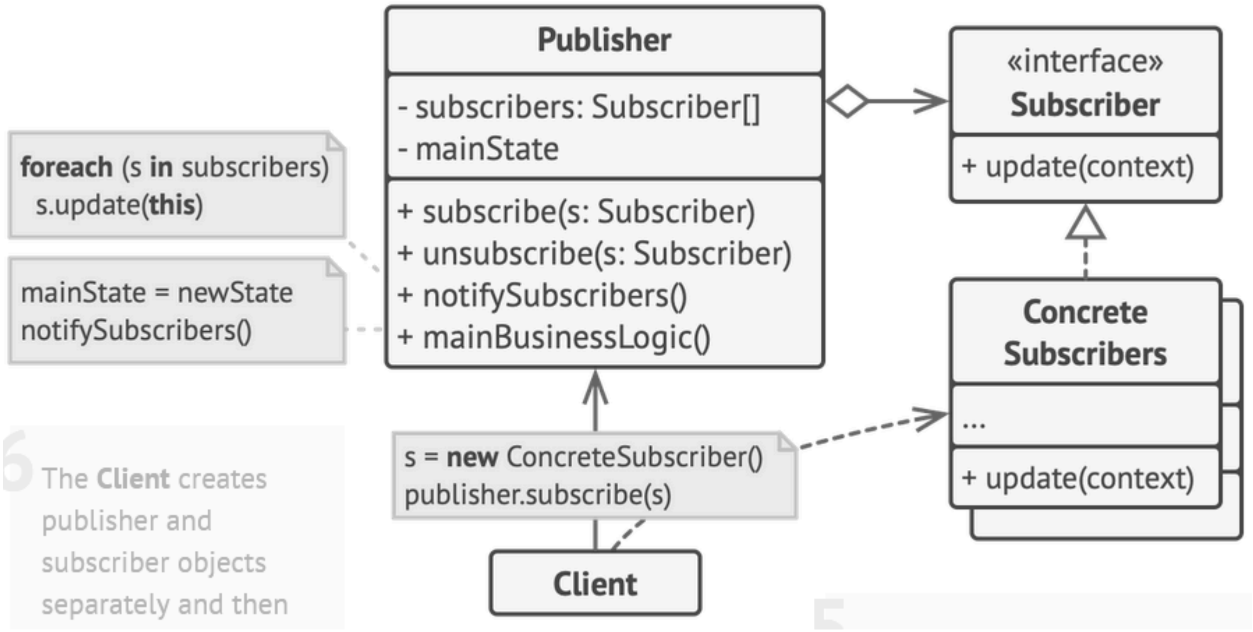
Observer Pattern

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Solution

The object that has some interesting state is often called *subject*, but since it's also going to notify other objects about the changes to its state, we'll call it ***publisher***. All other objects that want to track changes to the publisher's state are called ***subscribers***.

The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.



Relations with Other Patterns

- ***Command*** establishes unidirectional connections between senders and receivers.
- ***Mediator*** eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object.
- The difference between **Mediator** and **Observer** is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that.

Example

// CustomEvent/FileEvent.java

```
public class FileEvent {
    private final String eventType;
    private final File file;
    private final String timestamp;

    public FileEvent(String eventType, File file) {
        this.eventType = eventType;
        this.file = file;
        this.timestamp = java.time.LocalDateTime.now().toString();
    }
}
```

// listener/FileEventListener.java

```
public interface FileEventListener {
    void onFileEvent(FileEvent event);
}
```

// listener/EmailNotificationListener.java

```
public class EmailNotificationListener implements FileEventListener {
    private final String email;

    public EmailNotificationListener(String email) {
        this.email = email;
    }

    @Override
    public void onFileEvent(FileEvent event) {
        System.out.println("Email to " + email + ": File " + event.getFile().getName()
                + " was " + event.getEventType() + " at " + event.getTimestamp());
    }
}
```

// publisher/EventManager.java

```
public class EventManager {
    private final Map<String, List<FileEventListener>> listeners = new HashMap<>();

    public EventManager(String... operations) {
        for (String operation : operations) {
            // one array for each type of operation we want to listen to
            this.listeners.put(operation, new ArrayList<>());
        }
    } // Write this constructor at least

    // If can’t write these 3, just write names and comment the logic
    public void subscribe(String eventType, FileEventListener listener) {
        // Get list for that type of users/subscribers
        List<FileEventListener> users = listeners.get(eventType);
        if (users != null) {
            users.add(listener);
        }
    }
    public void unsubscribe(String eventType, FileEventListener listener) {
        // Get list for that type of users/subscribers
        List<FileEventListener> users = listeners.get(eventType);
        if (users != null) {
            users.remove(listener);
        }
    }
    public void notify(String eventType, File file) {
        // Get list for that type of users/subscribers
        List<FileEventListener> users = listeners.get(eventType);
        if (users != null) {
            FileEvent event = new FileEvent(eventType, file);
            for (FileEventListener listener : users) {
                listener.onFileEvent(event);
            }
        }
    }
}
```

// editor/FileEditor.java

```
public class FileEditor {
    public final EventManager events;
    private File file;

    public FileEditor() {
        this.events = new EventManager("open", "save", "close");
    }

    public void openFile(String filePath) {
        this.file = new File(filePath);
        System.out.println("Opening file: " + filePath);
        events.notify("open", file);
    }

    public void saveFile() throws Exception {
        if (this.file != null) {
            System.out.println("Saving changes to file: " + file.getName());
            events.notify("save", file);
        } else {
            throw new Exception("Please open a file first.");
        }
    }

    public void closeFile() throws Exception {
        if (this.file != null) {
            System.out.println("Closing file: " + file.getName());
            events.notify("close", file);
            this.file = null;
        } else {
            throw new Exception("No file is currently open.");
        }
    }
}
```

//App.java

```
public static void main(String[] args) throws Exception {
    FileEditor editor = new FileEditor();
    Scanner scanner = new Scanner(System.in);

    editor.events.subscribe("save", new EmailNotificationListener("admin@example.com"));
    editor.saveFile();
}
```

Prototype Pattern

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Solution

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single `clone` method.

Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.

- **Prototype** can help when you need to save copies of **Commands** into history.

- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.

- **Prototype** isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, *Prototype* requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn't require an initialization step.

- Sometimes **Prototype** can be a simpler alternative to **Memento**. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

Example

//Shapes/Shape.java

```
public abstract class Shape {
    public int x;
    public int y;
    public String color;

    public Shape() {
    }

    public Shape(Shape target) {
        if (target != null) {
            this.x = target.x;
            this.y = target.y;
            this.color = target.color;
        }
    }

    public abstract Shape clone();
}
```

//Shapes/Circle.java

```
public class Circle extends Shape {
    public int radius;

    public Circle() {
    }

    public Circle(Circle target) {
        super(target);
        if (target != null) {
            this.radius = target.radius;
        }
    }

    @Override
    public Shape clone() {
        return new Circle(this);
    }
}
```

//App.java

```
public class App {
    public static void main(String[] args) throws Exception {
        List<Shape> shapes = new ArrayList<>();
        List<Shape> shapesCopy = new ArrayList<>();

        Circle circle = new Circle();
        circle.x = 10;
        circle.y = 20;
        circle.radius = 15;
        circle.color = "red";
        shapes.add(circle);

        Circle anotherCircle = (Circle) circle.clone();
        shapes.add(anotherCircle);
    }
}
```