Lecture 10

Design Patterns (part 2)

Topics covered

- ✧ Structural design patterns
  - Bridge
  - Facade
  - Flyweight
- ✧ Behavioral design patterns
  - Iterator
  - Observer
  - Memento
  - State

Structural Design Patterns

Bridge pattern

- ✧ Decouples the functional abstraction from the implementation so that the two can vary independently
- ✧ **Usage:**
  - When you don't want a permanent binding between the functional abstraction and its implementation.
  - When both the functional abstraction and its implementation need to be extended using sub-classes.
  - It is mostly used in those places where changes made in the implementation does not affect the clients.

Bridge pattern motivation example (problem)

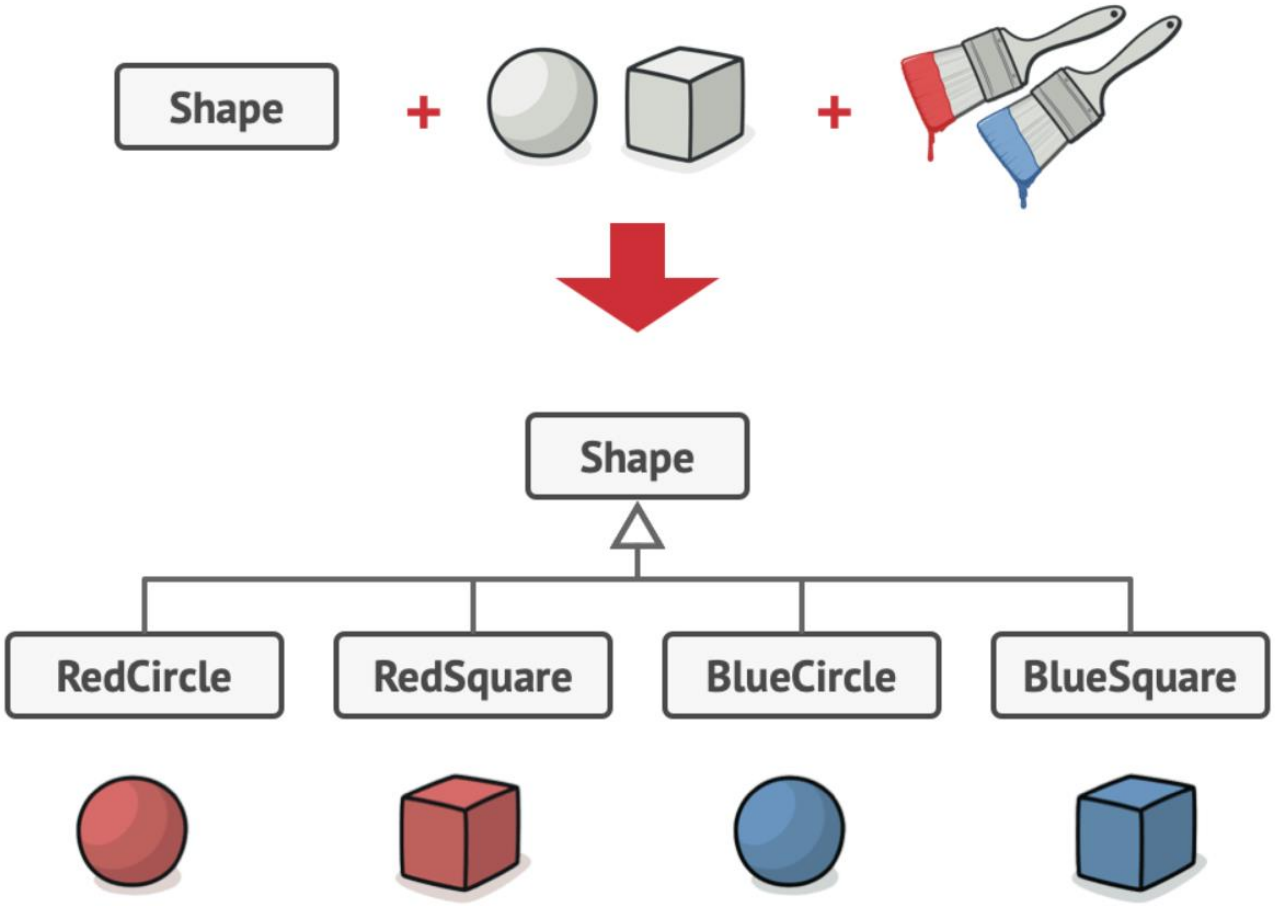
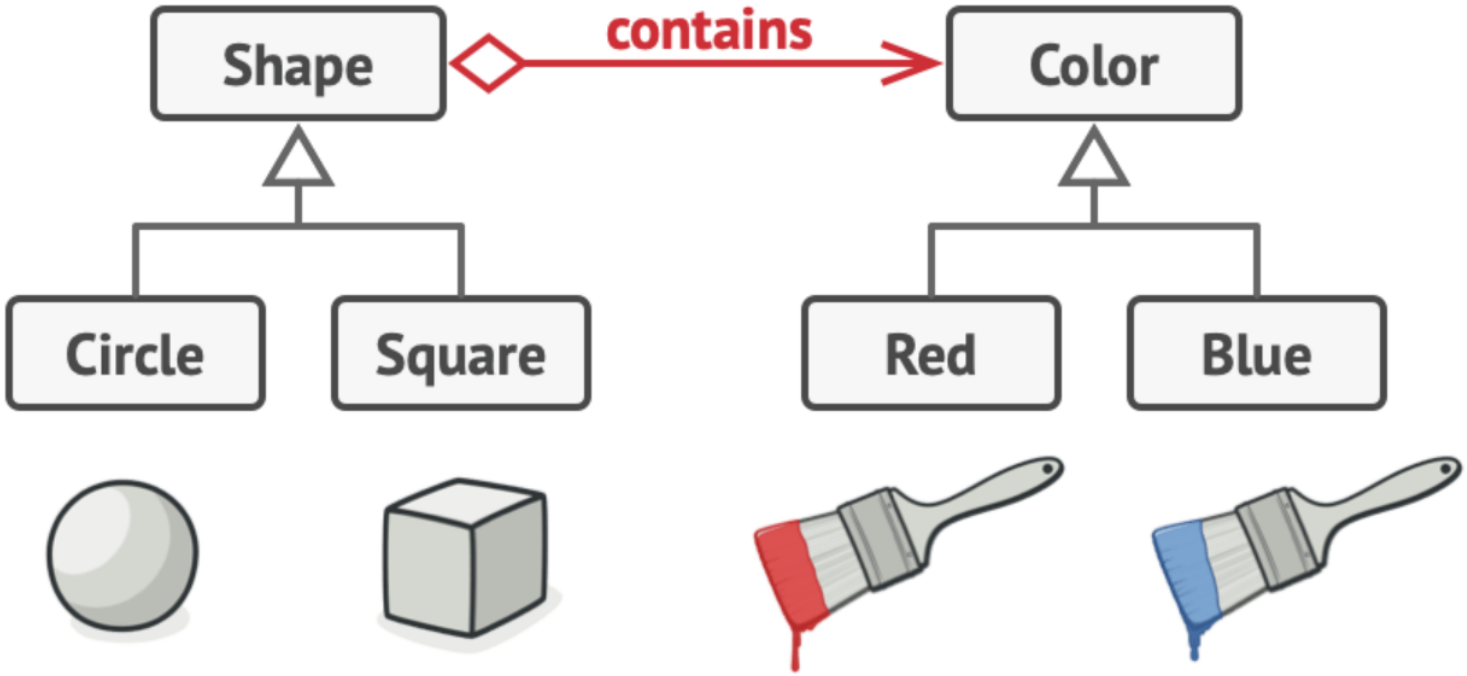


Image Credit: refactoring.guru

Bridge pattern

- **General solution:** switching from inheritance to object composition.



Bridge pattern code example (1)

```
interface Color {
    void applyColor();
}

class Red implements Color {
    @Override
    public void applyColor() {
        System.out.println("Applying red color");
    }
}

class Blue implements Color {
    @Override
    public void applyColor() {
        System.out.println("Applying blue color");
    }

}
```

Bridge pattern code example (2)

```
abstract class Shape {
    protected Color color;

    public Shape(Color color) {
        this.color = color;
    }

    abstract void draw();
}
```

Bridge pattern code example (3)

```
class Circle extends Shape {
    public Circle(Color color) { super(color); }

    @Override
    void draw() {
        System.out.print("Drawing a circle. ");
        color.applyColor();
    }
}

class Square extends Shape {
    public Square(Color color) { super(color); }

    @Override
    void draw() {
        System.out.print("Drawing a square. ");
        color.applyColor();
    }
}
```

Benefits of bridge pattern

✧ The Bridge pattern:

- decouples abstraction and implementation
- promotes flexibility and maintainability
- helps avoiding class explosion
- promotes code reuse

Facade pattern

- Façade: the front side of a building.
- Provides a unified and simplified interface to a set of interfaces in a subsystem
  - Hiding the complexities of the subsystem from the client
- Describes a higher-level interface that makes the sub-system easier to use
- **Usage:**
  - When you want to provide simple interface to a complex sub-system.
  - When several dependencies exist between clients and the implementation classes of an abstraction.

Facade pattern example: Video Conversion

- ✧ Imagine you're building an app that uploads short funny videos to social media.
- ✧ To handle video conversion, you could use a professional video conversion library with numerous features.
  - However, your app only needs a simple method to encode videos.
- ➢ You create a `VideoConversionFacade` class that encapsulates only the necessary functionalities.
  - The facade simplifies interaction with the complex video conversion library, allowing you to focus on what matters: encoding videos.

Facade pattern example: Report Generator

- ✧ Creating a report involves multiple steps:
  - Adding a header, footer, data rows, formatting, and writing the report in various formats (e.g., PDF, HTML)
- ✧ Solution: Instead of dealing directly with all the framework classes, you create a `ReportGeneratorFacade`.
- ✧ The facade coordinates the steps, ensuring that the client code interacts with a simplified interface, abstracting away the complexity of the subsystem.

Facade pattern example: Report Generator

```
class Report {
    private ReportHeader header;
    private ReportData data;
    private ReportFooter footer;
    // getter & setter methods
}

class ReportHeader { }
class ReportFooter { }
class ReportData { }
enum ReportType { PDF, HTML }
```

Facade pattern example: Report Generator

```
class Report {
    private ReportHeader header;
    private ReportData data;
    private ReportFooter footer;
    // getter & setter methods
}

class ReportHeader { }
class ReportFooter { }
class ReportData { }
enum ReportType { PDF, HTML }
```

Facade pattern example: Report Generator

```
class ReportWriter {
    public void writeHtmlReport(Report report, String location) {
        System.out.println("HTML Report written");
    }

    public void writePdfReport(Report report, String location) {
        System.out.println("PDF Report written");
    }

}
```

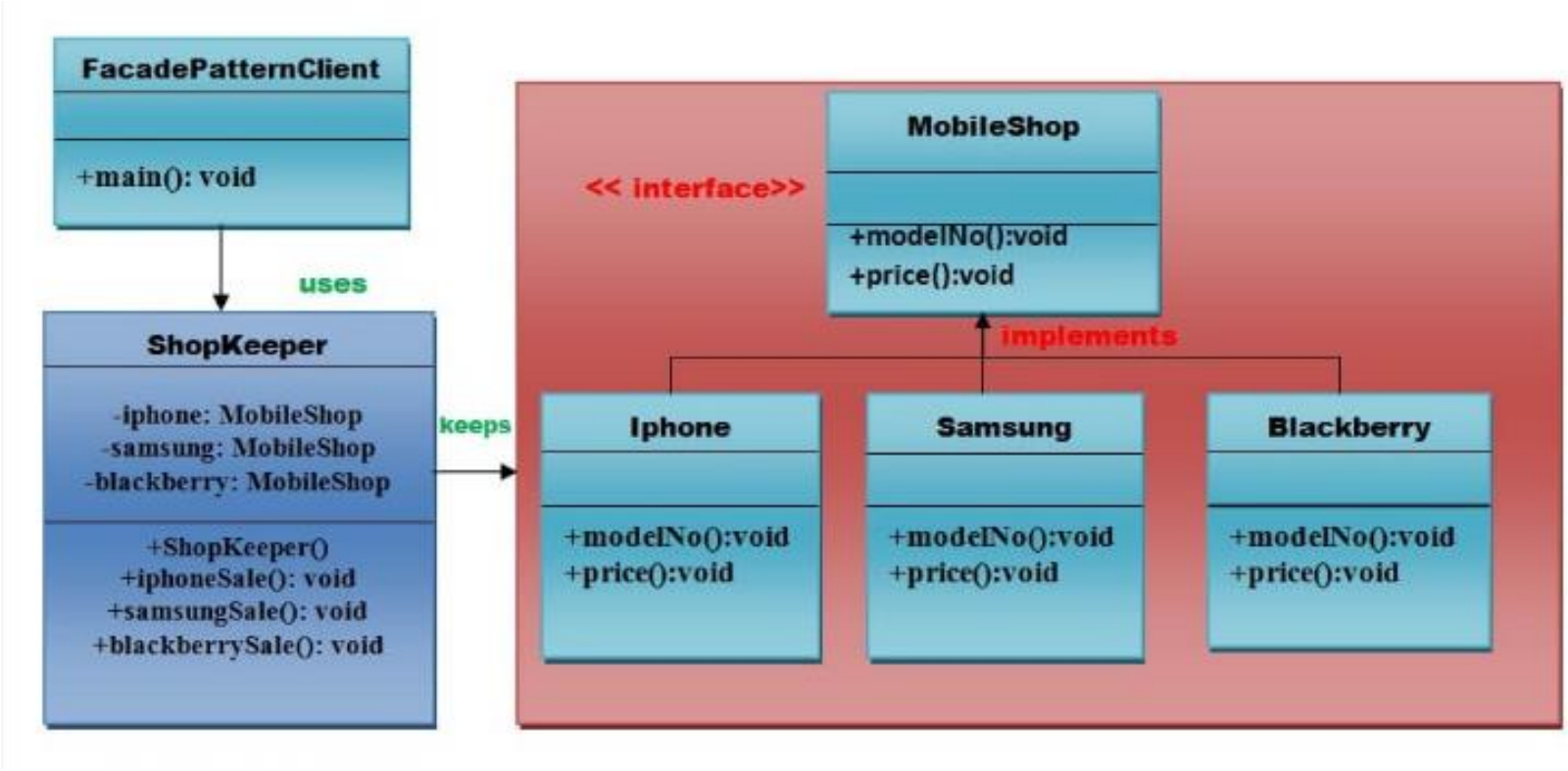
Facade pattern example: Report Generator

```
class ReportGeneratorFacade {
    private ReportWriter reportWriter = new ReportWriter();
    public void generateReport(Report report,
                               ReportType type,
                               String location) {
        switch (type) {
            case PDF:
                reportWriter.writePdfReport(report, location);
                break;
            case HTML:
                reportWriter.writeHtmlReport(report, location);
                break;
            default:
                System.out.println("Unsupported report type");
        }
    }

}
```

Facade pattern example: Report Generator

```
class ReportGeneratorFacade {
    private ReportWriter reportWriter = new ReportWriter();
    public void generateReport(Report report,
                               ReportType type,
                               String location) {
        switch (type) {
            case PDF:
                reportWriter.writePdfReport(report, location);
                break;
            case HTML:
                reportWriter.writeHtmlReport(report, location);
                break;
            default:
                System.out.println("Unsupported report type");
        }
    }

}
```

Facade pattern example: Mobile Shop



Flyweight pattern

- Reuses existing similar kind of objects by storing them and create new object when no matching object is found
- **Advantages:**
  - It reduces the number of objects
  - It reduces the amount of memory and storage devices required if the objects are persisted
- **Usage:**
  - When an application uses number of objects
  - When the storage cost is high because of the quantity of objects
  - When the application does not depend on object identity

Flyweight pattern example: Text Editor

```
class Character {
    private char character;

    public Character(char character) {
        this.character = character;
    }

    public void display() {
        System.out.println("Character: " + character);
    }
}
```

Flyweight pattern example: Text Editor

```
// Flyweight Factory
class CharacterFactory {
    private Map<Character, Character>
                    characterCache = new HashMap<>();

    public Character getCharacter(char c) {
        if (!characterCache.containsKey(c)) {
            characterCache.put(c, new Character(c));
        }
        return characterCache.get(c);
    }

}
```

Flyweight pattern example: Text Editor

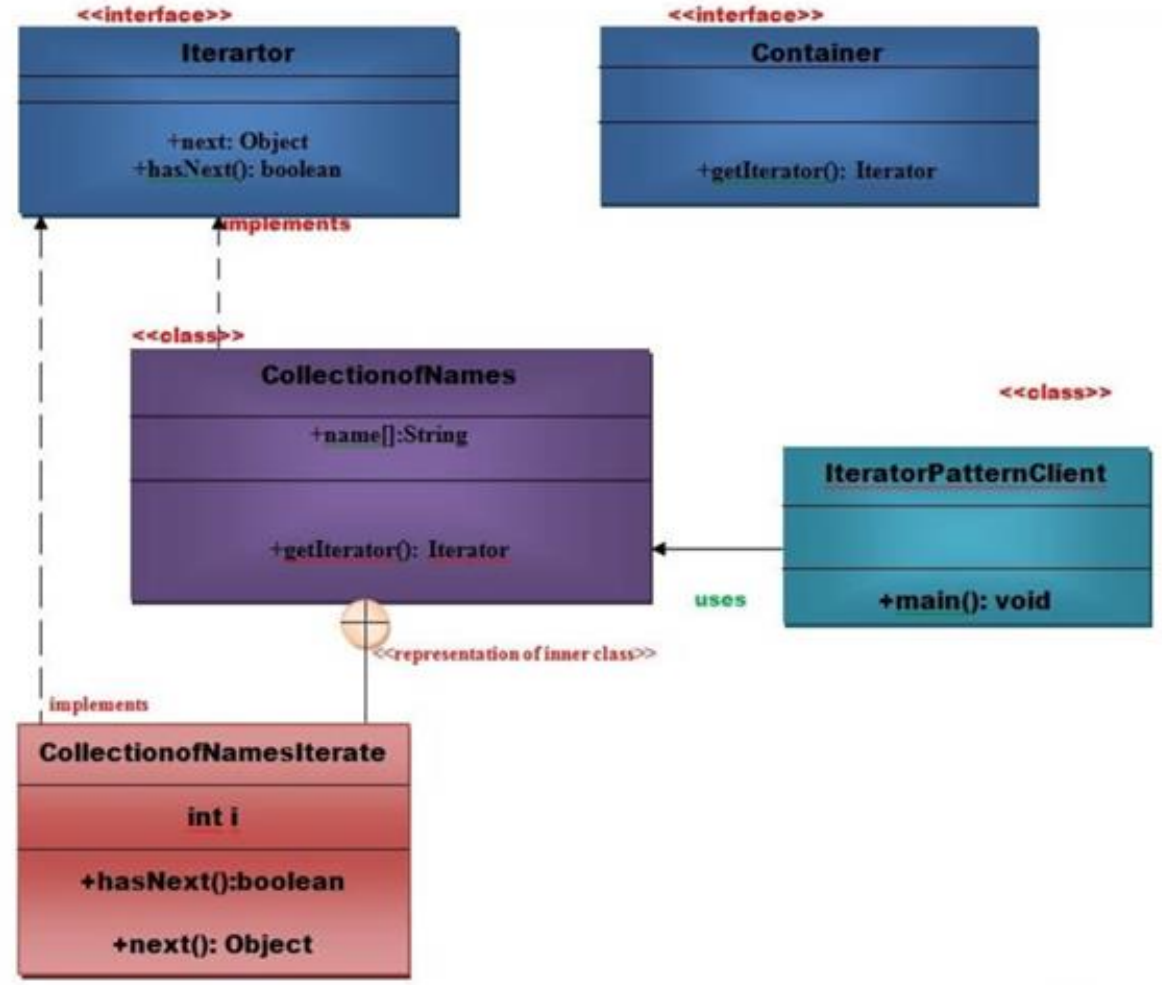
```
CharacterFactory characterFactory = new CharacterFactory();

// Create and display characters
Character a = characterFactory.getCharacter('A');
Character b = characterFactory.getCharacter('B');
// Reusing existing 'A'
Character aAgain = characterFactory.getCharacter('A');

a.display("Arial");
b.display("Times New Roman");

aAgain.display("Calibri"); // Same 'A' as before
```

Behavioral Design Patterns

Iterator pattern

- To access the elements of an aggregate object sequentially without exposing its underlying implementation
- **Usage:**
  - When you want to access a collection of objects without exposing its internal representation.
  - When there are multiple traversals of objects need to be supported in the collection.

Iterator pattern



Iterator pattern example: Radio Channels

```
public class Channel {
    private double frequency;
    private ChannelTypeEnum type;

    // constructor, getter & setters

    @Override
    public String toString() {
        return "Frequency=" + frequency + ", Type=" + type;
    }

}
```

Iterator pattern example: Radio Channels

```
public interface ChannelCollection {
    void addChannel(Channel channel);
    void removeChannel(Channel channel);
    ChannelIterator iterator(ChannelTypeEnum type);
}

public interface ChannelIterator {
    boolean hasNext();
    Channel next();
}
```

Iterator pattern example: Radio Channels

```
public class ChannelCollectionImpl implements ChannelCollection {
    private List<Channel> channels = new ArrayList<>();
    @Override
    public void addChannel(Channel channel) {
        channels.add(channel);
    }
    @Override
    public void removeChannel(Channel channel) {
        channels.remove(channel);
    }
    @Override
    public ChannelIterator iterator(ChannelTypeEnum type) {
        return new ChannelIteratorImpl(type, channels);
    }

}
```

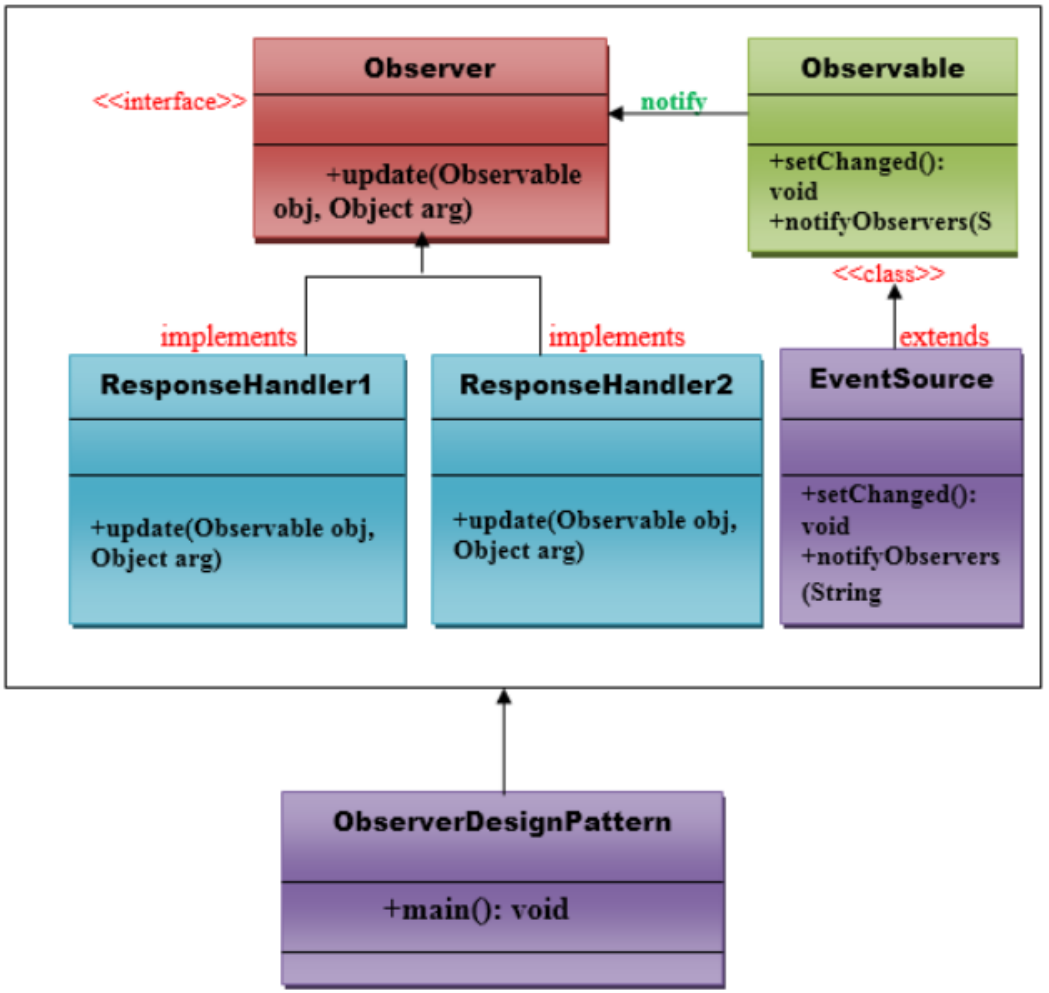
Iterator pattern example: Radio Channels

```
ChannelCollection channelCollection = new ChannelCollectionImpl();
channelCollection.addChannel(new Channel(98.5, ENGLISH));
channelCollection.addChannel(new Channel(102.3, HINDI));
channelCollection.addChannel(new Channel(91.7, FRENCH));

ChannelIterator englishIterator =
        channelCollection.iterator(ChannelTypeEnum.ENGLISH);
while (englishIterator.hasNext()) {
    System.out.println(englishIterator.next());
}
```

Observer pattern

- Defines a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically.
- The pattern facilitates communication between objects: *Observable* and *Observers*
- **Usage:**
  - When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
  - When the framework we writes and needs to be enhanced in future with new observers with minimal changes.

Observer pattern



Observer pattern example: NewsAgency

```
public class NewsAgency {
    private String news;
    private List<Channel> channels = new ArrayList<>();

    public void addObserver(Channel channel) {...}

    public void removeObserver(Channel channel) {...}

    public void setNews(String news) {
        this.news = news;
        for (Channel channel : this.channels) {
            channel.update(this.news);
        }
    }

}
```

Observer pattern example: NewsAgency

```
public interface Channel {
    void update(Object news);
}

public class NewsChannel implements Channel {
    private String news;

    @Override
    public void update(Object news) {
        this.setNews((String) news);
    }

    // getters & setters

}
```

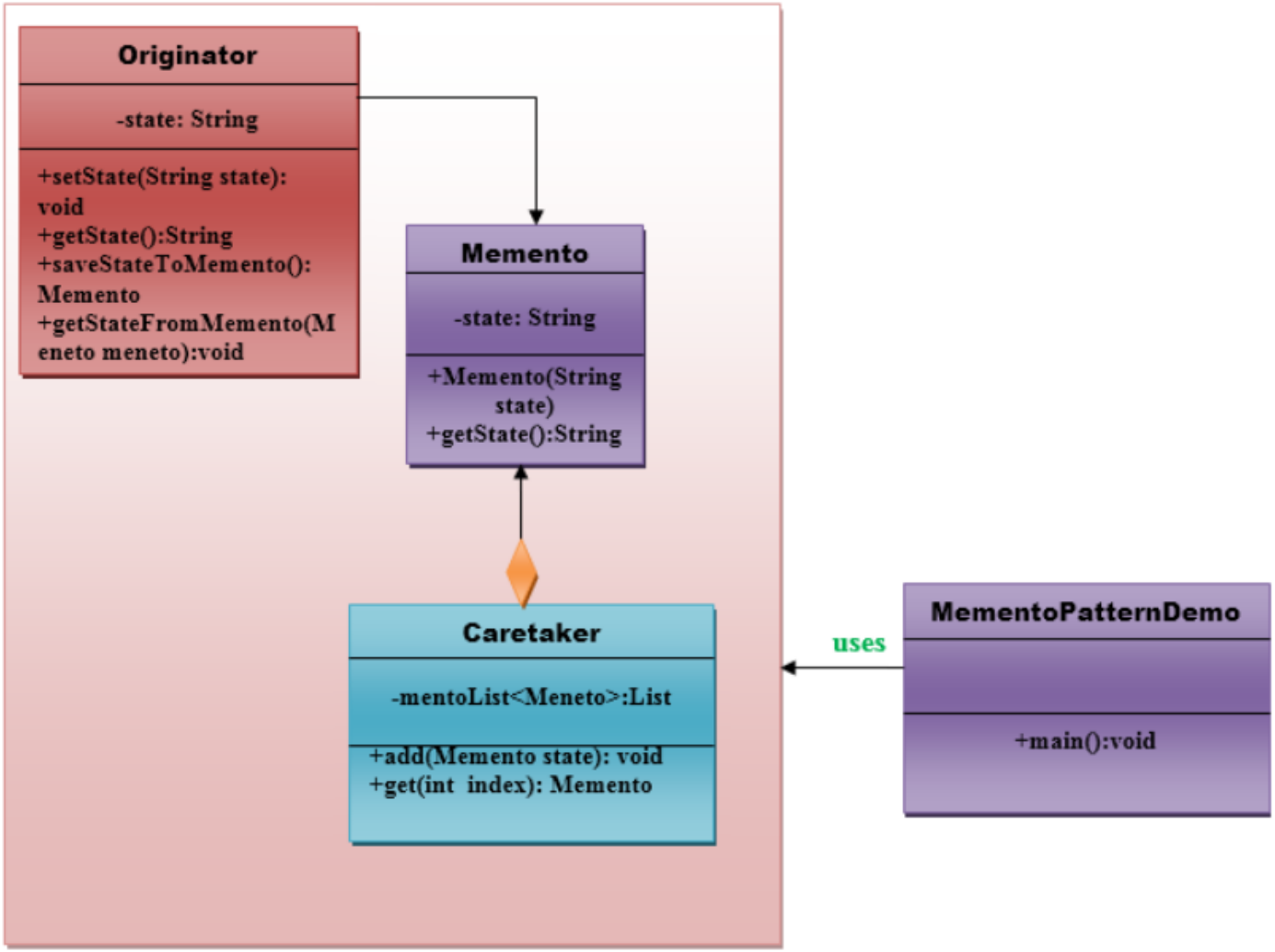
Observer pattern example: NewsAgency

```
NewsAgency observable = new NewsAgency();
NewsChannel observer = new NewsChannel();

observable.addObserver(observer);
observable.setNews("Breaking news: Earthquake reported!");

// Verify that the observer received the updated news
assertEquals(observer.getNews(),
        "Breaking news: Earthquake reported!");
```

Memento pattern

- ✧ Making snapshots (states) of an object and restores the object to its previous state
  - Must do this without violating Encapsulation
- ✧ Three main components:
  - **Originator:** the object whose state needs to be saved
  - **Memento:** he object that holds the saved state. It should expose as little information as possible to the outside world.
  - **Caretaker:** The object responsible for triggering the save and restore operations. It keeps track of the mementos

Memento pattern example



Memento pattern

✧ Advantages:

- It preserves encapsulation boundaries
- It simplifies the originator

✧ Usage:

- It is used in Undo and Redo operations in most software.
- It is also used in database transactions.

Memento pattern example: TextWindow

- **Goal:** save the state of `TextWindow` and restore it when needed.

```
// Originator: Represents the text window
public class TextWindow {
    private StringBuilder currentText = new StringBuilder();
    public void addText(String text) {
        currentText.append(text);
    }
    public String getCurrentText() {
        return currentText.toString();
    }
    // Create a memento with the current state
    public TextMemento save() {
        return new TextMemento(currentText.toString());
    }
    // Restore the state from a memento
    public void restore(TextMemento memento) {
        currentText = new StringBuilder(memento.getState());
    }
}
```

Memento pattern example: TextWindow

```
// Memento: Represents the saved state
public class TextMemento {
    private final String state;

    public TextMemento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}
```

Memento pattern example: TextWindow

```
// Memento: Represents the saved state
public class TextMemento {
    private final String state;

    public TextMemento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}
```

Memento pattern example: TextWindow

```
// Caretaker: Manages mementos
public class TextEditor {
    private TextMemento savedState;

    public void saveState(TextWindow textWindow) {
        savedState = textWindow.save();
    }

    public void undo(TextWindow textWindow) {
        textWindow.restore(savedState);
    }
}
```

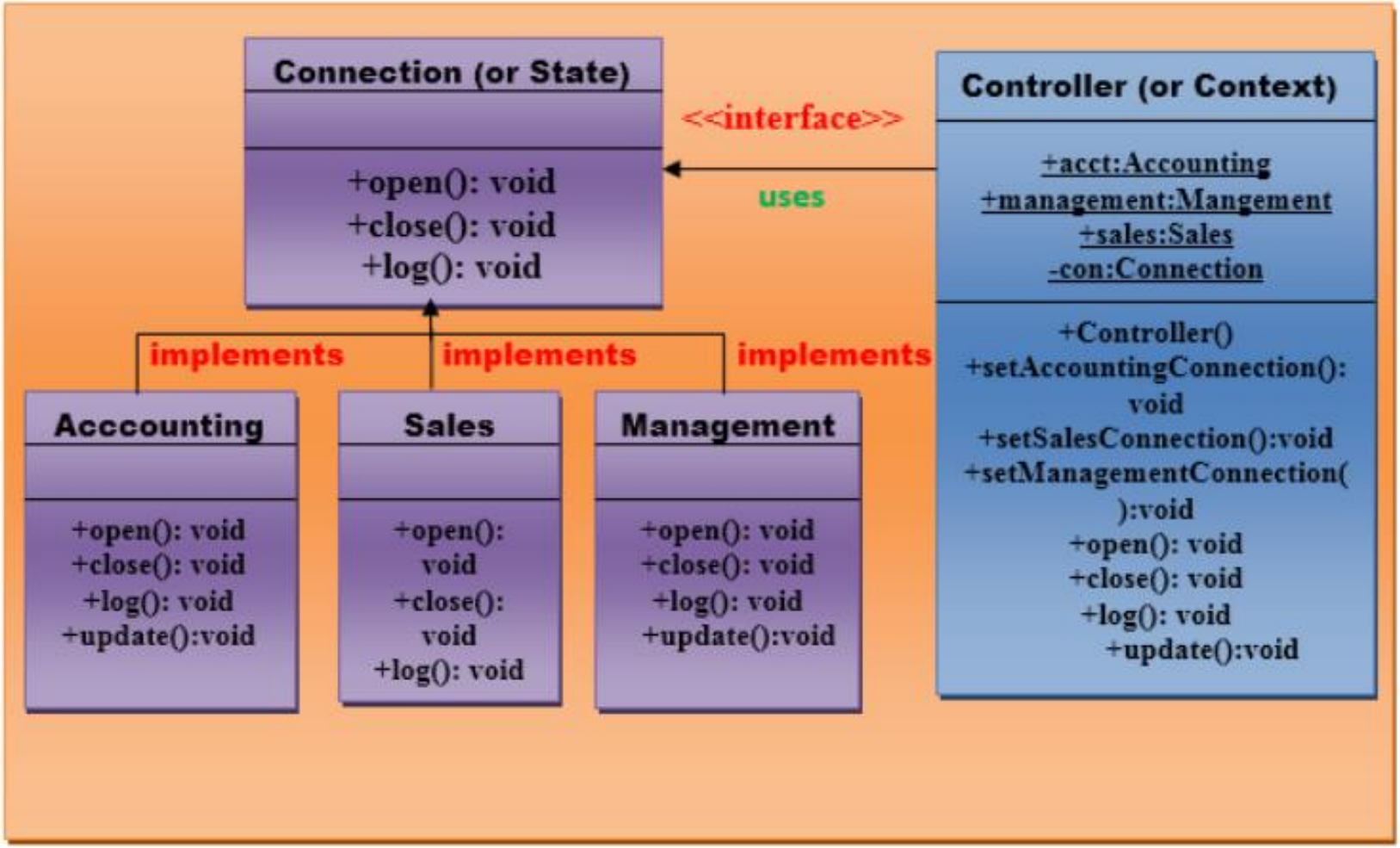
Memento pattern example: TextWindow

```
public static void main(String[] args) {
    TextWindow textWindow = new TextWindow();
    TextEditor caretaker = new TextEditor();

    textWindow.addText("Hello, ");
    caretaker.saveState(textWindow); // Save the state

    textWindow.addText("world!");
    System.out.println("Current text: " +
            textWindow.getCurrentText());

    caretaker.undo(textWindow); // Restore to the saved state
    System.out.println("Restored text: " +
            textWindow.getCurrentText());

}
```

State pattern

- The class behavior changes based on its state
- We create objects which represent various states and a context object whose behavior varies as its state object changes
  - Encapsulating the object's behavior within different state objects
  - Switches between these state objects as current state
- **Usage:**
  - When the behavior of object depends on its state and it must be able to change its behavior at runtime according to the new state.
  - It is used when the operations have large, multipart conditional statements that depend on the state of an object.

State pattern example: Connection states



State pattern example: Document

- We model a document with different states: *Draft*, *Moderation*, and *Published*.

```
class Document {
    private DocumentState currentState;

    public void publish() {
        currentState.publish(this);
    }

    public void changeState(DocumentState state) {
        this.currentState = state;
    }

    // Other methods for editing, moderation, etc.

}
```

State pattern example: Document

```
interface DocumentState {
    void publish(Document document);

}


class DraftState implements DocumentState {
    @Override
    public void publish(Document document) {
        // Move to moderation state
        document.setCurrentState(new ModerationState());
    }

}
```

State pattern example: Document

```
interface DocumentState {
    void publish(Document document);
}


class DraftState implements DocumentState {
    ...
}

class ModerationState implements DocumentState {
    ...
}

class PublishedState implements DocumentState {
    ...
}
```