Advanced Programming

Design Patterns

Design Patterns

- Someone has already solved your problem.
- Exploit the wisdom and lessons learned by other developers
- Load your brain with them and then recognize places in your designs and existing applications where you can apply them.
- Instead of code reuse, with patterns you get experience reuse.

Desirables

- Encapsulate what varies.
- Favor Composition over inheritance.
- Program to Interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.

Use Case

- Publish-Subscribe
- Job Portals
- News Feed
- Weather Services

Weather Service

```
public class WeatherData {

// instance variable declarations

public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
// other WeatherData methods here

}
```

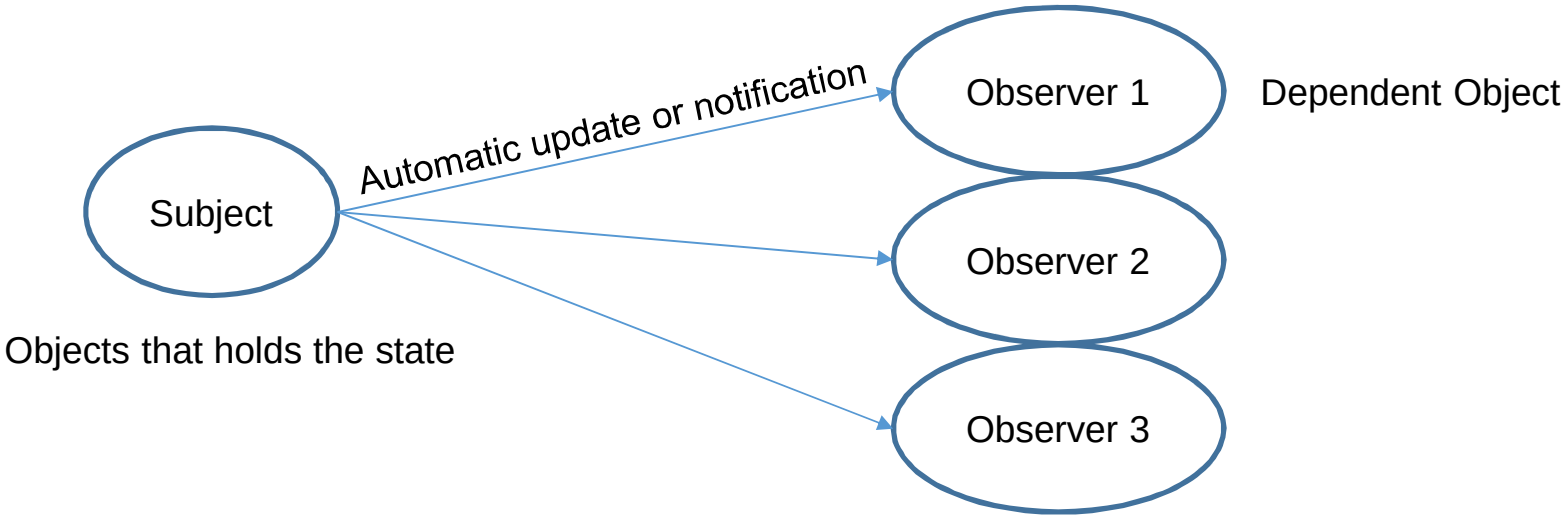
Weather Service

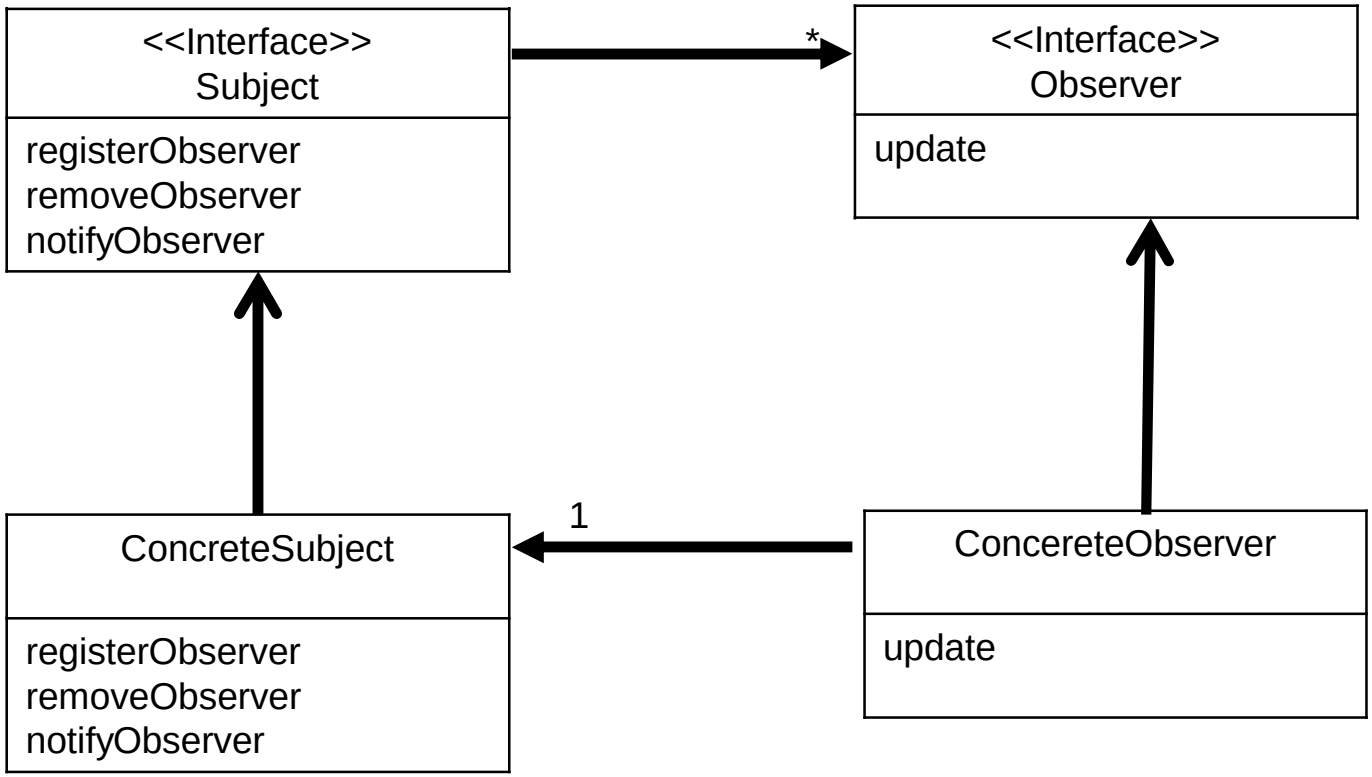
```
public class WeatherData {

// instance variable declarations

public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
// other WeatherData methods here

}
```

Every new observer will require a change here

Observer Pattern

- The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.




<<Interface>>
Subject
registerObserver
removeObserver
notifyObserver
*
<<Interface>>
Observer
update
1
ConcreteSubject
registerObserver
removeObserver
notifyObserver
ConcereteObserver
update

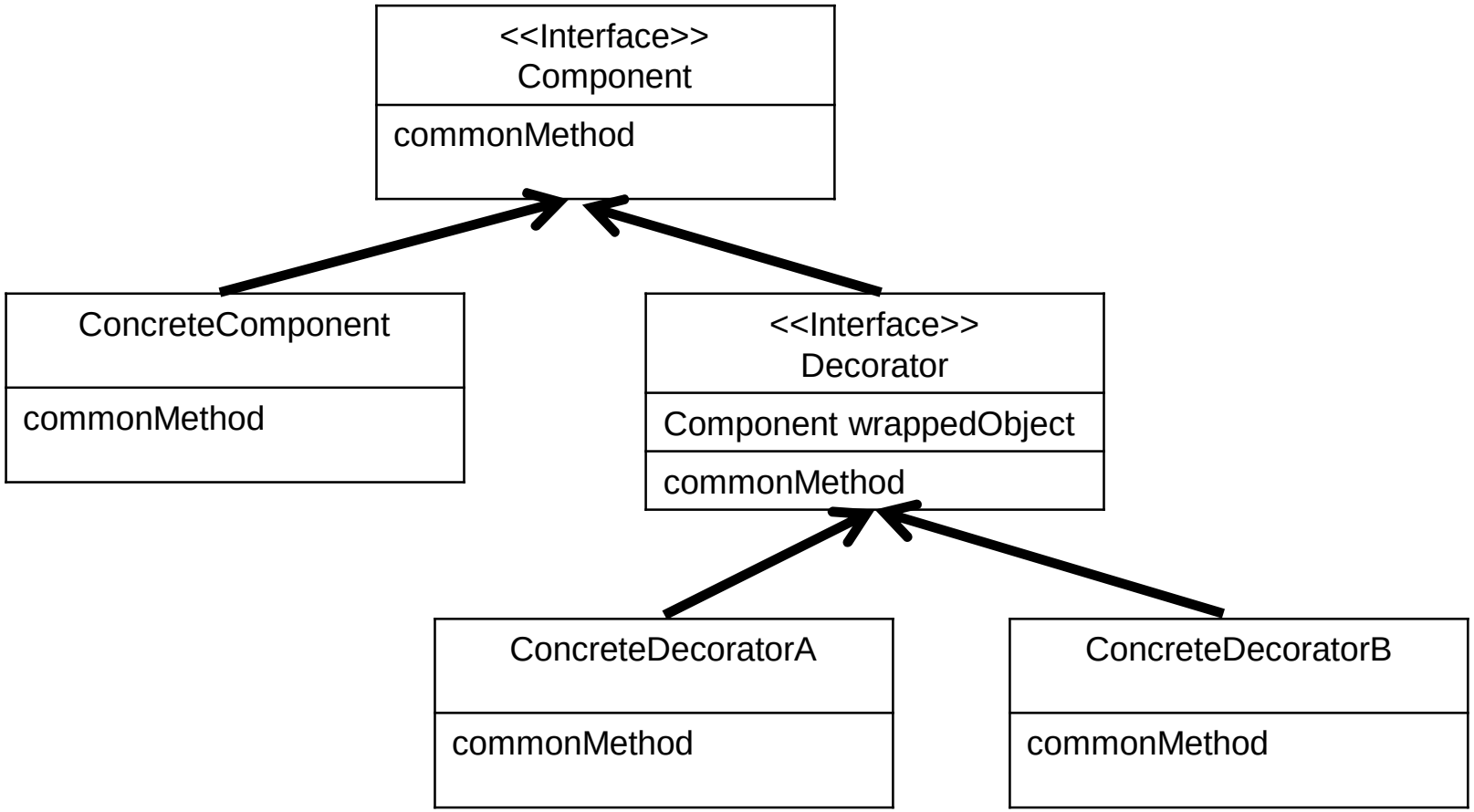
Use Case: Streams

- FileInputStream
- ByteArrayInputStream
- ObjectInputStream
- DataInputStream
- BufferedInputStream
- PushbackInputStream
- Zip
- Cipher

Class Explosion

Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.


<<Interface>>
Component
commonMethod
ConcreteComponent
commonMethod
<<Interface>>
Decorator
Component wrappedObject
commonMethod
ConcreteDecoratorA
commonMethod
ConcreteDecoratorB
commonMethod

Use Case

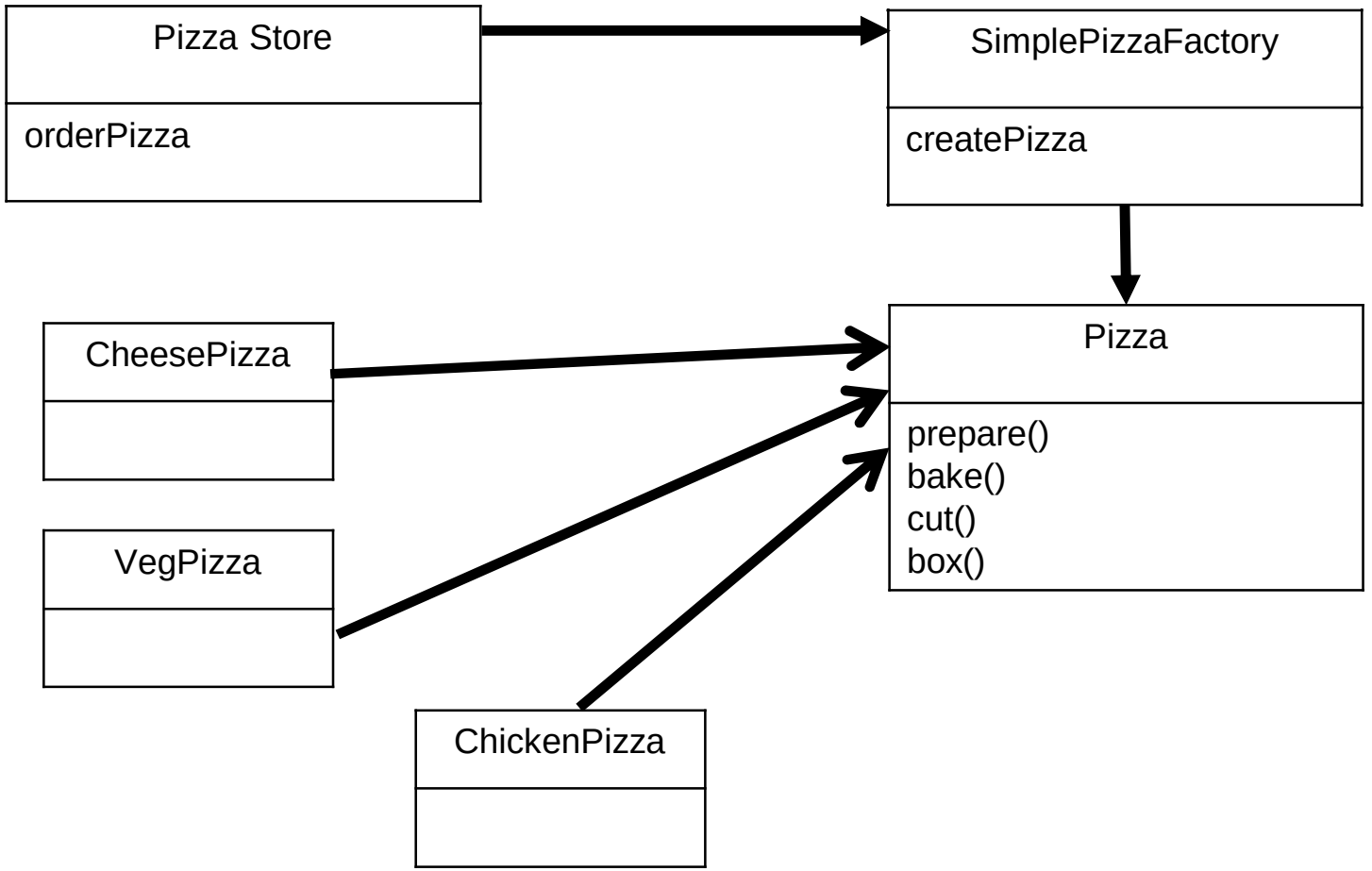
```
Pizza pizza;
if (type.equals(“cheese”)) {
        pizza = new CheesePizza();
} else if (type.equals(“greek”) {
        pizza = new GreekPizza();
} else if (type.equals(“pepperoni”) {
        pizza = new PepperoniPizza();
}
```

new is always concrete and brings in inflexibility

Factory

```
public class SimplePizzaFactory {
        public Pizza createPizza(String type) {
                Pizza pizza = null;
                if (type.equals(“cheese”)) {
                        pizza = new CheesePizza();
                } else if (type.equals(“pepperoni”)) {
                        pizza = new PepperoniPizza();
                } else if (type.equals(“clam”)) {
                        pizza = new ClamPizza();
                } else if (type.equals(“veggie”)) {
                        pizza = new VeggiePizza();
                }
                return pizza;
        }
}
```

```
public class PizzaStore {
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
  }
  public Pizza orderPizza(String type) {
    Pizza pizza;
    pizza = factory.createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
   return pizza;
 }
```


Pizza Store
orderPizza
SimplePizzaFactory
createPizza
Pizza
prepare()
bake()
cut()
box()
CheesePizza
VegPizza
ChickenPizza

Factory Method Pattern

- The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate.

- Factory Method lets a class defer instantiation to subclasses.

- Decouples the implementation of the product from its use. If you add additional products or change a product’s implementation, it will not affect your Creator

Private Constructor

- This is legal

```
public MyClass {
        private MyClass() {}
}
```

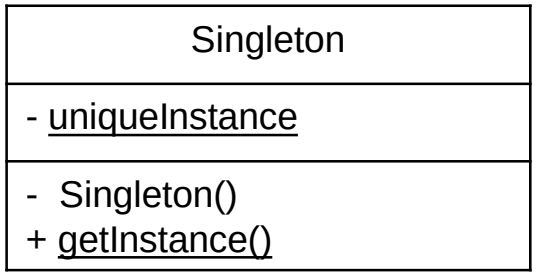
- Who can use this constructor?

Another style of factory

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

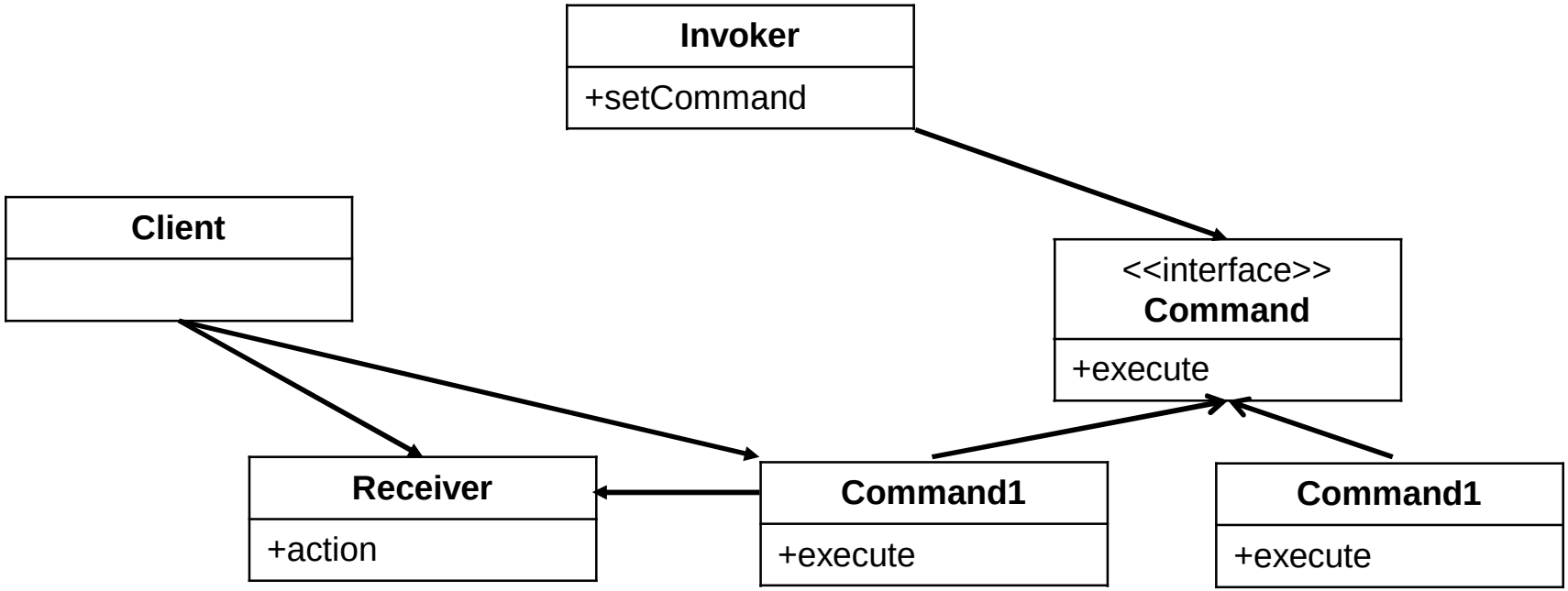
Singleton

```
public class Singleton {
    private static Singleton uniqueInstance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```



Encapsulating Functions

- Decouple object making a request from the objects that receive and execute those requests.
- Generic Remote
- Pre and Post Operations: Undo, Log
- Function Pointers


Invoker
+setCommand
Client
<<interface>>
Command
+execute
Receiver
+action
Command1
+execute
Command1
+execute

Command Pattern

- The Command Pattern decouples an object, making a request from the one that knows how to perform it.
- Command object encapsulates a receiver with an action (or set of actions) .
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.

Difference between Observer and Command

- Less in structure, more in intention.
- Command models an action to do. May or may not execute now.
- Observer notifies other objects about certain events. Does not know exactly which or how many objects to notify.
- Example: Menu. Exactly one action but command can support undo/logging
- You may observers that have commands they execute!

Use Case

- You have a library with an old interface. Rest of the system can only use library with a new API.

- Convert the interface of a class into another interface that client expects.

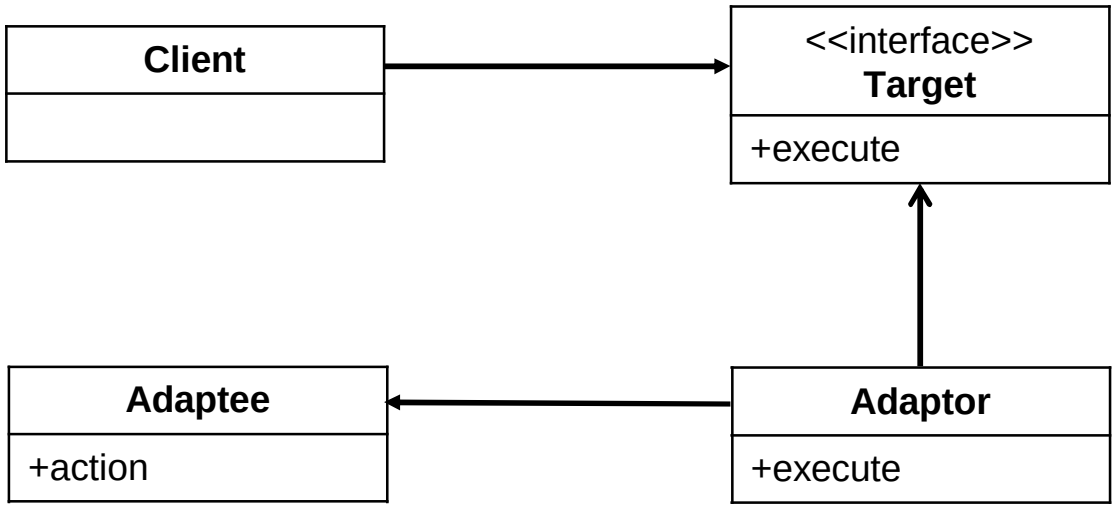
- Wrap an existing class with a new interface.

Adaptor Pattern

- The Adapter Pattern converts the interface of a class into another interface the clients expect.

- Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

- Class Adaptor Vs Object Adaptor


Client
<<interface>>
Target
+execute
Adaptee
+action
Adaptor
+execute

Decorator Vs Adaptor Pattern

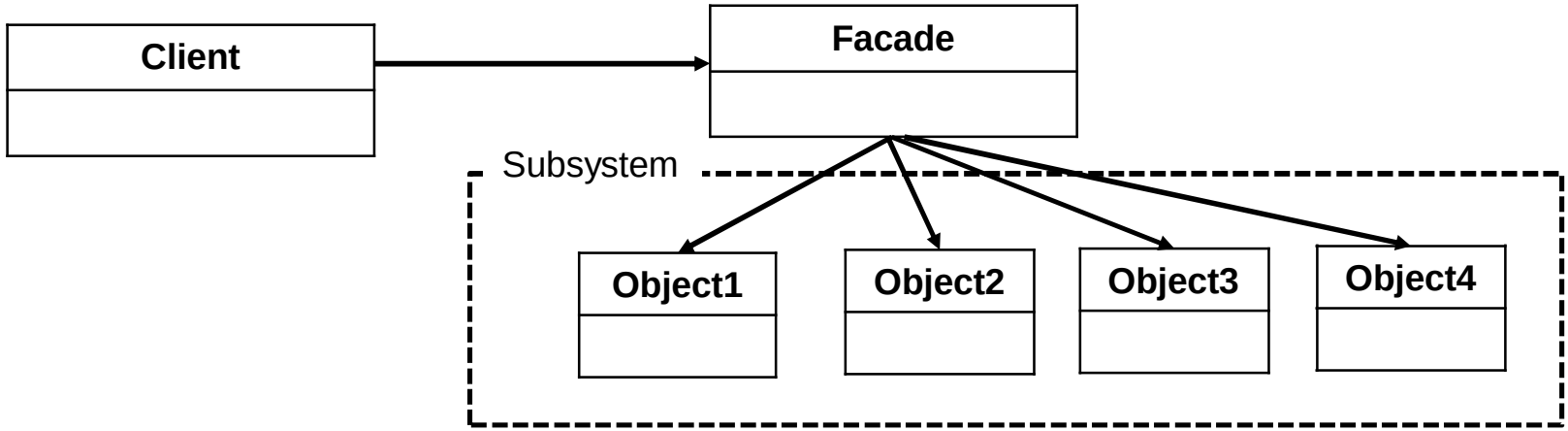
- A Decorator adds new features to the object it wraps.
- Adapter changes the interface of the object it wraps.

Use Case

- There is a complex system. Each operation requires handling multiple calls from various objects.
- CRM
- Helper routines

Facade Pattern

GoF: Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.



Facade Pattern

- Doesn’t hide subsystem interfaces from the client.
- Facade is usually applied when the number of interfaces grow and system gets complex.
- Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.
- Purpose of Facade pattern is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.

Facade Pattern

Decorator	Do not change the interface, but adds responsibility
Adapter	Convert one interface to another
Facade	Make interface simpler

Encapsulating Algorithms

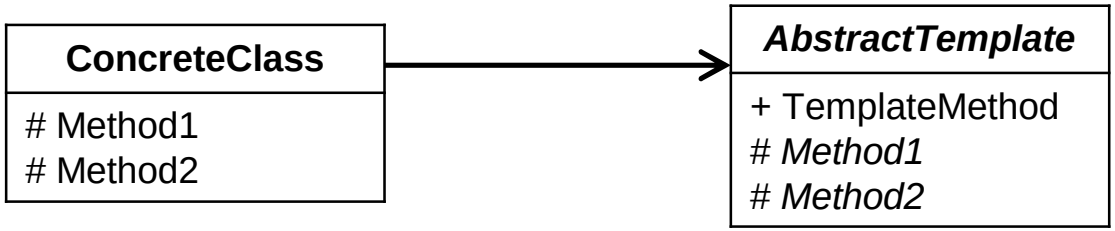
- Encapsulate the recipe
- Subclasses can hook into the recipe and override the steps.
- Shortest path algorithm…

Template Pattern

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

- Encapsulate interchangeable behaviors and use delegation to decide which behavior to use


ConcreteClass
Method1
Method2
AbstractTemplate
+ TemplateMethod
# Method1
# Method2

Use Case

- Application data and logic
- Servlet
- JSP/Servlet: HTML

Use Case

Document View Architecture

- Document: SDI / MDI
- View: CView, Scrollbar…
- Application: CFrameWnd/CWinApp

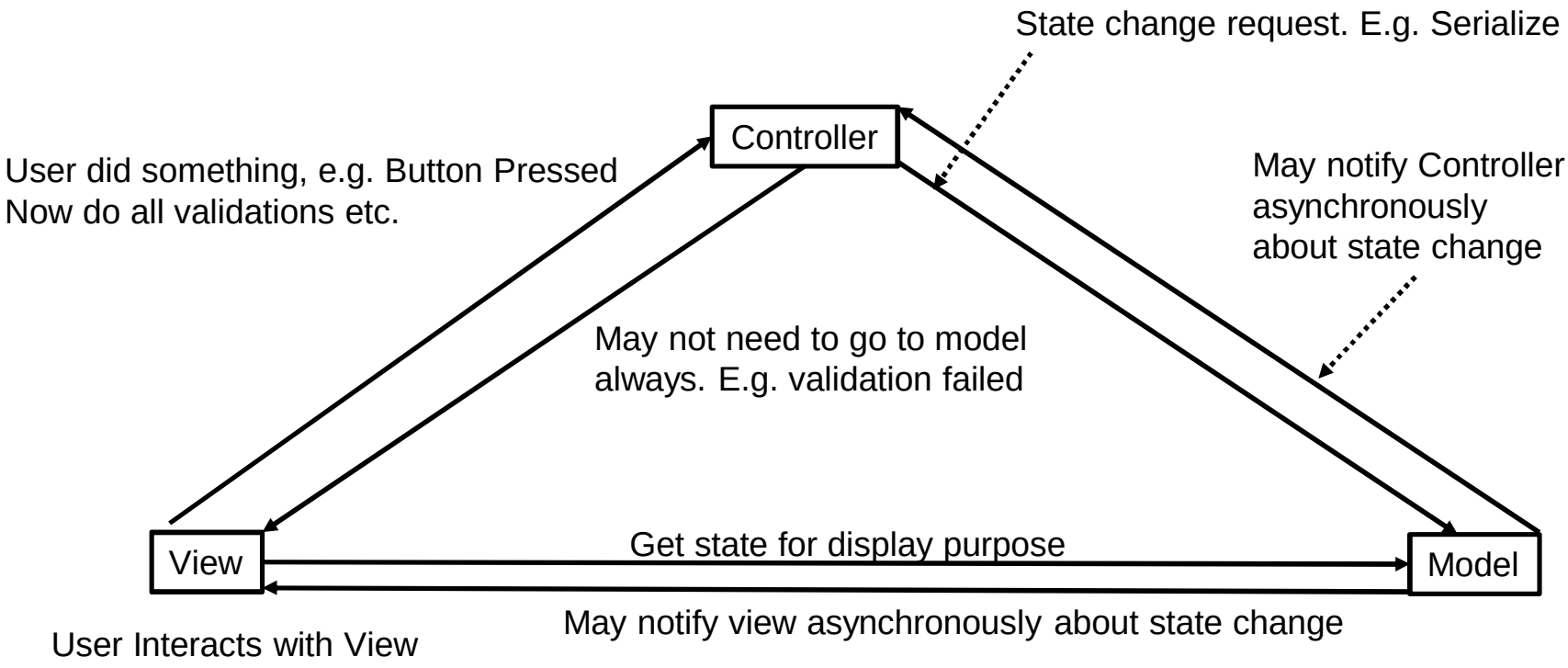
MVC Pattern

- Model: Holds the data, state and application logic. The model is oblivious to the view and controller. Provides an interface to manipulate and retrieve its state

- Controller: Takes user input and figures out what it means to the Model.

- View: A presentation of the model. The view usually gets the state and data it needs to display directly from the model.

MVC Pattern


State change request. E.g. Serialize
Controller
User did something, e.g. Button Pressed
Now do all validations etc.
May notify Controller
asynchronously
about state change
May not need to go to model
always. E.g. validation failed
View
Get state for display purpose
Model
May notify view asynchronously about state change
User Interacts with View

Use Case

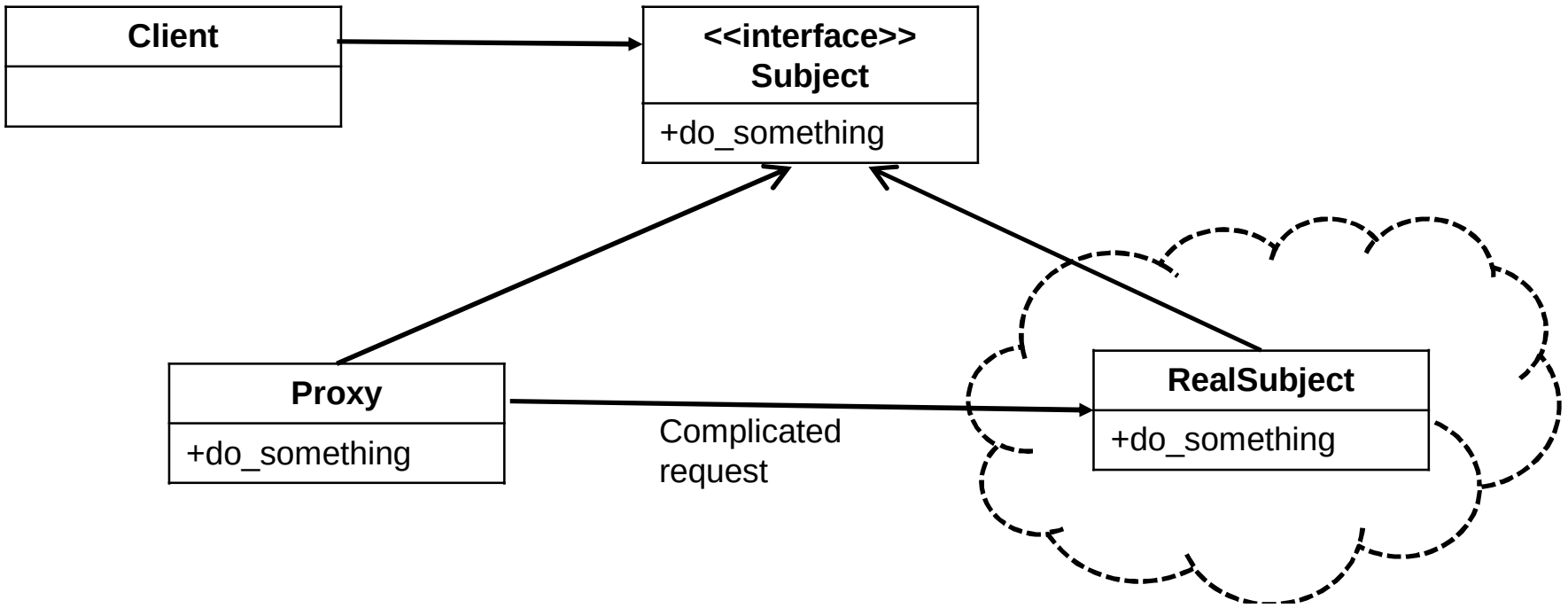
- Java RMI
- Remote Procedure Calls
- Stubs and Skeletons

Proxy Pattern

The Proxy Pattern

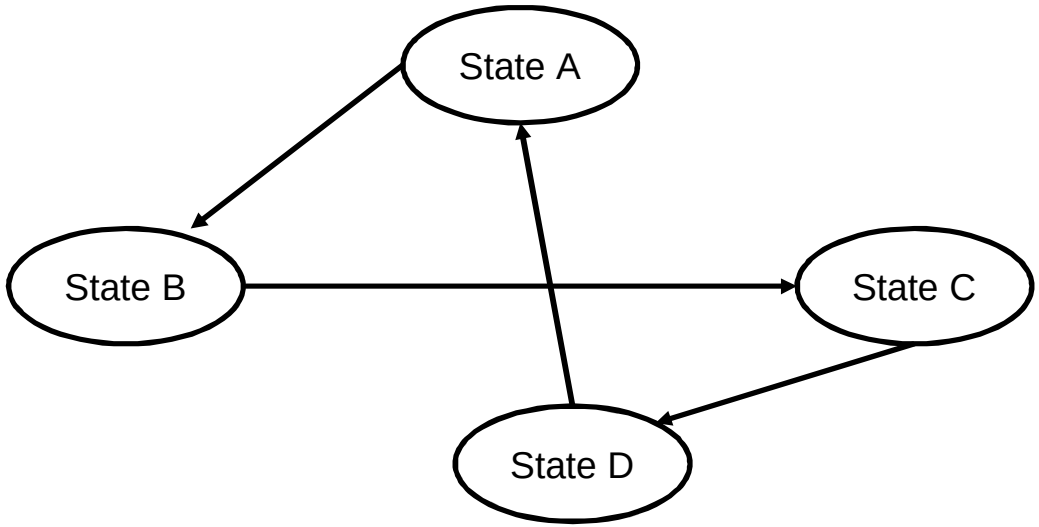
provides a surrogate or placeholder for another object to control access to it.

Proxy Pattern

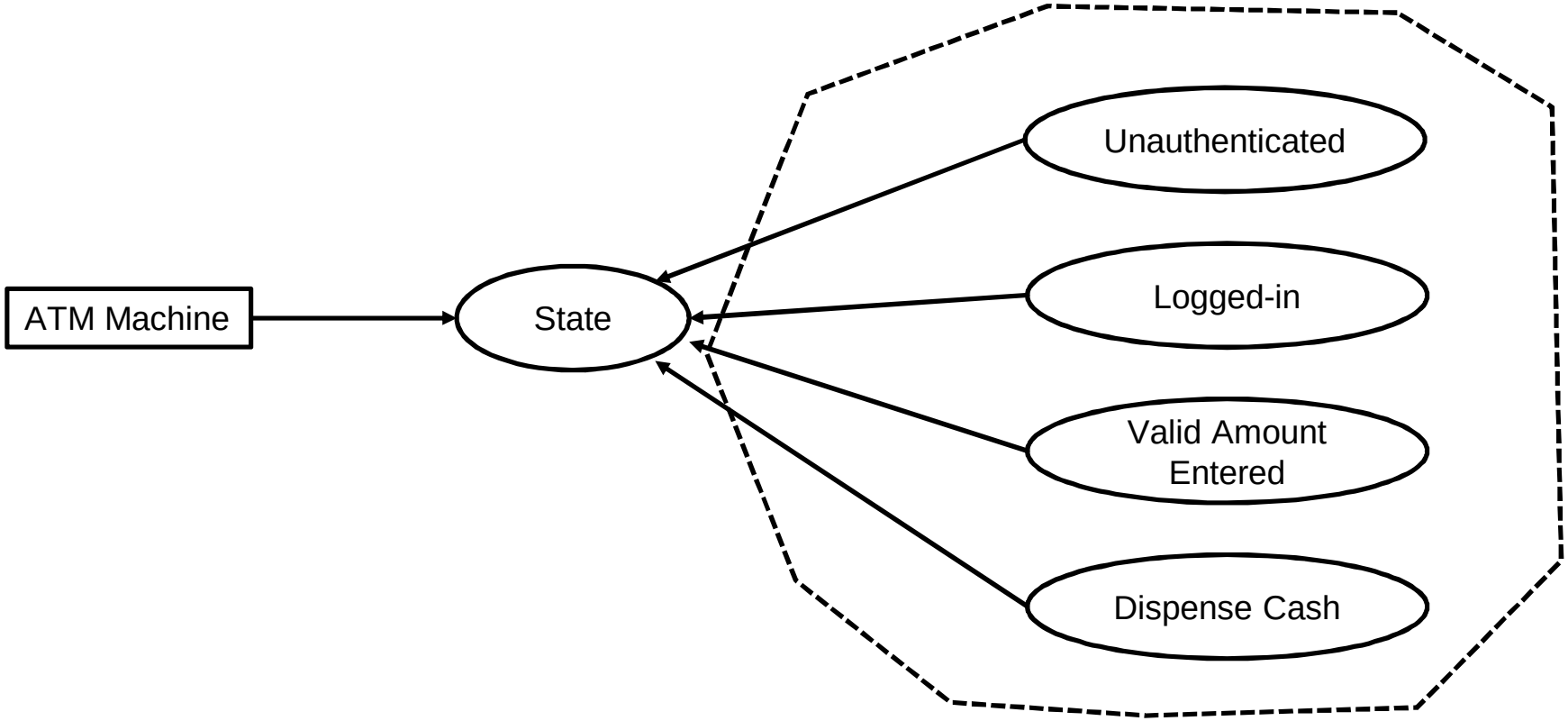


Use Case

- State Diagrams: ATM Machine



ATM Machine


ATM Machine
State
Unauthenticated
Logged-in
Valid Amount Entered
Dispense Cash

State Pattern

The State Pattern

allows an object to alter its behavior when its internal state changes.

The object appears to change its class.

Use Case

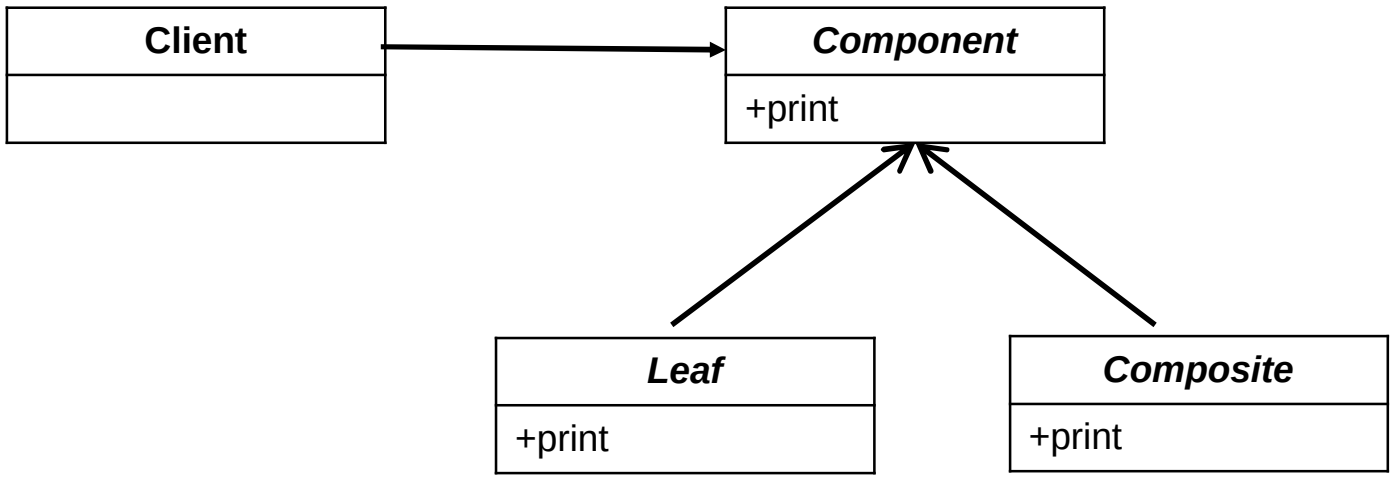
- Common Operation for set of objects
- Print, Add, Modify etc.
- E.g. Menu, Collections

Composite Pattern

The Composite Pattern

allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite Pattern



Summary

1. State	A. Wraps an object and provides a different interface to it.
2. Decorator	B. Ensures one and only object is created.
3. Facade	C. Clients treat collections of objects and individual objects uniformly.
4. Proxy	D. Wraps an object to control access to it.
5. Factory	E. Simplifies the interface of a set of classes.
6. Adaptor	F. Wraps an object to provide new behavior
7. Observer	G. Subclasses decide which concrete classes to create.
8. Template	H. Allows objects to be notified when state changes
9. Composite	I. Subclasses decide how to implement steps in an algorithm.
10. Singleton	J. Encapsulates a request as an object.
11. Command	K. Ensures one and only object is created.