



JUMP START TO OOAD & DESIGN PATTERNS (using C++)

Balaji Haridass



Why OOP?

Development/Design Problems?

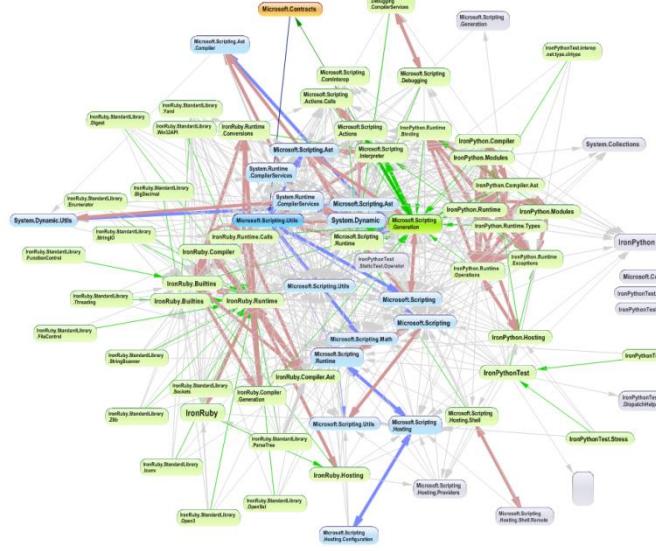


Problems we all have!



CHANGE

Dependency!



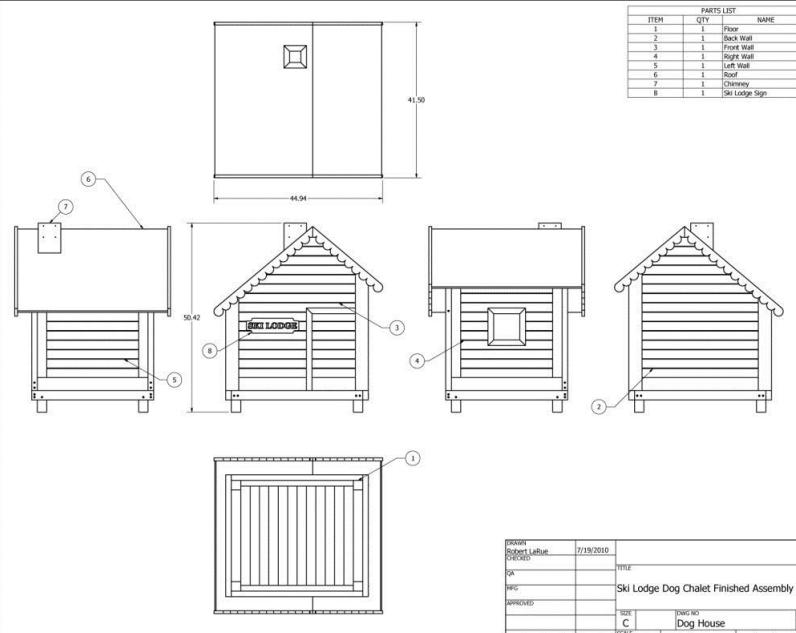
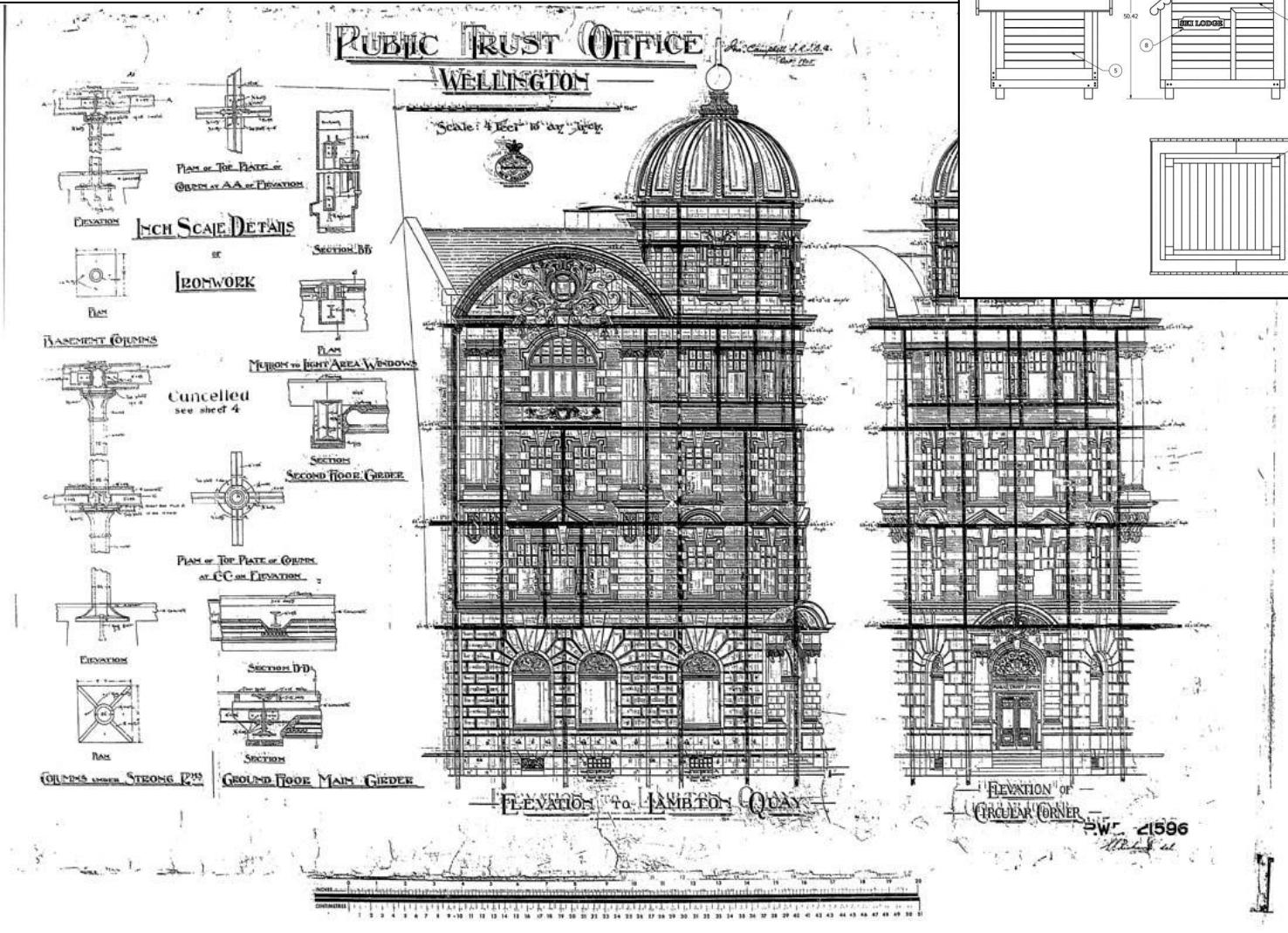
Defects!



Testable?



Problems we all have!



Complexity

Take Away



Will be able to Design that

- Can be RE-USABLE
- Will be EXTENDIBLE
- Will be FLEXIBLE TO CHANGE
- Will be TESTABLE
- Will be UNDERSTABLE



**Who all will you consider while doing Design,
Development, Review?**

OOA&D is about writing great software!



Customer is satisfied when their APPS

- WORK
- KEEP WORKING
- CAN BE UPGRADED



Programmer is satisfied when their APPS

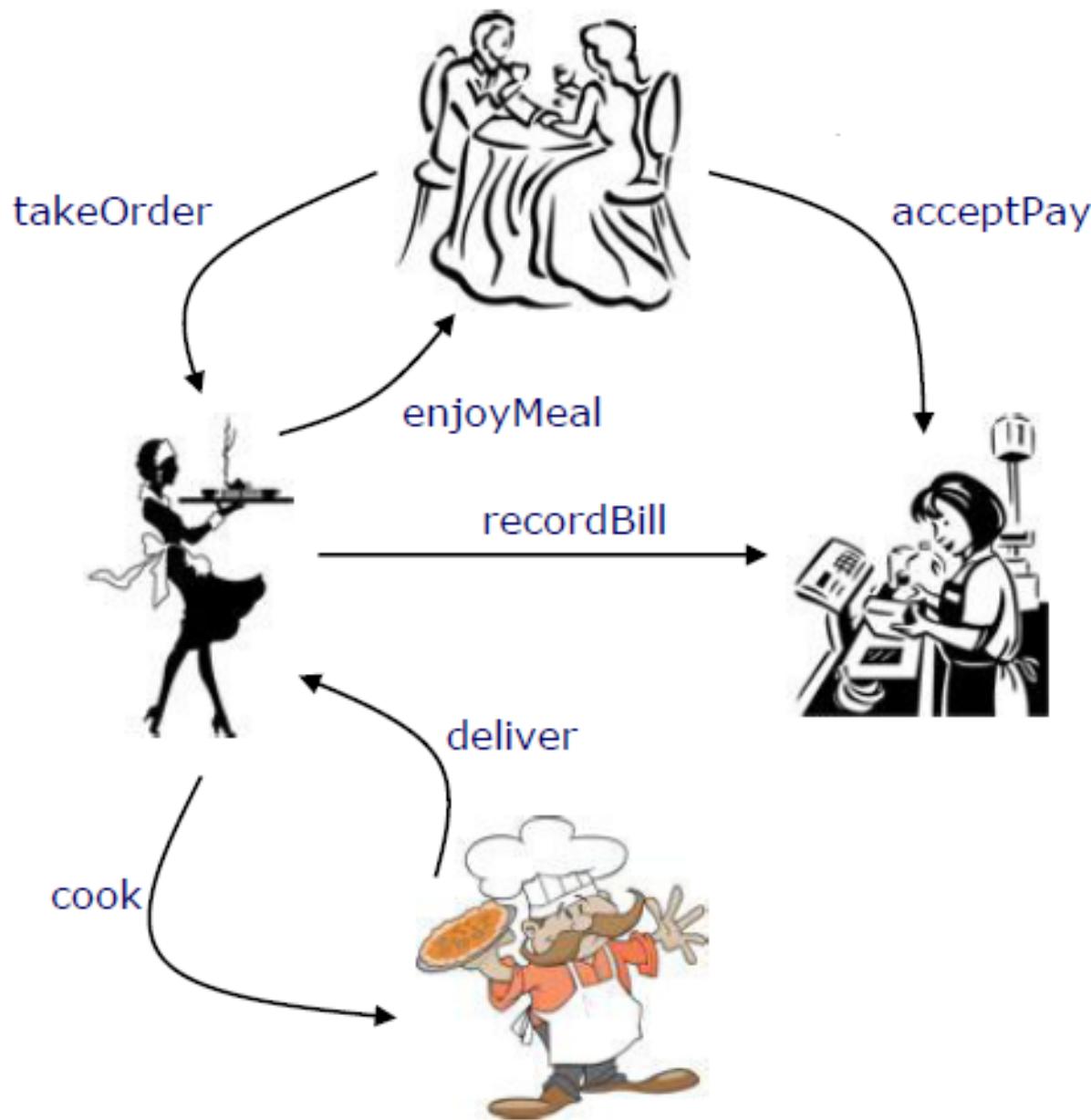
- RE-USABLE
- FLEXIBLE
- UNDERSTANDABLE
- TESTABLE

Why OOAD?



Take Software Design & Development Close to Real Life
- Inspired by everything we encounter in our Life

Objects Everywhere



Real Life is full of
objects

Objects
co-operate to
complete a task

4 Pillars of OOP

Abstraction



Inheritance

Encapsulation

Polymorphism



Battery! What is your perspective?



How about perspective of user, clock & battery manufacturer?



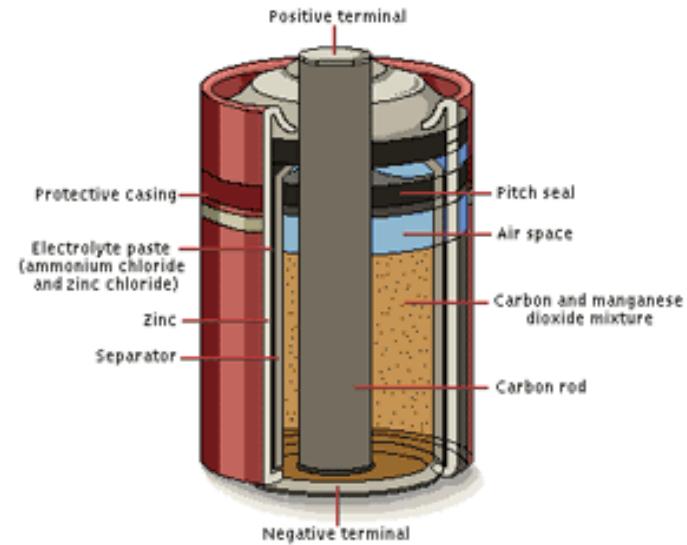
Separation of Concerns



Alarm Manufacturer



Battery Manufacturer



In real life, user need not worry about how an object is made or what it contains for using it

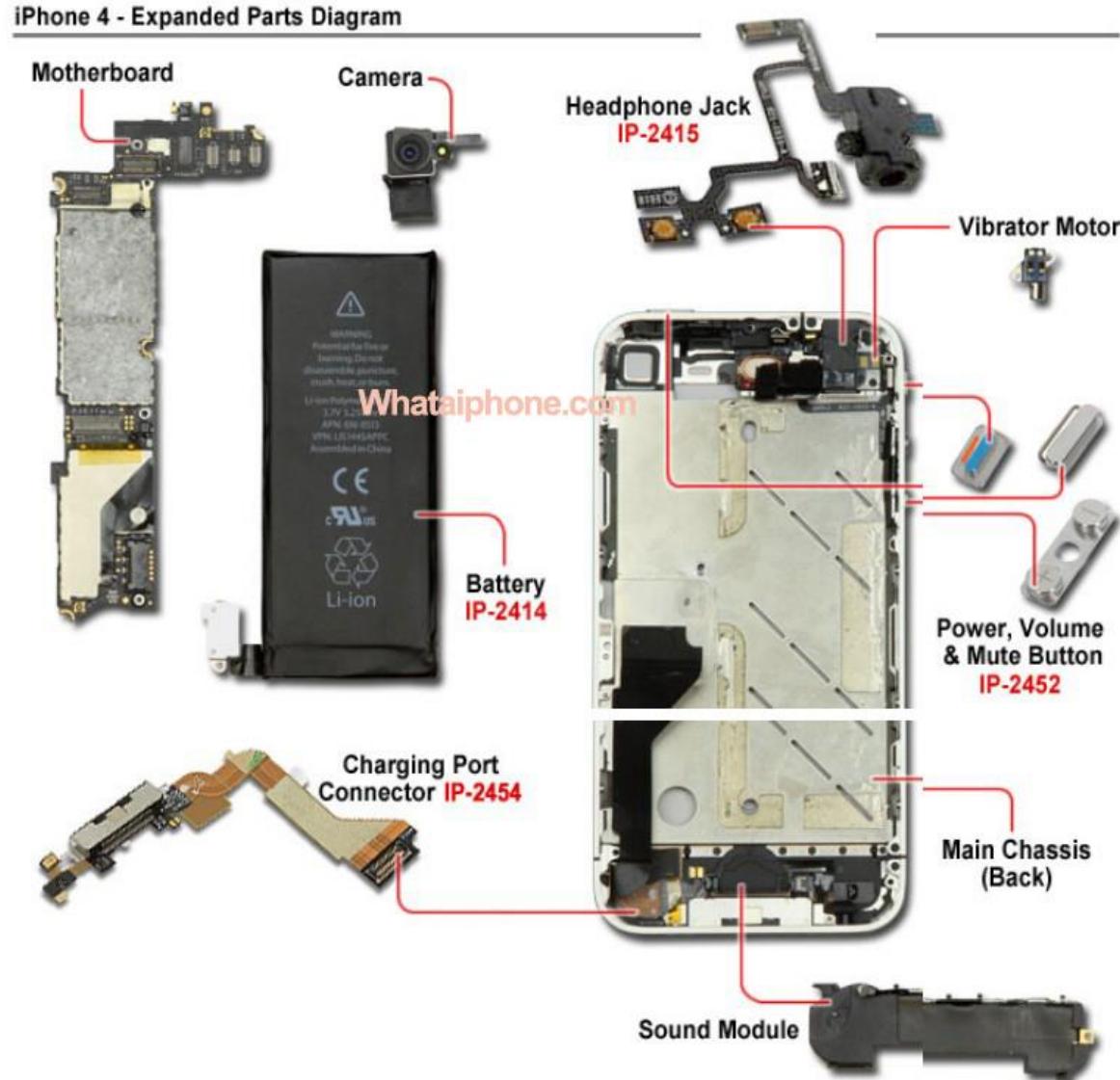
Develop Component in Parts

In real life, Developer of one component need “not worry” about the complete system

(or)

How the component will be used in the complete system

→ “Easy to Understand”



Component are Replaceable



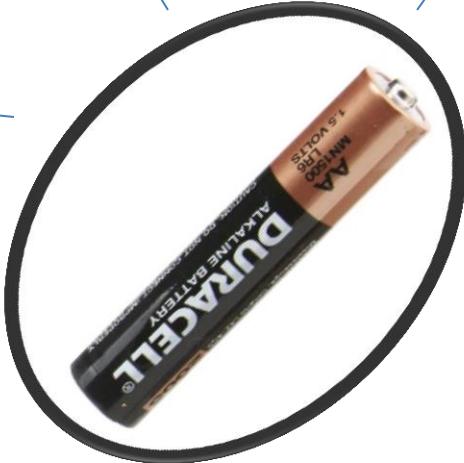
In real life, components of similar functionality are replaceable

Use & Throw



In real life, once we are done with the component, we discard it

Component Re-Use (Pluggable)



Devices...



In real life, one type component can be used in multiple systems

Think about structure first



Plumbing Plan

Floor Plan



Actual Instance

In real life, we think about structure, build prototype
before building the actual instance

What is your perspective?



Do we need to know the internals or just interfaces?



Standard Interfaces are better!



Real Life & OOP

Class

Attributes

Services

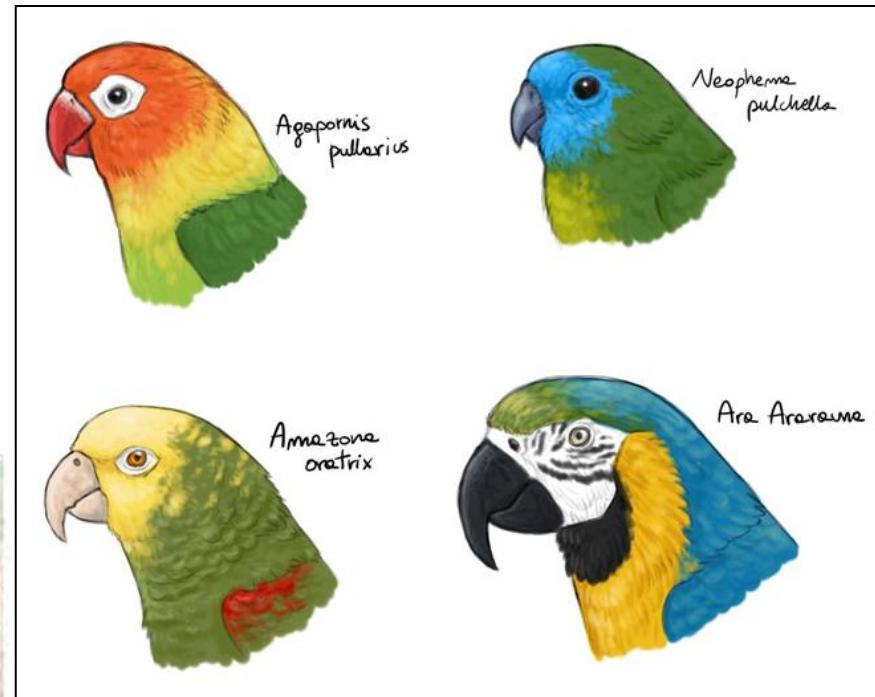
Parrot

-eyeColor: Color
-beakColor: Color
-eyeType: EyeType
-bodyPattern: ColorPattern

+Eat()
+Fly()
+Sleep()

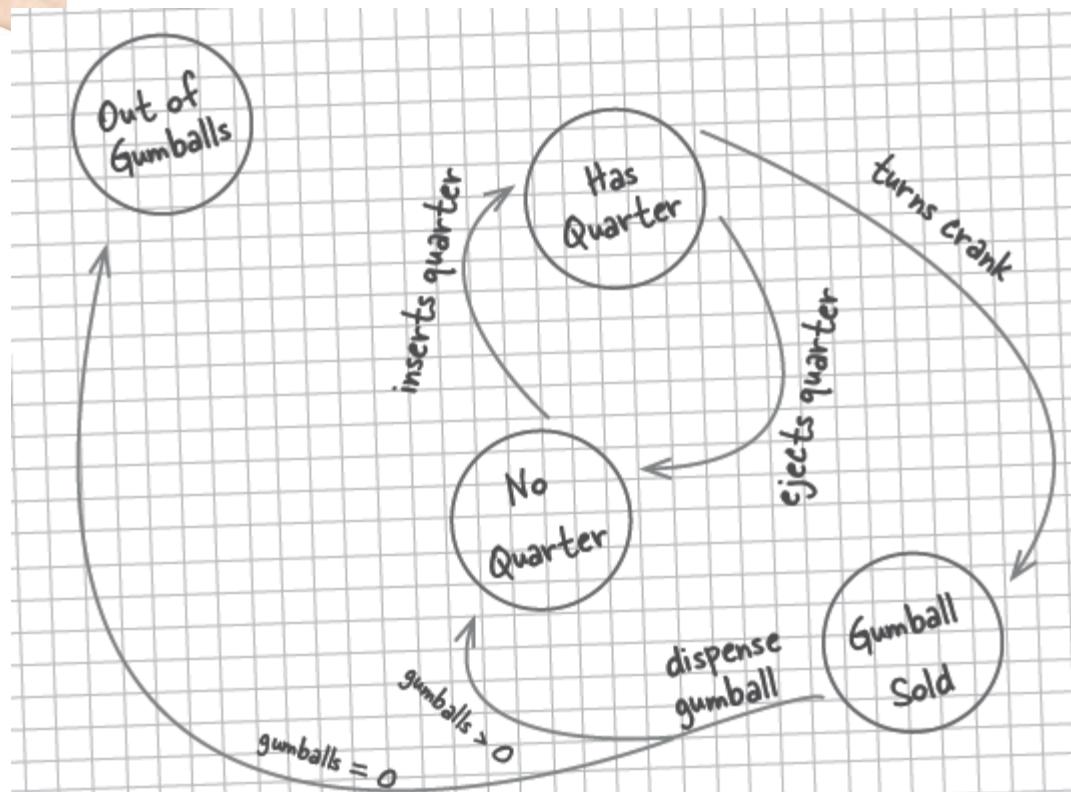
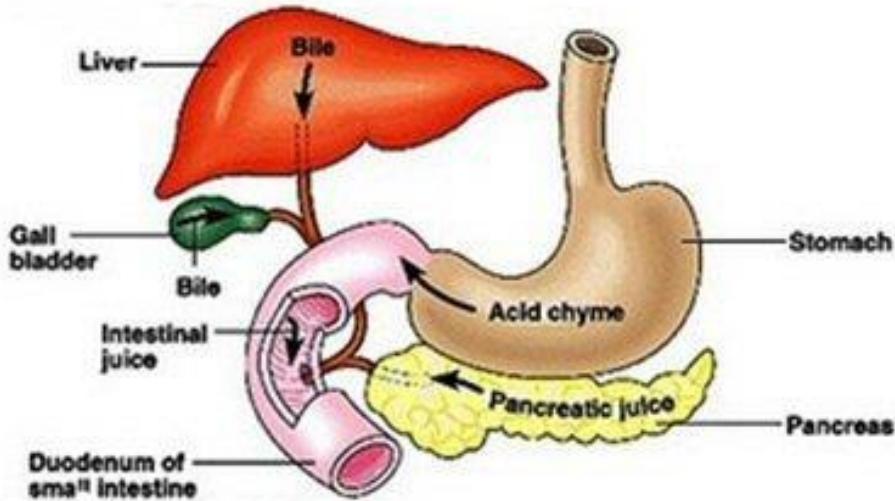


Multiple instance of same type



Objects / Instances

Objects/Systems have States



Life & OOP

Inheritance ,
Constructor

Birth



Polymorphism



Destructor

Death



Inheritance ,
Constructor

Birth

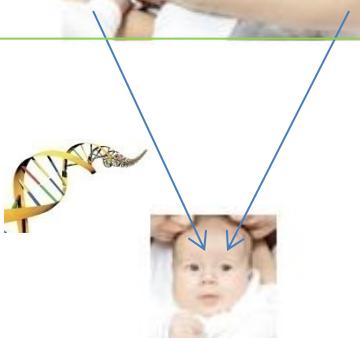


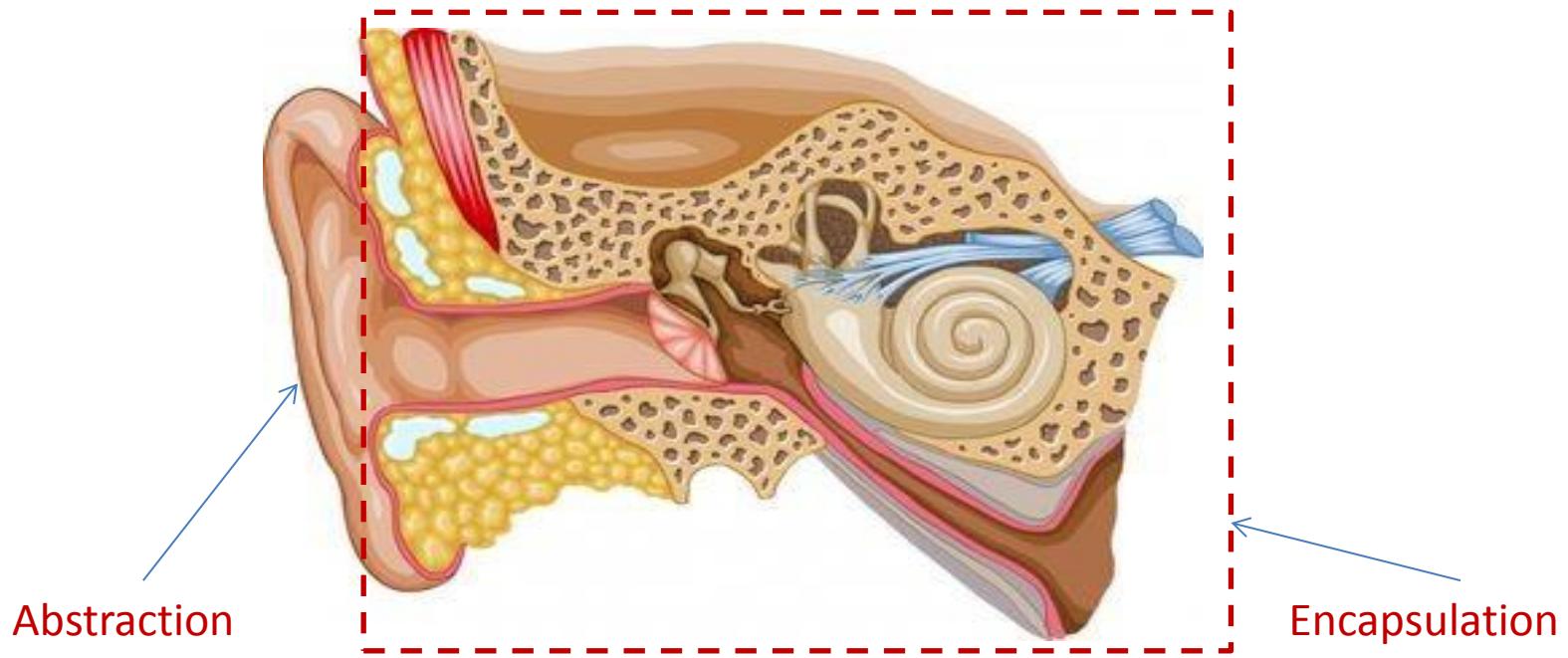
True - Polymorphism



Destructor

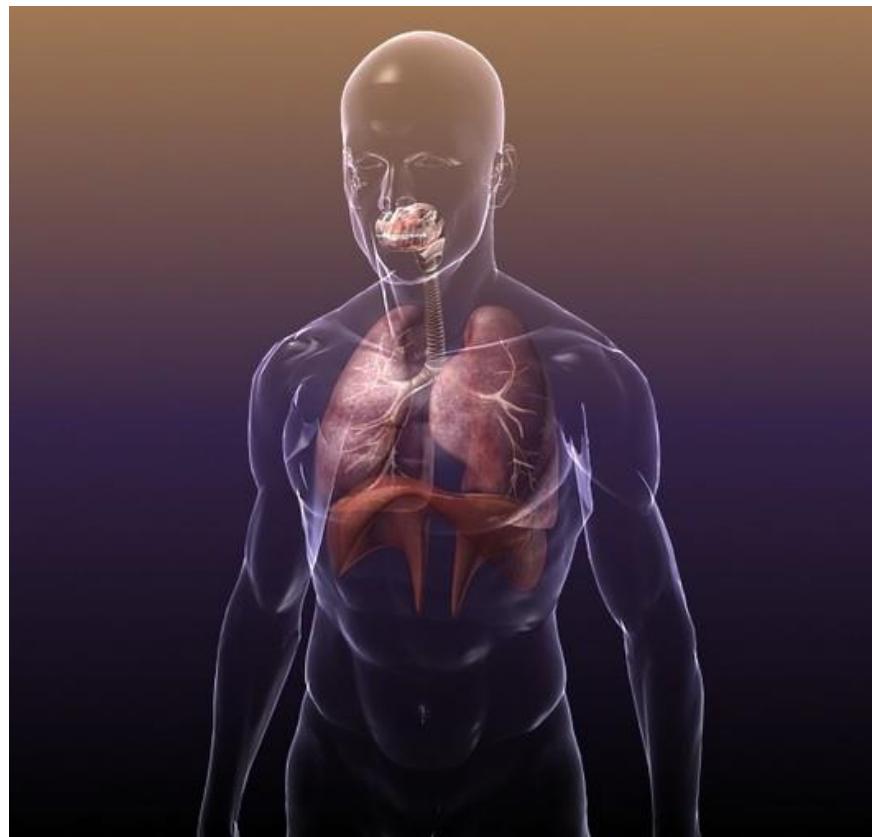
Death





- Abstraction means to hide the unnecessary data from the user.
- The user only needs the required functionality (or) the output according to his requirements
- Encapsulation is simply combining the data members and functions into a single entity called an object

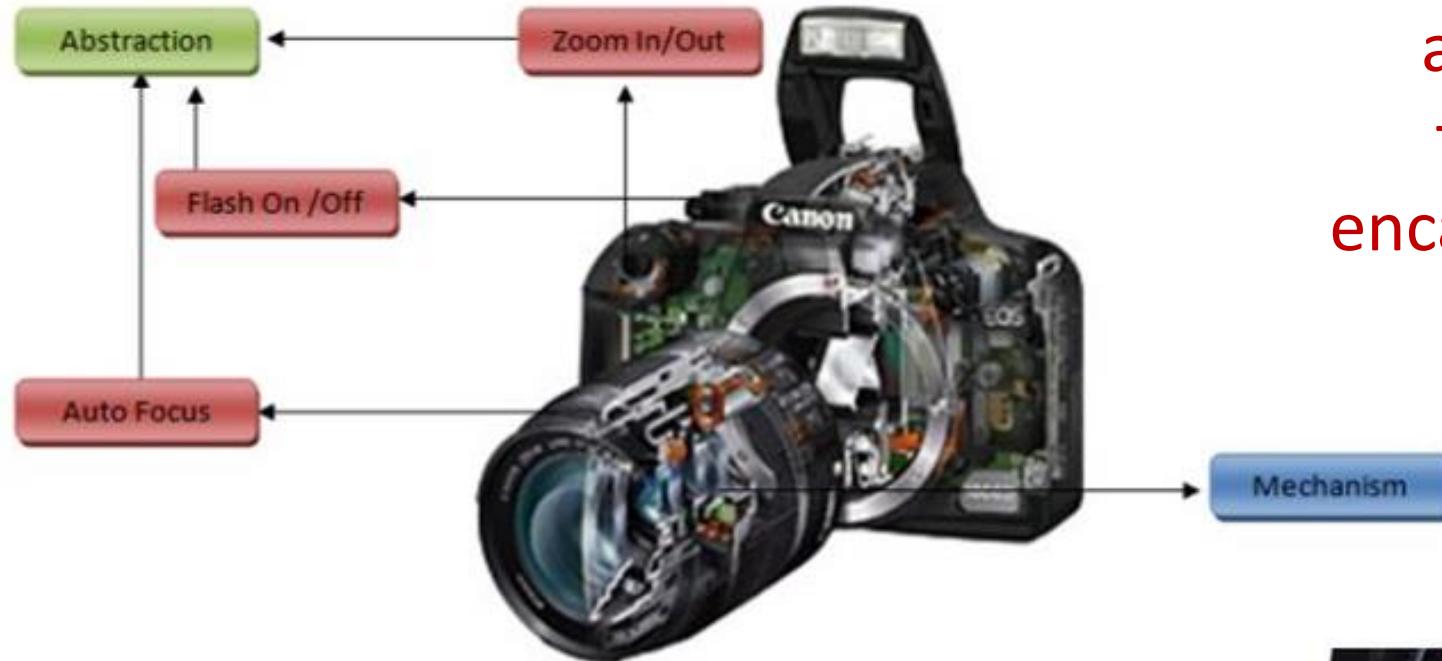
Life & OOP



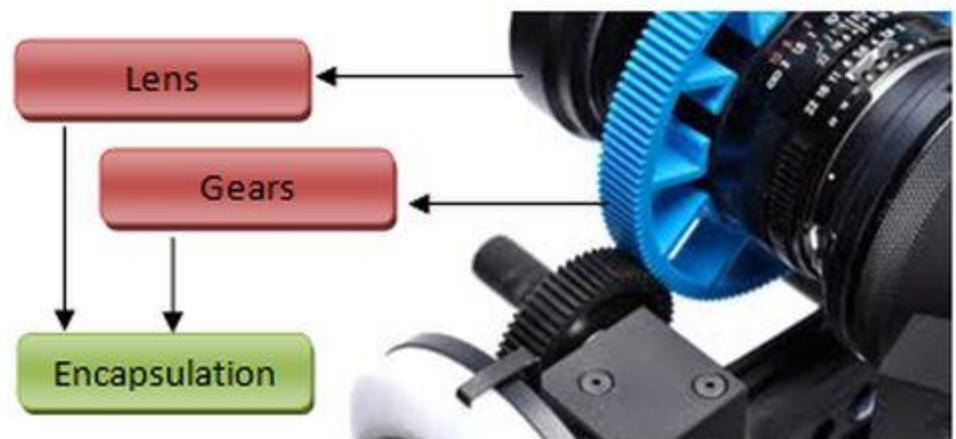
Abstraction & Encapsulation

Abstraction

"Abstraction is achieved through encapsulation"

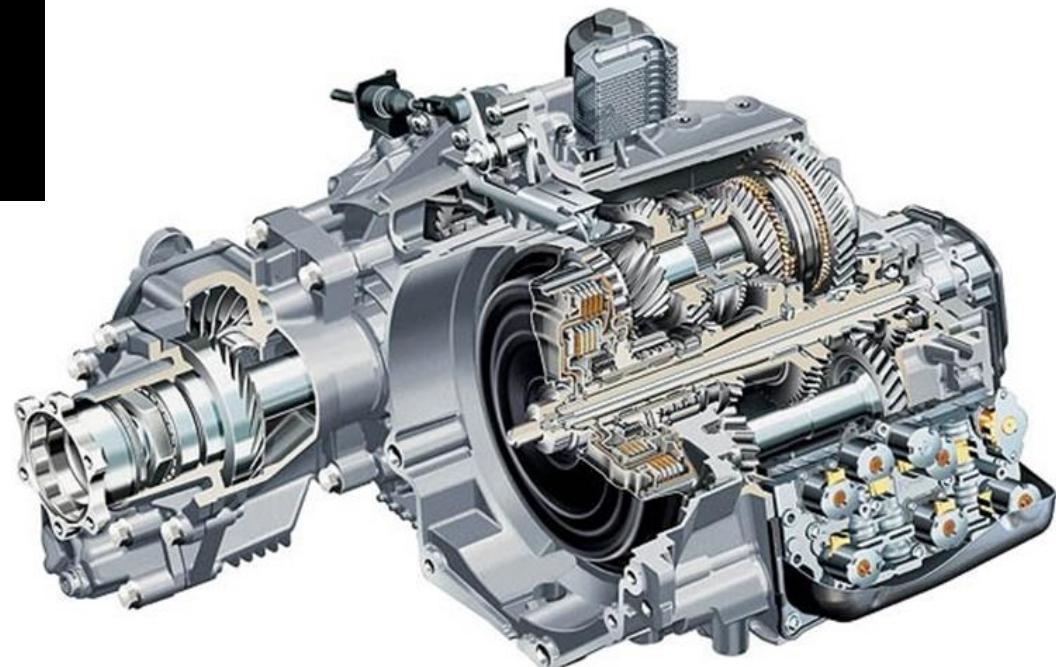


Abstraction solves the problem in the design side & encapsulation is the implementation.



Encapsulation

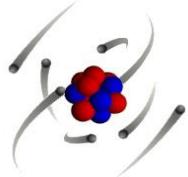
Abstraction



Encapsulation

Abstraction In Electronics

Nature



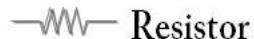
I (A)	V (V)
0	0
1	0.8
2	1.5
3	2.1
4	2.6
5	3
6	3.3
7	3.6
8	3.8
9	3.9
10	4

Laws

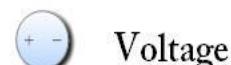
$$V = IR$$

$$\nabla \cdot D = \rho$$
$$\nabla \cdot B = 0$$
$$\nabla \times E = -\frac{\partial B}{\partial t}$$
$$\nabla \times H = J + \frac{\partial D}{\partial t}$$

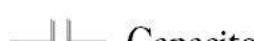
Lumped circuit abstractions



Resistor

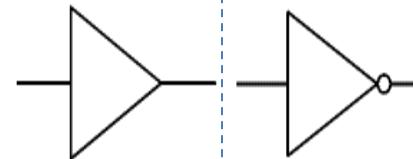


Voltage

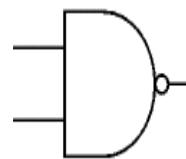


Capacitor

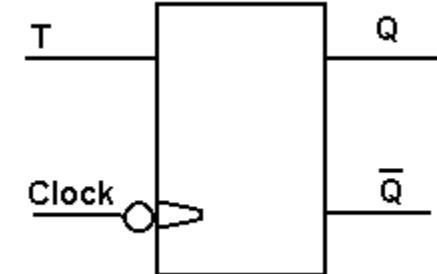
Amps



Digital



Time Functions



Instruction Set

ARM,
x86

Assembly Language
Code

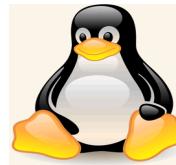
LOAD Y
ADD X
STORE Y

Language

C/C++

JAVA™

Software Systems



Useful things



Abstraction simplifies things, makes life easier

True Polymorphism

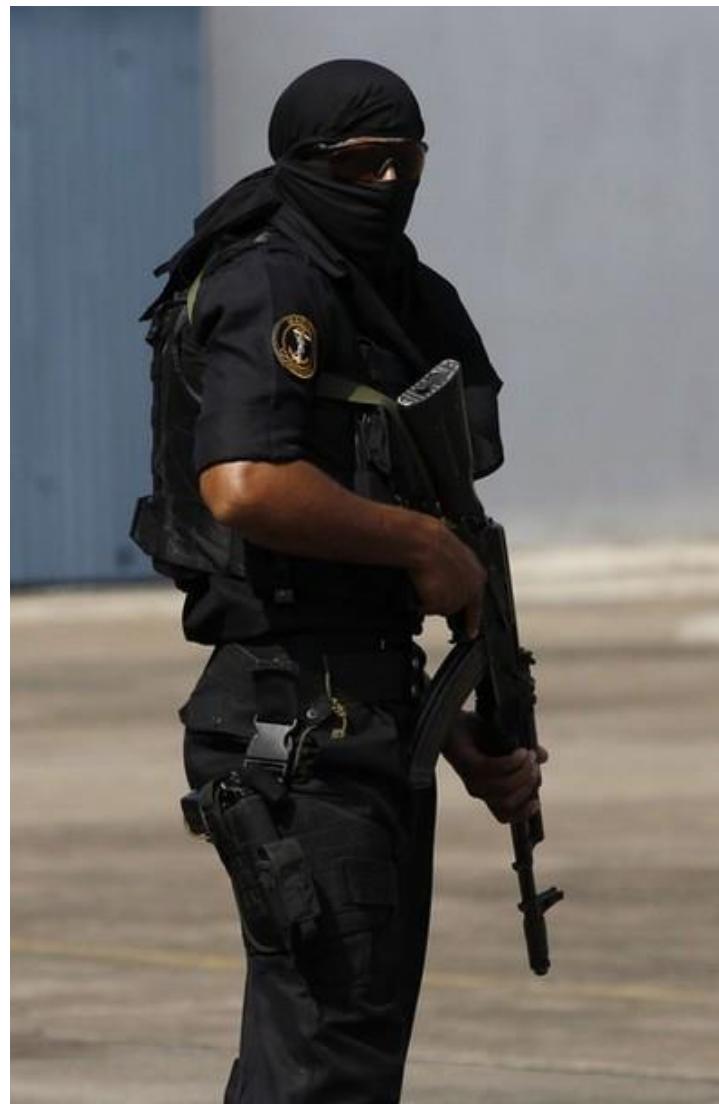
Which ONE do you prefer?



Problem Statement

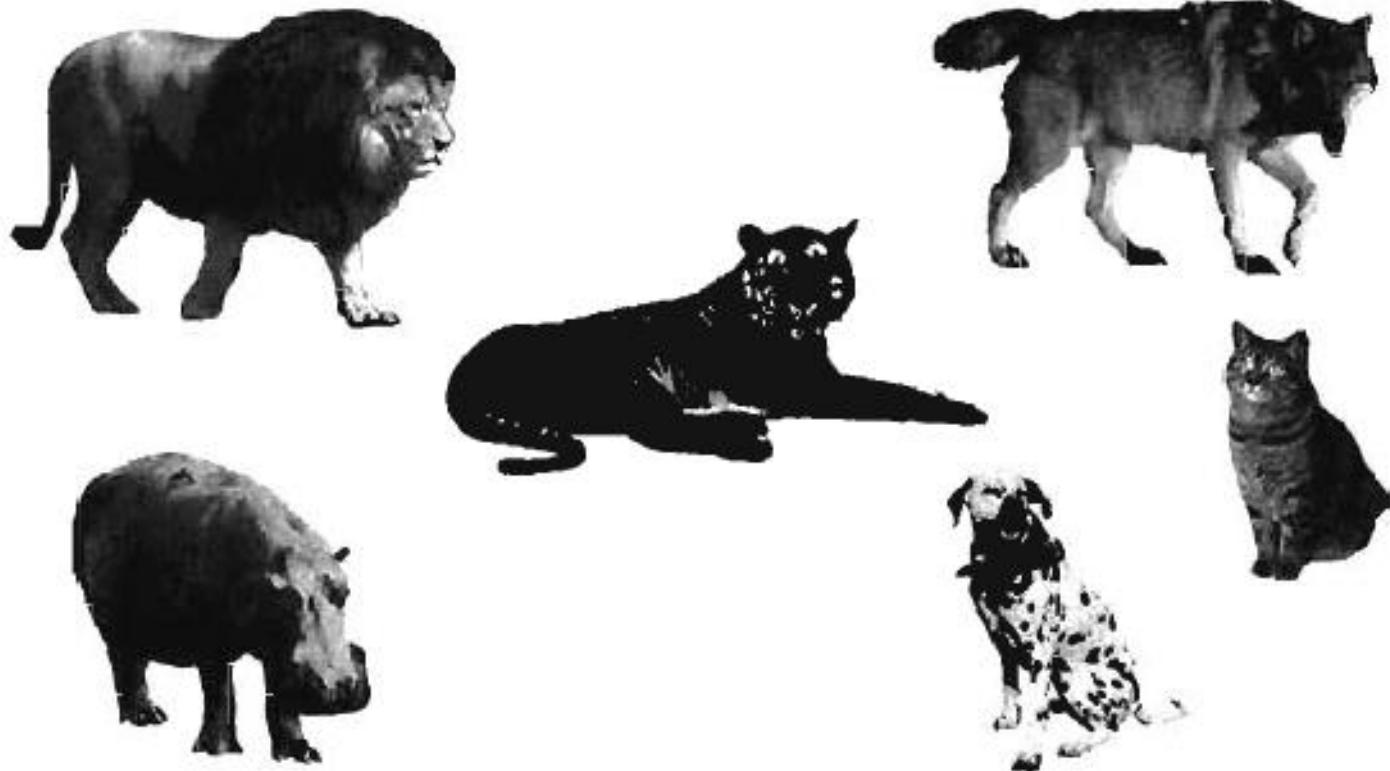


Design a Game with Weapon Switching Capability



Design Inheritance Tree

Let's design the inheritance tree for an Animal simulation program

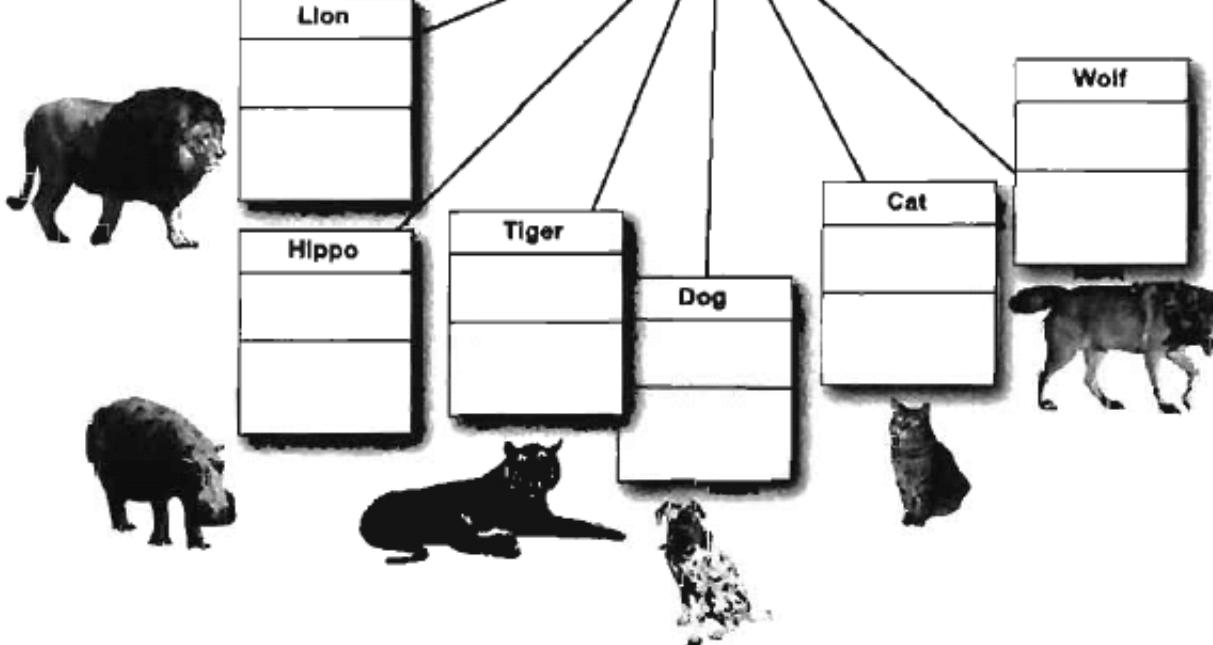


And we want other programmers to be able to add new kinds of animals to the program at any time.

Inheritance Tree Type I

Look for objects that have common attributes and behaviors.

Animal
picture
food
hunger
boundaries
location
makeNoise()
eat()
sleep()
roam()



Design a class that represents the common state and behavior.

These objects are all animals, so we'll make a common superclass called **Animal**.

We'll put in methods and instance variables that all animals might need.

Which Methods to override?

Animal

picture
food
hunger
boundaries
location

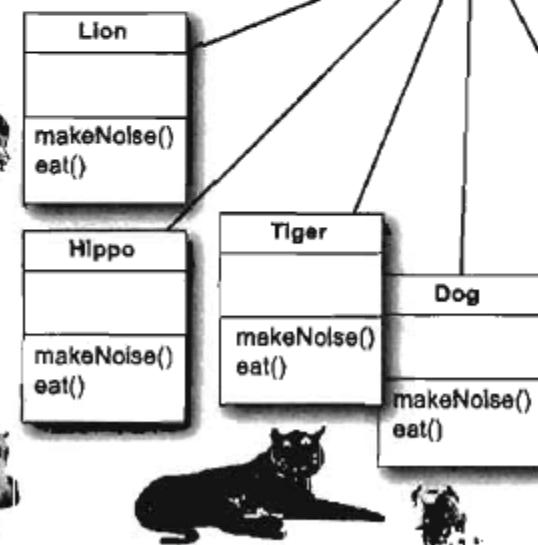
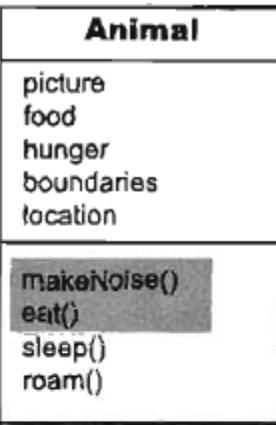
makeNoise()
eat()
sleep()
roam()

Hmmm... I wonder if Lion,
Tiger, and Cat would have
something in common.

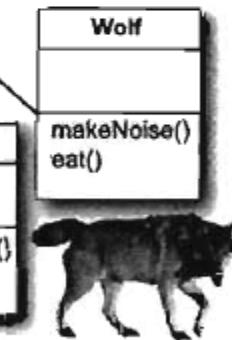


Decide if a subclass
needs behaviors (method
implementations) that are specific
to that particular subclass type.

Looking at the Animal class,
we decide that eat() and
makeNoise() should be overridden
by the individual subclasses.



Wolf and Dog are both canines...
maybe there's something that
BOTH classes could use...



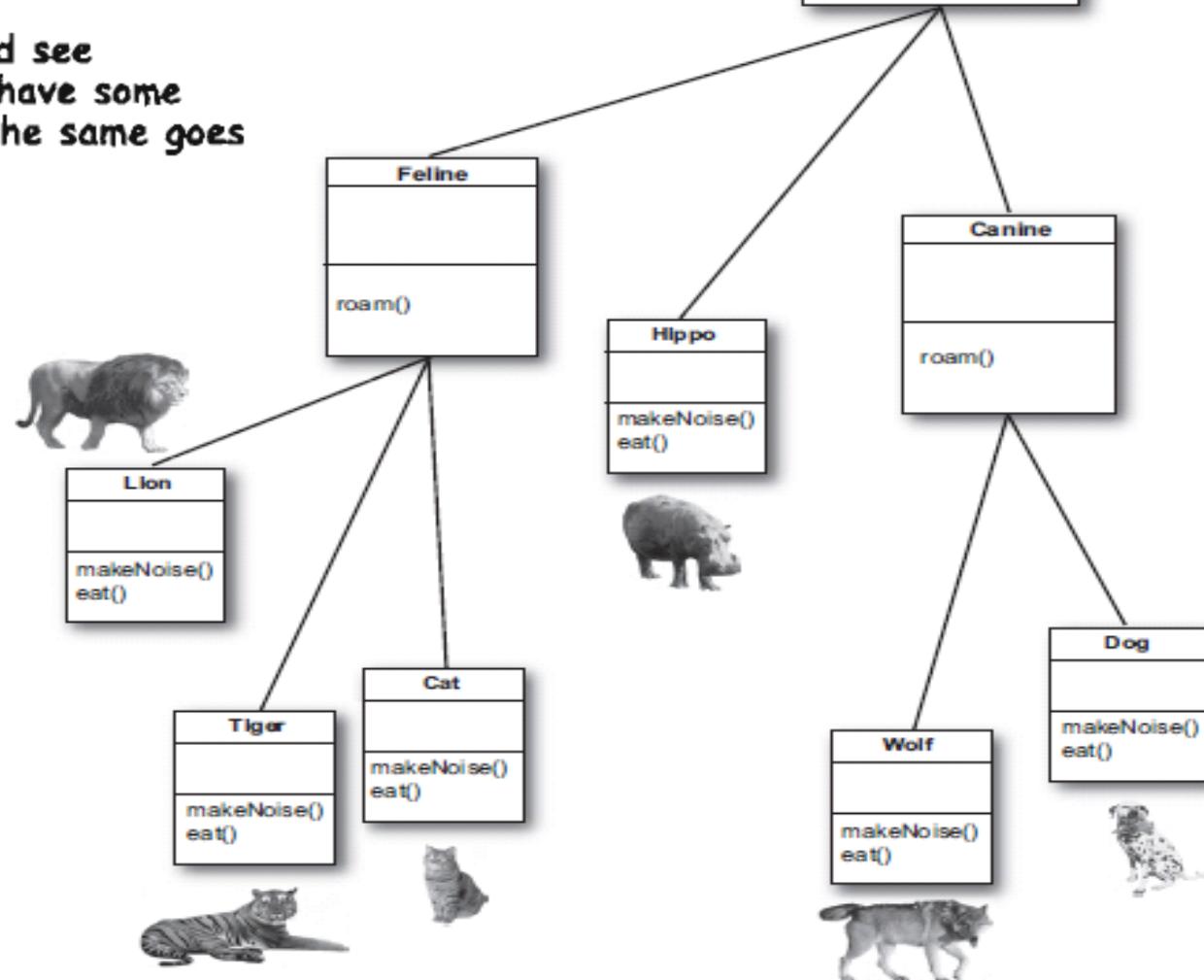
Do all animals eat the same way?

Looks for more opportunities

Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behavior.

We look at our classes and see that Wolf and Dog might have some behavior in common, and the same goes for Lion, Tiger, and Cat.

Animal
picture
food
hunger
boundaries
location
makeNoise()
eat()
sleep()
roam()



IS-A Relationship

Which method is called?

make a new Wolf object

calls the version in Wolf

calls the version in Canine

calls the version in Wolf

calls the version in Animal

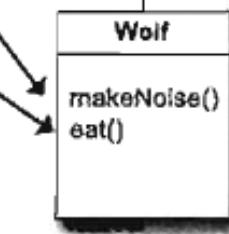
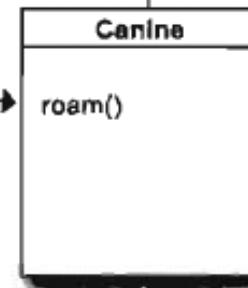
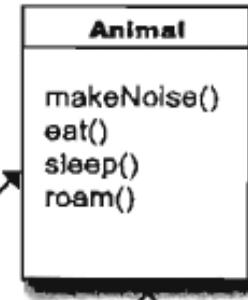
Wolf w;

w.makeNoise();

w.roam();

w.eat();

w.sleep();



Canine extends Animal

Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal

If class B extends class A, class B IS-A class A.

This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.



Strange Object!

What does a new **Animal()** object look like?



scary objects

Animal Object??



Some classes just should not be instantiated!

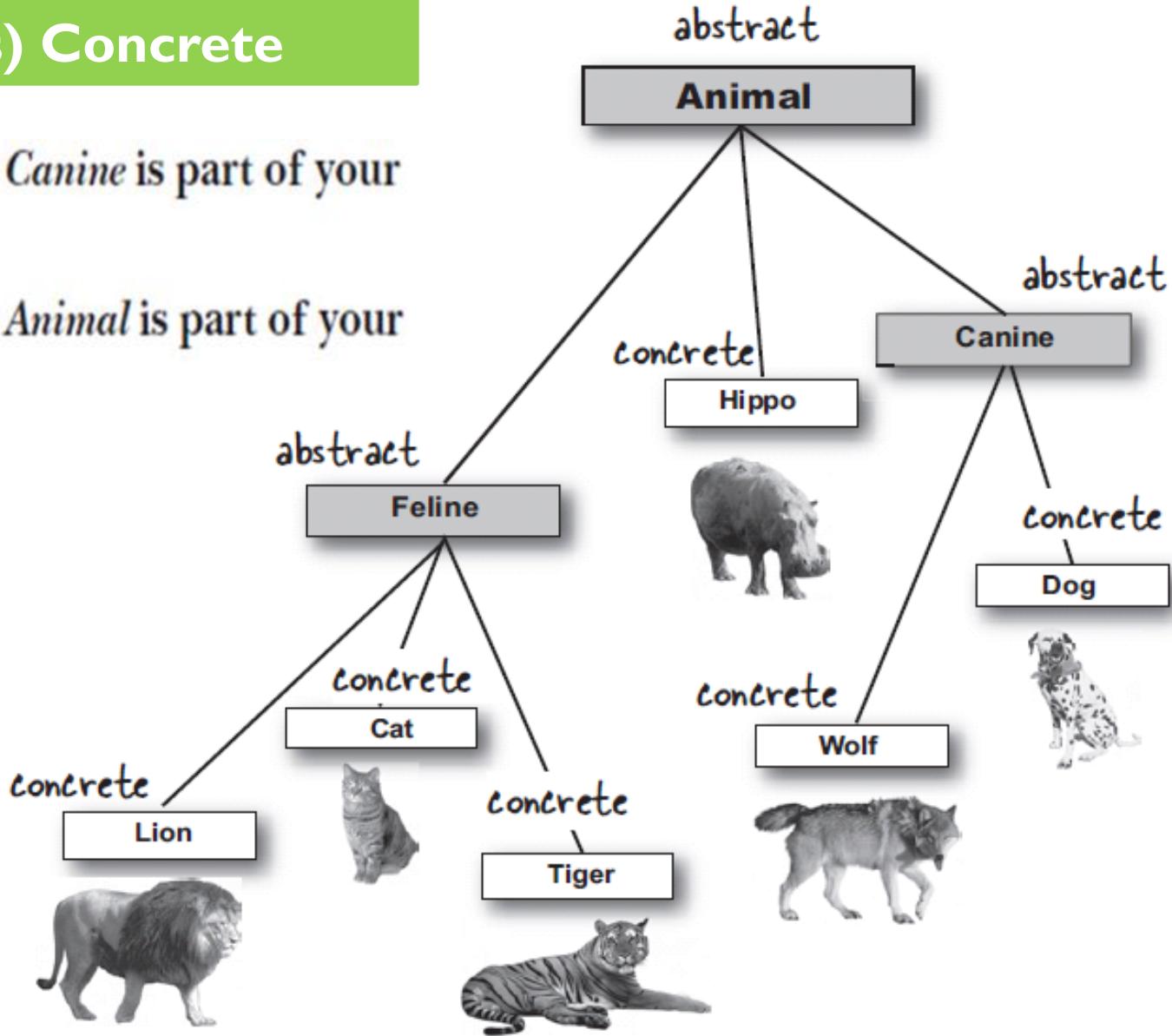
An **abstract class** has virtually* no use, no value, no purpose in life, unless it is **extended**.

With an abstract class, the guys doing the work at runtime are **instances of a subclass** of your abstract class.

Abstract (vs) Concrete

Everything in class *Canine* is part of your contract.

Everything in class *Animal* is part of your contract.



The compiler won't let you instantiate
an abstract class

Contract

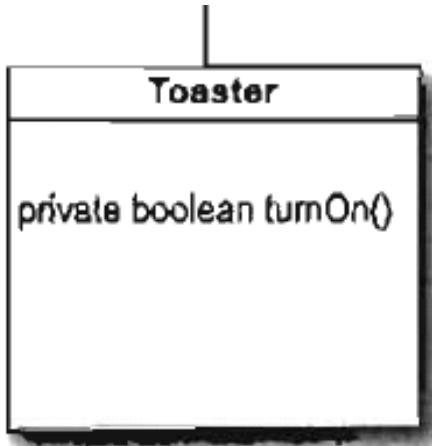
Keeping the contract: rules for overriding

The methods *are* the contract.

- Arguments must be the same, and return types must be compatible.
- The method can't be less accessible.

NOT LEGAL!

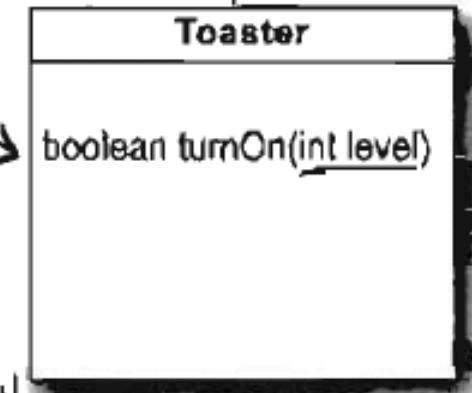
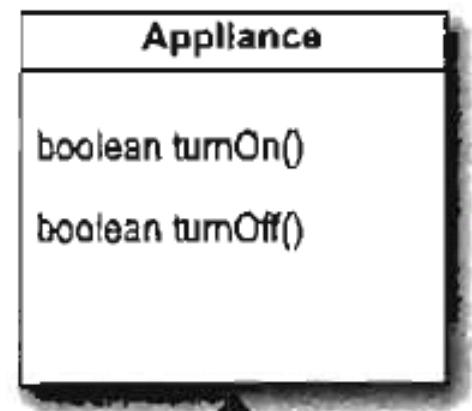
It's not a legal override because you restricted the access level. Nor is it a legal overLOAD, because you didn't change arguments.



This is NOT an override!

Can't change the arguments in an overriding method!

This is actually a legal overLOAD, but not an overRIDE.



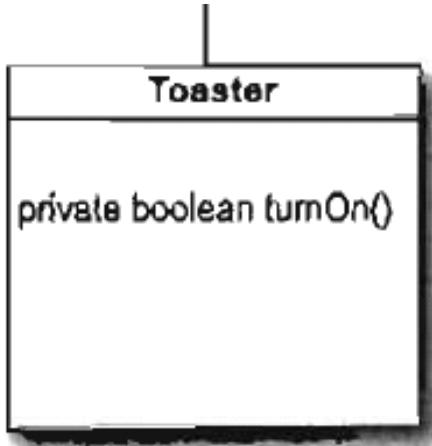
Contract

Keeping the contract: rules for overriding

The methods *are* the contract.

- Arguments must be the same, and return types must be compatible.
- The method can't be less accessible.

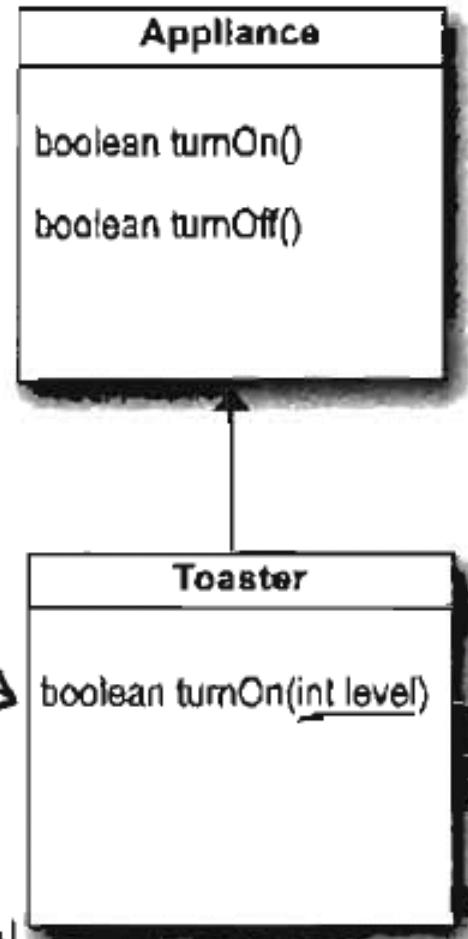
NOT LEGAL!
It's not a legal override because you restricted the access level. Nor is it a legal overLOAD, because you didn't change arguments.



This is NOT an override!

Can't change the arguments in an overriding method!

This is actually a legal overLOAD, but not an overRIDE.



Polymorphism

**'Polymorphism' means
'many forms'.**

```
Animal[] animals = new Animal[5];  
  
animals [0] = new Dog();  
  
animals [1] = new Cat();  
  
animals [2] = new Wolf();  
  
animals [3] = new Hippo();  
  
animals [4] = new Lion();  
  
for (int i = 0; i < animals.length; i++)  
  
    animals[i].eat();  
  
    animals[i].roam();  
}
```

```
class Vet {  
  
    public void giveShot(Animal a) {  
  
        // do horrible things to the Animal at  
  
        // the other end of the 'a' parameter  
  
        a.makeNoise();  
    }  
}  
  
class PetOwner {  
  
    public void start() {  
  
        Vet v = new Vet();  
  
        Dog d = new Dog();  
  
        Hippo h = new Hippo();  
  
        v.giveShot(d);  
  
        v.giveShot(h);  
    }  
}
```

With polymorphism, you can write code that doesn't have to change when you introduce new subclass types into the program.

Abstract Method



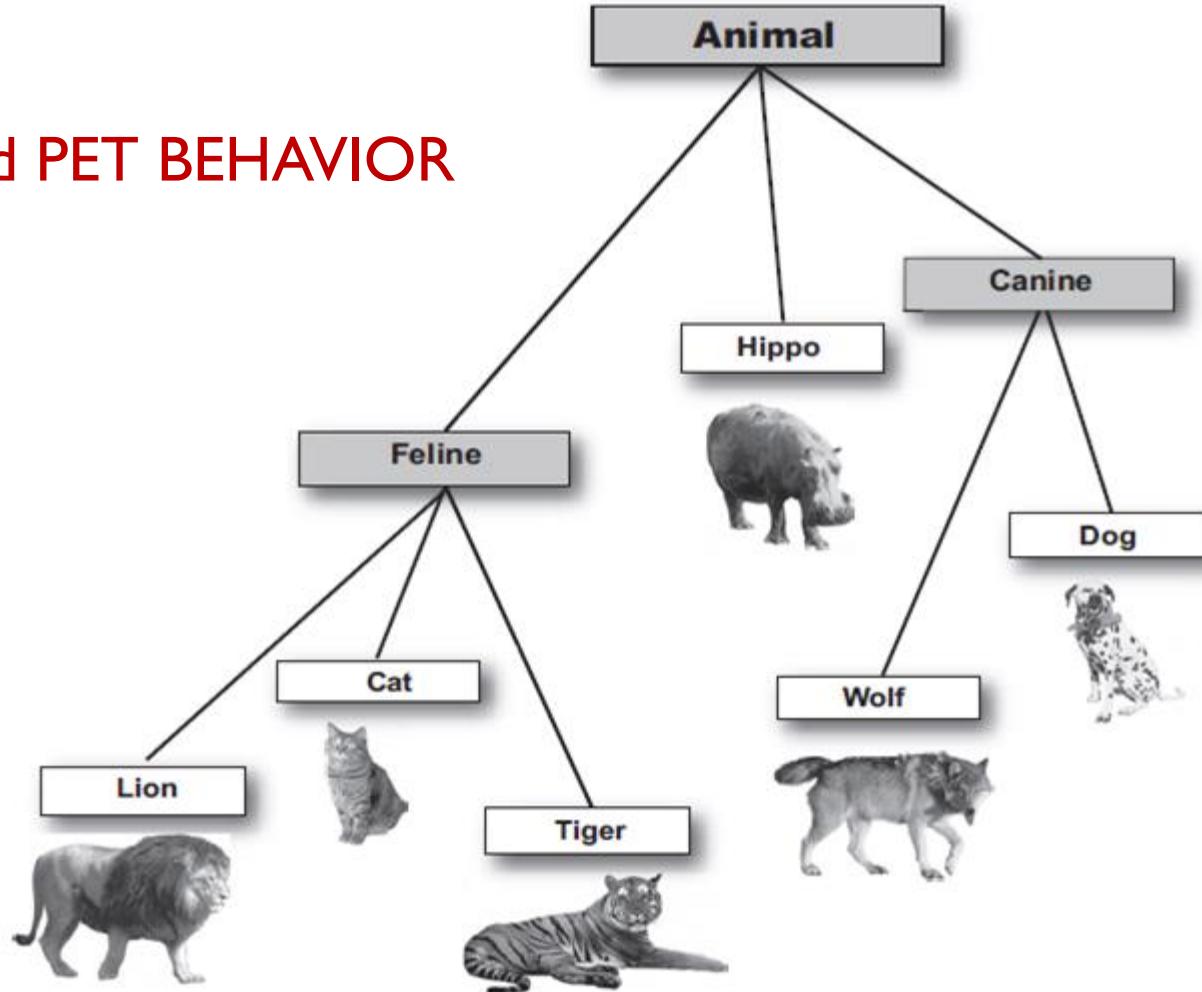
An abstract method has no body!

You **MUST** implement all abstract methods

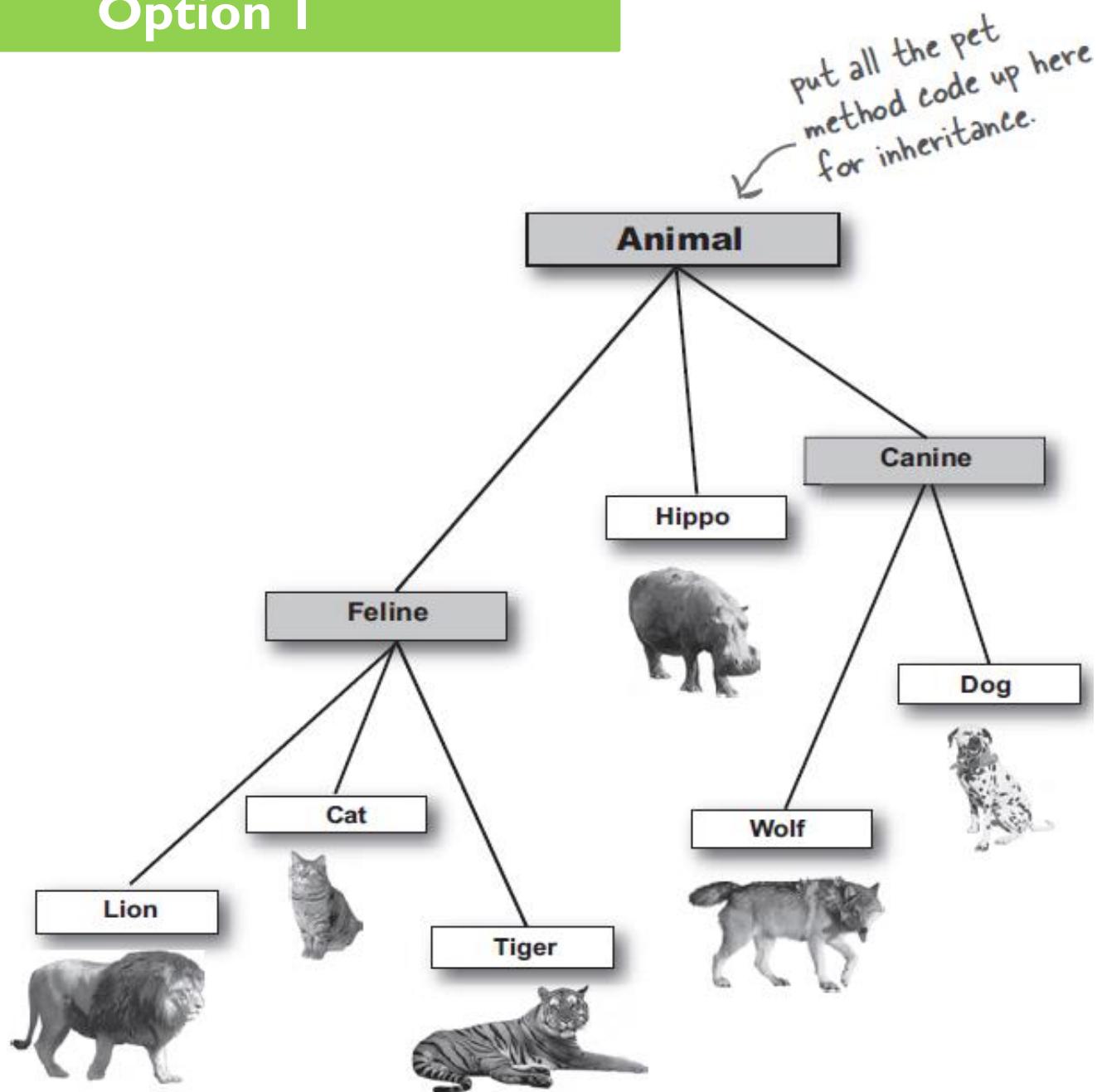
***Implementing an abstract
method is just like
overriding a method.***

Spec Change

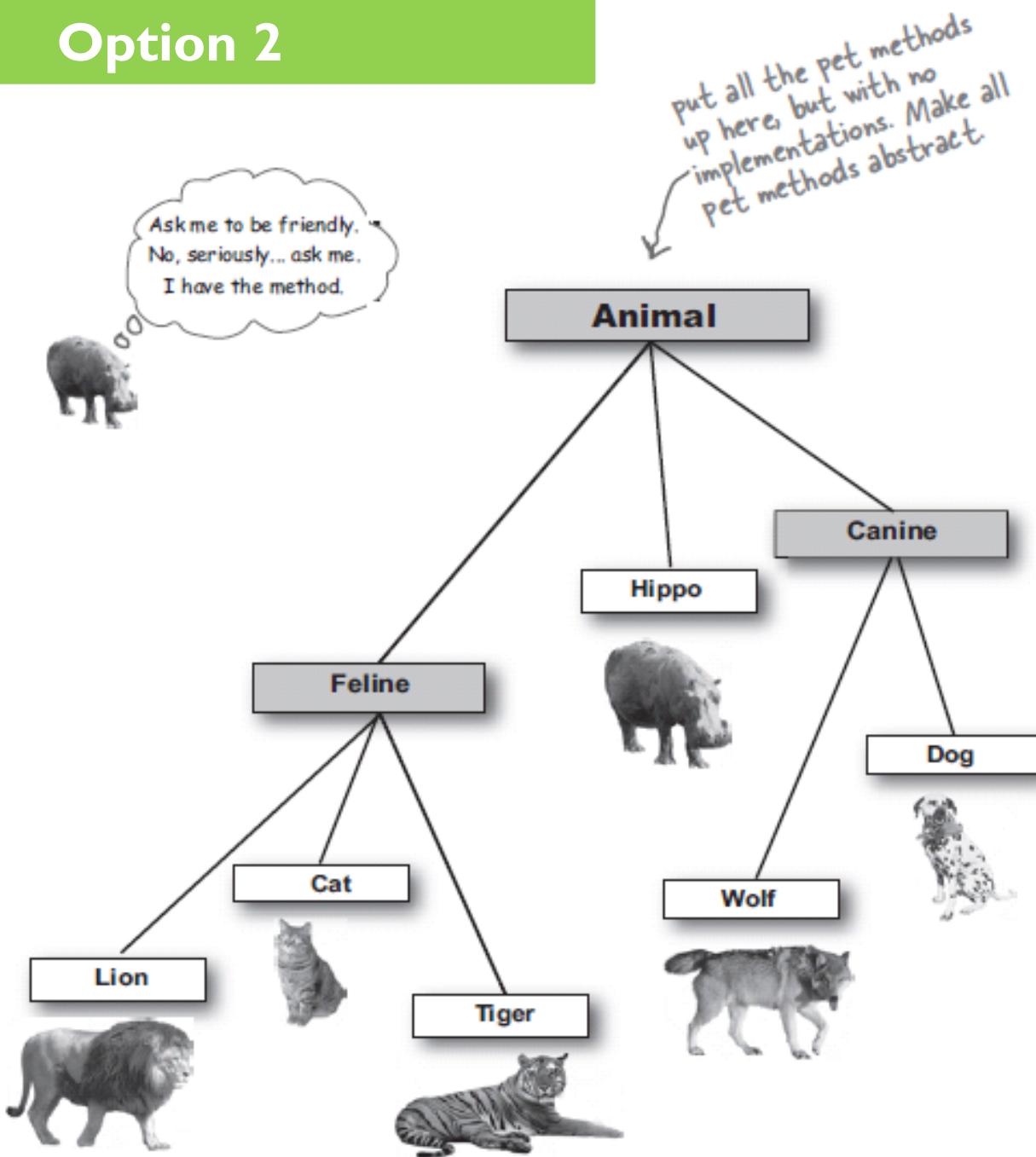
What will you do to add PET BEHAVIOR to Dog?



Option I

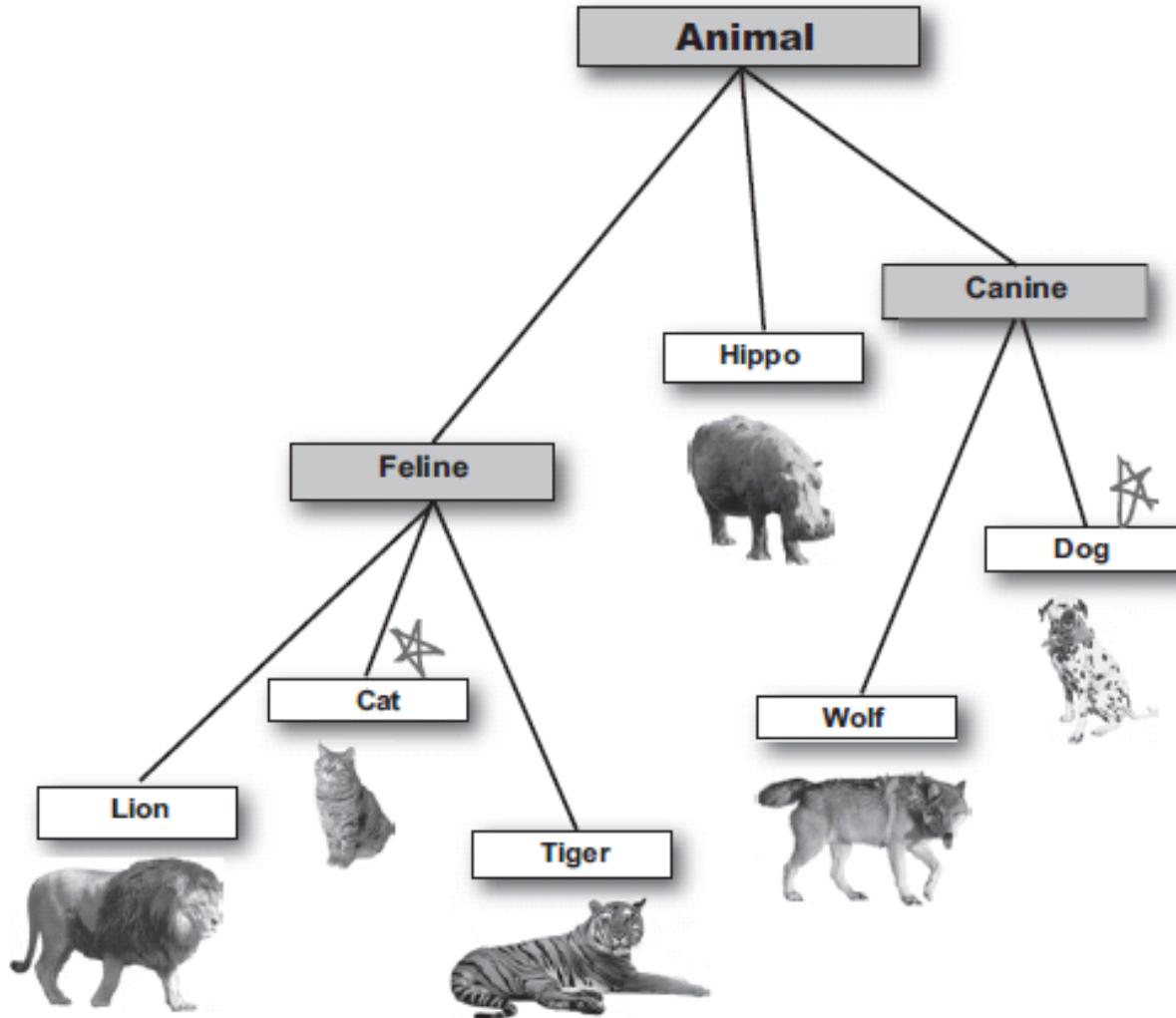


Option 2



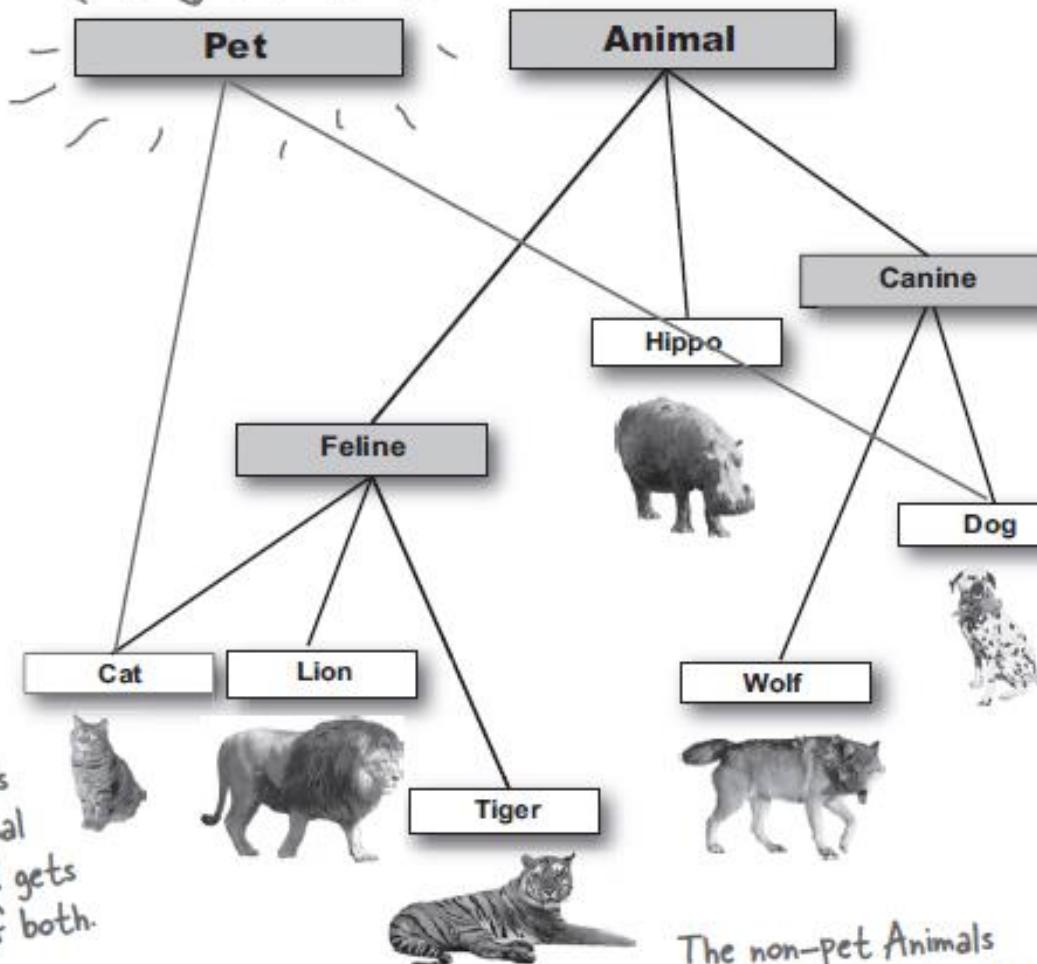
Option 3

* Put the pet methods ONLY in the Animal classes that can be pets, instead of in Animal.



Option 4

We make a new abstract superclass called Pet, and give it all the pet methods.



It looks like we need TWO superclasses at the top

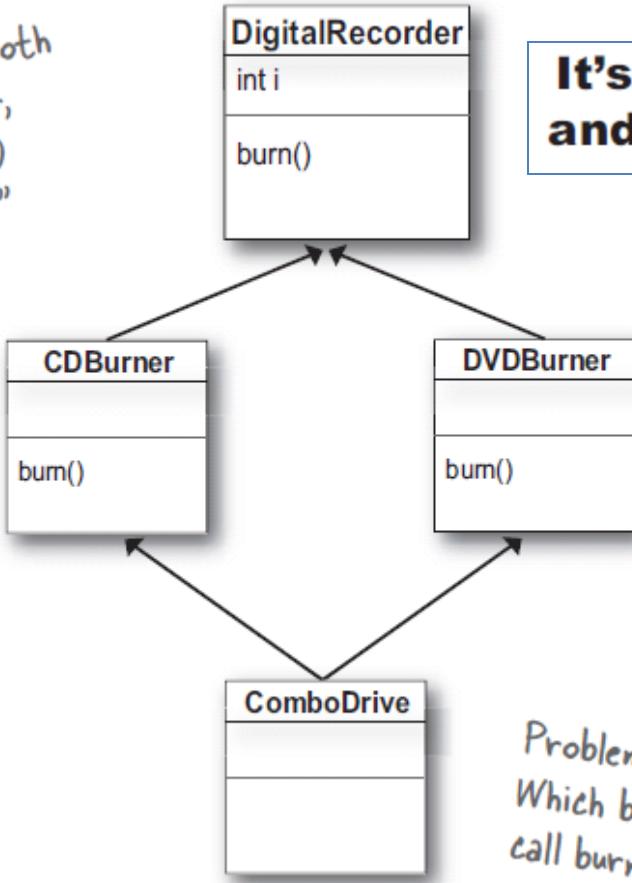
Cat now extends from both Animal AND Pet, so it gets the methods of both.

The non-pet Animals don't have any inherited Pet stuff.

Dog extends both Pet and Animal

Deadly Diamond of Death

CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.



It's called “multiple inheritance” and it can be a Really Bad Thing.

Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

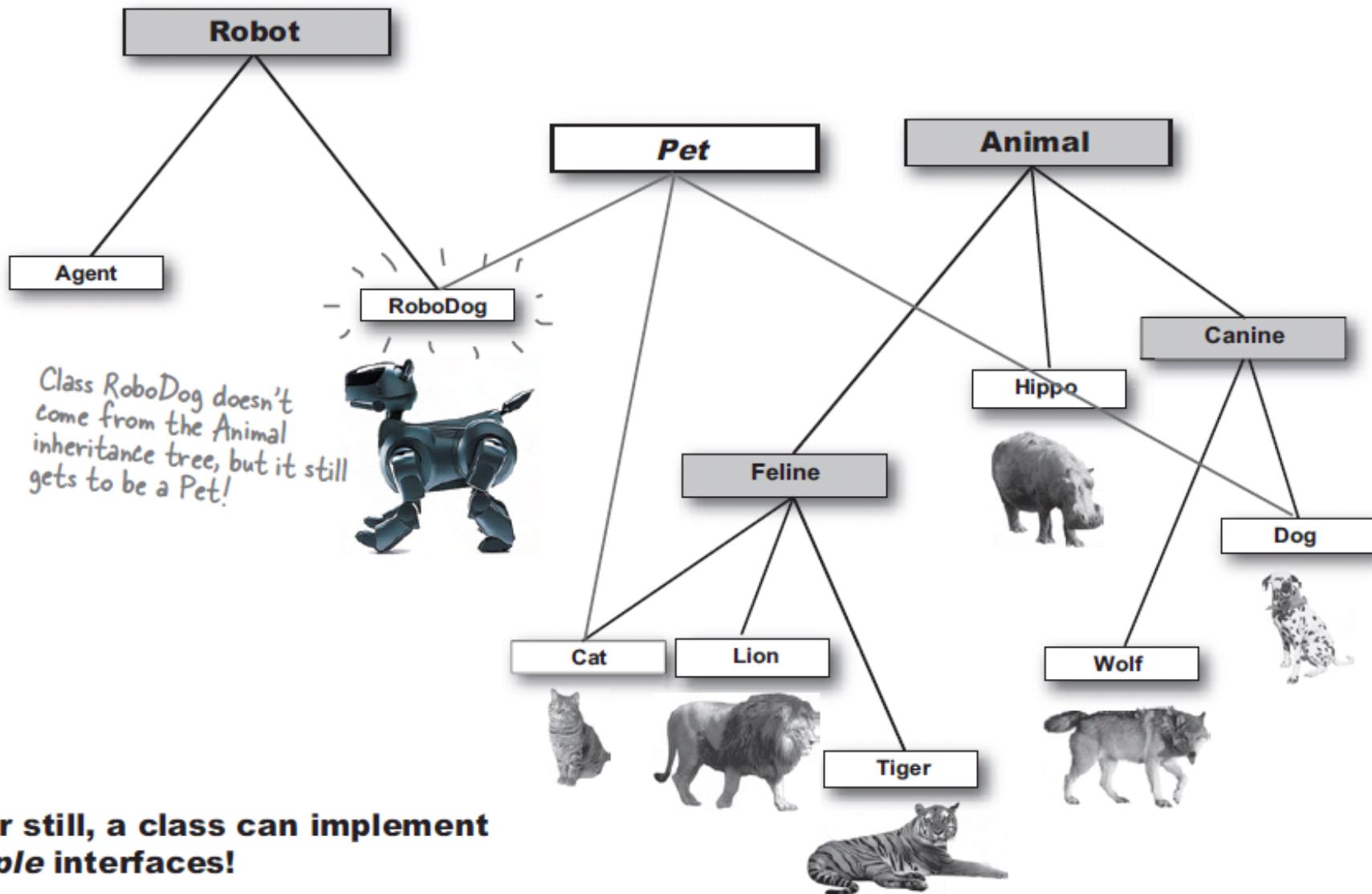
Problem with multiple inheritance.
Which burn() method runs when you call burn() on the ComboDrive?

Interface to the rescue!

interface is like a 100% pure abstract class.

Different Inheritance Tree

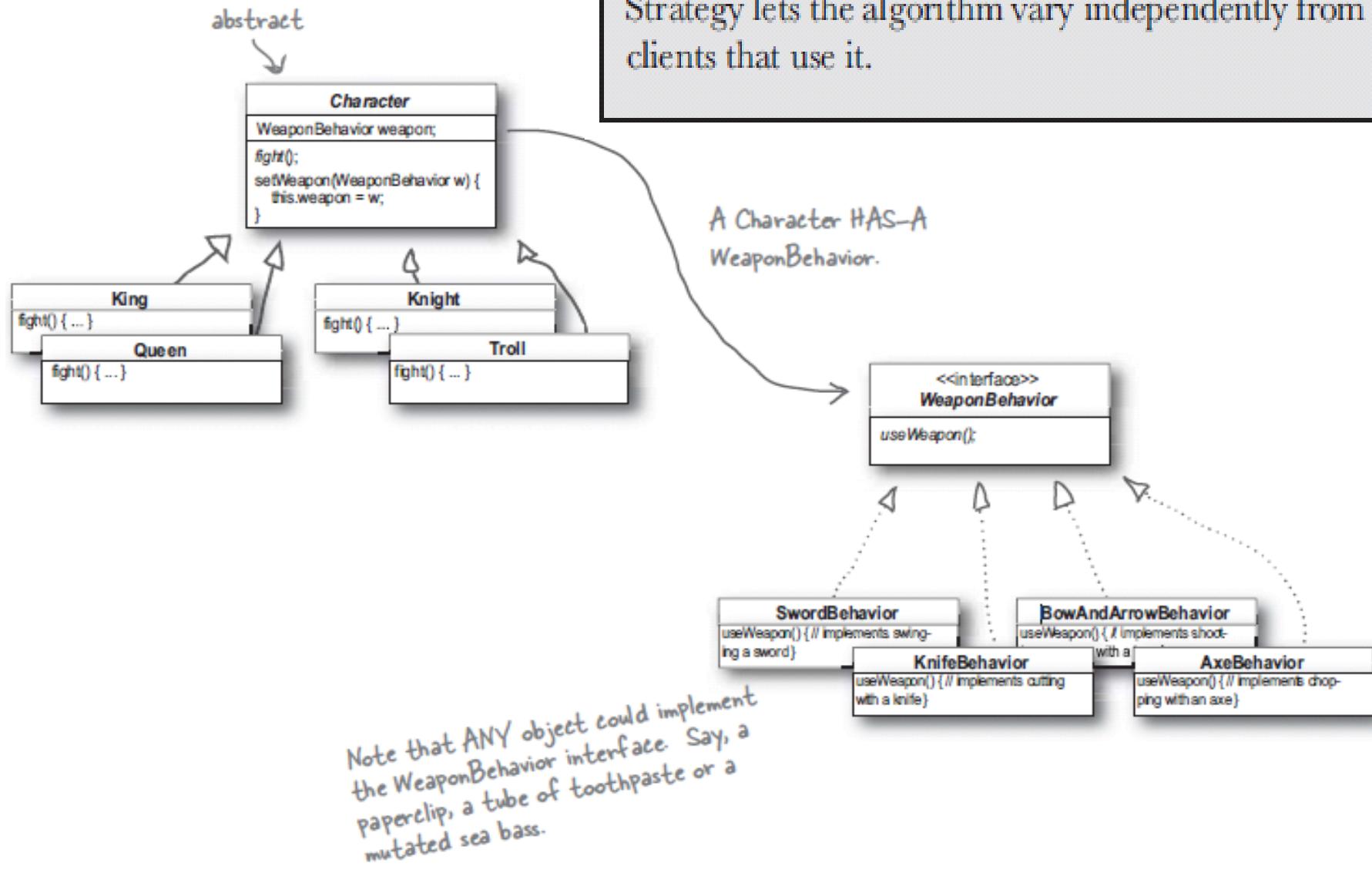
Classes from different inheritance trees can implement the same interface.



Design Patterns

We Just Applied “STRATEGY” Pattern!

The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Principles

HAS-A can be better than IS-A



Design Principle

Favor composition over inheritance.

lets you ***change behavior at runtime***



Design Principle

Program to an interface, not an implementation.

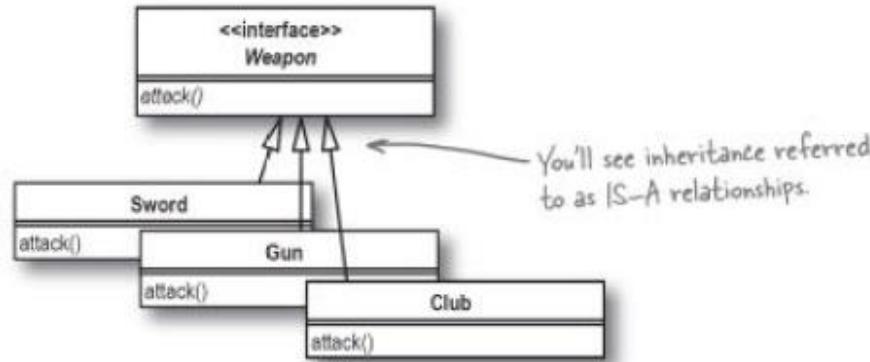
Coding to an interface, rather than to an implementation, makes your software easier to extend.

By coding to an interface, your code will work with all of the interface's subclasses—even ones that haven't been created yet.

IS-A & HAS-A

IS-A refers to inheritance

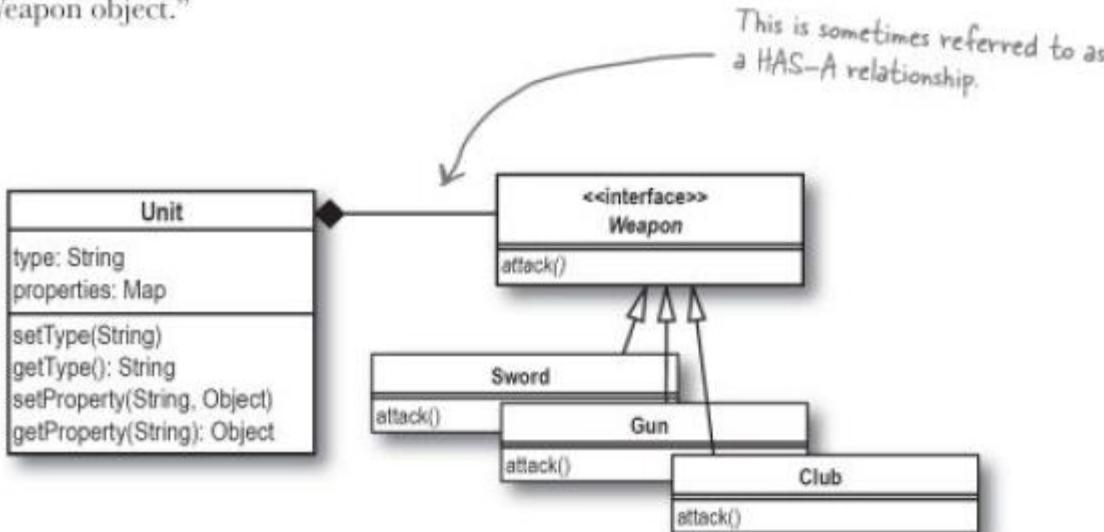
Usually, IS-A relates to inheritance, for example: "A Sword IS-A Weapon, so Sword should extend Weapon."



Use inheritance when one object behaves like another, rather than just when the IS-A relationship applies.

HAS-A refers to composition or aggregation.

HAS-A refers to composition and aggregation, so you might hear, "A Unit HAS-A Weapon, so a Unit can be composed with a Weapon object."



Cohesive Class

Cohesion, and one reason for a class to change

Sweet! Our software is easy to change...
...but what about that "cohesive" thing?

**A cohesive class does
one thing**

The more cohesive
your classes are, the
higher the cohesion
of your software.

Cohesive classes are focused
on specific tasks. Our
Inventory class worries about
just Rick's inventory, not
what woods can be used in
a guitar, or how to compare
two instrument specs.

**really well and
does not try to**

Look through the methods of
your classes—do they all relate
to the name of your class? If
you have a method that looks
out of place, it might belong
on another class.

do

or be

something else.

Instrument doesn't try to
handle searches, or keep
up with what woods are
available. It is focused on
describing an instrument—and
nothing else.

Tools: CRC Cards

CRC stands for Class, Responsibility, Collaborator. These cards are used to take a class and figure out what its responsibility should be, and what other classes it collaborates with.

Class: BarkRecognizer

Description: This class is the interface to the bark recognition hardware.

Responsibilities:

Name	Collaborator
Tell the door to open	DogDoor ←

If there are other classes involved in this job, list them in this column.

List each job this class needs to do.

Class: DogDoor

Description: Represents the physical dog door. This provides an interface to the hardware that actually controls the door.

Responsibilities:

Name	Collaborator
Open the door	
Close the door	

There's no collaborator class for these.

Be sure you write down things that this class does on its own, as well as things it collaborates with other classes on.

Tools: SRP

Any time you see "Gets", it's probably not the responsibility of this class to do a certain task.

Class: Automobile

Description: This class represents a car and its related functionality

Responsibilities:

Name	Collaborator
Starts itself.	
Stops itself.	
Gets tires changed	Mechanic, Tire
Gets driven	Driver
Gets washed	CarWash, Attendant
Gets oil checked	Mechanic
Reports on oil levels	

Technically, you don't need to list responsibilities that AREN'T this class's, but doing things this way can help you find tasks that really shouldn't be on this class.

CRC cards help implement the SRP

SRP Analysis for Automobile

The Automobile start[s] itself.
The Automobile stop[s] itself.
The Automobile changeTires itself.
The Automobile drive[s] itself.
The Automobile wash[es] itself.
The Automobile check[s] oil itself.
The Automobile get[s] oil itself.

Need for Design Patterns

Design patterns
help you recognize
and implement
GOOD solutions to
common problems.

Anti patterns are
about recognizing
and avoiding
BAD solutions to
common problems.

Shared Vocabulary?

Overheard at the local diner...

Alice

I need a Cream cheese with jelly on white bread, a chocolate soda with vanilla ice cream, a grilled cheese sandwich with bacon, a tuna fish salad on toast, a banana split with ice cream & sliced bananas and a coffee with a cream and two sugars, ... oh, and put a hamburger on the grill!

Flo

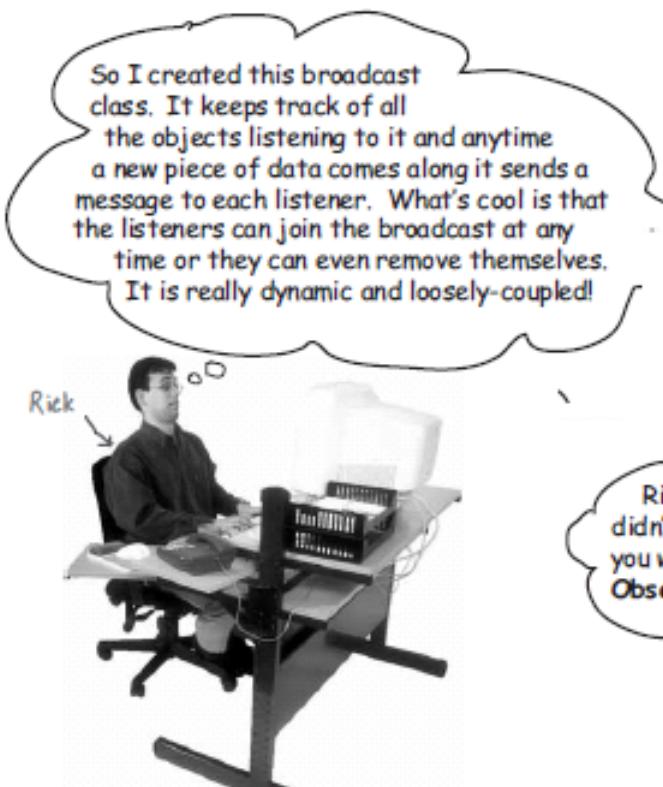
Give me a C.J. White, a black & white, a Jack Benny, a radio, a house boat, a coffee regular and burn one!



Cont...

The power of a shared pattern vocabulary

When you communicate using patterns you are doing more than just sharing LINGO.



Shared pattern vocabularies are POWERFUL.

Patterns allow you to say more with less.

Talking at the pattern level allows you to stay "in the design" longer.

Shared vocabularies can turbo charge your development team.

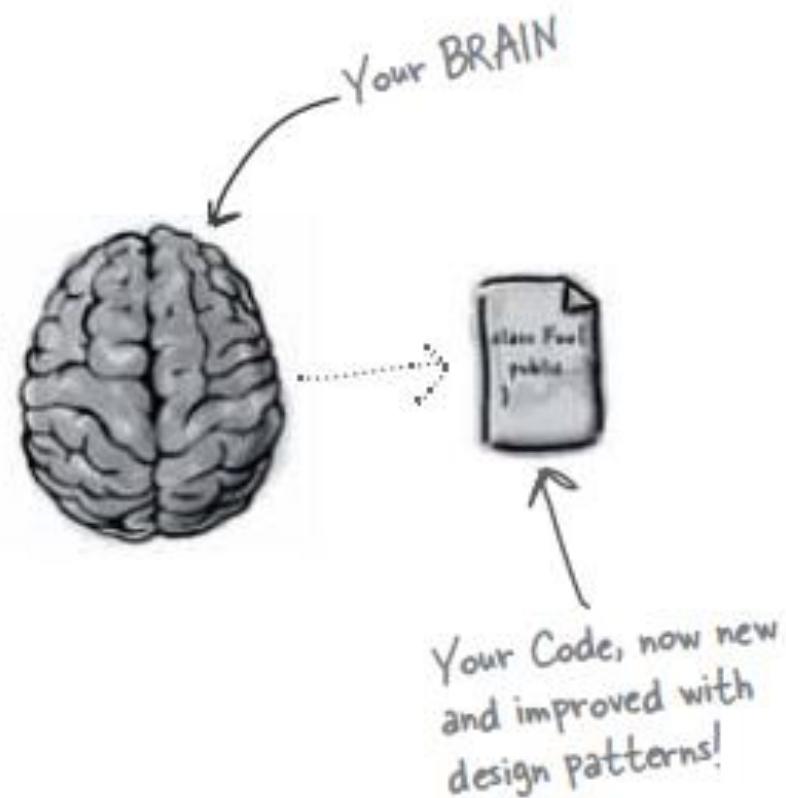
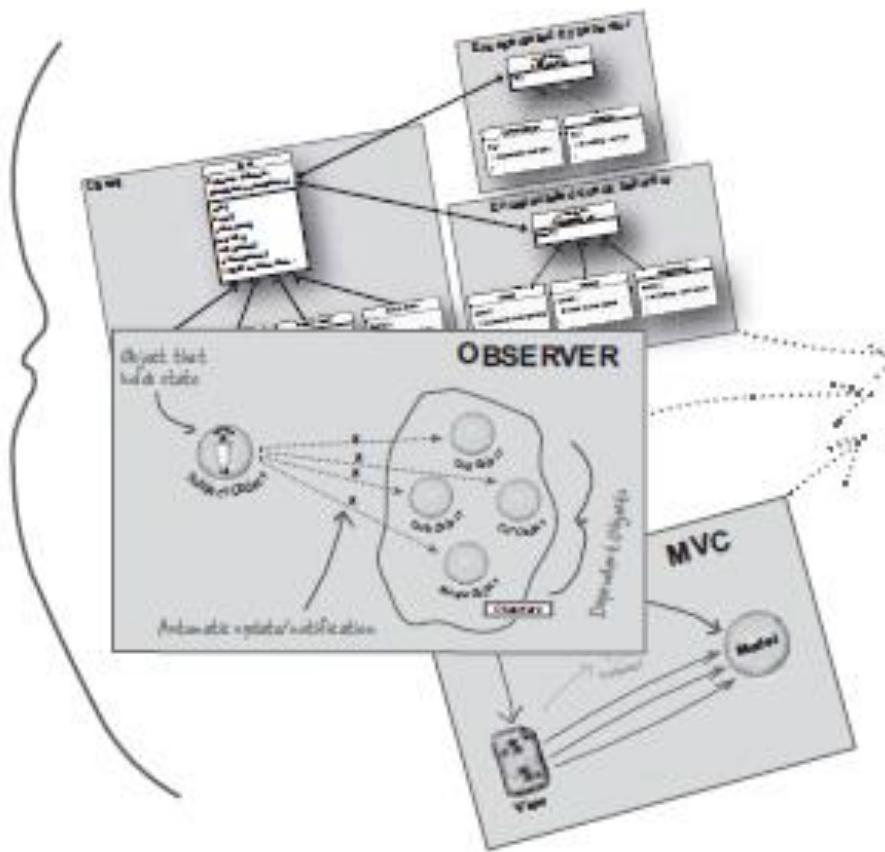
Shared vocabularies encourage more junior developers to get up to speed.

Exactly. If you communicate in patterns, then other developers know immediately and precisely the design you're describing. Just don't get Pattern Fever... you'll know you have it when you start using patterns for Hello World...

Usage

How do I use Design Patterns?

A Bunch of Patterns



Pattern Defined

A Pattern is a solution to a problem in a context.

The **context** is the situation in which the pattern applies. This should be a recurring situation.

The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.

The **solution** is what you're after: a general design that anyone can apply which resolves the goal and set of constraints.

Example: You have a collection of objects.

You need to step through the objects without exposing the collection's implementation.

Encapsulate the iteration into a separate class.

"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

GoF!

the Gang of Four

Today there are more patterns than in the GoF book; learn about them as well.

Shoot for practical extensibility. Don't provide hypothetical generality; be extensible in ways that matter.

Go for simplicity and don't become over-excited. If you can come up with a simpler solution without using a pattern, then go for it.



Patterns are tools not rules - they need to be tweaked and adapted to your problem.

Erich Gamma

Design Patterns
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



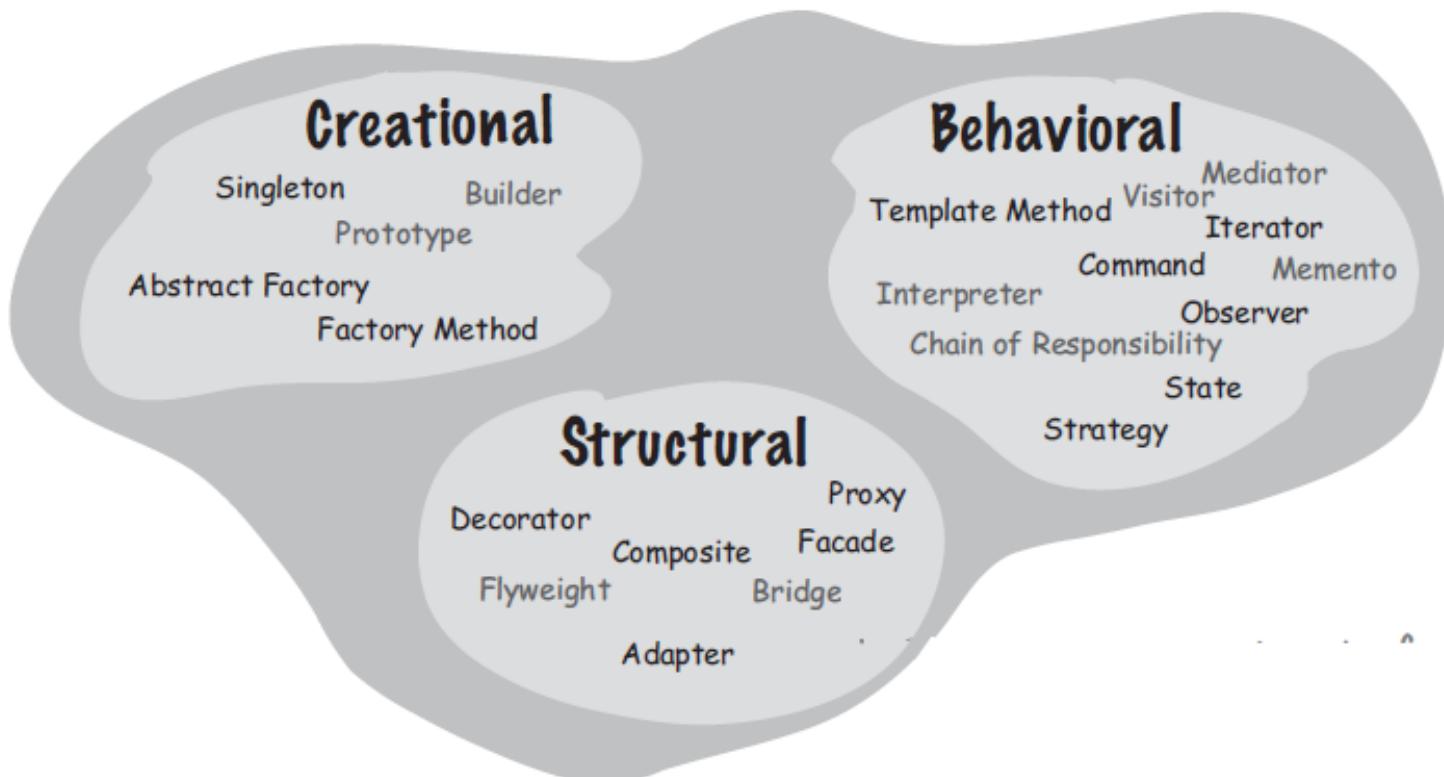
Foreword by Grady Booch



Design Pattern Categories

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



Structural patterns let you compose classes or objects into larger structures.

Summary of Discussions so far

TARGET



Always strive to achieve a Design

- That can Accommodate **CHANGE**
- To Achieve **RE-USABILITY**
- To Achieve **EXTENDIBILITY**
- To Achieve **MAINTANABILITY**
- To Reduce **RISK**



Design Approach

NOTHING EVER STAYS SAME – **CHANGE IS CONSTANT**

- Make sure the S/w does what it supposed to do
- Apply basic OO principles to add flexibility in code
- Strive for a maintainable & re-usable design

Approach:

- Break bigger problem to smaller sections
- Use the principles as guidance to design and code to small sections
- Iterate to complete the whole system

DESIGN USING OO PRINCIPLES [NOT AS PATTERNS]
– OUTPUT IS AUTOMATICALLY A PATTERN



✓ OO **BASICS** (4 Pillars)

- ABSTRACTION
- ENCAPSULATION
- POLYMORPHISM
- INHERITANCE

Design Guidelines / Checkpoints

– OO Principles



- ✓ Identify Items that can **CHANGE** in **FUTURE**
 - **ENCAPSULATE** what **VARIES**
- ✓ Identify areas of **DUPLICATION**
 - Look for **ABSTRACTION / ENCAPSULATION**
- ✓ Identify if we are programming to **CONCREATE** classes
 - Look for **ABSTRACTION** – use **INTERFACES**
- ✓ **FAVOR COMPOSITION (HAS-A)** instead of Inheritance wherever required
- ✓ Identify areas of dependency
 - Look for **HIGH CONHESION (SRP) & LOOSE COUPLING**
 - Look for **DELEGATION**
- ✓ Think in terms of OO Principles rather than Design patterns when starting to design
- ✓ Use Design Pattern Catalogue wherever required



SOLID - Design Smell / Principles



- ✓ **SRP** – Single Responsibility Principle
 - Class Should have a Single Responsibility
- ✓ **OCP** – Open Closed Principle
 - S/w Entities should be Open for Extension and Closed for Modification
- ✓ **LSP** – Liskov Substitution Principle
 - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program” – Design by Contract
- ✓ **ISP** – Interface Segregation Principle
 - Many client-specific interfaces are better than one general-purpose interface
 - Clients should not be forced to depend upon interfaces that they don't use
- ✓ **DIP** – Dependency Inversion Principle
 - One should “Depend upon Abstractions. Do not depend upon concretions
- ✓ **DRY** – Don’t Repeat Yourself
 - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Above all Never Forget To

“KISS YAGNI”

- Keep it Stupid Simple
- Keep it Simple & Straight Forward
- Keep it Simple & Stupid
- You Ain't Gonna Need it – Do the simplest that could possibly work

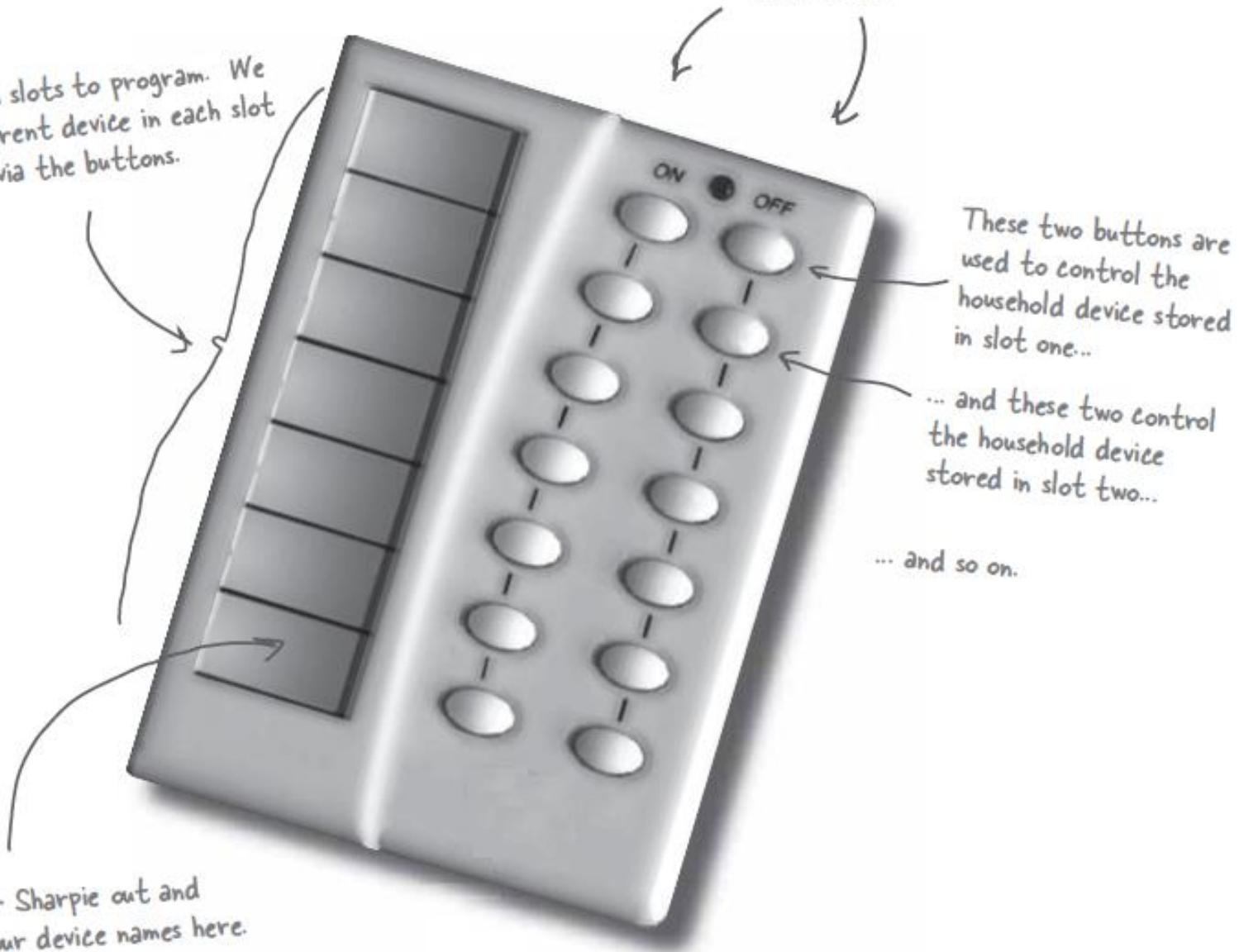
More Design Patterns

Pattern 2

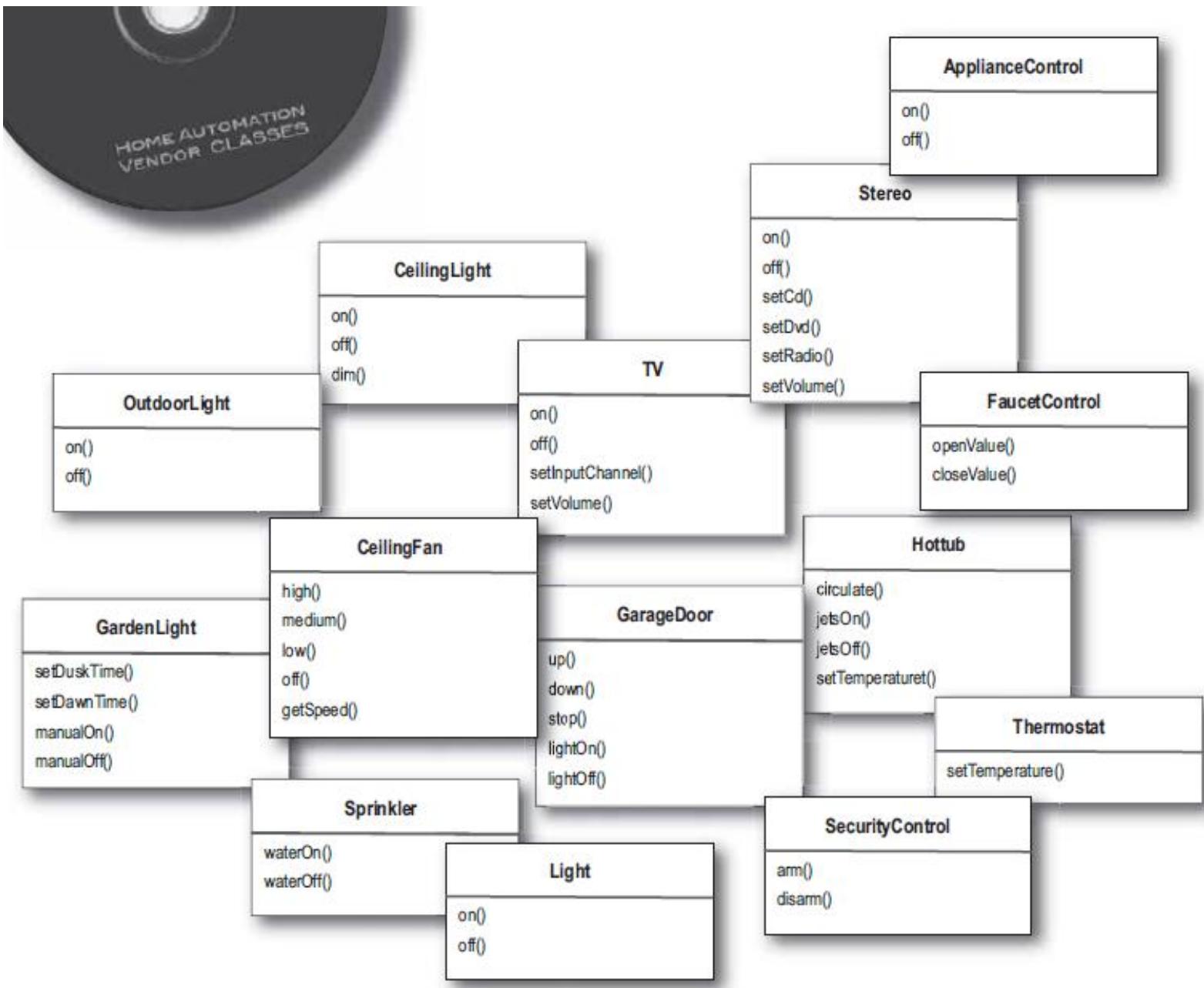
Home Automation

We've got seven slots to program. We can put a different device in each slot and control it via the buttons.

There are "on" and "off" buttons for each of the seven slots.

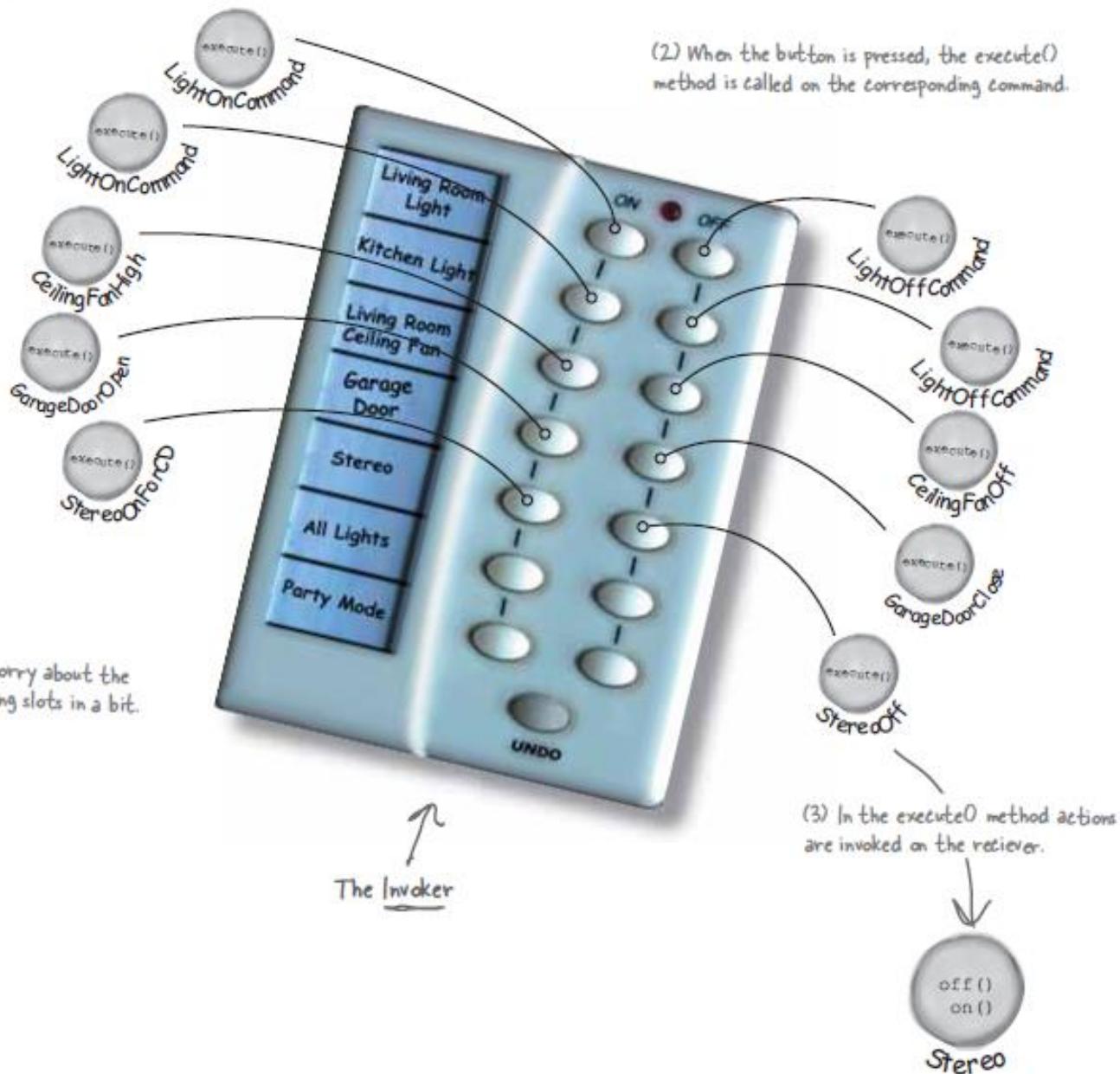


Cont... (Vendor Provided Classes)



Command Assigned to Slots

(1) Each slot gets a command.



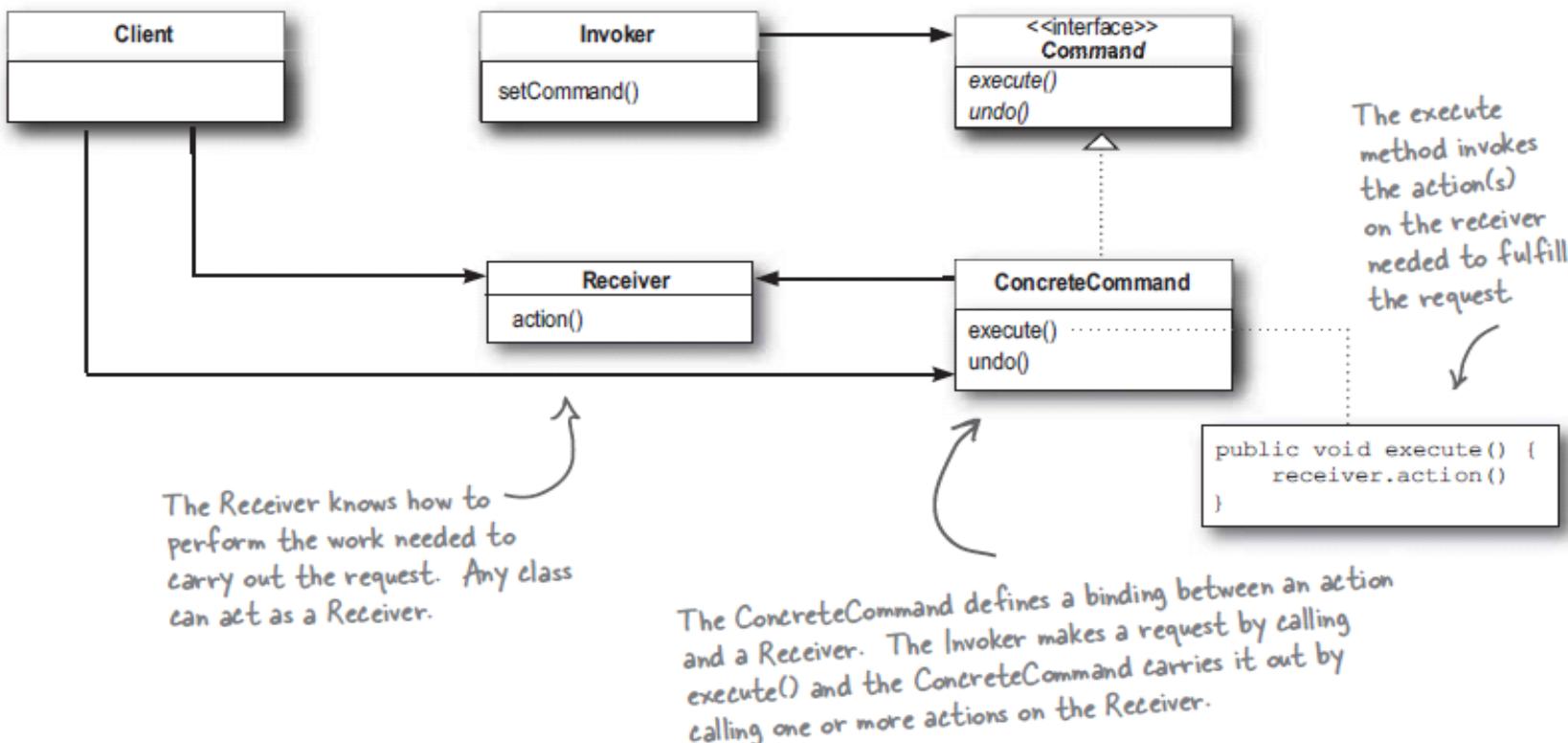
Command Pattern

The **Command Pattern** encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

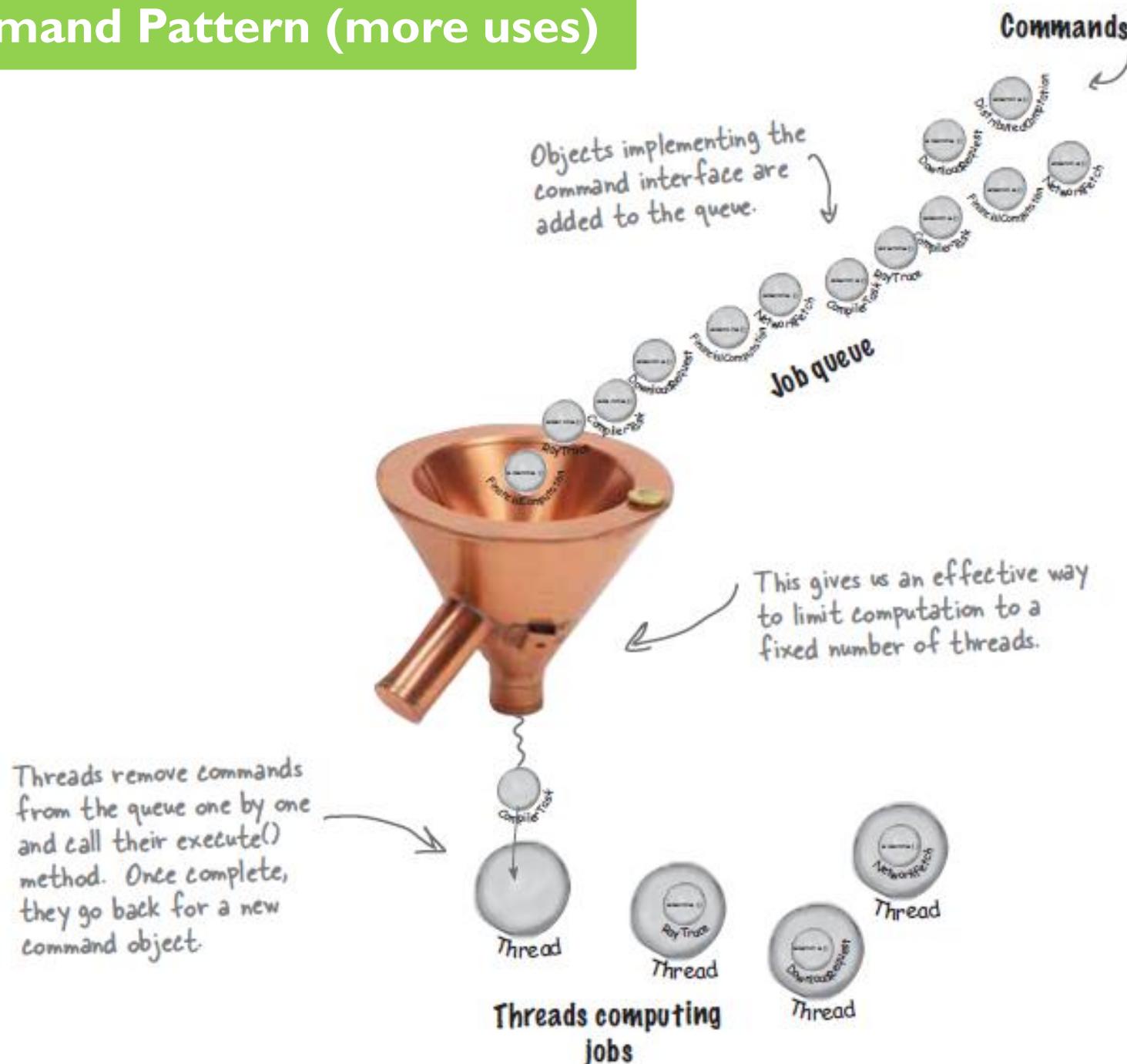
The Client is responsible for creating a **ConcreteCommand** and setting its **Receiver**.

The **Invoker** holds a command and at some point asks the command to carry out a request by calling its **execute()** method.

Command declares an interface for all commands. As you already know, a command is invoked through its **execute()** method, which asks a receiver to perform an action. You'll also notice this interface has an **undo()** method, which we'll cover a bit later in the chapter.



Command Pattern (more uses)

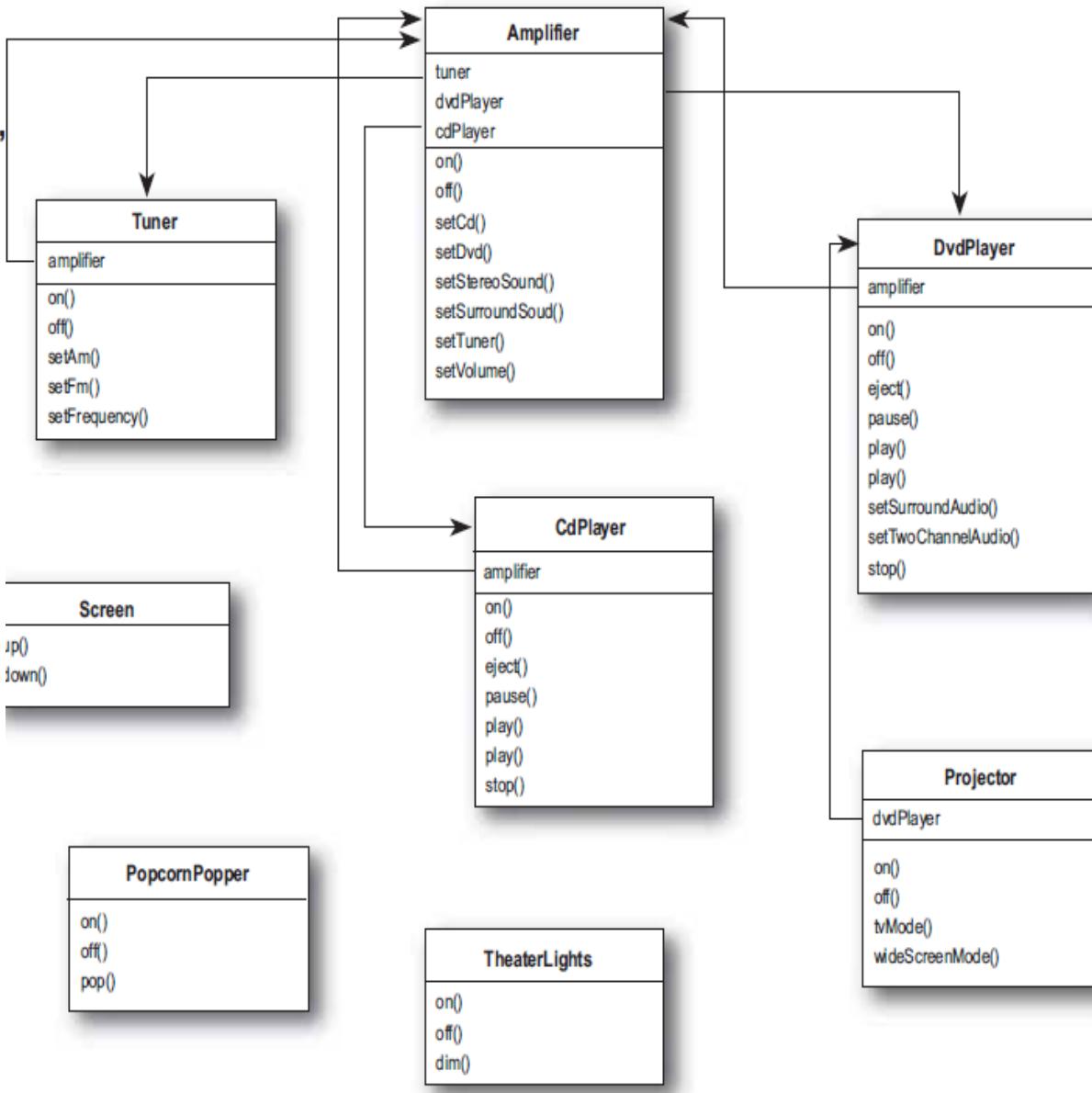


Pattern 3

Watching a movie (the hard way)

Pick out a DVD, relax, and get ready for movie magic. Oh, there's just one thing – to watch the movie, you need to perform a few tasks:

- ① Turn on the popcorn popper
- ② Start the popper popping
- ③ Dim the lights
- ④ Put the screen down
- ⑤ Turn the projector on
- ⑥ Set the projector input to DVD
- ⑦ Put the projector on wide-screen mode
- ⑧ Turn the sound amplifier on
- ⑨ Set the amplifier to DVD input
- ⑩ Set the amplifier to surround sound
- ⑪ Set the amplifier volume to medium (5)
- ⑫ Turn the DVD Player on
- ⑬ Start the DVD Player playing



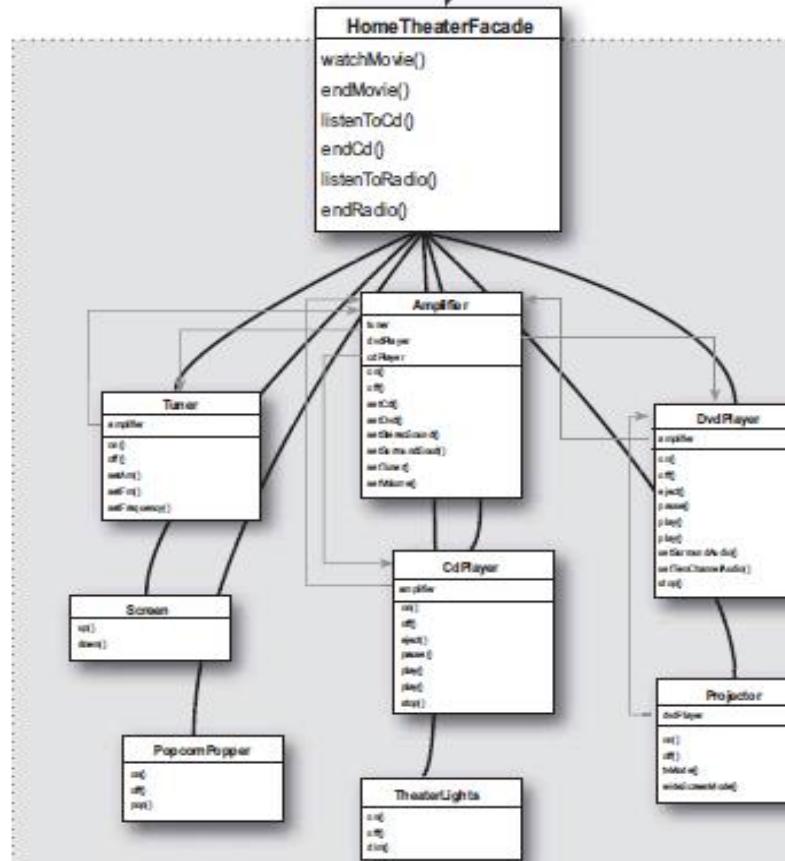
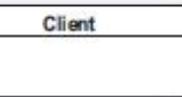
Home Theater Facade

The HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

We can upgrade the home theater components without affecting the client.

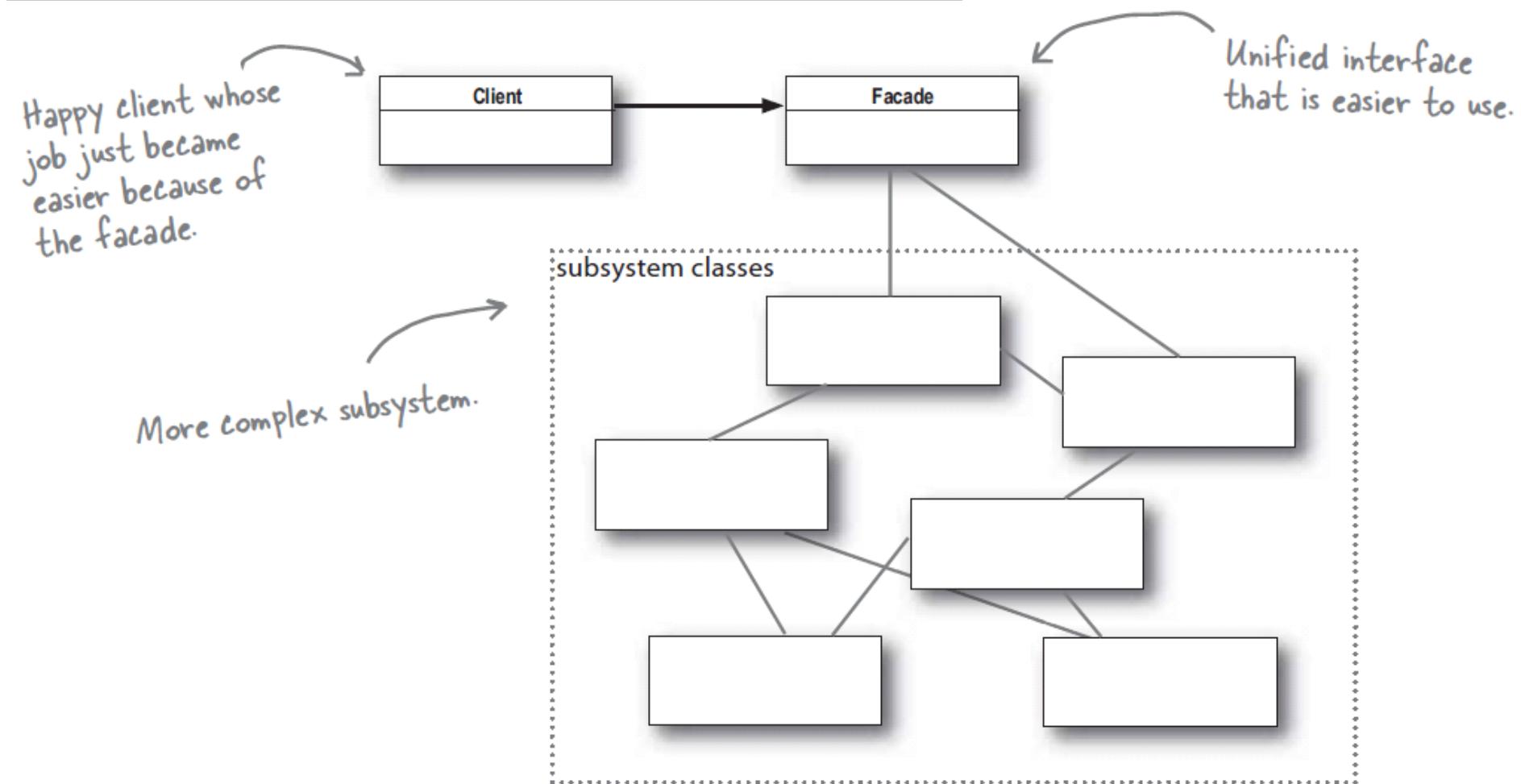
We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.

This client only has one friend; the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!

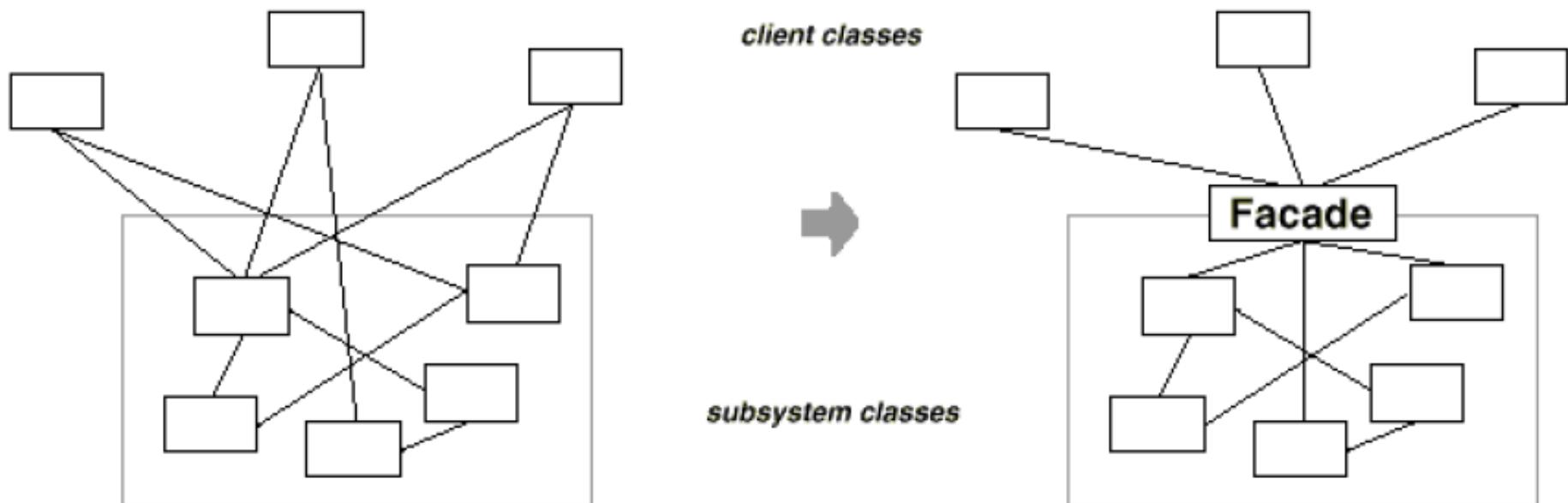


Façade Pattern

The **Facade Pattern** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Façade Pattern (Cont...)



Implementation

```
public class HomeTheaterFacade {  
    Amplifier amp;  
    Tuner tuner;  
    DvdPlayer dvd;  
    CdPlayer cd;  
    Projector projector;  
    TheaterLights lights;  
    Screen screen;  
    PopcornPopper popper;  
  
    public HomeTheaterFacade(Amplifier amp,  
                             Tuner tuner,  
                             DvdPlayer dvd,  
                             CdPlayer cd,  
                             Projector projector,  
                             Screen screen,  
                             TheaterLights lights,  
                             PopcornPopper popper) {  
  
        this.amp = amp;  
        this.tuner = tuner;  
        this.dvd = dvd;  
        this.cd = cd;  
        this.projector = projector;  
        this.screen = screen;  
        this.lights = lights;  
        this.popper = popper;  
    }  
  
    // other methods here  
}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Implementation (Cont...)

```
public void watchMovie(String movie) {  
    System.out.println("Get ready to watch a movie...");  
    popper.on();  
    popper.pop();  
    lights.dim(10);  
    screen.down();  
    projector.on();  
    projector.wideScreenMode();  
    amp.on();  
    amp.setDvd(dvd);  
    amp.setSurroundSound();  
    amp.setVolume(5);  
    dvd.on();  
    dvd.play(movie);  
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

```
public void endMovie() {  
    System.out.println("Shutting movie theater down...");  
    popper.off();  
    lights.on();  
    screen.up();  
    projector.off();  
    amp.off();  
    dvd.stop();  
    dvd.eject();  
    dvd.off();  
}
```

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

The Principle of Least Knowledge

The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close “friends.”
The principle is usually stated as:

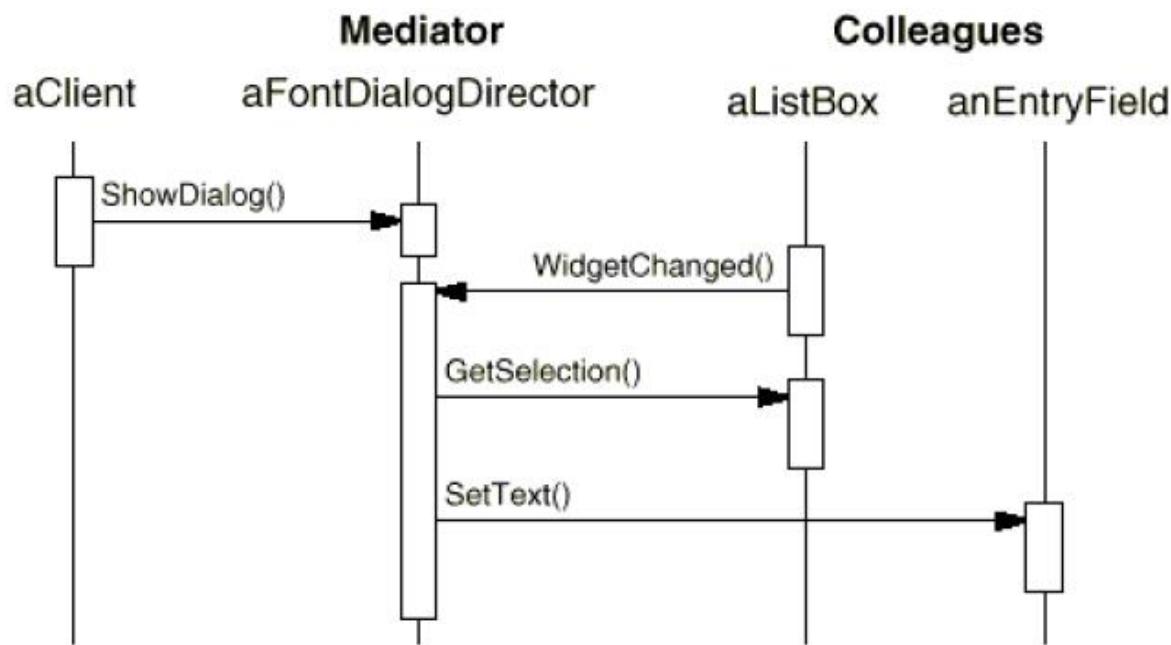
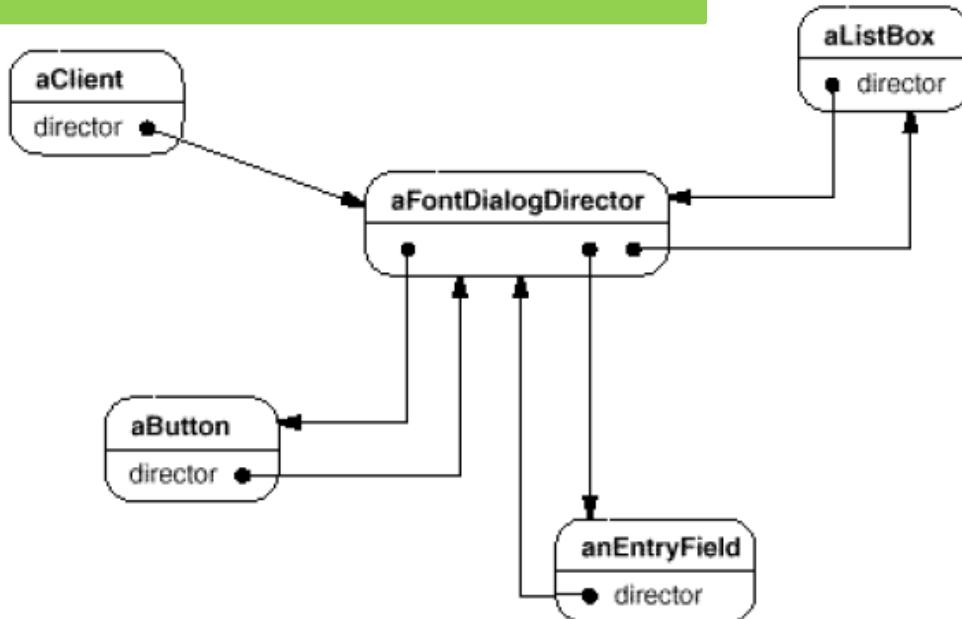


Design Principle

*Principle of Least Knowledge -
talk only to your immediate friends.*

But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

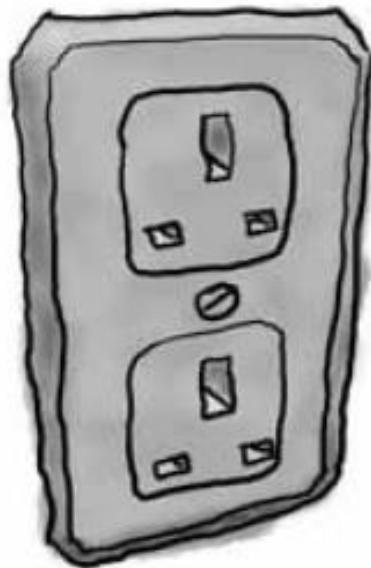
Pattern 4: Mediator



Pattern 5

Adapters all around us

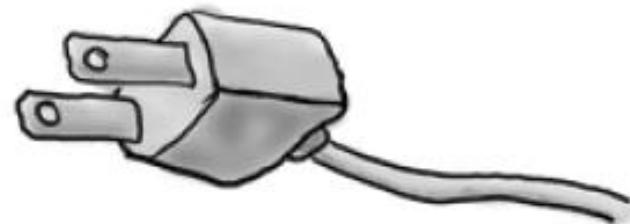
European Wall Outlet



AC Power Adapter



Standard AC Plug

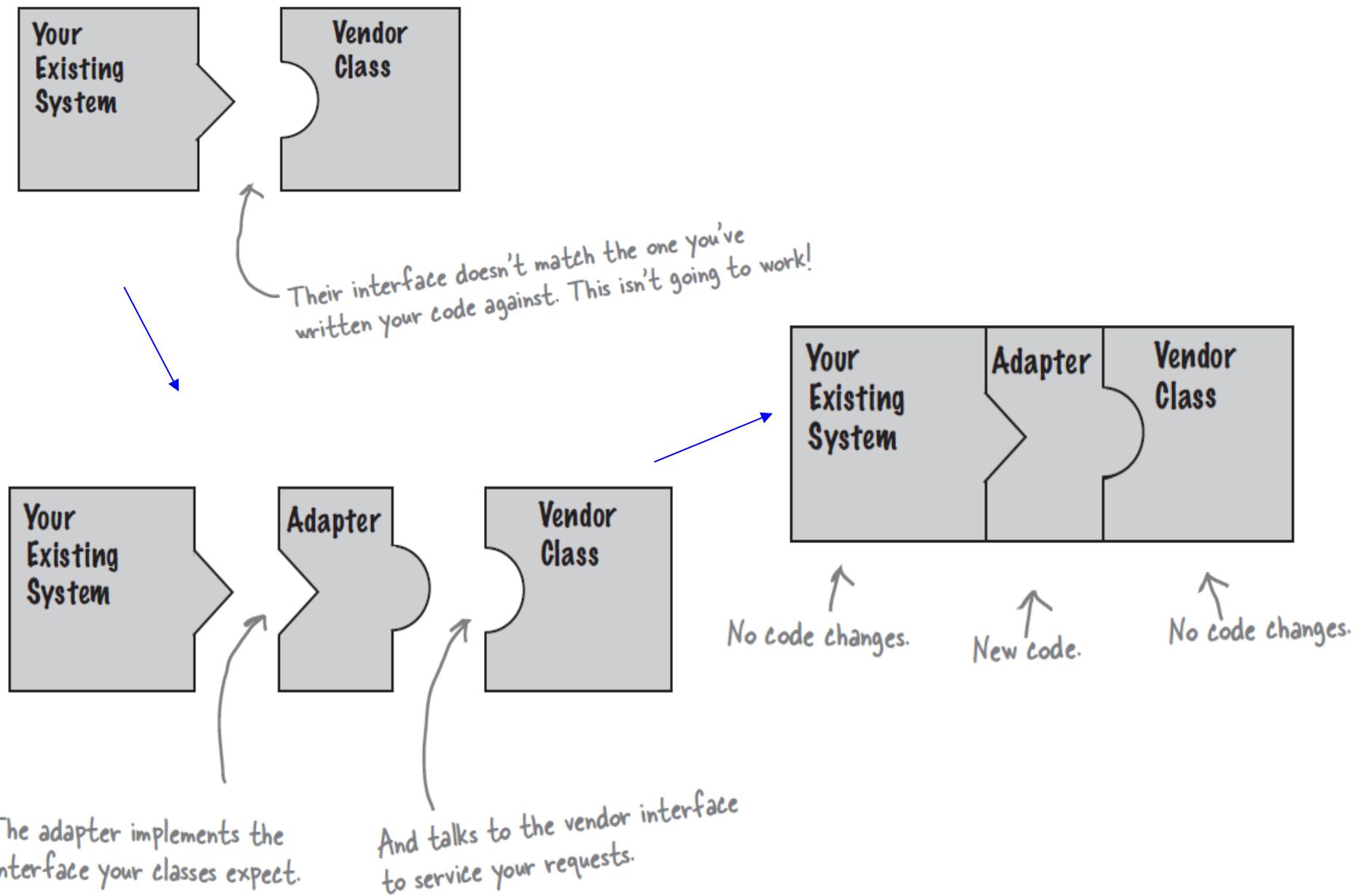


The European wall outlet exposes
one interface for getting power.

The adapter converts one
interface into another.

The US laptop expects
another interface.

OO Adaptors

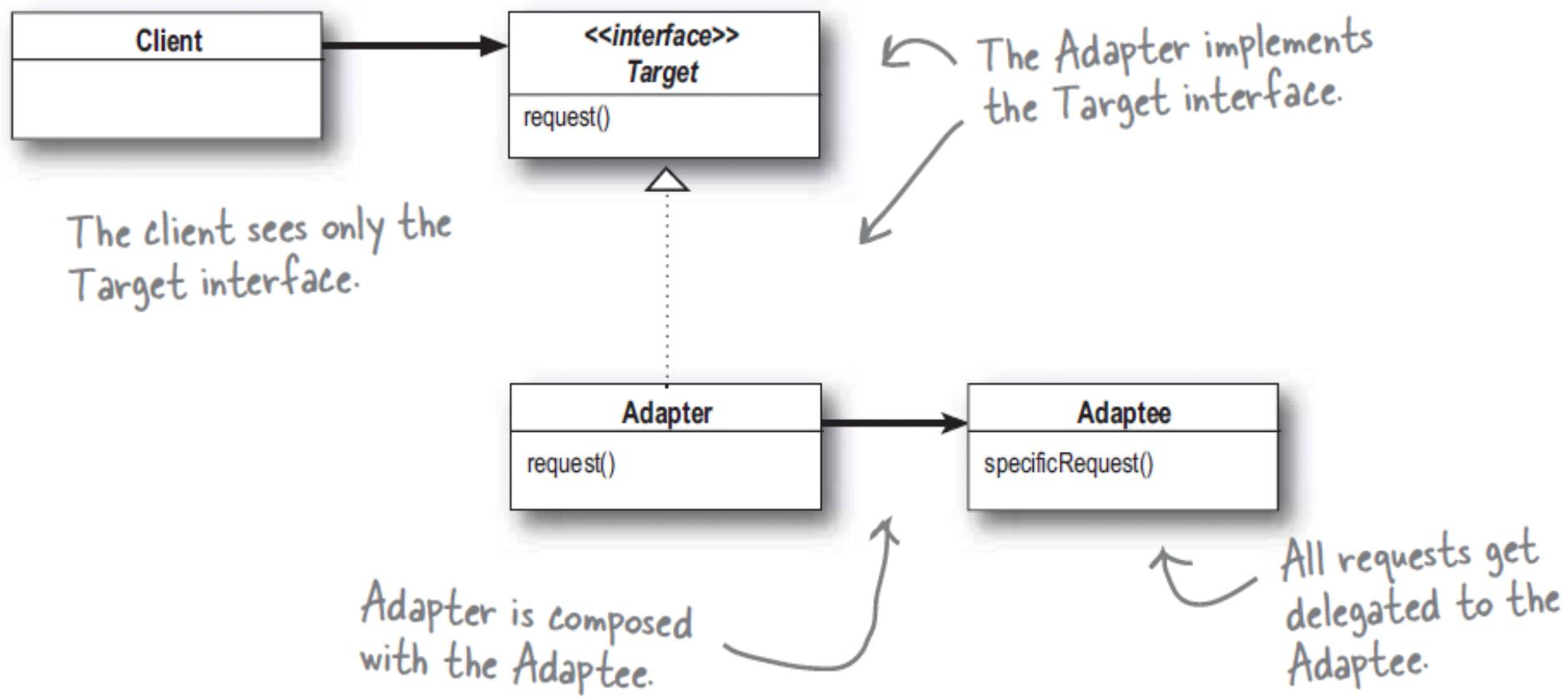


Camera Lens Adaptor



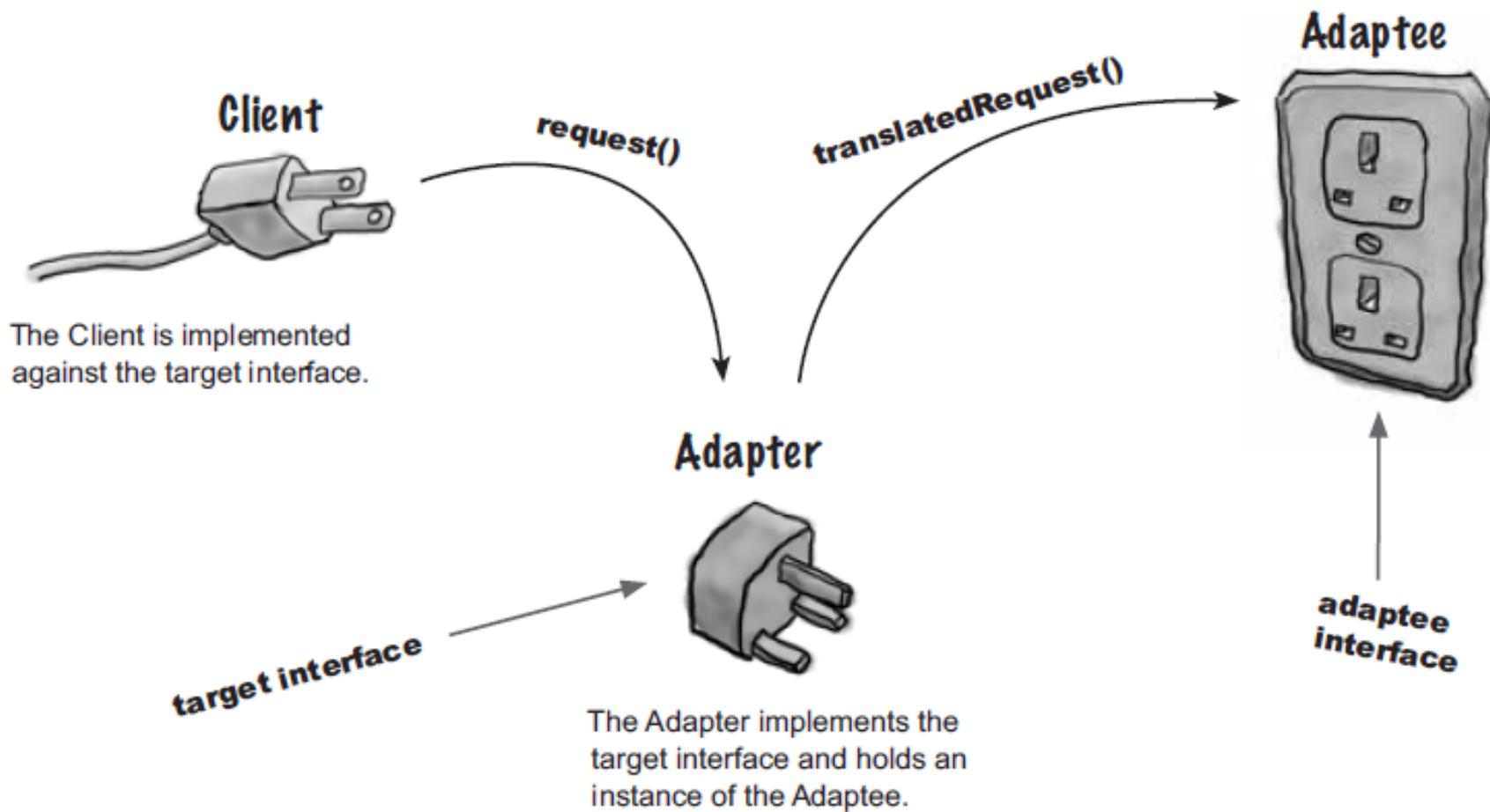
Adaptor Pattern

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.



Adaptor Explanation

The Adapter Pattern explained

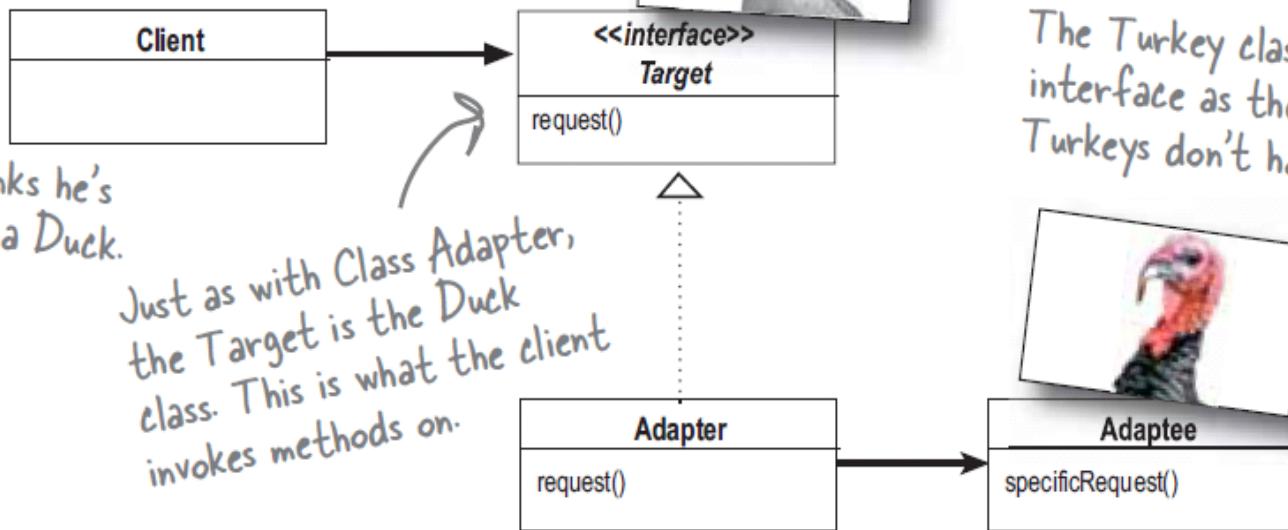


Object Adaptor

Object Adapter

Client thinks he's talking to a Duck.

Just as with Class Adapter, the Target is the Duck class. This is what the client invokes methods on.



Duck interface.



The Turkey class doesn't have the same interface as the Duck. In other words, Turkeys don't have quack() methods, etc.

Turkey object.



The Adapter implements the Duck interface, but when it gets a method call it turns around and delegates the calls to a Turkey.

Thanks to the Adapter, the Turkey (Adaptee) will get calls that the client makes on the Duck interface.

Pattern 6

Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

- (1) Boil some water
- (2) Brew coffee in boiling water
- (3) Pour coffee in cup
- (4) Add sugar and milk

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

Starbuzz Tea Recipe

- (1) Boil some water
- (2) Steep tea in boiling water
- (3) Pour tea in cup
- (4) Add lemon

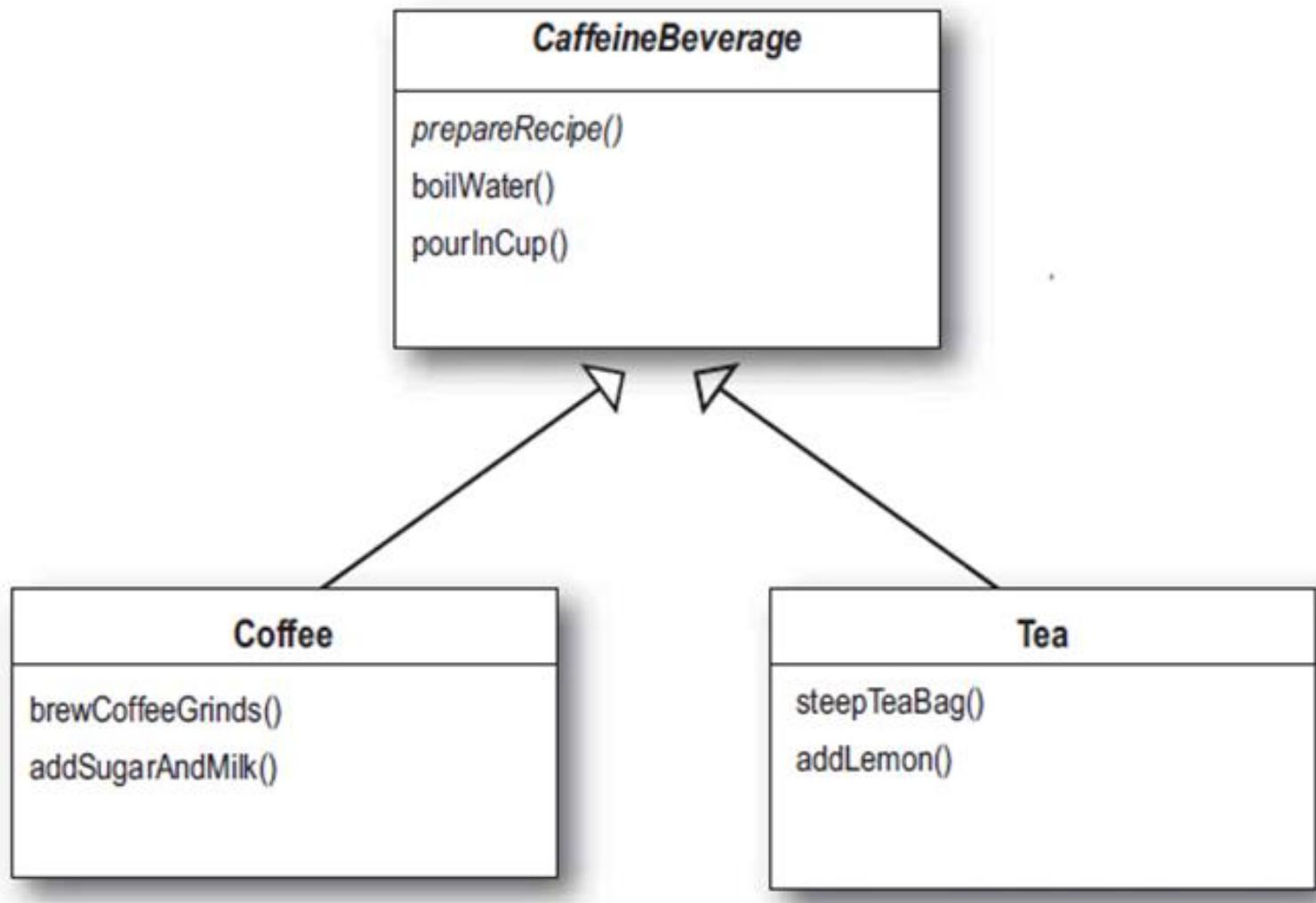
All recipes are Starbuzz coffee trade secrets and should be kept strictly confidential.

Classes

```
public class Coffee {  
  
    void prepareRecipe() {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    public void brewCoffeeGrinds() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    public void addSugarAndMilk() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

```
public class Tea {  
  
    void prepareRecipe() {  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    public void steepTeaBag() {  
        System.out.println("Steeping the tea");  
    }  
  
    public void addLemon() {  
        System.out.println("Adding Lemon");  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

Abstract common functions



Convert to abstract ones

Coffee

```
void prepareRecipe() {  
    boilWater();  
    brewCoffeeGrinds();  
    pourInCup();  
    addSugarAndMilk();  
}
```

Tea

```
void prepareRecipe() {  
    boilWater();  
    steepTeaBag();  
    pourInCup();  
    addLemon();  
}
```

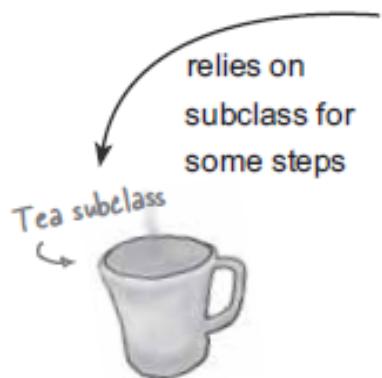
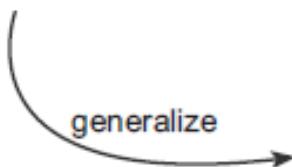


```
void prepareRecipe() {  
    boilWater();  
    brew();  
    pourInCup();  
    addCondiments();  
}
```

What have we done?

Tea

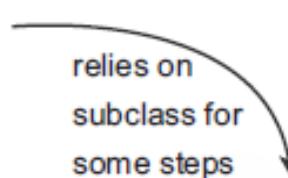
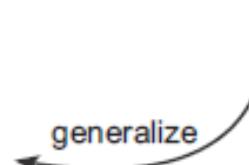
- ❶ Boil some water
- ❷ Steep the teabag in the water
- ❸ Pour tea in a cup
- ❹ Add lemon



- ❷ Steep the teabag in the water
- ❹ Add lemon

Coffee

- ❶ Boil some water
- ❷ Brew the coffee grinds
- ❸ Pour coffee in a cup
- ❹ Add sugar and milk



- ❷ Brew the coffee grinds
- ❹ Add sugar and milk

Caffeine Beverage

- ❶ Boil some water
- ❷ Brew
- ❸ Pour beverage in a cup
- ❹ Add condiments

Cont...

```
public abstract class CaffeineBeverage {  
  
    final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water")  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

Cont...

```
public class Tea extends CaffeineBeverage {  
    public void brew() {  
        System.out.println("Steeping the tea");  
    }  
    public void addCondiments() {  
        System.out.println("Adding Lemon");  
    }  
}
```

```
public class Coffee extends CaffeineBeverage {  
    public void brew() {  
        System.out.println("Dripping Coffee through filter");  
    }  
    public void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

Template structure

```
public abstract class CaffeineBeverage {
```

```
    void final prepareRecipe() {
```

```
        boilWater();
```

```
        brew();
```

```
        pourInCup();
```

```
        addCondiments();
```

```
}
```

```
abstract void brew();
```

```
abstract void addCondiments();
```

```
void boilWater() {  
    // implementation  
}
```

```
void pourInCup() {  
    // implementation  
}
```

```
}
```

prepareRecipe() is our template method.
Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class..

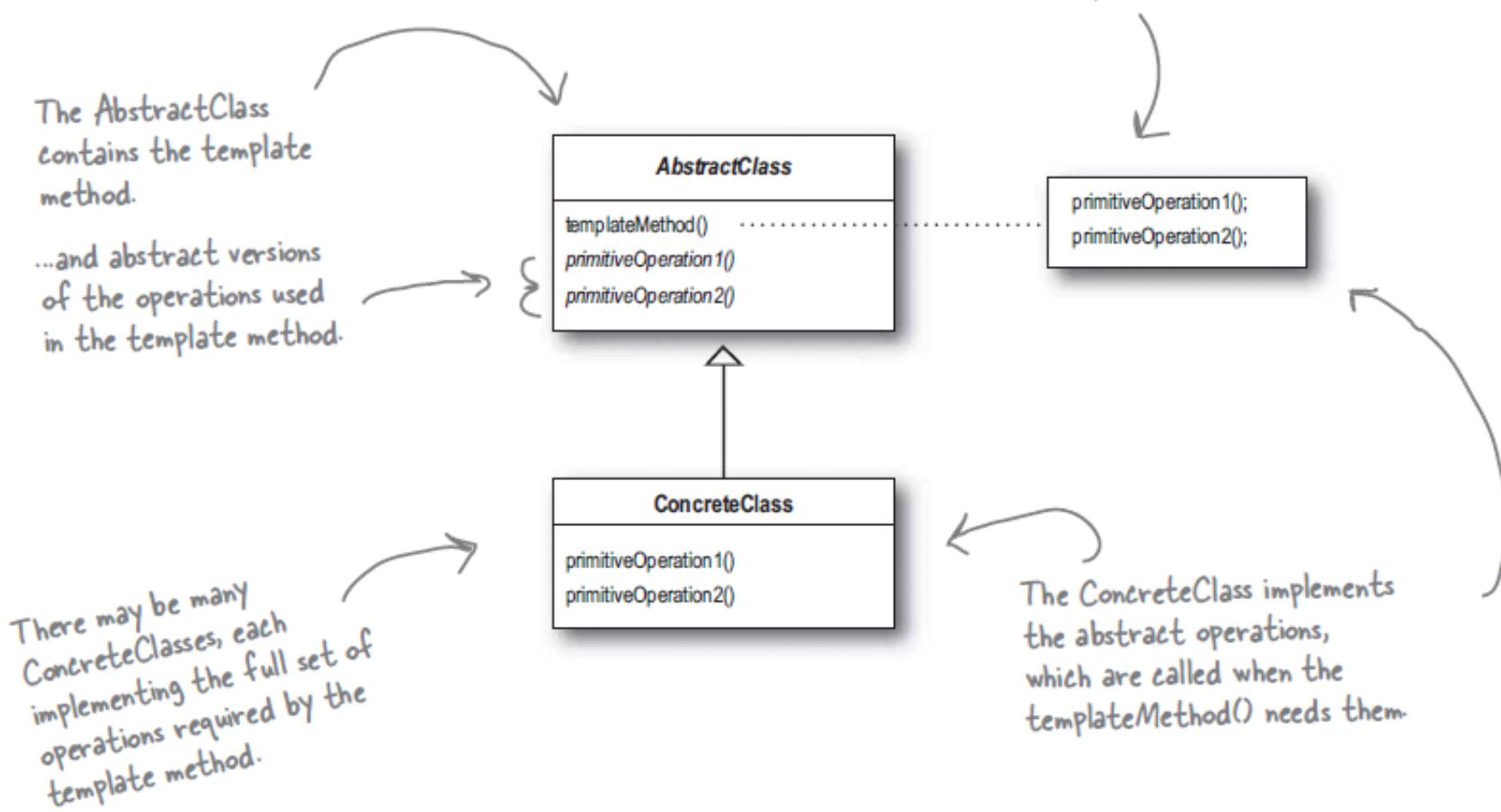
...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

Template Method 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.

The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.



Template Hooks

```
public abstract class CaffeineBeverageWithHook {  
  
    void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        if (customerWantsCondiments()) {  
            addCondiments();  
        }  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    boolean customerWantsCondiments() {  
        return true;  
    }  
}
```

Template Hooks (Cont...)

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {  
  
    public void brew() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
  
    public boolean customerWantsCondiments() {  
        String answer = getUserInput();  
  
        if (answer.toLowerCase().startsWith("y")) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    private String getUserInput() {  
        String answer = null;
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false depending on the input.

Pattern 7: Singleton Pattern



Pattern 8

Objectville Diner

Vegetarian BLT

(Fakin') Bacon with lettuce
whole wheat

2.99

BLT

Bacon with lettuce & tomato

Soup of the day

A bowl of the soup of the day
a side of potato salad

Hot Dog

A hot dog, with sauerkraut
topped with cheese

Steamed Veggies and Broccoli

A medley of steamed veg-

Objectville Pancake House

K&B's Pancake Breakfast

Pancakes with scrambled eggs, and toast

2.99

Regular Pancake Breakfast

Pancakes with fried eggs, sausage

2.99

Blueberry Pancakes

Pancakes made with fresh blueberries,
and blueberry syrup

3.49

Waffles

Waffles, with your choice of blueberries
or strawberries

3.59

Access Menu Spec

Java-Enabled Waitress: code-name "Alice"

`printMenu()`

- prints every item on the menu

`printBreakfastMenu()`

- prints just breakfast items

`printLunchMenu()`

- prints just lunch items

`printVegetarianMenu()`

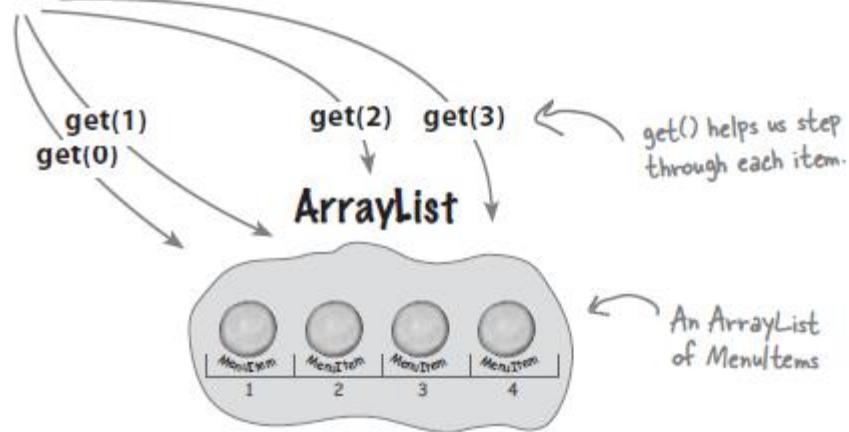
- prints all vegetarian menu items

`isItemVegetarian(name)`

- given the name of an item, returns true
if the item is vegetarian, otherwise,
returns false

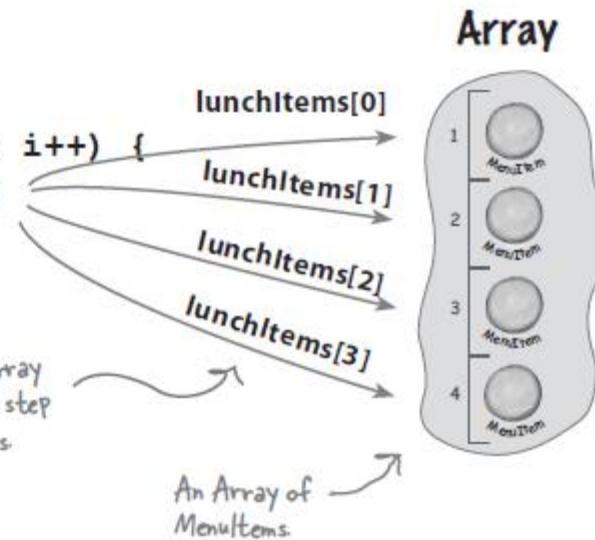
Individual Access

```
for (int i = 0; i < breakfastItems.size(); i++) {  
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);  
}
```



```
for (int i = 0; i < lunchItems.length; i++) {  
    MenuItem menuItem = lunchItems[i];  
}
```

We use the array subscripts to step through items.

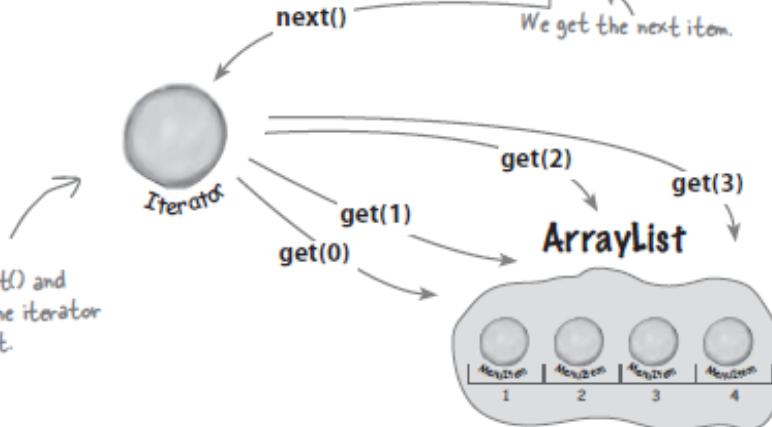


Iterator Access

```
Iterator iterator = breakfastMenu.createIterator();

while (iterator.hasNext()) {           ← And while there are more items left...
    MenuItem menuItem = (MenuItem) iterator.next();
}
```

The client just calls `hasNext()` and `next()`; behind the scenes the iterator calls `get()` on the `ArrayList`.

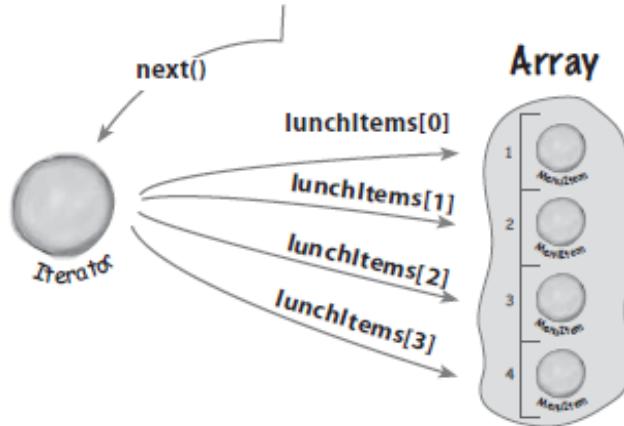


```
Iterator iterator = lunchMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem) iterator.next();
}
```

Wow, this code
is exactly the
same as the
breakfastMenu
code.

Same situation here: the client just calls `hasNext()` and `next()`; behind the scenes, the iterator indexes into the `Array`.

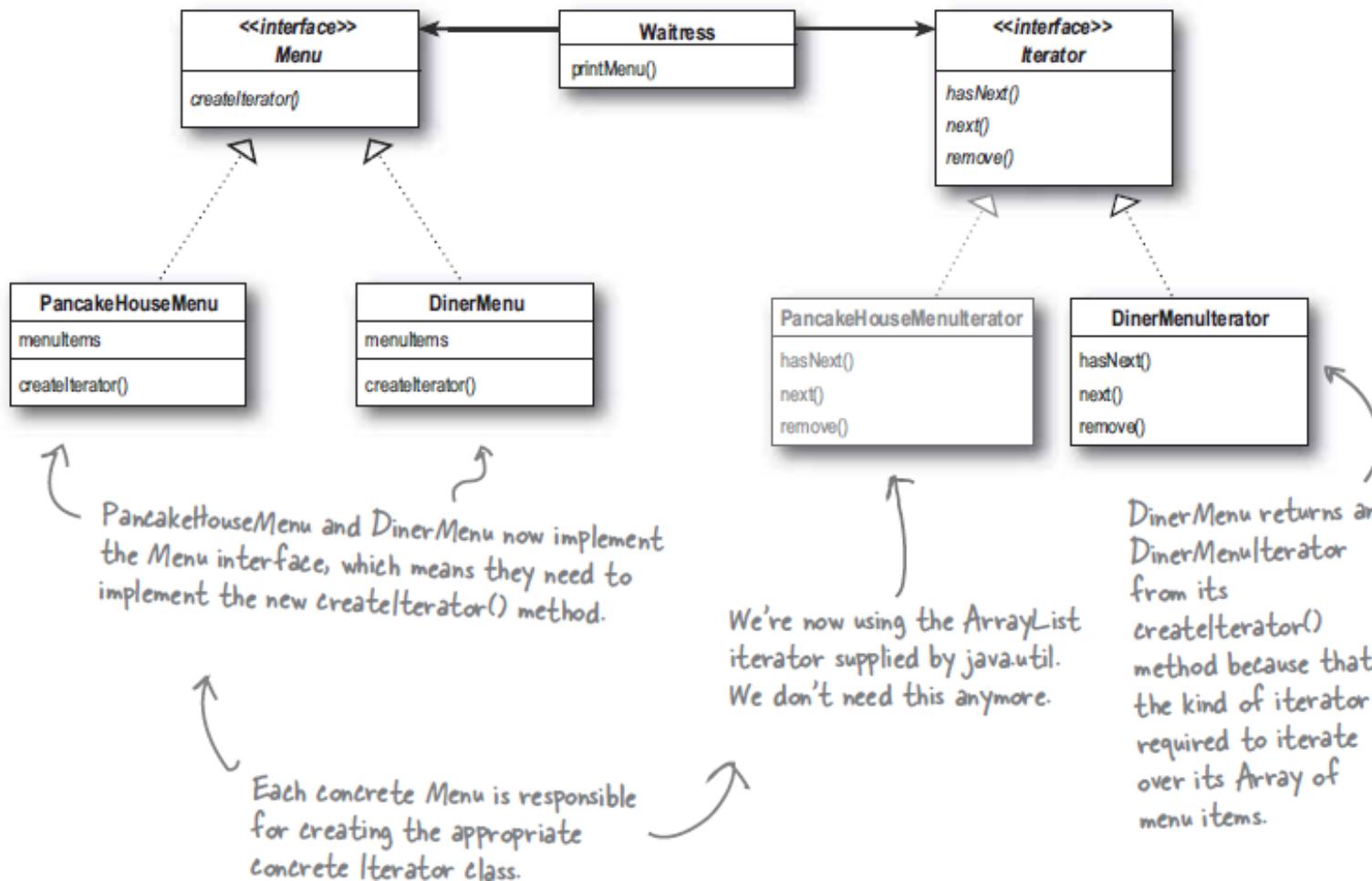


What did we do?

Here's our new Menu interface. It specifies the new method, `createIterator()`.

Now, Waitress only needs to be concerned with Menus and Iterators.

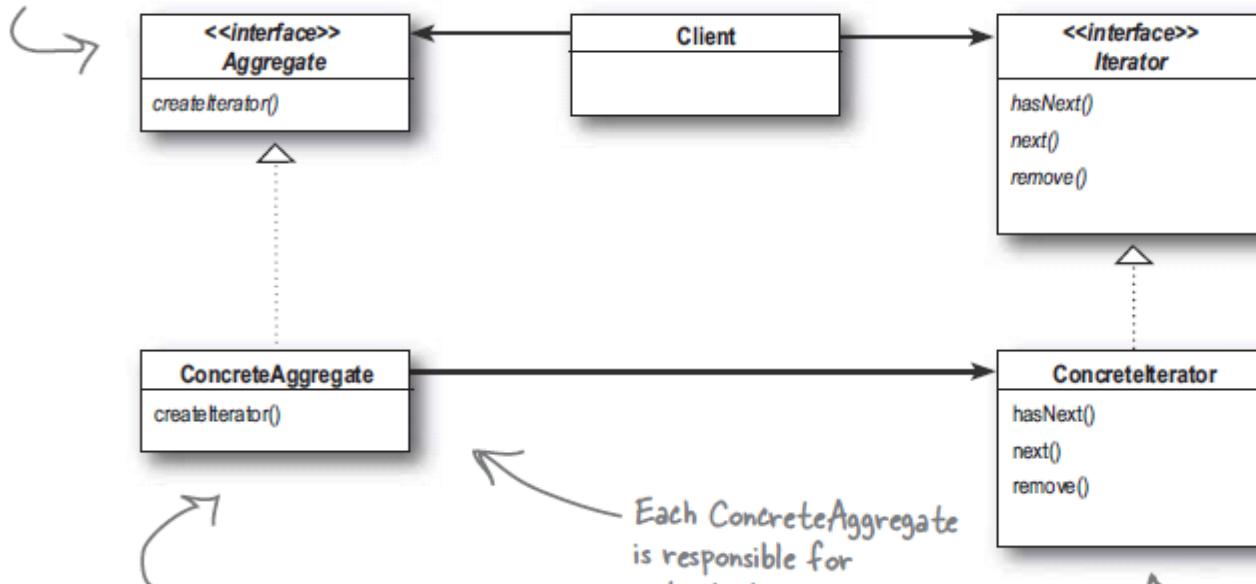
We've decoupled Waitress from the implementation of the menus, so now we can use an Iterator to iterate over any list of menu items without having to know about how the list of items is implemented.



Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.

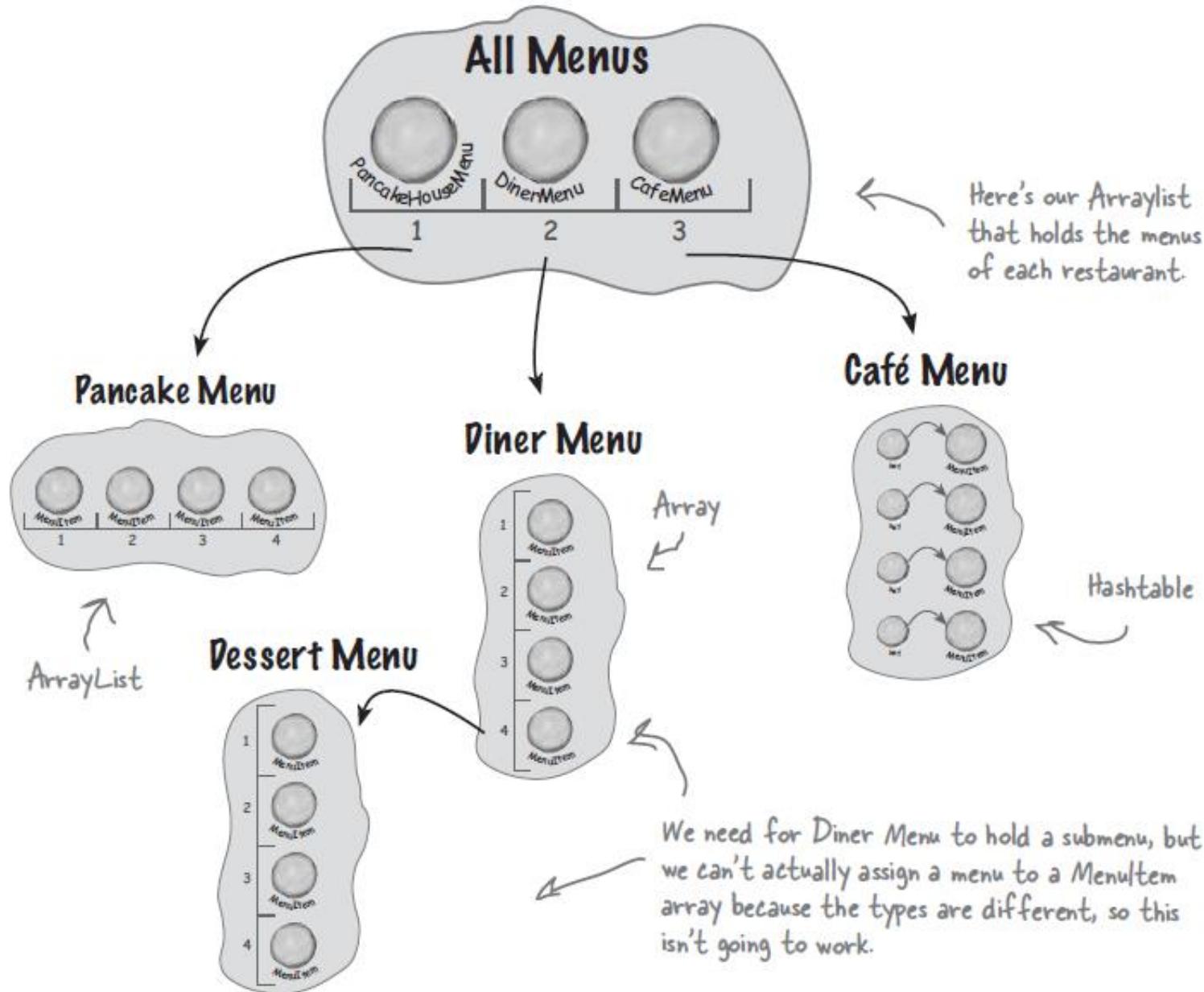


The **ConcreteAggregate** has a collection of objects and implements the method that returns an **Iterator** for its collection.

Each **ConcreteAggregate** is responsible for instantiating a **Concreteliterator** that can iterate over its collection of objects.

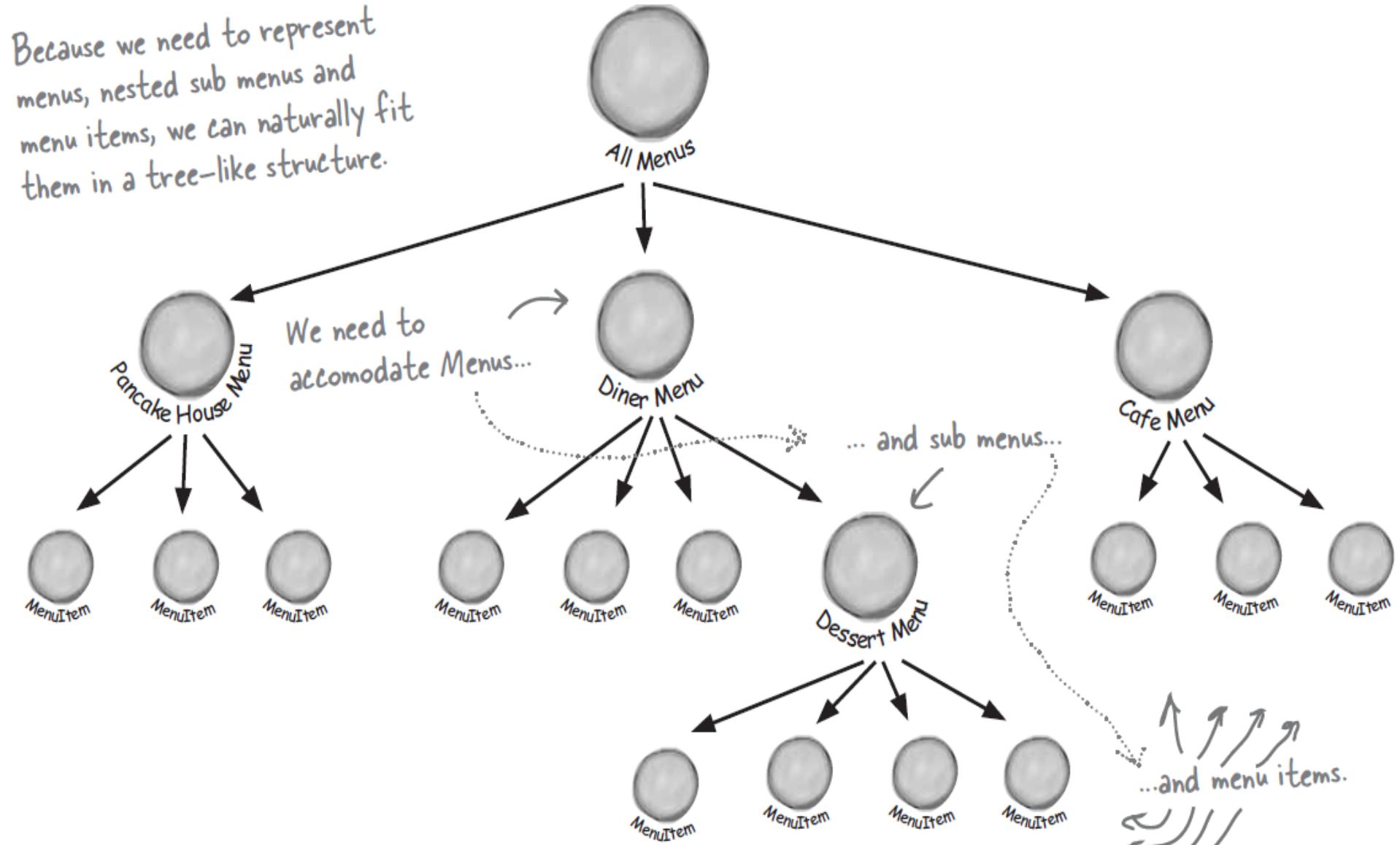
The **Concreteliterator** is responsible for managing the current position of the iteration.

Pattern 9: Change

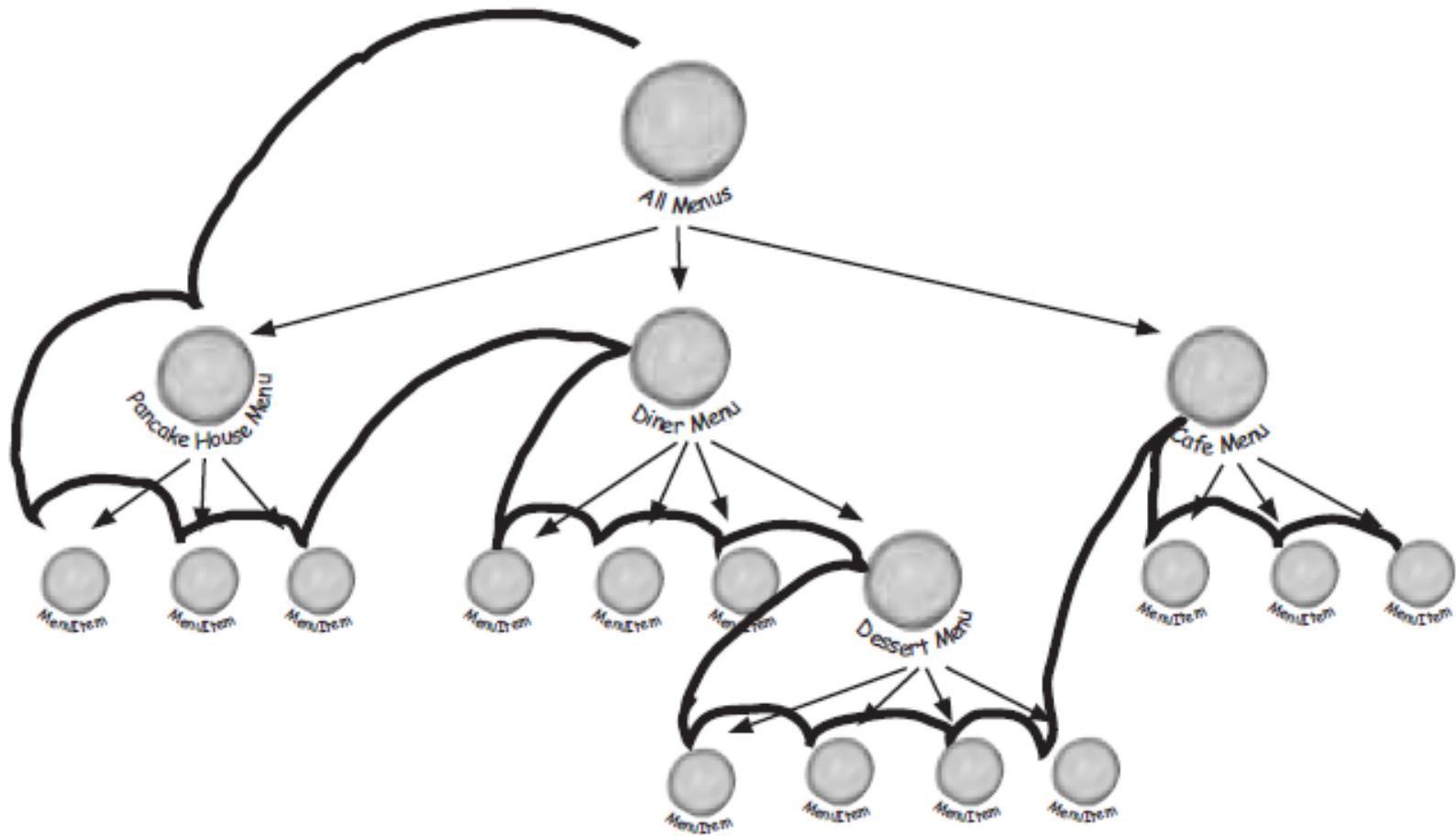


Representation

Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.

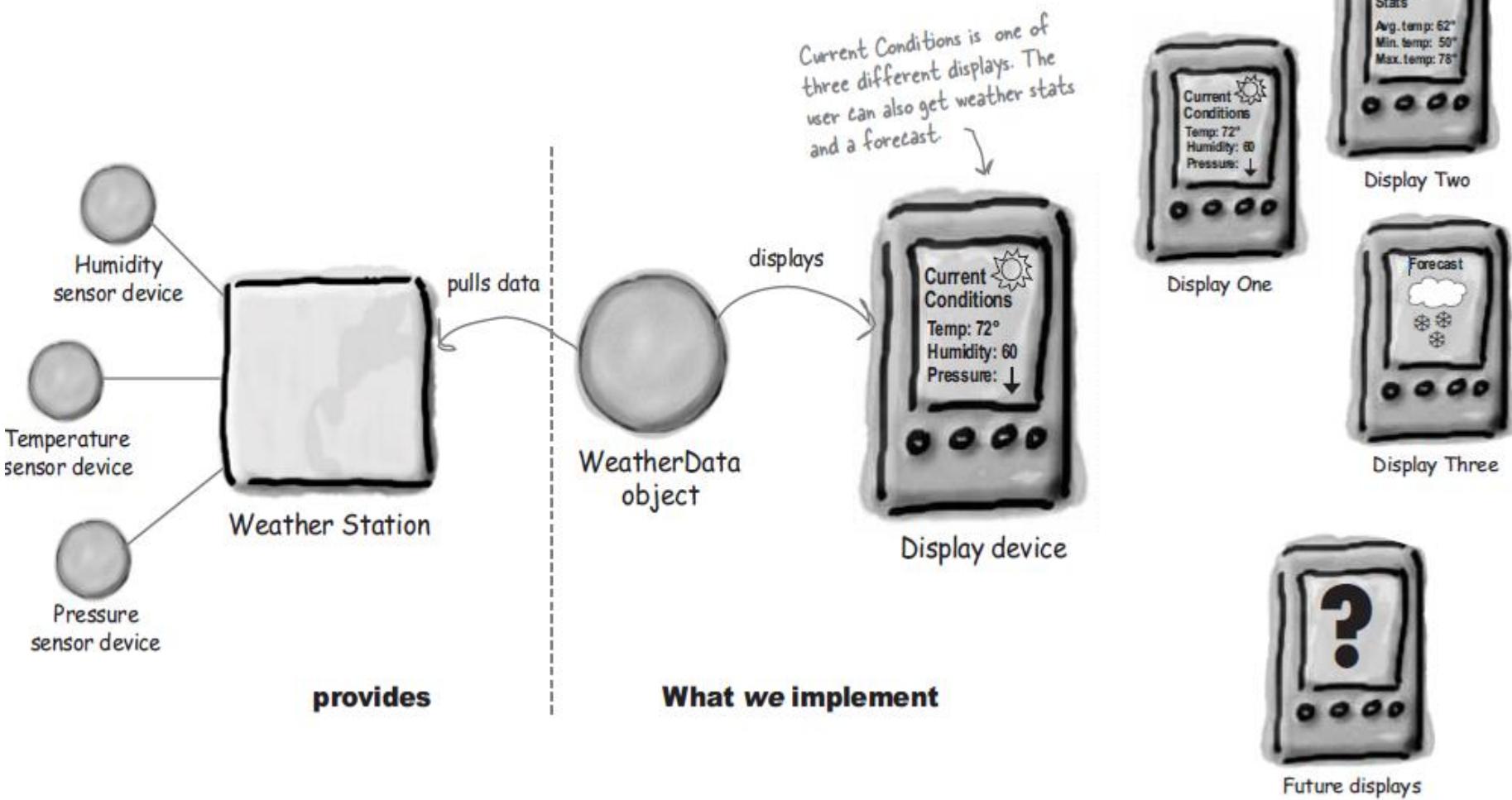


Traverse



Pattern 10

The Weather Monitoring application overview



Observer Pattern

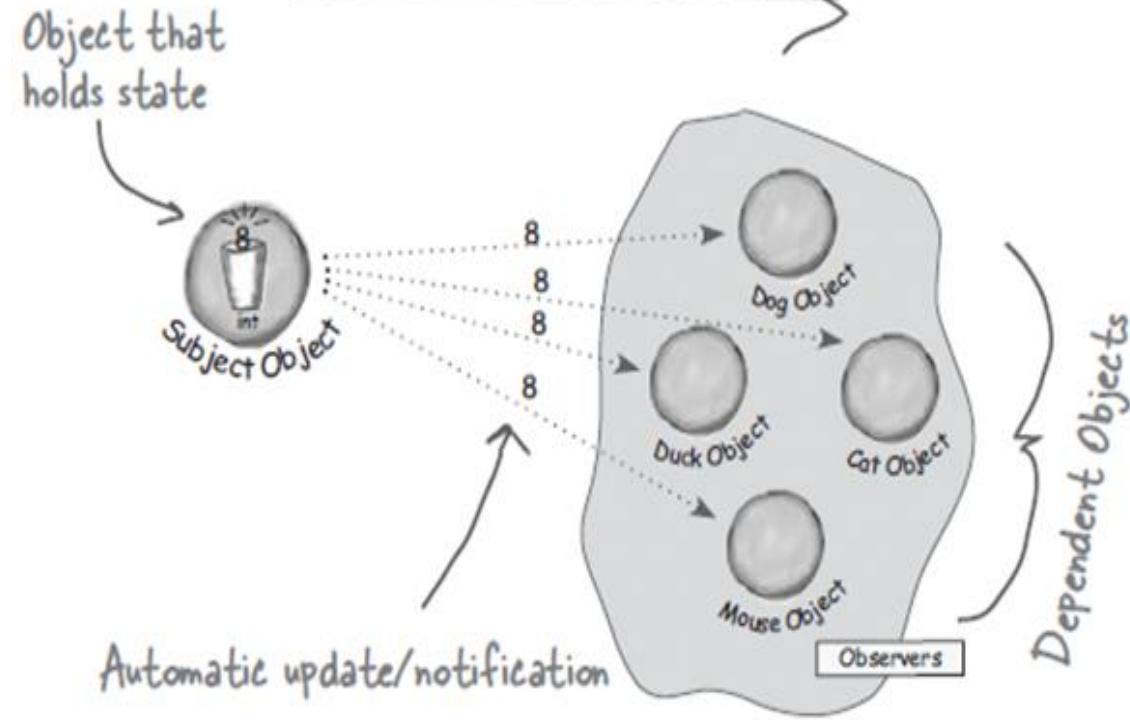
Publishers + Subscribers = Observer Pattern

Like

News Paper

The Observer Pattern defines a one-to-many relationship between a set of objects.

ONE TO MANY RELATIONSHIP



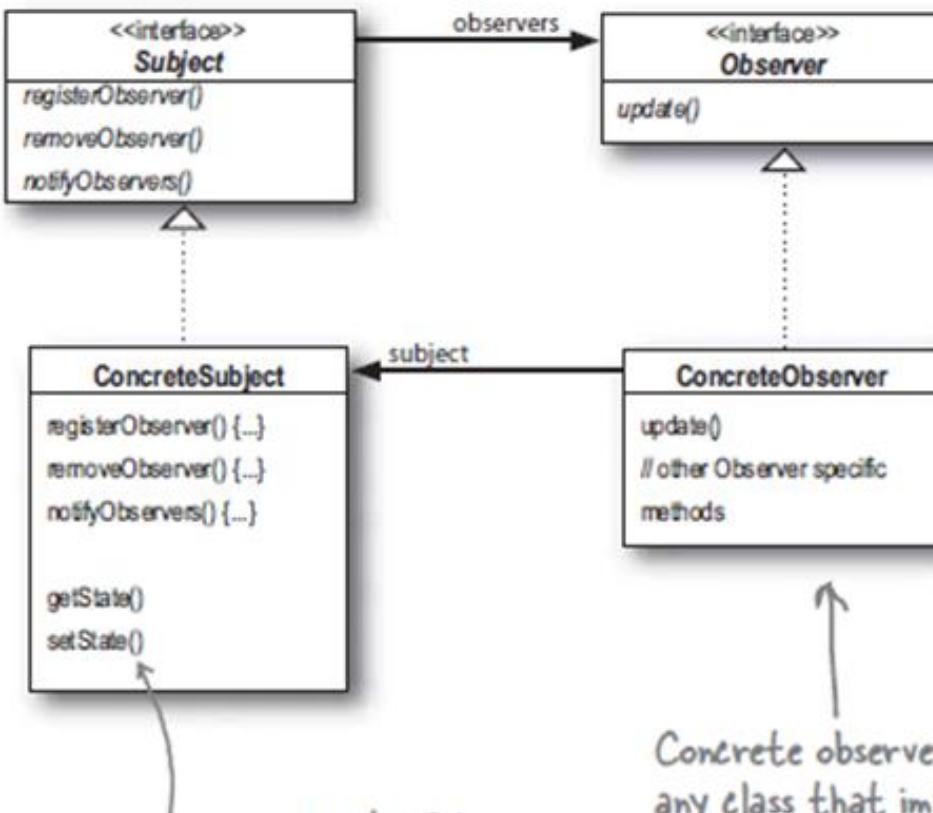
When the state of one object changes, all of its dependents are notified.

Observer Pattern Defined

Here's the Subject interface.
Objects use this interface to register
as observers and also to remove
themselves from being observers.

Each subject
can have many
observers.

All potential observers need
to implement the Observer
interface. This interface
just has one method, update(),
that gets called when the
Subject's state changes.



A concrete subject always
implements the Subject
interface. In addition to
the register and remove
methods, the concrete subject
implements a `notifyObservers()`
method that is used to update
all the current observers
whenever state changes.

The concrete subject may
also have methods for
setting and getting its state
(more about this later).

Concrete observers can be
any class that implements the
Observer interface. Each
observer registers with a concrete
subject to receive updates.

Loose Coupling

The power of Loose Coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

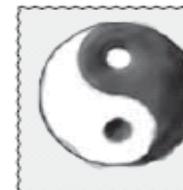
The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).

We can add new observers at any time.

We never need to modify the subject to add new types of observers.

We can reuse subjects or observers independently of each other.

Changes to either the subject or an observer will not affect the other.



Design Principle

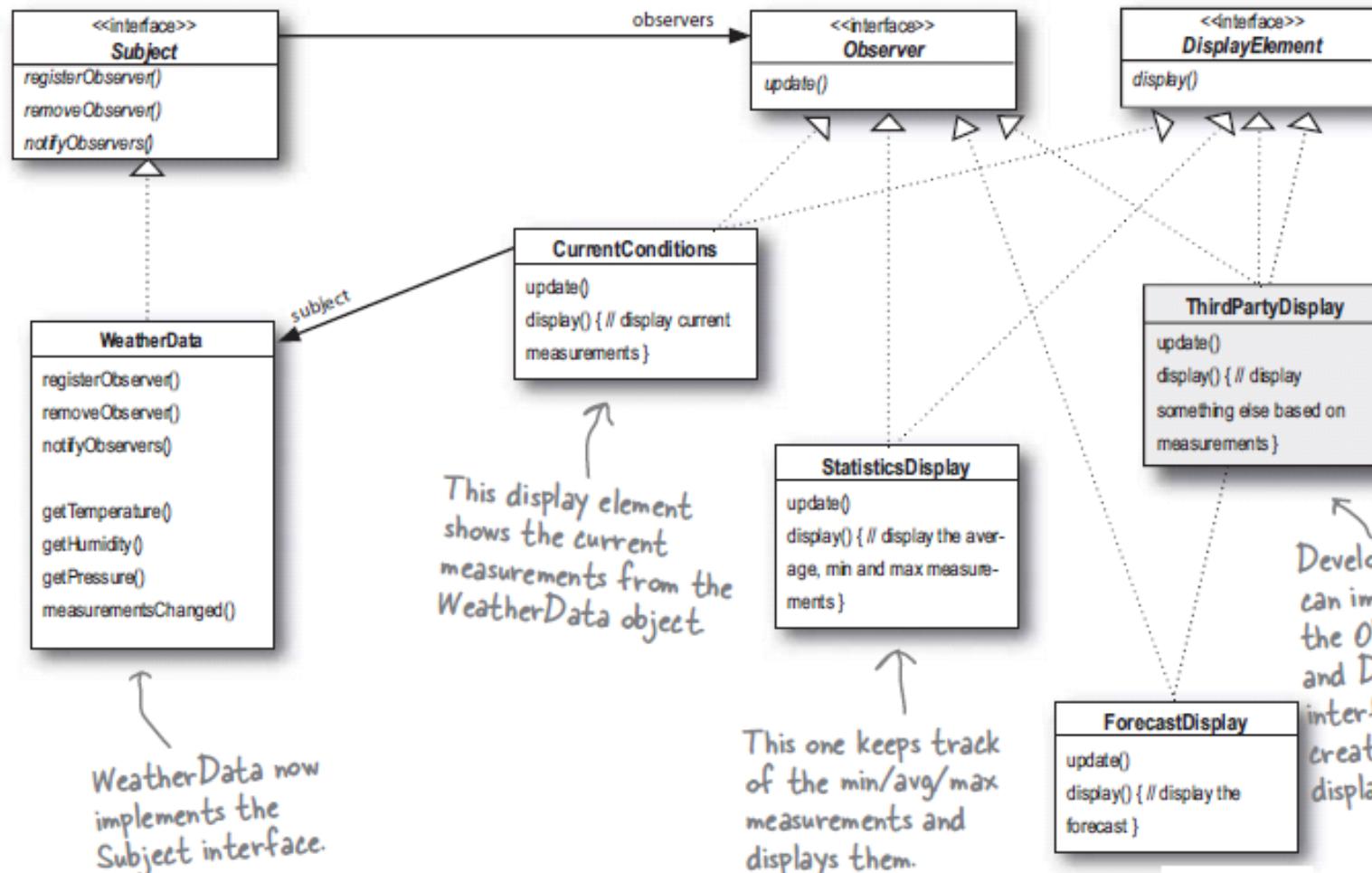
Strive for loosely coupled designs between objects that interact.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Observer Pattern: Implementation

Here's our subject interface, this should look familiar.

All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

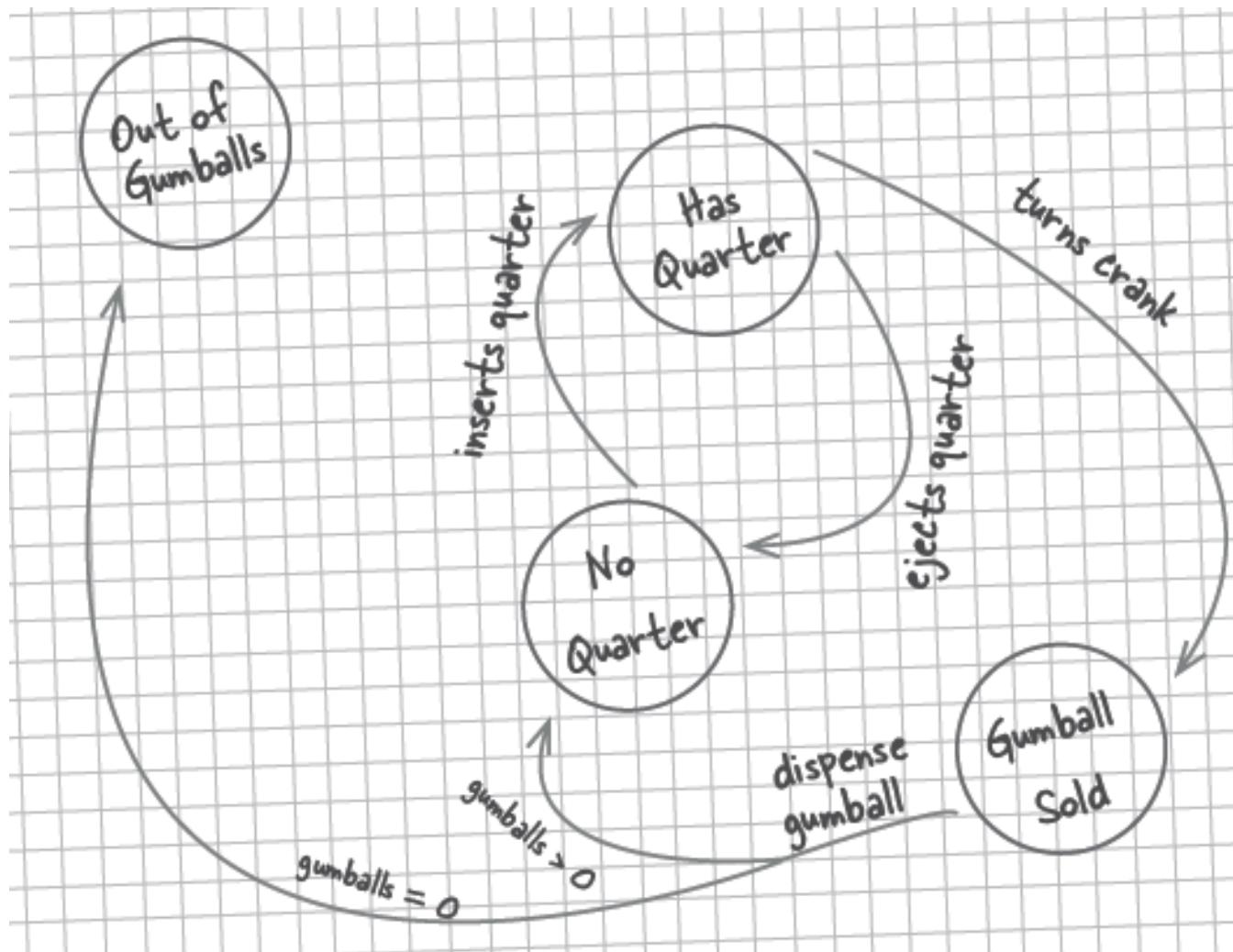


Let's also create an interface for all display elements to implement. The display elements just need to implement a `display()` method.



Developers can implement the Observer and Display interfaces to create their own display element.

Pattern II



Change



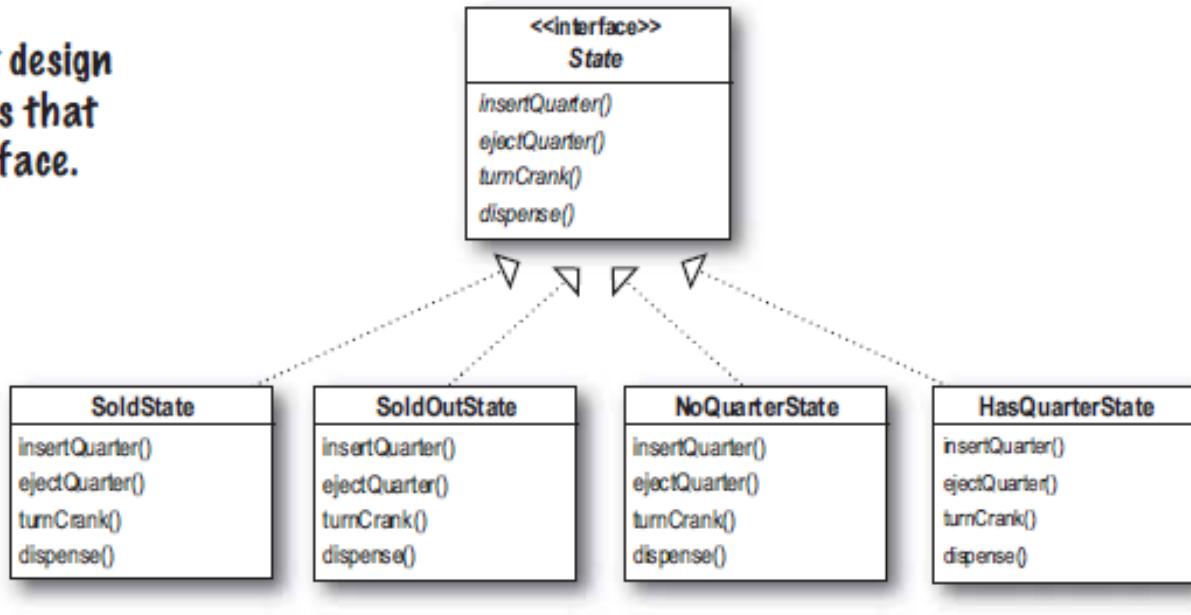
10% of the time,
when the crank
is turned, the
customer gets two
gumballs instead
of one.



What did we do?

Then take each state in our design and encapsulate it in a class that implements the State interface.

To figure out what states we need, we look at our previous code...



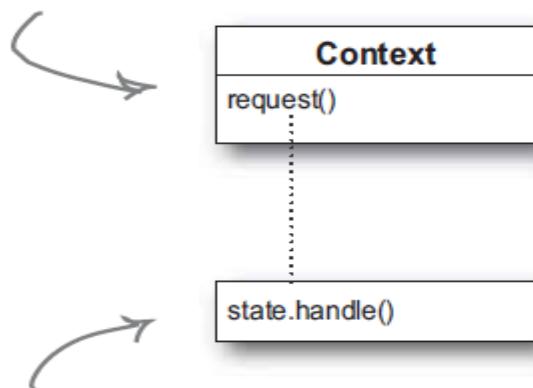
```
public class GumballMachine {  
    final static int SOLD_OUT = 0;  
    final static int NO_QUARTER = 1;  
    final static int HAS_QUARTER = 2;  
    final static int SOLD = 3;  
  
    int state = SOLD_OUT;  
    int count = 0;
```

... and we map each state directly to a class.

State Pattern

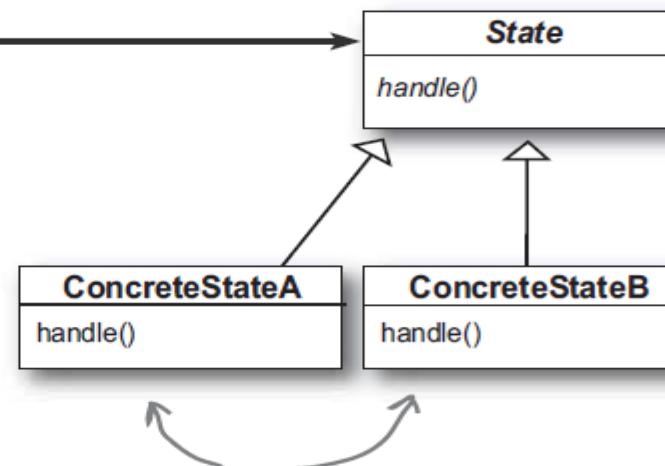
The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.



Whenever the `request()` is made on the Context it is delegated to the state to handle.

The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.

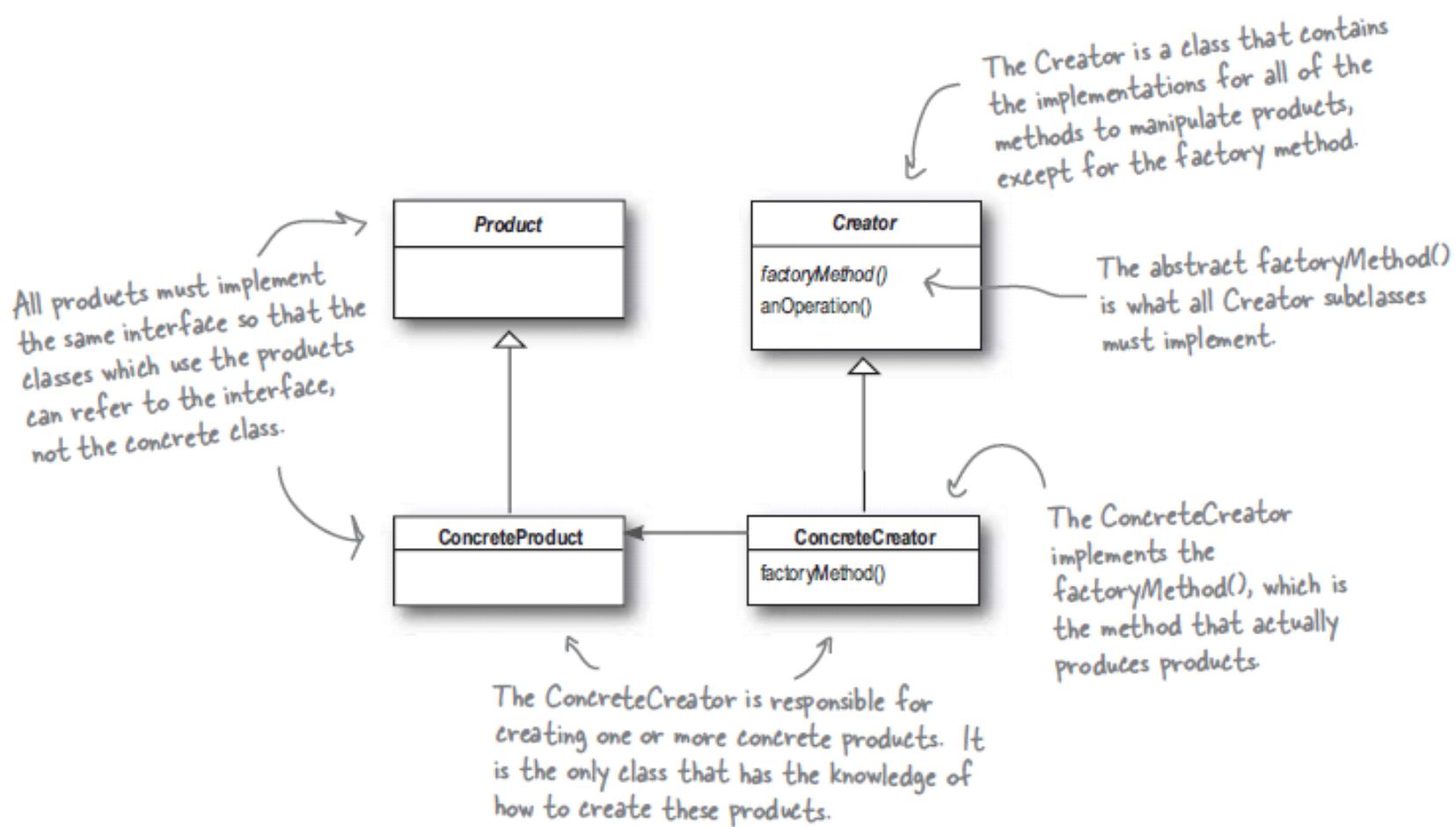


ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

Many concrete states are possible.

Pattern 12: 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.



Pattern 13

Welcome to Starbuzz Coffee

Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.

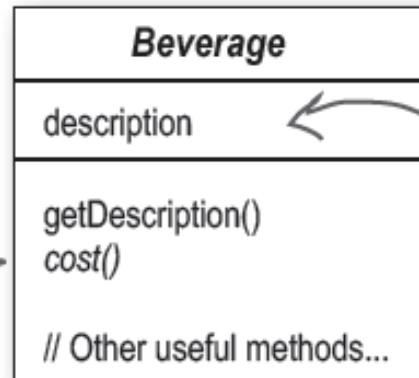
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.



Initial Design

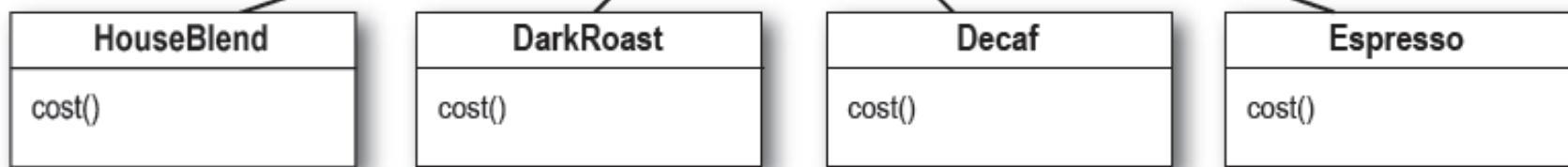
Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The cost() method is abstract; subclasses need to define their own implementation.



The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The get>Description() method returns the description.



Each subclass implements cost() to return the cost of the beverage.

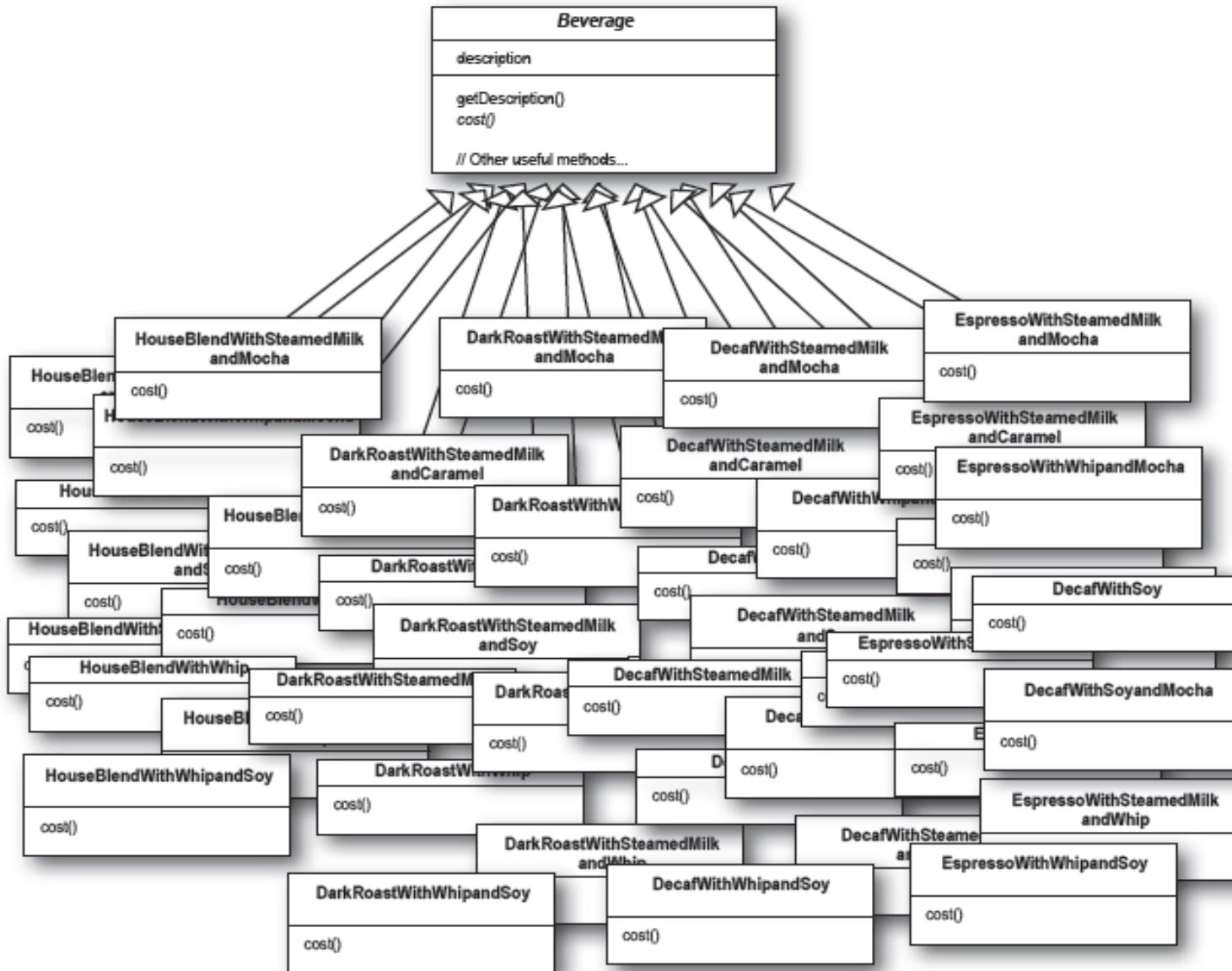
CHANGE – Add Condiments

Mocha Chocolate Chip Frappuccino®



Coffee with rich mocha-flavoured sauce blended with milk, chocolaty chips and ice. Topped with sweetened whipped cream and chocolate-flavoured drizzle.

Design?

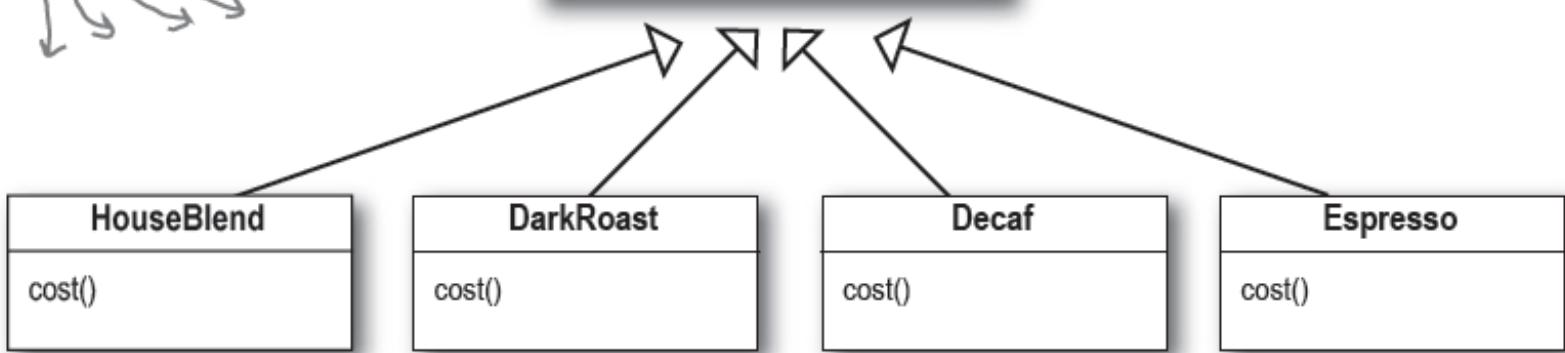
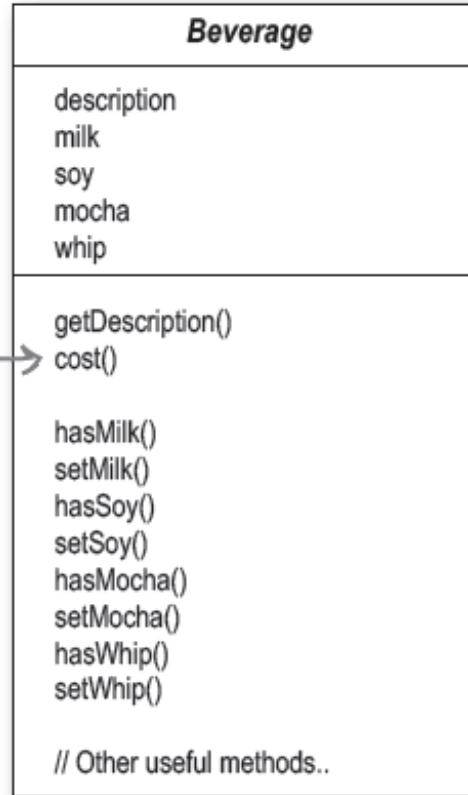


How about this?

Now let's add in the subclasses, one for each beverage on the menu:

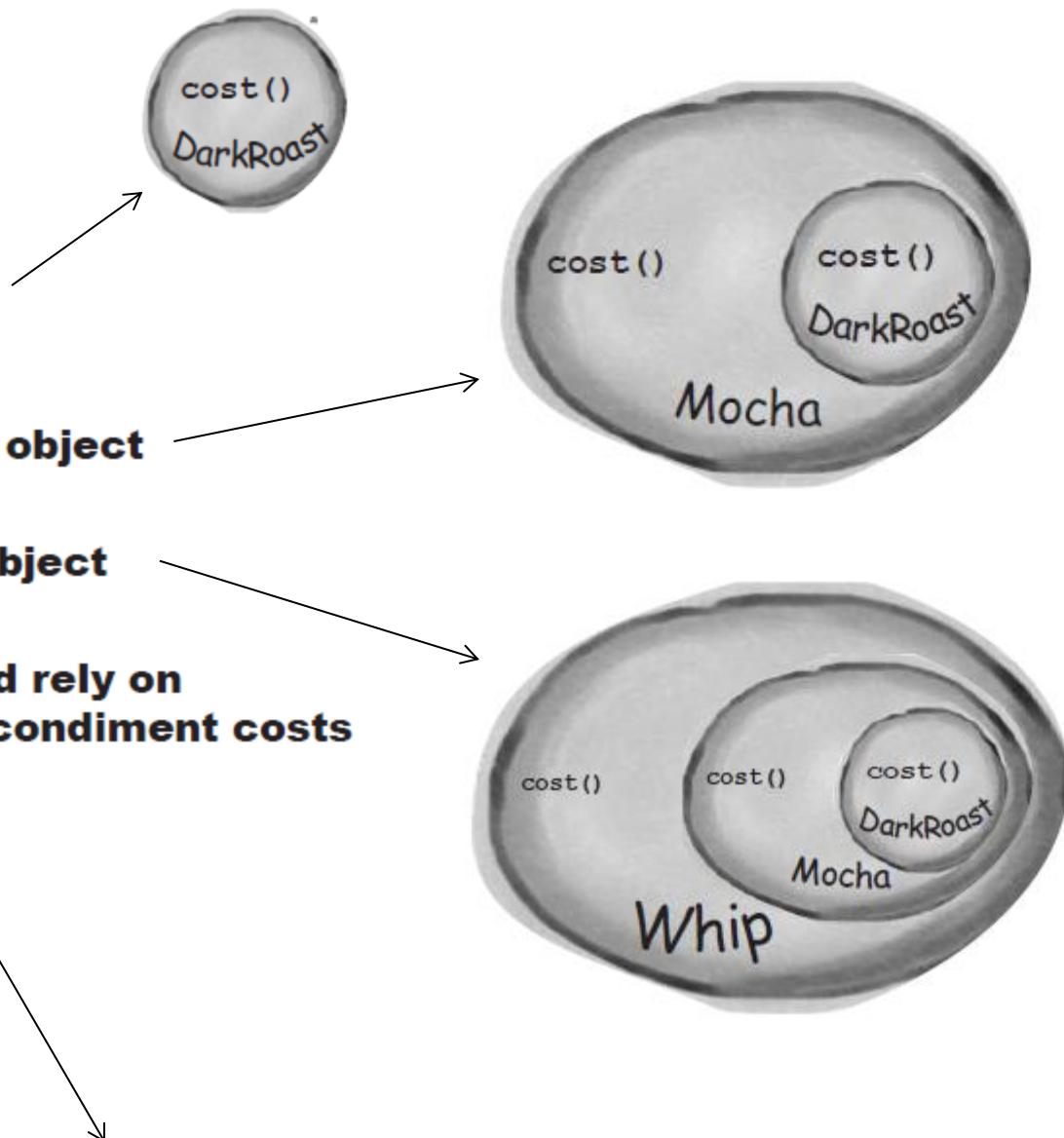
The superclass `cost()` will calculate the costs for all of the condiments, while the overridden `cost()` in the subclasses will extend that functionality to include costs for that specific beverage type.

Each `cost()` method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of `cost()`.



Decoration

- ① Take a **DarkRoast** object
- ② Decorate it with a **Mocha** object
- ③ Decorate it with a **Whip** object
- ④ Call the **cost()** method and rely on delegation to add on the condiment costs



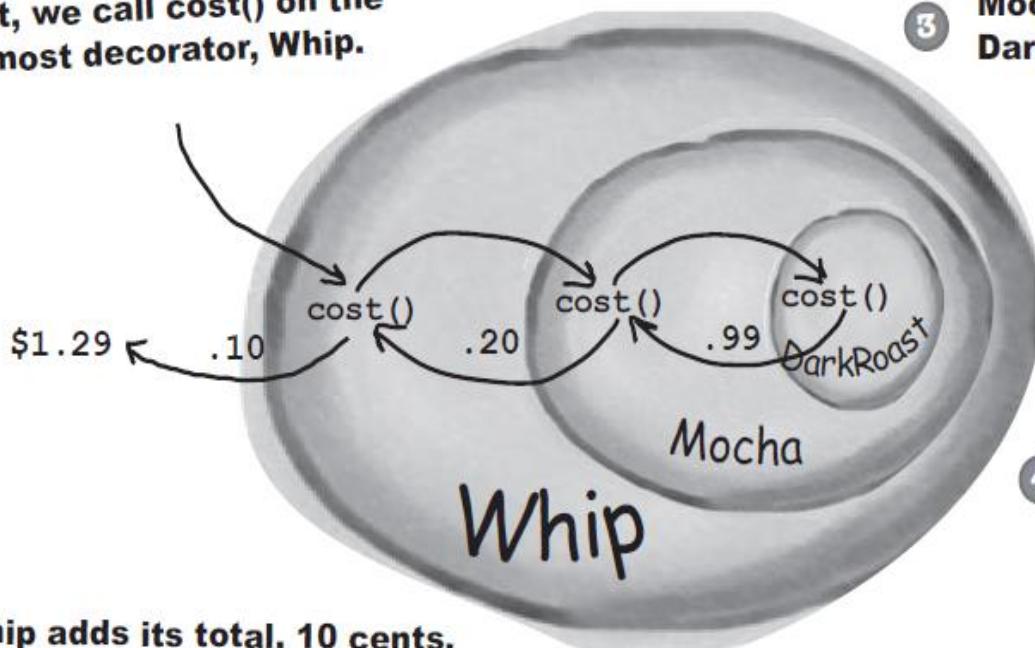
Decorator Used

- ❶ Take a `DarkRoast` object
- ❷ Decorate it with a `Mocha` object
- ❸ Decorate it with a `Whip` object
- ❹ Call the `cost()` method and rely on delegation to add on the condiment costs

❷ `Whip` calls `cost()` on `Mocha`.

❶ First, we call `cost()` on the outmost decorator, `Whip`.

❸ `Mocha` calls `cost()` on `DarkRoast`.



❸ `Whip` adds its total, 10 cents, to the result from `Mocha`, and returns the final result—\$1.29.

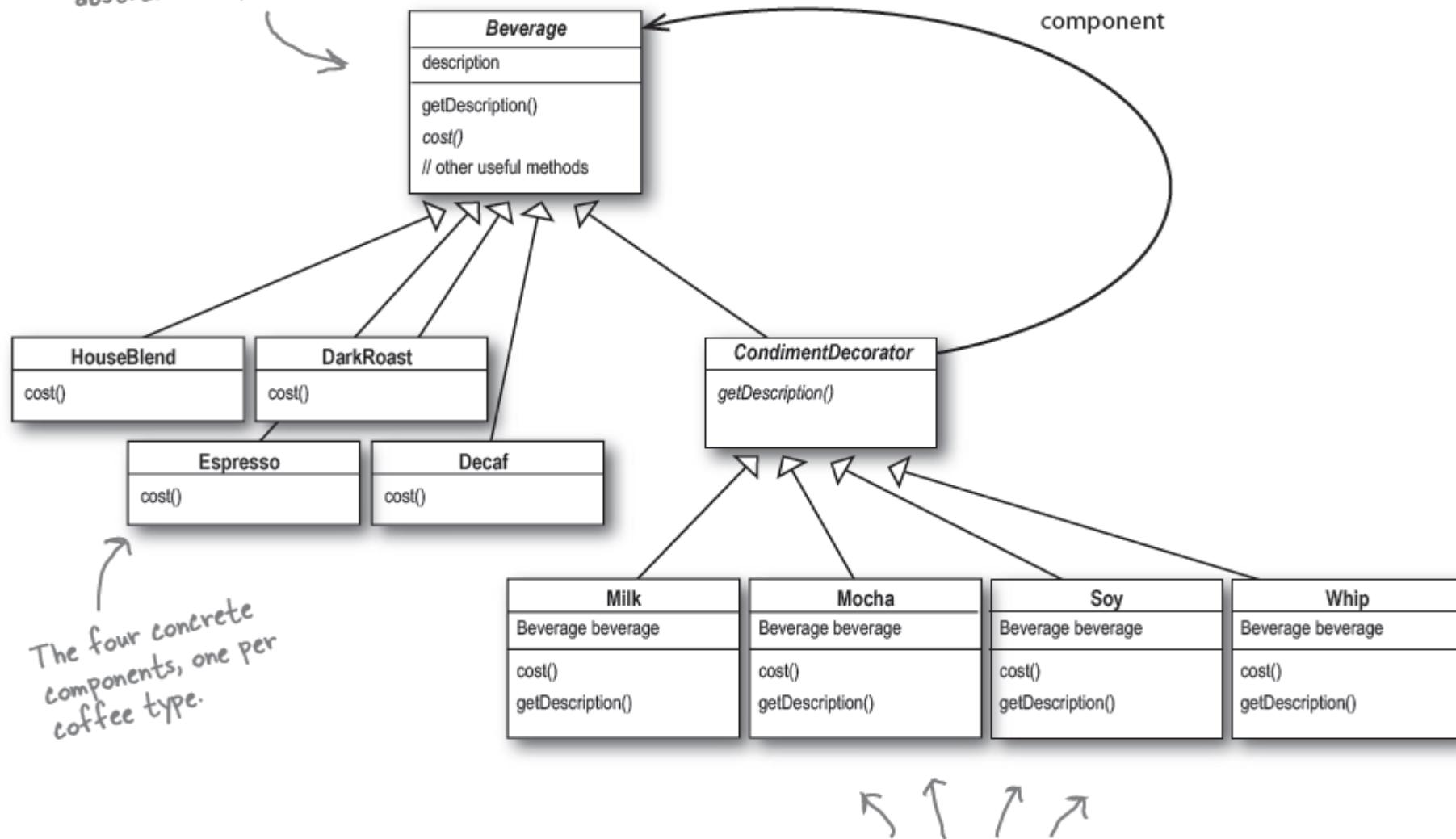
❹ `Mocha` adds its cost, 20 cents, to the result from `DarkRoast`, and returns the new total, \$1.19.

❺ `DarkRoast` returns its cost, 99 cents.

Starbuzz Coffee		
<u>Coffees</u>		
House Blend	.89	
Dark Roast	.99	
Decaf	1.05	
Espresso	1.99	
<u>Condiments</u>		
Steamed Milk	.10	
Mocha	.20	
Soy	.15	
Whip	.10	

Apply to Star Buzz

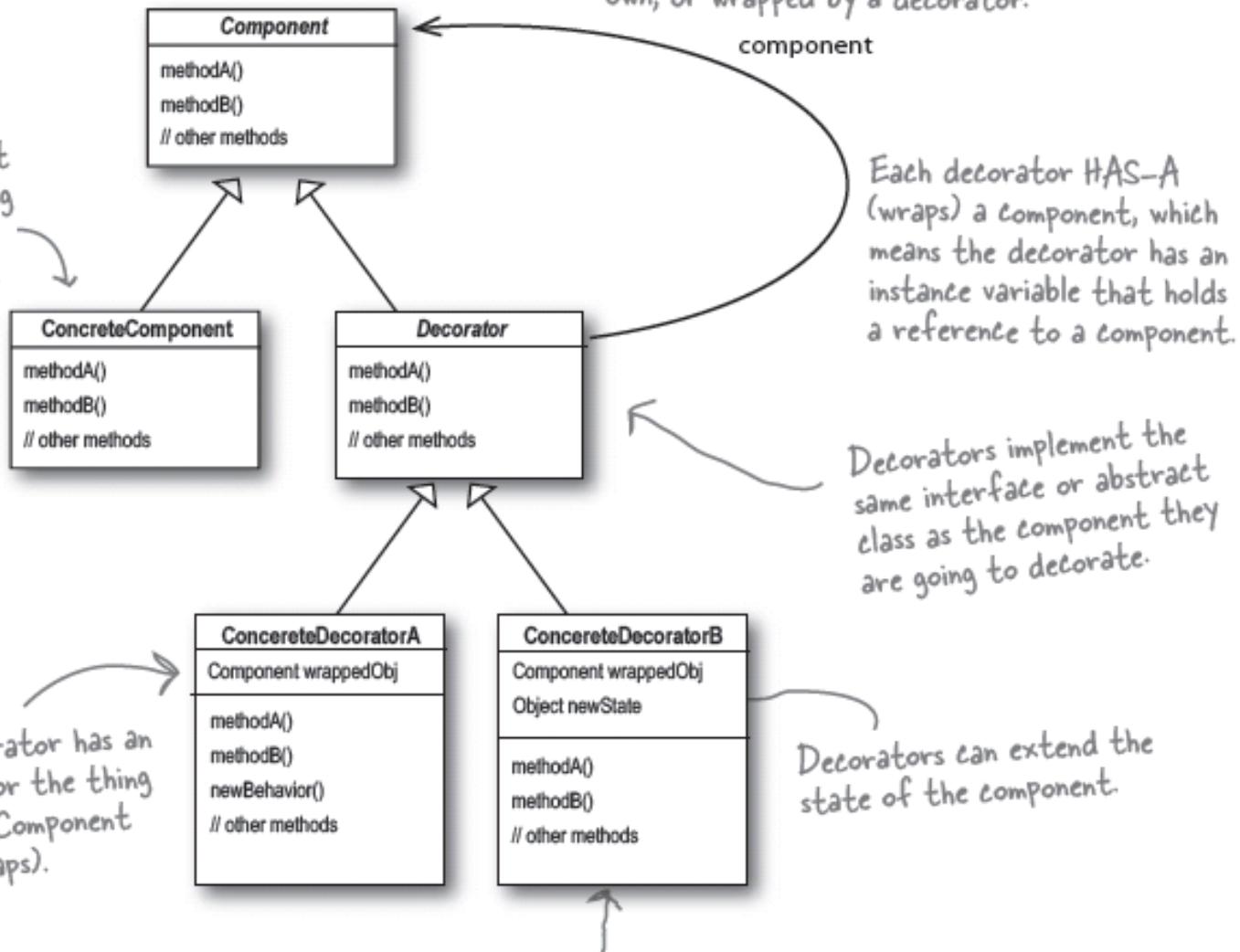
Beverage acts as our abstract component class.



And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

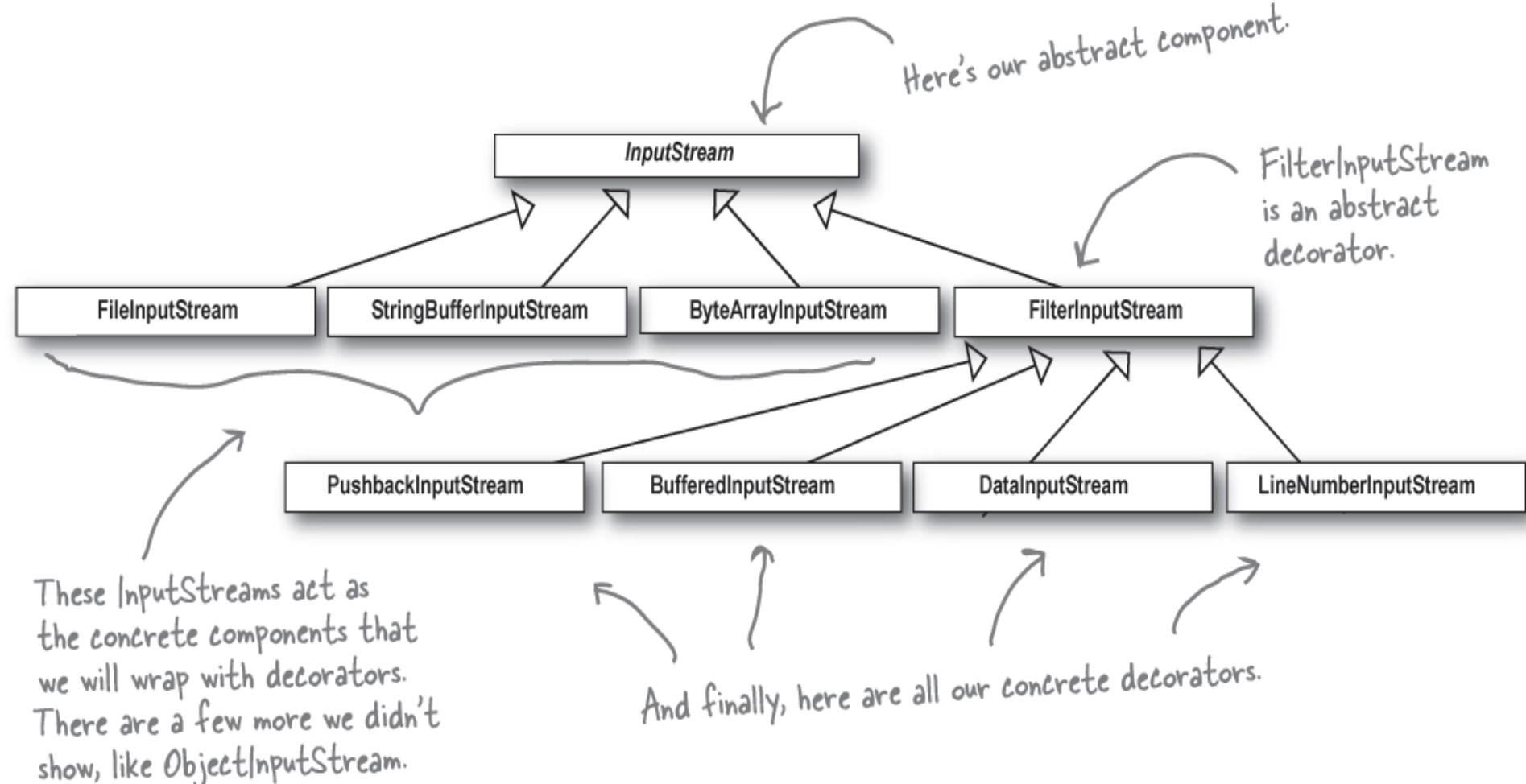
Decorator Defined

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

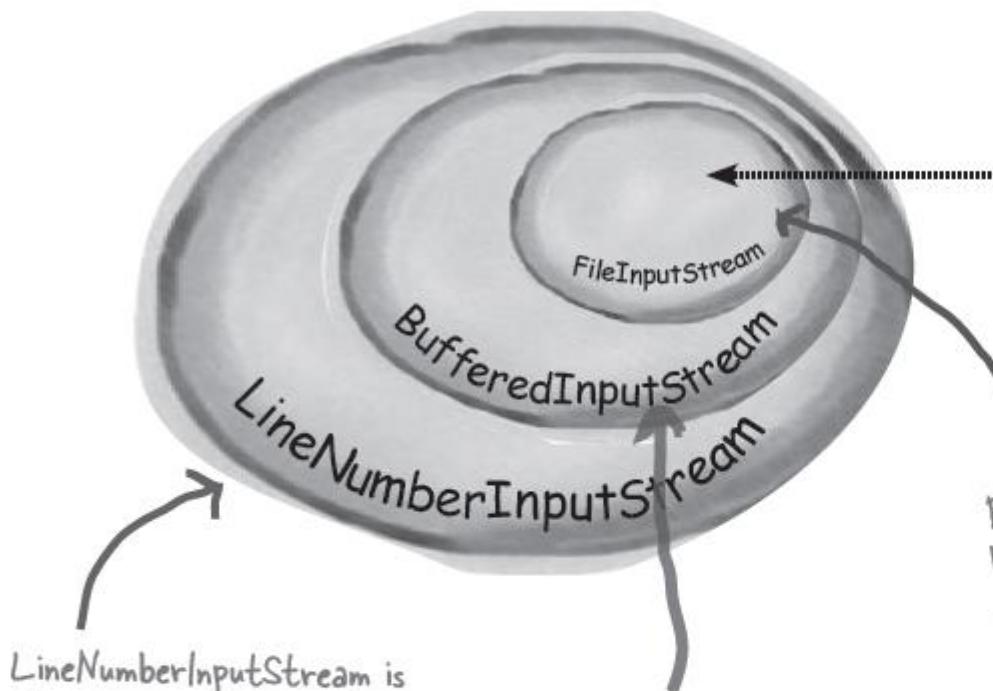


Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

File I/O Decorators in Java



File I/O Decorators



LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method `readLine()` for reading character-based input, a line at a time.

FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

Pattern of Patterns

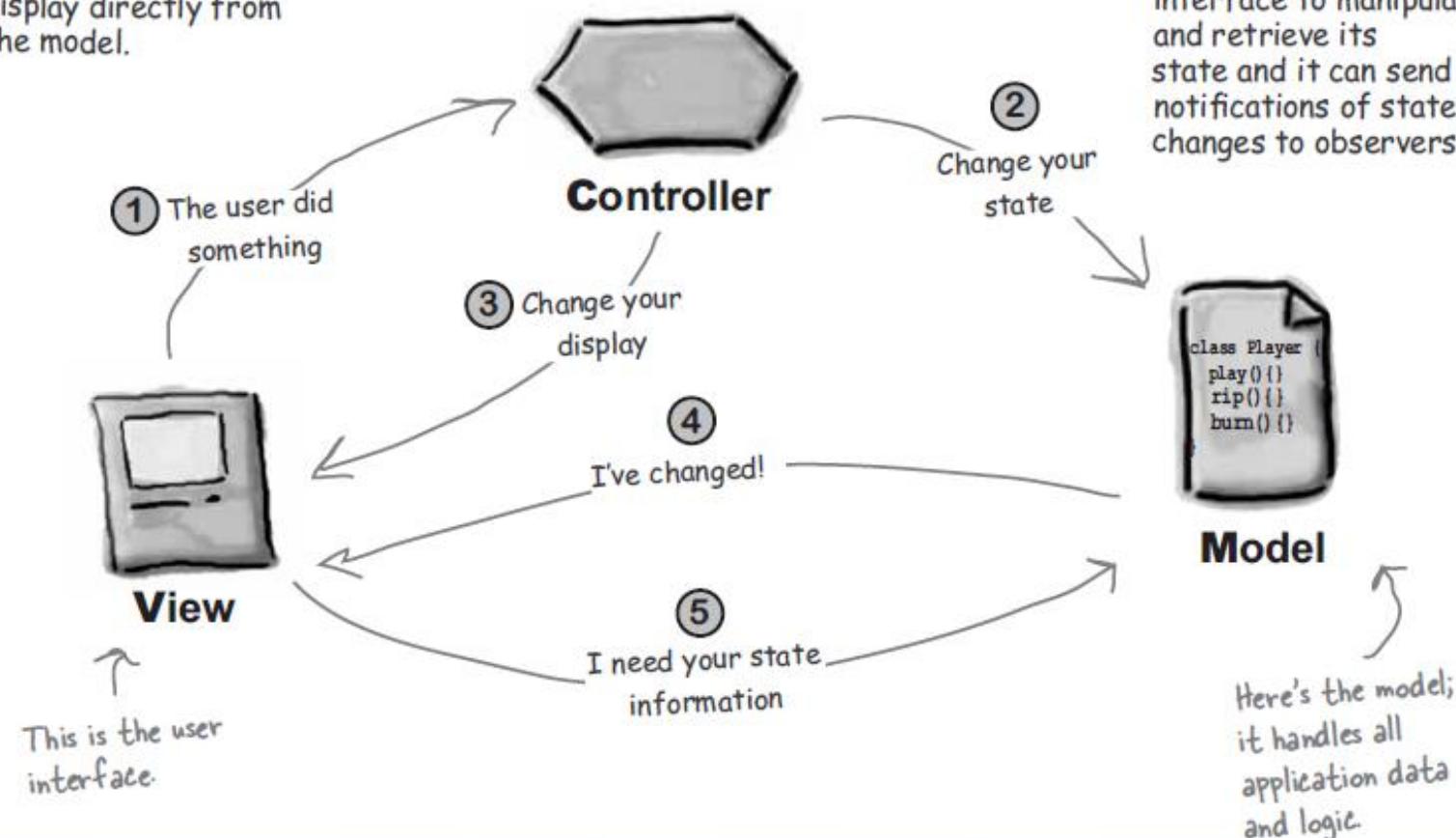
CONTROLLER

Takes user input and figures out what it means to the model.

VIEW

Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.

Here's the creamy controller; it lives in the middle. ↗



MODEL

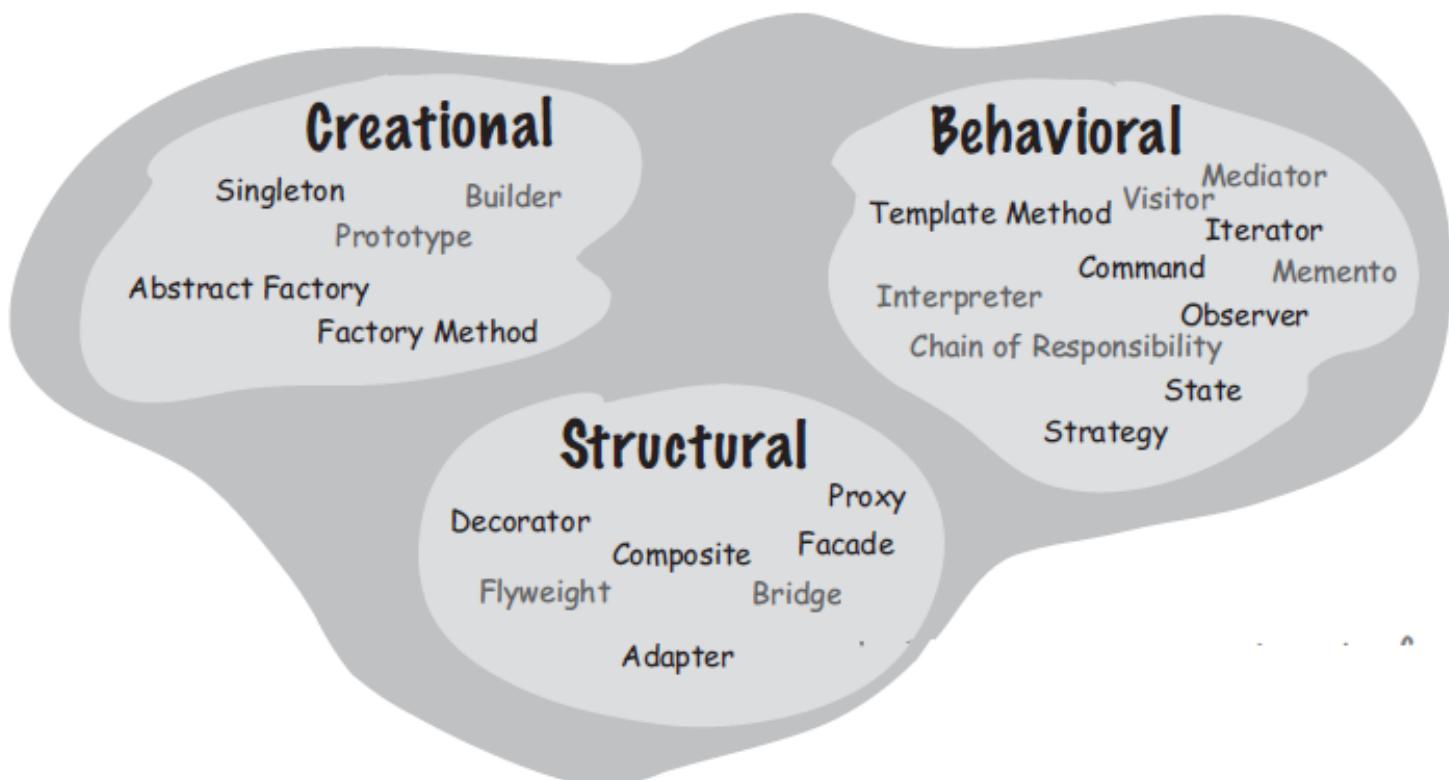
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.

Summary...

Design Pattern Categories

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



Structural patterns let you compose classes or objects into larger structures.

Class and Object Patterns

Class patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Class

Template Method
Factory Method Adapter
Interpreter

Object

Composite
Decorator
Proxy
Strategy
Bridge
Flyweight
Abstract Factory
Singleton
Visitor
Command
Facade
Chain of Responsibility
Mediator
Prototype
Builder
State
Iterator
Memento
Observer

Object patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.

Quiz

Match each pattern with its description:

Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.

Your mind on Patterns

"I need a pattern for Hello World."

BEGINNER MIND

As learning progresses, the Intermediate mind starts to see where patterns are needed and where they aren't. The intermediate mind still tries to fit too many square patterns into round holes, but also begins to see that patterns can be adapted to



ZEN MIND

"This is a natural place for Decorator."

The Beginner uses patterns everywhere. This is good: the beginner gets lots of experience with and practice using patterns. The beginner also thinks, "The more patterns I use, the better the design." The beginner will learn this is not so, that all designs should be as simple as possible. Complexity and patterns should only be used where they are needed for practical extensibility.

INTERMEDIATE MIND

"Maybe I need a Singleton here."

The Zen mind is able to see patterns where they fit naturally. The Zen mind is not obsessed with using patterns; rather it looks for simple solutions that best solve the problem. The Zen mind thinks in terms of the object principles and their trade-offs. When a need for a pattern naturally arises, the Zen mind applies it knowing well that it may require adaptation. The Zen mind also sees relationships to similar patterns and understands the subtleties of differences in the intent of related patterns. *The Zen mind is also a Beginner mind* — it doesn't let all that pattern knowledge overly influence design decisions.

UML

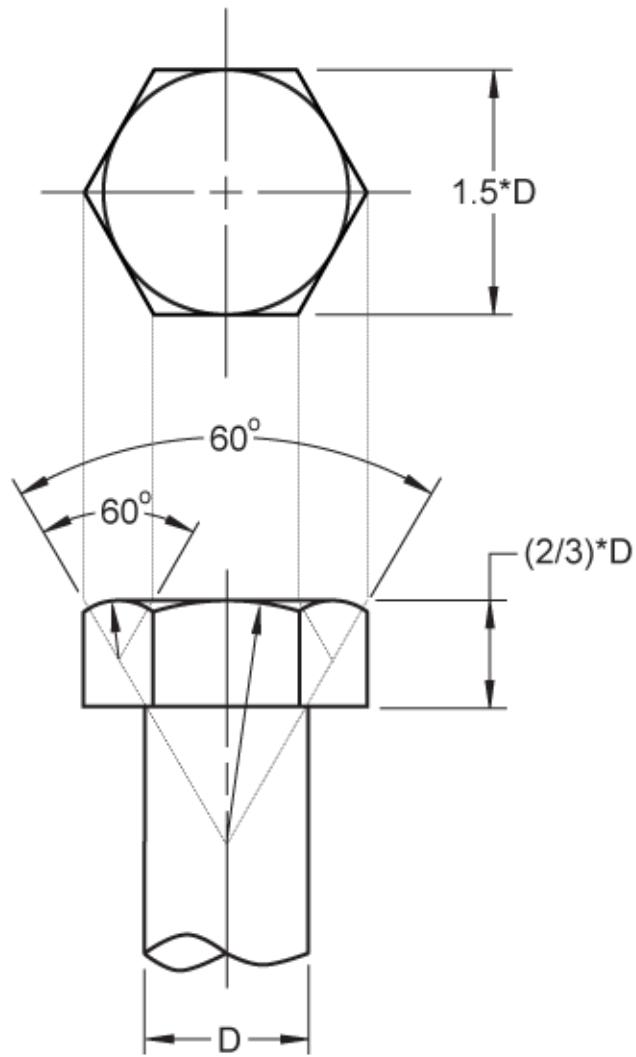
With StarUml

Credits:

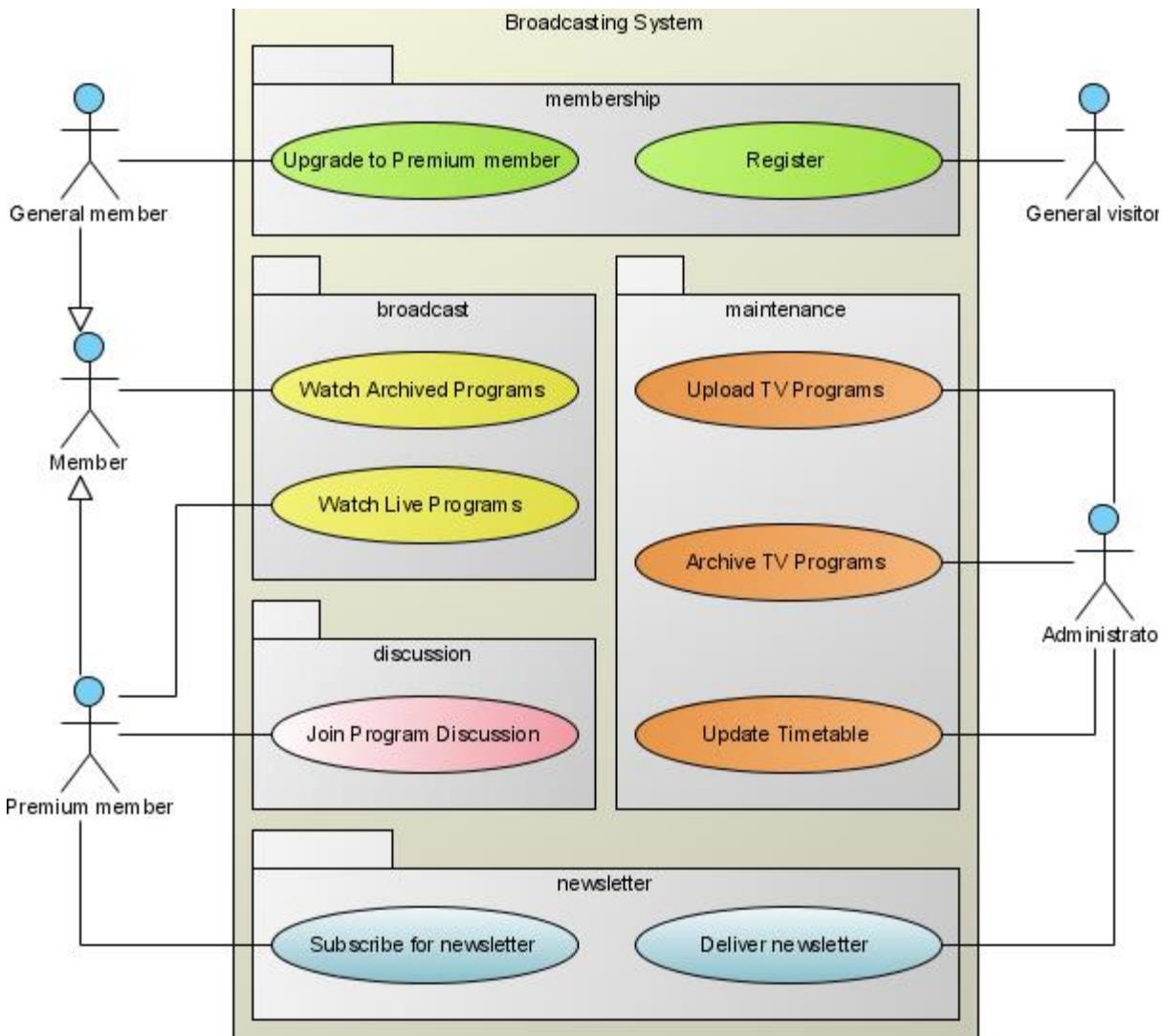
<http://www.uml-diagrams.org>

<http://www.visual-paradigm.com/support/documents>

The need for UML



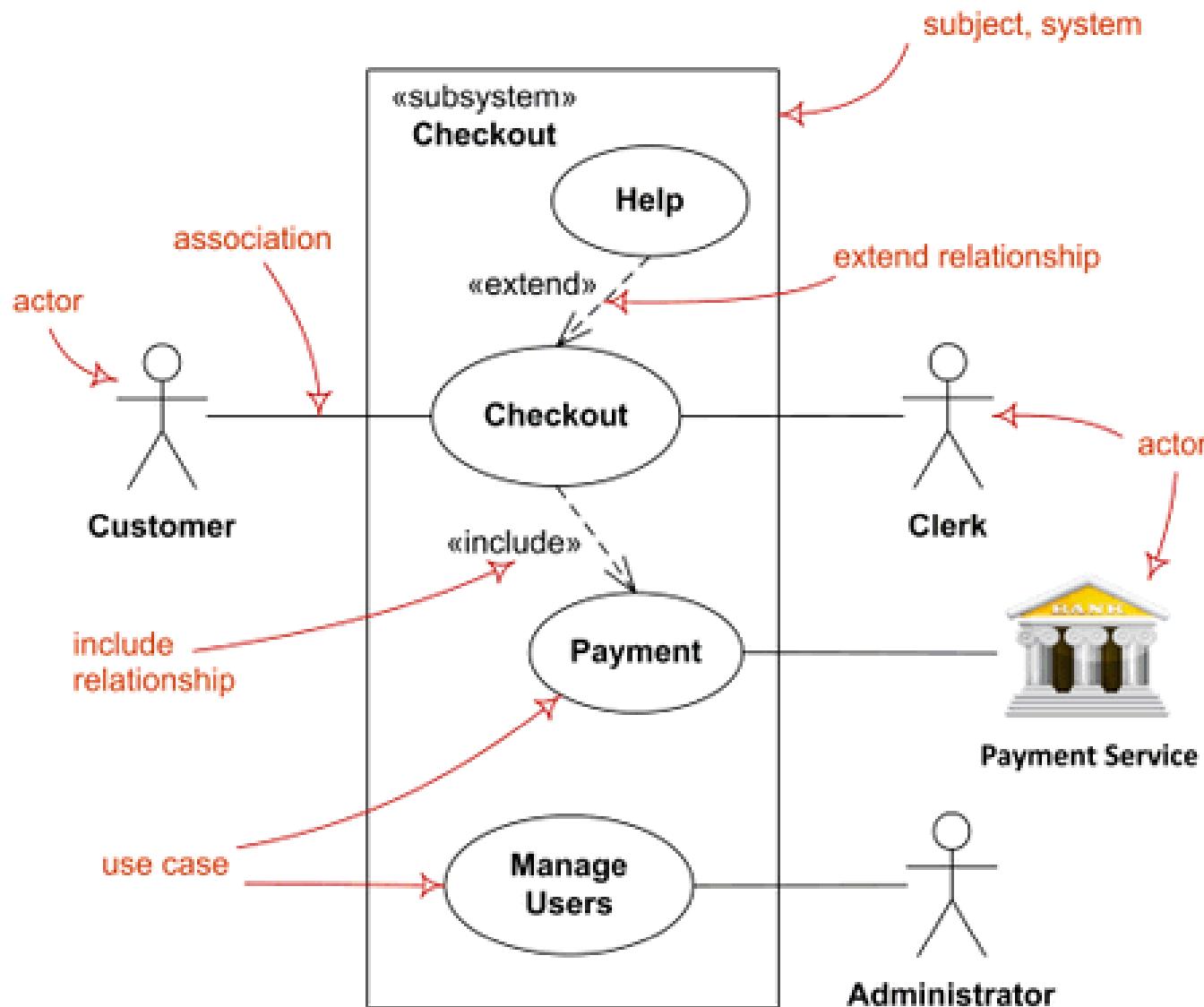
Use Case Diagram



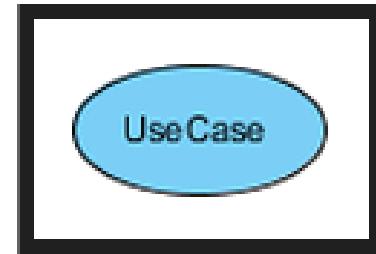
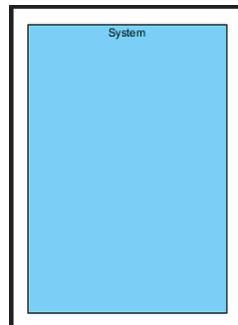
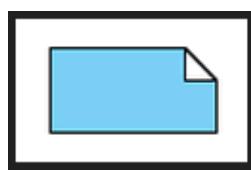
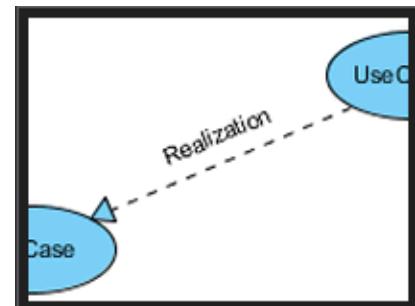
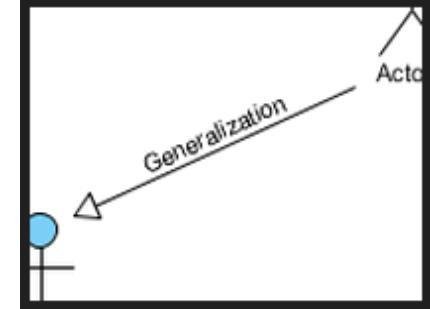
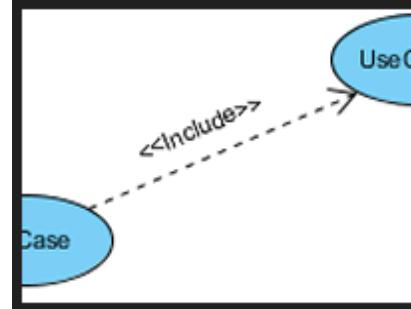
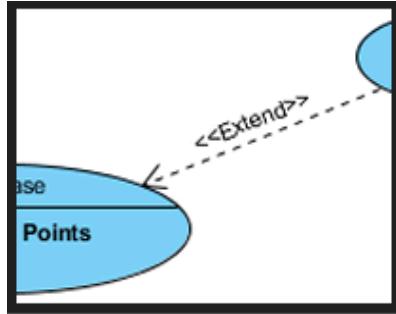
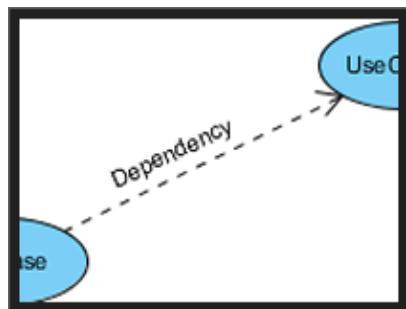
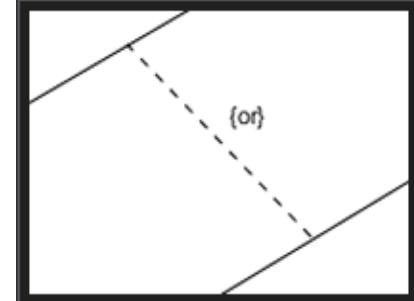
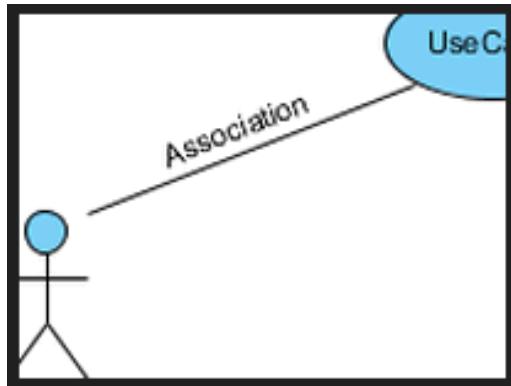
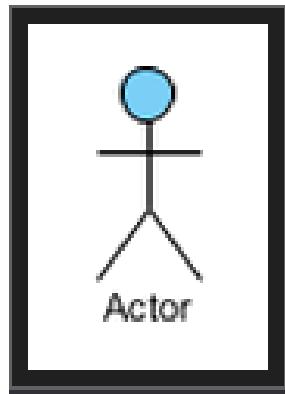
Notation

	Actor		Association
	Collaboration		Constraint
	Dependency		Extend
	Generalization		Include
	Note		Realization
	System		Use Case

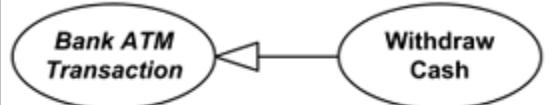
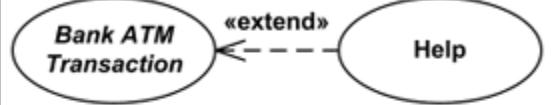
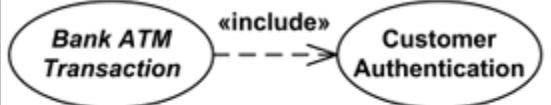
Use Case Diagram Reference

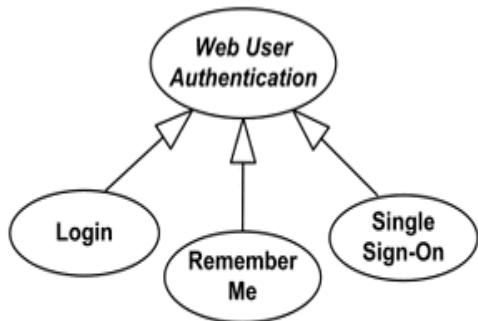


Use Case Diagram (Notations)

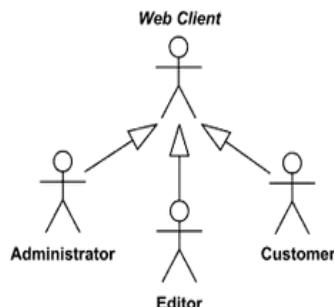


Use Case Diagram (Relationships)

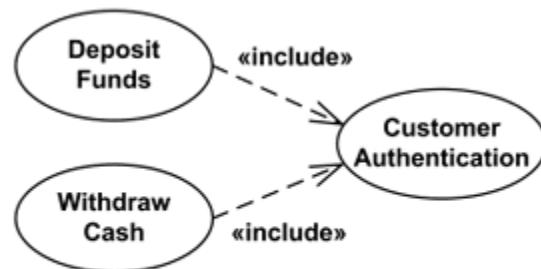
Generalization	Extend	Include
		
Base use case could be abstract use case (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (abstract use case).
Specialized use case is required, not optional, if base use case is abstract.	Extending use case is optional, supplementary.	Included use case required, not optional.
No explicit location to use specialization.	Has at least one explicit extension location.	No explicit inclusion location but is included at some location.
No explicit condition to use specialization.	Could have optional extension condition.	No explicit inclusion condition.



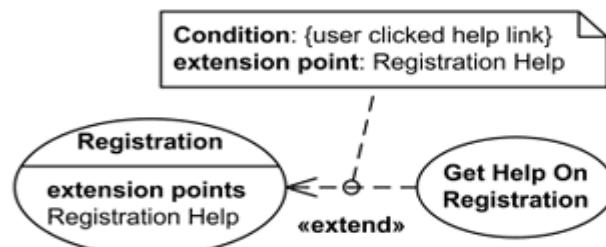
Web User Authentication use case is **abstract use case** specialized by Login, Remember Me and Single Sign-On use cases.



Web Client actor is abstract superclass for Administrator, Editor and Customer

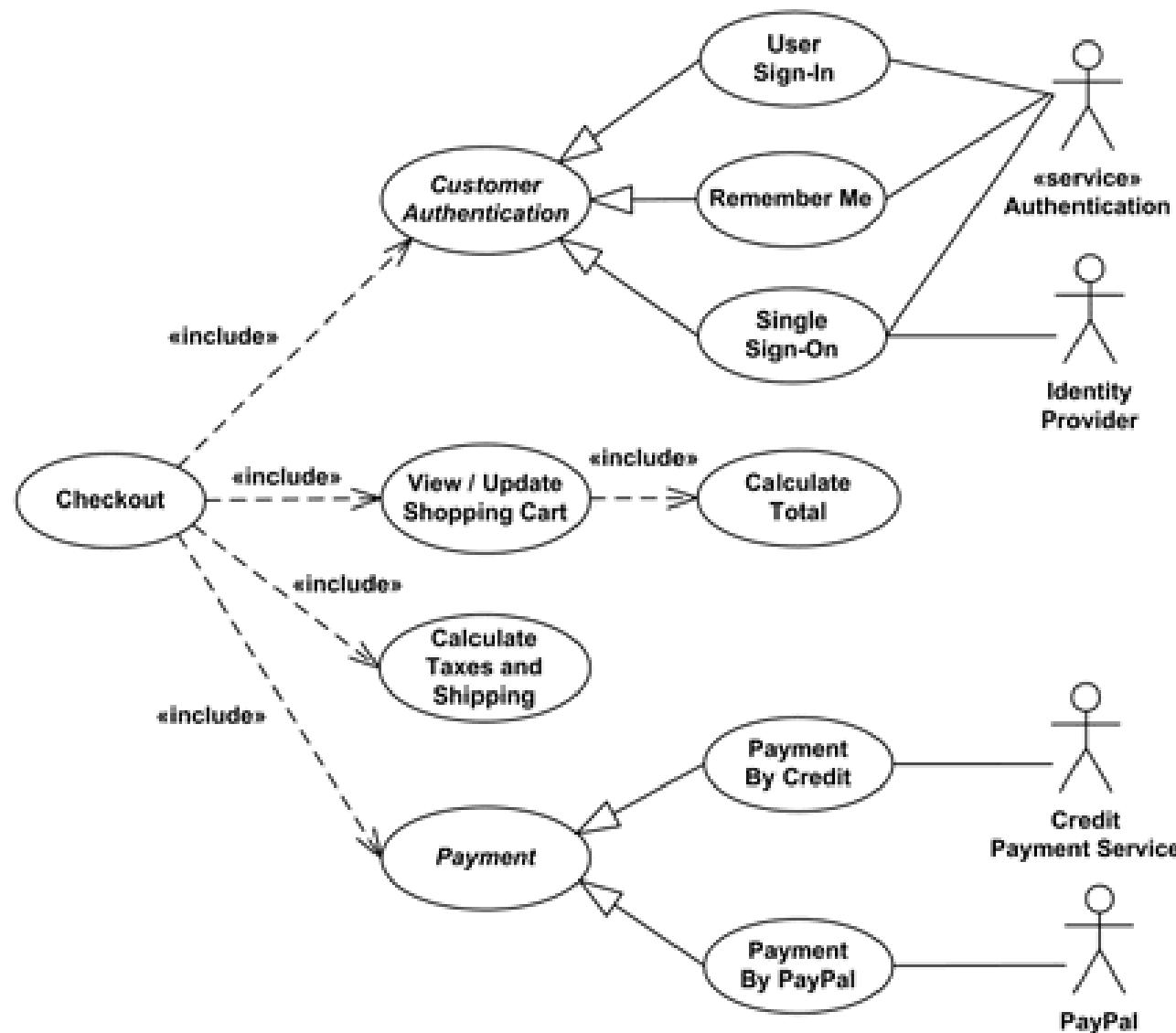


Both **Deposit Funds** and **Withdraw Cash** use cases require (**include**) **Customer Authentication** use case.

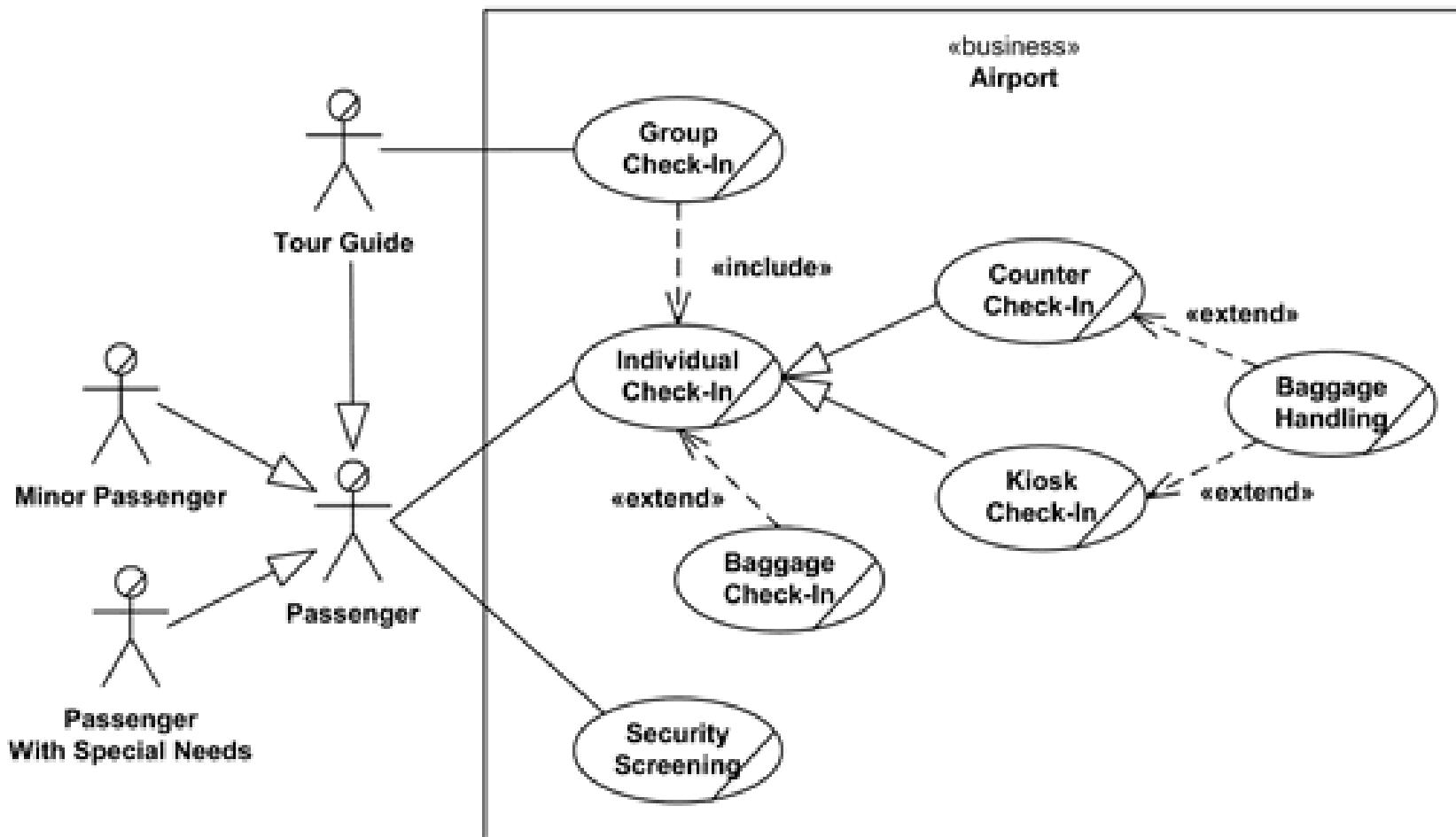


Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

Online Shopping Checkout

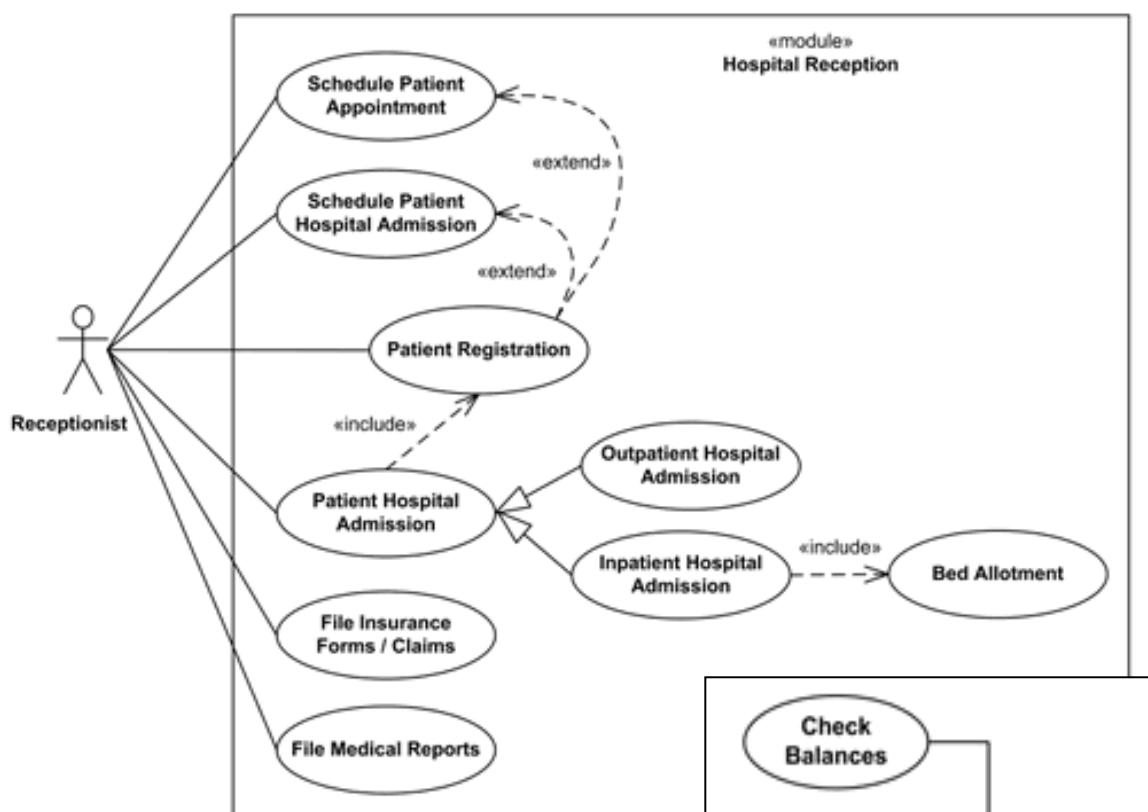


Airport Check-in

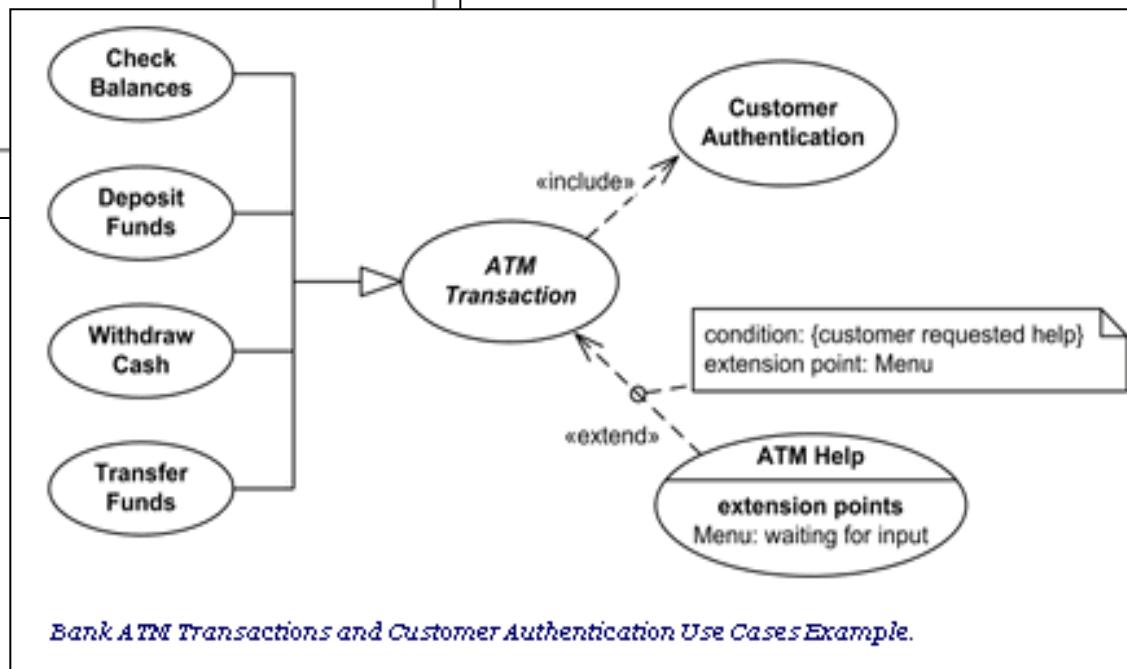


An example of use case diagram for airport check-in and security screening

More Examples



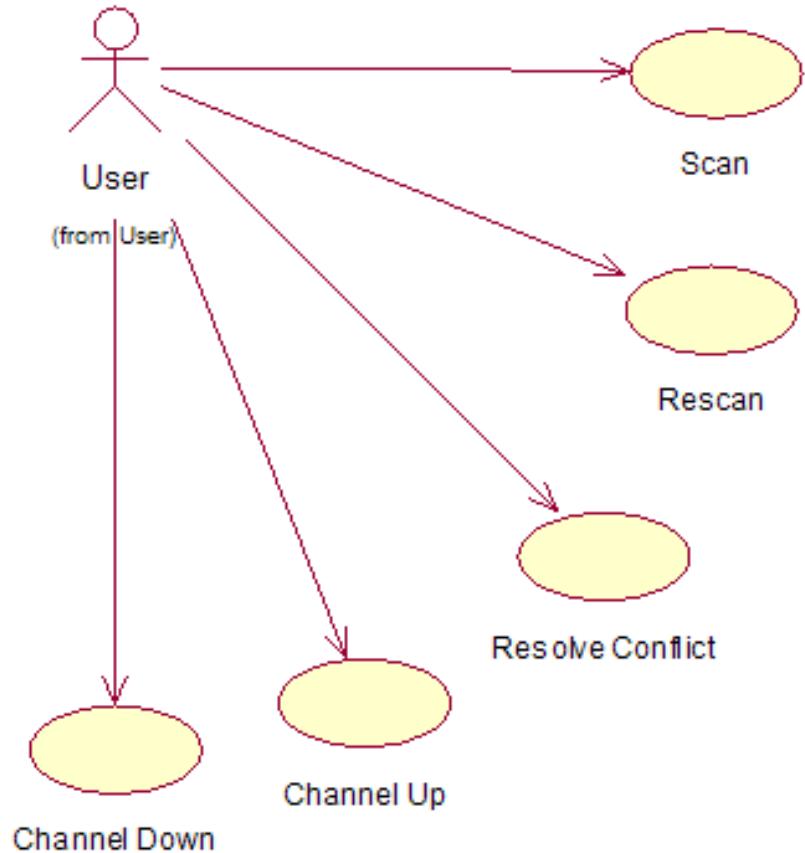
An example of use case diagram for Hospital Reception.



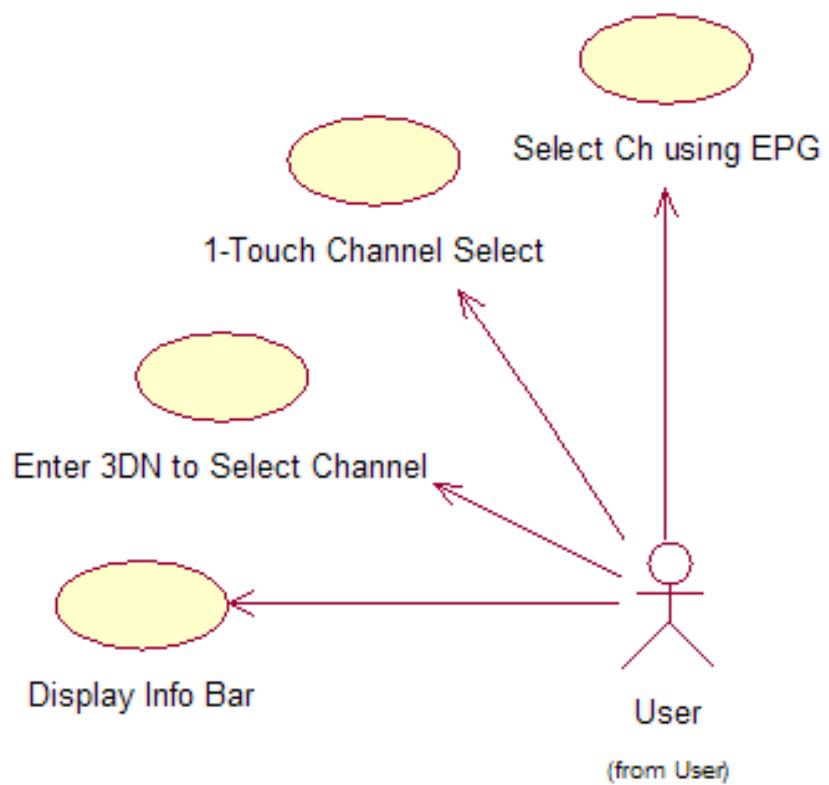
Bank ATM Transactions and Customer Authentication Use Cases Example.

User Accessing TV

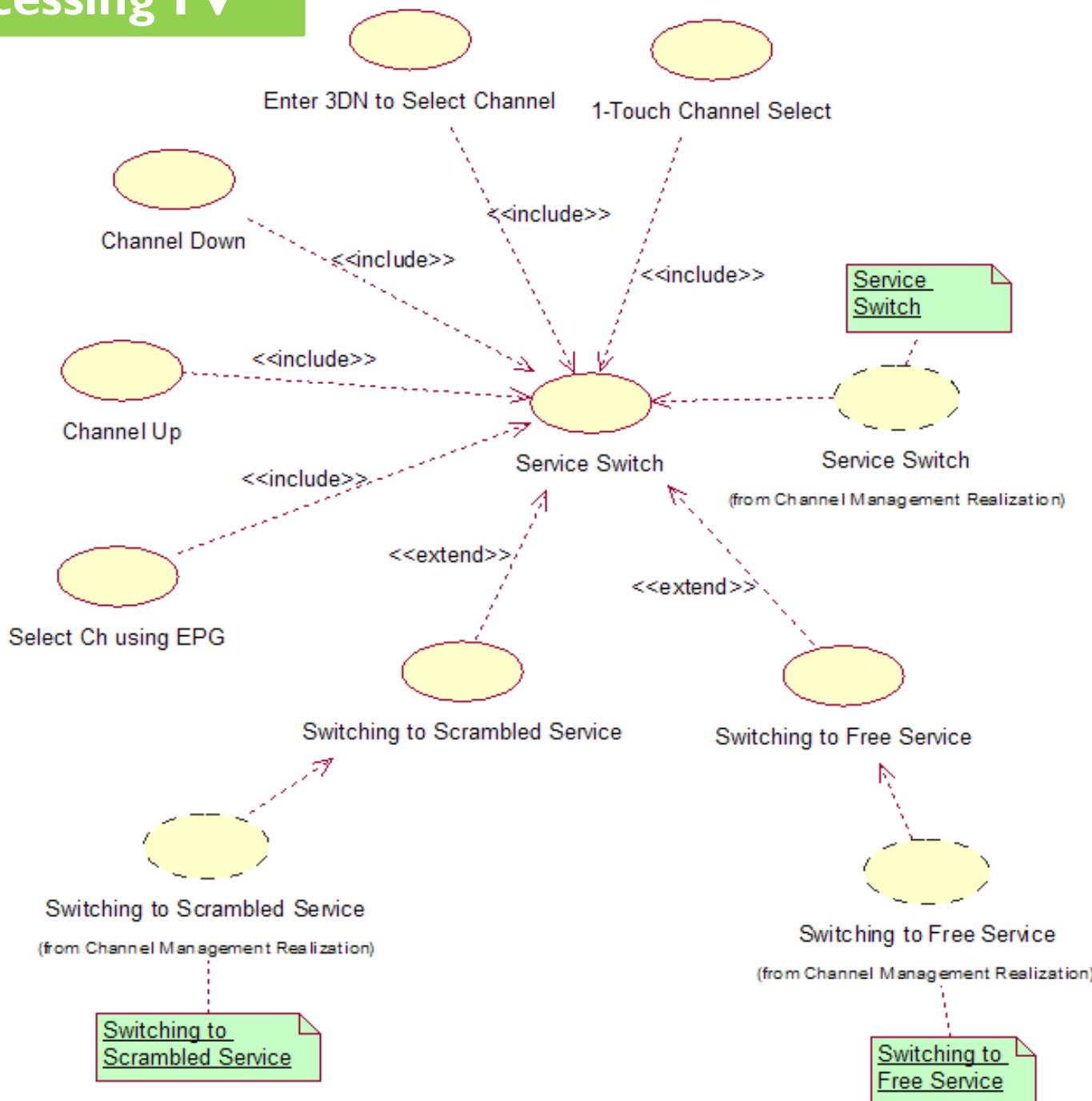
Channel Mgmt. Use Case View



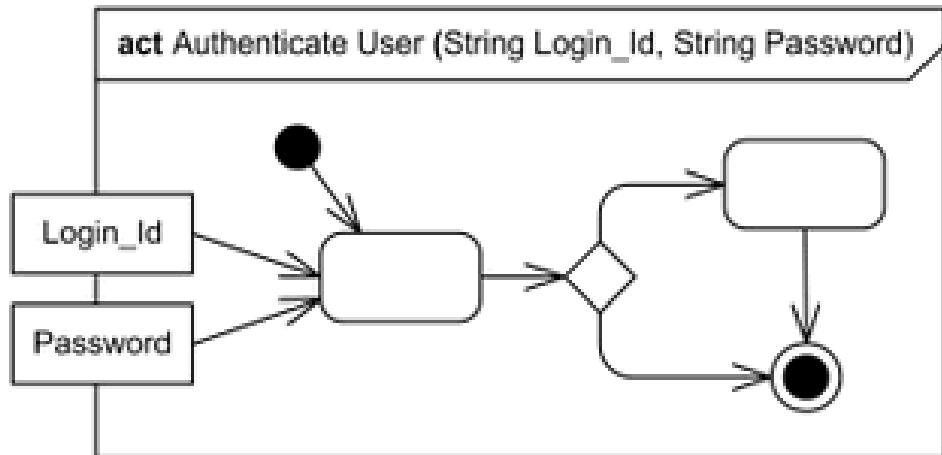
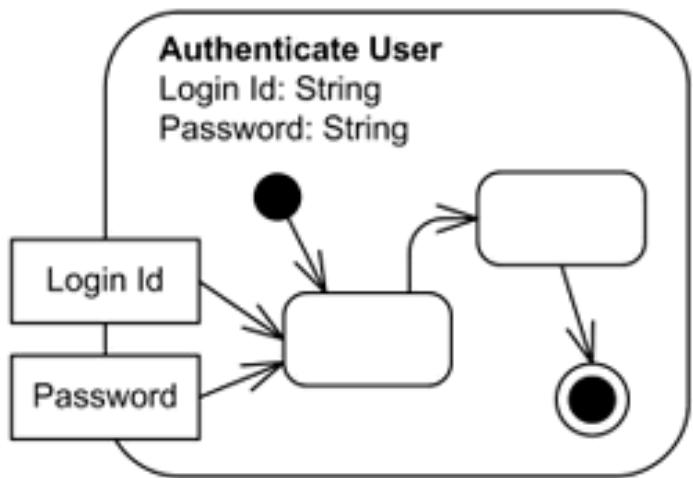
Channel Mgmt. Use Case Scenarios



User Accessing TV



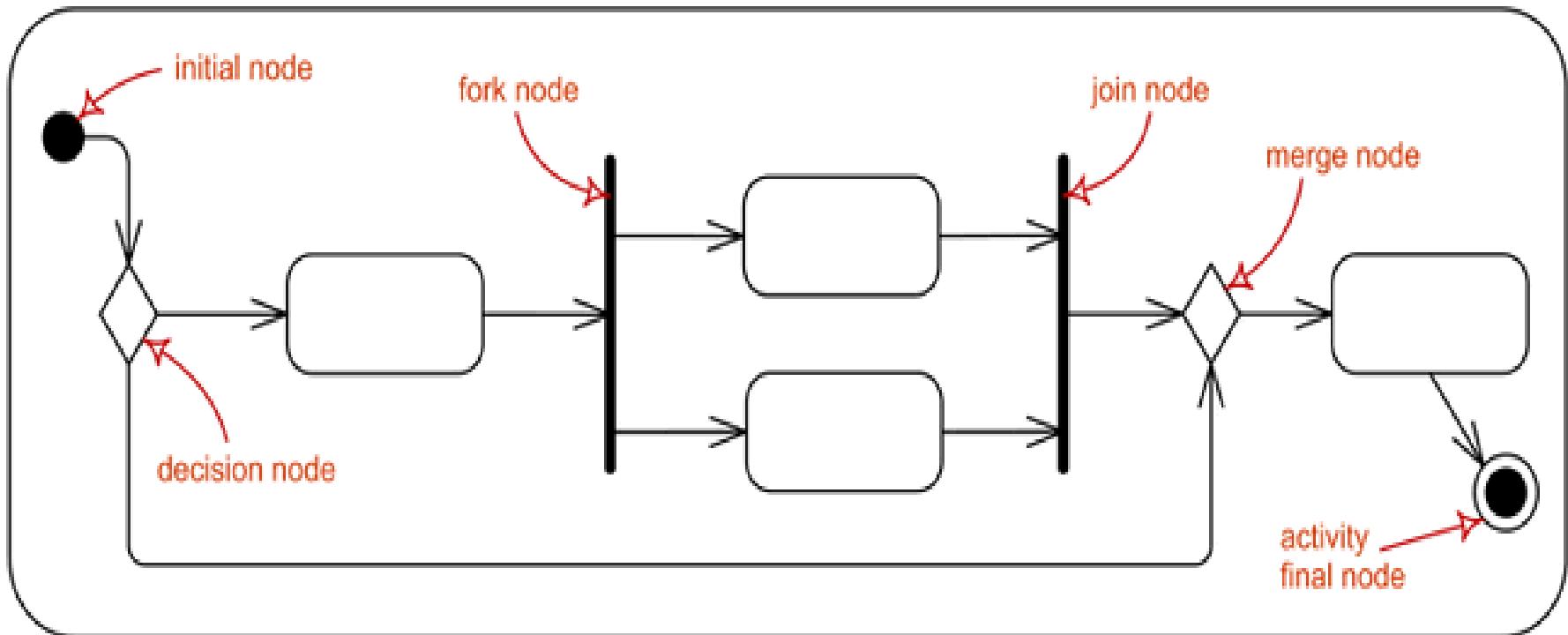
Activity Diagram



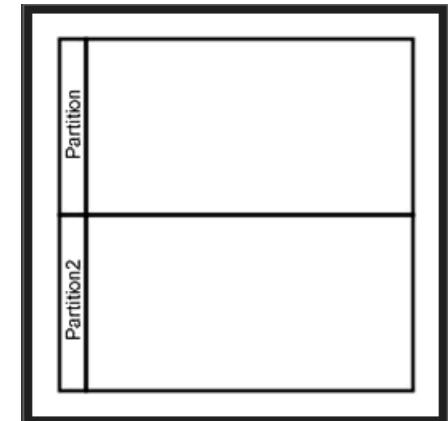
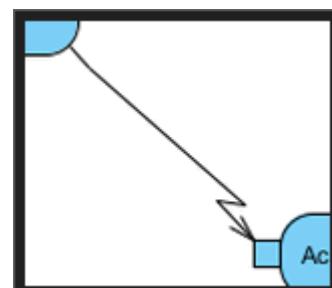
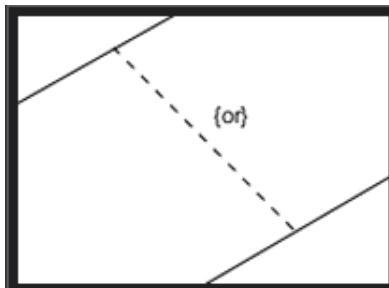
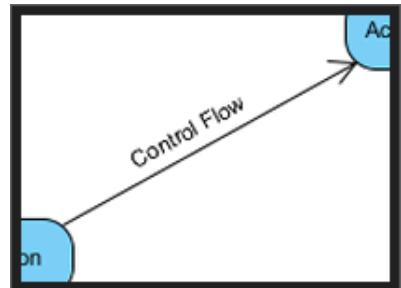
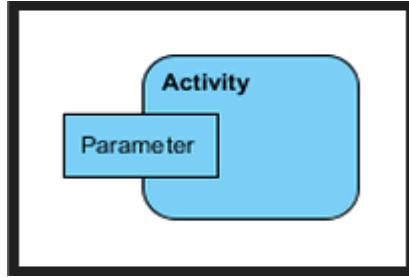
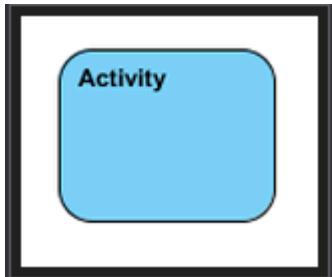
Notation

	Activity		Activity Parameter Node
	Action		Accept Event Action
	Accept Time Event Action		Activity Final Node
	Central Buffer Node		Conditional Node Specification
	Control Flow		Constraint
	Data Store Node		Decision Node
	Exception Handler		Expansion Node
	Expansion Region		Flow Final Node
	Fork Node		Initial Node
	Input Pin		Interruptible Activity Region
	Join Node		Loop Node
	Merge Node		Note
	Object Flow		Object Node
	Output Pin		Send Signal Action
	Sequence Node		Structured Activity Node
	Swimlane		Value Pin

Activity Diagram



Activity Diagram (Notations)



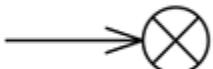
Activity Diagram (Notations)

Initial Node



Activity initial node.

Flow Final Node



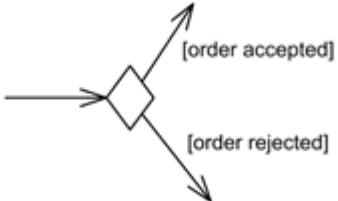
Flow final node.

Activity Final Node

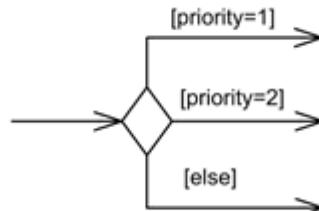


Activity final node.

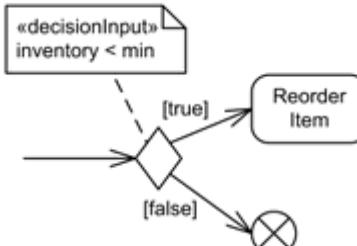
Decision



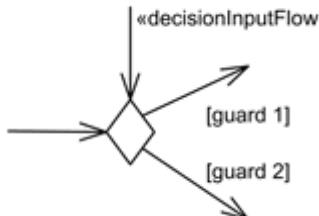
Decision node with two outgoing edges with guards.



Decision node with three outgoing edges and [else] guard.

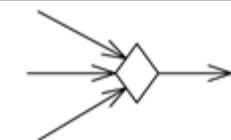


Decision node with decision input behavior.



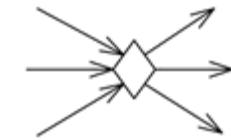
Decision node with decision input flow.

Merge



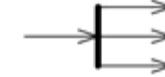
Merge node with three incoming edges and a single outgoing edge.

Merge and decision combined



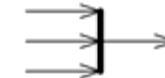
Merge node and decision node combined.

Fork



Fork node with a single activity edge entering it, and three edges leaving it.

Join Node



Join node with three activity edges entering it, and a single edge leaving it.



Join node with join specification shown in curly braces.

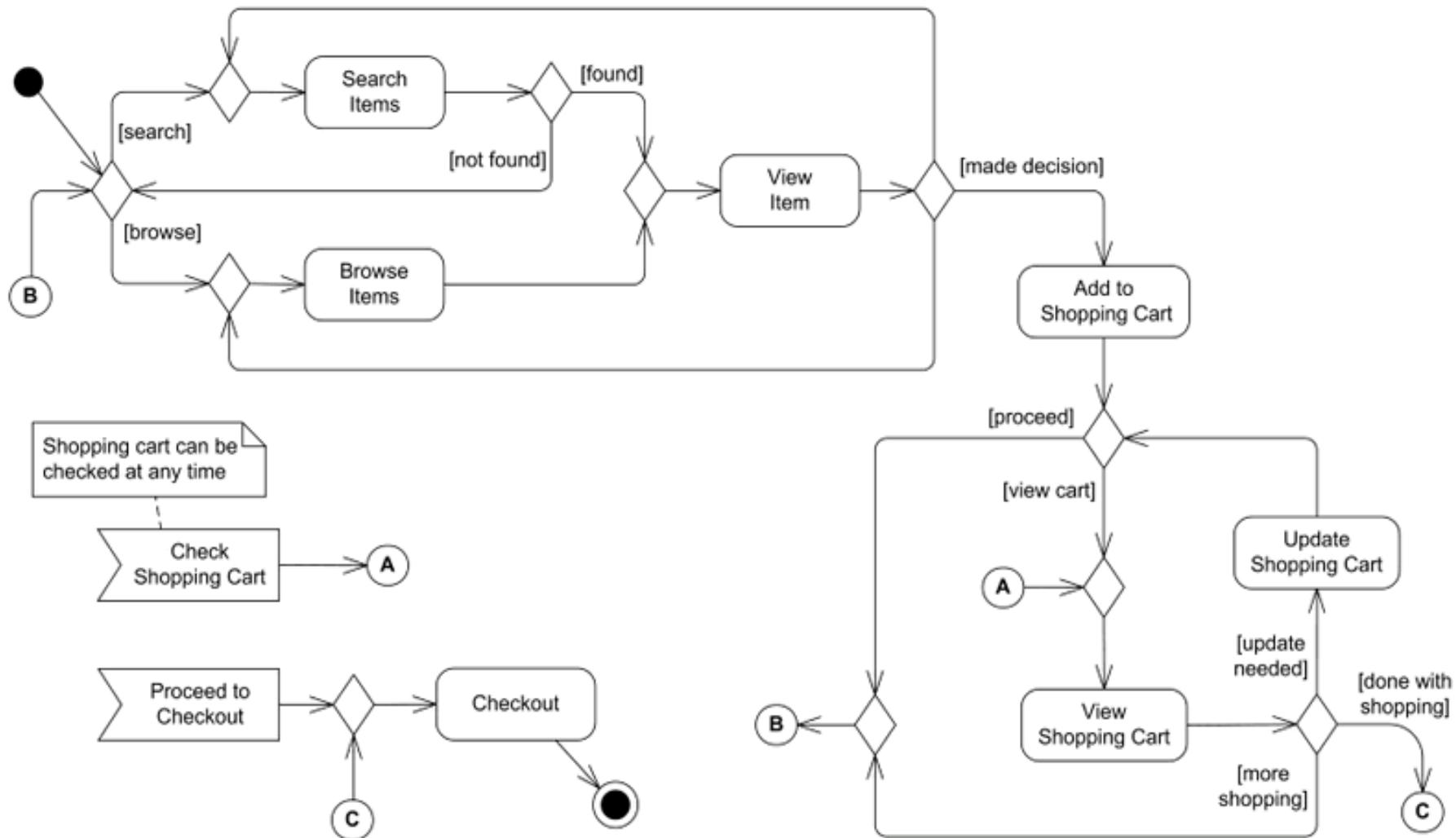
Join and fork combined



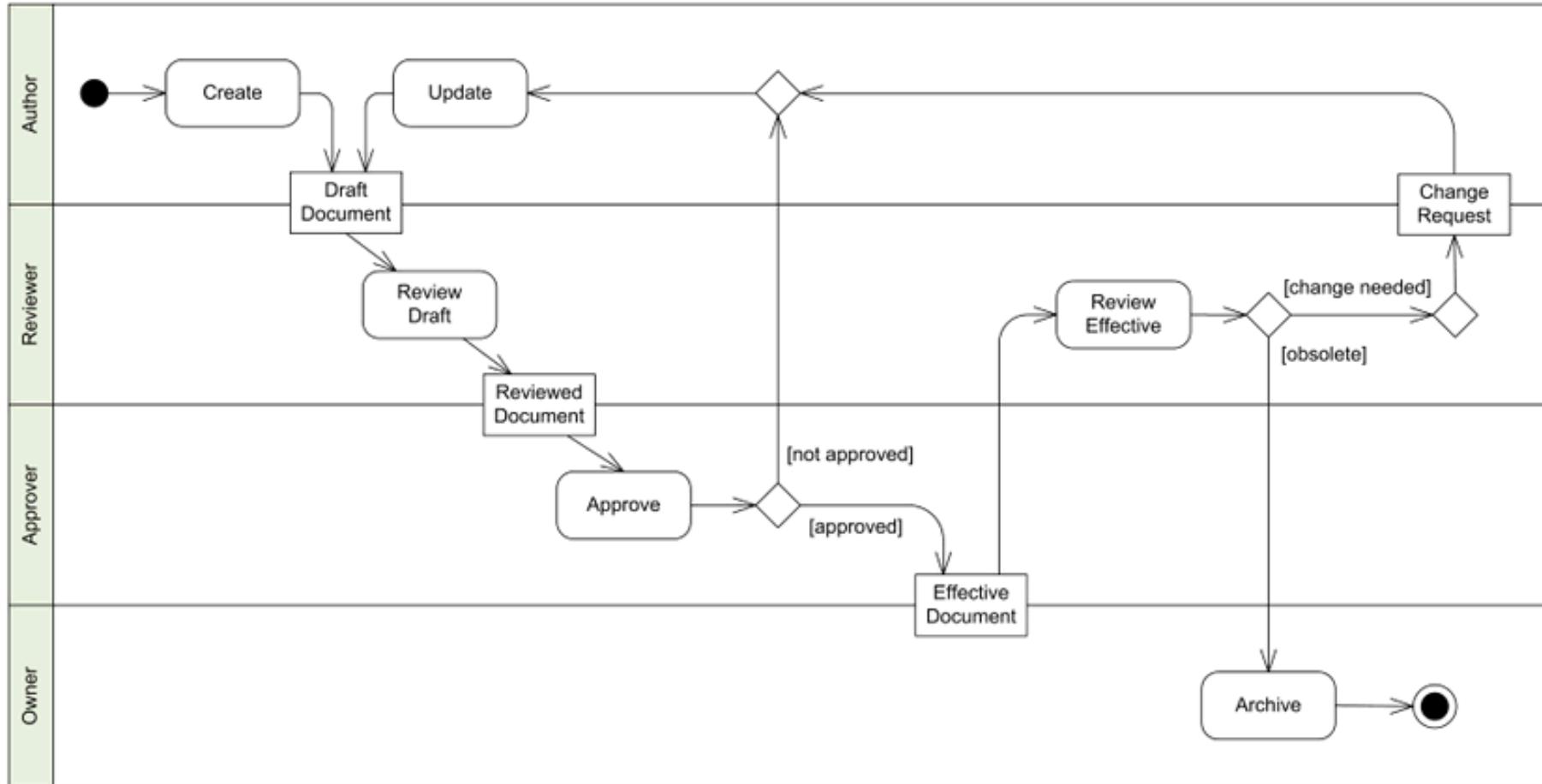
Combined join node and fork node.

Activity Diagram - Online Shopping

Online Shopping

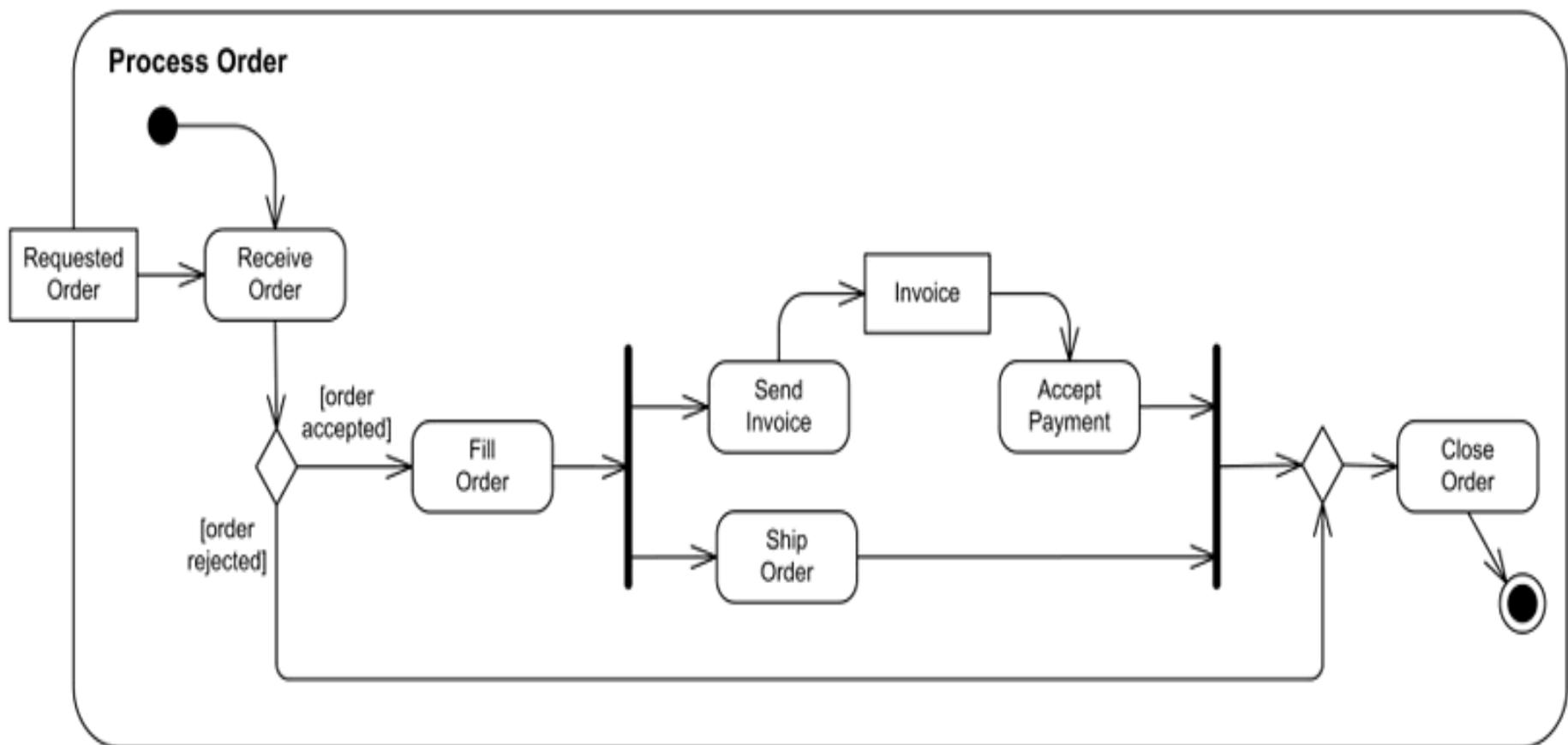


Activity Diagram with Swim Lanes

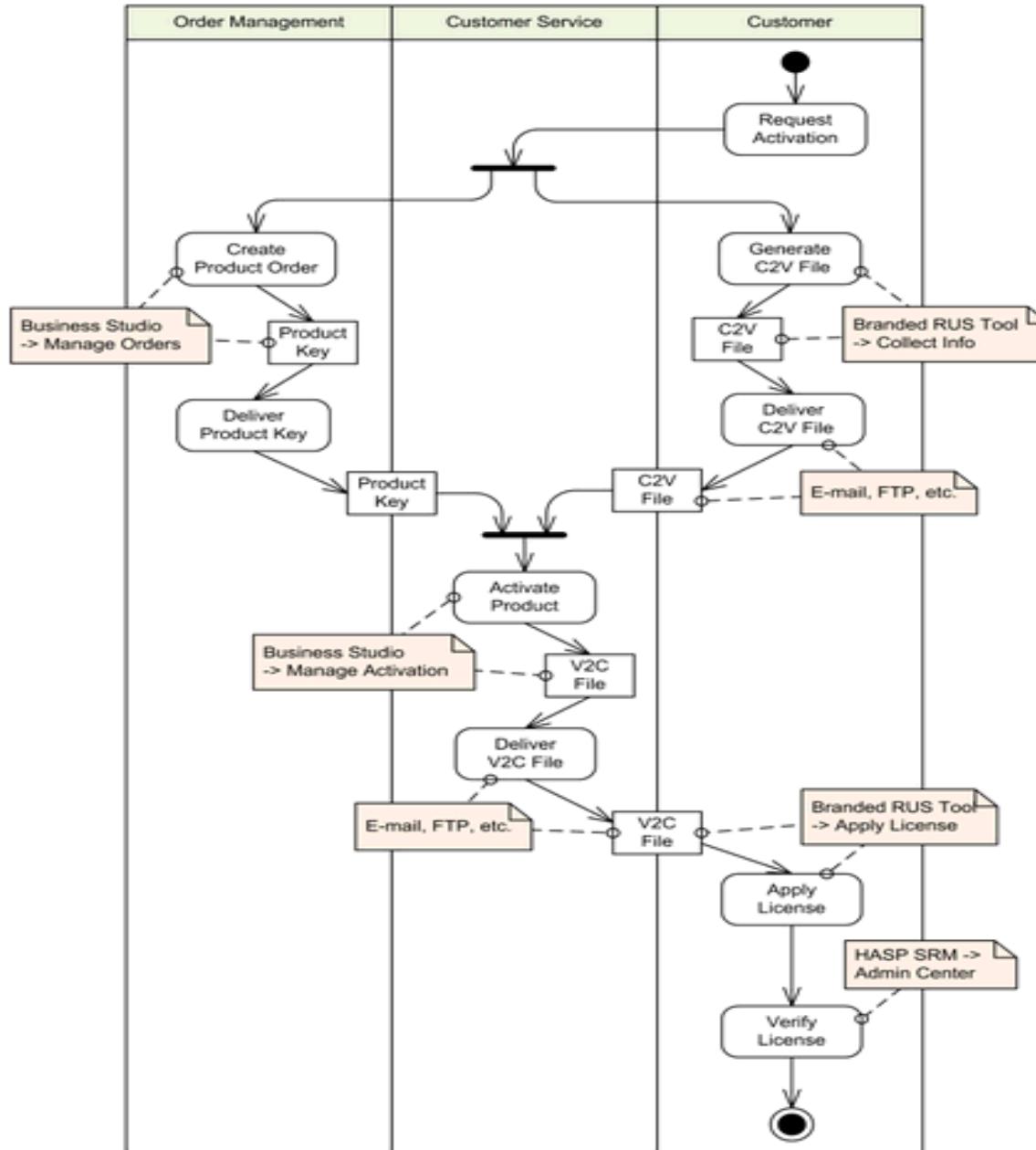


An example of Document Management Process activity.

Activity Diagram with Fork

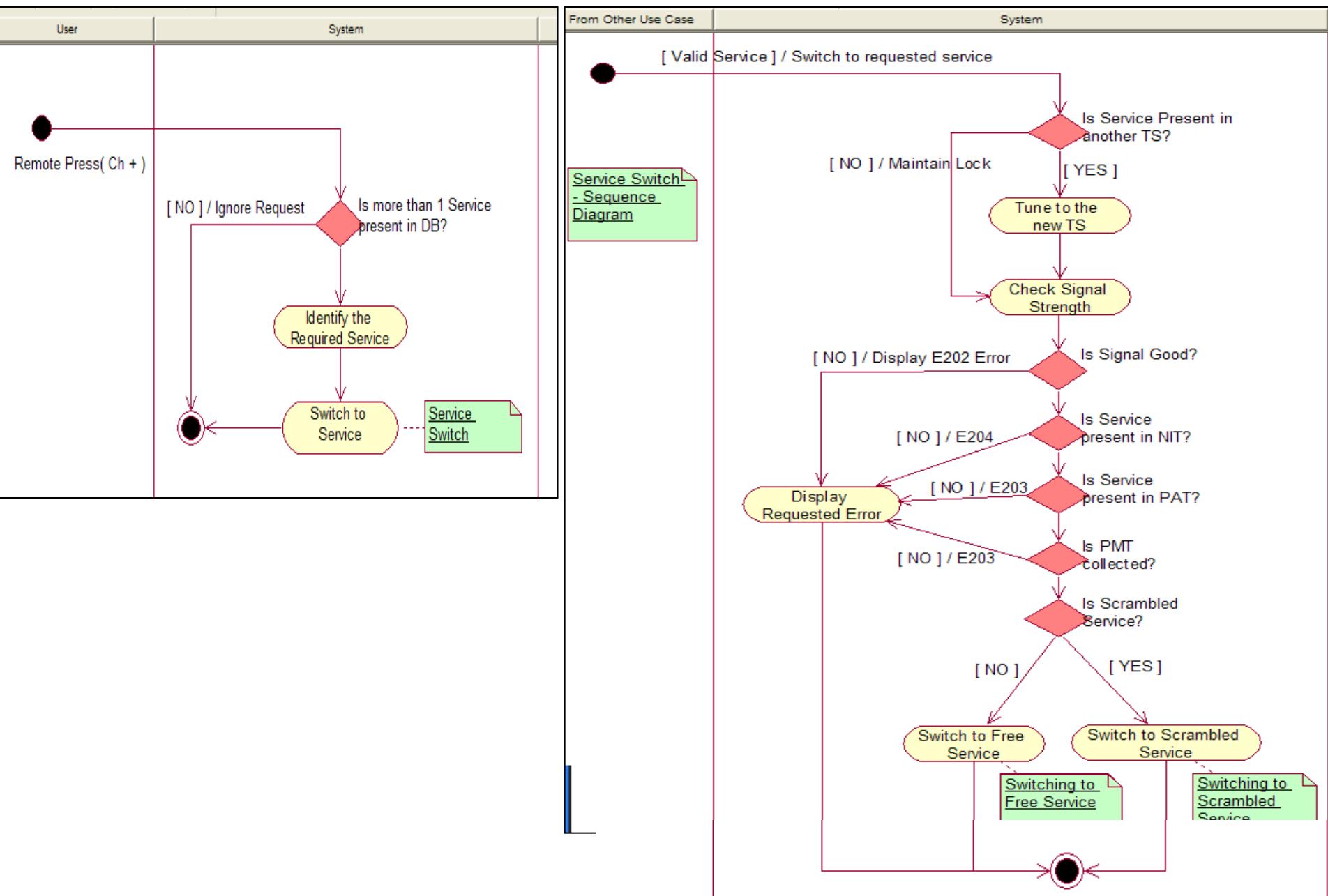


Activity Diagram with Fork

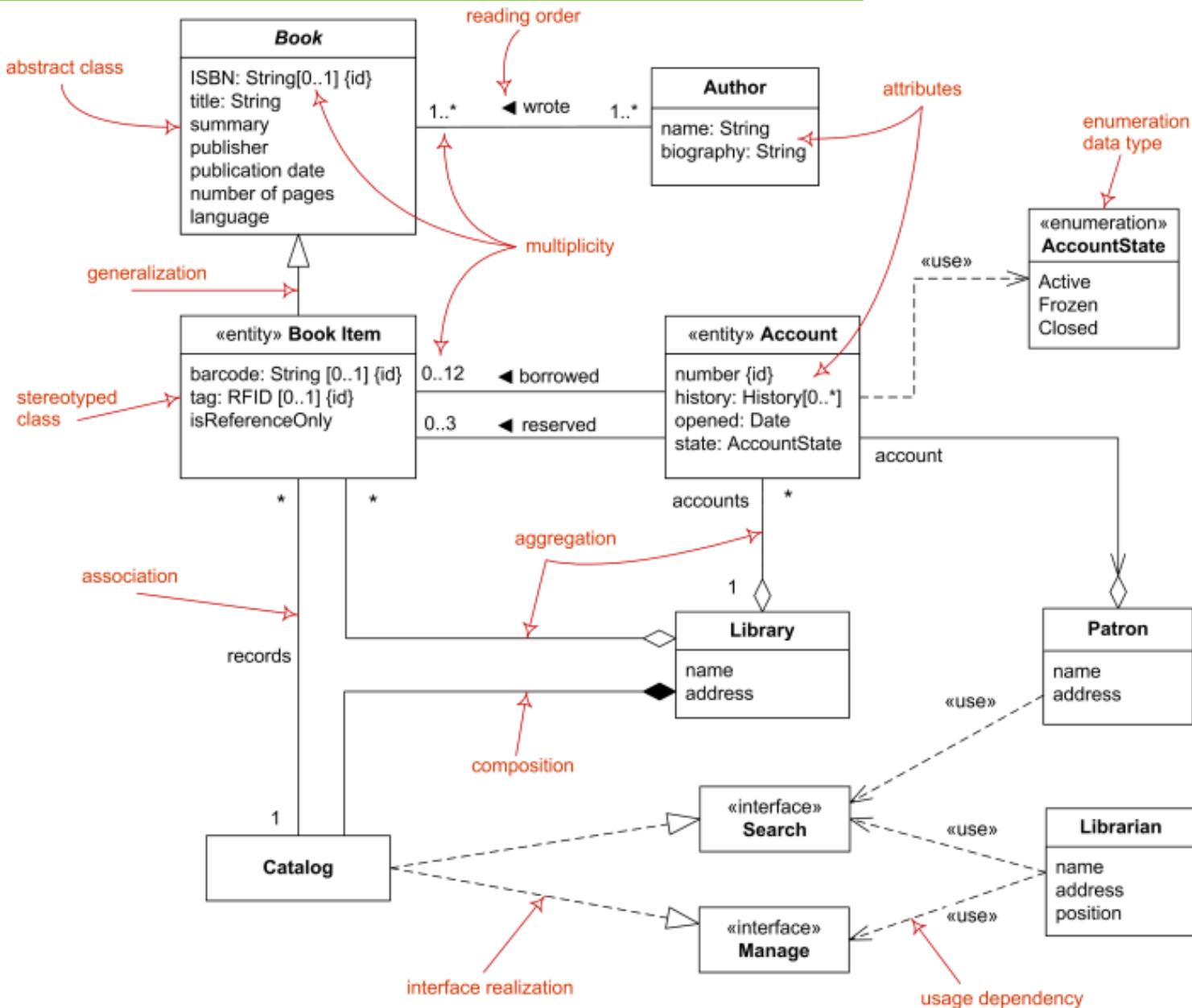


An example of activity for manual activation of trial product protected by HASP SL.

Activity Diagram

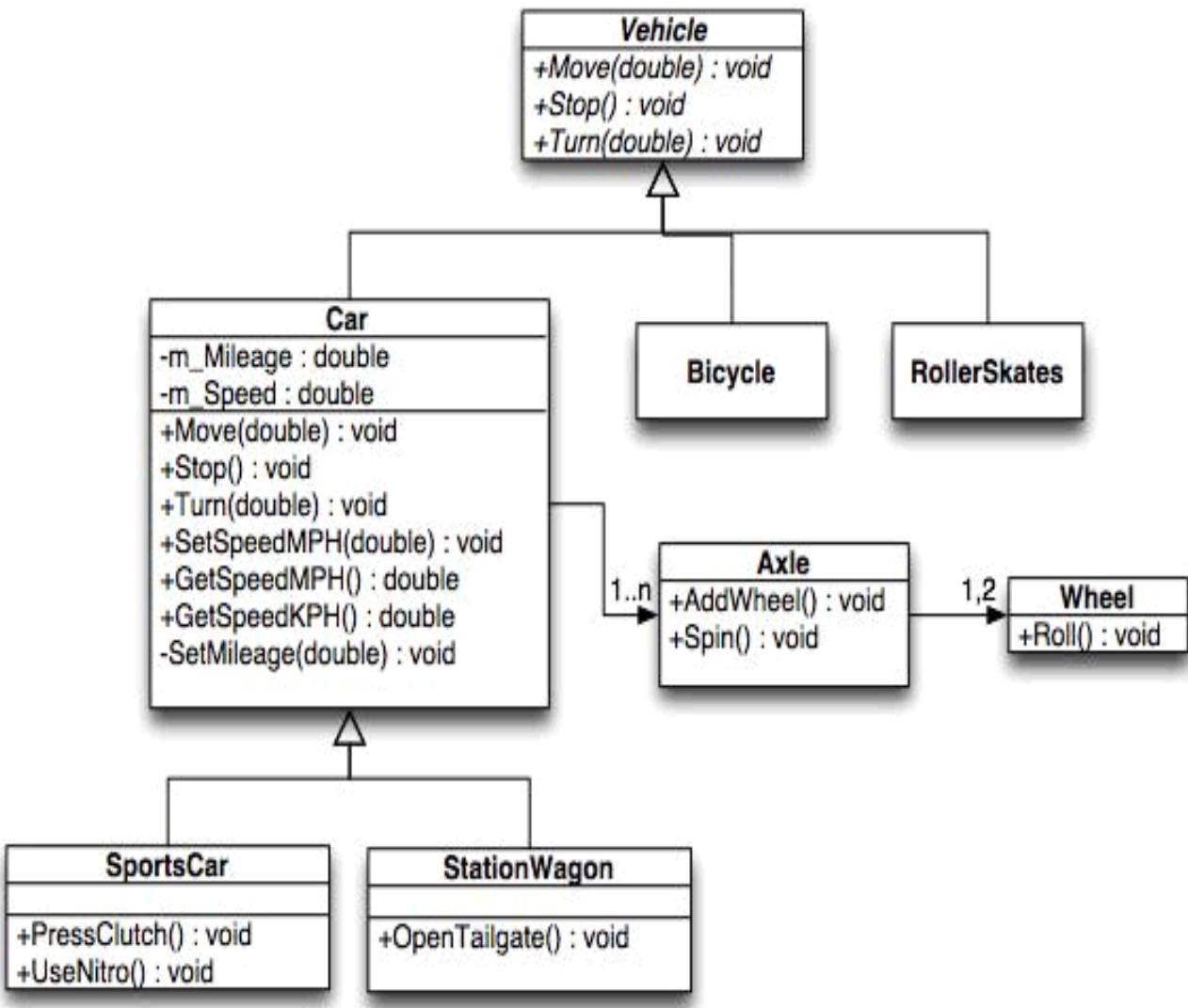


Domain Model Diagram



Domain diagram overview - classes, interfaces, associations, usage, realization, multiplicity.

Class Diagram



Notation

	Abstraction		Access
	Aggregation (Shared association)		Association (Without aggregation)
	Association Class		Binding
	Class		Class
	Class <<Interface>>		Class <<Primitive>>
	Class <<ORM-Persistable>>		Class <<ORM-Abstract-Persistable>>
	Class <<ORM-User-Type>>		Class <<ORM-Parameterized-Type>>
	Class <<Entity Bean>>		Collaboration
	Composition (Composite association)		Constraint
	Dependency		Derive
	Generalization		Import
	Instantiation		Merge
	Model		NARY
	Note		Permission
	Realization		Refine
	Substitution		Trace
	Usage		

Class Diagram (Notations)

Class

<<enumeration>>
Class

<<Interface>>
Class

<<primitive>>
Class

Collaboration

Association

Generalization

Realization

Dependency

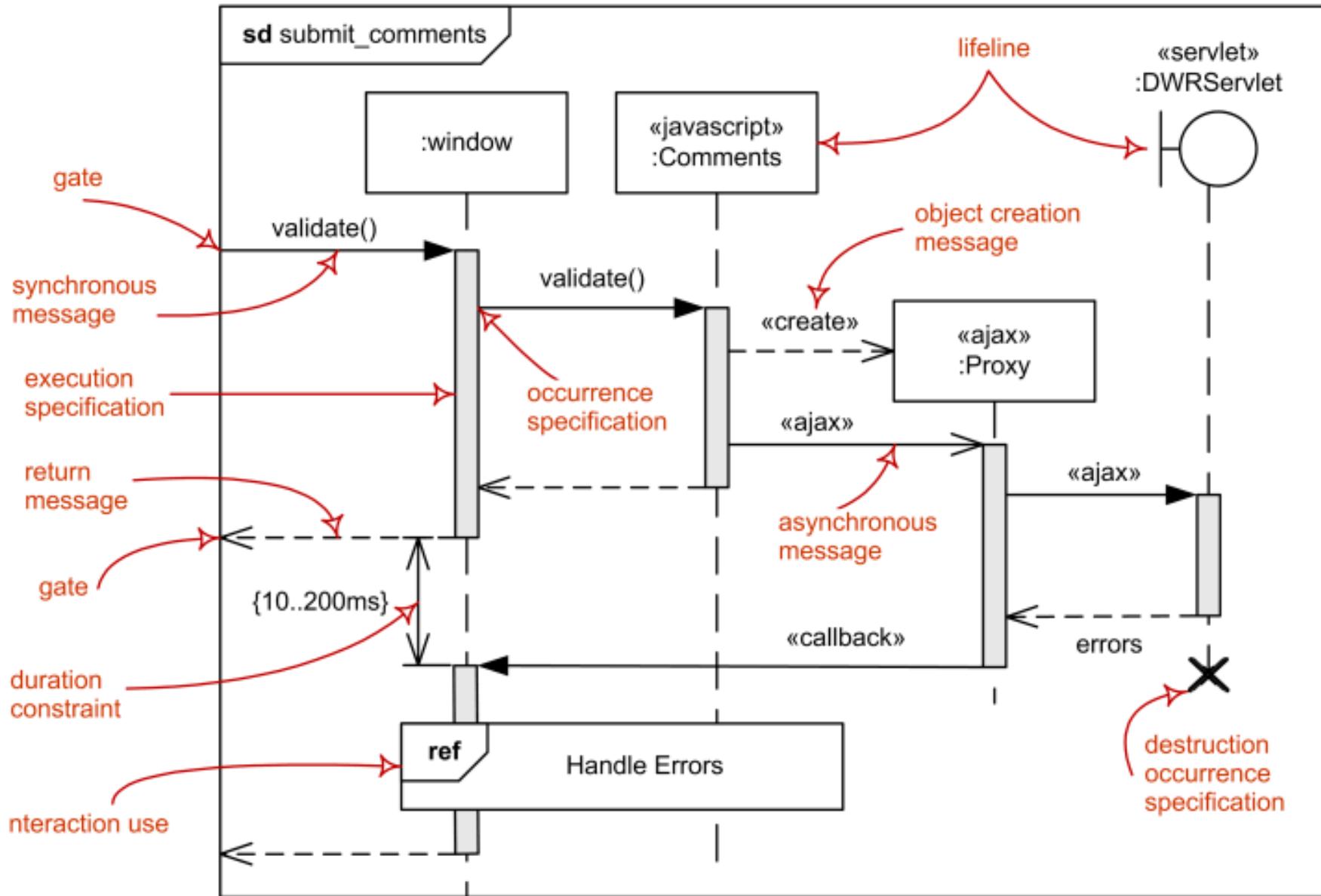
Composition

Aggregation

(or)

<<import>>
Import

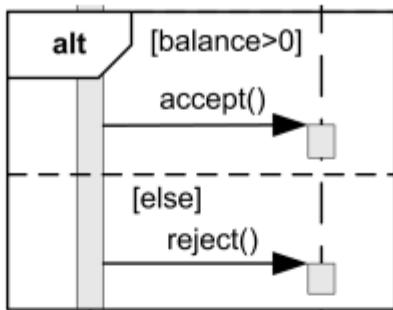
Sequence Diagram



Sequence diagram major elements.

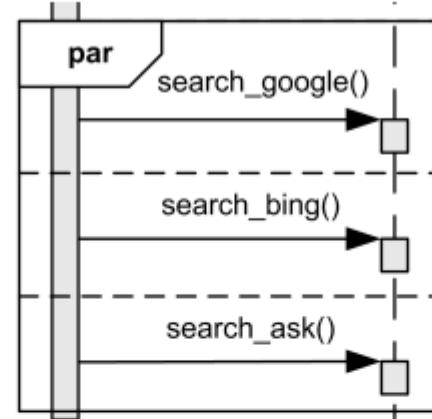
Sequence Diagram (Notations)

Alternatives



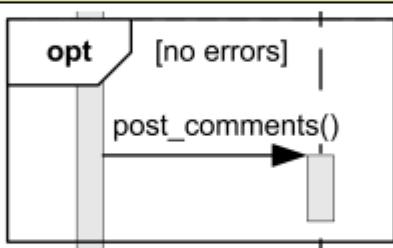
Call accept() if balance > 0, call reject() otherwise.

Parallel



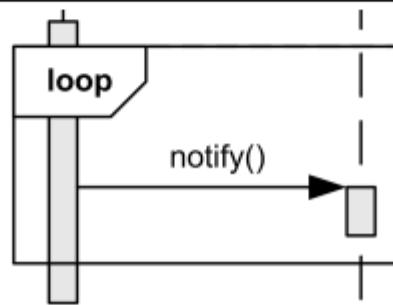
Search Google, Bing and Ask in any order, possibly parallel.

Option

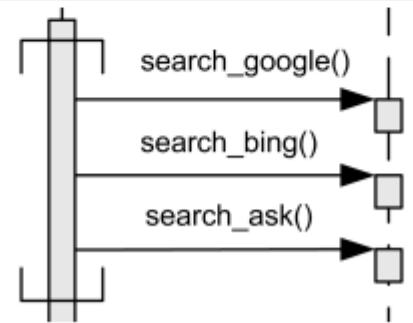


Post comments if there were no errors.

Loop

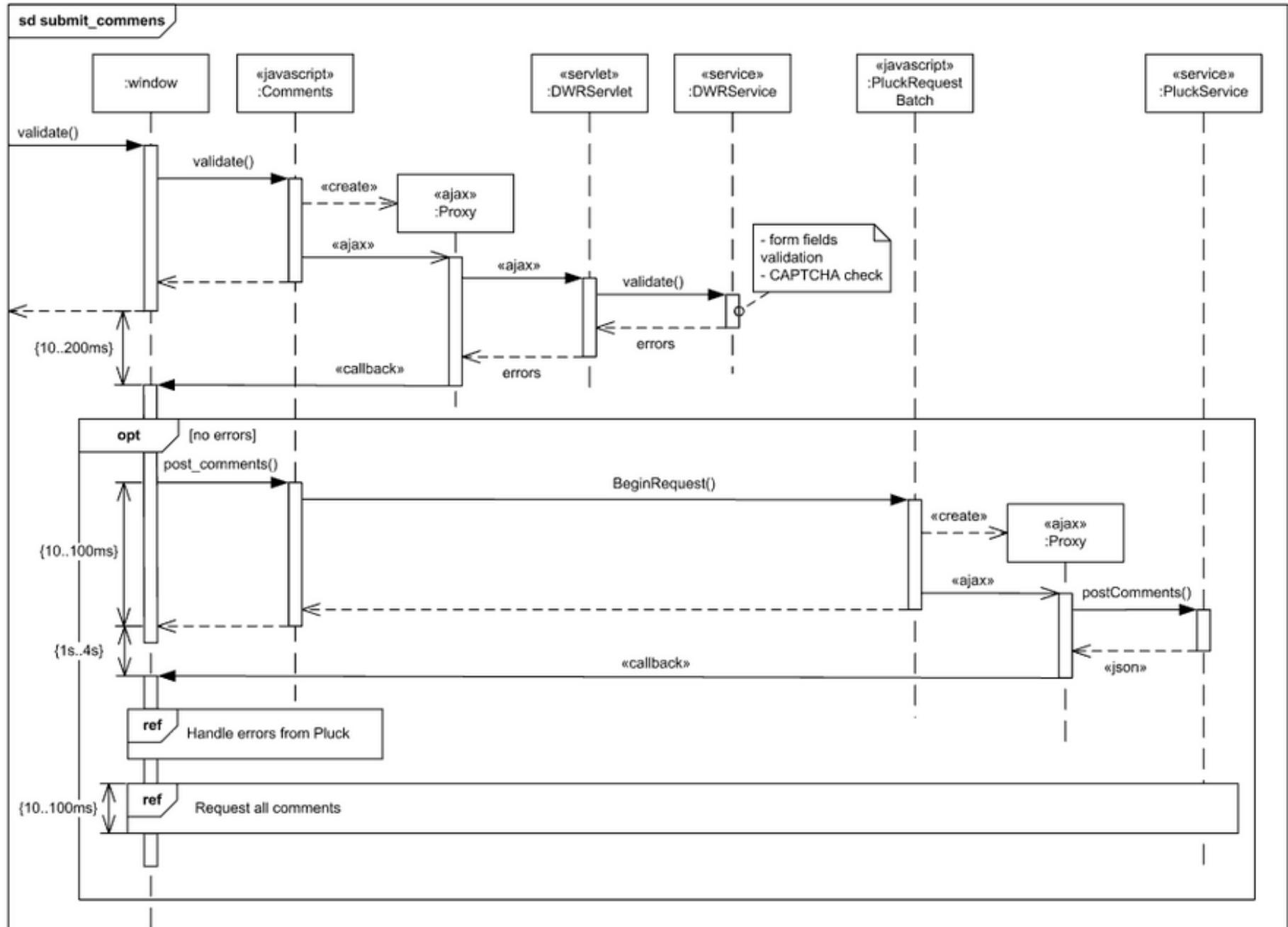


Potentially infinite loop.

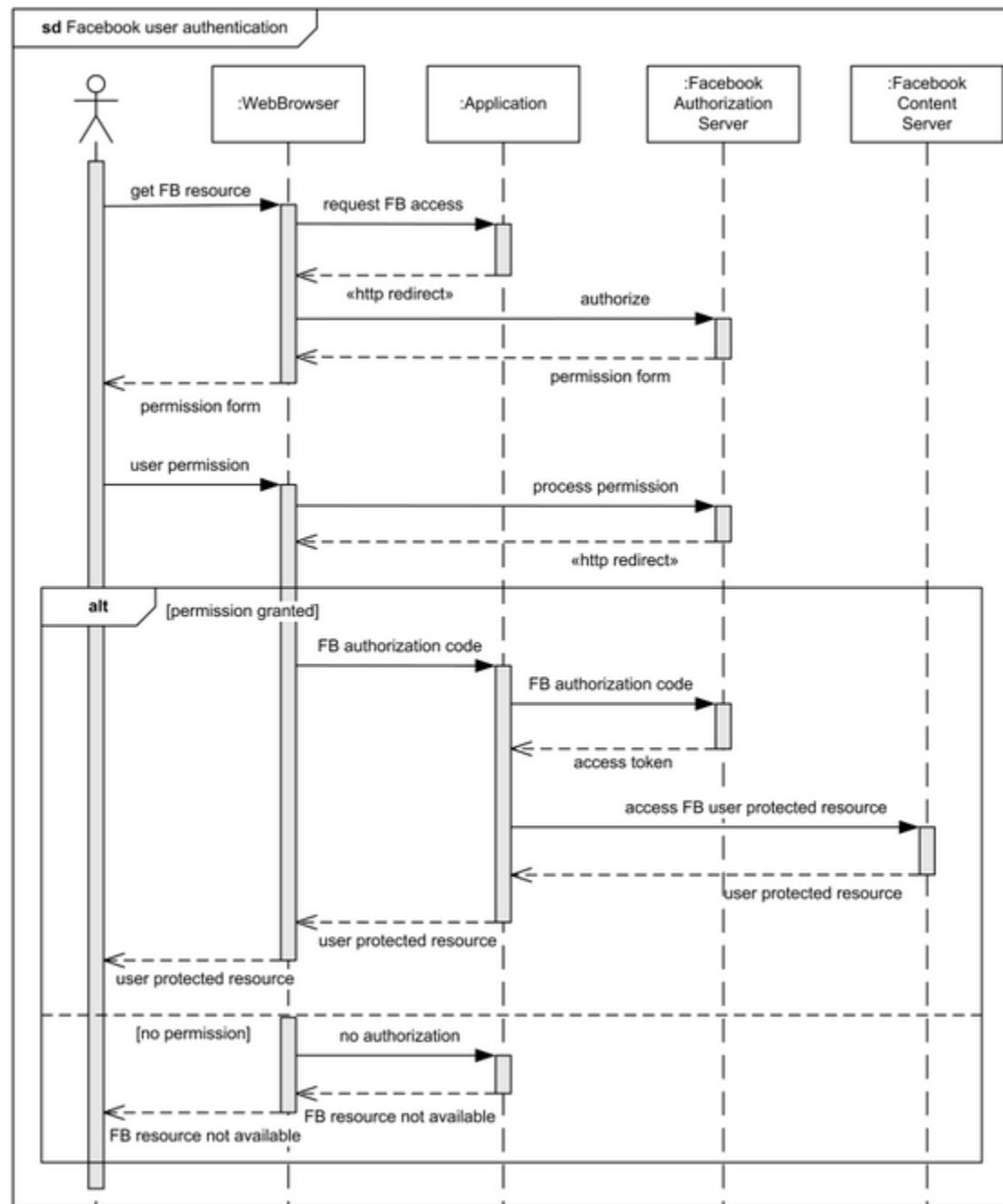


Coregion - search Google, Bing and Ask in any order, possibly parallel.

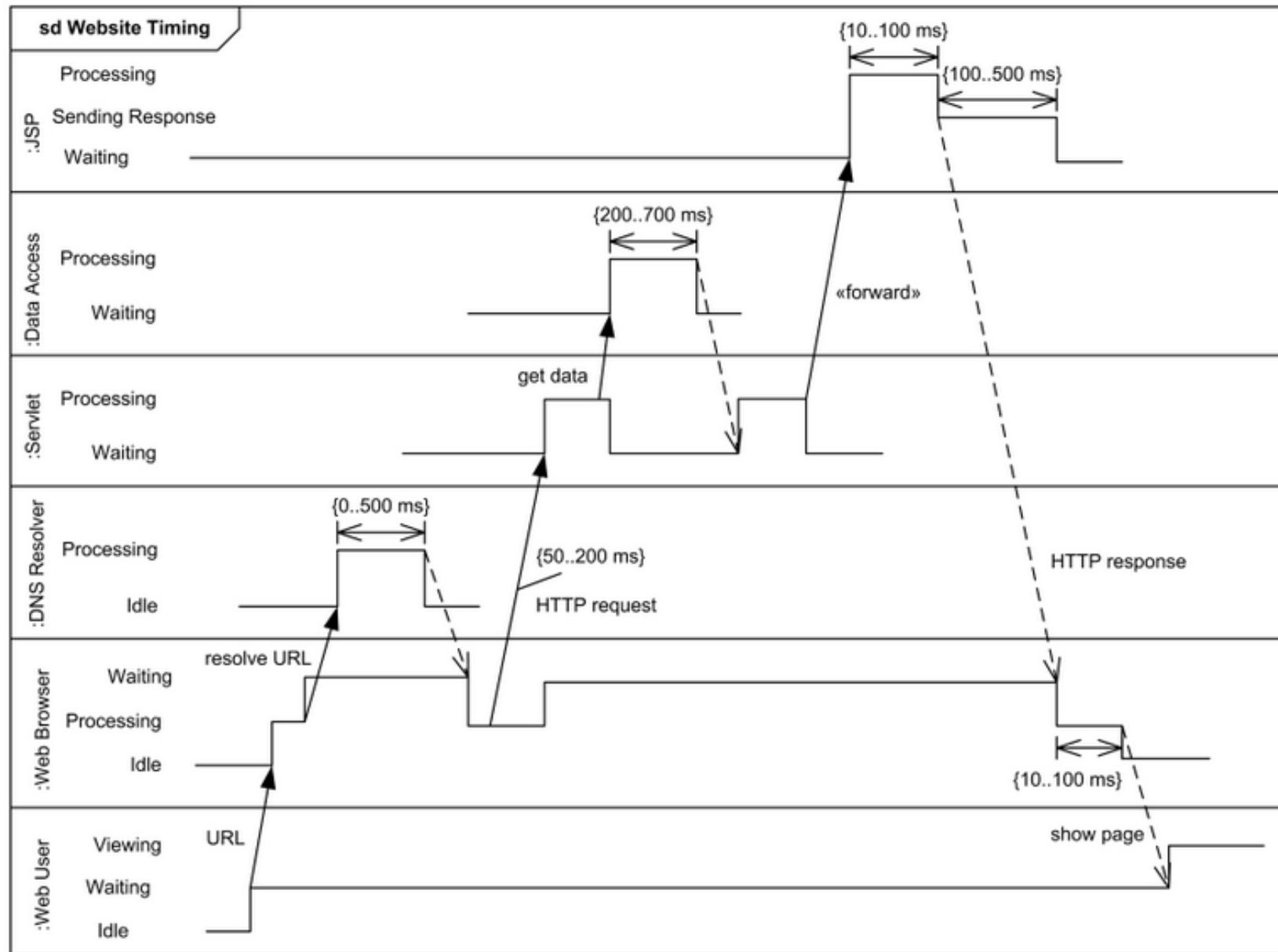
Sequence Diagram (Example)



Sequence Diagram (Cont...)



Timing Diagrams



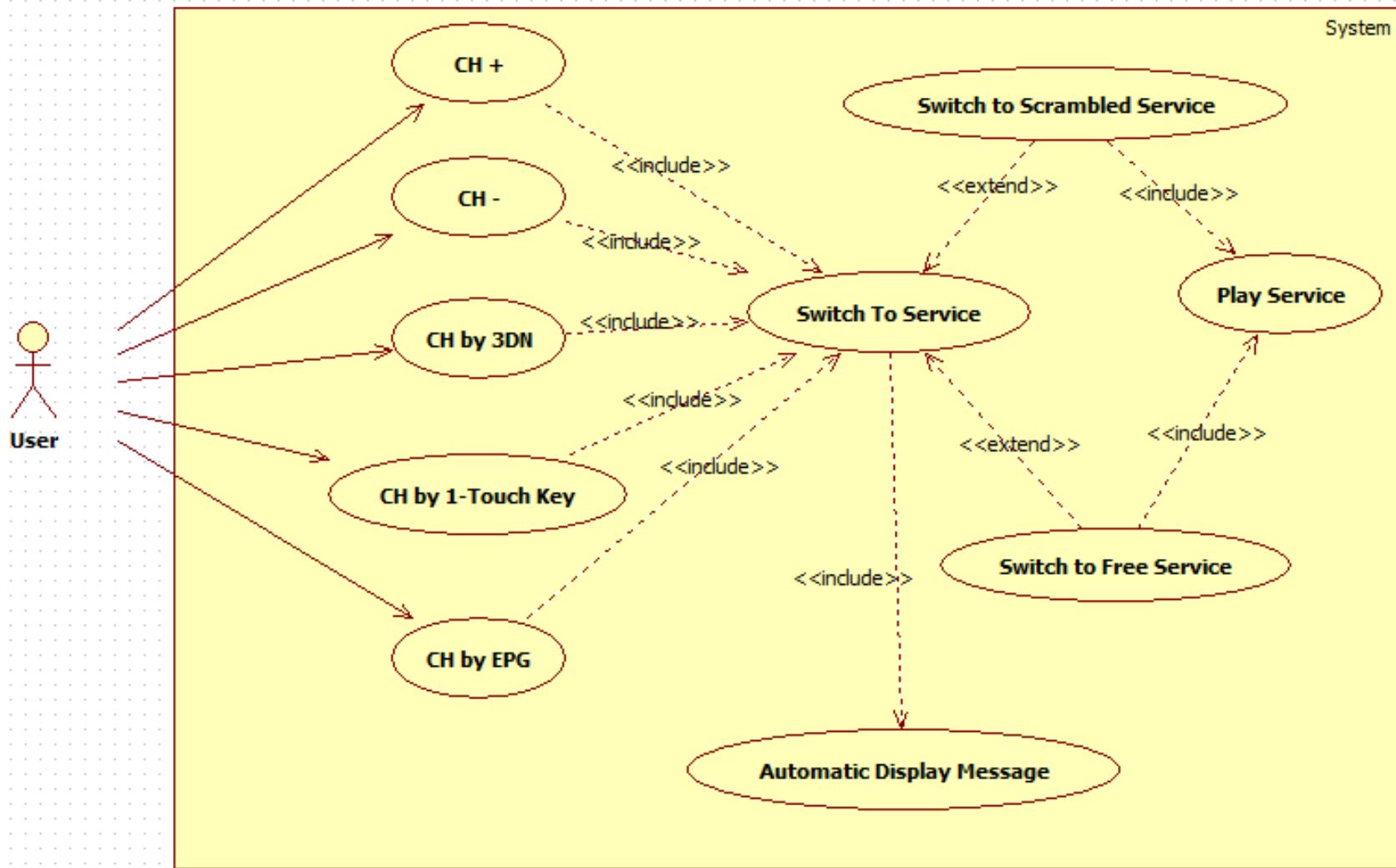
Timing Diagram Example - User Experience Website Latency

UML Approach



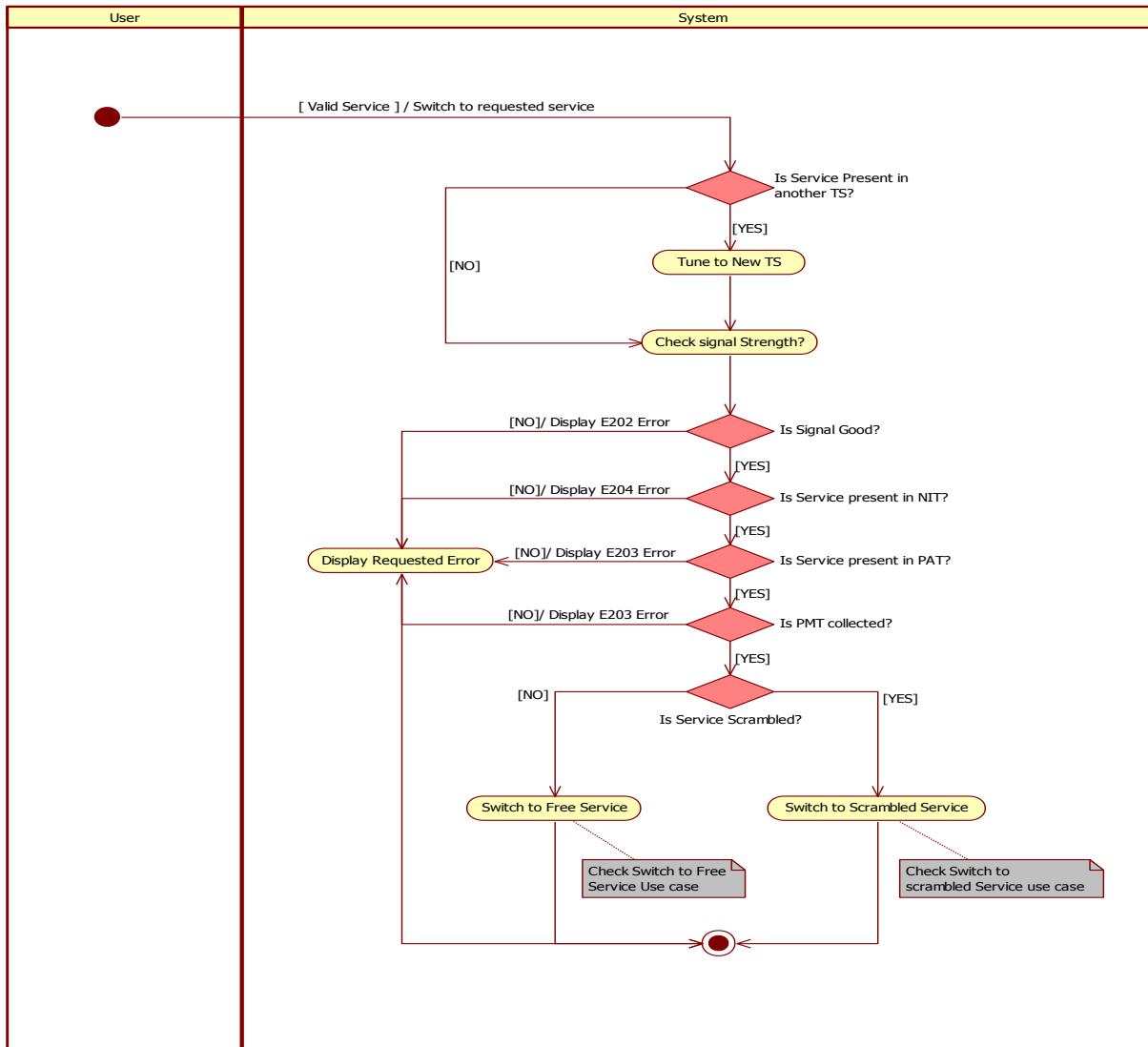
UML Approach

- <Use-Case View> (1) Start with a Use-Case Diagram



UML Approach (Cont..)

➤ <Use Case View>(2) Add Activity-Diagram to each Use-Case

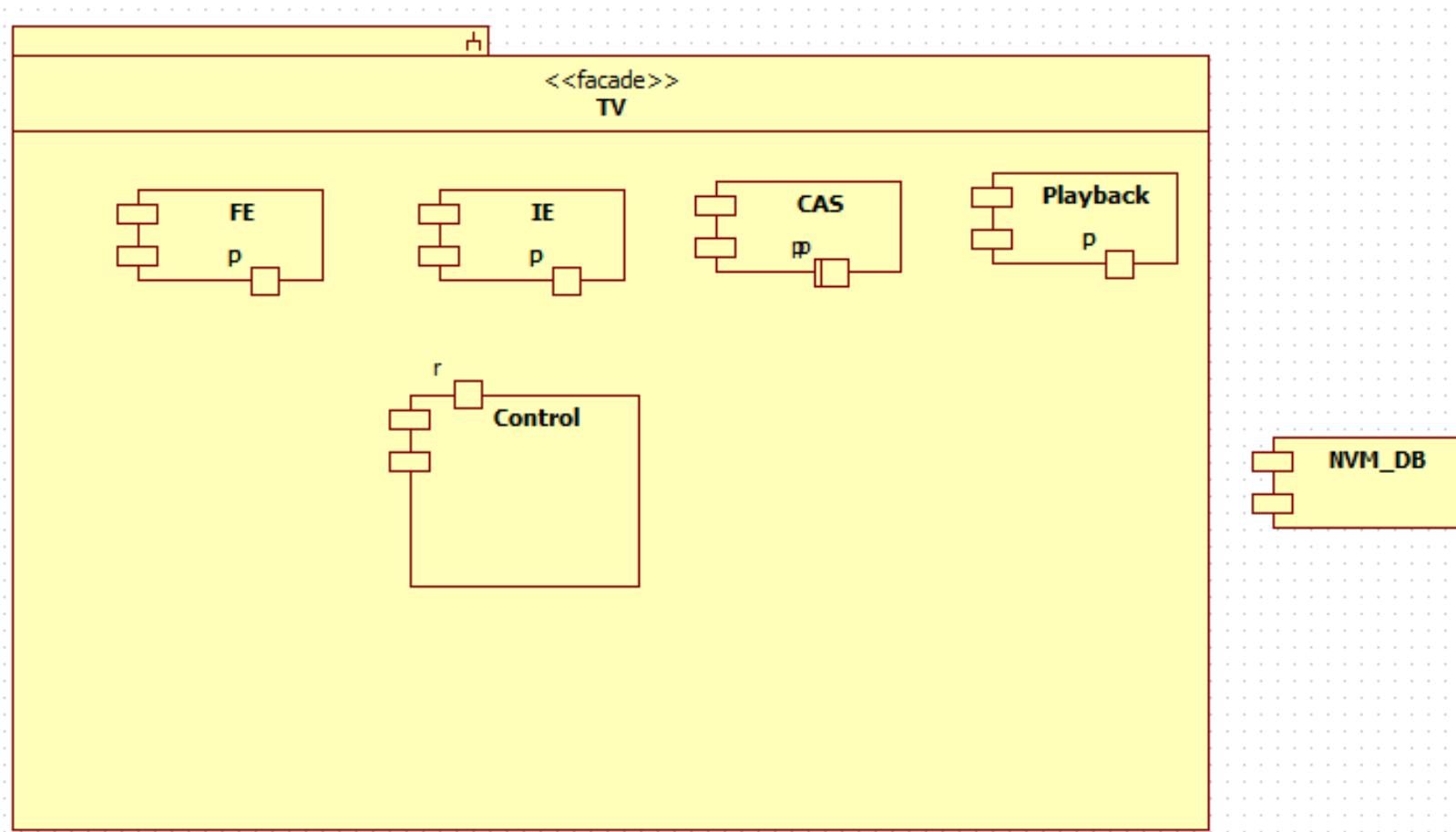


❖ NOTE

Think how to
Achieve
RE-USABLE
Diagrams

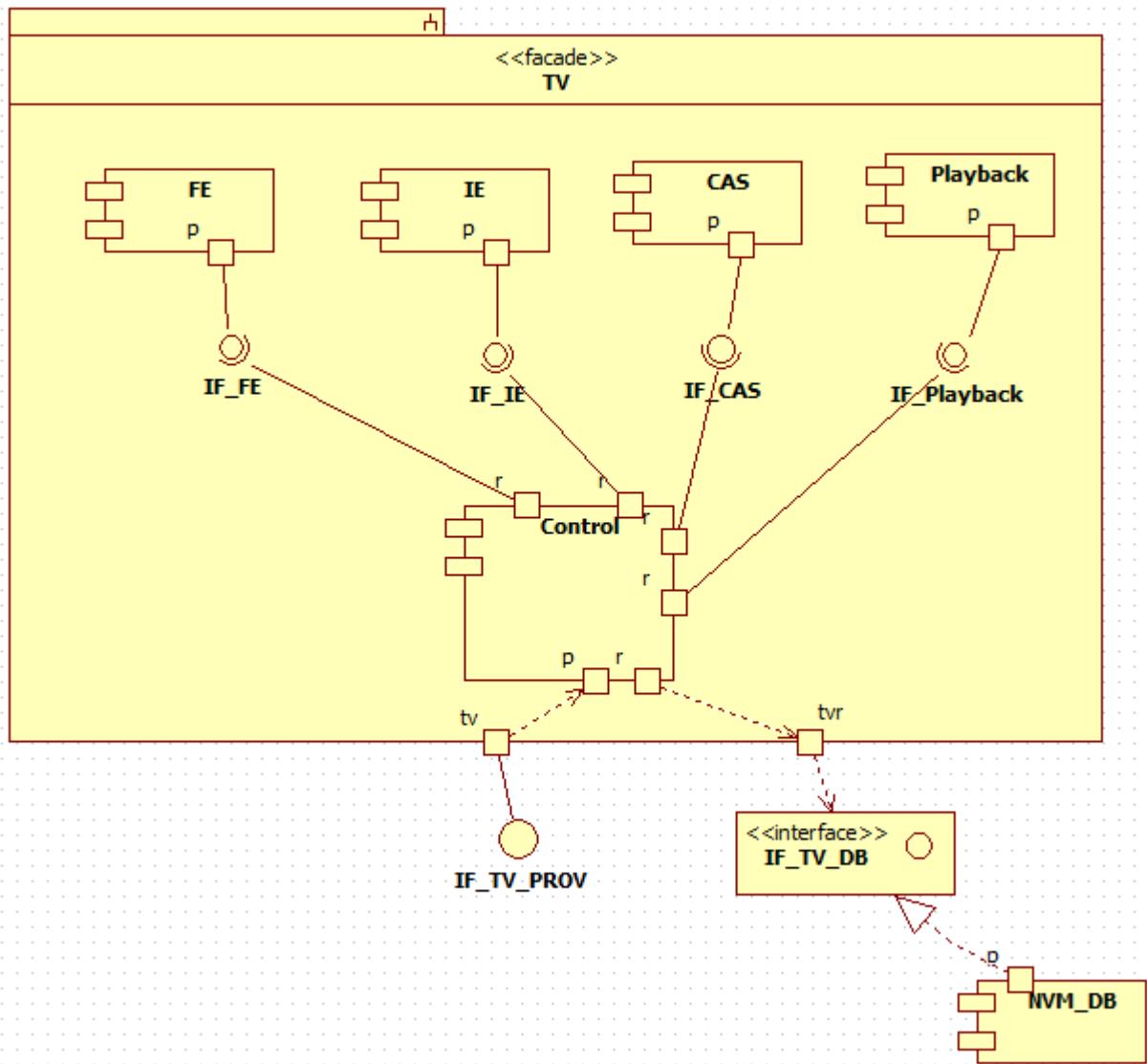
UML Approach (Cont..)

- <Component View>(3) Decide Sub-Systems & Components required based on our understanding of Use-Cases



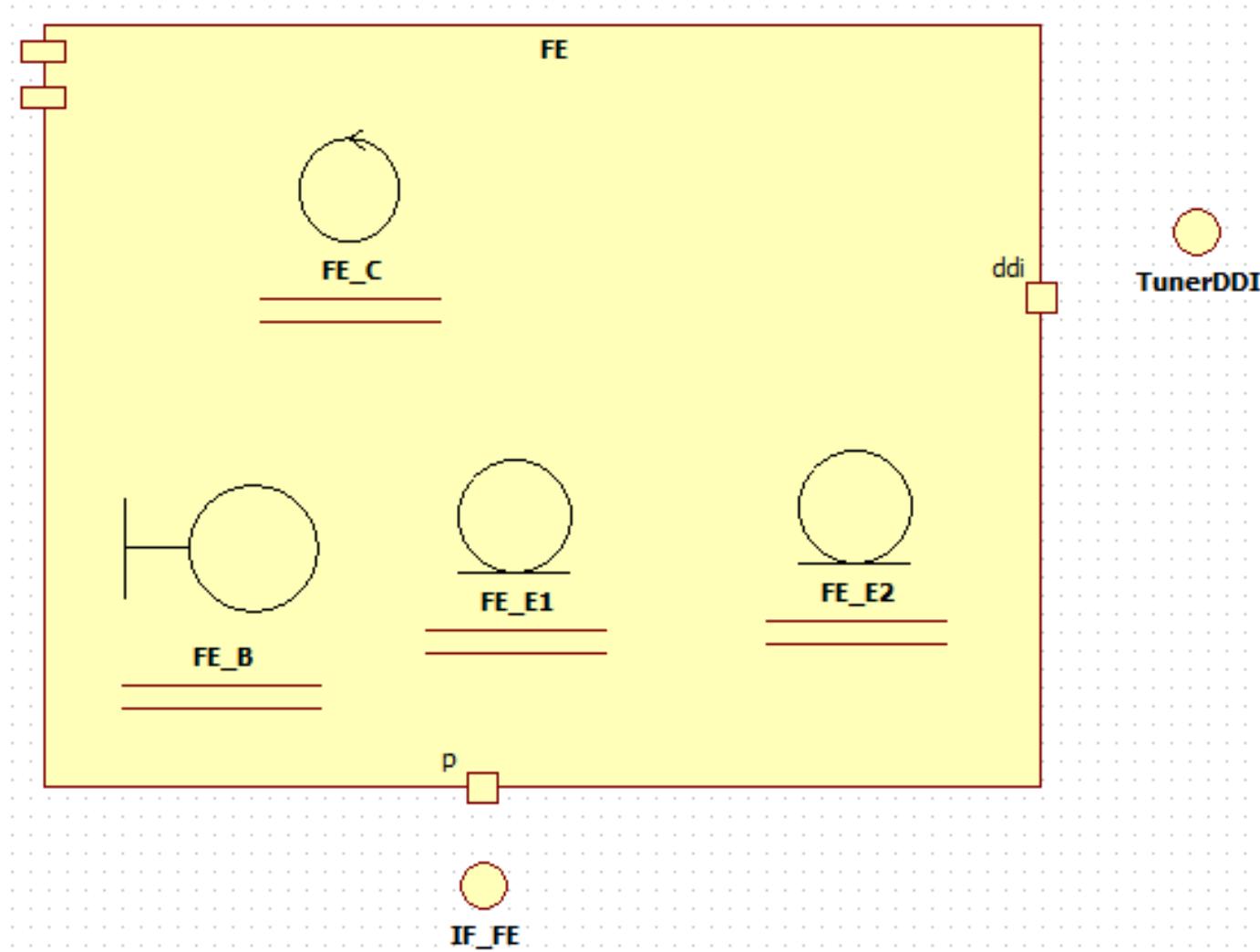
UML Approach (Cont..)

- <Component View>(4) Add Component Diagram to establish INTERFACE / RELATIONSHIPS between Components



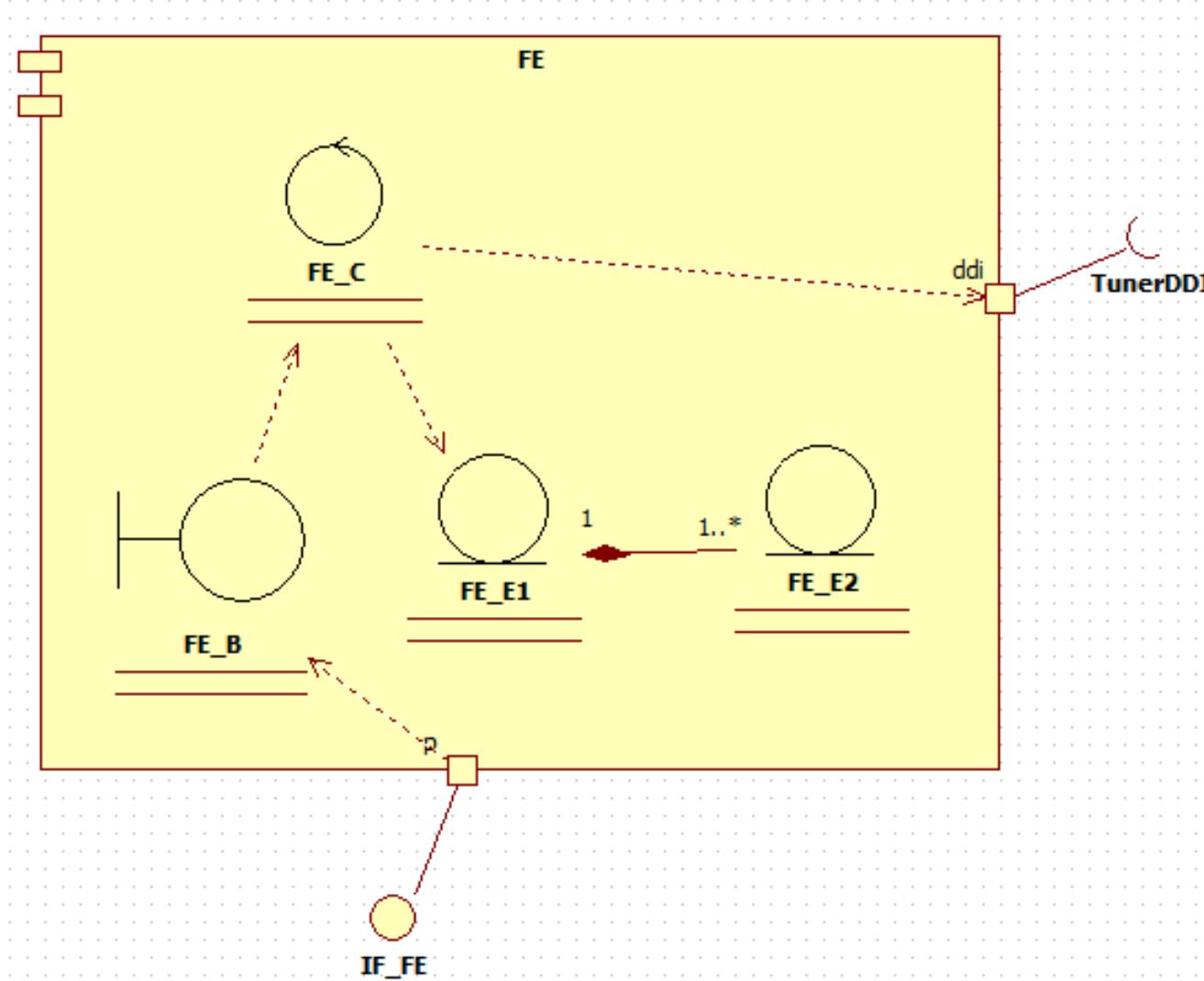
UML Approach (Cont..)

- <Logical View>(5) Realization for each Use-Case
- <Logical View>(6) add CLASSES required to realize Use-Case



UML Approach (Cont..)

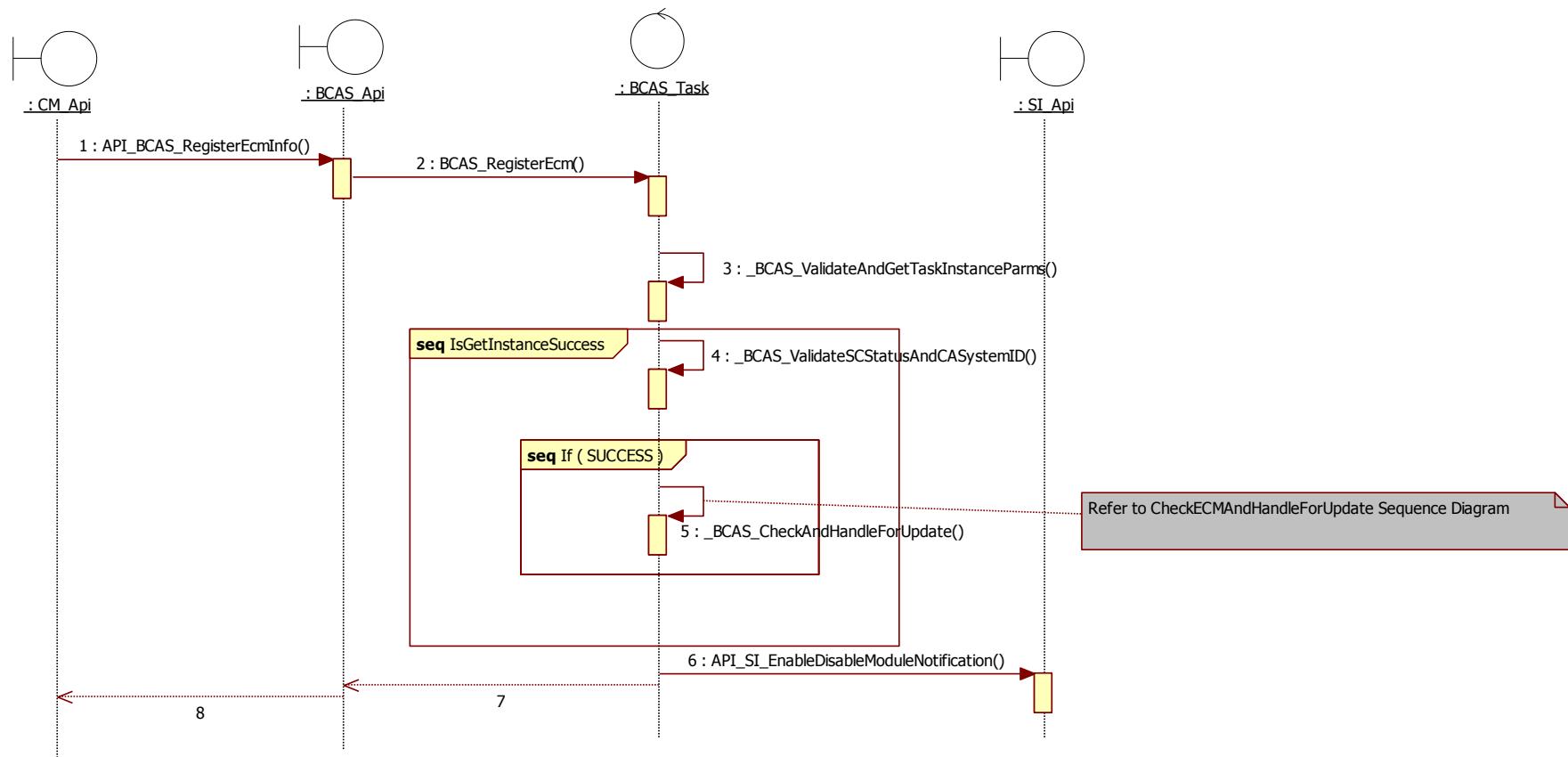
- <Logical View>(7) add Class Diagram to establish RELATIONSHIPS between classes



- <Logical View>(8) add following to components
 - State Diagram for a component – if required
 - Activity Diagram for each State – if required
- <Logical View>(9) add following to classes
 - Member Functions
 - Parameters / Return to Member Functions
 - State Diagram for a class – if class has internal states
- <Logical View>(10) add following to class functions – if required
 - Activity Diagram (flowchart) for a Function
 - State Diagram for a Function
 - Sequence Diagram for a Function

UML Approach (Cont..)

- <Logical View>(11) add **Sequence Diagram** to realize each Use-Case by involving all objects designed so far



UML Approach (Cont..)

- <Deployment View>(12) add Deployment Diagram to represent the H/w and S/w artifacts deployment scenario
- <REVIEW>(12) do Design Review based on the guidelines
- <REVIEW>(13) allocate time / resource for **REFACTORING** to improve the design

- Case Study using UML 2.0



Case Study

Next Steps...

References

- Design Patterns:
 - <http://sourcemaking.com/>
 - <http://www.dofactory.com/Default.aspx>
 - <http://www.oodesign.com/>
 - <http://www.c-sharpcorner.com>
- UML:
 - <http://www.uml-diagrams.org/>
 - <http://www.visual-paradigm.com/VPGallery/diagrams/UseCase.html>
 - <http://edn.embarcadero.com/article/31863>
 - http://www.ibm.com/developerworks/views/rational/libraryview.jsp?search_by=UML+basics:
- CPP:
 - www.learnCPP.com
 - [wwwcplusplus.com](http://www.cplusplus.com)
 - C++ Primer : Stanley, Josee and Barbar

Credits

- Books:
 - Head First OOAD & Design patterns
- Web Links:
 - <http://www.c-sharpcorner.com/UploadFile/e881fb/simplest-way-to-learn-object-oriented-programming/>
 - http://www.slideshare.net/nishithshukla/jump-start-to-oop-oad-and-design-pattern?from_search=1
 - <http://www.c-sharpcorner.com/UploadFile/e881fb/object-oriented-programming-using-C-Sharp-part-10/>
 - <http://www.uml-diagrams.org>
 - <http://www.visual-paradigm.com/support/documents>

Pattern Reference Cards



Reference Card



Reference Card
(Short)

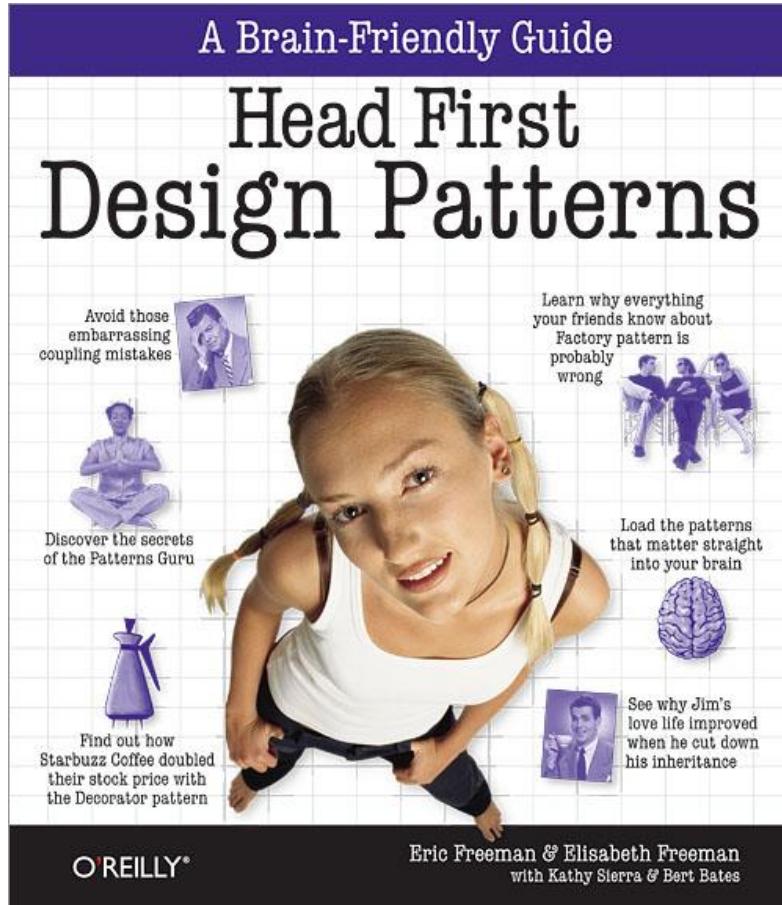
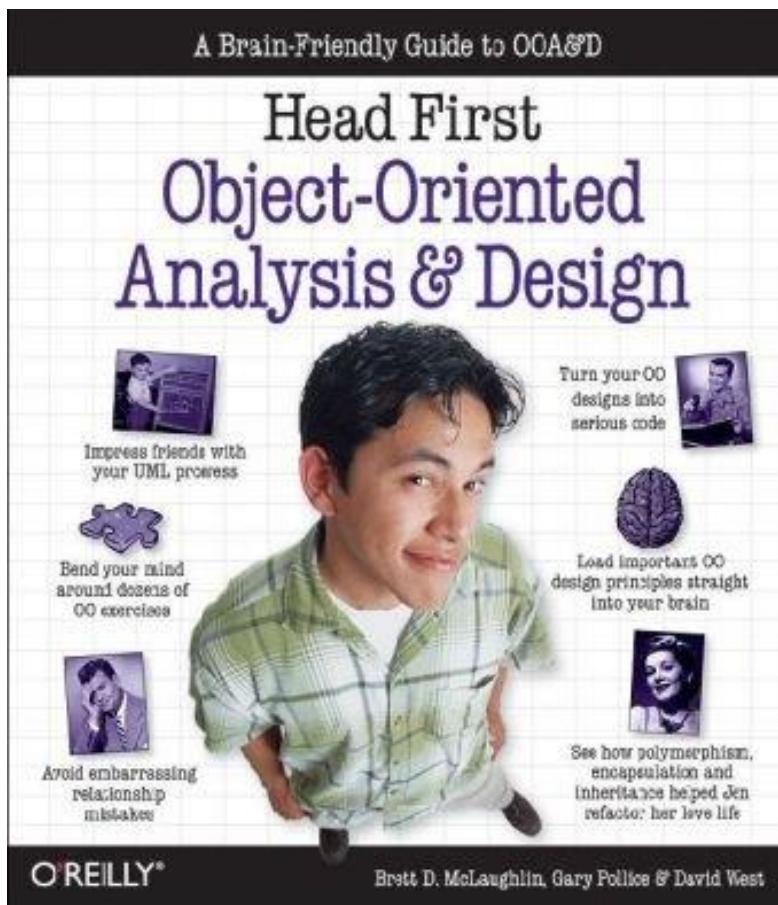
Books (Must Have)



Design Patterns: Elements of Reusable Object-Oriented Software

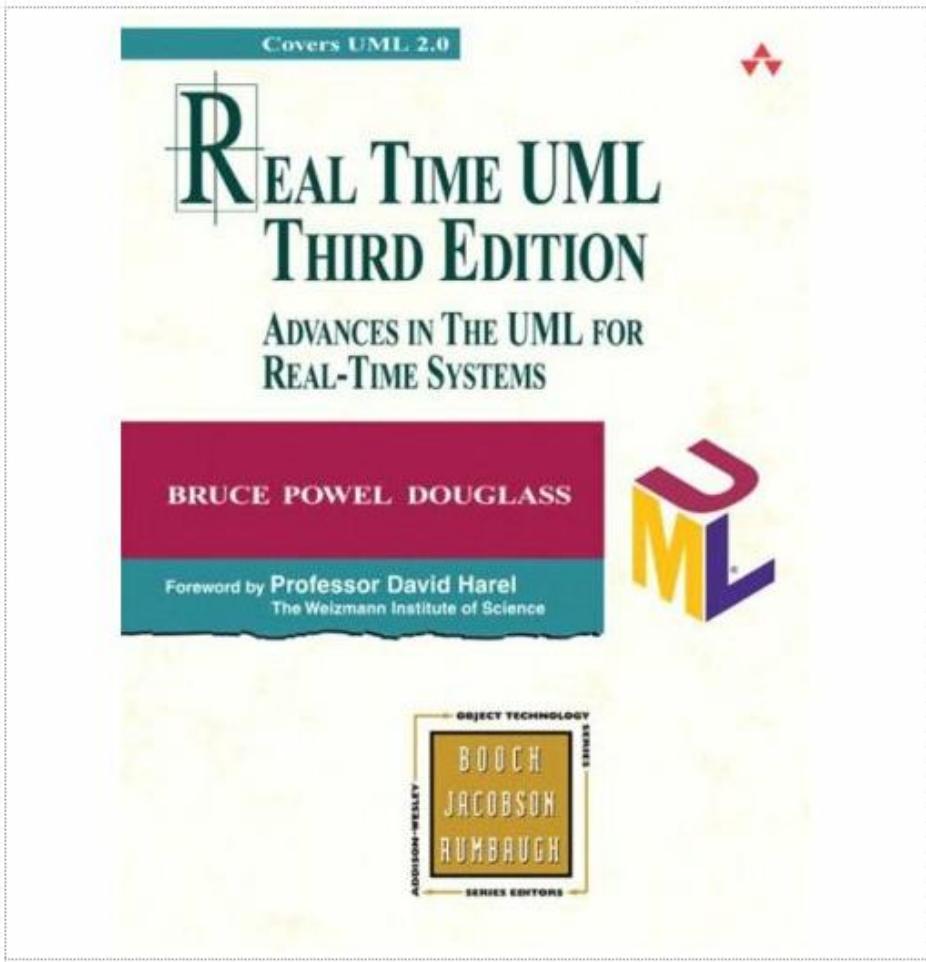
by [Erich Gamma](#), [Richard Helm](#), [Ralph Johnson](#), [John Vlissides](#)

Books (Must Have)

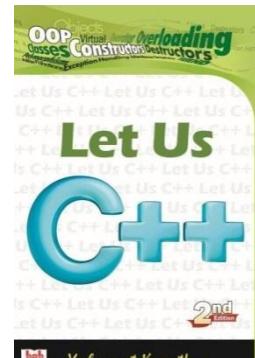
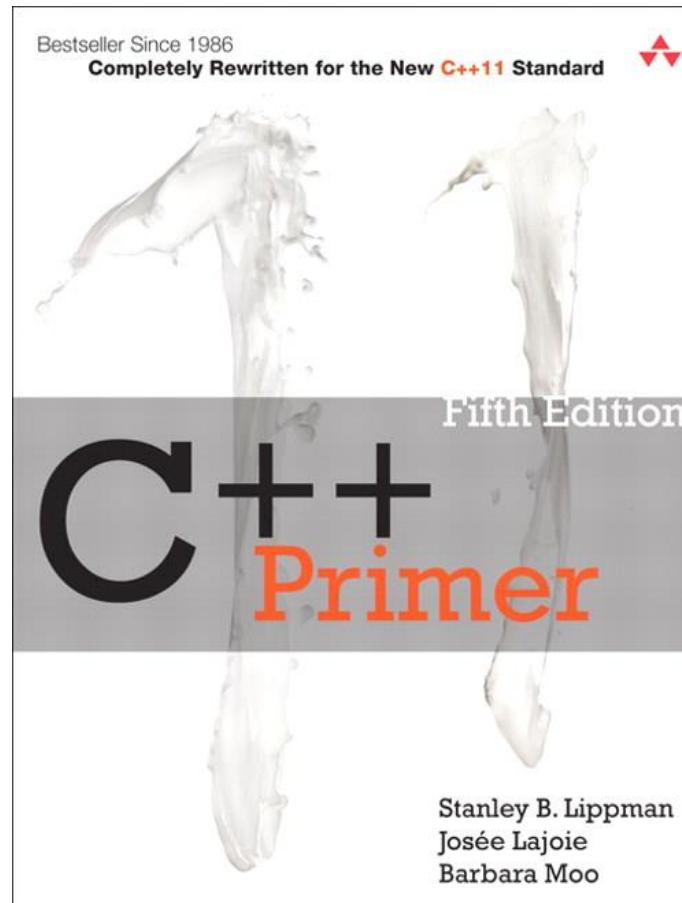
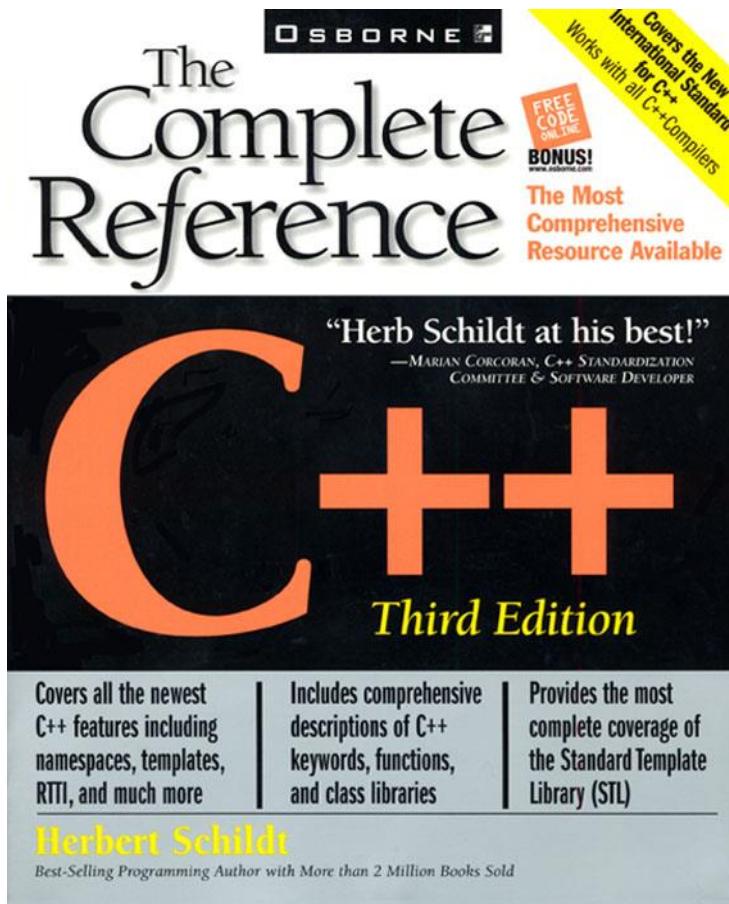


Books (Must Have)

Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)



Books (Must Have)



Yashavant Kanetkar

Thank You

