Design Patterns

Status	Completed

▼ Lecture 4: Creational & Structural Patterns

02/18/24

What is a design pattern?

- In software engineering, there are many common design problems that appear across different projects.
    - Over time, certain solutions are preferred over others because they are more flexible or reusable.
- Design patterns are practical, proven solutions to common design problems.
    - These design patterns are actual solutions used by experts in large-scale software.
- Design patterns assist software developers so that they have a guide to solve design patterns in the way experts do.
- Design patterns create a shared vocabulary among software developers, which makes communication easier.'

Creational, structural, and behavioral patterns

- One of the most famous books on design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software*, which identifies 23 different design patterns.
    - This book is written by four authors, commonly referred to as the gang of four.
    - These design patterns are split into three categories.
        1. Creational patterns handle creating new objects.
        2. Structural patterns describe how objects are connected to each other.
            - A structural pattern is defined by the types of relationships between the different objects in that pattern.
        3. Behavioral patterns focus on how objects distribute work to achieve a common goal.
- A collection of patterns related to a problem space is called a pattern language.

Singleton pattern

- Singleton is a creational pattern that refers to having one and only one object of a class, which is globally accessible.
    - For example, a `GameSettings` object in a game would follow the Singleton design pattern.
- To disallow many instances of a Singleton class from being created, the Singleton is given a private constructor.
    - The class stores a single instance of the Singleton as a class variable.
    - A public class method returns the instance of the Singleton.
        - This class method creates an instance of the Singleton if and only if an instance of the class does not already exist.
- One advantage of the Singleton class is that is not instantiated until it is used, a process called lazy creation.
    - This is similar to thunking, where an expression is not evaluated until it is used.

```
public class Singleton {
    // Store the instance of the Singleton as a class variable
    private static Singleton = null;

    private Singleton () {
        ...
    }

    // A class method to return the class variable
    // This class method uses lazy creation
    public static Singleton getInstance() {
        // Create an instance of the Singleton if and only if it does not already e
        if (instance == null){
            instance = new Singleton();
        }

        return instance;
    }
}


// Use the Singleton
Singleton.getInstance();
```

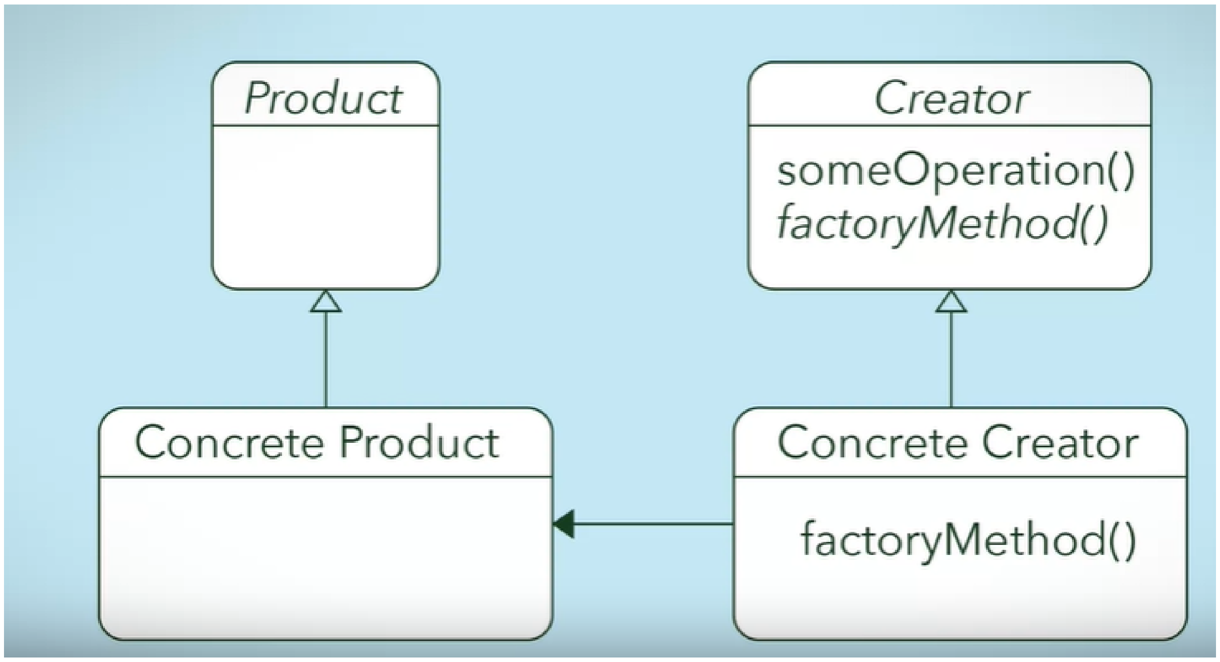
02/23/24

Factory method pattern

- Factory objects are a creational patterns that creates product objects of a set number of types.
    - The act of instantiating a class to create an object of a specific type is called concrete instantiation.
    - This makes software easier to maintain and change, as product object creation happens in Factory objects.
    - If there are multiple client objects that want to instantiate the same set of classes, a Factory will cut out redundant code and make software easier to modify.
        - As far as the client objects are concerned, they do not need to know how the Factory object creates its product objects.
- This design pattern concerns coding to an interface, not an implementation.

```
// A knife Factory which creates different types of knives based on an input string
public class KnifeFactory {
    public Knife createKnife (String knifeType) {
        Knife knife = null;

        // create a product object through concrete instantiation
        if (knifeType.equals("bread")) {
            knife = new BreadKnife();
        }
        else if (knifeType.equals("butter")) {
            knife = new ButterKnife();
        }
        else if (knifeType.equals("paring")) {
            knife = new ParingKnife();
        }
        ...

        return knife;
    }
}
```

- The Factory object is not actually one of the gang-of-four design patterns, but the Factory method pattern is.
- Factory methods are also a creation pattern which uses a separate method to create new objects, rather than an entire new class.
    - Like Factory objects, the Factory method's intent is to define an interface for creating objects.
    - The difference is that the Factory methods allows sub-classes to decide which classes to instantiate.
- The super-class that utilizes the Factory method is `abstract`, and therefore cannot be instantiated.
    - Because the Factory method is `abstract`, it must be implemented by the sub-classes.



```
// The super-class is abstract, and therefore cannot be instantiated
public abstract class KnifeStore {
    public Knife orderKnife (String knifeType) {
        // Creating a knife is now a method in the class
        knife = createKnife(knifeType);
        return knife
    }


    // Because the Factory method is abstract,
    // it must be implemented by its subclasses
    abstract Knife createKnife (String knifeType);
}


// KnifeStore sub-class
public BudgetKnifeStore extends KnifeStore {
    // Factory method for the sub-class
    Knife createKnife (String knifeType) {
        // Any budget knife can be created from this subtype,
        // again using concrete instantiation
        if (knifeType.equals("bread")) {
            knife = new BudgetBreadKnife();
        }
        else if (knifeType.equals("butter")) {
            knife = new BudgetButterKnife();
        }
        else if (knifeType.equals("paring")) {
            knife = new BudgetParingKnife();
        }
        else {
            return null;
        }
    }
}
```

- The UML class diagram for the general Factory methods patterns is shown in the figure.
    - In UML diagrams, `abstract` classes or methods are italicized.
    - The concrete creator is responsible for concrete instantiation.
    - The concrete creator instantiates concrete products, which inherit from `abstract` products.
- Factory methods allow a creator's sub-classes decide how objects are made, rather than a whole Factory object.

02/24/24

Facade pattern

- Facade is a structural pattern that provides a single, simplified interface for client classes to interact with a subsystem.
    - Facade classes abstract complex subsystems by acting as a wrapper class that hides a subsystem's complexity.
    - A Facade class can be used to wrap all the interfaces and classes in a subsystem if needed.

- Facades do not add more functionality to subsystems, but provide a point of entry into them.
    - Facades use encapsulation to hide information from client classes.
    - This removes the need for client classes to manage a subsystem on their own, resulting in less coupling between the subsystem and the client classes.

```
// 1. An interace is created
// The inteface interacts with a subsystem,
// and is not known to any client classes
public interface IAccount {
    public void deposit (...);
    public void withdraw (...);
    ...
}

// 2. One or more classes are implemented using the interface
public class Checking implements IAccount { ... }
public class Sawving implements IAccount { ... }

// 3. Create the Facade class and wrap
//    classes that implement the interface
public class BankService {
    public int createNewAccount (String type) {
        ...
    }
    public void transferMoney(...) {
        ...
    }
    ...
}

// 4. Use the Facade class (BankService) to access the subsystem
public class Customer {
    public static void main(String args[]) {
        // Access the subsystem
        BankService bankService = new BankService();
        // Use the facade to interact with the subsystem
        int savings = bankService.createNewAccount("saving");
        savings.transferMoney(...);
    }
}
```

Adapter pattern

- Adapter is a structural pattern facilitates communication between two systems by providing a compatible interface.
    - Adapter patterns implements a target interface that a client class can send information too.
    - The Adapter communicates this information to the adaptee class.

- The Adapter encapsulates the adaptee and presents a new interface that makes sense to the client class.
- The client class only needs to know about the target interface at the adapter.
    - Its important to realize that the adaptee may be a third-party library or separate system that the client may not have direct access too.

```
classDiagram
    direction TB
        TargetInterface <-- Client
        TargetInterface <|.. Adapter
    end
    direction TB
        Adapter --> Adaptee
    end
```

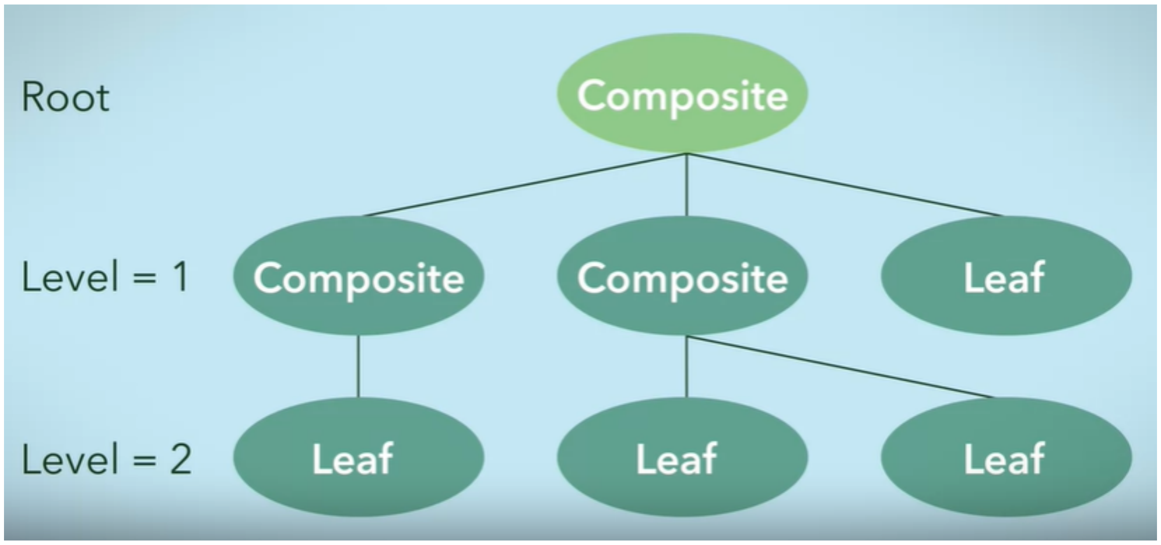
```
// 1. Design the target interface that the Adapter will implement
public interface WebRequester {
    public int request(Object);
}

// 2. Implement the target interface with the Adapter class
public class WebAdapter implements WebRequester {
    private WebService service;

    public void connect (WebService currentService) {
        this.service = currentService;
    }

    public int request (Object request) {
        ...
    }
}

// 3. Send the request from the client to the Adapter using the target interface
public class WebClient {
    private WebRequester webRequester;

    public WebClient (WebRequester webRequester) {
        ...
    }

    public void doWork () {
        Object object = makeObject();
        int status = webRequester.request(object);

        ...
    }
}

// 4. Utilize the Adapter class
public class Program {
    public static void main (String args[]) {
        WebService service = new Webservice("Host: https://...")
        WebAdapter adapter = new WebAdapter();
        adapter.connect(service);

        // Notice how the client does not need to directly interact with the adapte
        // Instead, the client communicates with the adaptee through the Adapter cl
        WebClient client = new WebClient(adapter);
        client.doWork();
    }
}
```

02/25/24

Composite pattern

- Composite is a structural pattern that has two purposes.
    1. It composes nested structures of objects.
    2. It deals with the classes from those objects uniformly.
- All classes inherit from an interface/abstract class, giving them a uniform behavior.
- Each inheriting class is one of two types.
    - Composite classes can contain other composite classes or leaf classes.
    - Leaf classes cannot contain other classes.

```
classDiagram
    IComponent <|.. Leaf
    IComponent <|.. Composite
    IComponent "0..*" ..o Composite

    class IComponent {
      <<interface>>
        +operation()
    }
    class Composite {
        +operation()
    }
    class Leaf {
        +operation()
    }
```



- Composite pattern allows classes to be organized into a recursive, tree-like structure.
  - This is called recursive composition, where objects can be composed of other objects that are of a common type.
  - This is enforced through polymorphism across each class, and allows Composite objects to be built quickly and easily without type checking.

```
// 1. Design the interface that defines the abstract class
public interface IStructure {
    public void enter();
    public void exit();
    public void location();
}

// 2. Implement the composite class(es)
// The composite class implements the interface
public class Housing implements IStructure {
    // The composite class can contain other strucutres
    public Housing (String address) {
        this.strucutres = new ArrayList<IStructure>();
    }

    public void enter () {
        ...
    }

    public void exit () {
        ...
    }

    public void location () {
        ...
    }
}

// 3. Implement the leaf class(es)
// The lead class also implements the interface
public class Room implements IStrucutre {
    public void enter () {
        ...
    }

    public void exit () {
        ...
    }

    public void location () {
        ...
    }
}
```

Proxy pattern

- Proxy is a structural pattern that is a “proxy” class in the place of a real, subject class.
  - The Proxy class wraps the subject class and delegates calls to it.
- There are three common reasons why a Proxy may be necessary.
  1. To act as a virtual proxy where the subject class may be resource intensive to instantiate.
  2. To act as a protection proxy in order to control access to the real subject class that may contain sensitive information.
     - In other words, it protects the real subject class by checking a client class’s request and controlling access to the real subject.
     - However this control is not achieved through an interface, as the Proxy and subject implement the same classes from the same interface.
  3. To act as a remote proxy where the Proxy is local, and the real subject class is remote.
- As the Proxy must implement the same methods as the subject class, they must both inherit from the same abstract superclass or implement the same interface.
  - Using this structure, a client class can interact with the abstract subject class as a proxy to the real subject class.

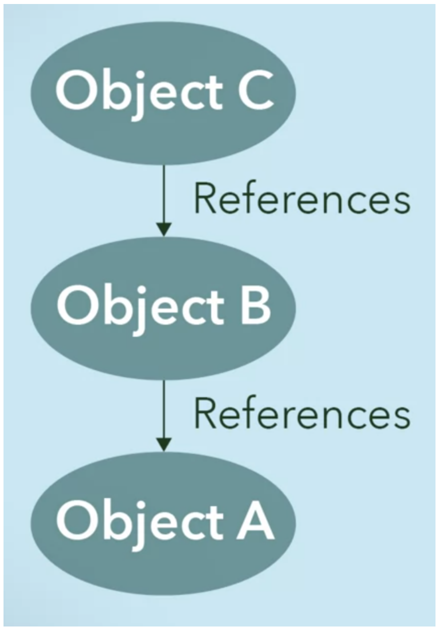
```
classDiagram
    direction TB
    Subject <|.. Proxy
    Subject <|.. RealSubject
    end
    direction LR
    Proxy --> RealSubject
    Client ..> Subject
    end

    class Subject {
      <<interface>>
    }
```

```
// 1. Design the subject interface
public interface IOrder {
    public void fulfillOrder(Order order);
}

// 2. Implement the real subject class
public class Warehouse implements IOrder {
    public void fulfillOrder (Order order) {
        ...
    }
    ...
}

// 3. Implement the proxy class
public class OrderFulfillment implements IOrder {
    private List<Warehouse> warehouses;

    public void fulfillOrder(Order order) {
        for (Warehouse warehouse: warehouse) {
            // check if any of the subject warehouses can fulfill the order
        }
    }
}
```

Decorator pattern

- Decorator is a structural pattern that uses aggregation to combine class behaviors at runtime.
    - As a reminder, aggregation refers to a weak "has a" behavior for multiple objects.
    - These multiple *part* objects can exist separately without the *whole* object.
- Classes inherit their behaviors at compile time, and therefore making classes behave differently would require making many new classes.
    - Decorator patterns allow one to attach new behaviors to classes dynamically, without having to create additional single-purpose classes.
- This aggregation behavior is used to build an aggregation stack of part objects.
    - Stacks are lists of elements that have a "first-in-last-out" behavior, where the elements are pushed (added to the end) and popped (removed from the end) from the stack.
    - Each level of the stack contains an object with its own behavior, which augments the behavior of the object beneath it in the stack.

- Object C augments the behaviors of object B, which augments the behaviors of object C.
- The top element (object C) is called, which would reference objects lower in the stack for behaviors that are not contained within the top element.



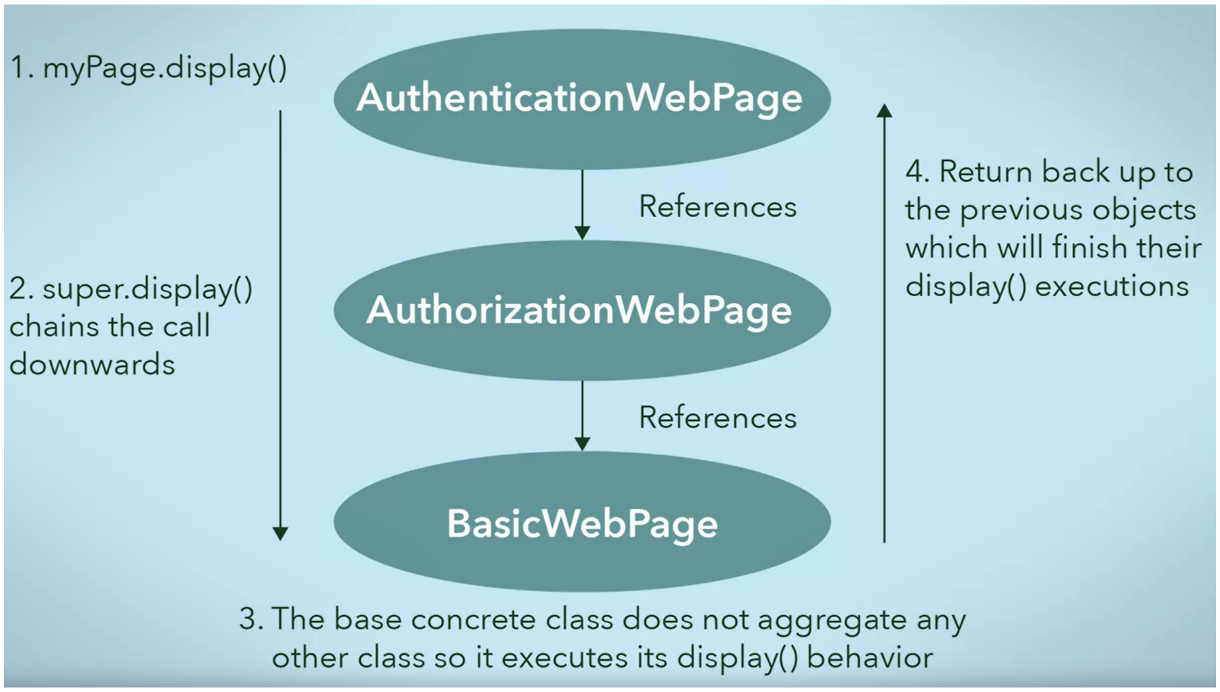
- A component interface is used to define a common type for all the components in the aggregation stack.
- Concrete components implement the component interface.
- Decorators are *abstract* classes which also implement the component interface.
    - This serves as the superclass of the concrete decorator "whole" class which aggregates other components and stores them in an aggregation stack.

```
classDiagram
    direction TB
        Component <|.. ConcreteComponent
        Component <|.. Decorator
        Component "1" --* Decorator
    end
    direction TB
        Decorator <|-- ConcreteDecorator1
        Decorator <|-- ConcreteDecorator2
        Decorator <|-- ConcreteDecoratorN
    end

    class Component {
        <<interface>>
        +operation()
    }
    class ConcreteComponent {
        +operation()
    }
    class Decorator {
        # component
        +operation()
    }
    class ConcreteDecorator1 {
        +operation()
    }
    class ConcreteDecorator2 {
        +operation()
    }
    class ConcreteDecoratorN {
        +operation()
    }
```

- `#` in a class diagram indicates that an attribute is `protected`.
- The decorator allows a component to be decomposed into a concrete component and additional behaviors, those additional behaviors being held by the Decorator.
- In order to recursively call the shared behavior, the concrete decorators invoke the super-class's (Decorator's) behavior using `super`.

```
// 1. Design the component interface
public interface WebPage {
    public void display();
}

// 2. Implement the interface with the concrete component class
public class BasicWebPage implements WebPage {
    public void display () {
        ...
    }
}

// 3. Implement the interface with the abstract Decorator class
public abstract class WebPageDecorator implements WebPage {
    protected WebPage webPage;

    public WebPageDecorator (WebPage webPage) {
        this.webPage = webPage;
    }

    public void display () {
        ...
    }
}

// 4. Inherit from the abstract Decorator and implement the component
//    interface with concrete decorator classes
```

```
public class AuthorizedWebPage extends WebPageDecorator {
    public AuthorizedWebPage (Webpage decoratedPage) {
        super(decoratedPage);
    }

    public void authorizeUser () {
        // Authorize user
    }

    public display () {
        super.display();
        this.authorizeUser();
    }
}
public class Authenticated extends WebPageDecorator {
    public AuthenticatedWebPage (Webpage decoratedPage) {
        super(decoratedPage);
    }

    public void authenticateUser () {
        // Authenticate user
    }

    public display () {
        super.display();
        this.authenticateUser();
    }
    ...
}
// More decorator classes as needed...

// 5. Link the concrete decorators together
public class Program {
    public static void main (String[] args) {
        // Create the concrete component
        WebPage webPage = new BasicWebPage();
        // Add first concrete decorator to the concrete component
        webPage = new AuthorizedWebPage(webPage);
        // Add the second concrete decorator to the concrete component
        webPage = new AuthenticatedWebPage(webPage);
        // Add more decorators as needed...

        // Now when display() is called, it will call all the methods
        // from all the concrete decorator classes in the aggregation stack
        webPage.display();
    }
}
```


1. myPage.display()
AuthenticationWebPage
References
AuthorizationWebPage
References
BasicWebPage
2. super.display() chains the call downwards
4. Return back up to the previous objects which will finish their display() executions
3. The base concrete class does not aggregate any other class so it executes its display() behavior

▼ Lecture 5: Behavioral Patterns

02/25/24

Template method pattern

- Behavioral design patterns focus on how individual objects collaborate to achieve a common goal.
- Template is a behavioral pattern that defines an algorithm of steps generally, and defers the implementation of some of those steps to sub-classes.
    - The methods that can be shared are implemented by an abstract super-class and inherited by sub-classes.
    - The methods that are sub-class specific but necessary are defined as abstract methods in the super-class.
        - It is then required for the sub-classes to implement these abstract methods in their own way.
- Template is a practical application of generalization and inheritance.

```
classDiagram
    SuperClass <|-- SubClass1
    SuperClass <|-- SubClass2
    SuperClass <|-- SubClassN

    class SuperClass {
        <<abstract>>
        +sharedMethod(s)() void
      # abstractMethod(s)()* void
    }
    class SubClass1 {
      # abstractMethod(s)() void
    }
    class SubClass2 {
      # abstractMethod(s)() void
    }
    class SubClassN {
      # abstractMethod(s)() void
    }
```

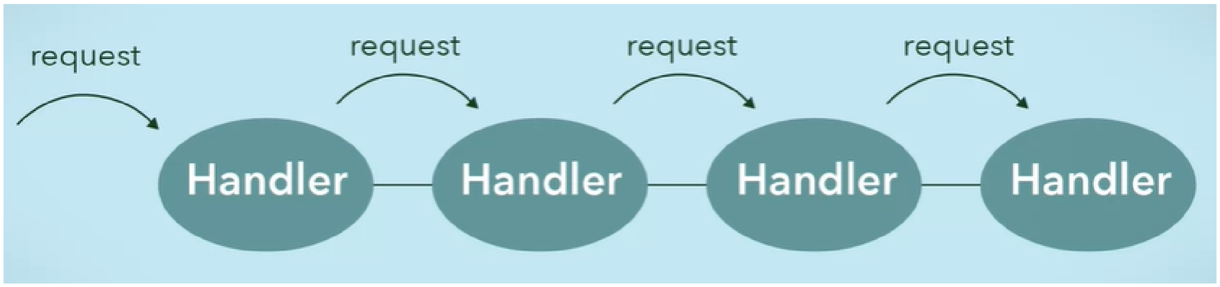
```
// 1. Create abstract super-class for sub-classes to inherit from
public abstract class PastaDish {
    // The "final" keyword makes this method immutable
    public final void makeRecipe () {
        boilWater();
        addPasta();
    }

    // Method that differs in functionality among
    // sub-classes is define but not implemented
    protected abstract void addPasta();

    // Method that has common functionality among
    // all sub-classes is implemented in the super-class
    private void boilWater() {
        // Boil water
    }
}

// 2. Create sub-classes that extend the super-class
public class Spaghetti extends PastaDish {
    protected void addPasta () {
        // Add spaghetti
    }
}
public class Penne extends PastaDish {
    protected void addPasta () {
        // Add penne
    }
}
```

Chain of responsibility pattern

- Chain of responsibility is a behavioral pattern that consists of a series of handler objects that are linked together.
    - These handlers have methods that are written to handle specific requests.
    - The intent is to avoid coupling each sender to their respective receivers.
- When a client sends a request to the handler chain, the following steps happen.

1. The first handler in the chain attempts to process the request.
2. If the current handler can process the request, then nothin happens after.
3. If not, the next handler in the chain tries to process the request.

    - It is not guaranteed that each handler will use this same algorithm to process a request.
        - This is a flaw of the design pattern that can be remedied by combining the Chain of responsibility with a Template pattern to designate a common handling algorithm to each handler.



- This pattern is very similar to exception handling, where a series of `try` and `catch` blocks are written.
    - This “try-catch” chain is iterated through, and at any point an exception can be raised by an exception handler.
- Each concrete handler inherits from an abstract handler super-class.
    - Each concrete handler is required to implement the abstract handler’s abstract handling method (what a mouthful).

```
classDiagram
    direction LR
        Client -- Handler
    end
    Handler " request" -- Handler
    Handler <|-- ConcreteHandler1
    Handler <|-- ConcreteHandler2
    Handler <|-- ConcreteHandlerN

    class Handler {
        <<abstract>>
        +handleRequest()*
    }
    class ConcreteHandler1 {
        +handleRequest()*
    }
    class ConcreteHandler2 {
        +handleRequest()
    }
    class ConcreteHandlerN {
        +handleRequest()
    }
```

State pattern

- State is a behavioral pattern that changes the behavior of an object based on the the state that it’s in at runtime.
    - A State interface is defined which contains methods that each state must respond too.
- A context class delegates to a state object to handle a request.
    - Each state object implements the same interface, and therefore the context class is able to handle all method calls from the state class based on the current state.
    - Because all states implement the same interface, the context class can delegate method calls to the state class without exceptions, even if the context class does not know what the current state is.

```
classDiagram
    Context o.. State
    direction TB
        State <-- ConcreteState1
        State <-- ConcreteState2
        State <-- ConcreteStateN
    end
    class Context {
        +request()
    }
    class State {
        +handle()
    }
    class ConcreteState1 {
        +handle()
    }
    class ConcreteState2 {
        +handle()
    }
    class ConcreteStateN {
        +handle()
    }
```

```
// 1. Create an interface that defines methods that states must implement
public interface State {
    public void insertDollar (VendingMachine vendingMachine);
    public void ejectMoney (VendingMachine vendingMachine);
    public void dispense (VendingMachine vendingMachine);
}

// 2. Define different state classes which implement the State interface
public class Stocked implements State {
    public void insertDollar (VendingMachine vendingMachine) {
        ...
    }
    public void ejectMoney (VendingMachine vendingMachine) {
        ...
    }
    public void dispense (VendingMachine vendingMachine) {
        ...
    }
}
public class OutOfStock implements State {
    public void insertDollar (VendingMachine vendingMachine) {
        ...
    }
    public void ejectMoney (VendingMachine vendingMachine) {
        ...
    }
    public void dispense (VendingMachine vendingMachine) {
        ...
    }
}

// 3. Create the context class that contains all the states
public class VendingMachine {
    // The different states are stored as private attributes
    private State stocked;
    private State outOfStock;

    private int count;

    public VendingMachine (int count) {
        // The different states are instantiated in the constructor
        stocked = new Stocked();
        outOfStock = new OutOfStock();

        if (count > 0) {
            currentState = stocked;
        }
        else {
            currentState = outOfStock;
        }
    }

    // The context class is now able to handle all of the methods based
    // on the current state, and does not need to know what specific
    // state it is calling on
    public void insertDollar () {
        currentState.insertDollar(this);
    }
    public void ejectMoney () {
        currentState.ejectMoney(this);
    }
    public void dispense () {
        currentState.dispense(this);
    }
}
```

02/26/24

Command pattern

- Command is a behavioral pattern which encapsulates a request as an object of its own.
  - A sender object does not send a message directly to a receiver object.
  - Instead, the request itself is an object that any available receiver objects can handle.
- An invoker object invokes command objects to handle the tasks they are asked to complete using the command manager.
  - A command manager is a Singleton that keeps track of commands, manipulates them, and invokes them.
- One purpose of the command pattern is to store and schedule different requests simultaneously.
  - This allows the commands to be stored in lists, manipulated, or put into a queue.
- The invoker creates a concrete command from the Command super-class.
  - These concrete commands can be received by the receiver.

```
classDiagram
direction LR
    Invoker o-- Command
end
direction TB
    Command <|-- ConcreteCommand
end
direction LR
    Receiver --o ConcreteCommand
end

class Command {
    +execute()
    +unexpected()
    +isReversible() boolean
}
class ConcreteCommand {
    +execute()
    +unexpected()
    +isReversible() boolean
}
class Receiver {
    +action()
}
```

- Command managers decouple the objects in a program, as any object making a request does not need to know about the other objects in the program.
- Command objects decouple the logic from an interface.
    - For example, the command pattern would make it easy to create new functionality when a new button is added to a user interface.
        - Rather than adding a new method to the user interface Singleton, a new command object can be created instead.

```
// 1. Create the Command super-class
public class Command {
    public abstract void execute ();
    public abstract void unexecute ();
    public abstract boolean isReversible ();
}

// 2. Implement a concrete command which extends Command
public class PasteCommand extends Command {
    private Document document; // The receiver
    private int position;
    private String text;

    public PasteCommand (Document document, int position, String text) {
        this.document = document;
        this.position = position;
        this.text = text;
    }

    public void execute () {
        document.insertText (position, text);
    }

    public void unexecute () {
        document deleteText (position, text.length());
    }

    public boolean isReversible () {
        return true;
    }
}

// 3. The invoker uses the command manager class to invoke a new command
CommandManager commandManager = CommandManager.getInstance()
Command command = new PasteCommand(document, position, text);
commandManager.invokeCommand(command);
```

Mediator pattern

- Mediator is a behavioral pattern that acts a “mediator” to handle communication between multiple objects.
    - The object informs the mediator when an event happens, and the mediator can perform logic in response to these events.
    - The mediator is able to request information or behavior from an object.
- Concrete mediators inherit from an abstract mediator class.
- The objects associated with a Mediator are called colleagues.
    - All concrete colleagues inherit from an abstract colleague class.
- Concrete mediators and colleagues are instantiated as necessary.
- The behavior of mediators allows for looser coupling between colleagues by centralizing the logic of interaction between them.
    - One disadvantage of this behavior is that the Mediator itself can become very large, meaning that the Mediator has low cohesion.

```
classDiagram
direction TB
Mediator <|-- ConcreteMediator
end
direction LR
Mediator <-- Colleague
end
direction TB
Colleague <|-- ConcreteColleague1
Colleague <|-- ConcreteColleague2
end
direction LR
ConcreteMediator --> ConcreteColleague1
ConcreteMediator --> ConcreteColleague2
end

class Mediator {
    <<abstract>>
}
class Colleague {
    <<abstract>>
}
```

Observer pattern

- Observer is a behavioral pattern that allows observers to subscribe to a subject class.
    - The subject notifies the observers to any changes in the state of the blog.
    - The subject may have zero or more observers subscribed to it at any given moment.
- The subject class must have three methods:
    1. They can allow a new observer to subscribe.

   2. They can allow a current observer to unsubscribe.
   3. They can notify all observers when it changes state.
- An an observer interface is implemented by a subscriber to be able to subscribe to the subject.
   - A subject must always have some way to update itself through implementing the observer interface.

```
classDiagram
direction TD
    Subject "observers" o-- "0..*" Observer
    Observer <.. Subscriber
end

class Subject {
    +registerObserver(Observer) void
    +unregisterObserver(Observer) void
    +notify() void
}
class Observer {
    <<interface>>
    +update() void
}
class Subscriber {
    +update() void
}
```

```
// 1. Create the subject class
public class Subject {
    // An array list is used to keep track of the current observers
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public void registerObserver (Observer observer) {
        observers.add(observer);
    }
    public void unregisterOberserver (Observer observer) {
        observers.remove(observer);
    }
    public void notify () {
        for (Oberver observer : observers) {
            observer.update();
        }
    }
}

// 2. Create the Observer interface
public interface Observer () {
    public void update ();
}
```

```
// 3. Implement the observer interface in a subscriber class
public class Subscriber implements Observer {
    public void update () {
        // receive the change from the Subject through the Observer interface
    }
}
```

```
sequenceDiagram
    activate Observer
    Observer ->> Subject: subscribe()
    activate Observer
    deactivate Observer
    activate Subject
    deactivate Subject
    activate Subject
    Subject->> Subject: notify()
    Subject ->> Observer: update()
    activate Observer
    Observer ->> Subject: getState()
    deactivate Subject
    Observer ->> Subject: unsubscribe()
    activate Subject
    deactivate Subject
    deactivate Observer
```

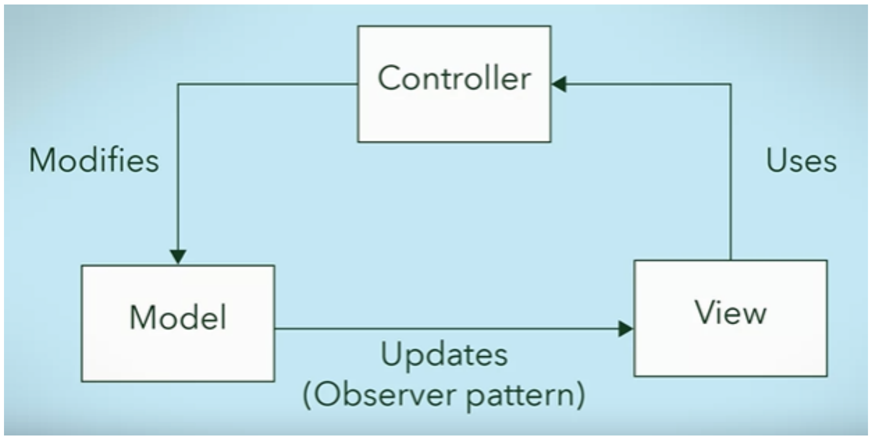
▼ Lecture 6: Working with Design Patterns & Anti-patterns

02/26/24

MVC Pattern

- The Model view controller (MVC) is a self-contained design pattern that is commonly used for user interfaces.
    - The MVC pattern separates concerns of a user interface into a model, a view, and a controller.
    - The model is an *entity object* which contains the underlying data and logic that users want to see and manipulate.
        - This is analogous to the back-end of a software system.
    - The view is a *boundary object* which presents the model information to the user and allows them to interact with the model.
        - This is analogous to the front-end of a software system.
        - It is possible to have multiple separate views that all update based on the model.
        - The view updates based on the current model state using the Observer design pattern.
            - `java.util` contains an `Observable` class that can be extended to create an Observer class.

- The controller is a *control object* which is responsible for interpreting user interaction with the view and modifying the model accordingly.



- The model should be able to exist on its own without any views or controllers.
- The defining feature of the MVC pattern is the separation of concerns between the back-end, front-end, and the coordination between the two.

03/05/24

Liskov substitution principle

- Subtyping allows subclasses to be used as substitutes for their base class.
- The substitution principle states that any subclass S can be used to replace a base class B, if and only if, S is a subtype of B.
    - In other words, a subclass should be able to stand in for its base class.
- The Liskov substitution principle states that if a subclass S is a subtype of a base class B, then S can be used to replace all instances of B without changing the behaviors of the program.
    - The behaviors of a base/super-class should be broad enough in its methods and attributes so that all subclasses of it can utilize them.
- If a subclass cannot utilize one or more attributes or methods of its superclass, then this is a violation of the Liskov substitution principle.
- The Liskov substitution principle places constraints on subclasses to enforce the proper use of inheritance:
    1. A subclass cannot add more conditions than a superclass to determine if an inherited method should be called.
    2. When a subclass calls a method, the state of a program should be in the same state as if the method were called on the superclass.
    3. All invariant conditions of the superclass should stay invariant in the subclass.
    4. All immutable characteristics of the superclass should stay immutable in the subclass.

Open/closed principle

- The open/closed principle states that classes should be open for extension, but closed to change.

- Once the development of a class in finalized, it should be “closed” so that it cannot be changed in the future.
  - This prevents undesirable behaviors in a program that may happen when a long-standing class is changed.
- Even though existing, functioning classes cannot be changed, there are two ways in which new functionality can be added to a system:
  1. New subclasses can be created which inherit from an existing superclass.
     - In Java, to prevent a class or method from being extendible, the keyword `final` is used.
  2. The class is abstract, and therefore enforces the open/closed principle through polymorphism.
- The open/closed principle is advantageous in the fact that it keeps the functional, stable parts of a system separate from the developing, varying parts.

Dependency inversion principle

- One should consider where a system’s dependencies are, or where reliances between different parts of the system are.
- Sometimes systems are reliant on a particular resource or class, that cannot be substituted for anything else.
- The dependency inversion principle remedies this by stating that high level modules should depend on high level generalizations, not on low level details.
- Interfaces and abstract classes are high level generalizations, where concrete resources are low level details.
  - Client classes should depend on interfaces or abstract classes, not concrete resources.
    - This means that a client class is dependent on expected behaviors, not a specific implementation of those behaviors.
  - Concrete resources should have their behaviors generalized into an interface or abstract class.
- In effect, the dependency inversion principle reduces coupling between modules.

Composing objects principle

- One goal of design patterns is to address the problem of tight coupling.
  - While inheritance is a great solution for code reuse, it comes with the problem of tight coupling between superclasses and subclasses.
- The composing objects principle achieves code reuse through aggregation *rather than inheritance*.
  - The system should be designed so that concrete classes can delegate tasks to other concrete classes.
  - Some design patterns such as Composite and Decorator use the composing objects principle, where their overall behavior results from the composed sum of individual objects.
- Composing objects removes the need to find commonalities between two classes so that one can inherit from the other.
- Composing objects allows the behaviors of objects to change during runtime.
  - A new combination of behavior can be created.
  - This varies with inheritance, which is fixed at compile-time.
- The largest disadvantage of composition is that there is less code reuse that with inheritance.
  - The behaviors for each concrete class must be implemented separately, meaning there may be a large amount of code reuse or similarity.
- Overall, the needs of a system should be examined to determine whether composition or inheritance is more appropriate.

    - Are the classes in a system related or unrelated in some way?
    - Are specialized classes necessary to handle specific cases, or is a different implementation of the same behavior needed?

Interface segregation principle

- Design patterns generalize concrete classes into interfaces or abstract classes so that client classes become less dependent on concrete classes.
- The interface segregation principle states that a class should not be forced to depend on methods it does not use.
    - Any classes that implement an interface should not have to create useless implementations of methods that are defined in the interface, but that the class does not use.
- In the example below, `SelfCheckout` and `Cashier` both inherit from an `ICashier` interface.
    - The `SelfCheckout` class should not be forced to implement the `takeBreak()` method from `ICashier` as self-checkout machines do not need to take breaks, and therefore it has no use for it.
    - This implementation would be considered a “dumb” or “dummy” implementation, as it would not do anything.

```
classDiagram
    direction TB
        ICashier <.. SelfCheckout
        ICashier <.. Cashier
    end

    class ICashier {
        <<interface>>
        +scanItem() void
        +takePayment() void
        +takeBreak() void
    }

    class SelfCheckout {
        +scanItem() void
        +takePayment() void
        +takeBreak() void
    }

    class Cashier {
        +scanItem() void
        +takePayment() void
        +takeBreak() void
    }
```

- To achieve this principle, large interfaces should be split into smaller generalizations, which in turn increases interface cohesion.
- Interfaces should be split up in such a way so that they can properly describe the separate functionalities of a system.

```
classDiagram
    direction TB
        ICashier <.. SelfCheckout
        ICashier <.. Cashier
        IHumanWorker <.. Cashier
    end

    class ICashier {
        <<interface>>
        +scanItem() void
        +takePayment() void
    }

    class IHumanWorker {
        <<interface>>
        +takeBreak() void
    }

    class SelfCheckout {
        +scanItem() void
        +takePayment() void
    }

    class Cashier {
        +scanItem() void
        +takePayment() void
        +takeBreak() void
    }
```

Principle of least knowledge

- Classes should be designed to that they do not need to depend upon almost every other class in a system.
- The principle of least knowledge states that classes should now about and interact with as few other classes as possible.
    - A method should not invoke methods of any object that is not local to that method, or the class that the method is contained in.
    - Essentially, classes and methods should only have access to their “neighboring” classes and methods.
- Classes and method should not have access to the behaviors of an entire system, as this would create a high degree of coupling.
- A method M should only call other methods if:
    1. They are encapsulated within the same object.
    2. They are encapsulated within an object that is in the parameters of the method M.
    3. They are encapsulated within an object that is instantiated inside the method M.
    4. They are encapsulated within an object that is referenced in an instance variable of the class of the method M.

Code smells

- It is extremely rare that the software system will be designed perfectly on the first attempt.
    - Refactoring is the process of making changes to code that that the external behaviors of the code are not changed, but the internal structure is improved.
    - It is important to run tests frequently to make sure that refactoring has not altered the behavior of the code.
- Similarly to design patterns that assist in creating reusable, flexible, and maintainable programs, anti-patterns or code smells are common patterns that come up which inhibit these traits.
    - Many of these code smells are identified in the book *Refactoring: Improving the Design of Existing Code*.
    - They are called code smells, as one can use these common anti-patterns to “sniff out” what is bad about a program.
- There are two types of code smells:
    - Static code smells are anti-patterns that appear in code.
    - Dynamic code smells are anti-patterns that arise when changing code.
- There are many different static code smells:
    - Too many comments or a lack of comments is a code smell.
        - Comments are helpful to explain what code does, but comments can fall out of sync with refactored code.
        - Too many comments may be compensating for that fact that code is structured in a way that is not self-explanatory.
            - Too many comments to explain a complex design can indicate that that is a bad design to begin with.
        - Comments *should* be used for documenting APIs, and design or algorithm decisions.
            - Comments should explain the “why” of a design or algorithm, not the “how”.
    - Too much duplicated code is a code smell.
        - Duplicated code refers to exact or *similar* code in multiple places in a program.
        - Having to update code in only one location makes is easier to modify, and reduces the chance for bugs to appear as a result.
        - In other words, *don’t repeat yourself*.
    - A long method is a code smell.
        - A “long” method can indicate that there is more occurring in that method than there should be.
        - It is debatable whether lines of code is a good measure of code complexity, as this can depend on the method’s purpose and the programming language’s syntax.
    - Similarly, large classes are a code smell.
        - These “large” classes are commonly referred to as “god”, “blob”, or “black hole” classes.
        - These classes often start out small, but enlarge over time due to that fact that they seem like the appropriate place to add more and more functionality to a software system.
        - To avoid large classes, the purpose of the class should be explicit, and the class should stay highly cohesive.
    - Conversely, small classes are also a code smell.

        - Small classes are often “data” classes, which contain only data (attributes) and have no real functionality (methods), except for perhaps getter and setter methods.
        - A related issue is data clumps, where groups of data appear together in the instance variables of a class, or as parameters to methods.
            - It would make sense to group these variables that often appear together into an object, with those variables as attributes.
    - Long parameter lists are a code smell.
        - Having a method with a long parameter list can be difficult to use.
        - Unfortunately, a common solution to this is to use global variables, which often come with problems of their own.
        - Therefore, extensive comments can be used to explain each parameter in detail.
            - However, too many comments to explain a bad design is *also* a code smell.
        - The best solution to long parameter lists is to use parameter objects, which hold all the information for a parameter list in a specific context/state object.
- There are many dynamic code smells as well:
    - Divergent changes are a code smell that occurs when a class is changed in many different ways for many different reasons.
        - This is an indicator that the responsibilities of the class should be broken up into separate classes.
    - Shotgun surgery refers to when a change in one place of a program results in one having to fix many other areas of the code.
        - This can usually be remedied by consolidating methods from many classes into a single, or a few, classes.
            - This should only be done if it makes sense to, as it could result in the large class code smell.
    - Feature envy occurs when a method is more interested in the details of a class, other than the one it is in.
        - Those two classes should be consolidated if it makes sense to do so.
    - Inappropriate intimacy refers to when two classes rely on each other too much through two-way communication.
        - This means that the two classes are highly coupled.
    - Method/message chains violate the principle of least knowledge by having the object in a method/message chain call a method from its “neighbor”, which in turn calls a method that is not the original object’s “neighbor”.
    - Primitive obsession is when a program relies on built-in types too much.
        - This specifically refers too when appropriate abstractions are not being identified, and suitable classes are not being created.
        - Not abstracting primitive information into more self-explanatory classes can result in magic values, which are harder to understand.
        - Classes, unlike primitive types, can include methods to validate or modify values stored in them.
    - Switch statements can be code smells if they are used to check what class an object is.
    - Speculative generality occurs when implementing a program feature that is not needed at the time, but “might” be useful in the future.

        - Speculative generality leads to over-engineered software systems.
        - This is opposite of what Agile development emphasizes as “just in time design”.
            - Just in time design refers to having just enough design to transform certain requirements into a working system.
        - One shouldn’t spend time writing code that may never be used.
    - Refused bequest refers to when a subclass inherits a behavior, but does not need it.
        - This is a violation of the Liskov substitution principle.
- It is important to review code frequently for code smells to make sure that it remains reusable, flexible, and maintainable.