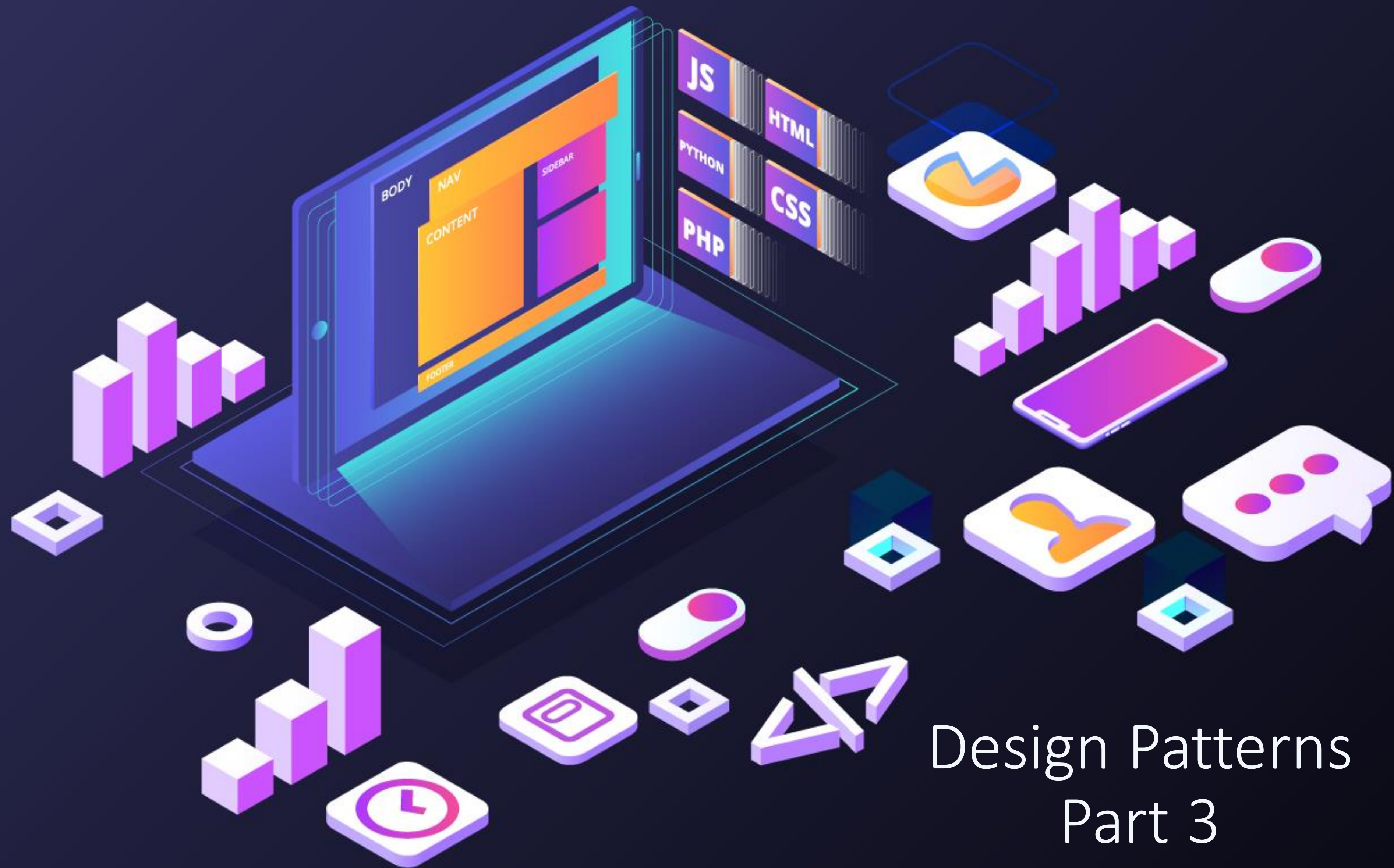


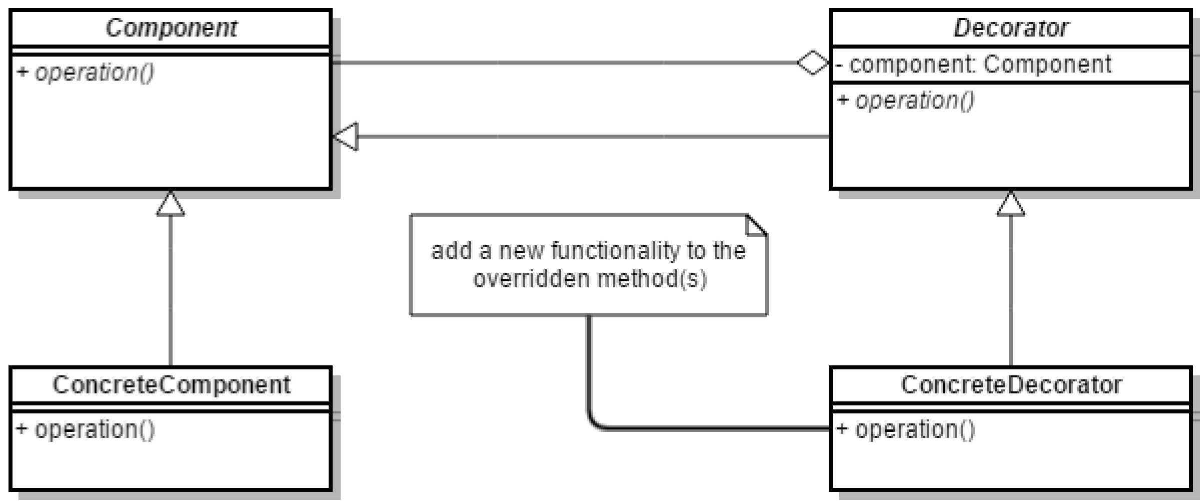
Design Patterns
Part 3

by Vlad Ungureanu

Agenda

Design Patterns

- Decorator
- Façade
- Flyweight
- Front Controller


Component
+ operation()
Decorator
- component: Component
+ operation()
add a new functionality to the overridden method(s)
ConcreteComponent
+ operation()
ConcreteDecorator
+ operation()

Decorator – Implementation

```
public abstract class Worker {
	public void work(){
	  System.out.println("work!");
	}
}


public class Decorator extends Worker {
	Worker worker;

	public Decorator(Worker worker){
	  this.worker = worker;
	}

	public void work(){
	  this.worker.work();
	}
}
```

Decorator – Implementation

```
public class ComplexWorker extends Worker{

    public void work(){
      System.out.print("Complex ");
      super.work();
    }
}

public class SimpleWorker extends Worker{

    public void work(){
      System.out.print("Simple ");
      super.work();
    }
}
```

LEARN stuff IO

Decorator – Implementation

```
public class InheritenceDecorator extends Original{

    public void doExtendedWork(){
      this.extendedWork();
      super.work();
    }

    public void extendedWork(){
      System.out.print("Extended: ");
    }
}
```

Decorator – Implementation

```
public class DecoratorMain {

    public static void main(String[] args){
      InheritenceDecorator decorator = new InheritenceDecorator();
      decorator.doExtendedWork();

      Decorator decoratorOne = new Decorator(new SimpleWorker());
      decoratorOne.work();
      Decorator decoratorTwo = new Decorator(new ComplexWorker());
      decoratorTwo.work();
    }
}
```

LEARN stuff IO

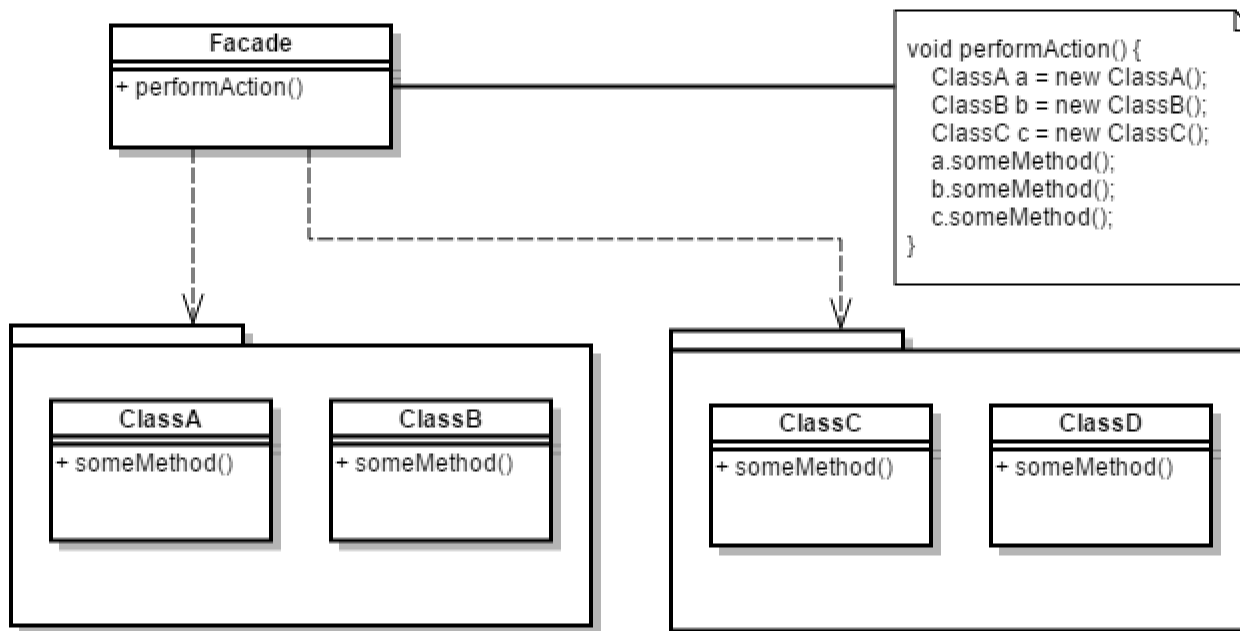
Decorator – Why ?

- For responsibilities that can be withdrawn
- When extension by sub-classing is impractical.
- Sometimes, a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
- Class definition may be hidden or otherwise unavailable for sub-classing.
- Decorator Pattern is flexible than inheritance because inheritance add responsibilities at compile time and it will add at run-time.
- Decorator pattern enhance or modify the object functionality

LEARN stuff IO

Decorator – Why not ?

- Extra layer of complexion
- If multiple decorators are use intent might be obscured
- Code maintenance might get difficult


Facade
+ performAction()
void performAction() {
ClassA a = new ClassA();
ClassB b = new ClassB();
ClassC c = new ClassC();
a.someMethod();
b.someMethod();
c.someMethod();
}
ClassA
+ someMethod()
ClassB
+ someMethod()
ClassC
+ someMethod()
ClassD
+ someMethod()

Façade – Implementation

```
• import ro.learnstuff.implementation.package1.OperationsOne;
• import ro.learnstuff.implementation.package2.OperationsTwo;
• import ro.learnstuff.implementation.package3.OperationsThree;

• public class Facade {

      public static void operate(){
      OperationsOne.operate(new OperationsTwo());
      OperationsThree.operate();
      }
• }

• public class FacadeMain {

      public static void main(String[] args){
        Facade.operate();
      }
• }
```

LEARN stuff IO

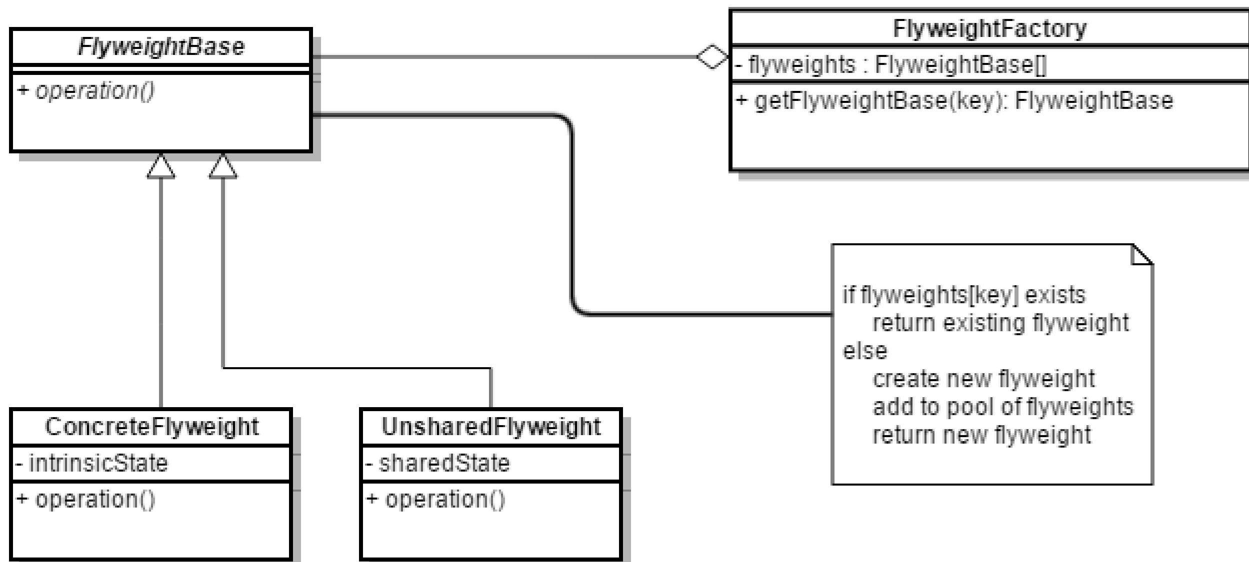
Façade – Why ?

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.
- Hide package dependencies and group them into a single point

LEARN stuff IO

Façade – Why not ?

- Extra layer of complexity
- Can hide application logic or business logic
- Changes in subsystems might be hidden through a façade, but behavior might change
- Might make system difficult to understand


FlyweightBase
+ operation()
FlyweightFactory
- flyweights : FlyweightBase[]
+ getFlyweightBase(key): FlyweightBase
ConcreteFlyweight
- intrinsicState
+ operation()
UnsharedFlyweight
- sharedState
+ operation()
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight

Flyweight– Implementation

```
public interface Service {
	void perform();
}

public class GreatService implements Service {
	@Override
	public void perform() {
	  System.out.println("Nice service performed!");
	}
}

public class NiceService implements Service {
	@Override
	public void perform() {
	  System.out.println("Nice service performed!");
	}
}
```

Flyweight– Implementation

```
public class Flyweight {

 private static final HashMap<String,Service> services = new HashMap<>();

        public static Service getService(String type) {
                eturn services.get(type);
        }

        public static void addService(String type, Service service){
                services.put(type, service);
        }
}
```

Flyweight– Implementation

```
public class FlyweightMain {

    public static void main(String[] args){
        Flyweight.addService("nice", new NiceService());
        Flyweight.addService("great", new GreatService());

        Service service = Flyweight.getService("nice");
        service.perform();
    }
}
```

LEARN stuff IO

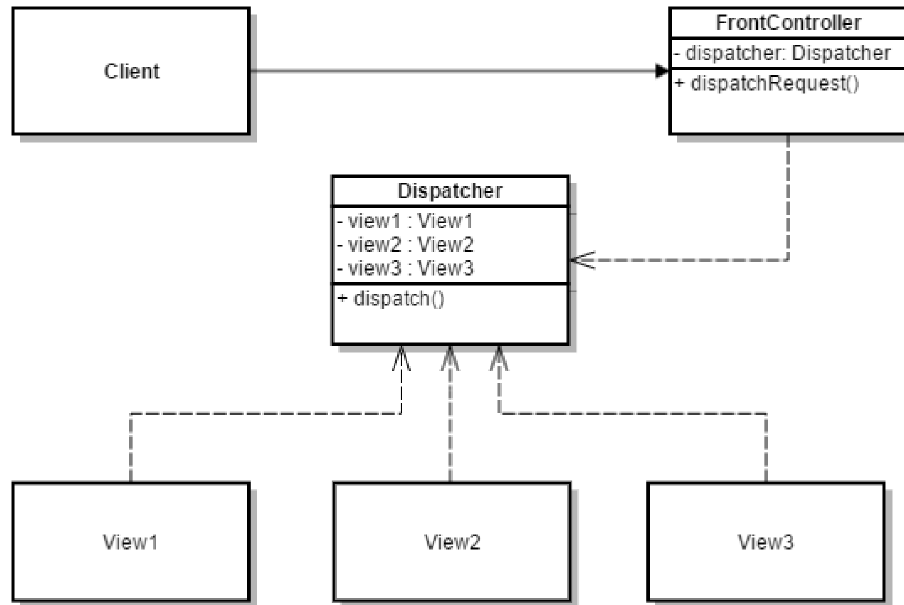
Flyweight – Why ?

- Many similar objects are used and the storage cost is high
- The majority of each object's state data can be made extrinsic
- A few shared objects would easily replace many unshared objects
- The identity of each object does not matter

LEARN stuff IO

Flyweight – Why not ?

- All instances of the class are related
- Extra layer of abstraction
- If logic changes for some implementation they require massive refactoring as they all depend on a interface


Client
FrontController
- dispatcher: Dispatcher
+ dispatchRequest()
Dispatcher
- view1 : View1
- view2 : View2
- view3 : View3
+ dispatch()
View1
View2
View3

Front Controller – Implementation

```
public interface Page {
	void show();
}

public class HomePage implements Page{

	@Override
	public void show(){
	System.out.println("Home page!");
	}
}
public class LoginPage implements Page{

	@Override
	public void show(){
	System.out.println("Login page!");
	}
}
```

Front Controller – Implementation

```
public class FrontController {

    static Dispatcher dispatcher = new Dispatcher();

    public static void processRequest(String request, String authenticationStatus){
            if ("authenticated".equals(authenticationStatus)){
                dispatcher.dispatch(request);
            } else {
                dispatcher.dispatch("login");
            }

    }
}
```

Front Controller – Implementation

```
public class Dispatcher {

public static Map<String, Page> pages = new HashMap<>();

    public Dispatcher() {
        pages.put("home", new HomePage());
        pages.put("login", new LoginPage());
    }

    public void dispatch(String request) {
     pages.get(request).show();
    }
}
```

Front Controller – Why ?

- Common system services processing completes per request. For example, the security service completes authentication and authorization checks.
- Logic that is best handled in one central location is instead replicated within numerous views.
- Decision points exist with respect to the retrieval and manipulation of data.
- Multiple views are used to respond to similar business requests.
- A centralized point of contact for handling a request may be useful, for example, to control and log a user's progress through the site.
- System services and view management logic are relatively sophisticated.

Front Controller – Why not ?

- Each HTTP request is unique and separate and should be treated as such.
- If you break a web application into small modules that are loosely coupled its easier to test the unit/module (your not testing the architecture as well as the controller for example) .
- Performance is better if you deal with a single request uniquely.

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –