

COMP2100/6442

Software Design Methodologies / Software Construction

Design Patterns

Bernardo Pereira Nunes and
Sergio J. Rodríguez Méndez

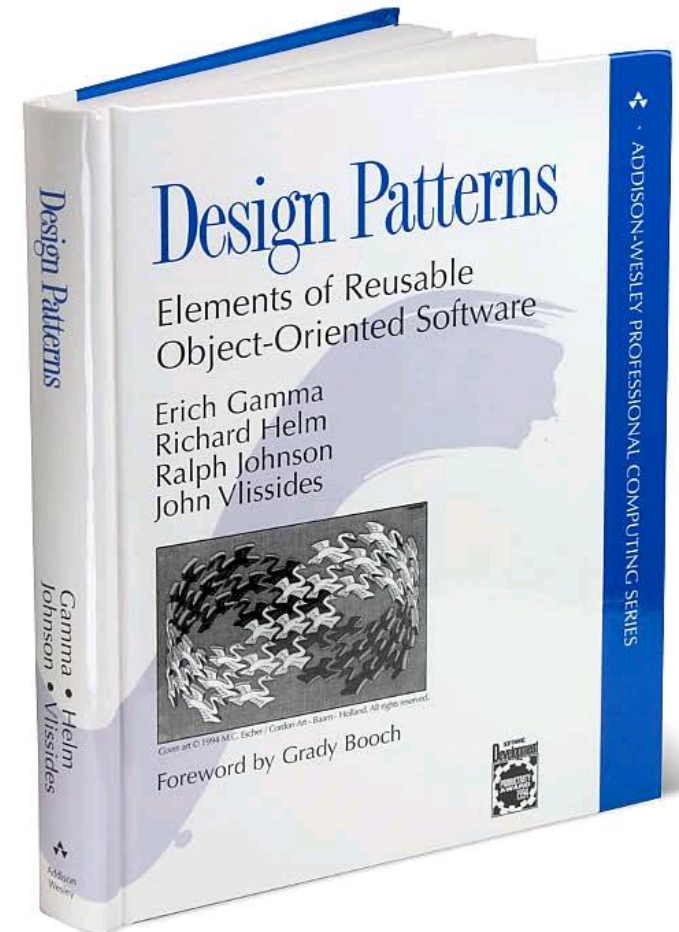


Outline

- > Design Patterns (DP)
 - >> History / Motivation
 - >> DP Classification
 - > Behavioural
 - > Structural
 - > Creational
 - >> Exercises

Design Patterns (Gang of Four [GoF])

“Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. **Your design should be specific to the problem at hand but also general enough to address future problems and requirements.** You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get “right” the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.” [GoF]



Design Patterns

... are repeatable solutions to reoccurring problems in software design.

“Reuse solutions that have worked ... in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts.” [GoF]

“The goal is to capture design experience in a form that people can use effectively.” [GoF]



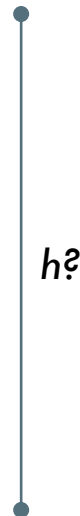
Tea? Hot Chocolate?



Designing with patterns started with **urban planning** and **building architecture**.

In the 70's, **Christopher Alexander** created a **pattern language (253 patterns)** that ordinary people could use **to create more enjoyable buildings and public spaces**.

Each pattern provided a general solution to a reoccurring design problem.





How do patterns work in general?

Let's see an example of a pattern proposed by Chris.

> Consider the pattern named Low Sill, a pattern for determining the height of a windowsill.

Context

>> Windows are being planned for a wall

Problem

>> **How high should the windowsill be from the floor?** If it is too high, it cuts you off from the outside world. If it is too low, it could be mistaken for a door and is potentially unsafe.

Solution

Design the windowsill to be 12 to 14 inches from the floor. On upper floors, for safety, make them higher, around 20 inches. The primary function of a window is to connect building occupants to the outside world. The link is more meaningful when both the ground and horizon are visible when standing a foot or two away from the window.



What if you are designing a window for a jail cell?



- > **Design** is about balancing conflicting forces or constraints
- > **Engineering** provides general solutions at a medium level of abstraction
- > **Design Patterns** do not give exact answers (i.e., precise measurements), but still, they are more concrete than abstract principles.

What is the optimal height?

- >>> It is the value that best balances the conflicting forces.
- >>> ~~Connection to the outside world~~ **and Safety**
(in the case of jails)

Design Patterns

- > **Designing** is probably the most **challenging** activity in the software development life cycle
- > There is **no algorithm for deriving abstract solution** models from requirements
- > A design pattern is a **problem-solution pair**
- > Design patterns provide a **mapping from a specific design problem to a generic solution**

Design Patterns

Design Problem

How can I ensure a class only has one instance?

Generic Solution

Singleton

```
class Singleton {
    -instance: Singleton
    -Singleton()
    *instance(): Singleton

    public synchronized static Singleton instance() {
        if( instance == null ) {
            instance = new Singleton( );
        }
        return instance;
    }
}
```

How can I make dependent objects aware of state changes in a subject object without using polling?

Observer

```
class Observable {
    -obs: Vector
    -changed: boolean

    addObserver(Observer o)
    deleteObserver(Observer o)
    setChange()
    notifyObservers()
    notifyObservers(Object arg)
}

class Observer {
    <<interface>>
    update(Observable obs, Object arg)
}

public void notifyObservers(Object arg) {
    if (!changed) return;
    changed = false;
    for each observer o
        o.update(this, arg);
}
```

problem-solution pair | mapping from a specific design problem to a generic solution

Design Patterns > Advantages

> General and documented solutions

A template solution (no code) that can be applied to different situations
Not specific to a language or a particular problem

> Faster Development Process

The solution has been extensively “tested” by others
Design reuse – instead of creating an original design look for existing ones

> Knowledge Transfer

Experience is conveyed through code from other developers
It defines a shared vocabulary for discussing design and architecture

> Code readability

Common solutions are easier to identify and to understand

Design Patterns

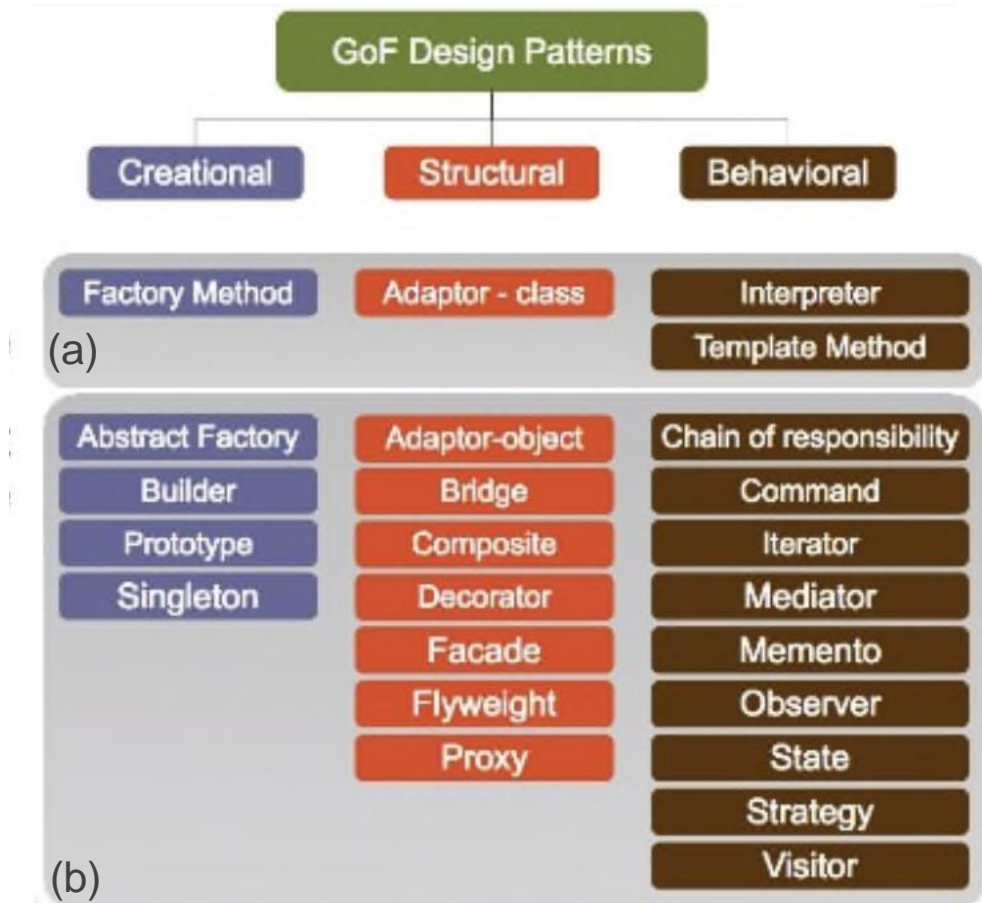
are often classified by purpose and scope

> Purposes

- I. Creational
- II. Structural
- III. Behavioural

> Scope

- a. Class
- b. Object



There are hundreds of design patterns! This is just a list from GoF.

Creational Patterns

- > The instantiation process is abstracted
- > They help to make a system independent of how its objects are created, composed and represented
- >> Creational patterns for classes use inheritance to vary the class that is instantiated
- >> Creational patterns for objects delegate instantiation to another object
- >>> Factory, Singleton

Design Patterns: Factory Method [GoF]

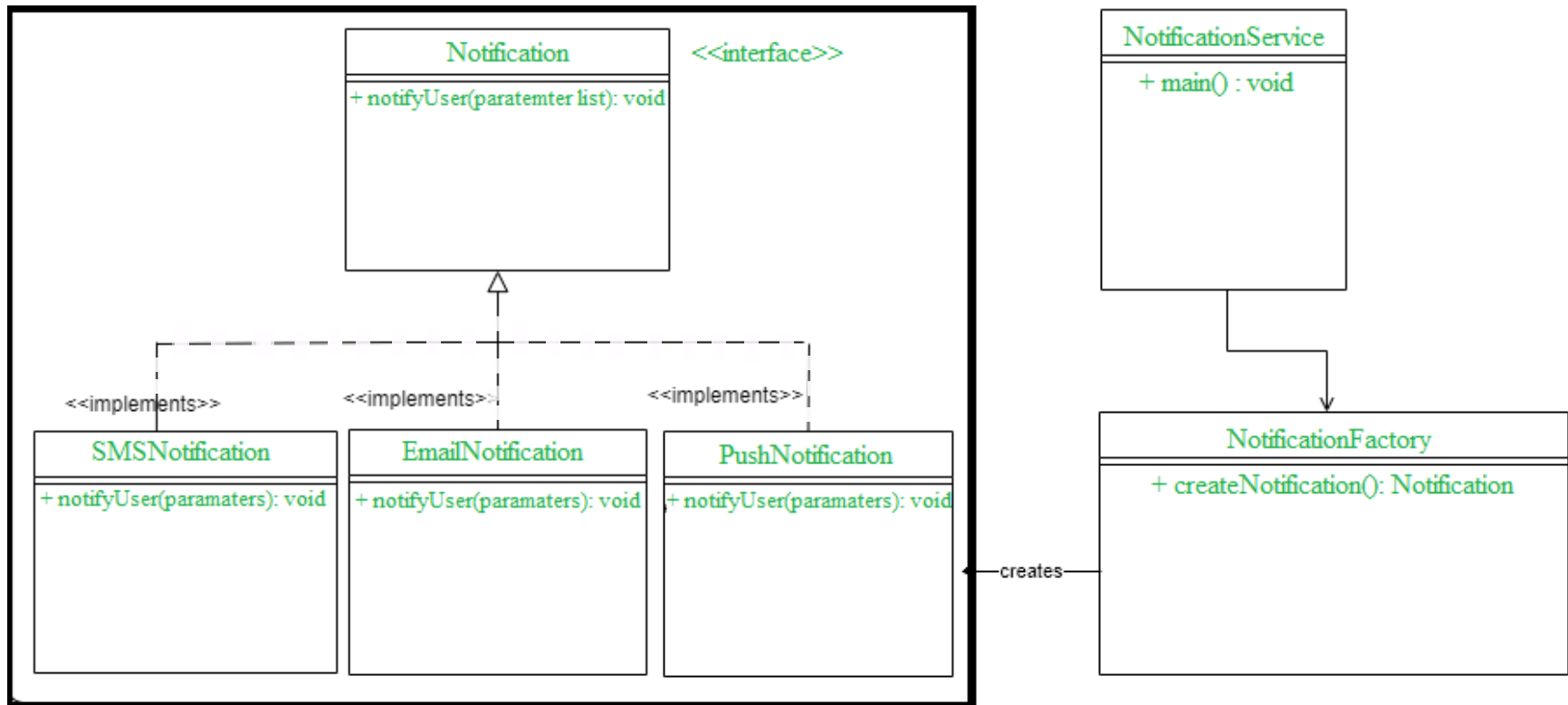
Intent

- > Define an interface for creating an object, but let subclasses decide which class to instantiate
- > Factory Method lets a class defer instantiation to subclasses

Use the Factory Method pattern when:

- > a class cannot anticipate the class of objects it must create
- > a class wants its subclasses to specify the objects it creates
- > classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Design Patterns: Factory Method



Design Patterns: Factory Method

```
public interface Notification {  
    void notifyUser();  
}  
  
public class SMSNotification implements Notification{  
    @Override  
    public void notifyUser() {  
        System.out.println("Sending an SMS Notification!");  
    }  
}  
  
public class PushNotification implements Notification{  
    @Override  
    public void notifyUser() {  
        System.out.println("Sending a Push Notification!");  
    }  
}  
  
public class EmailNotification implements Notification{  
    @Override  
    public void notifyUser() {  
        System.out.println("Sending an E-Mail Notification!");  
    }  
}
```


Design Patterns: Factory Method

```
public class NotificationFactory {  
    public Notification createNotification(String msg, String c)  
    {  
        String channel = c;  
        //if notification type is not defined, randomly choose one  
        //Another Example: if it was a game, we could balance the number of enemies  
        //or depending on the difficulty level, create different enemies, etc.  
        if (channel == null || channel.isEmpty()) {  
            List<String> l = Arrays.asList("SMS", "EMAIL", "PUSH");  
            Random rand = new Random();  
            channel = l.get(rand.nextInt(l.size()));  
        }  
  
        if ("SMS".equalsIgnoreCase(channel)) {  
            return new SMSNotification();  
        }  
        else if ("EMAIL".equalsIgnoreCase(channel)) {  
            return new EmailNotification();  
        }  
        else if ("PUSH".equalsIgnoreCase(channel)) {  
            return new PushNotification();  
        }  
  
        return null;  
    }  
}
```

Design Patterns: Factory Method

```
public class NotificationService {  
  
    public static void main(String[] args)  
    {  
        NotificationFactory notificationFactory = new NotificationFactory();  
        Notification notification = notificationFactory.createNotification("Hi, this is my notification!", "");  
  
        notification.notifyUser();  
    }  
}
```

Design Patterns: Singleton [GoF]

Intent

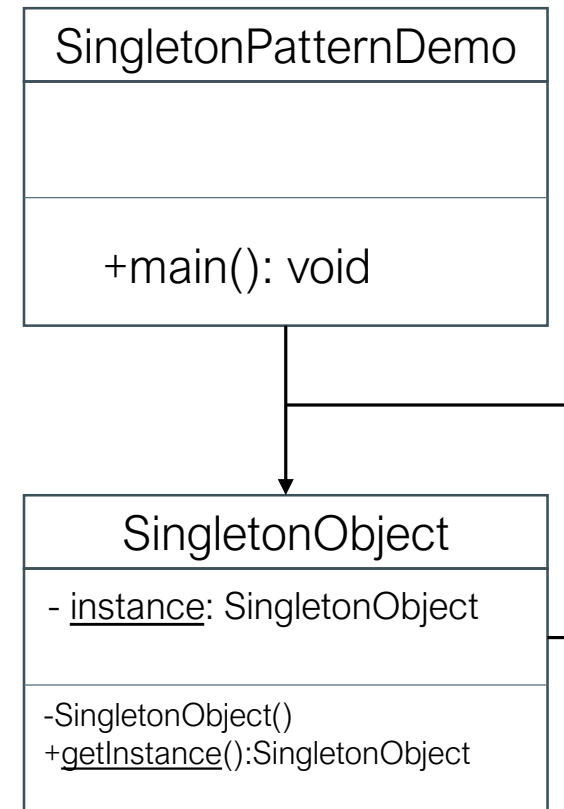
> Ensure a class only has one instance, and provide a global point of access to it.

> Use the Singleton pattern when:

>> there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

>> when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

>> Examples: File System, print queue, database connections, configuration settings...

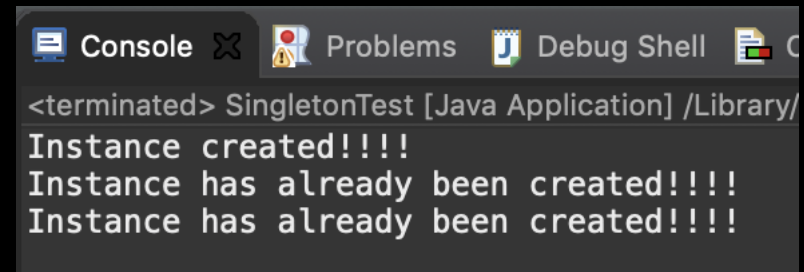


Design Patterns: Singleton

```
public class SingletonConnection {  
  
    //note that the constructor and instance variable are private  
    private static SingletonConnection instance = null;  
    private SingletonConnection(){};  
  
    //Note that this is the only method that can be accessed  
    public static SingletonConnection getInstance(){  
        if(instance == null) {  
            System.out.println("Instance created!!!!");  
            instance = new SingletonConnection();  
        }  
        else  
            System.out.println("Instance has already been created!!!!");  
  
        return instance;  
    }  
}
```

Design Patterns: Singleton

```
public class SingletonTest {  
  
    public static void main(String args[]) {  
        //We cannot instantiate it!  
        //SingletonConnection db = new SingletonConnection();  
        SingletonConnection.getInstance();  
        SingletonConnection.getInstance();  
        SingletonConnection.getInstance();  
    }  
}
```



```
<terminated> SingletonTest [Java Application] /Library/  
Instance created!!!!  
Instance has already been created!!!!  
Instance has already been created!!!!
```

Behavioural Patterns

- > Behavioural patterns are concerned with algorithms and the assignment of responsibilities between objects
- >> Behavioural class patterns use inheritance to distribute behaviour between classes
- >> Behavioural object patterns use composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself
- >>> Observer, Iterator, Template Method, State

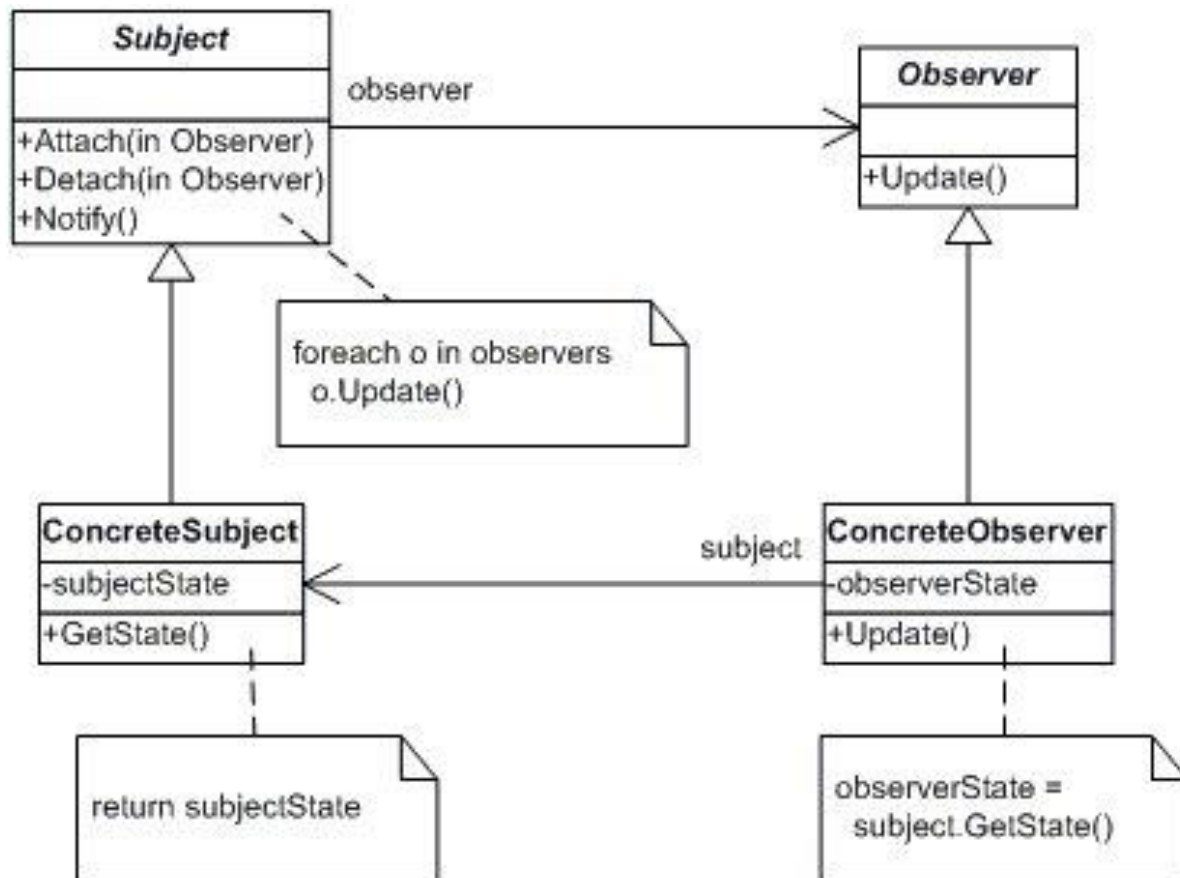
Design Patterns: Observer [GoF]

Intent

- > Define a **one-to-many dependency between objects** so that when one object changes state, all its dependents are notified and updated automatically.

- > Use the Observer pattern when:
 - >> an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - >> a change to one object requires changing others, and you don't know how many objects need to be changed.
 - >> an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Design Patterns: Observer [GoF]




```
public interface Subject { //interface Subject
    public void attach(Observer observer);
    public void detach(Observer observer);
    public void notifyAllObservers();
}
import java.util.ArrayList;
public class Place implements Subject{//class Place
    private ArrayList<Observer> observers;
    private String name;

    public Place(String name) {
        this.name = name;
        observers = new ArrayList<Observer>();
    }
    public void setCorona() { //you can add some conditions here (boolean corona)
        notifyAllObservers();
    }
    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyAllObservers() {
        for(Observer obs : this.observers)
            obs.update(this.name + " has a confirmed case. ");
    }
}
```



```
public interface Observer {  
    public void update(String msg);  
}
```

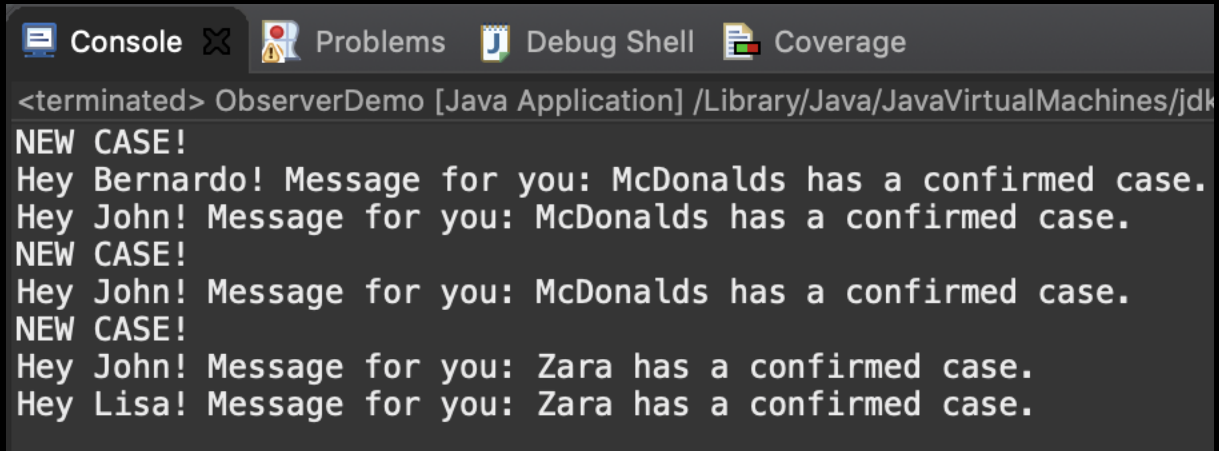
```
public class Customer implements Observer{  
    private String name;
```

```
    public Customer(String name) {  
        this.name = name;  
    }
```

```
    @Override
```

```
    public void update(String msg) {  
        System.out.println("Hey " + this.name + "! Message for you: " + msg );  
    }  
}
```

```
public class ObserverDemo {  
    public static void main(String[] args) {  
        Customer c1 = new Customer("Bernardo");  
        Customer c2 = new Customer("John");  
        Customer c3 = new Customer("Lisa");  
        Place p1 = new Place("McDonalds");  
        Place p2 = new Place("Zara");  
        p1.attach(c1);  
        p1.attach(c2);  
        p2.attach(c2);  
        p2.attach(c3);  
        System.out.println("NEW CASE!");  
        p1.setCorona();  
        p1.detach(c1);  
        System.out.println("NEW CASE!");  
        p1.setCorona();  
        System.out.println("NEW CASE!");  
        p2.setCorona();  
    }  
}
```



The image shows a screenshot of an IDE's console window. The window has a title bar with four tabs: 'Console', 'Problems', 'Debug Shell', and 'Coverage'. The 'Console' tab is active, displaying the output of a Java application named 'ObserverDemo'. The output text is as follows:

```
<terminated> ObserverDemo [Java Application] /Library/Java/JavaVirtualMachines/jdk  
NEW CASE!  
Hey Bernardo! Message for you: McDonalds has a confirmed case.  
Hey John! Message for you: McDonalds has a confirmed case.  
NEW CASE!  
Hey John! Message for you: McDonalds has a confirmed case.  
NEW CASE!  
Hey John! Message for you: Zara has a confirmed case.  
Hey Lisa! Message for you: Zara has a confirmed case.
```

Design Patterns: State

Intent

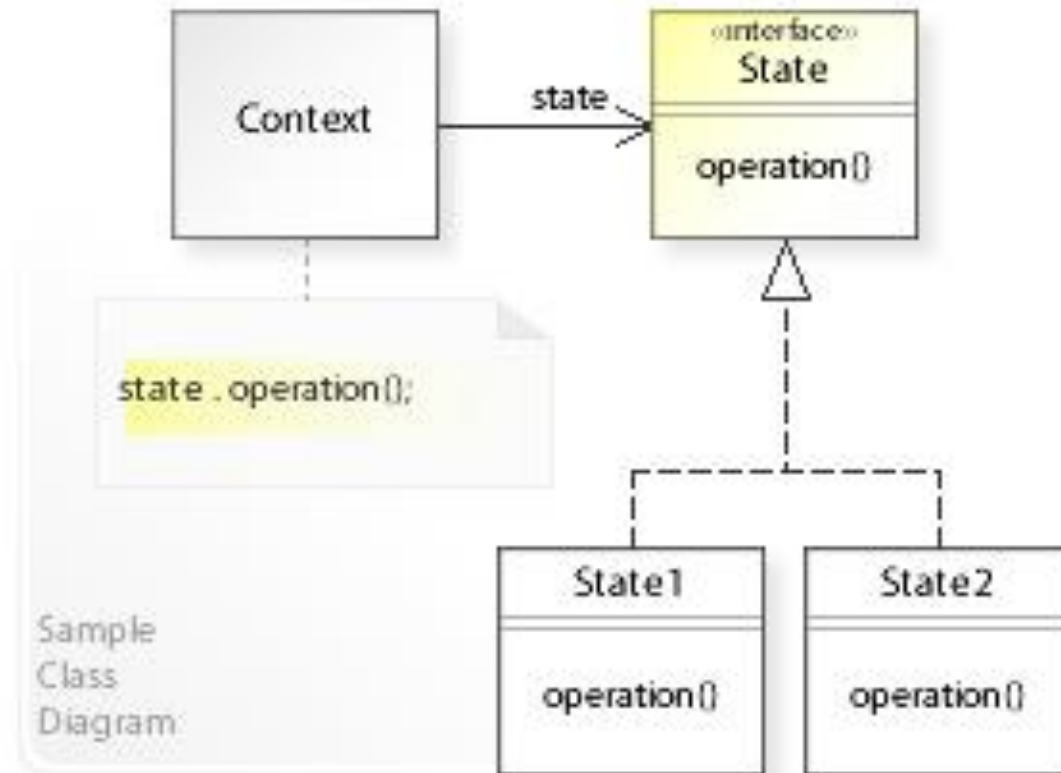
- > The state pattern allows an object to change its behaviour based on internal or external events.

- > Use the State pattern when:
 - >> If an object goes through clearly identifiable states, and the object's behaviour is especially dependent on its state, it is a good candidate for this pattern.

 - >> This pattern is related to the finite-state machine (a program can transition from one state to another. In each state the program will behave differently).

 - >>> Examples: User session (login, logout); Publishing (draft, moderation, published) [e.g., WordPress pages]; Games (play, pause, etc).

Design Patterns: State



```
public abstract class UserState { //it could be an interface instead (changes required)

    protected UserSession userSession;

    public UserState(UserSession userSession) {
        this.userSession = userSession;
    }

    public abstract boolean login(String username, String password);

    public abstract UserActivity createPost(String content);

    public abstract boolean likePost(Integer postId);

    public abstract boolean logout();

    public abstract List<ActionCount> generateActionCountReport();

    public abstract List<UserActivity> findAllPosts();

}
```

```
public class SessionState extends UserState {

    public SessionState(UserSession userSession) {
        super(userSession);
    }

    @Override
    public boolean login(String username, String password) {
        System.out.println("You cannot log in the session state");
        return false;
    }

    @Override
    public UserActivity createPost(String content) {
        UserActivityDao userActivityDao = UserActivityDao.getInstance();
        UserActivity userActivity = userActivityDao.createPost(userSession.getUsername(), content);
        System.out.println("Post created with id " + userActivity.getIdPost());
        return userActivity;
    }

    // ...
}
```

@Override

```
public boolean likePost(Integer postId) {  
    UserActivityDao userActivityDao = UserActivityDao.getInstance();  
    userActivityDao.likePost(userSession.getUsername(), postId);  
    System.out.println("You liked post " + postId);  
    return true;  
}
```

@Override

```
public boolean logout() {  
    System.out.println("Logout done");  
    userSession.changeState(new NoSessionState(userSession));  
    return true;  
}
```

@Override

```
public List<ActionCount> generateActionCountReport() {  
    CsvActionReport csvActionReport = new CsvActionReport();  
    UserActivityDao userActivityDao = UserActivityDao.getInstance();  
    List<ActionCount> actionCountList = csvActionReport.generateReport(userActivityDao.getFilePath());  
    return actionCountList;  
}
```

@Override

```
public List<UserActivity> findAllPosts() {  
    UserActivityDao userActivityDao = UserActivityDao.getInstance();  
    return userActivityDao.findAllPosts();  
}  
}
```



```
public class NoSessionState extends UserState {

    public NoSessionState(UserSession userSession) {
        super(userSession);
    }

    @Override //this is a dummy example (no auth method should be hard-coded)
    public boolean login(String username, String password) {
        if("admin".equals(username) && "123".equals(password)) {
            System.out.println("Login done");
            SessionState nextState = new SessionState(userSession);
            userSession.changeState(nextState);
            return true;
        }
        System.out.println("Login failed");
        return false;
    }

    @Override
    public UserActivity createPost(String content) {
        System.out.println("You cannot create post in no session state");
        return null;
    }
    //...
```

```
@Override
public boolean likePost(Integer postId) {
    System.out.println("You cannot like post in no session state");
    return false;
}

@Override
public boolean logout() {
    System.out.println("You cannot logout in no session state");
    return false;
}

@Override
public List<ActionCount> generateActionCountReport() {
    System.out.println("You cannot generate report in no session state");
    return null;
}

@Override
public List<UserActivity> findAllPosts() {
    System.out.println("You cannot get all posts in no session state");
    return null;
}
}
```

```
public class UserSession {

    // current state of the session
    UserState userState;
    // user on session
    String username;

    public UserSession() {
        UserState defaultState = new NoSessionState(this);
        changeState(defaultState);
    }

    public void changeState(UserState state) {
        this.userState = state;
    }

    public boolean login(String username, String password) {
        boolean loginOk = userState.login(username, password);
        if(loginOk) {
            this.username = username;
        }
        return loginOk;
    }

    public UserActivity createPost(String content) {
        return userState.createPost(content);
    }

    //...
```



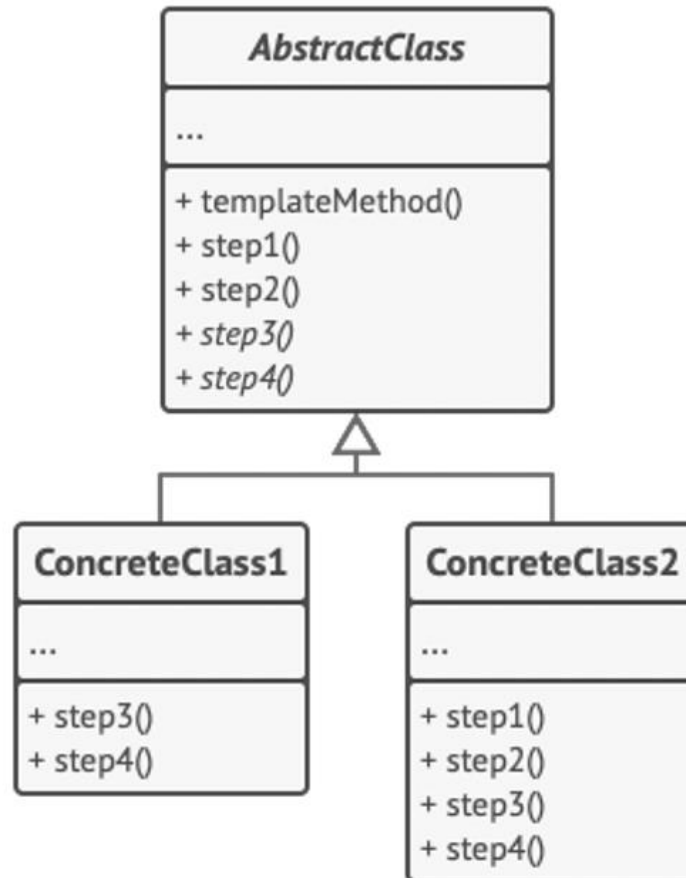
```
public boolean likePost(Integer postId) {  
    return userState.likePost(postId);  
}  
  
public List<ActionCount> generateActionCountReport() {  
    return userState.generateActionCountReport();  
}  
  
public boolean logout() {  
    boolean logoutOk = userState.logout();  
    if(logoutOk) {  
        this.username = null;  
    }  
    return logoutOk;  
}  
  
public List<UserActivity> findAllPosts() {  
    return userState.findAllPosts();  
}  
  
public String getUsername() {  
    return username;  
}  
}
```

Design Patterns: Template Method

Intent

- > “Define the **skeleton of an algorithm** in an operation, **deferring some steps to subclasses**. Template Method lets **subclasses redefine certain steps** of an algorithm without changing the algorithm's structure.” [GoF]
- > Use the Template Method pattern when:
 - >> you want to let clients extend only particular steps of an algorithm/procedure, but not the whole algorithm or its structure.
 - >> It is extremely important for code reuse (and reduce redundancy) && the template method leads to an *inversion of control* paradigm -> subclasses no longer control how the behaviour of a parent class is redefined.
 - >>> Examples: reports

Design Patterns: Template Method



```
public abstract class ActionReport {  
    /**  
     *  
     * Read data from file and return it as String.  
     */  
    public String readData(String filePath) throws IOException {  
        // get the path as an object  
        Path path = Paths.get(filePath);  
        // read file as byte array  
        byte[] bytes = Files.readAllBytes(path);  
        // bytes to String  
        String fileContent = new String(bytes);  
        return fileContent;  
    }  
  
    /**  
     *  
     * Gets the file content and structure it as UserActivity instances  
     */  
    public abstract List<UserActivity> parseFileContent(String rawData);  
  
    /**  
     * Handle exception if needed  
     */  
    public void handleException(Exception e) {  
        System.out.println("Ops. Something went wrong");  
    }  
}
```

```
/**
 * Count all actions
 */
public List<ActionCount> countActions(List<UserActivity> userActivityList) {
    if(userActivityList != null && !userActivityList.isEmpty()) {
        Map<String, ActionCount> actionCountMap = new HashMap<>();
        for(UserActivity userActivity : userActivityList) {
            if(userActivity != null && userActivity.getAction() != null) {
                String action = userActivity.getAction();
                if(!actionCountMap.containsKey(action)) {
                    ActionCount actionCount = new ActionCount();
                    actionCount.setAction(action);
                    actionCount.setCount(0);
                    actionCountMap.put(action, actionCount);
                }
                ActionCount actionCount = actionCountMap.get(action);
                actionCount.incrementCount();
            }
        }
        return new ArrayList<>(actionCountMap.values());
    }
    return null;
}
//...
```




```
/**
 *
 * This is the template method with all steps to generate the report
 */
public List<ActionCount> generateReport(String filePath) {
    try {
        String data = readData(filePath); //step 1
        List<UserActivity> activityList = parseFileContent(data); //step 2
        List<ActionCount> actionCountList = countActions(activityList); //step 3
        return actionCountList;
    } catch (Exception e) {
        handleException(e);
    }
    return null;
}
```

```
public class CsvActionReport extends ActionReport {

    @Override
    public List<UserActivity> parseFileContent(String rawData) {
        List<UserActivity> userActivityList = new ArrayList<>();
        if (rawData != null) {
            String[] lines = rawData.split("\n");
            for(String line : lines) {
                String[] strings = line.split(";");
                if(strings != null && strings.length == 4){
                    String username = strings[0];
                    String action = strings[1];
                    String content = strings[2];
                    Integer id = Integer.parseInt(strings[3]);
                    UserActivity userActivity = new UserActivity(username, action, content, id);
                    userActivityList.add(userActivity);
                }
            }
        }
        return userActivityList;
    }
}
```

```
public class XmlActionReport extends ActionReport {
    @Override
    public List<UserActivity> parseFileContent(String rawData) {
        List<UserActivity> userActivityList = new ArrayList<>();
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            InputSource is = new InputSource(new StringReader(rawData));
            Document doc = db.parse(is);
            Element root = doc.getDocumentElement();
            NodeList nodeList = root.getElementsByTagName("action");
            for (int i = 0; i < nodeList.getLength(); i++) {
                org.w3c.dom.Node n = nodeList.item(i);
                if (n.getNodeType() == org.w3c.dom.Node.ELEMENT_NODE) {
                    Element elem = (Element) n;
                    String username = elem.getAttribute("username");
                    String actionName = elem.getAttribute("actionName");
                    String content = elem.getAttribute("content");
                    Integer id = Integer.parseInt(elem.getAttribute("id"));
                    UserActivity userActivity = new UserActivity(username, actionName, content, id);
                    userActivityList.add(userActivity);
                }
            }
        } catch (ParserConfigurationException e) { e.printStackTrace(); }
        } catch (IOException e) { e.printStackTrace(); }
        } catch (SAXException e) { e.printStackTrace(); }
        }
        return userActivityList;
    }
}
```



```
public class ActionCount { //only a support class
```

```
    private String action;  
    private Integer count;
```

```
    public String getAction() {  
        return action;  
    }
```

```
    public void setAction(String action) {  
        this.action = action;  
    }
```

```
    public Integer getCount() {  
        return count;  
    }
```

```
    public void setCount(Integer count) {  
        this.count = count;  
    }
```

```
    public void incrementCount() {  
        count++;  
    }  
}
```

Design Patterns: Iterator

Intent

> The iterator design pattern solves the problem of how to traverse the elements of a collection object in a way that keeps client code doing the traversing loosely coupled to the collection object as well as the traversal algorithm.

>> In other words, access and traverse a collection without exposing its underlying representation (list, trees, stacks, graphs, ...)

> Use the Iterator pattern when:

>> Access elements in a specific order (trees: depth or breadth search; front to back, back to front, skip specific objects, ...)

>> Examples: traverse a collection

Bad Example

```
public class ProductPortfolio {  
  
    private Product products[];  
    //Not recommended. Promotes high coupling  
  
    public Product[] getProducts() {  
        return products;  
    }  
}
```

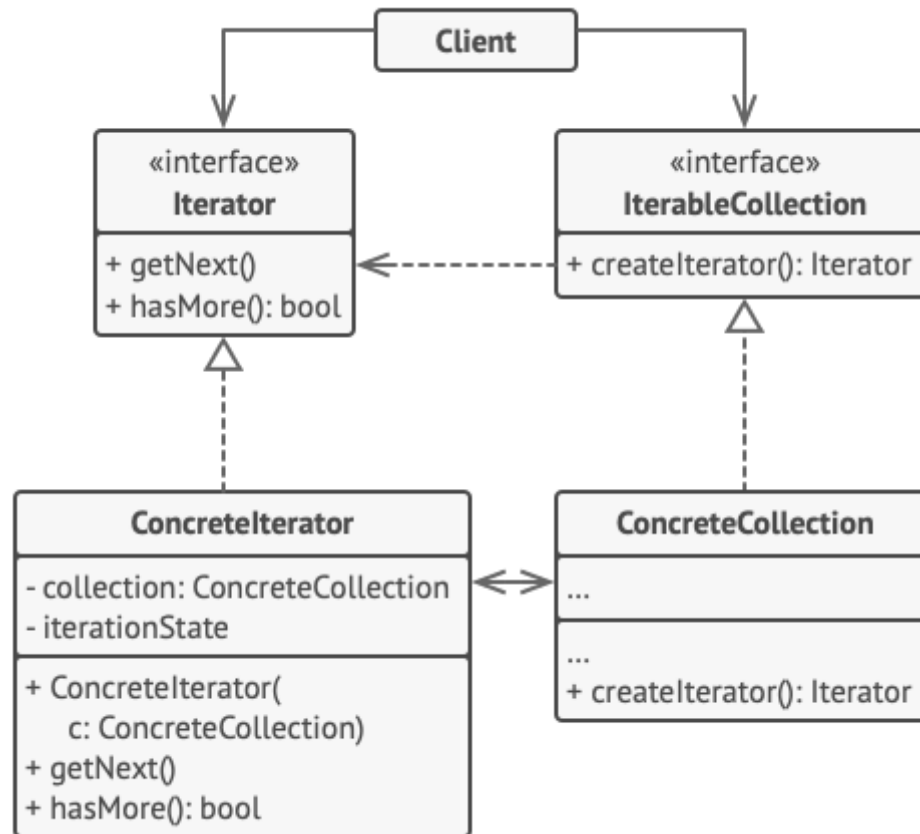
Client Code – Example

```
void static clientCode(ProductPortfolio portfolio) {  
    Product products[] = portfolio.getProducts();  
    for (int i=0; i<products.size; i++)  
        process(products[i]);  
}
```

The main problem with this solution is **it violates the principle of information hiding**. Giving **clients access to the internal data structure** used to store product data increases coupling between clients and ProductPortfolio

- > There is **no language mechanism preventing clients from directly adding or deleting** products from the internal data structure
- > The **cost of making changes to the internal data structure** of ProductPortfolio goes up with each new client access

Design Patterns: Iterator





```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
}
```

```
public interface IterableCollection {  
    public Iterator createIterator();  
}
```




```
public class FriendsConcreteCollection implements IterableCollection{

    //dummy example
    private String names[] = {"Friend_1", "Friend_2", "Friend_3"};

    public void addItem() {
        //...
    }

    @Override
    public Iterator createIterator() {
        return new FriendsConcreteIterator();
    }

    //...
```

```
//inner class
private class FriendsConcreteliterator implements Iterator {

    int index;

    @Override
    public boolean hasNext() {
        if(names != null && index < names.length){
            return true;
        }
        return false;
    }

    @Override
    public Object next() {
        if(this.hasNext()){
            return names[index++];
        }
        return null;
    }
}
```



```
public class IteratorDemo {  
  
    public static void main(String[] args) {  
        FriendsConcreteCollection f = new FriendsConcreteCollection();  
        //f.addItem();  
  
        for(Iterator iter = f.createIterator(); iter.hasNext();){  
            String name = (String) iter.next();  
            System.out.println("Name : " + name);  
        }  
    }  
}
```

A screenshot of an IDE's console window. The window has tabs for 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Coverage'. The 'Console' tab is active, showing the output of the Java application. The output text is: '<terminated> IteratorDemo [Java Application] /Library/Java/JavaVirtualMachines/jdk... Name : Friend_1 Name : Friend_2 Name : Friend_3'.

```
<terminated> IteratorDemo [Java Application] /Library/Java/JavaVirtualMachines/jdk...  
Name : Friend_1  
Name : Friend_2  
Name : Friend_3
```

Structural Patterns

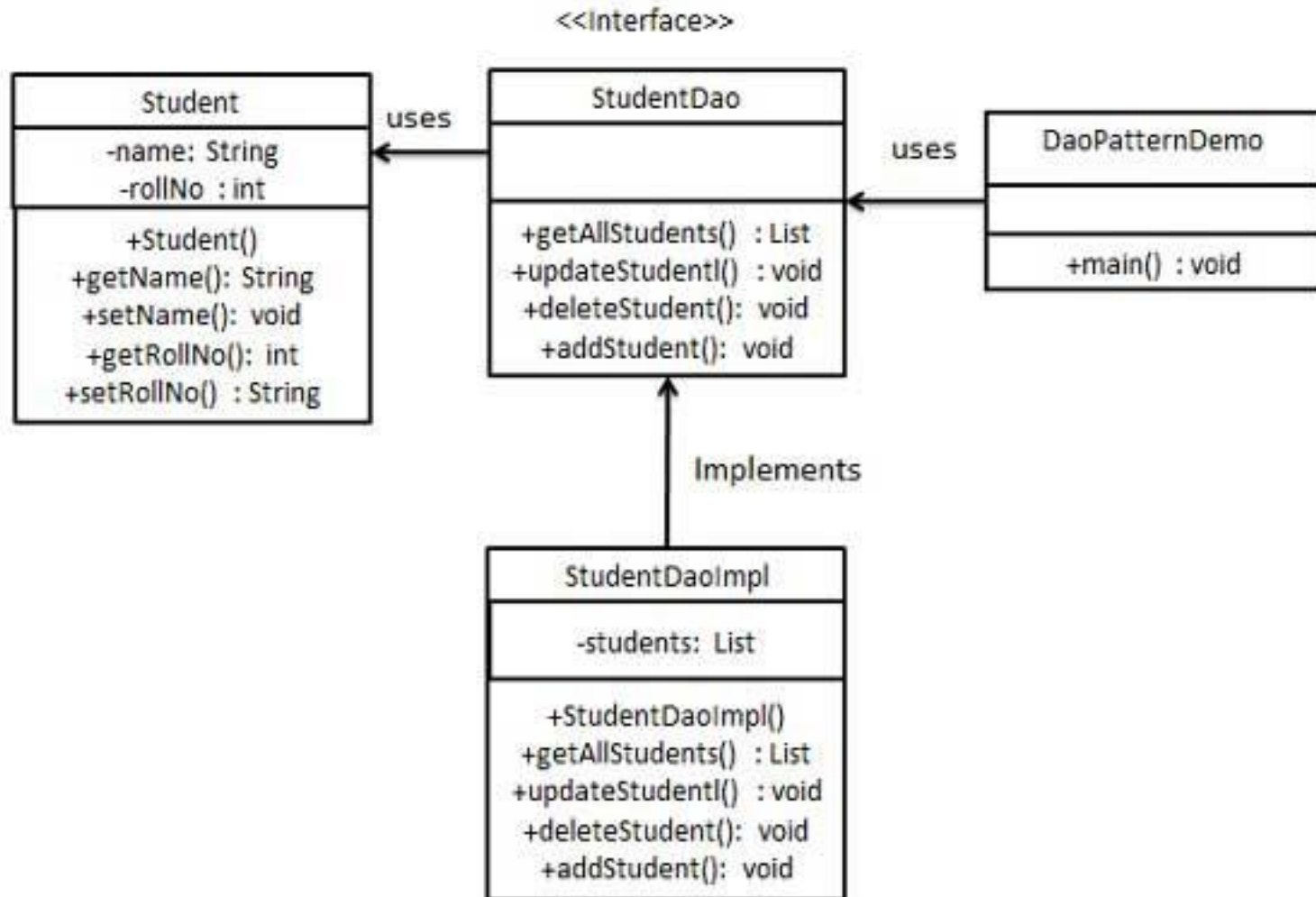
- > Structural Patterns are concerned with how classes and objects are composed to form larger structures
- >> Structural class patterns use inheritance to compose interfaces or implementations
- >> Structural object patterns describe ways to compose objects to realise new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition
- >>> DAO, Façade

Design Patterns: DAO

Intent

- > Data Access Object Pattern (DAO) is used to decouple domain logic from persistence mechanisms. It is used to do not expose details of the data storage (database, flat files, ...).
- > Use the DAO pattern when:
 - >> Access to data varies depending on the source of the data
 - >> When business components need to access a data source, they can use the appropriate API to achieve connectivity and manipulate the data source

Design Patterns: DAO



```
public class UserActivity {  
  
    private String username;  
    private String action;  
    private String content;  
    private Integer idPost;  
  
    public UserActivity(String username, String action, String content, Integer idPost) {  
        this.username = username;  
        this.action = action;  
        this.idPost = idPost;  
        this.content = content;  
    }  
  
    public String getUsername() {  
        return username;  
    }  
  
    public String getAction() {  
        return action;  
    }  
  
    public String getContent() {  
        return content;  
    }  
  
    public Integer getIdPost() {  
        return idPost;  
    }  
}
```

```
import java.util.List;

public interface UserActivityDaoInterface {

    public UserActivity createPost(String username, String postContent);

    public UserActivity likePost(String username, Integer idPost);

    public List<UserActivity> findAllPosts();

    public String getFilePath();

    public void deleteAll();

}
```



```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
import java.util.ArrayList;
import java.util.List;

public class UserActivityDao implements UserActivityDaoInterface{
    // singleton
    private static UserActivityDao instance;

    private static File file;
    private static Integer idCount = 0;
    static {
        try {
            file = File.createTempFile("user-action", ".csv");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private UserActivityDao() {}
```

```
public static UserActivityDao getInstance() {
    if(instance == null) {
        instance = new UserActivityDao();
    }
    return instance;
}

public UserActivity createPost(String username, String postContent) {
    try {
        idCount++;
        String action = "create-post";
        String text = username + ";" + action + ";" + postContent + ";" + idCount + "\n";
        Files.write(file.toPath(), text.getBytes(), StandardOpenOption.APPEND);
        System.out.println("Post saved in " + file.getAbsolutePath());
        UserActivity userActivity = new UserActivity(username, action, postContent, idCount);
        return userActivity;
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```



```
public UserActivity likePost(String username, Integer idPost) {  
    try {  
        String action = "like-post";  
        String content = "+1";  
        String text = username + ";" + action + ";" + content + ";" + idPost + "\n";  
        Files.write(file.toPath(), text.getBytes(), StandardOpenOption.APPEND);  
        System.out.println("Like saved in " + file.getAbsolutePath());  
        UserActivity userActivity = new UserActivity(username, action, content, idPost);  
        return userActivity;  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return null;  
}
```

```
public List<UserActivity> findAllPosts() {  
    List<UserActivity> userActivityList = new ArrayList<>();  
    try {  
        byte[] bytes = Files.readAllBytes(file.toPath());  
        String fileContent = new String(bytes);  
        String[] lines = fileContent.split("\n");  
        if(lines != null) {  
            for(String line : lines) {  
                String[] strings = line.split(";");  
                if(strings != null && strings.length == 4) {  
                    String username = strings[0];  
                    String action = strings[1];  
                    String content = strings[2];  
                    Integer id = Integer.parseInt(strings[3]);  
                    if("create-post".equals(action)) {  
                        UserActivity userActivity = new UserActivity(username, action, content, id);  
                        userActivityList.add(userActivity);  
                    }  
                }  
            }  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return userActivityList;  
}
```

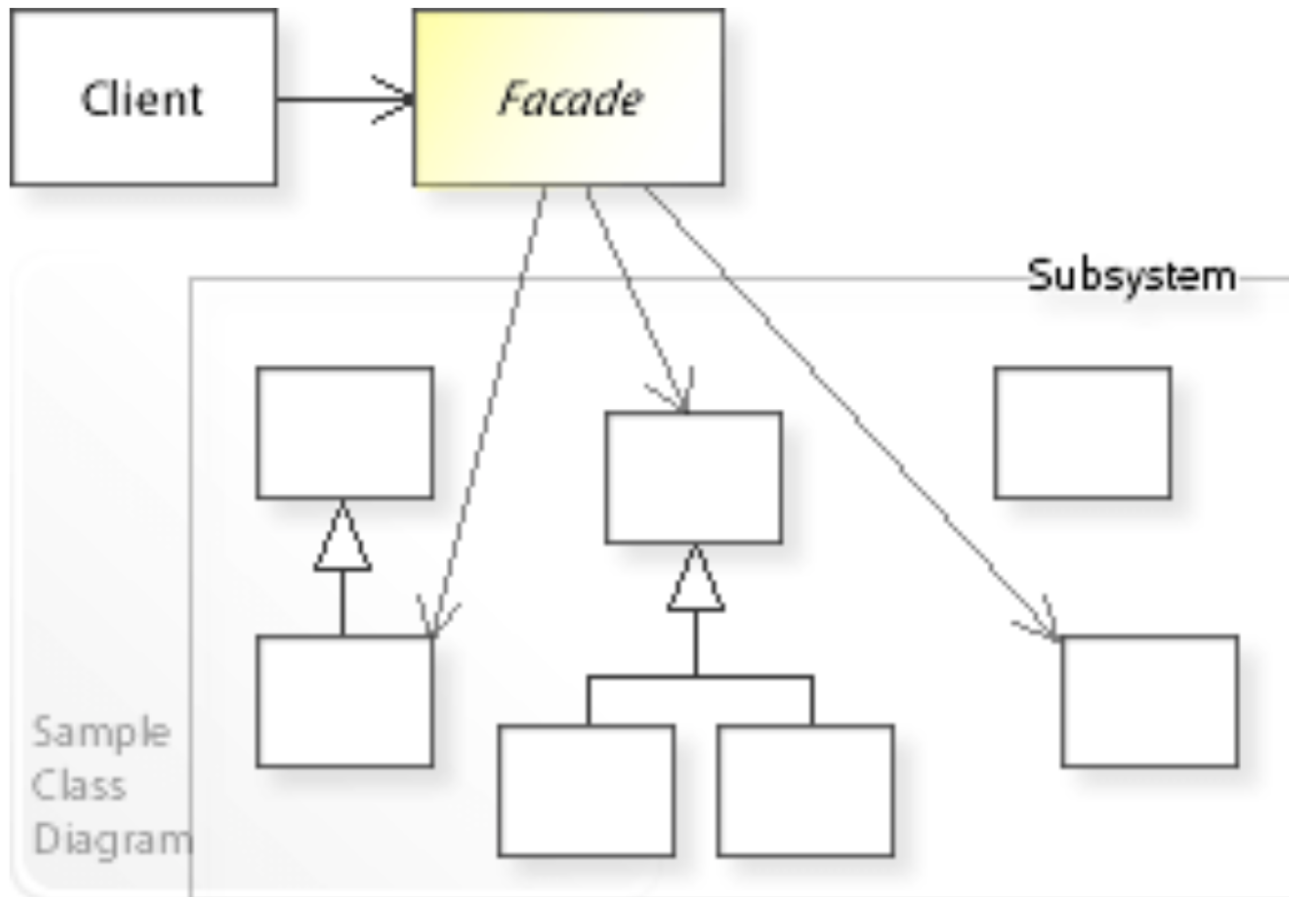
```
public String getFilePath() {  
    return file.getAbsolutePath();  
}  
  
public void deleteAll() {  
    try {  
        if(file.exists()) {  
            file.delete();  
        }  
        file = File.createTempFile("user-action", ".csv");  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
}
```

Design Patterns: Façade

Intent

- > "Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use." [GoF]
- > One of the easiest to understand and implement! It takes only one class to implement the Façade and does not use inheritance or polymorphism.
- > Use the Façade pattern when:
 - >> you want to define a single point of access for clients (unification). Clients deal with one class as opposed to a large interface or complex set of interfaces.
 - >> you want to offer a level of abstraction to the client classes (simplifying client code)Abstraction and Simplification.
- >> you want to decrease coupling – clients that access the services of a subsystem through a façade class are buffered from changes in the subsystem.

Design Patterns: Façade



```
//wraps complex operations
public class UserFacade {

    public UserDto createPost(String username, String password, String content) {
        UserSession userSession = new UserSession();
        userSession.login(username, password);
        UserActivity userActivity = userSession.createPost(content);
        List<ActionCount> actionCountList = userSession.generateActionCountReport();
        userSession.logout();
        UserDto userDto = new UserDto(userActivity, actionCountList);
        return userDto;
    }

    public UserDto likeAllPosts(String username, String password) {
        UserSession userSession = new UserSession();
        userSession.login(username, password);
        List<UserActivity> allPosts = userSession.findAllPosts();
        UserActivity lastUserActivity = null;
        if(allPosts != null && !allPosts.isEmpty()) {
            for(UserActivity userActivity : allPosts) {
                userSession.likePost(userActivity.getIdPost());
                lastUserActivity = userActivity;
            }
        }

        List<ActionCount> actionCountList = userSession.generateActionCountReport();
        userSession.logout();
        UserDto userDto = new UserDto(lastUserActivity, actionCountList);
        return userDto;
    }
}
//...
```




```
public static void main(String[] args) {  
    UserFacade userFacade = new UserFacade();  
    userFacade.createPost("admin", "123", "abc");  
    userFacade.createPost("admin", "123", "abcd");  
    UserDto userDto = userFacade.createPost("admin", "123", "abcf");  
    System.out.println(userDto);  
}  
  
}
```

```
public class UserDto {  
  
    private UserActivity lastPostCreated;  
    private List<ActionCount> communityReport;  
  
    public UserDto(UserActivity lastPostCreated, List<ActionCount> communityReport){  
        this.lastPostCreated = lastPostCreated;  
        this.communityReport = communityReport;  
    }  
  
    public UserActivity getLastPostCreated() {  
        return lastPostCreated;  
    }  
  
    public List<ActionCount> getCommunityReport() {  
        return communityReport;  
    }  
}
```

Exercise I

Considering the code below, which alternative is INCORRECT:

```
public class NotificationFactory {  
  
    public Notification createNotification(String msg, String c)  
    {  
        String channel = c;  
  
        if (channel == null || channel.isEmpty()) {  
            List<String> l = Arrays.asList("SMS", "EMAIL", "PUSH");  
            Random rand = new Random();  
            channel = l.get(rand.nextInt(l.size()));  
        }  
        if ("SMS".equalsIgnoreCase(channel)) {  
            return new SMSNotification();  
        } else if ("EMAIL".equalsIgnoreCase(channel)) {  
            return new EmailNotification();  
        } else if ("PUSH".equalsIgnoreCase(channel)) {  
            return new PushNotification();  
        }  
        return null;  
    }  
}
```

- (a) The method createNotification will return an instance of Notification or null.
- (b) The class NotificationFactory is a typical example of Factory Method pattern.
- (c) If the channel is not defined, the method randomly chooses one.
- (d) Notification can be an interface that SMSNotification, EmailNotification, and PushNotification implements. Notification cannot be a superclass of SMSNotification, EmailNotification, and PushNotification.

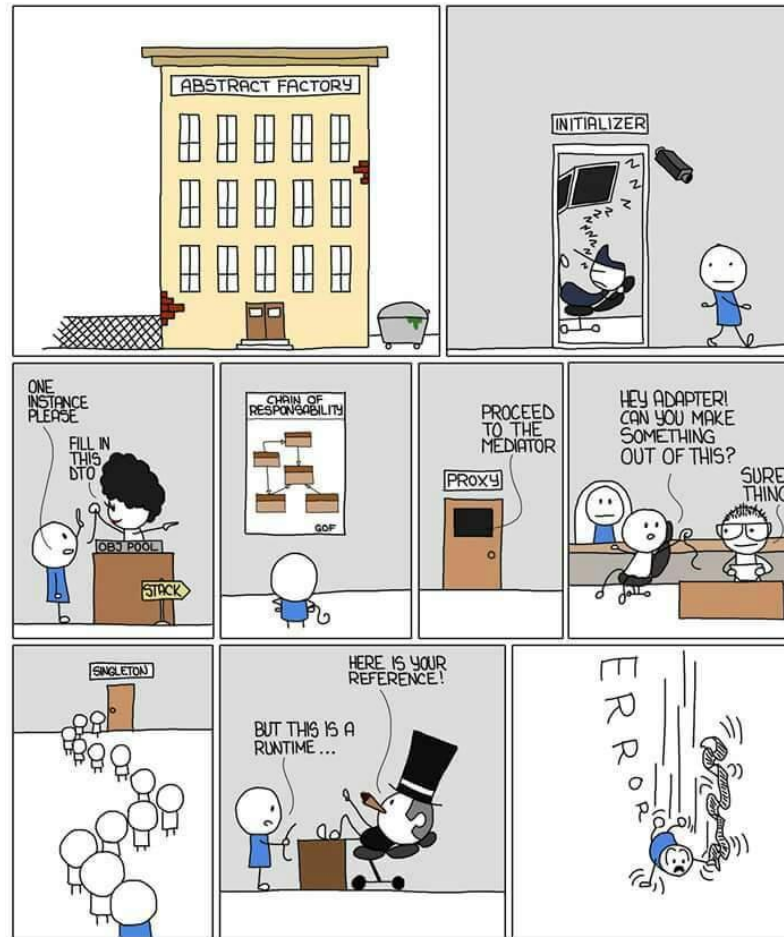
Exercise II

Which design patterns ensure that only one instance of a particular class gets created and promotes weak coupling between subsystem and its clients, respectively?

- (a) Factory and Façade
- (b) Singleton and Observer
- (c) Observer and Iterator
- (d) Façade and DAO
- (e) Singleton and Façade

Meme for today's lecture! Keep practicing!

DESIGN PATTERNS - BUREAUCRACY



MONKEYUSER.COM

Main References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., USA.
- [2] Eddie Burris, Programming in the Large with Design Patterns, Pretty Print Press, 2012. ISBN: 978-0615662145
- [3] Refactoring Guru. <https://refactoring.guru/design-patterns/>

Week 2 – Lab / Warm-up

- Attend Labs this week!
 - If you did not sign up for one of the labs, please, do it now using MyTimetable
 - GitLab link is available on Wattle