GoF Pattern Catalog

Creational patterns (creating objects): factory, prototype, *singleton*

Structural patterns (connecting objects): *adapter, composite*, bridge, decorator, proxy, facade

Behavioural patterns (distributing duties): chain of responsibility, *command*, iterator, *observer*, state, *template method*

Singleton Pattern: ensure a class has only one instance, and provide global point of access to it.

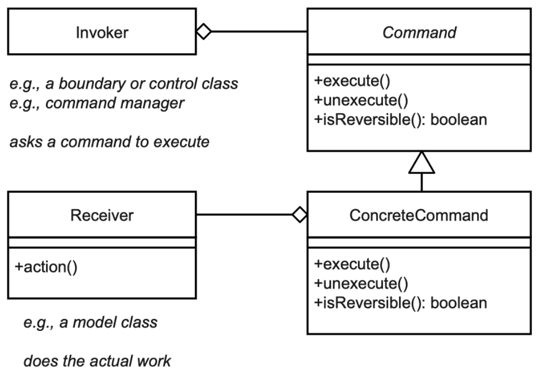
Composite Pattern: composite individual objects to build up a tree structure

- a folder can have files and other folders, "recursive composition"

Command Pattern: encapsulate a request as an object, so you can run, queue, log, undo/redo requests

- motivation: a class may want issue a request without knowing anything about operation/receiver

Command Pattern Structure



Coupling: how much a class needs another. (e.g. a chicken doesn't need to know where it lives).

Cohesion: the degree of connectivity among the elements within the same class.

Template Method Pattern

Design intent: define the skeleton of an algorithm in a method, deferring (推迟) some steps to subclasses.

Consequences: inverted control: superclass method calling subclass method

Hooks: methods in the superclass which provide default behaviour that the superclasses may override (optional method). Often do nothing by default.

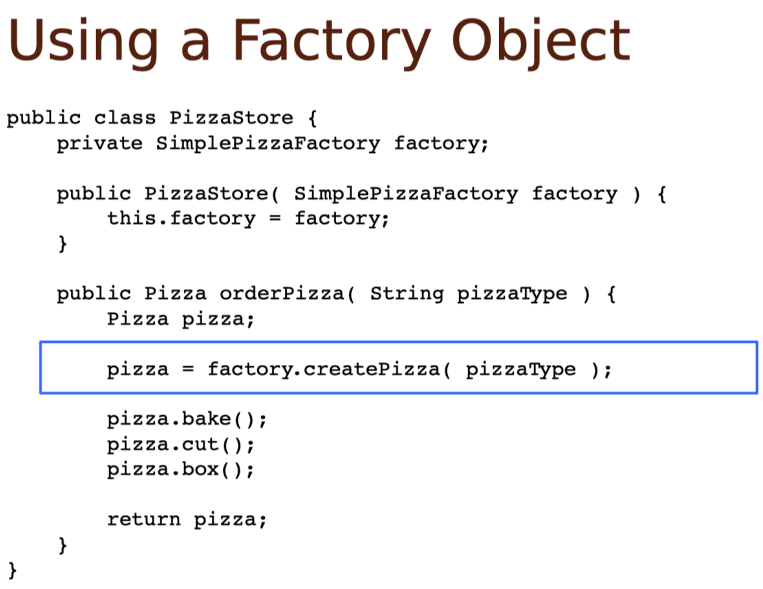
Why Template Method?

Before:	After:
Coffee and Tea have the algorithm	HotCaffeineBeverage has the algorithm
near duplicated code in Coffee and Tea	reduces duplication and enhances reuse
changing the algorithm requires opening the subclasses and making multiple changes	algorithm is found in one place, so changes to it are localized

Before:	After:
original structure requires more work to add a new subclass (need to provide the whole algorithm again)	new structure provides a framework to add a new subclass (need to provide just the distinctive parts of the algorithm)

Factory Method Pattern

Design intent: define an interface for creating an object, but let subclasses decide which actual class to instantiate.



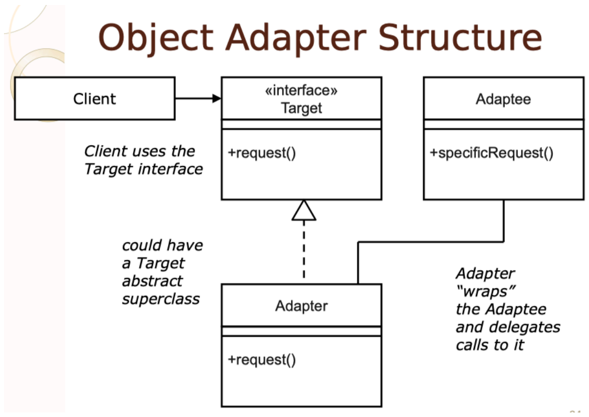
Using a Factory Object

```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore( SimplePizzaFactory factory ) {
        this.factory = factory;
    }

    public Pizza orderPizza( String pizzaType ) {
        Pizza pizza;

        pizza = factory.createPizza( pizzaType );

        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Adapter Pattern

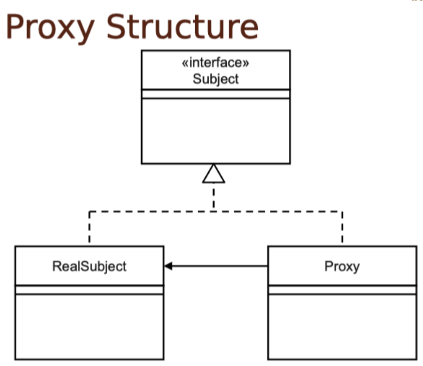
Design intent: convert the interface of a class into another interface that clients expect.

Consequences: object adapter: more flexible since a single Adapter could adapt many adaptees. Class adapter: related to Adaptee via implementation inheritance, can override Adaptee.


Object Adapter Structure
Client
«interface» Target
+request()
Adaptee
+specificRequest()
Client uses the Target interface
could have a Target abstract superclass
Adapter
+request()
Adapter "wraps" the Adaptee and delegates calls to it

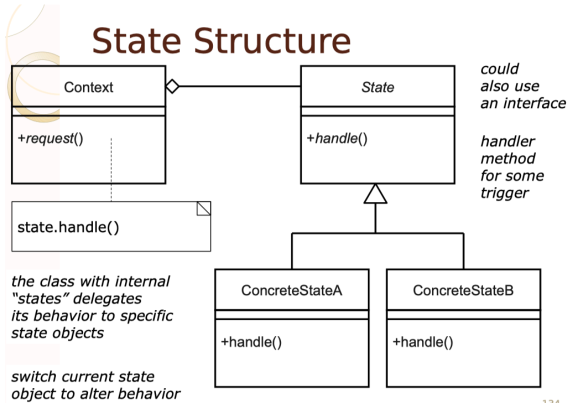
Proxy Pattern

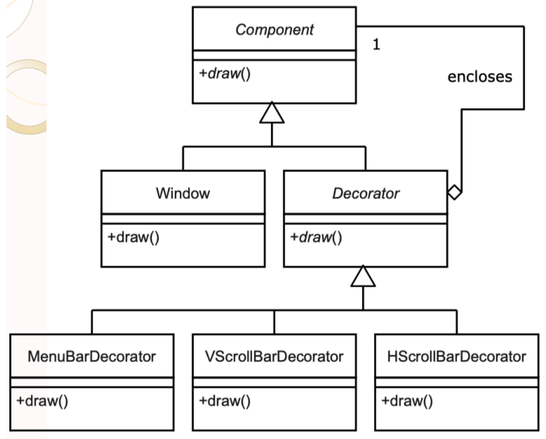
Design intent: provide a surrogate or place holder for another object to control access to it.
Motivation: defer the full cost of creation and initialization of an object until we actually need to use it.



State Pattern

Design intent: allow an object to alter its behavior when its internal state changes, simplify operations with long conditionals that depend on the object's state.


State Structure
Context
+request()
state.handle()
State
+handle()
could also use an interface
handler method for some trigger
ConcreteStateA
+handle()
ConcreteStateB
+handle()
the class with internal "states" delegates its behavior to specific state objects
switch current state object to alter behavior

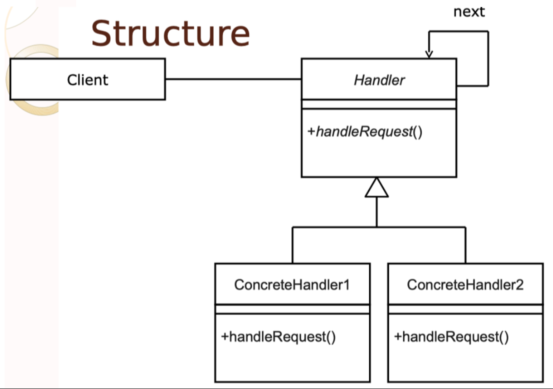


Decorator Pattern

Design intent: attach additional responsibilities to an object dynamically (e.g. UI components).

Motivation: UI embellishments; don't want too many new subclasses; use aggregation instead of inheritance.

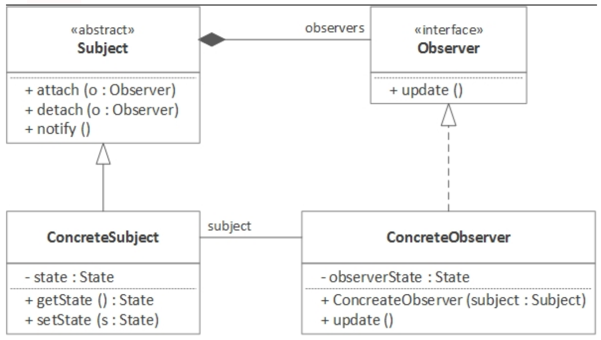
Chain of Responsibility Pattern



Design intent: avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. "chain the receiving objects and pass the requests along the chain until an object handles it".

Consequences: reduced coupling: frees an object from knowing which other object handles a request; sender and receiver do not have direct knowledge about each other.

Observer Pattern



Design intent: when a change in one object requires changing other objects and also you don't know how many and what types of objects needed to be changed. Also you used it when an object M should be able to notify other objects without making assumptions about who they are or what they are, you don't want them to be tightly coupled.

Design principles

Goal: enhance flexibility under changing needs; Improve reusability in different contexts.

Open Closed Principle: Classes should be open for extension, but closed for modification. Encapsulate varying part from stable part.

Dependency Inversion Principle: Depend upon abstractions instead of concrete classes. Program to interfaces, not implementations. Favor composing objects over implementation inheritance. (Composing object: run-time dependency; black box "arms length" reuse via well defined interfaces; delegation. Implementation inheritance: compile-time dependency; white box reuse of superclass; tight coupling, limits reuse of only subclass.)

Principle of Lease Knowledge: for an object, reduce the number of classes it knows about and interacts with. Coupling reduced and changes cascading throughout the system. Law of Demeter (avoid calling methods of objects returned by other methods): for method M of object O, only call methods of the following objects: object O itself; parameters of M; any objects instantiated within M; direct component objects of O.

Refactoring

Idea: change a software system so that the external behavior does not change but the internal structure is *improved*.

Consequences: more OO: decomposes big methods into smaller ones; distributes responsibilities among classes. More code. Slower performance?

Duplicated Code: same functionality appears in more than 1 places. Refactoring: Extract method, Pull up method.

Long Method: long, hard-understanding methods. Refactoring: Extract method.

Large (blob/god) Class: a class trying to do too many things. Refactoring: Extract class.

Divergent Change: a class is commonly changed in different ways for different reasons. Refactoring: Extract Class.

Shotgun Surgery: making a change requires many little changes across many different classes or methods. Refactoring: Move Method.

Long Parameter List: passing in lots of parameters to a method. Refactoring: Replace parameter with method; Introduce parameter object.

Feature Envy (嫉妒): a method seems more interested in the details of a class other than the one it is in. Refactoring: Move method; Extract method.

Data Class: classes that are just data (manipulated by other classes with getters and setters). Refactoring: Encapsulate field; Extract method; Move method.

Data Clumps: groups of data appearing together in the instance variables of classes, parameters to methods, etc. Refactoring: Extract class; Introduce parameter object.

Primitive Obsession: using the built-in types too much (non-OO design). Refactoring: Replace data value with object.

Switch Statements: long conditionals on type codes defined in other classes. Refactoring: Extract method, Move method; Replace type code; Replace conditional with polymorphism.

Speculative Generality: “we might need this someday”. Refactoring: Collapse hierarchy; Remove parameter.

Message Chains: long chains of navigation to get to an object. Refactoring: Hide delegate.

Inappropriate Intimacy (亲密): two classes that depend too much on each other, with lots of bidirectional communication. Refactoring: Move method; Extract class.

Refused Bequest: When a subclass inherits something that is not needed; When a superclass does not contain truly common state or behavior. Refactoring: Push down method and push down field; Replace inheritance with delegation.

Refactoring Principles: catalog of refactoring; do not change outward behavior; reduce risk of change; one thing at a time; TEST each step; iterate. **Outcomes:** encode design intent within class structure; reorganizing code; sharing logic; express conditional logic. **Potential limitations:** too much indirection; performance impact; changing published interfaces. **When not to refactor:** when you should rewrite; when you are close to a deadline.

Redesigning with Patterns: creating an object by naming the class directly, fix with Abstract Factory or Factory Method; dependence on specific hard-core requests, fix with Chain of Responsibility or Command; algorithmic dependencies, fix with Template Method; tight coupling, fix with Observer or Facade; too much subclassing, fix with Decorator.

Kinds of Refactoring: Creating methods: reduce size of method and improve readability of code, use Extract Method, Inline Method, Replace Temp with Query. Moving features b/w objects: wrong place for some responsibilities or a class has too many responsibilities, use Move Method, Move Field, Extract Class. Organizing data: sometimes objects can be used instead of simple data items, use Replace Data Value with Object, Replace Array with Object. Simplifying conditional expressions: conditional expressions and logic can be difficult to understand, use Replace Conditional with Polymorphism. Making method calls simpler: complicated programming interfaces can be difficult to use, use Rename Method, Add Parameter. Dealing with generalization: getting methods and subclasses to the right place, use Pull Up Method, Push Down Method, Extract Subclass, Extract Superclass.

Java General Techniques: Prefer polymorphism to *instanceof* (more extensible code); Set object references to *null* when they are no longer needed (save memory usage); Make immutable classes (stable class), *private* data, set all data in constructor, declare the class *final*. Use inheritance or delegation to define immutable classes from mutable ones.

Effective Java: Methods common to all objects. Classes and interfaces: minimize accessibility of classes and members, prefer interfaces to abstract classes, use interfaces only to define types. Methods: check parameters for validity, make defensive copies, return zero-length arrays instead of *null*s.

Delegation vs Inheritance in Java

Inheritance in Java programming is the process by which one class takes the property of another other class. i.e. the new classes, known as derived or child class, take over the attributes and behavior of the pre-existing classes, which are referred to as base classes or super or parent class.

Delegation is simply passing a duty off to someone/something else.

- Delegation can be an alternative to inheritance.
- Delegation means that you use an object of another class as an instance variable, and forward messages to the instance.
- It is better than inheritance for many cases because it makes you to think about each message you forward, because the instance is of a known class, rather than a new class, and because it doesn't force you to accept all the methods of the super class: you can provide only the methods that really make sense.
- Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate.
- The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time. But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn't facilitate dynamic polymorphism.

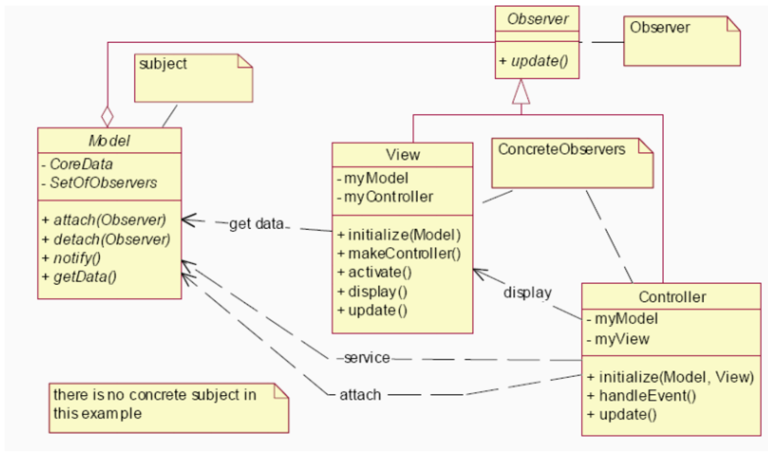
MVC & Observer Pattern

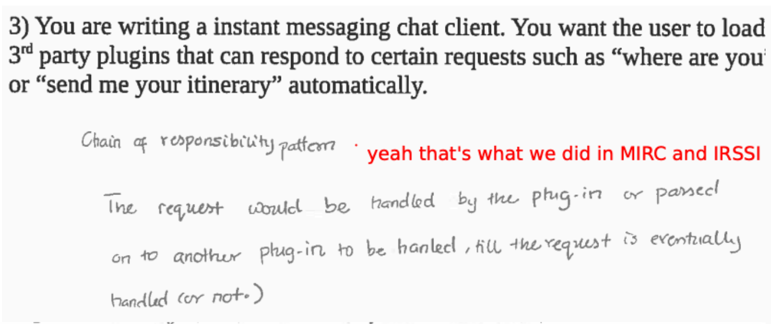
MVC is more an architecture style rather than a design pattern.

Observer Design pattern is a Behavioural pattern which is used when we want to notify all of the dependents of an object(say x) in the event of change of the object x. **Using observer pattern keeps model completely independent of views and controllers. It allows us to use different views with same model, or even multiple views at once.**

And they both are closely related, as MVC you would see from MVC diagram - for example: A Change in 'View' Has to be notified to 'Model' and 'Controller' One efficient way to achieve such feature is Observer design pattern.

The Observer Pattern in the MVC

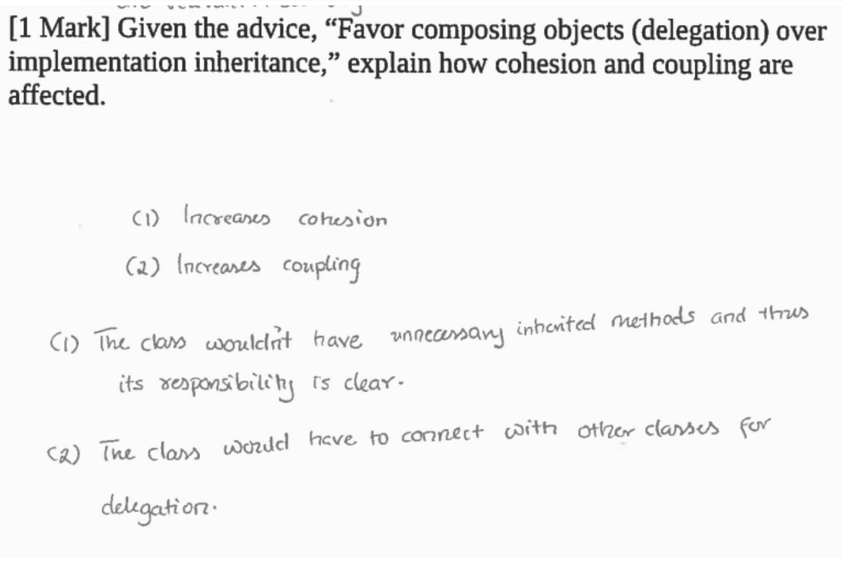



3) You are writing a instant messaging chat client. You want the user to load 3rd party plugins that can respond to certain requests such as "where are you" or "send me your itinerary" automatically.

Chain of responsibility pattern

yeah that's what we did in MIRC and IRSSI

The request would be handled by the plug-in or passed on to another plug-in to be hanled, till the request is eventually handled (or not.)

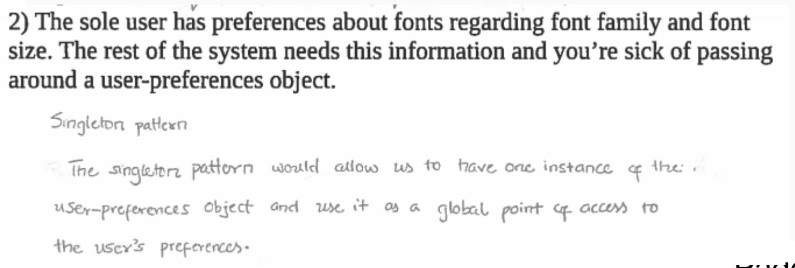

[1 Mark] Given the advice, "Favor composing objects (delegation) over implementation inheritance," explain how cohesion and coupling are affected.

(1) Increases cohesion

(2) Increases coupling

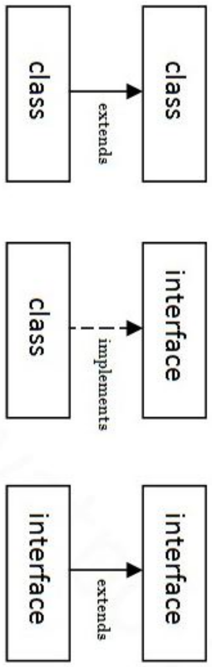
(1) The class wouldn't have unnecessary inherited methods and thus its responsibility is clear.

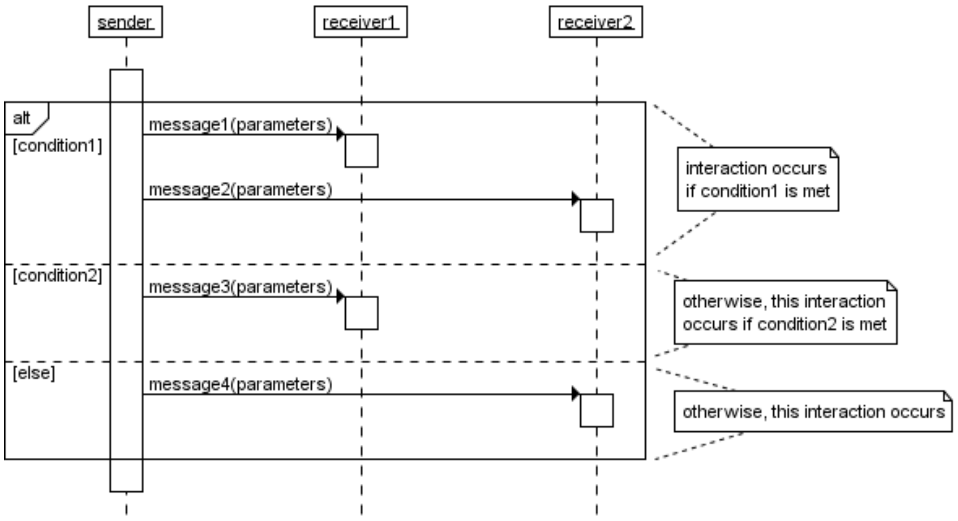
(2) The class would have to connect with other classes for delegation.

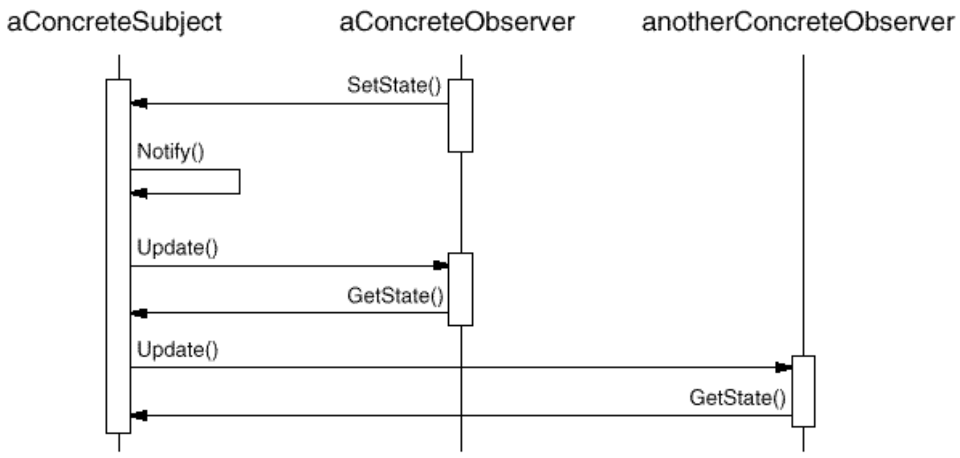

2) The sole user has preferences about fonts regarding font family and font size. The rest of the system needs this information and you're sick of passing around a user-preferences object.

Singleton pattern

The singleton pattern would allow us to have one instance of the user-preferences object and use it as a global point of access to the user's preferences.





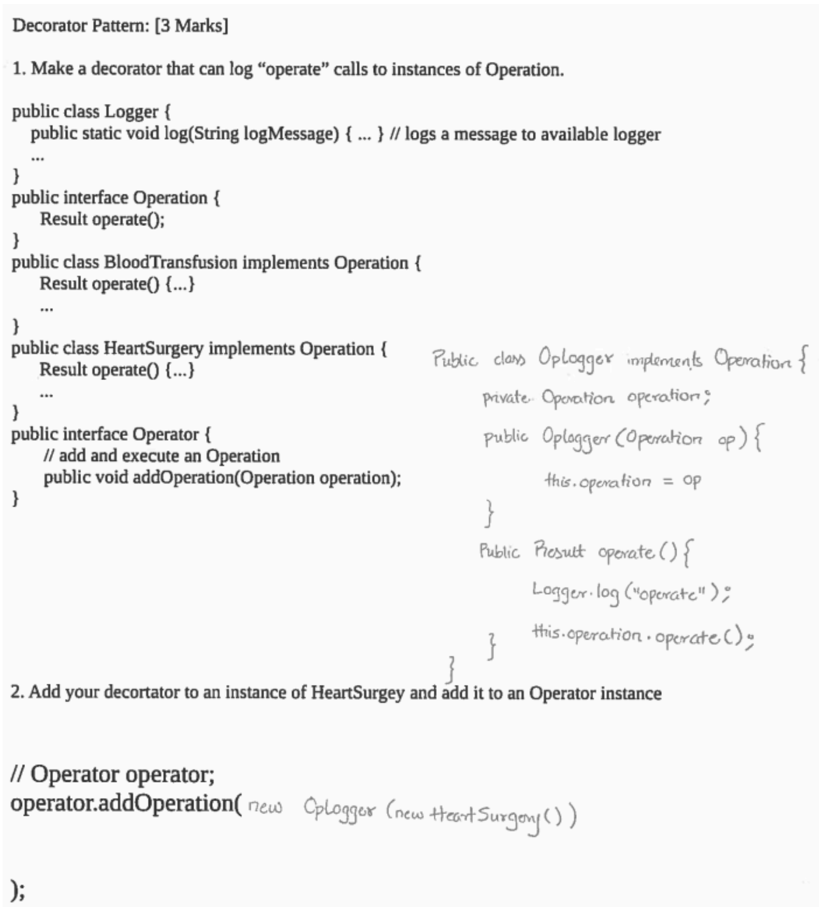


Polymorphism (Before)

```
class Price {
    …
    public double getCharge( int daysRented ) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
    …
}
```

Polymorphism (After)

```
class RegularPrice {
    public double getCharge( int daysRented ) {
        double result = 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        return result;
    }
}
class NewReleasePrice {
    public double getCharge( int daysRented ) {
        return daysRented * 3;
    }
}
class ChildrensPrice {
    public double getCharge( int daysRented ) {
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    }
}
```



Decorator Pattern: [3 Marks]

1. Make a decorator that can log "operate" calls to instances of Operation.

```
public class Logger {
    public static void log(String logMessage) { ... } // logs a message to available logger
    ...
}
public interface Operation {
    Result operate();
}
public class BloodTransfusion implements Operation {
    Result operate() {...}
    ...
}
public class HeartSurgery implements Operation {
    Result operate() {...}
    ...
}
public interface Operator {
    // add and execute an Operation
    public void addOperation(Operation operation);
}
```

```
Public class OpLogger implements Operation {
    private Operation operation;
    public OpLogger(Operation op) {
        this.operation = op
    }
    Public Result operate() {
        Logger.log("operate");
        this.operation.operate();
    }
}
```

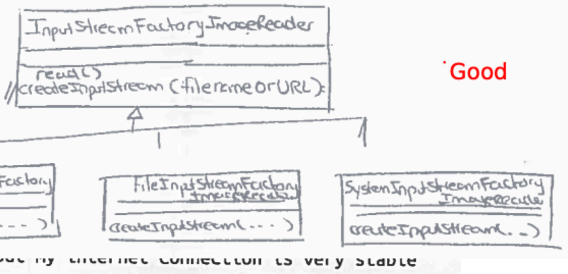
2. Add your decortator to an instance of HeartSurgey and add it to an Operator instance

```
// Operator operator;
operator.addOperation( new OpLogger (new HeartSurgery())
);
```

```
class ImageReader {
      ...
      Image read(String filenameOrURL) {
            InputStream in = null;
            if (isURL(filenameOrURL)) {
                  in = new HttpInputStream( this.filename );
            } else if (filenameOrURL.equals("STDIN")) {
                  in = System.in;
            } else if (isFile(filenameOrURL)) {
                  in = new FileInputStream( this.filename );
            } else {
                  throw new InvalidFileSpecException();
            }
            Image image = Image.imageFromStream( in );
            return image;
      }
      abstract createInputStream(String filenameOrURL)
}
```

factory method

createInputStream()



InputStreamFactoryImageReader
read()
//createInputStream(filenameOrURL):
URLInputStreamFactoryImageReader
createInputStream(- - -)
FileInputStreamFactoryImageReader
createInputStream(- - -)
SystemInputStreamFactoryImageReader
createInputStream(. .)

Good

1) You're making a mind-map program, to model a web knowledge, where users can make entries that can be related to 0 or more other entries.

composite pattern?

2) You're making a programmable text editor in the cloud (on the web), it can be controlled via a webpage or by other software through an API. You have a handful of atomic operations but you want to allow automation of these operations by scripts and services. You want to compose operations together.

Command pattern (composite pattern)

related to operations

```
// Prints 3D Shapes on a 3D printer in plastic
class ThreeDeePrinter {
    ThreeDeePrinter( USBConnection usbConnection ) {
        this.usbConnection = usbConnection;
    }
    boolean extrudeShape3D(Shape3D shape, int triesLeft) {
        try {
            usbConnection.restorePrinterState();
            usbConnection.send(shape);
        } catch (OutOfABSPlasticError e) {
            if (triesLeft > 0 && usbConnection.waitForReload()) {
11:             return extrudeShape3D( shape, triesLeft - 1);
            }
            return false;
        }
    }
    // waits until ABS Plastic spool is reloaded (true) returns false if cancelled
    boolean waitForReload();
    ...
}
interface USBConnection {
    void restorePrinterState();
    void send(Shape3D shape) throws OutOfABSPlasticError;
    boolean waitForReload();
}
class Test3DPrinter extends TestCase {
    void testPrinterRetry() {
        Shape3D shape = new TestShape();
        ThreeDeePrinter printer = new ThreeDeePrinter(new MockUSBConnection());
        assert(false == printer.extrudeShape3D( shape, 0));
        assert(false == printer.extrudeShape3D( shape, 1));
        assert(false == printer.extrudeShape3D( shape, 3));
    }
}
```

No constructor needed

```
class MockUSBConnection implements USBConnection{
    void restore.....(){...}
    void send .......
    boolean waitForReload(){return True;}
```

```
class ImageShare {
    ImageShare(Image image, URL url, HTTPClient client) { ... };
    void upload(int retries) {
        try {
            client.httpPost( imagePostBody() );
            success = true;
        } catch (HTTPNetworkConnectionException e) {
9:          scheduleARetry(retries - 1);
        }
        return success;
    }
    void scheduleARetry(int retry);
    boolean wasSuccessful();
    int retriesLeft();
    ...
}
class TestImageShare extends TestCase {
    void testImageShareScheduleRetry() {
        ImageShare is = new ImageShare( defaultImage, defaultURL,
                                        new MockHttpClient());
        assert(false == is.upload(33));
        assert(retriesList() == 32); // 1 less retry!
    }
}
```

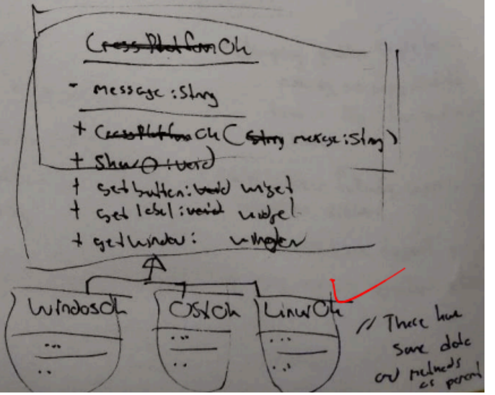
there's a typo in this exam :-/

```
class MockHttpClient {
    void httpPost(...){
        throw new HTTPNetWorkConnectionException();
    }
}
```

CrossPlatformOK displays an OK Button dialogue using native widgets for Win32 and OSX, we're going to add a QT version for Linux. Before we can do this we need to apply two refactorings for CrossPlatformOK to enable the **Factory Method** pattern.
1. Provide ONLY the source code of the new show() method in CrossPlatformOK.
2. Provide the **UML class diagram** and of CrossPlatformOK and related classes after the refactoring.

```
class CrossPlatformOK {
        static int WIN32 = 1;
        static int OSX   = 2;
        String message;
        int platform;
        CrossPlatformOK(String message, int platform) {
                this.message = message;
        }
        void show() {
                if (platform == WIN32) {
                        Widget button = new Win32Button("OK");
                        Widget label  = new Win32Label(message);
                        Window window = new Win32DialogueWindow();
                } else {
                        Widget button = new OSXButton("OK");
                        Widget label  = new OSXLabel(message);
                        Window window = new OSXDialogueWindow();
                }
                window.add(label);
                window.add(button);
                window.show();
        }
}
```

```
Void show() {
    widget button = get button();
    widget label = get label();
window ~~widget~~ window = get window();
    window. add label();
    window. add button();
    window.show();
}
```



~~CrossPlatform~~OK
- message : String
+ ~~CrossPlatform~~OK(~~String~~ message : String)
+ Show() : void
+ get button : ~~void~~ widget
+ get label : ~~void~~ widget
+ getwindow : window
WindowsOK
OSXOK
LinuxOK
// These have same data and methods as parent

```
class PersonTracker {
    Location lastLocation = null;
    // called every second
    void updateLocation(Location l) throws GPSException {
        if (lastLocation!=null) {
            if (lastLocation.distance(l) > 300.0*1000/3600) { // 300km/h in m/s
8:            GPS.getGPS().disable(300);
              throw new GPSException("Too Fast!");
            }
        }
        lastLocation = l;
    }
}
// distance in meters;
interface Location { public double distance(Location l); }
class TestPersonTracker extends TestCase {
    void testTooFast() {
        PersonTracker p = new PersonTracker();
        Location l = new MockLocation(0,0);
        try {
            p.updateLocation(l);
            p.updateLocation(l);
            assert(false, "This was supposed to fail");
        } catch (GPSException e) {
            return; // we succeeded
        }
    }
}
```

```
class MockLocation {
    public double distance (Location l){
        return (300.0*1000/3600)+1;
    }
}
```