JUMP START

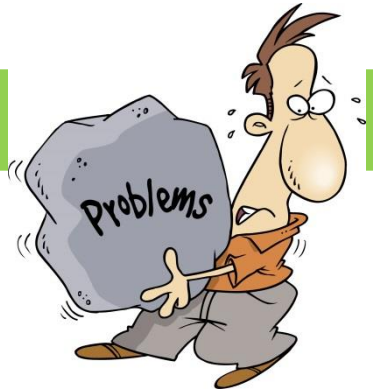
TO

OOAD
&
DESIGN PATTERNS
(using C++)

Balaji Haridass

Why OOP?

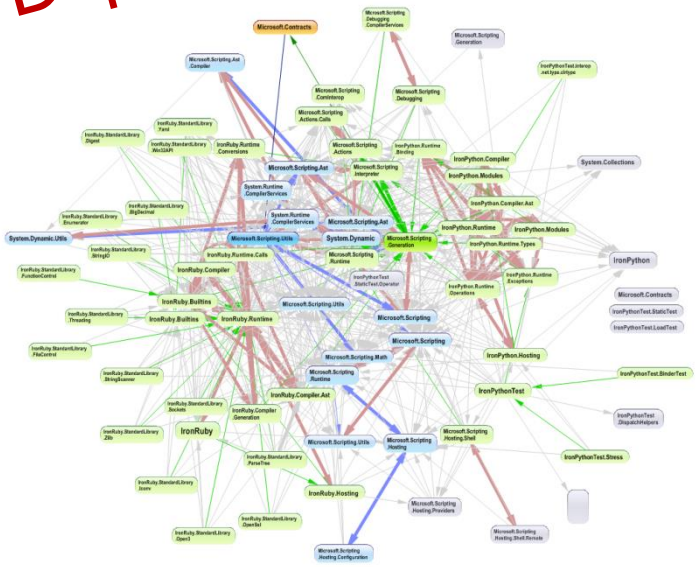
Development/Design Problems?



Problems we all have!

Not Reusable!

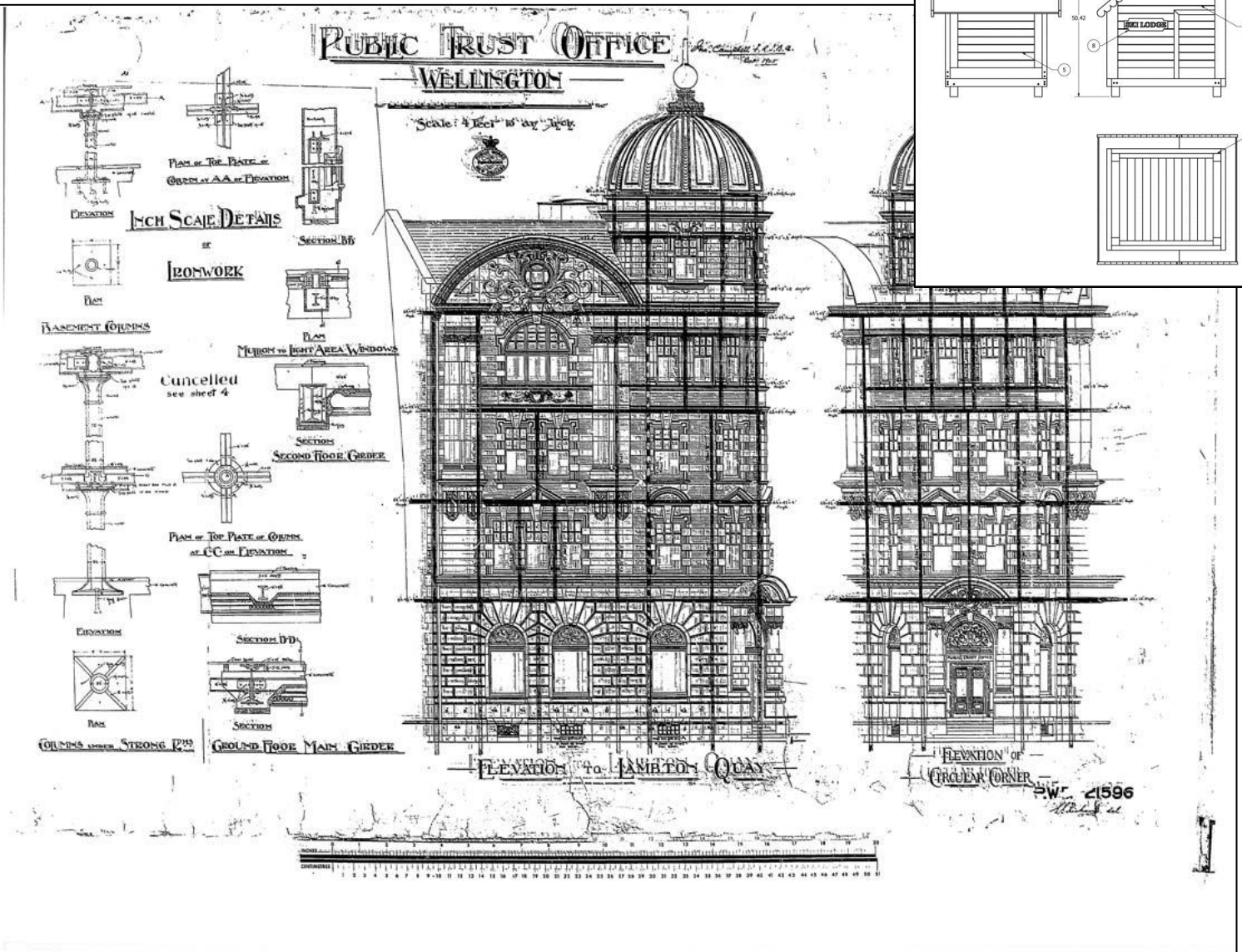
Dependency!

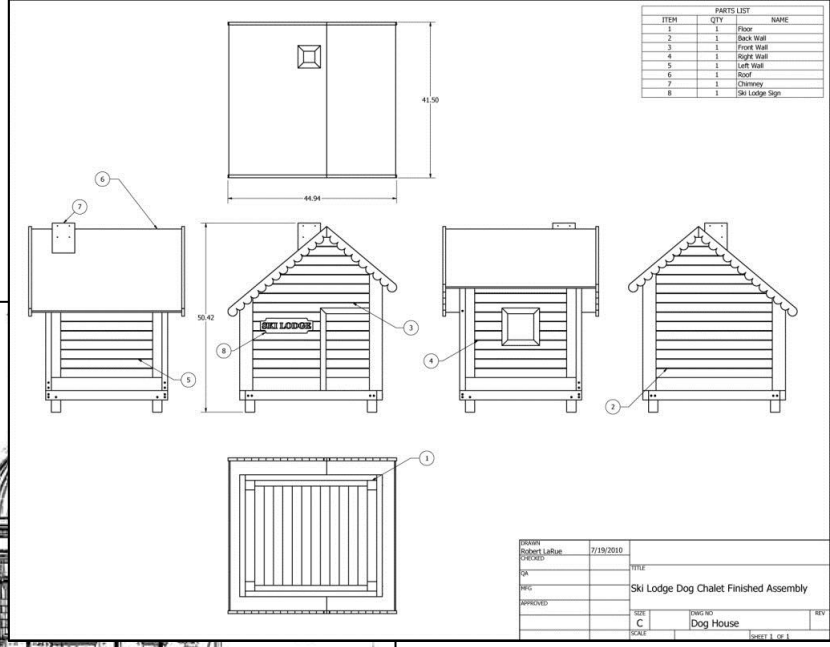


Defects!

Testable?

Problems we all have!





Complexity

Take Away

Will be able to Design that

- Can be RE-USED
- Will be EXTENDIBLE
- Will be FLEXIBLE T0 CHANGE
- Will be TESTABLE
- Will be UNDERSTABLE

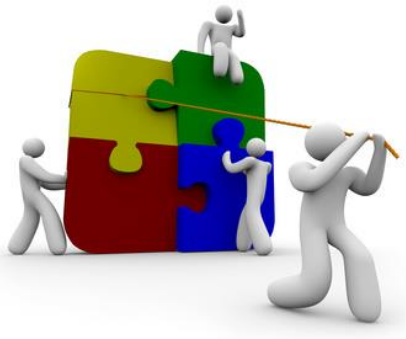
Who all will you consider while doing Design, Development, Review?

Who should we care about?

OOA&D is about writing great software!

Customer is satisfied when their APPS

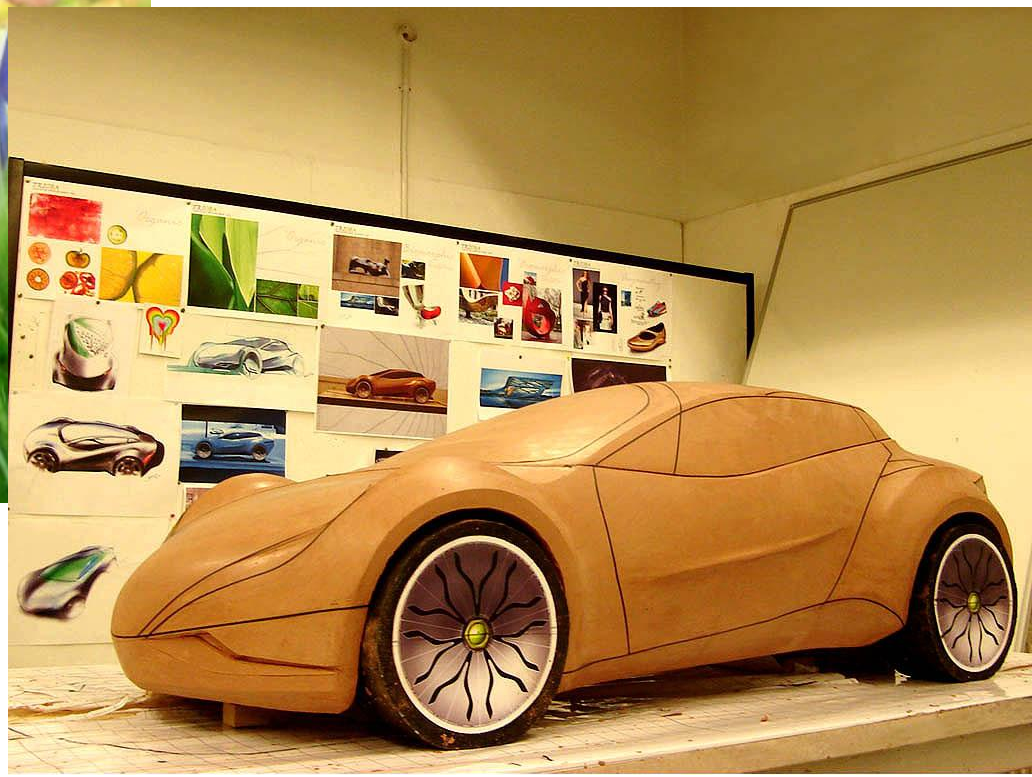
- WORK
- KEEP WORKING
- CAN BE UPGRADED



Programmer is satisfied when their APPS

- RE-USED
- FLEXIBLE
- UNDERSTANDABLE
- TESTABLE

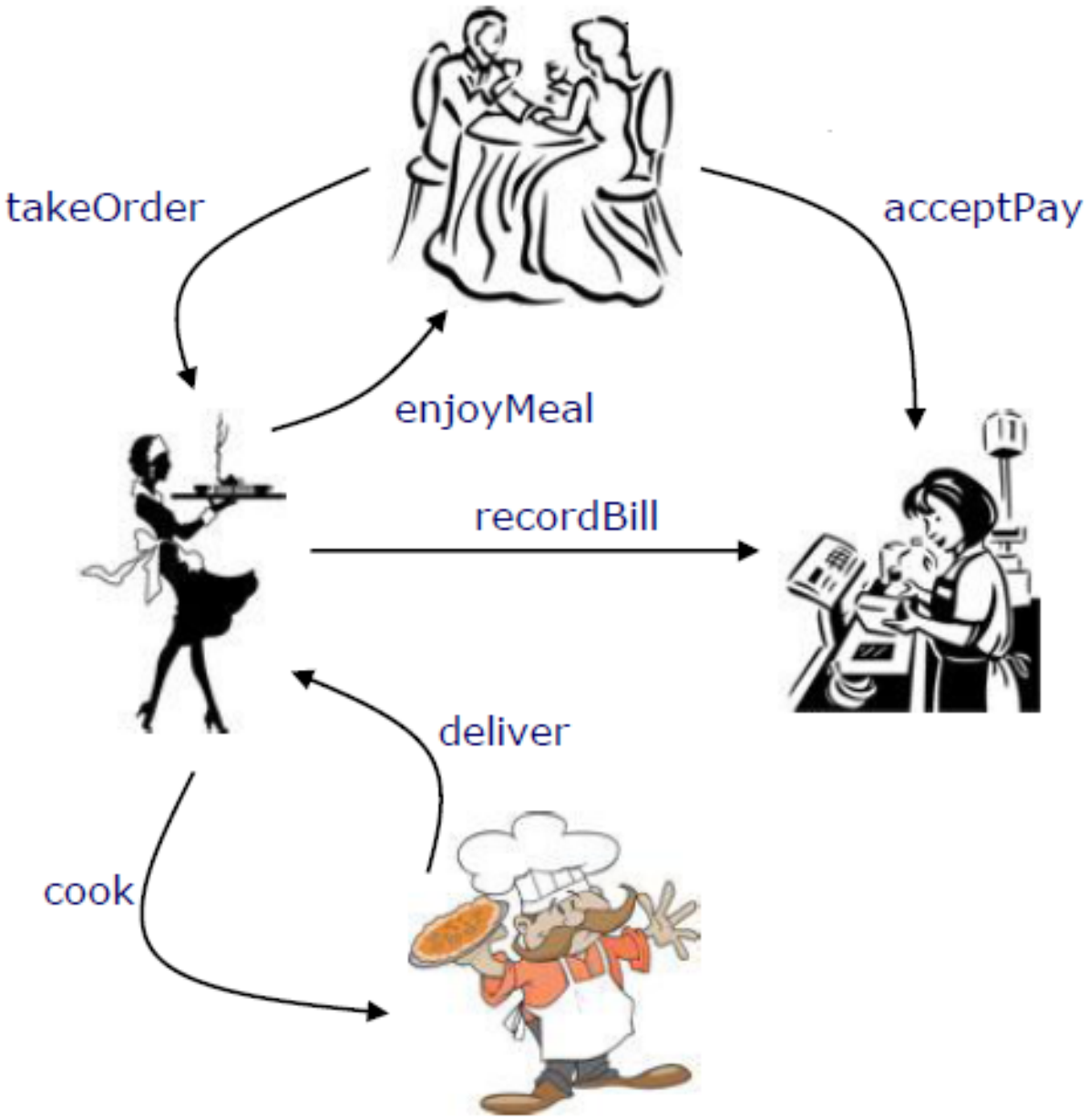
Why OOAD?



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

Encapsulation

Inheritance

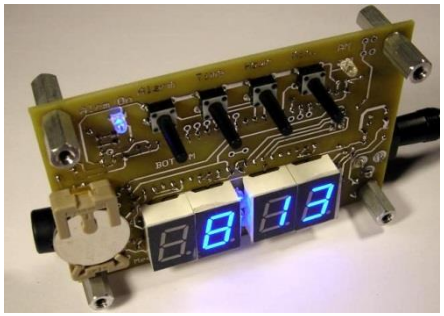
Polymorphism

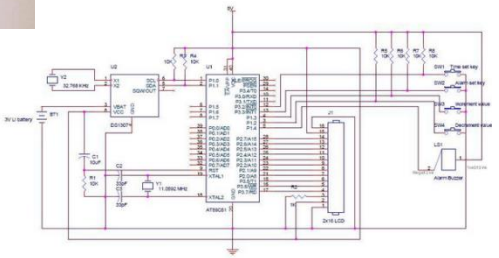
Battery! What is your perspective?

How about perspective of user, clock & battery manufacturer?

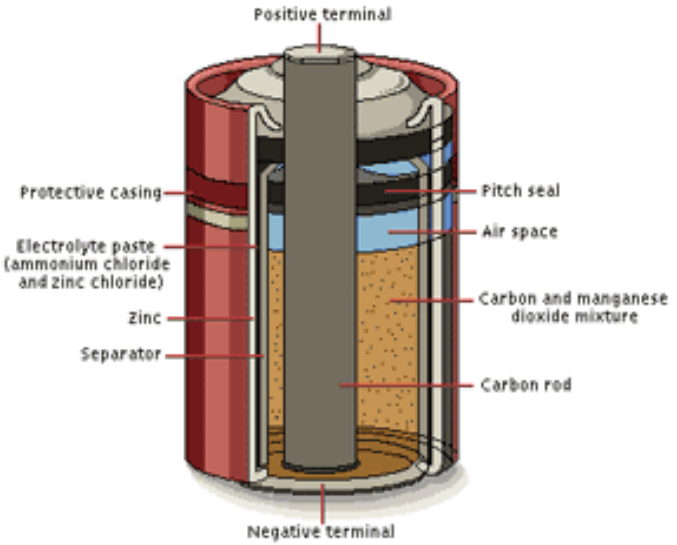
Separation of Concerns

Alarm Manufacturer





Battery Manufacturer



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

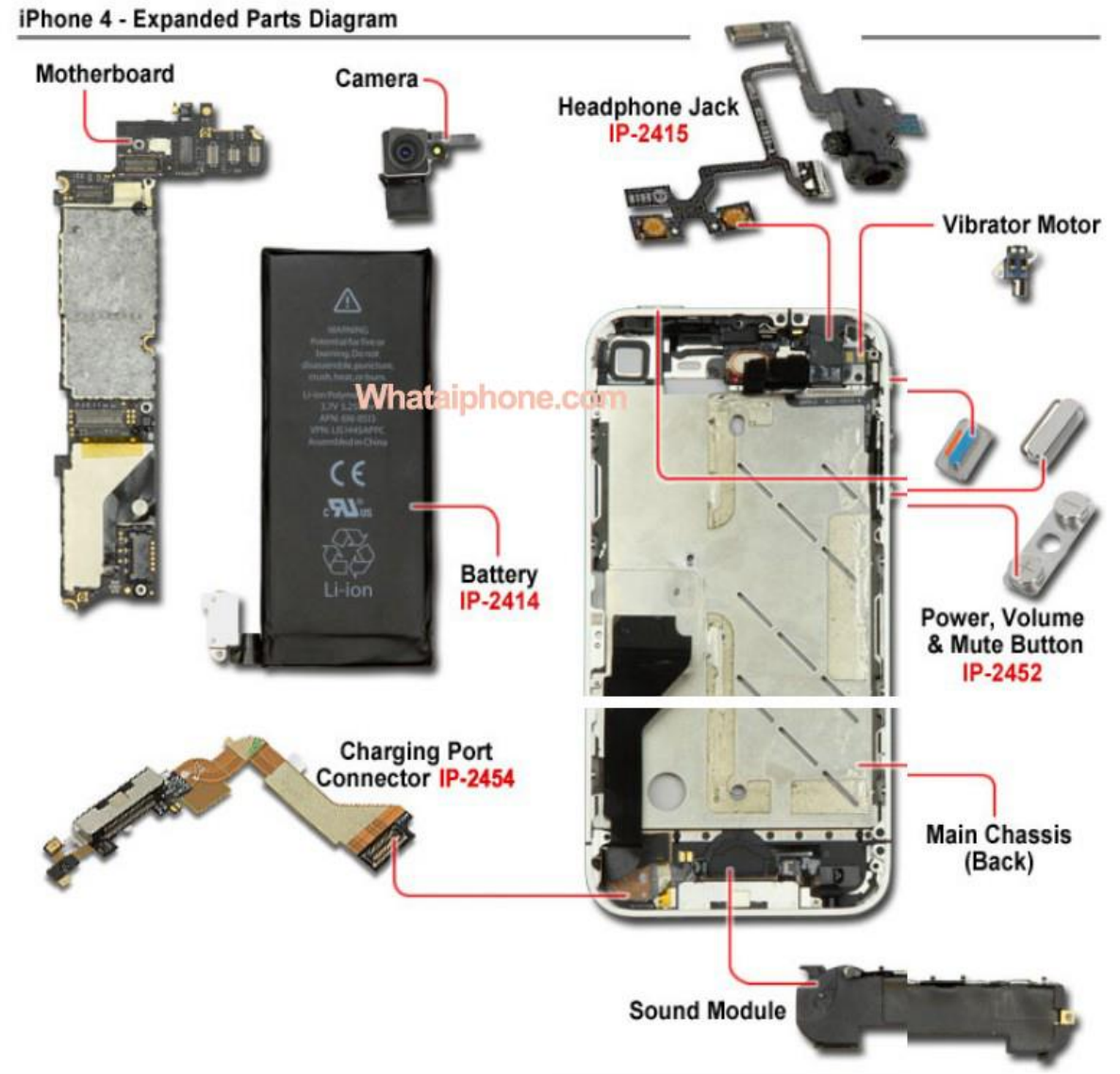
Develop Component in Parts

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)

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

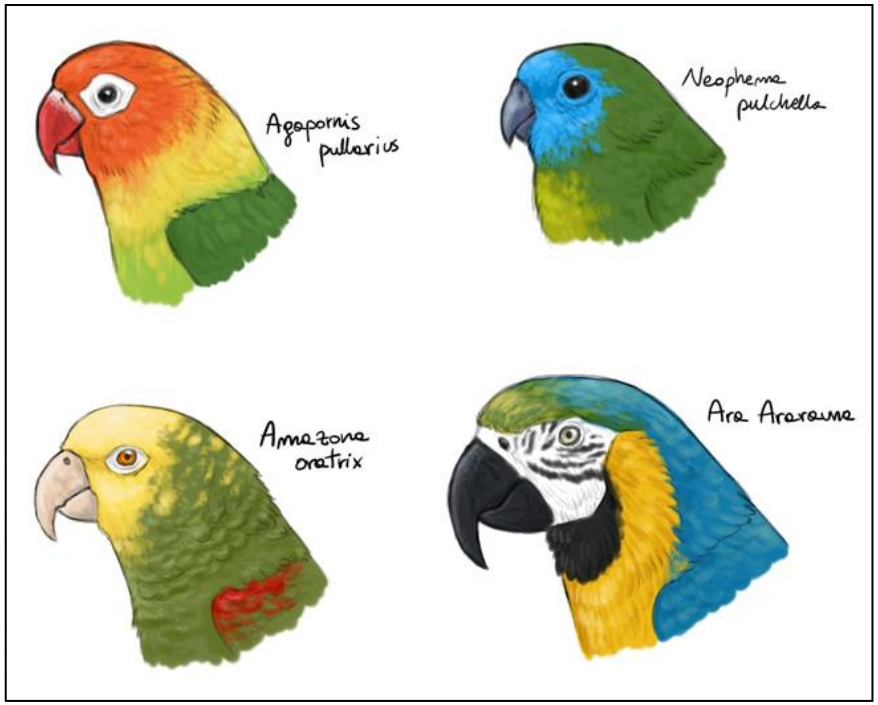
Class

Attributes

Services

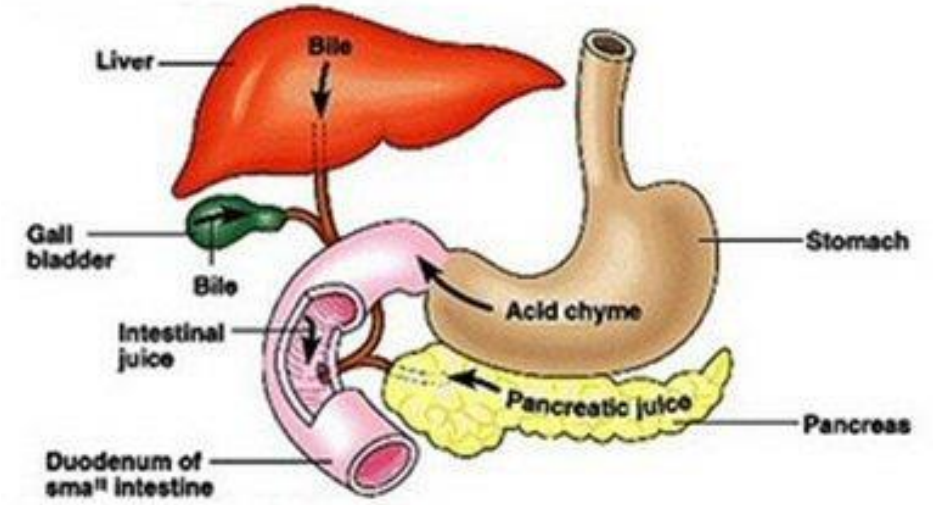
Parrot
-eyeColor: Color -beakColor: Color -eyeType: EyeType -bodyPattern: ColorPattern +Eat() +Fly() +Sleep()

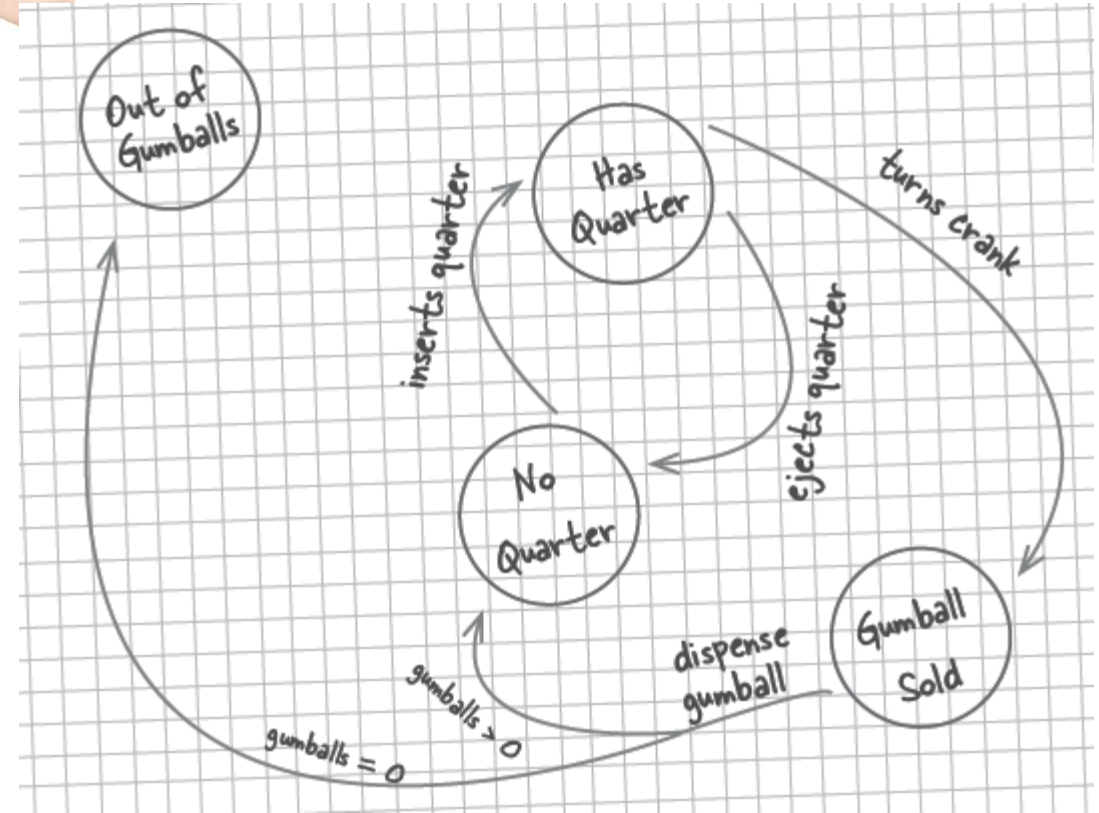
Multiple instance of same type



Objects / Instances

Objects/Systems have States





Life & OOP

Inheritance ,
Constructor

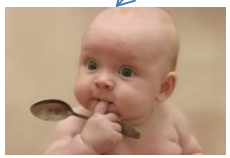
Birth

Polymorphism

Life

Destructor

Death



Life & OOP

Inheritance , Constructor

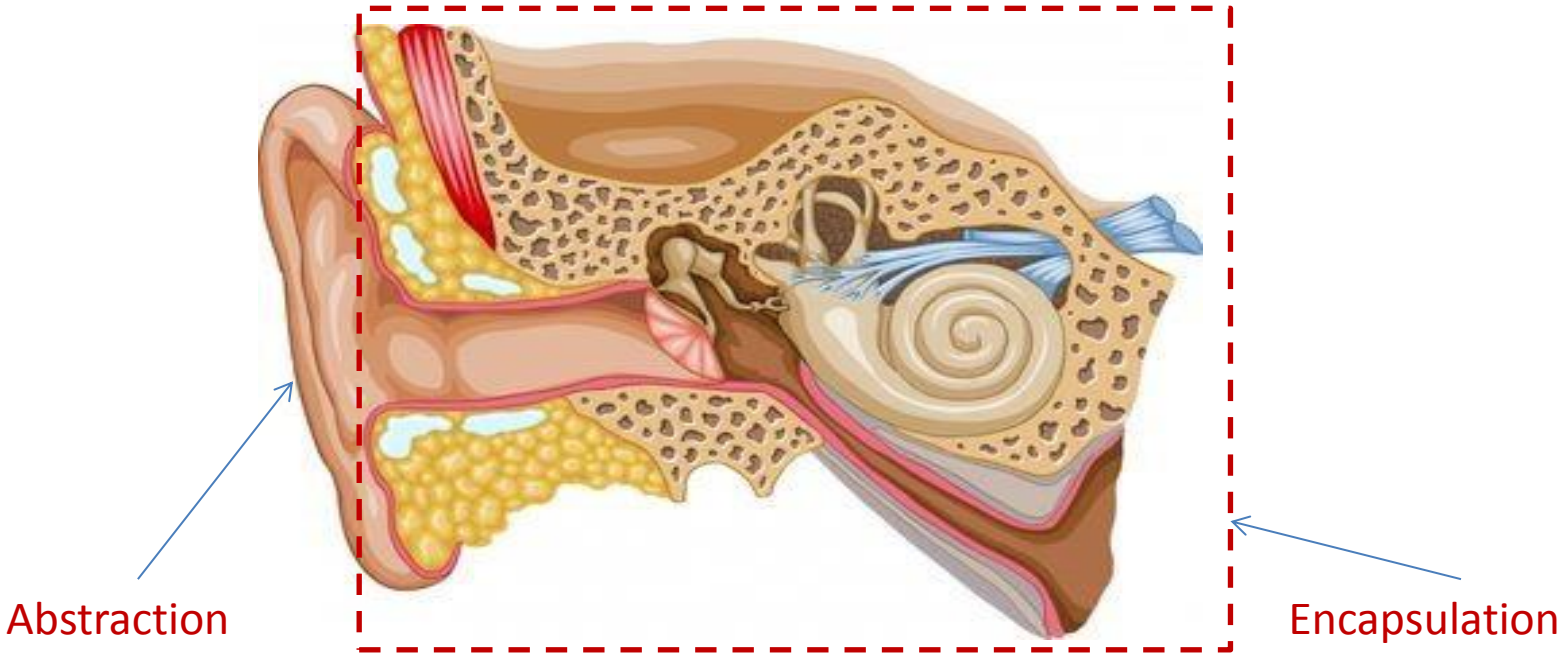
Birth

True - Polymorphism

Destructor

Death

Life & OOP



- 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

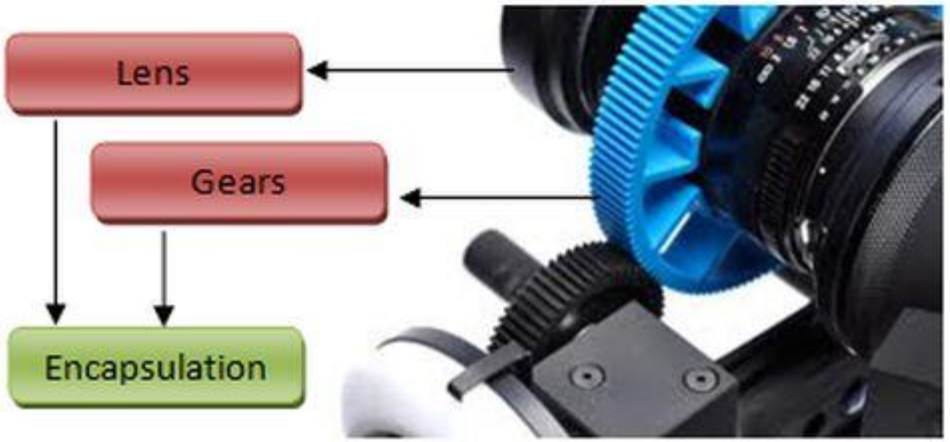
Life & OOP

Abstraction & Encapsulation

Abstraction

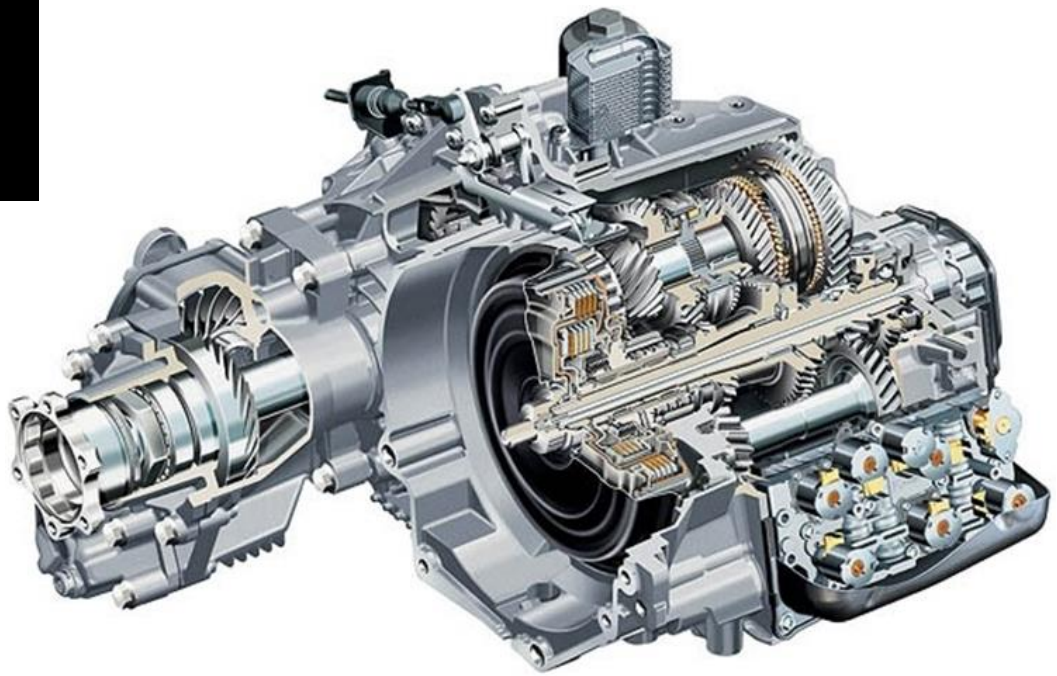
"Abstraction is achieved through encapsulation"

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



Encapsulation

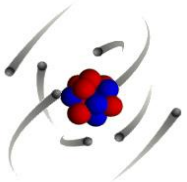
Abstraction



Encapsulation

Abstraction In Electronics

Nature



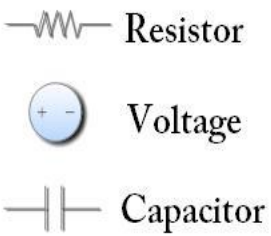
I (A)	V(V)
0	0
1	0.8
2	1.5
3	2.1
4	2.6
5	3
6	3.3
7	3.6
8	3.8
9	3.9
10	4

Laws

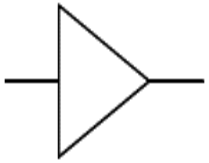
$$V = IR$$

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

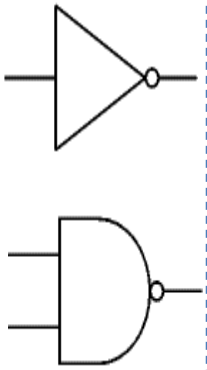
Lumped circuit abstractions



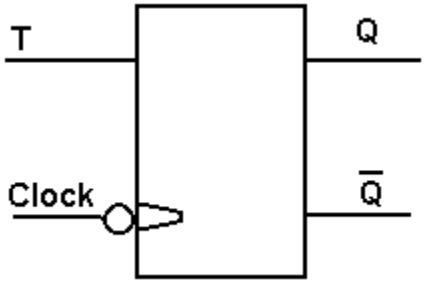
Amps



Digital



Time Functions

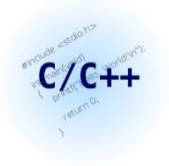


Instruction Set

ARM,
x86

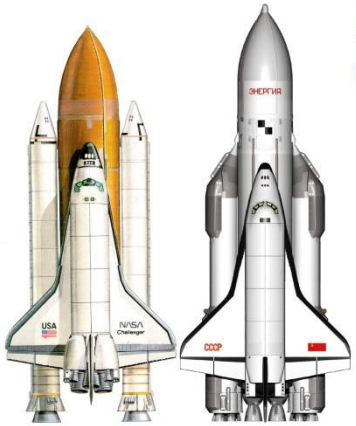
```
Assembly Language
      Code

LOAD   Y
ADD    X
STORE  Y
```

Language



Software Systems

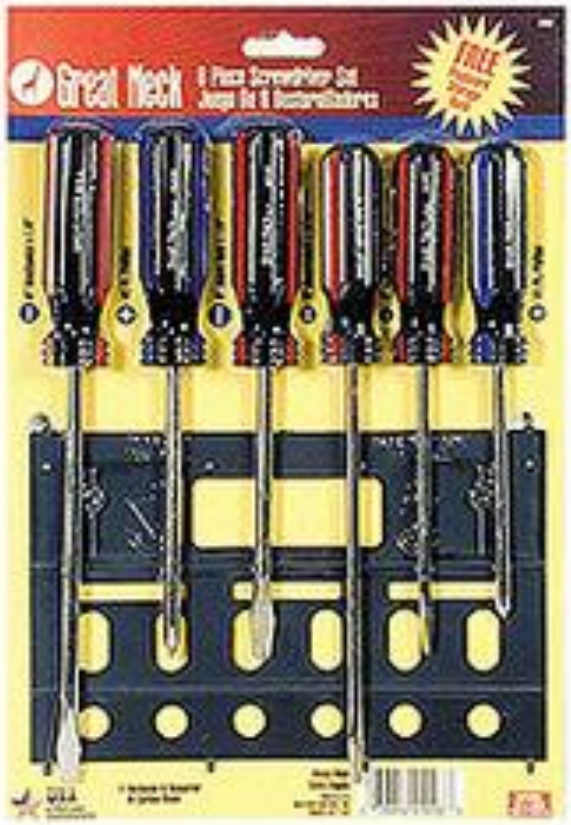
Useful things



Abstraction simplifies things, makes life easier

True Polymorphism

Which ONE do you prefer?

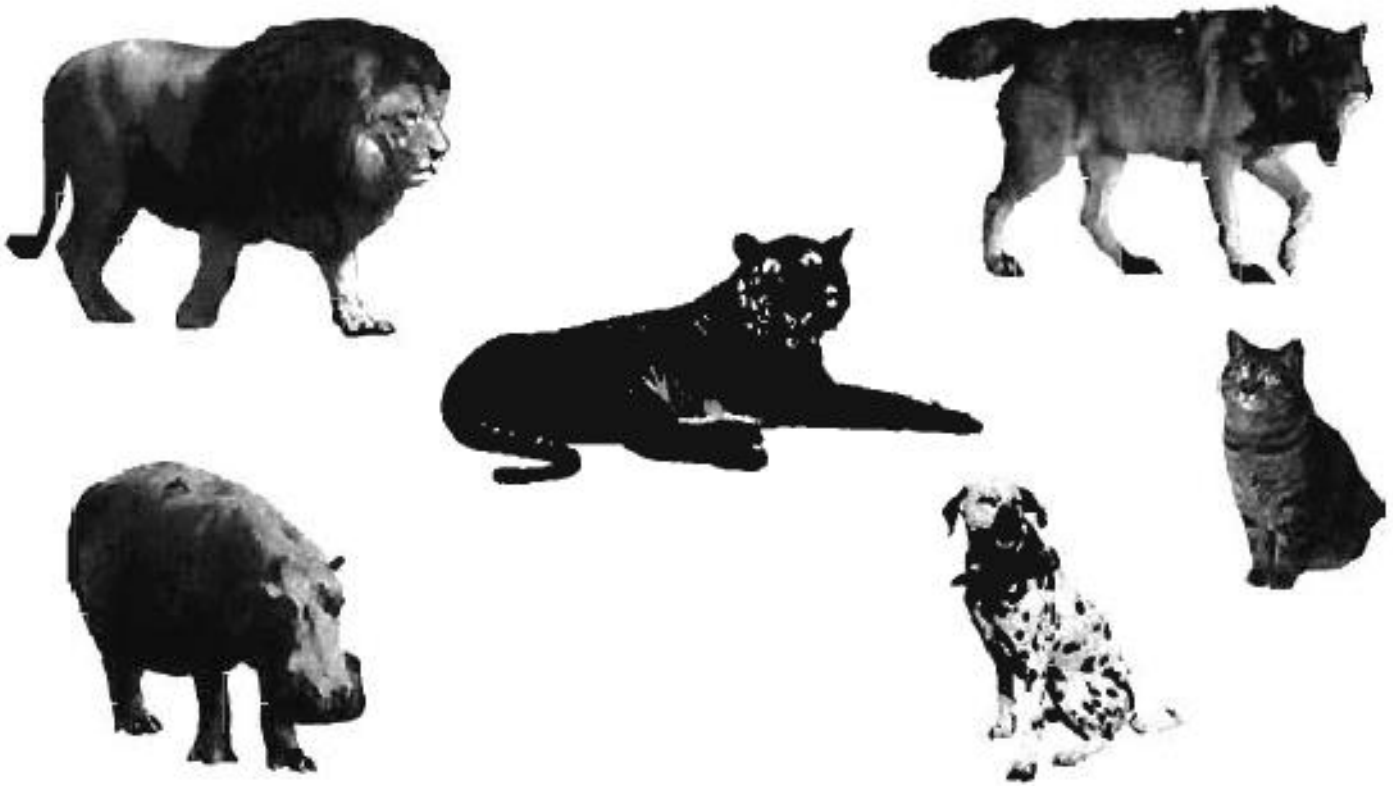


Problem Statement

Design a Game with Weapon Switching Capability

Design Inheritance Tree

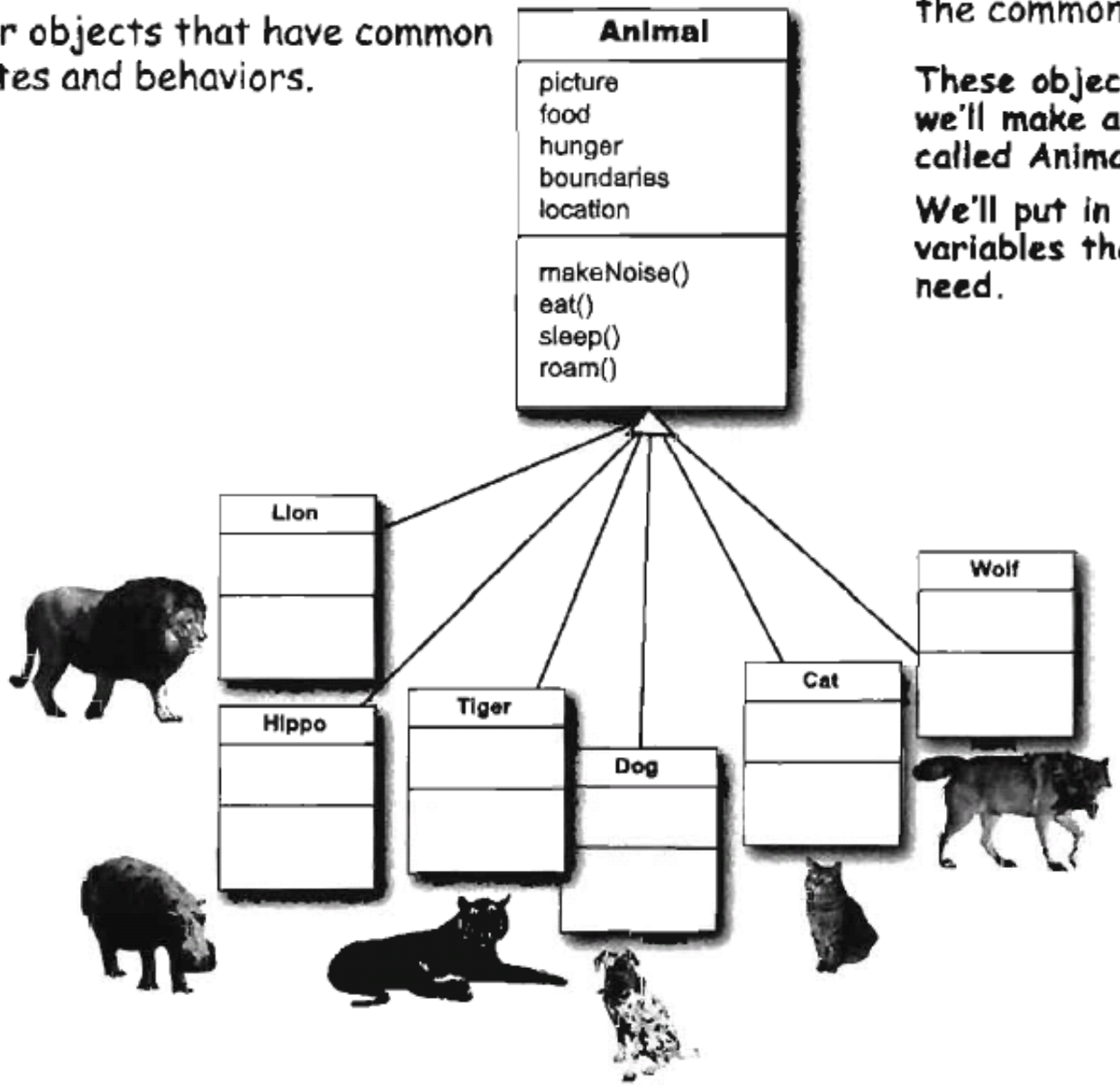
Let's design the inheritance tree for an Animal simulation program



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.

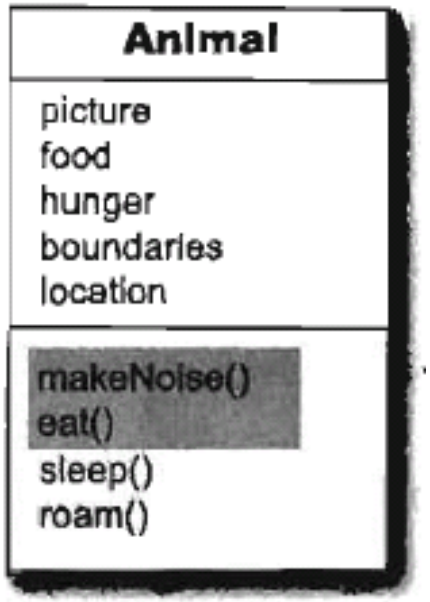


Design a class that represents the common state and behavior.

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

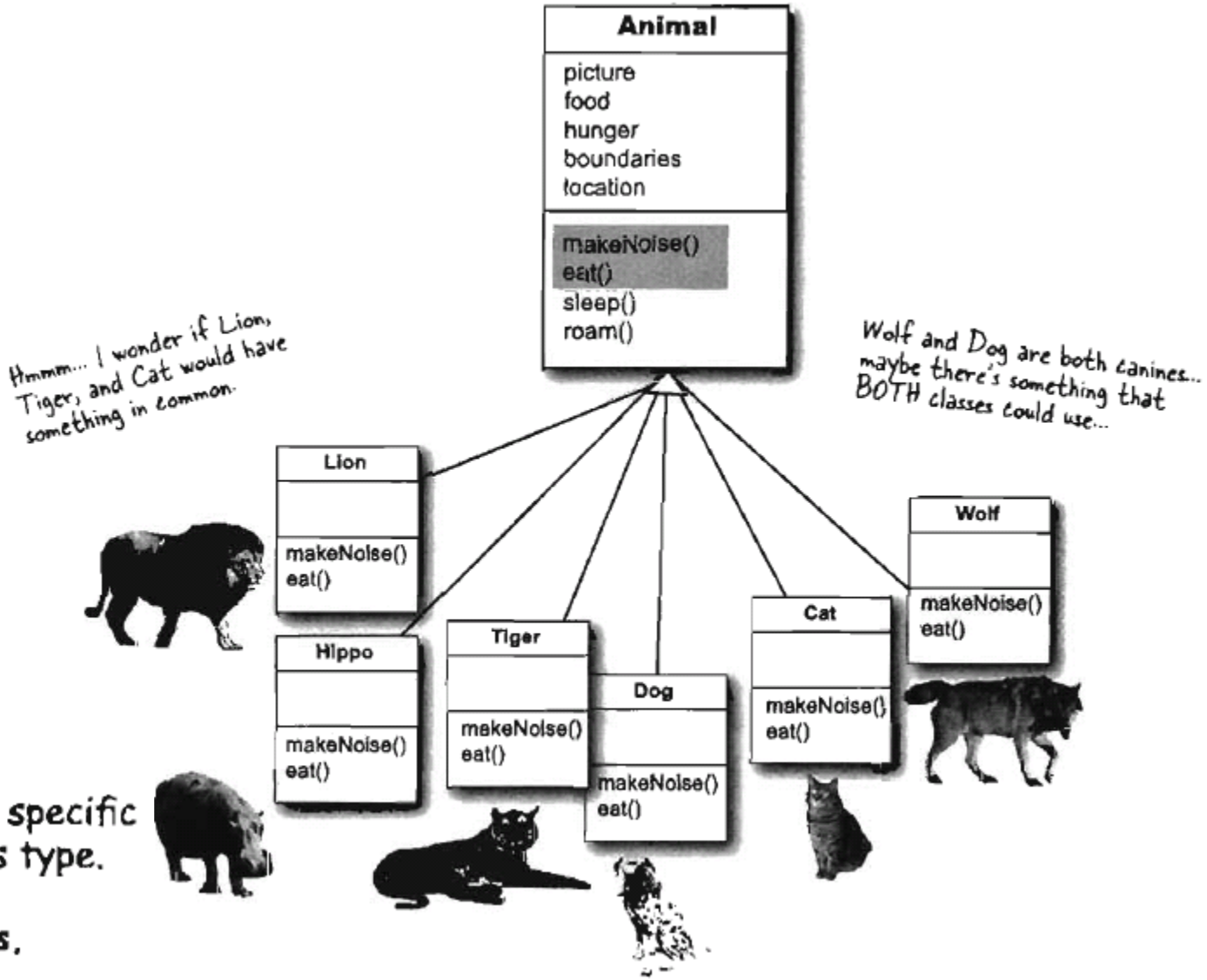
We'll put in methods and instance variables that all animals might need.

Which Methods to override?



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.

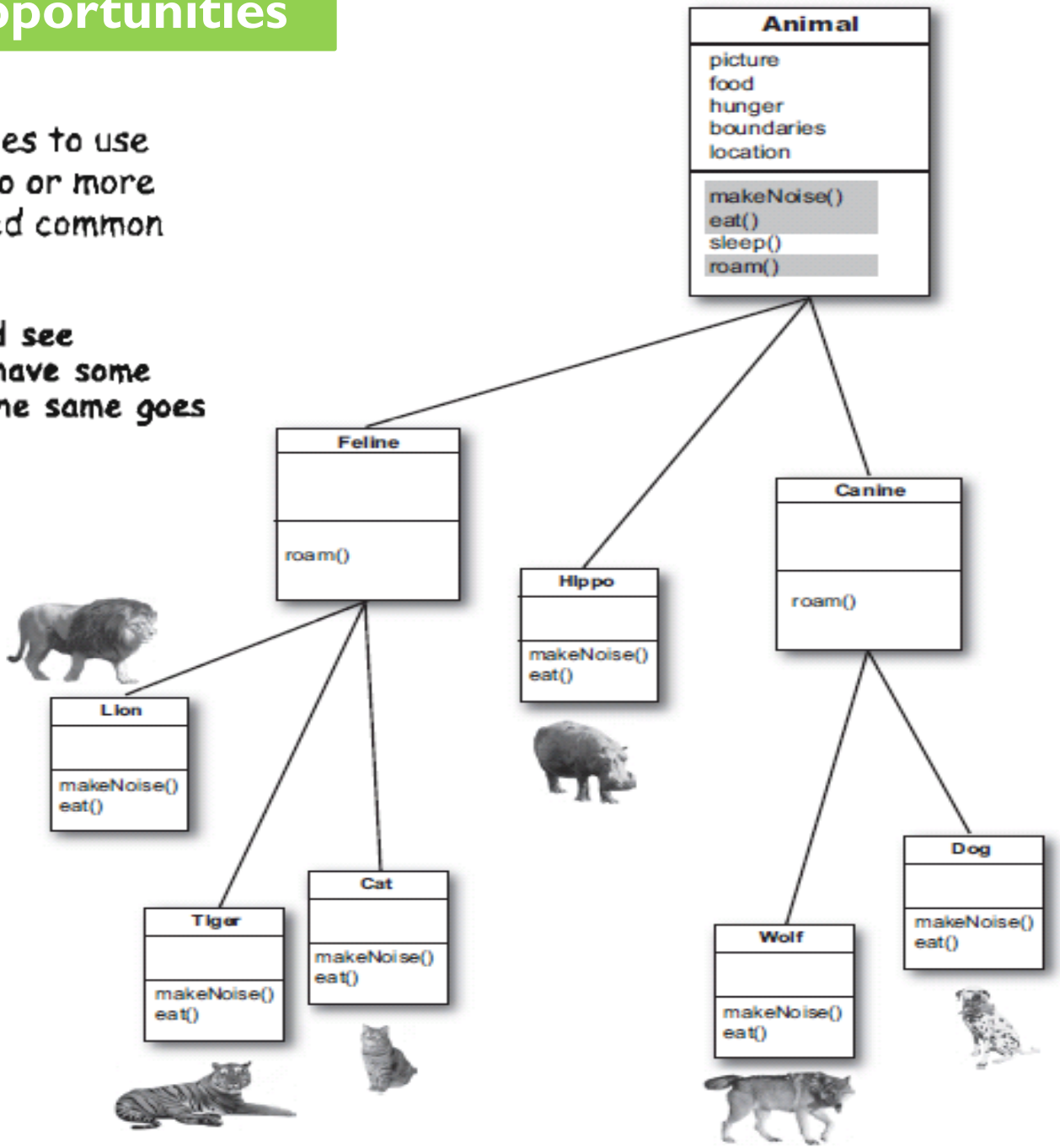


Do all animals eat the same way?

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.



IS-A Relationship

Which method is called?

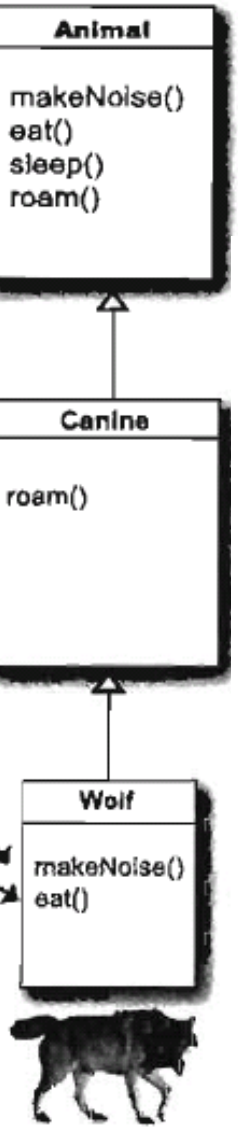
make a new Wolf object — `Wolf w;`

calls the version in Wolf — `w.makeNoise();`

calls the version in Canine — `w.roam();`

calls the version in Wolf — `w.eat();`

calls the version in Animal — `w.sleep();`



Canine extends Animal

Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

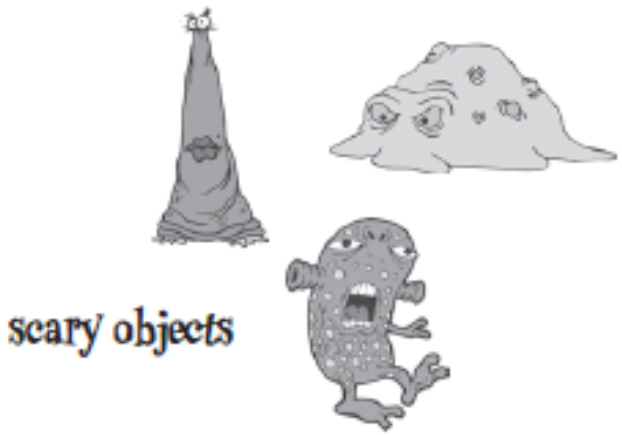
Wolf IS-A Animal

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?



Animal Object??

Some classes just should *not* be instantiated!

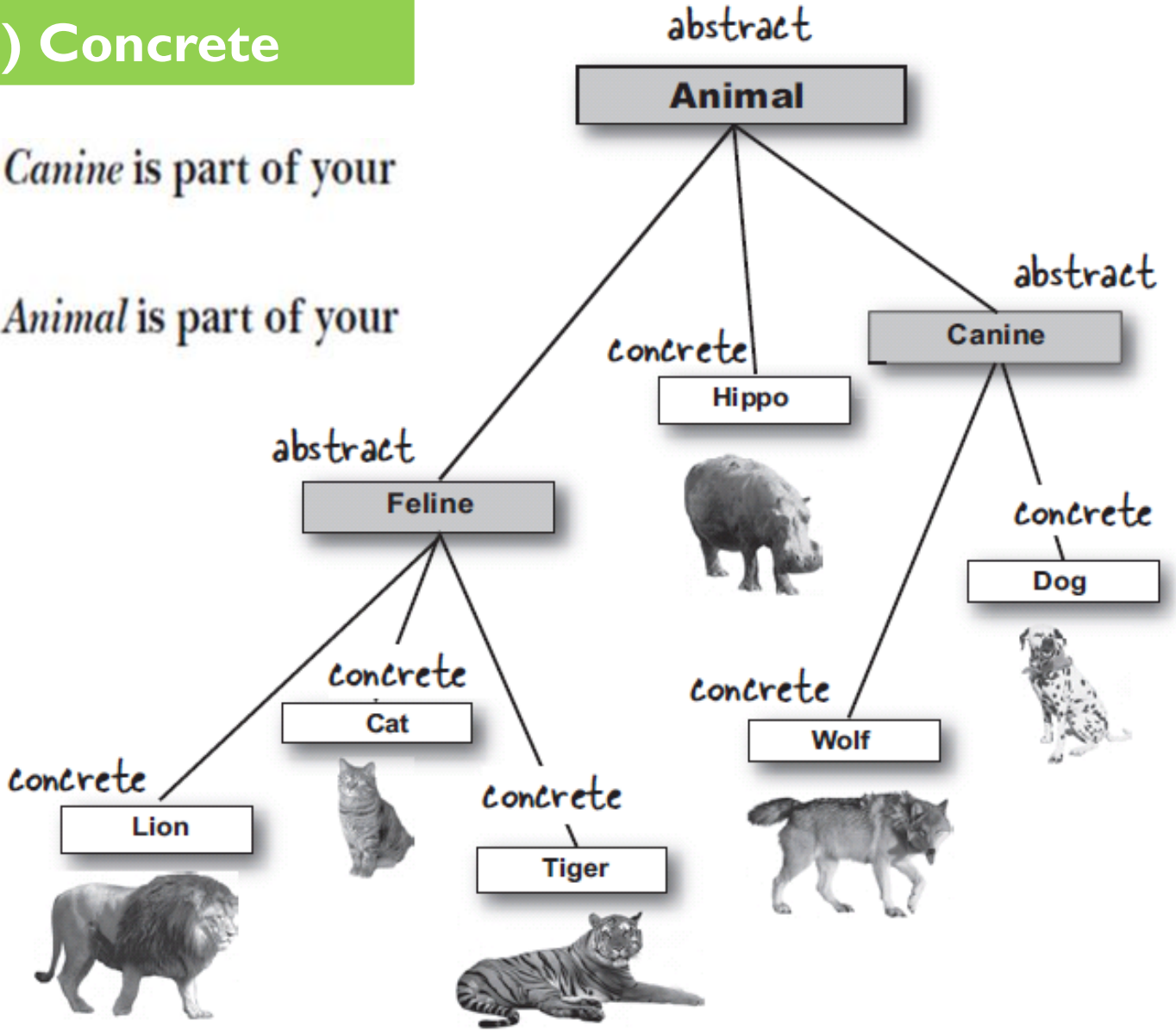
An **abstract class** has virtually* no use, no value, no purpose in life, unless it is ***extended.***

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.



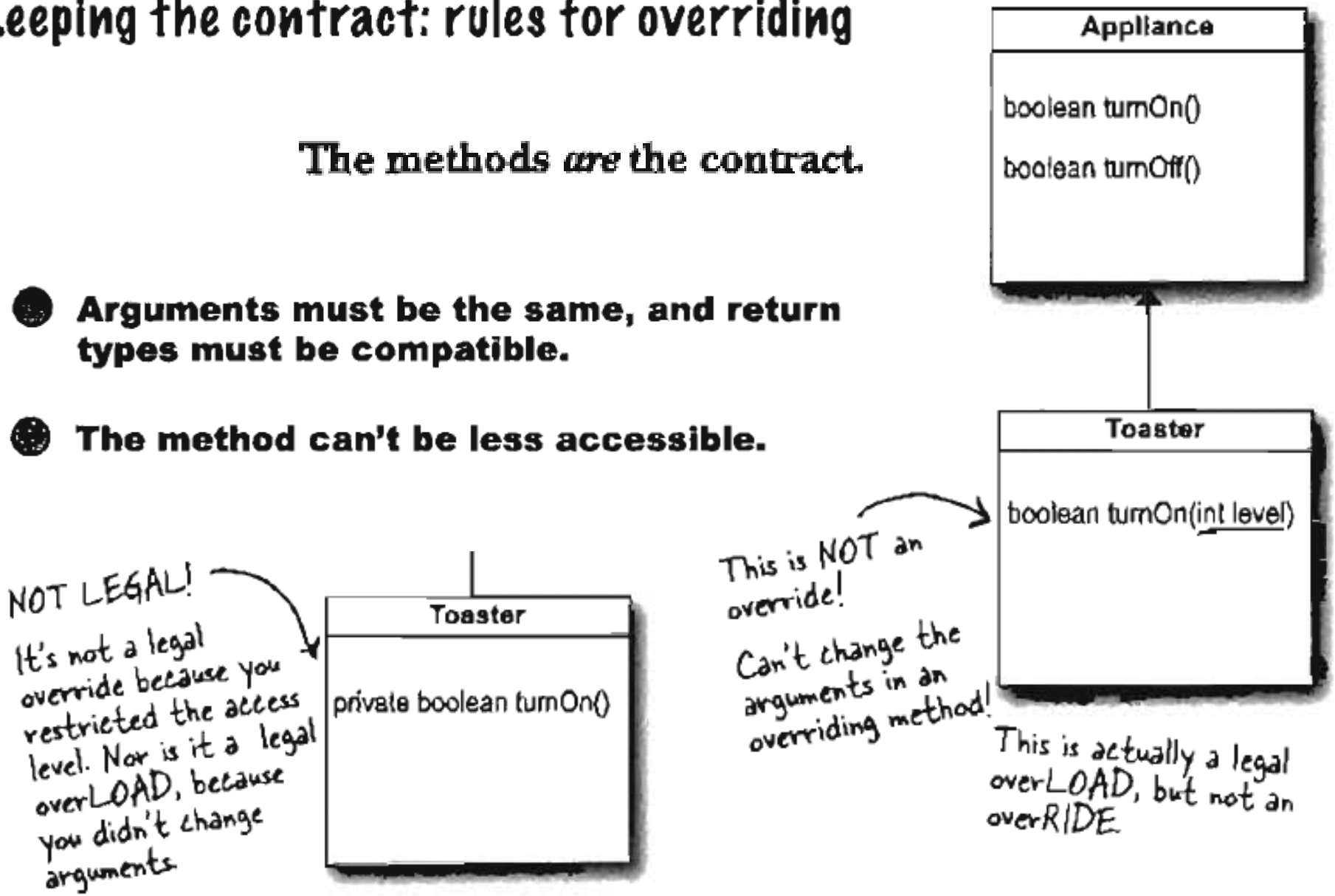
The compiler won't let you instantiate an abstract class

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.**

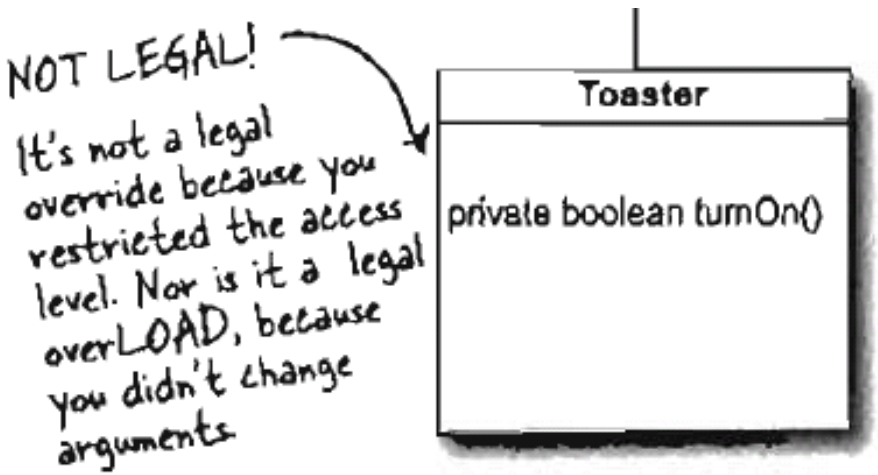


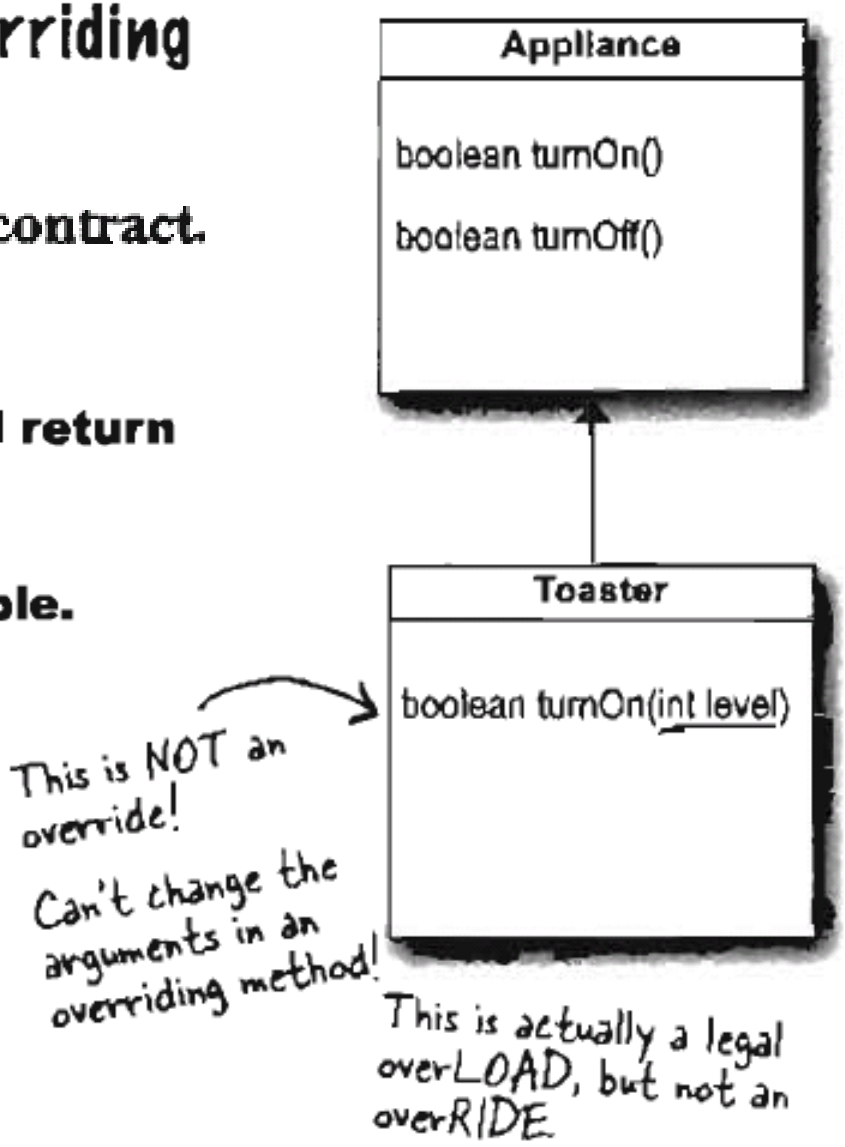
Contract

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.**





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

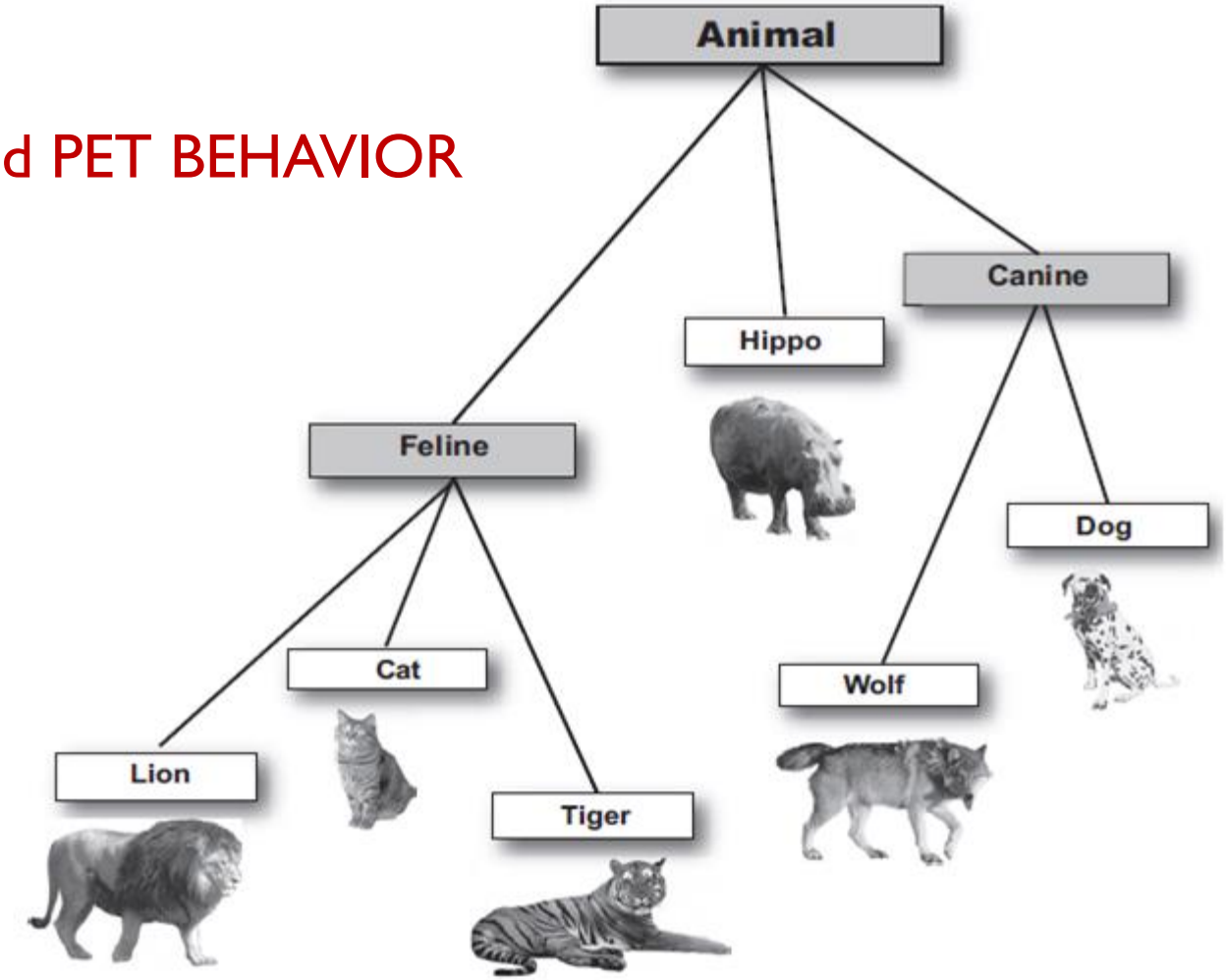
An abstract method has no body!

You MUST implement all abstract methods

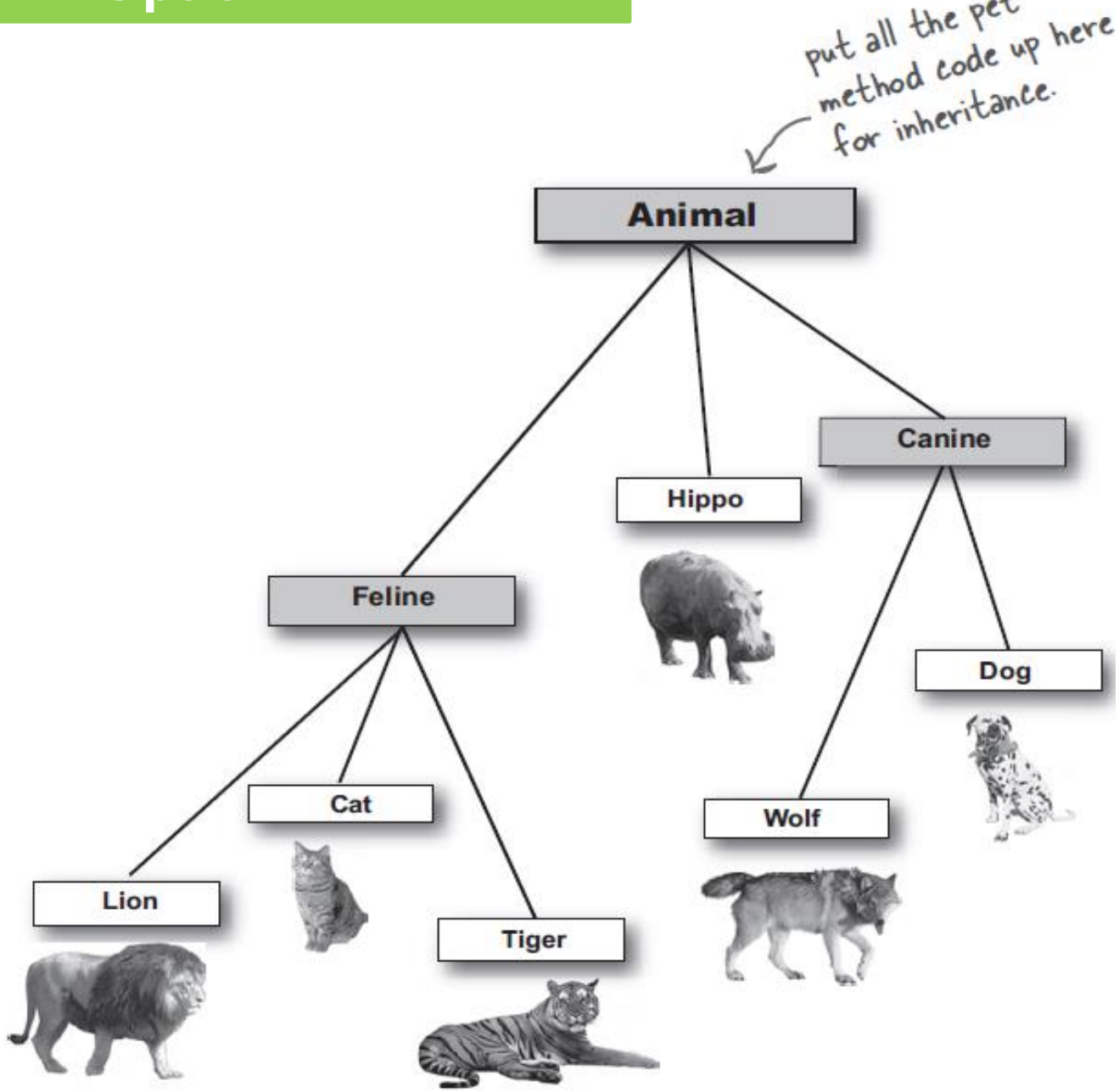
Implementing **an abstract method is just like** ***overriding*** **a method.**

Spec Change

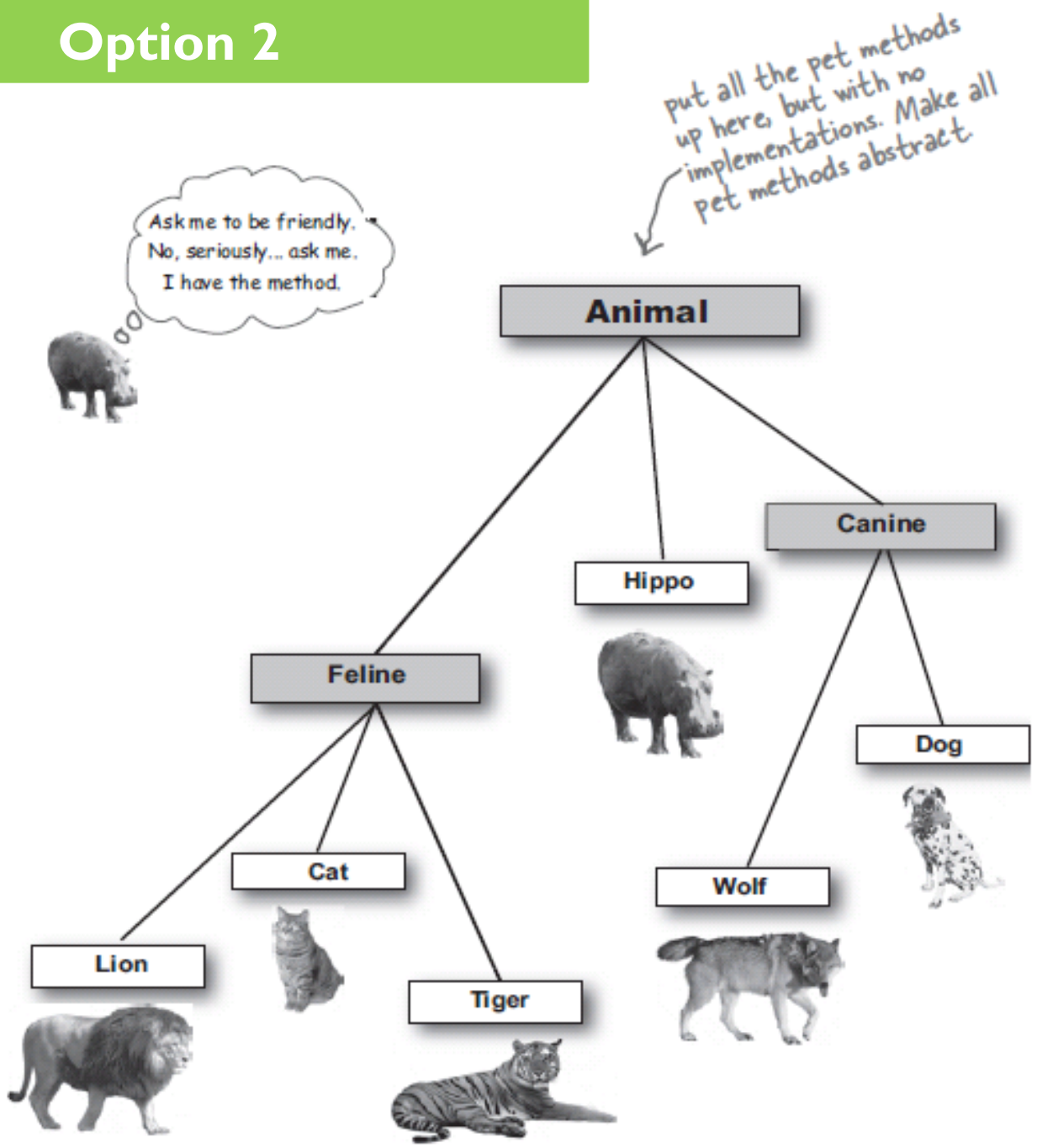
What will you do to add PET BEHAVIOR to Dog?



Option 1

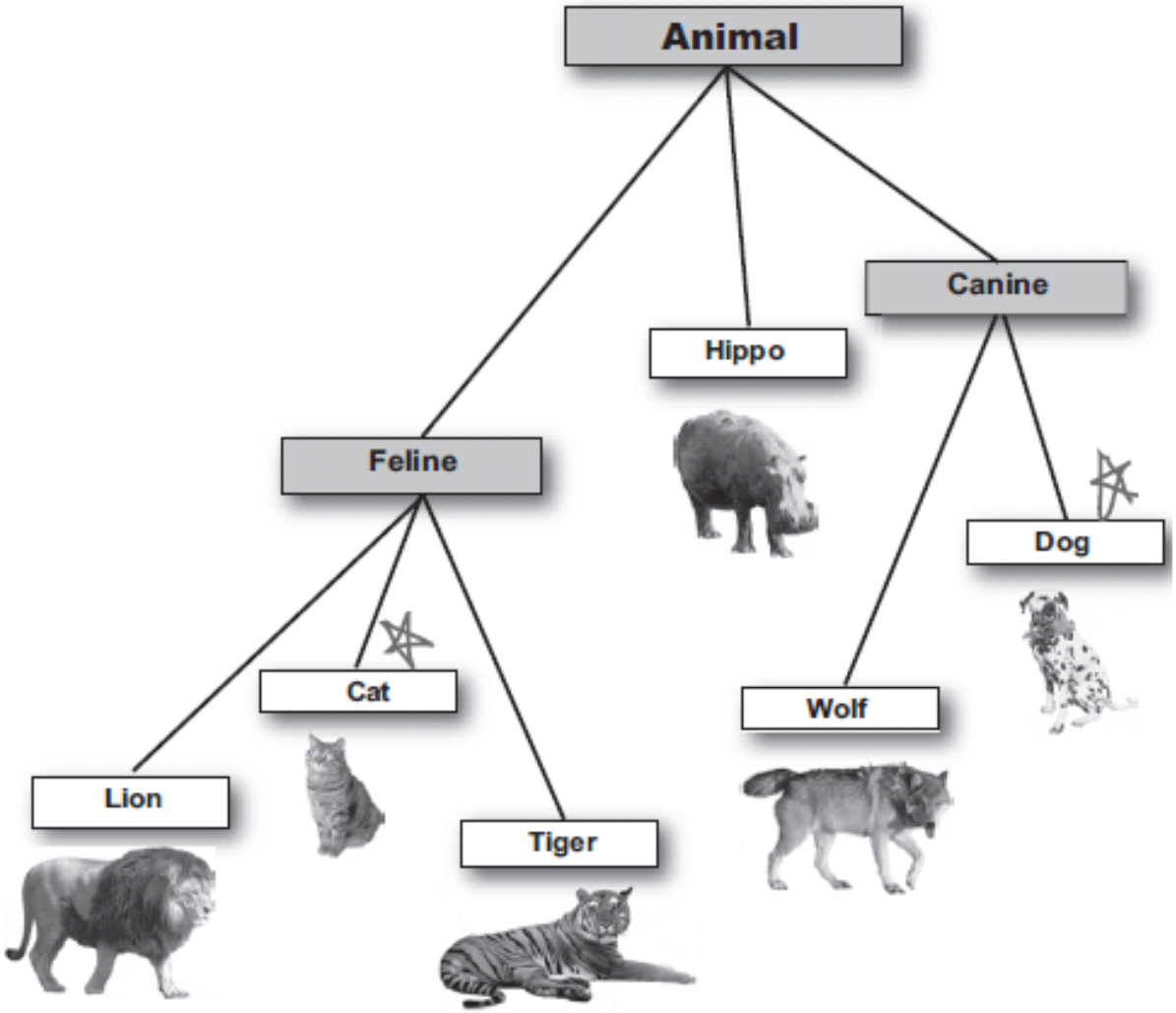

put all the pet method code up here for inheritance.
Animal
Feline
Hippo
Canine
Cat
Dog
Lion
Tiger
Wolf

Option 2

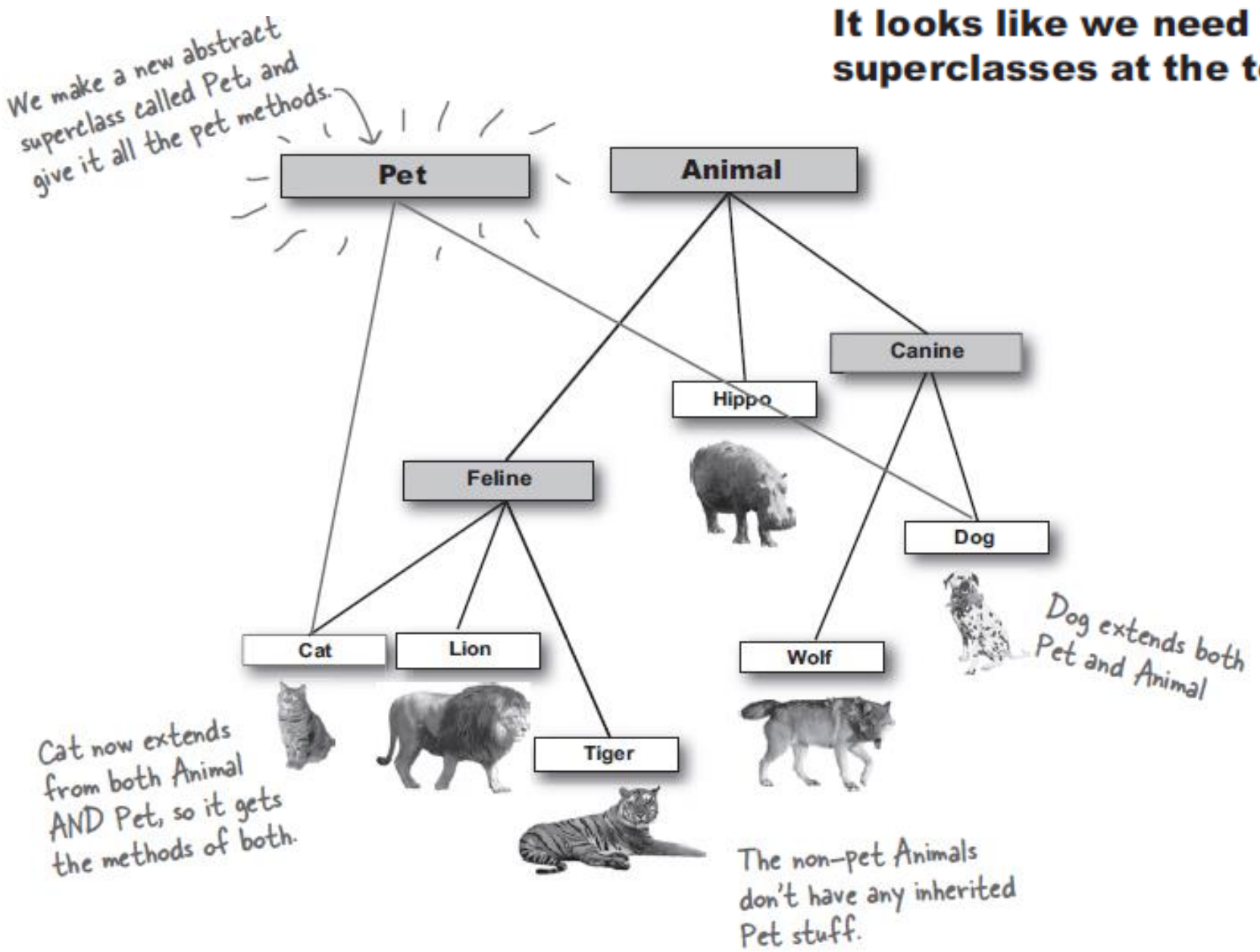

Ask me to be friendly.
No, seriously... ask me.
I have the method.
put all the pet methods up here, but with no implementations. Make all pet methods abstract
Animal
Hippo
Canine
Feline
Dog
Cat
Wolf
Lion
Tiger

Option 3

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



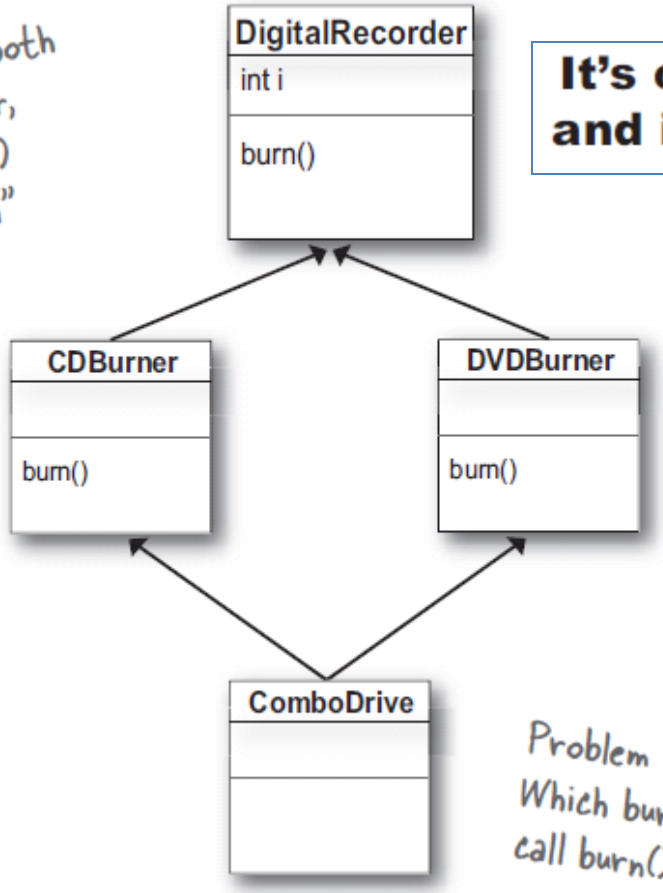
Option 4


It looks like we need TWO superclasses at the top
We make a new abstract superclass called Pet, and give it all the pet methods.
Pet
Animal
Canine
Hippo
Feline
Dog
Cat
Lion
Wolf
Tiger
Dog extends both Pet and Animal
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.

DDD!

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"?

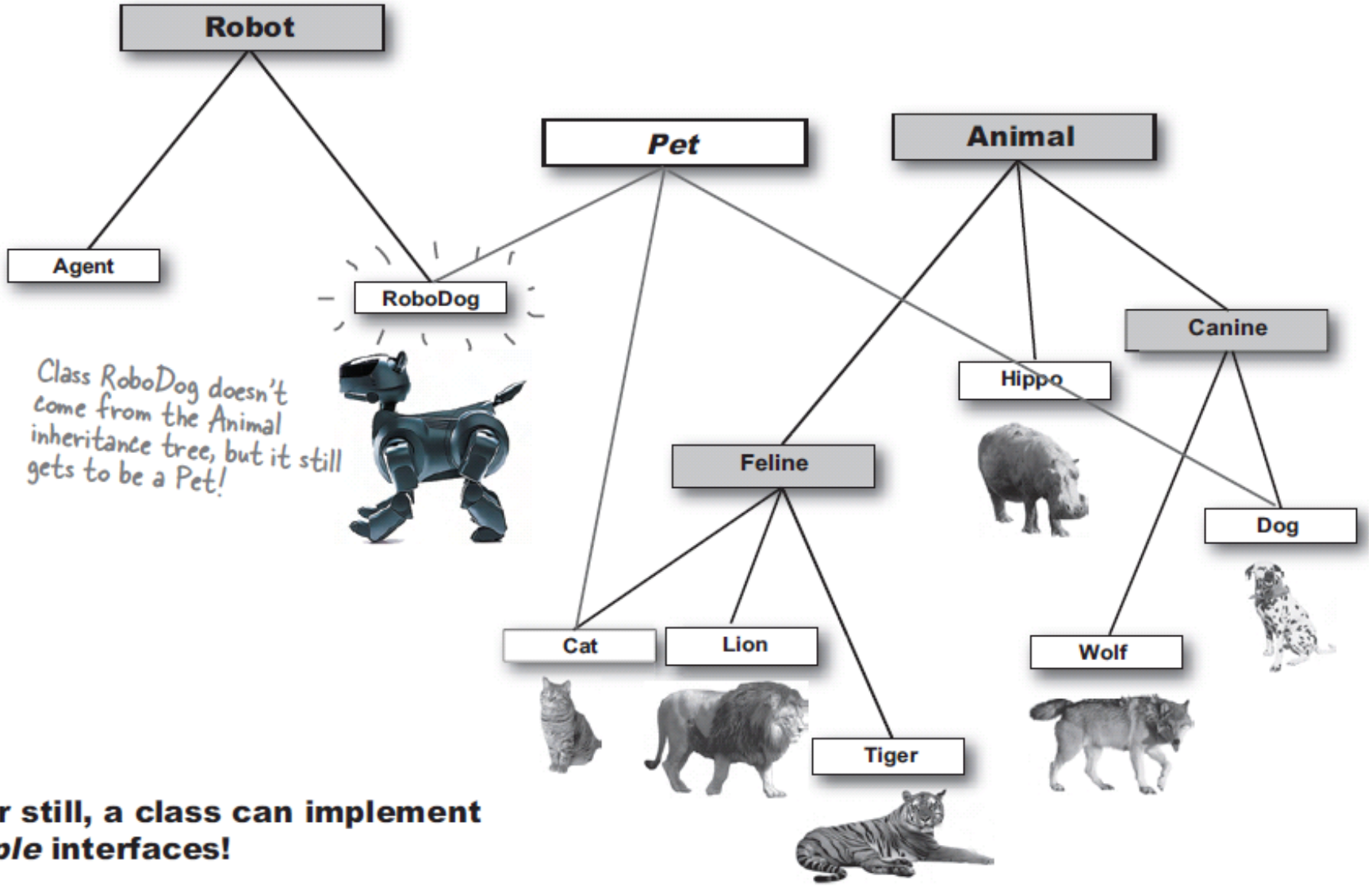
Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

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.

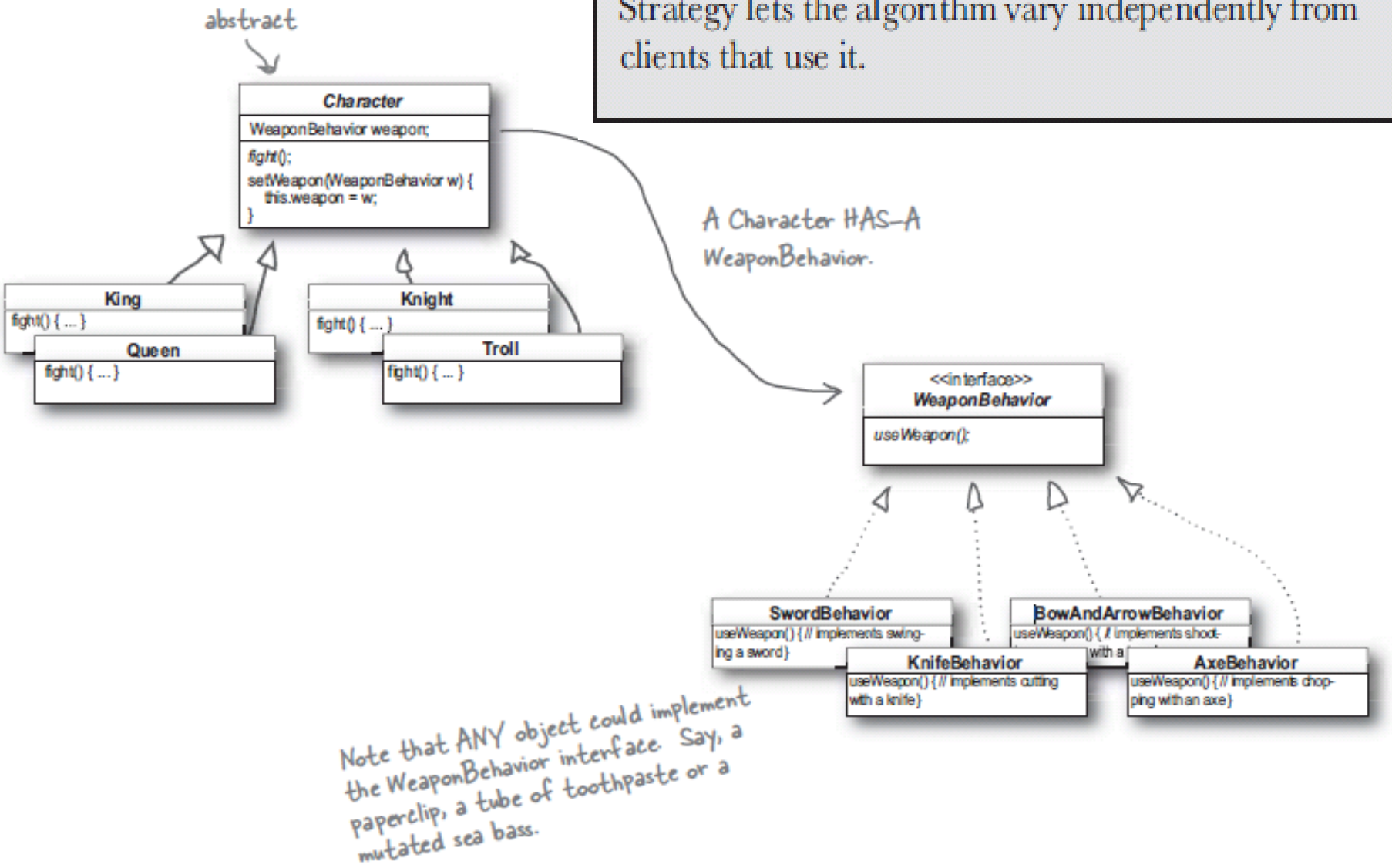


Better still, a class can implement *multiple* interfaces!

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.

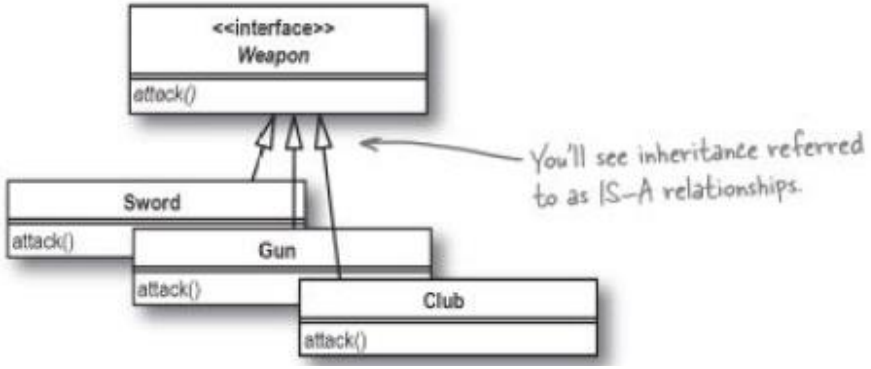
Coding to an interface, rather than to an implementation, makes your software easier to extend.

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

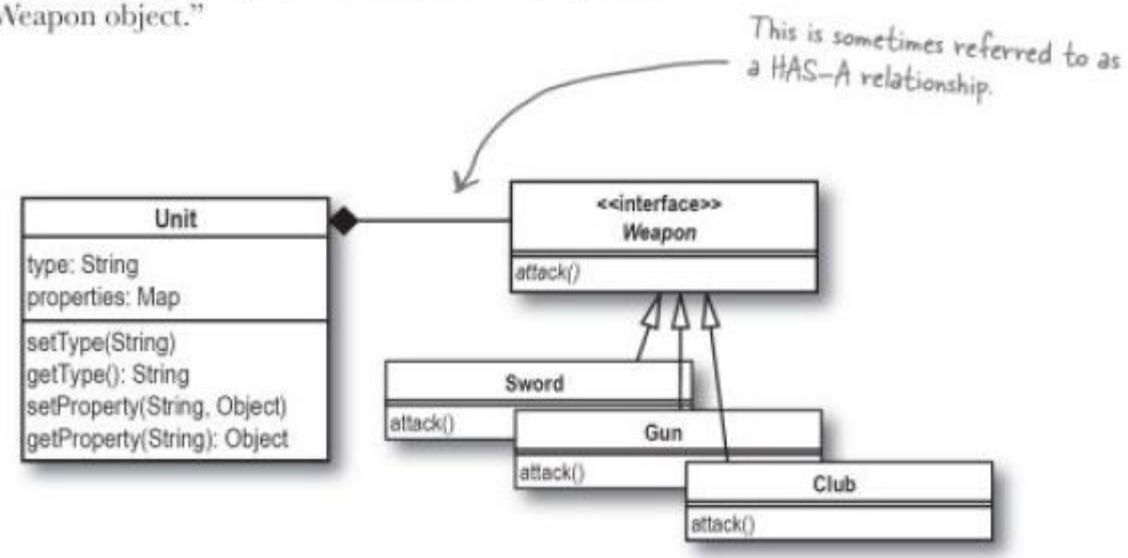
Usually, IS-A relates to inheritance, for example: "A Sword IS-A Weapon, so Sword should extend Weapon."



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

really well and

does not try to

do

or be

something else.

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

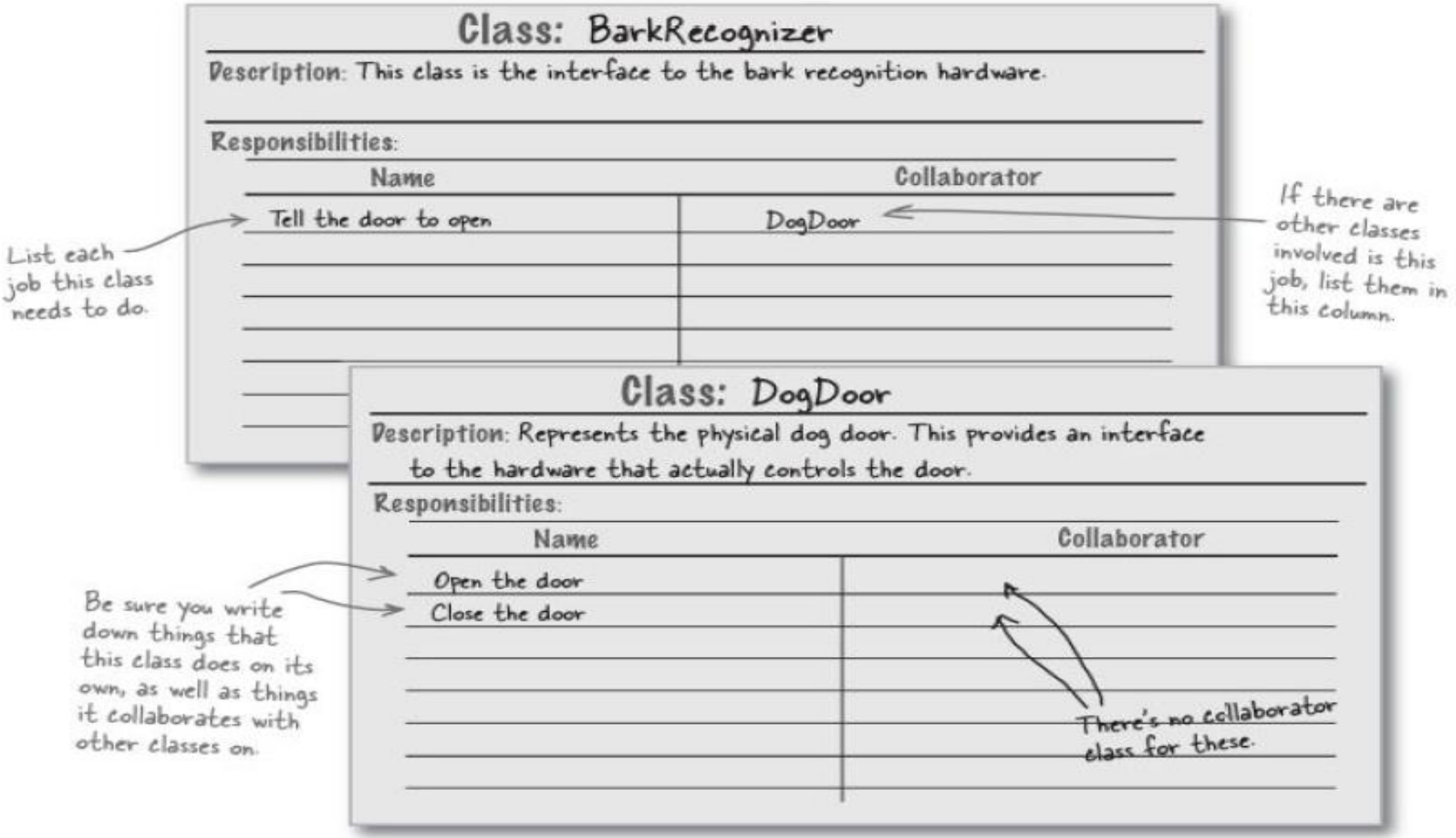
Cohesive classes are focused on specific tasks. Our Inventory class worries about just Rick's inventory, not what woods can be used in a guitar, or how to compare two instrument specs.

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.

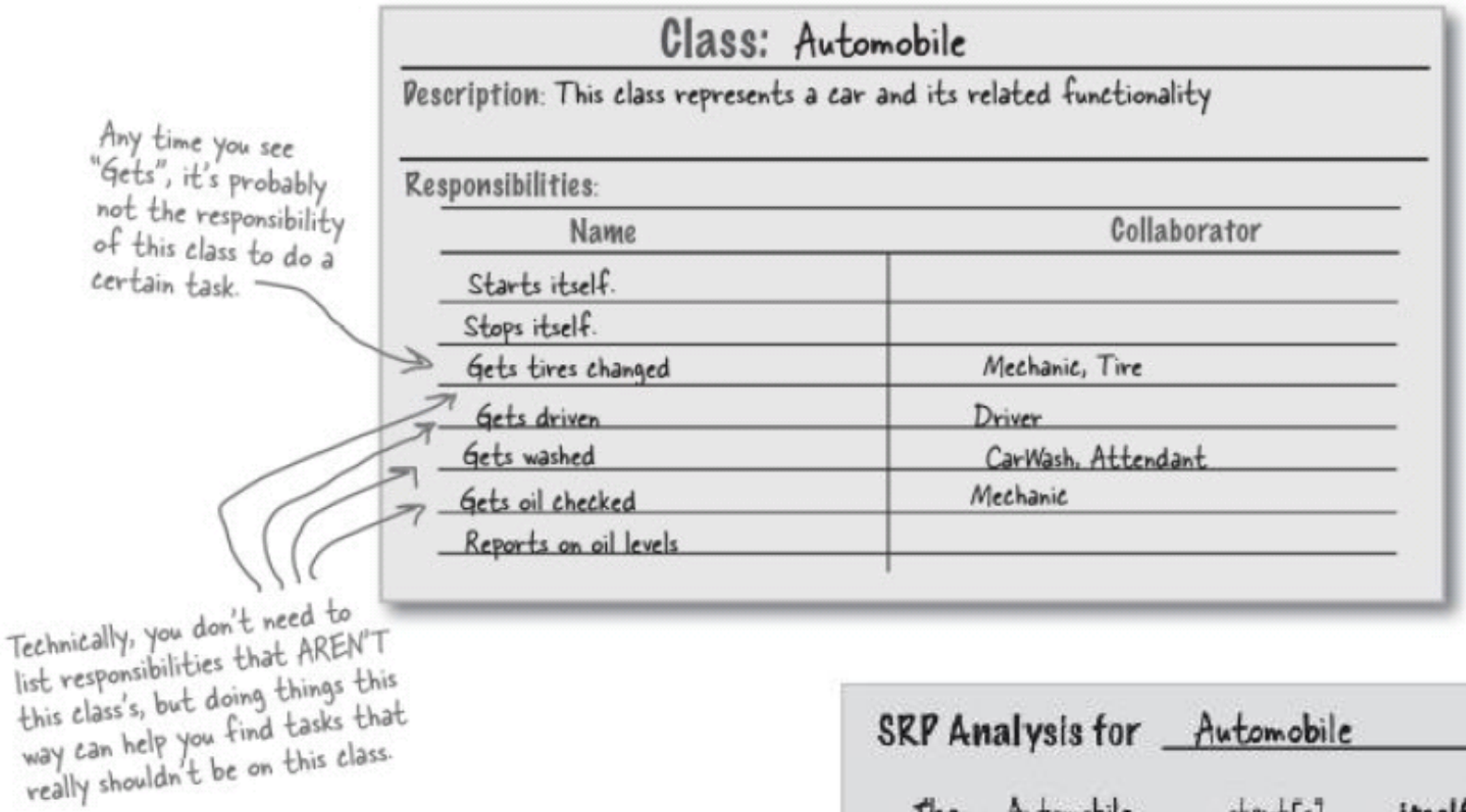
Instrument doesn't try to handle searches, or keep up with what woods are available. It is focused on describing an instrument–and nothing else.

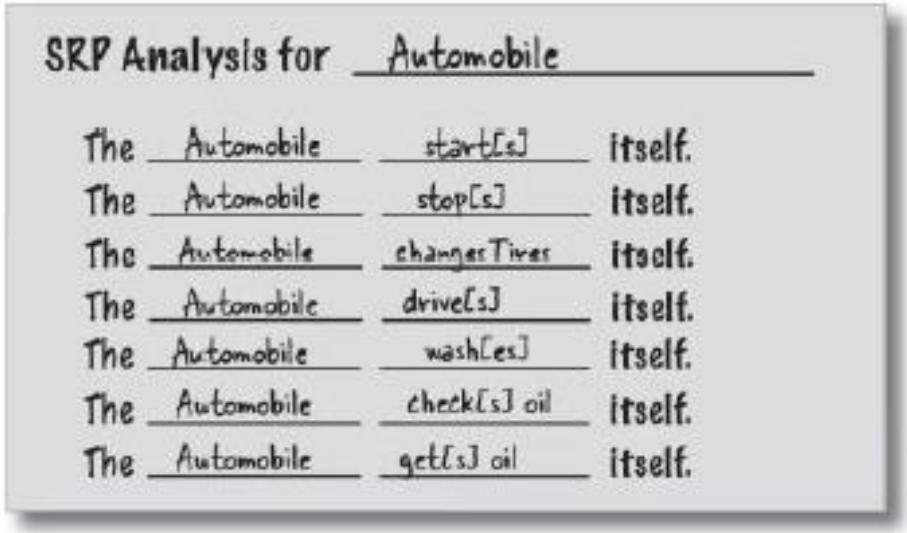
Tools: CRC Cards

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.



Tools: SRP





CRC cards help implement the SRP

Need for Design Patterns

Design & Anti-Design Patterns

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

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

Shared Vocabulary?

Overheard at the local diner...

Cont…

The power of a shared pattern vocabulary

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

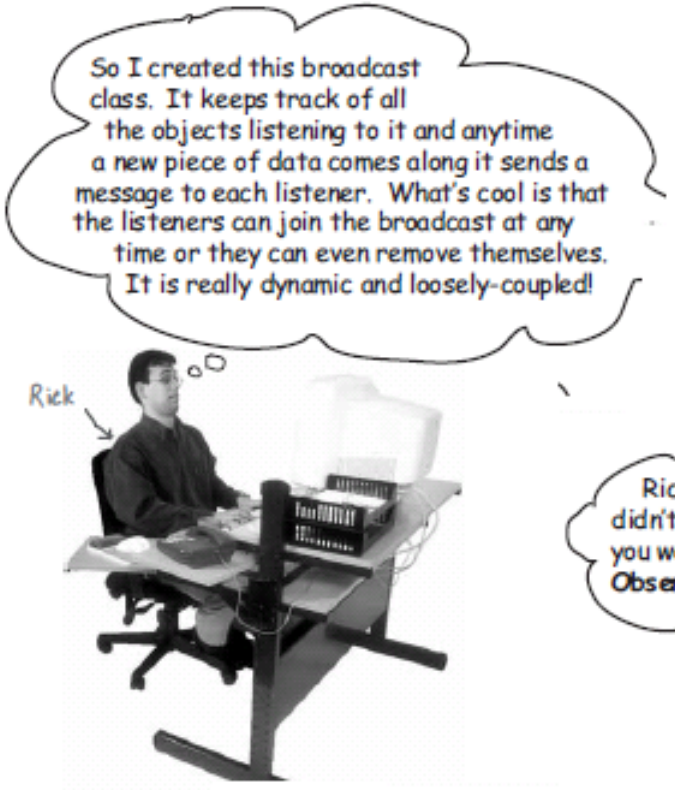
Shared pattern vocabularies are POWERFUL.

Patterns allow you to say more with less.

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

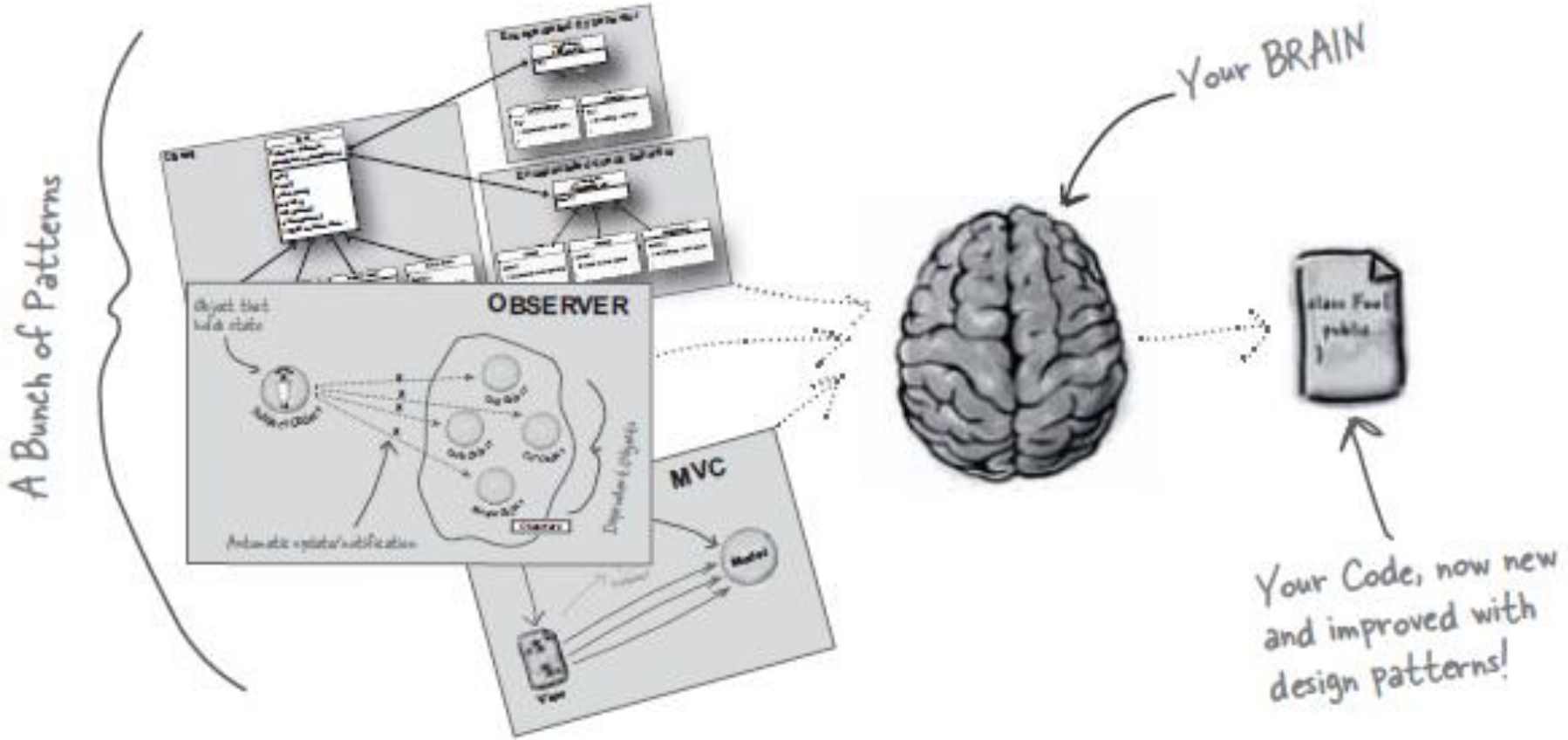
Shared vocabularies can turbo charge your development team.

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



Usage

How do I use Design 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.

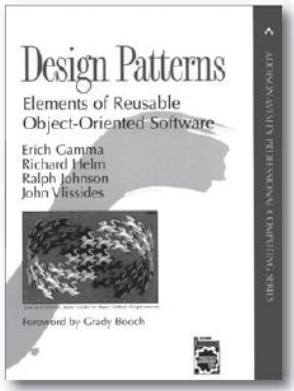
Richard Helm

Ralph Johnson

John Vlissides

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

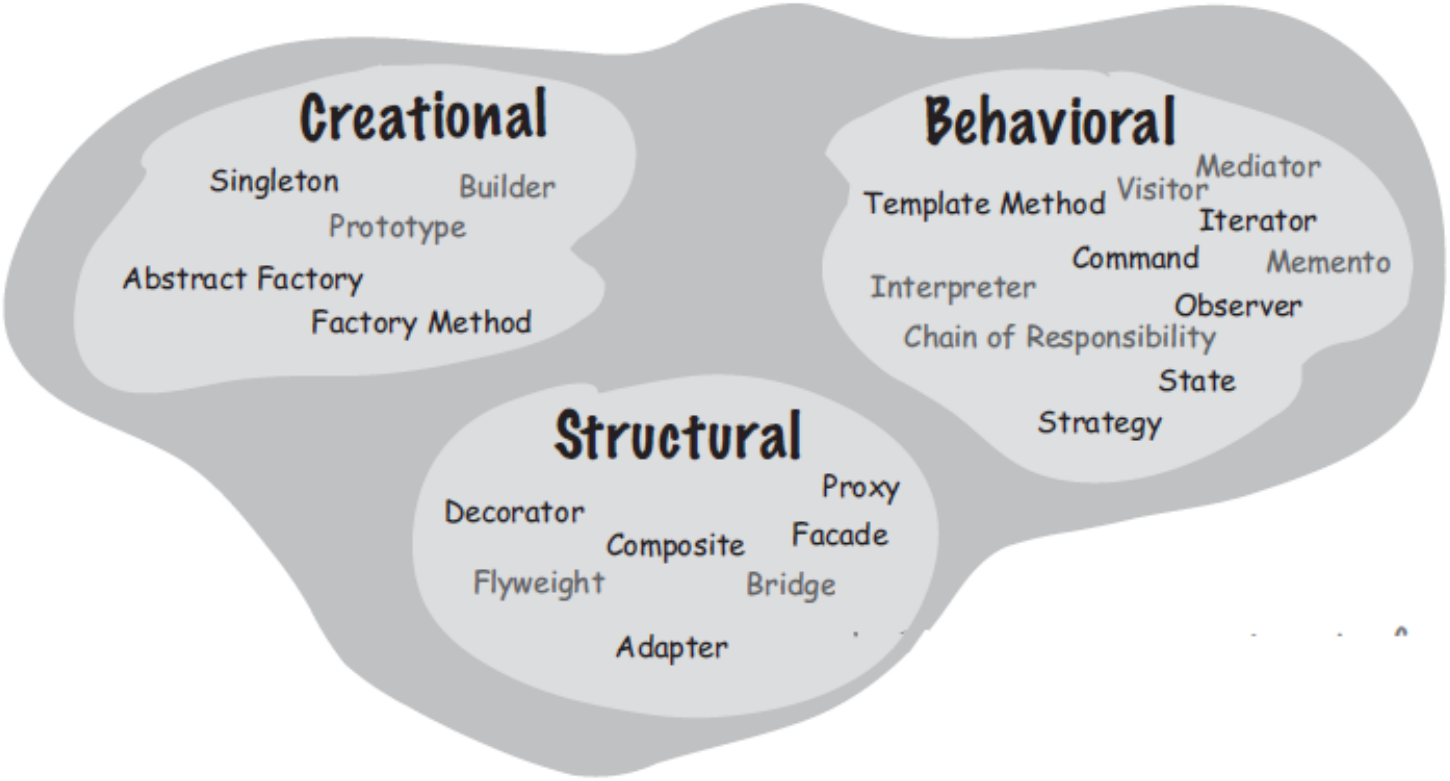
Erich Gamma



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

Design Guidelines / Checkpoints

- ✓ 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

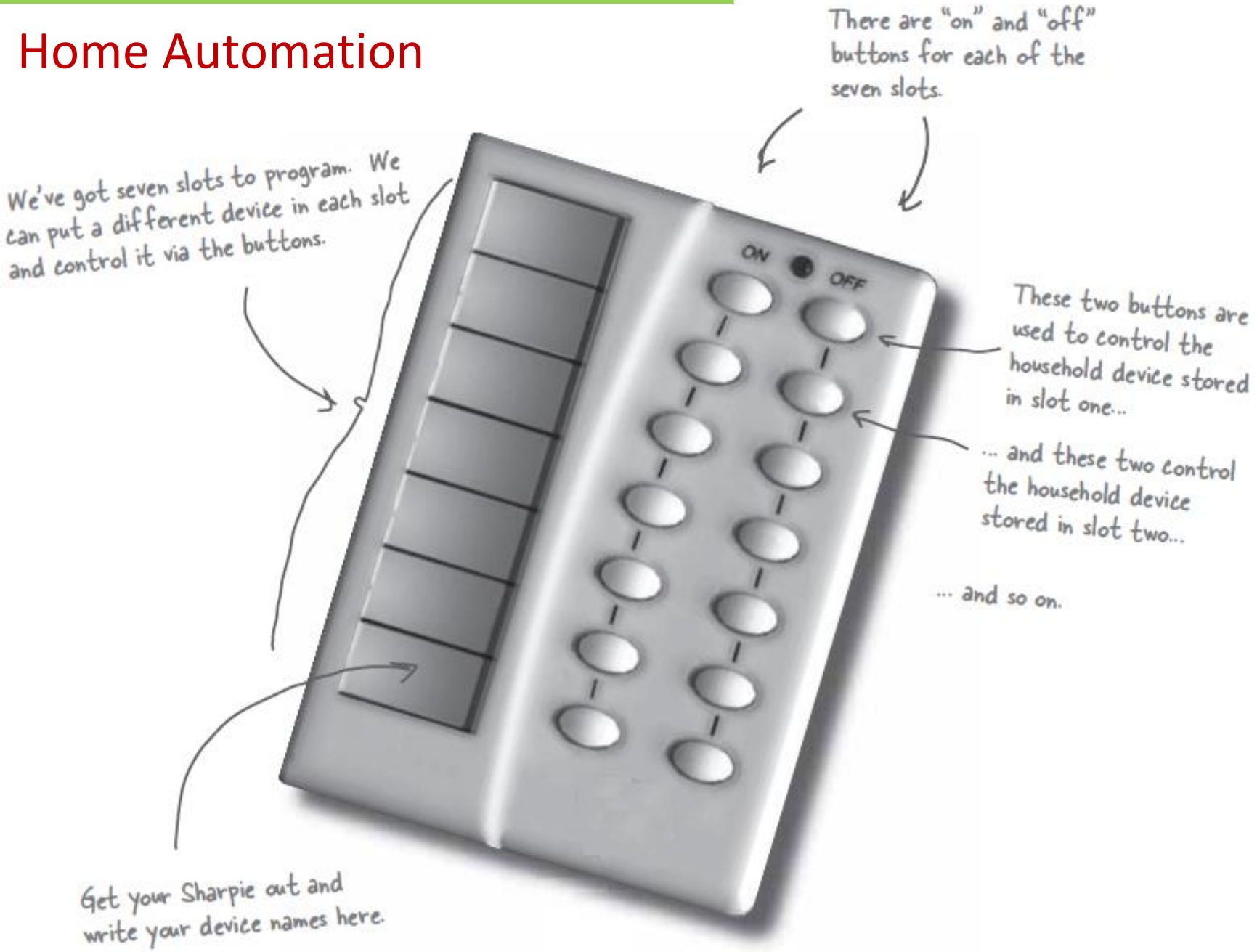
“KISS YAGNI”

- Keep it Stupid Simple
- Keep it Simple & Straight Forward
- Keep it Simple & Stupid

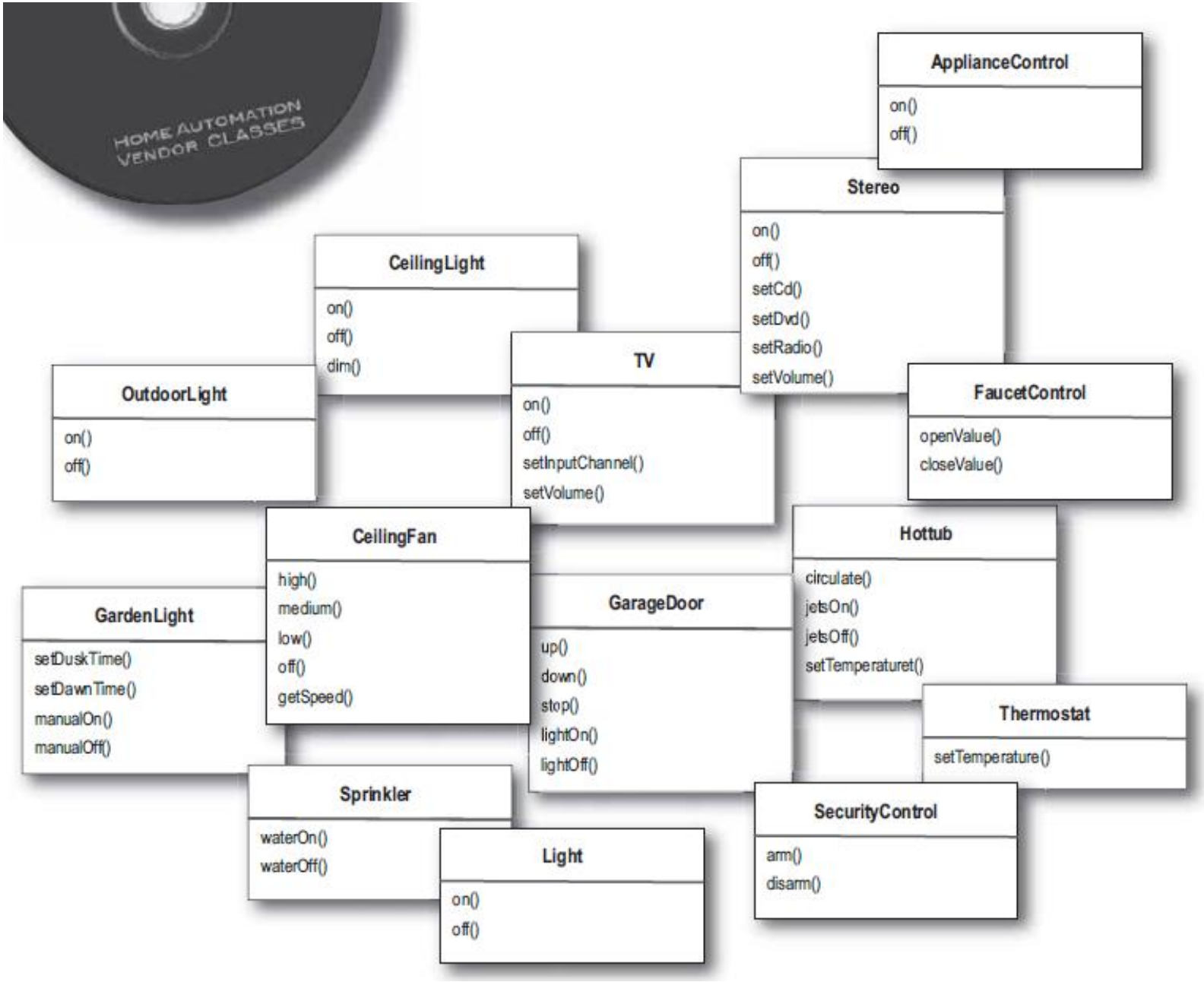
- You Ain’t Gonna Need it – Do the simplest that could possibly work

More Design Patterns

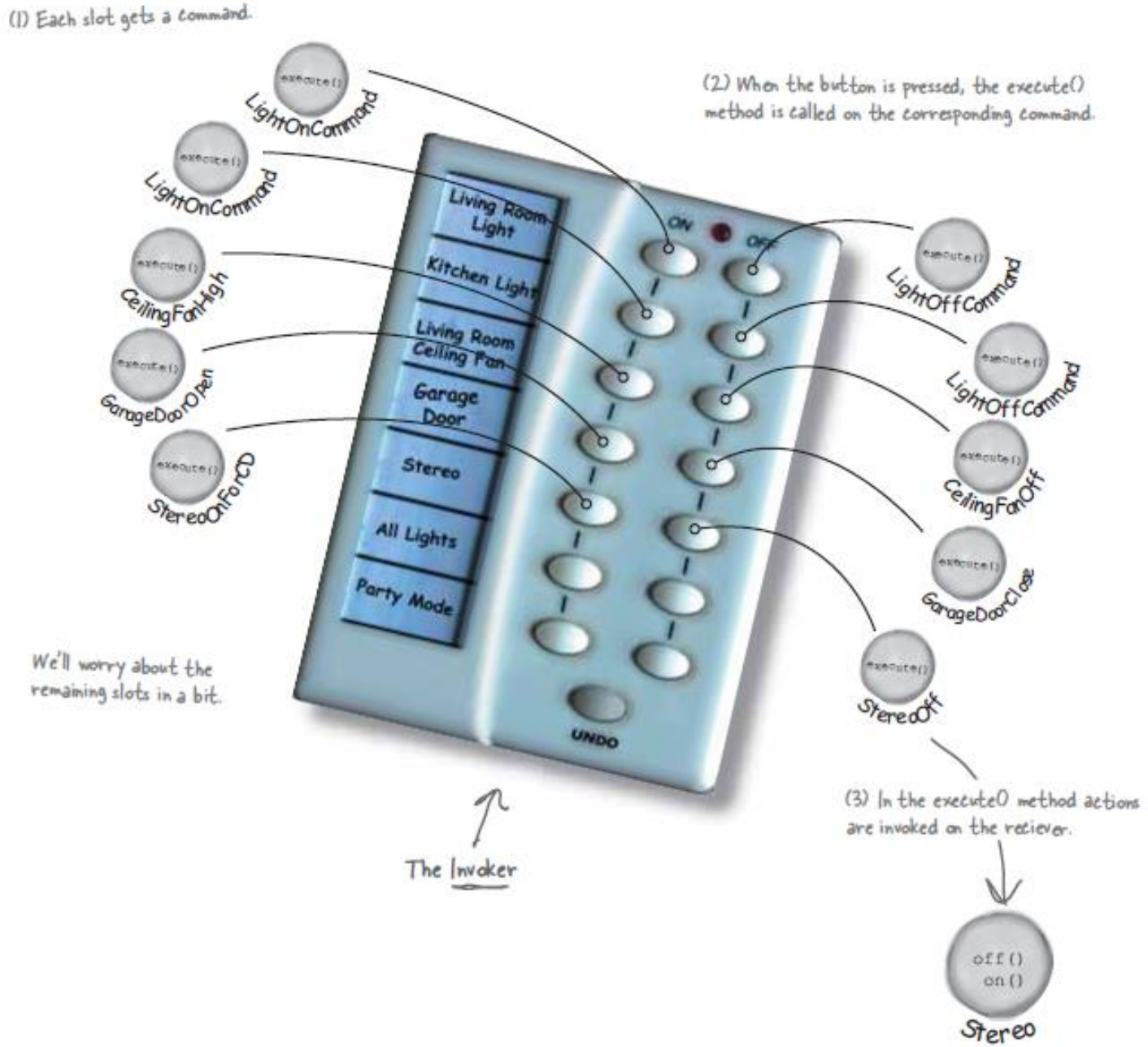
Home Automation



Cont… (Vendor Provided Classes)

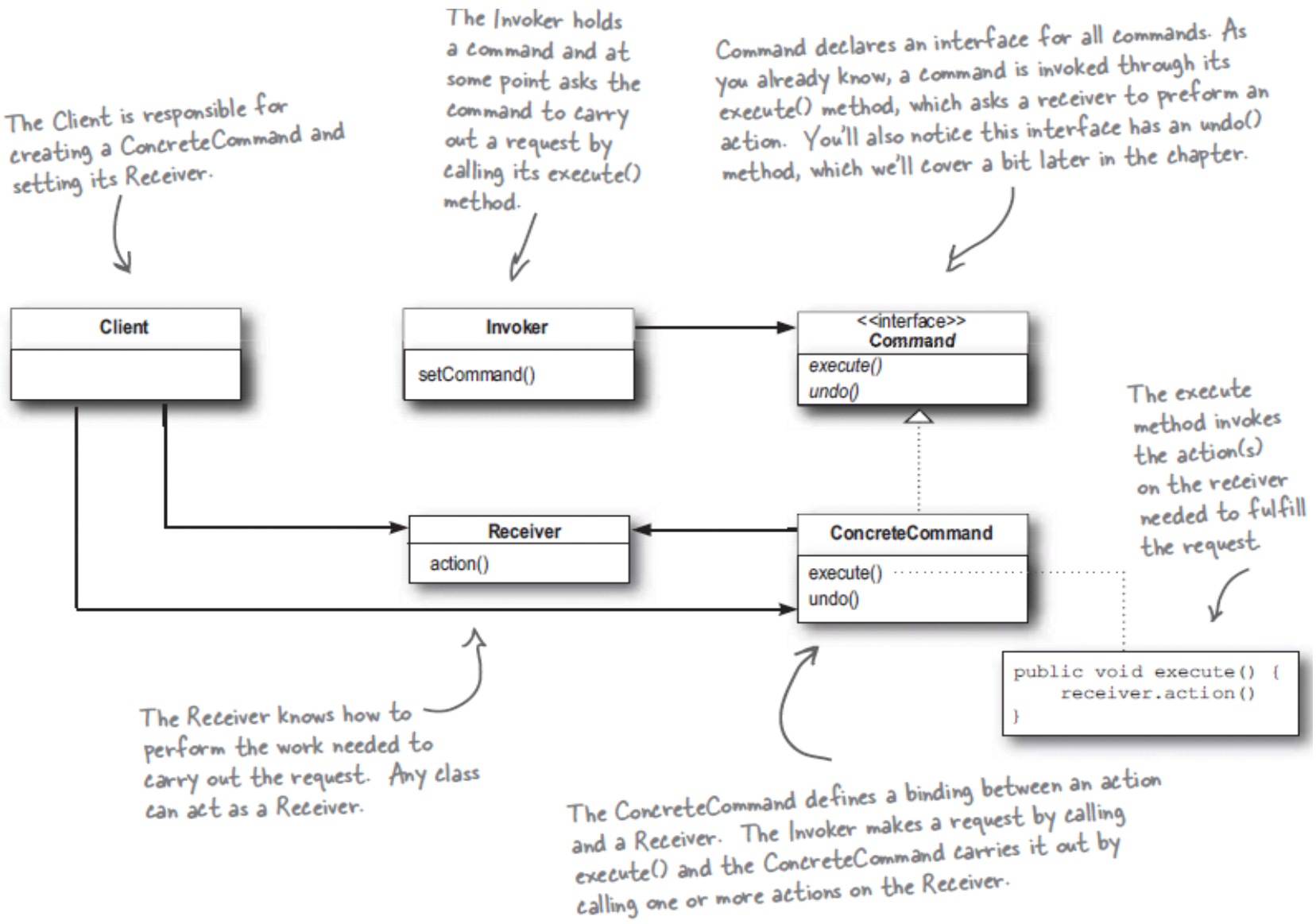


Command Assigned to Slots

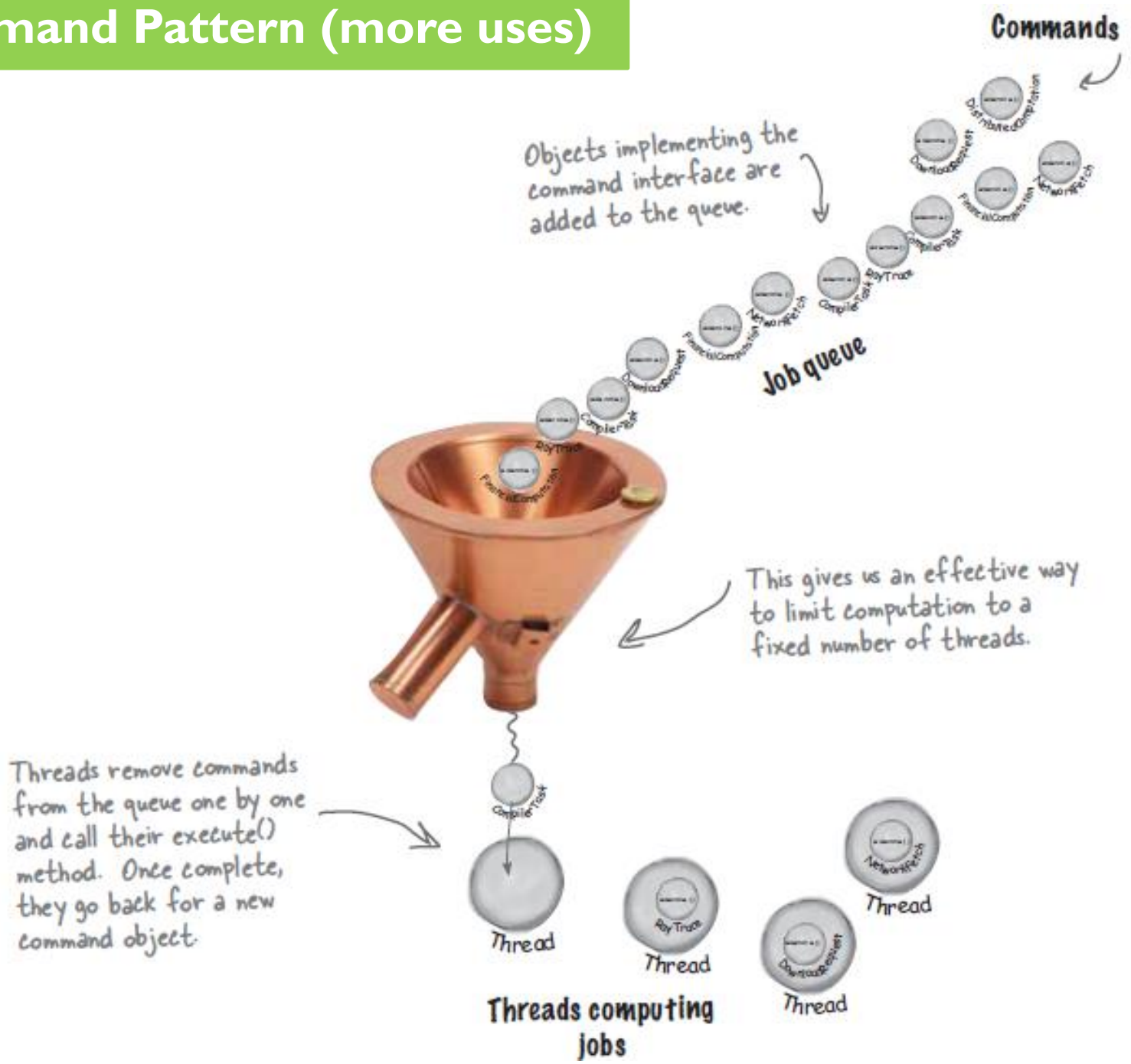


Command Pattern

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.



Command Pattern (more uses)

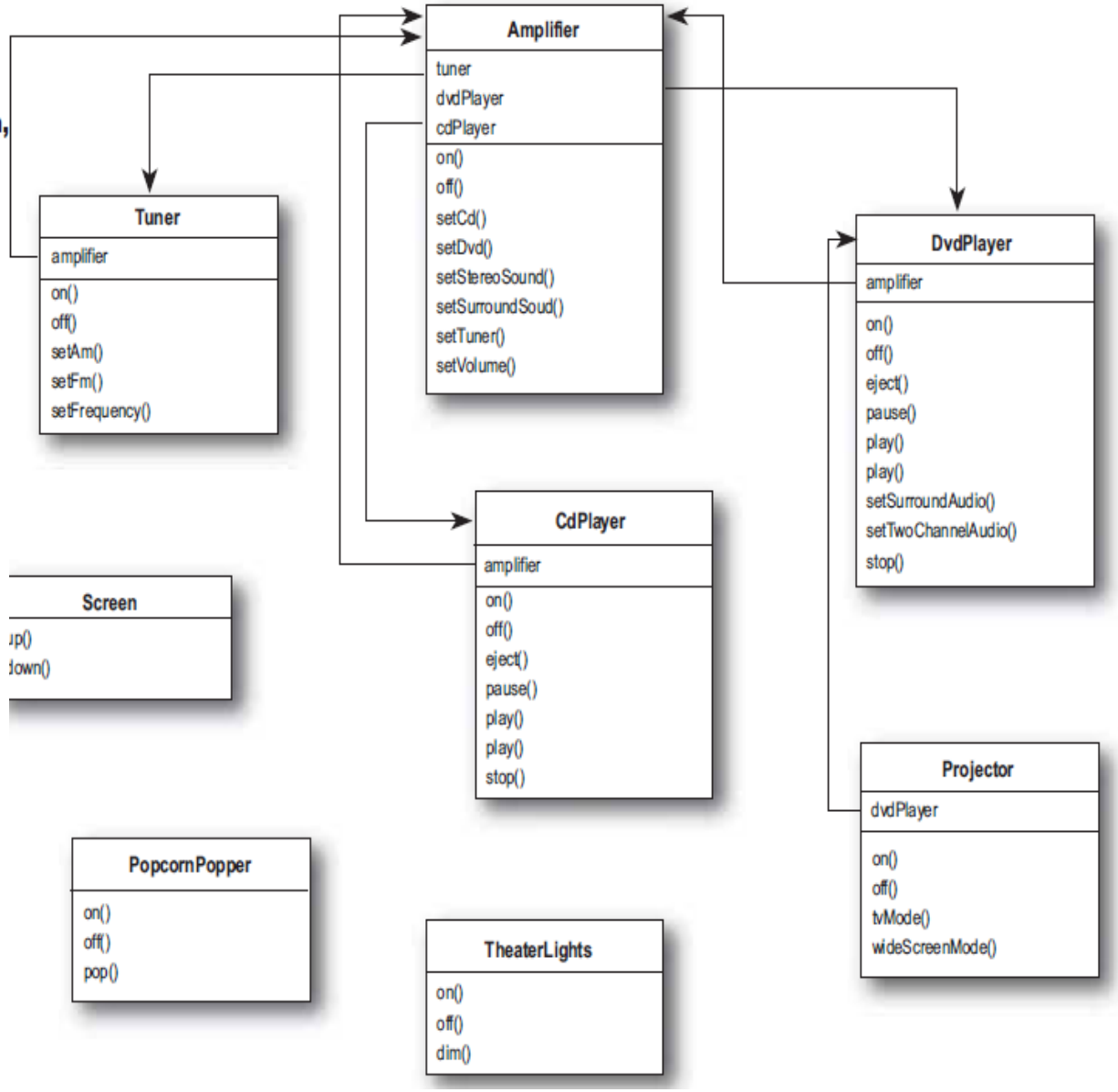


Pattern 3

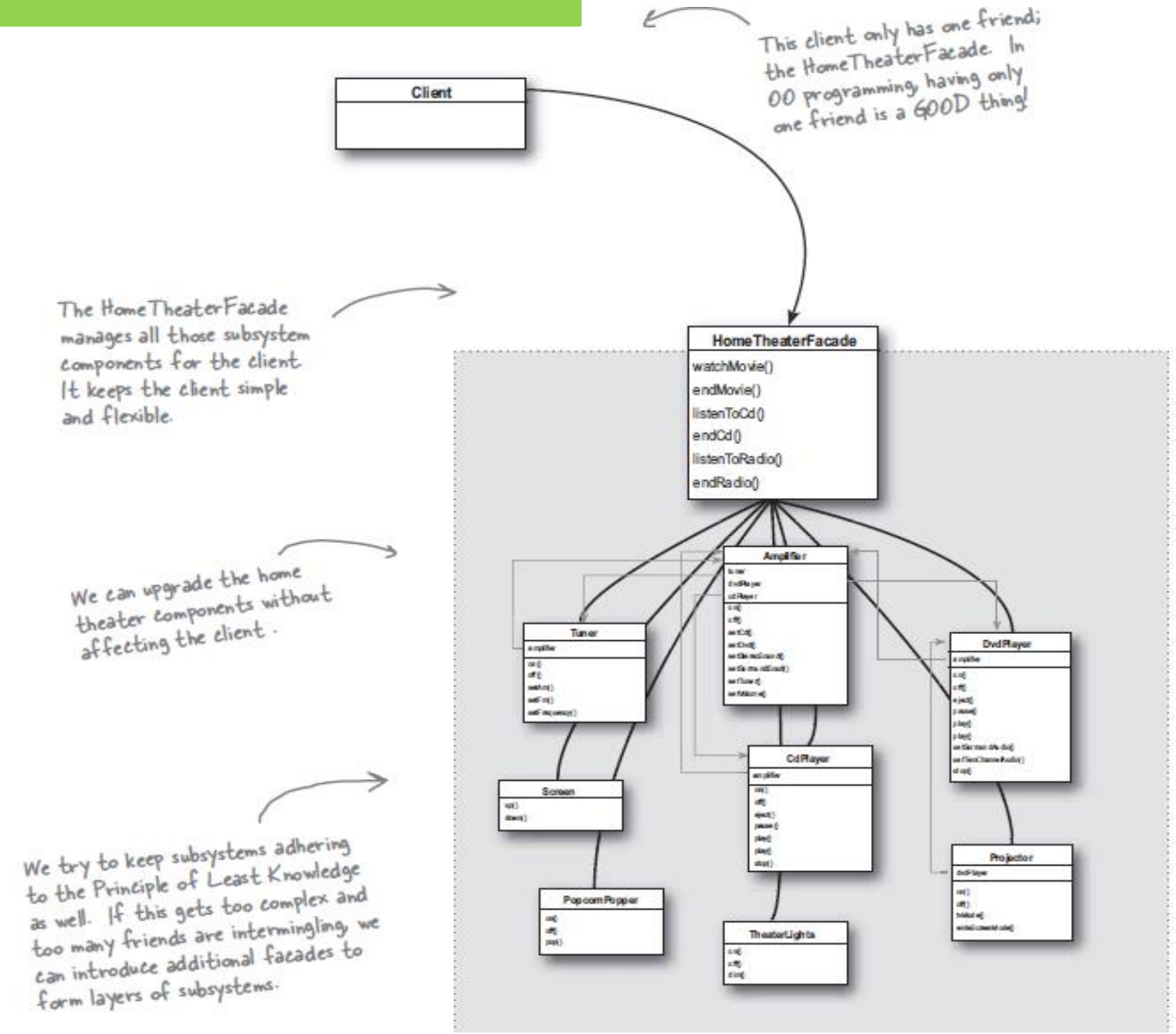
Watching a movie (the hard way)

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

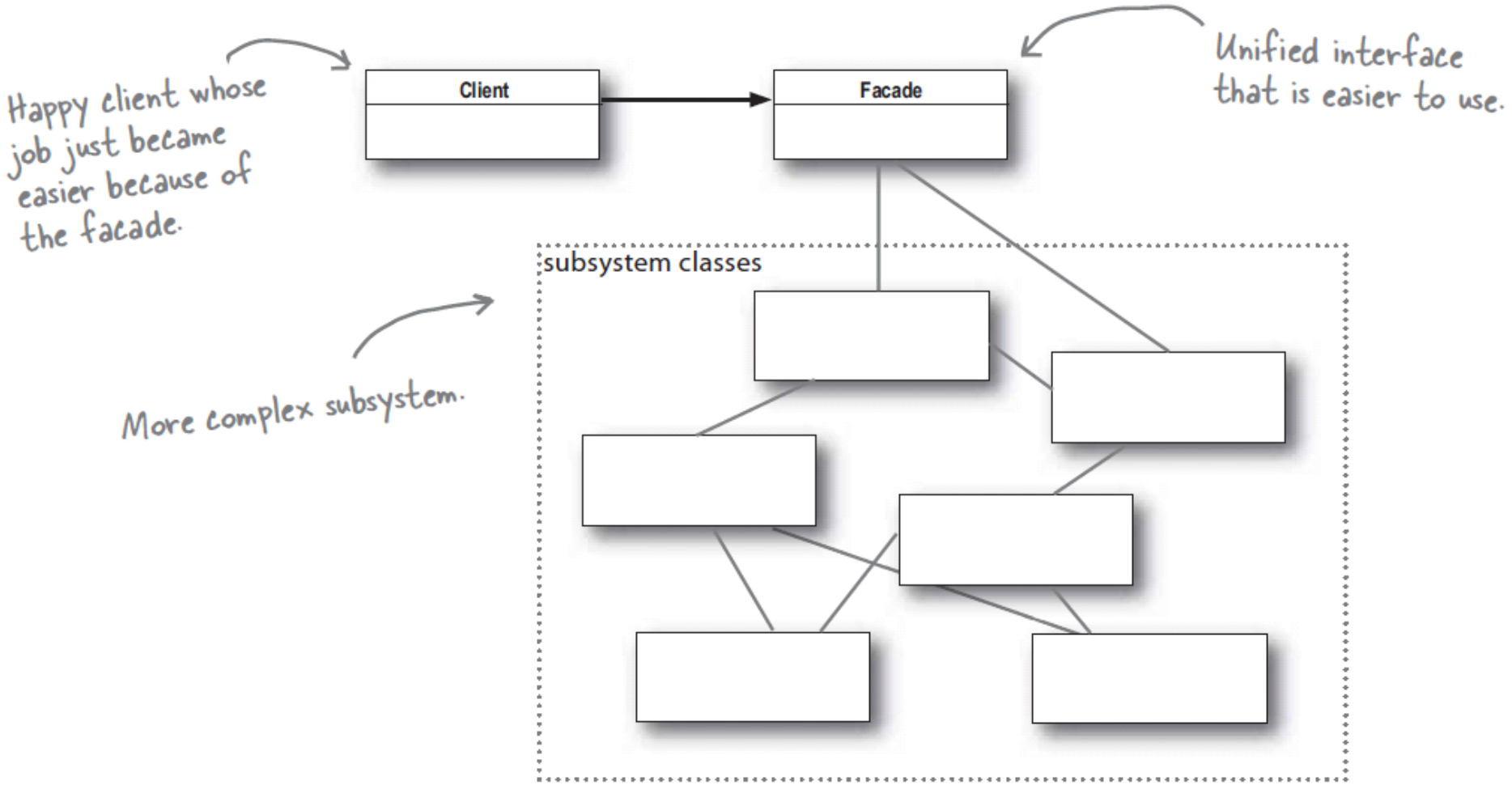


Home Theater Facade

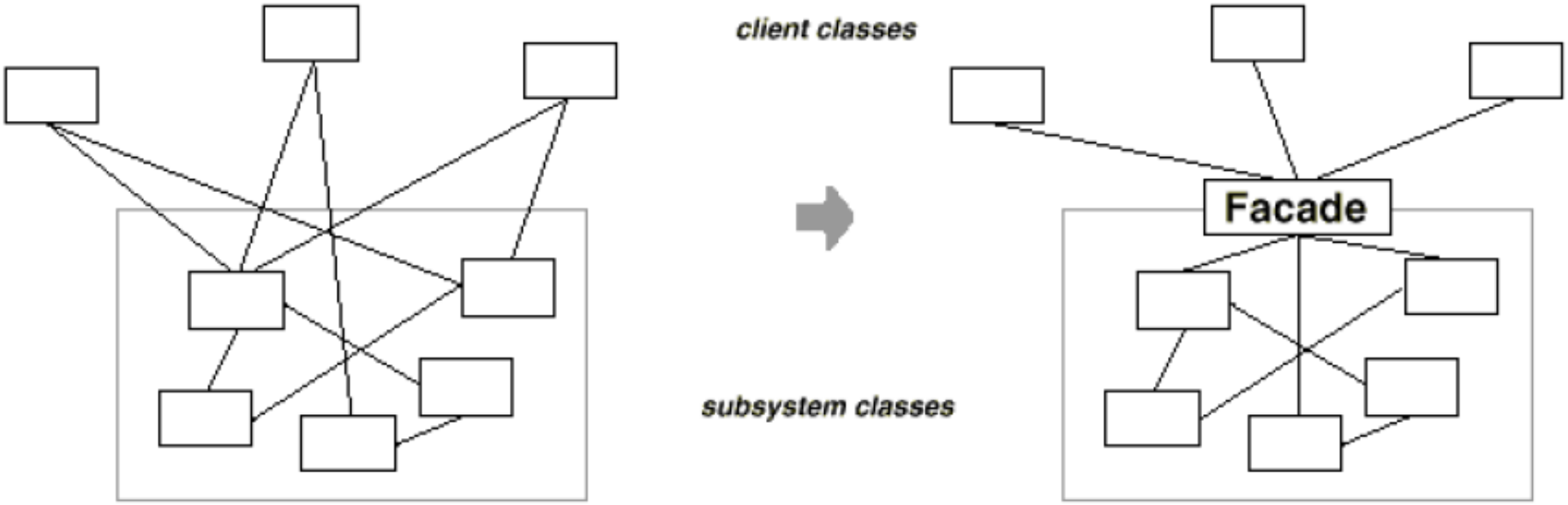


Façade Pattern

The Facade Pattern provides a unified interface to a set of interfaces in a subsytem. 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);
}

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();
}
```

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.

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

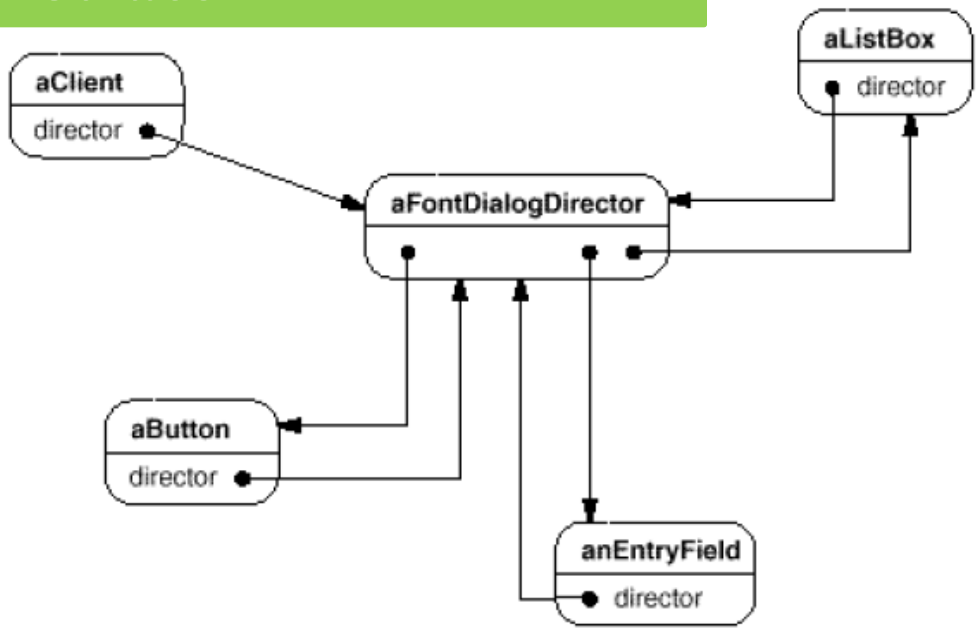
The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close "friends." The principle is usually stated as:

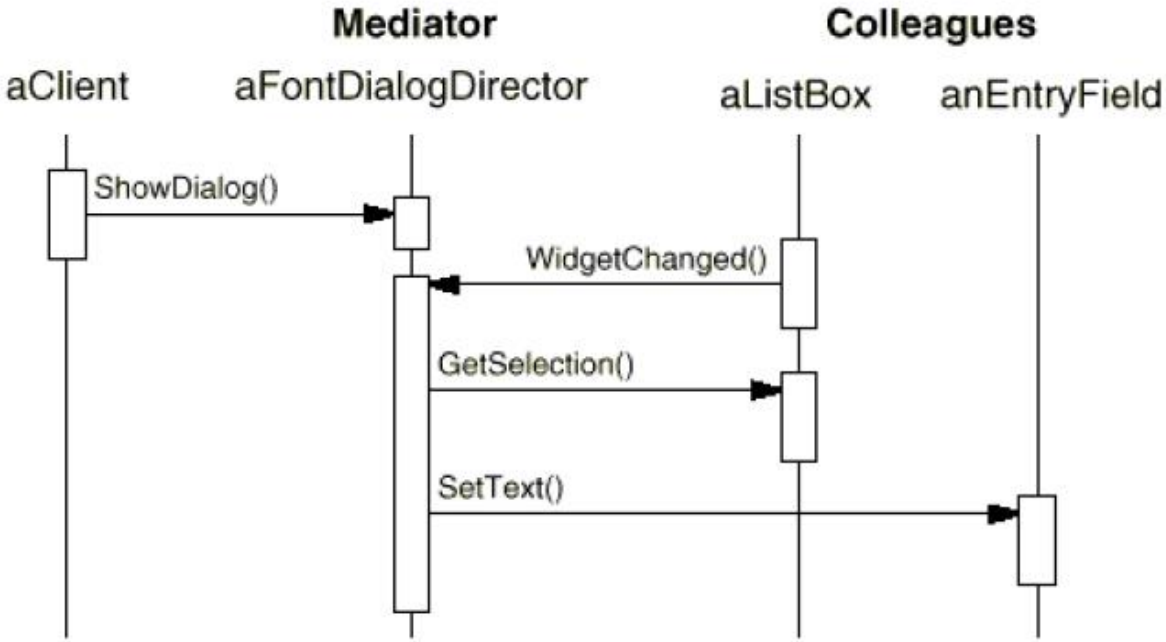
Design Principle

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

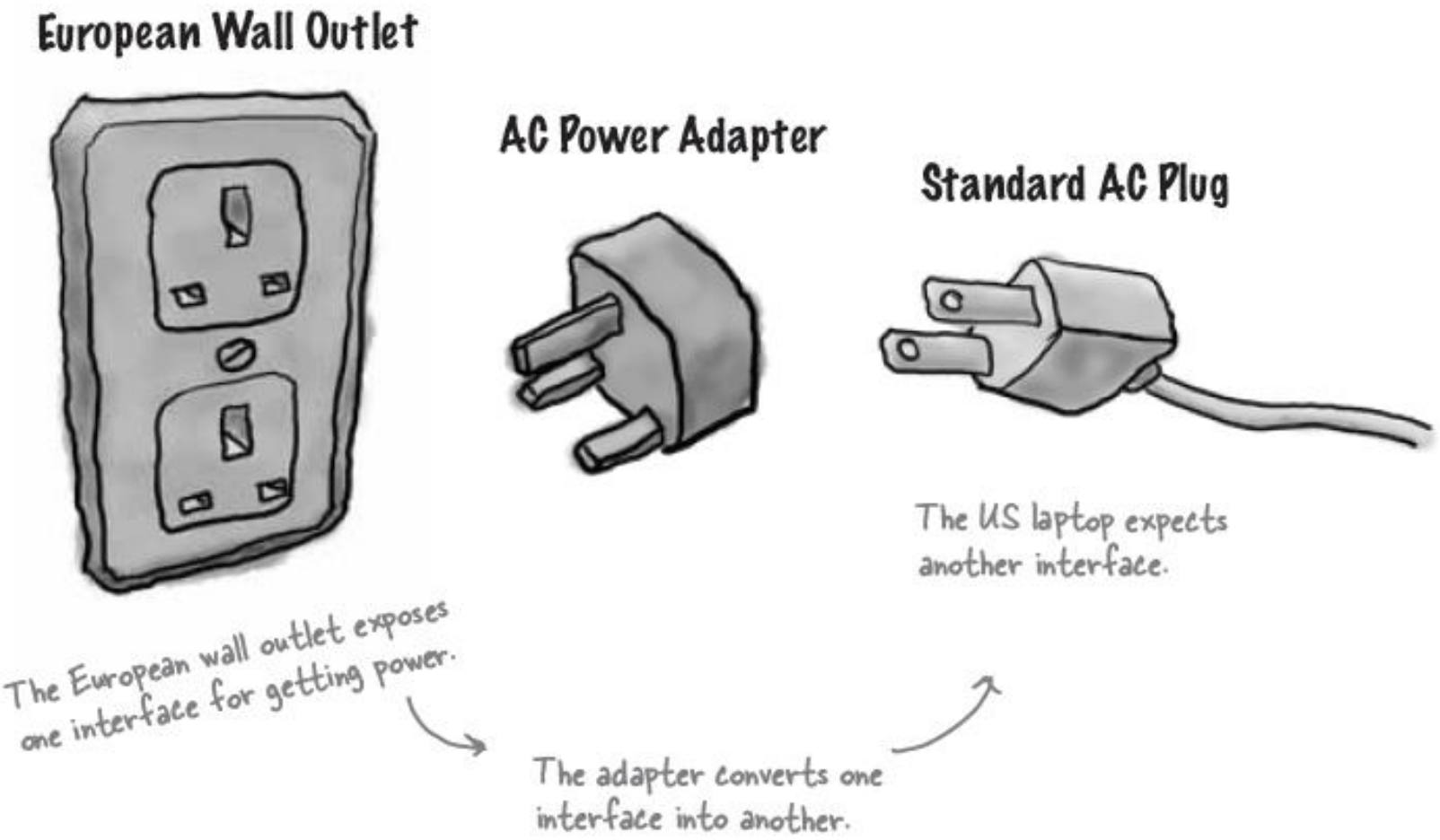
But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

Pattern 4: Mediator

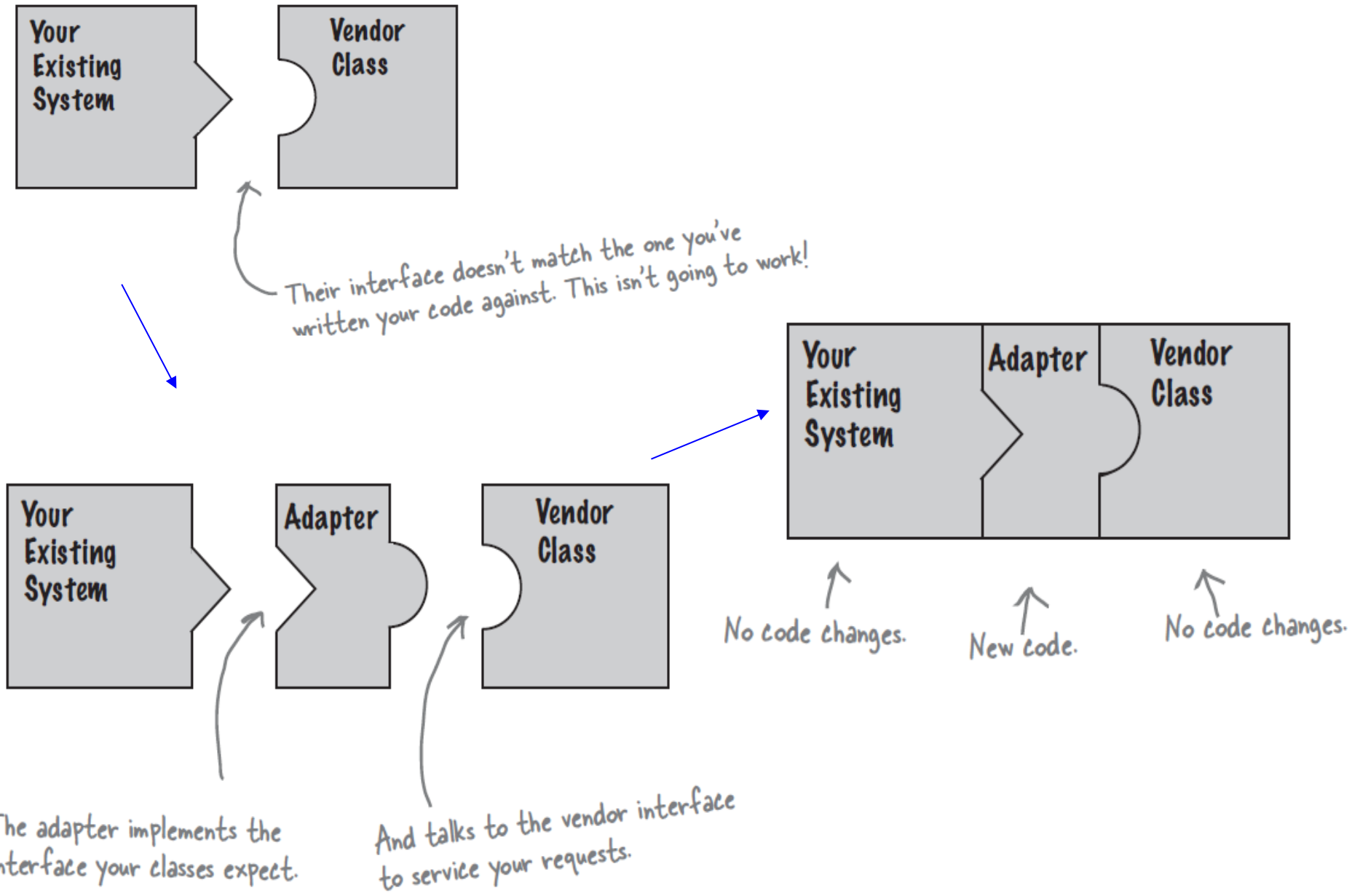




Adapters all around us



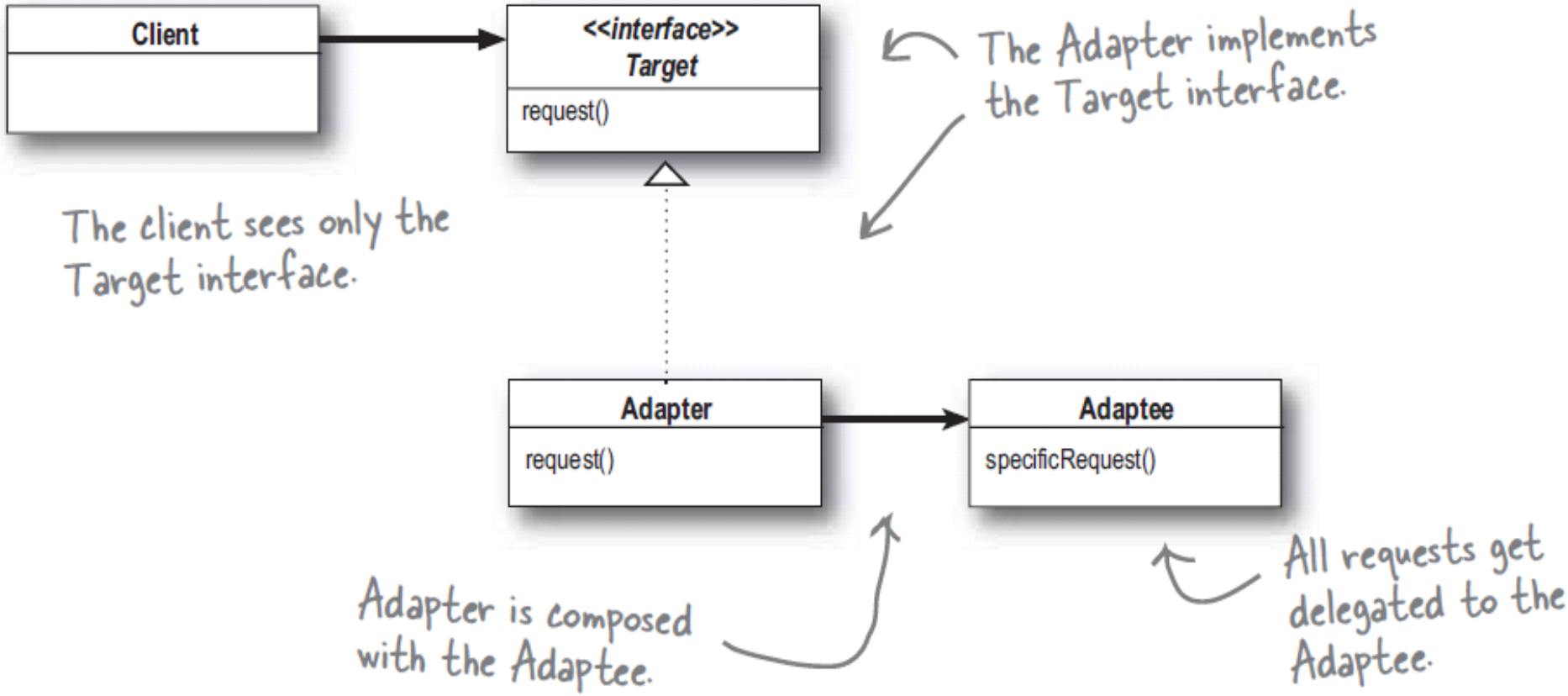
OO Adaptors


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!
Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.
Your Existing System
Adapter
Vendor Class
No code changes.
New code.
No code changes.

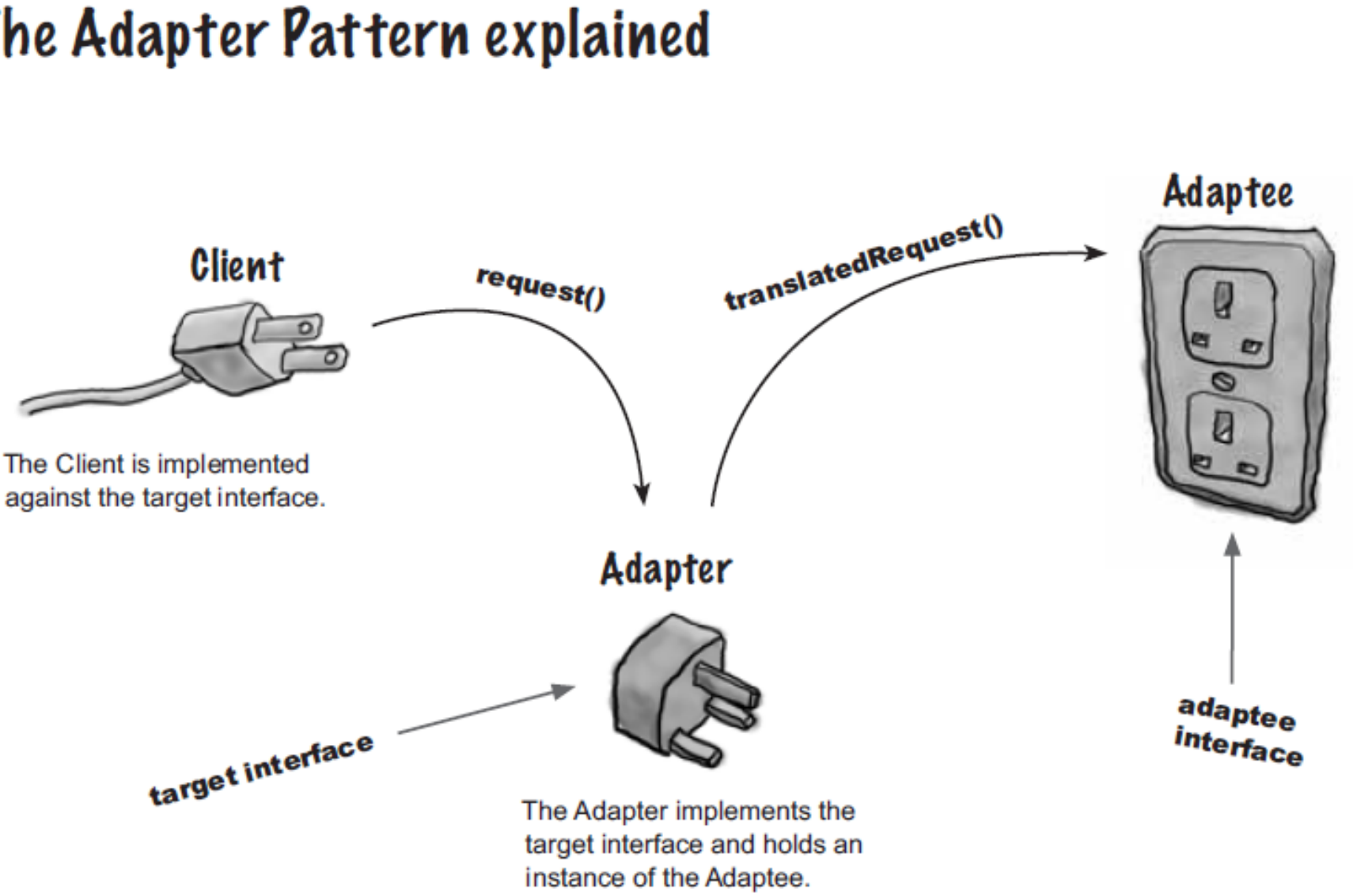
Camera Lens Adaptor

Adaptor Pattern

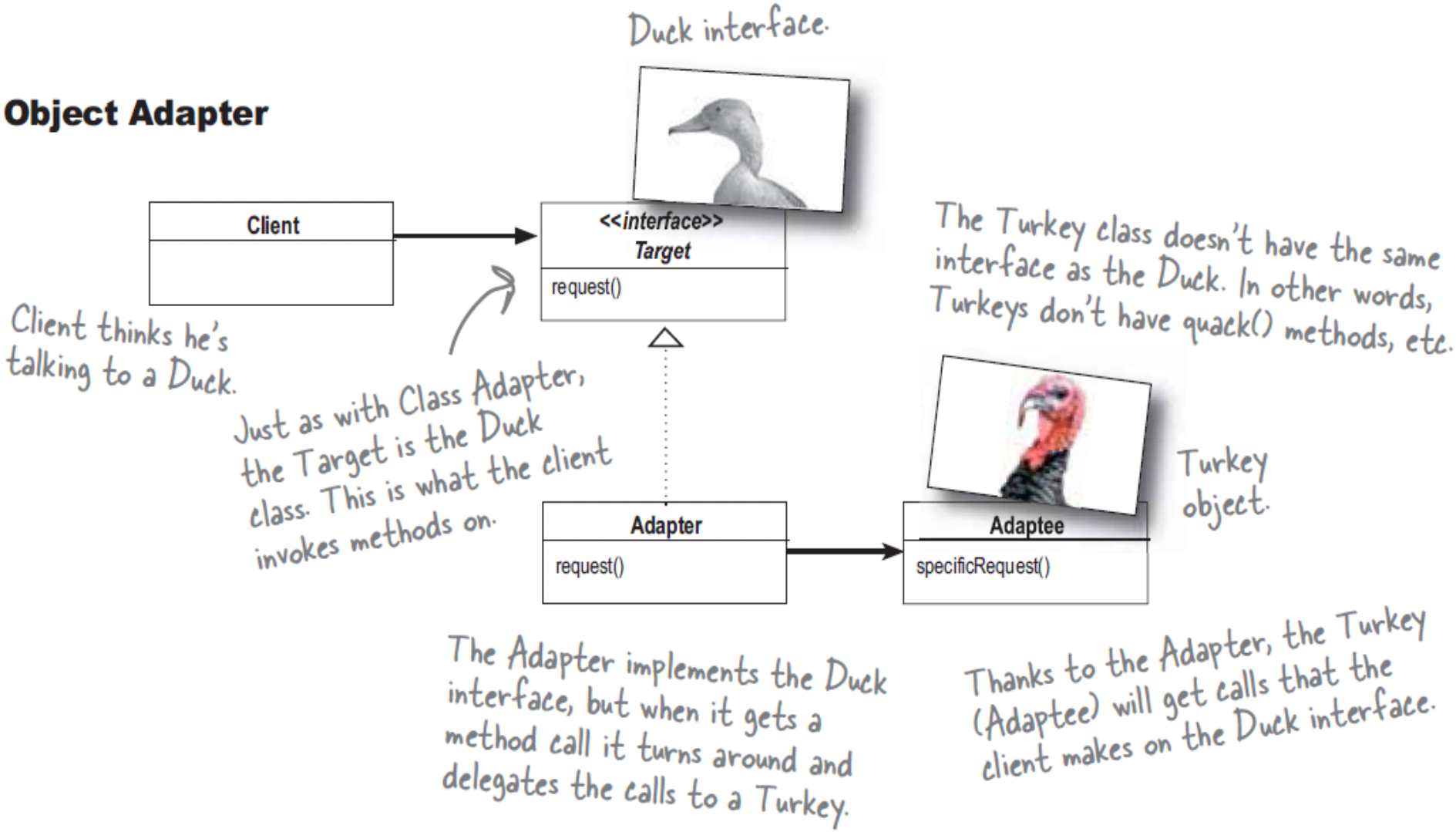
The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Adaptor Explanation



Object Adaptor



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

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.

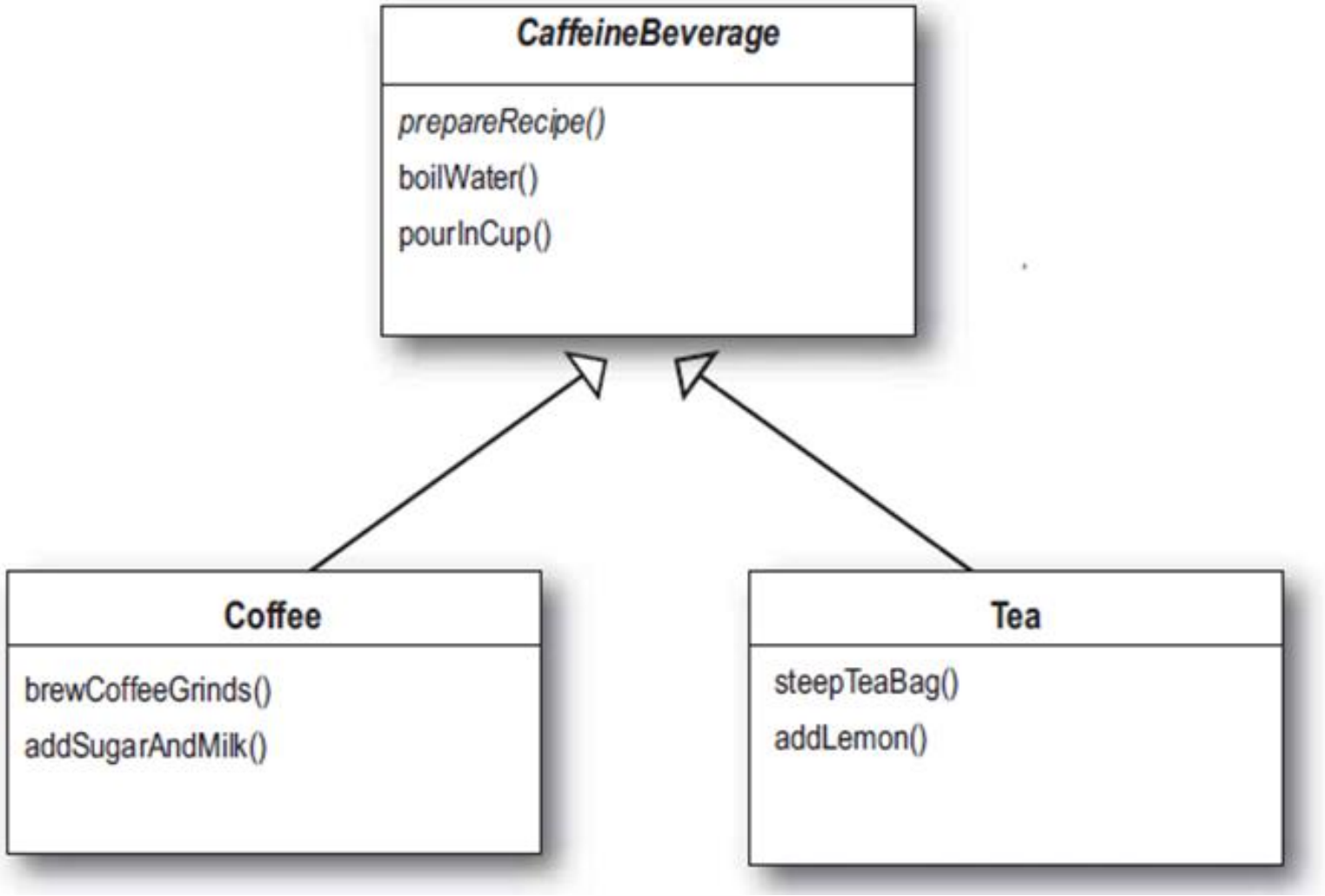
The recipe for coffee looks a lot like the recipe for tea, doesn't it?

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

1. Boil some water
2. Steep the teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

generalize

generalize

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

Coffee subclass

2. Steep the teabag in the water
4. Add lemon

2. Brew the coffee grinds
4. Add sugar and milk

Cont…

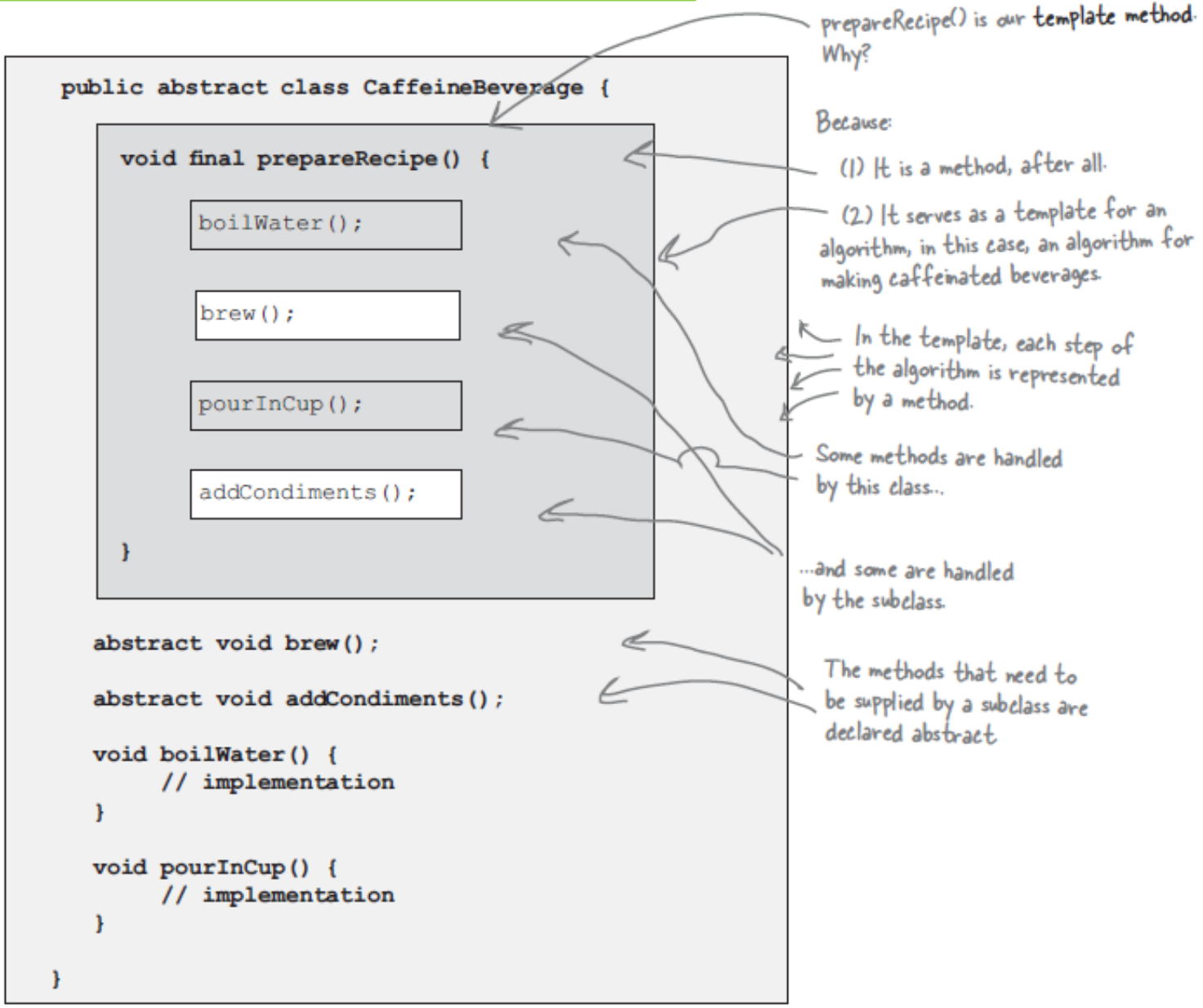
```
public abstract class CaffeineBeverage {

    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");
    }
}
```

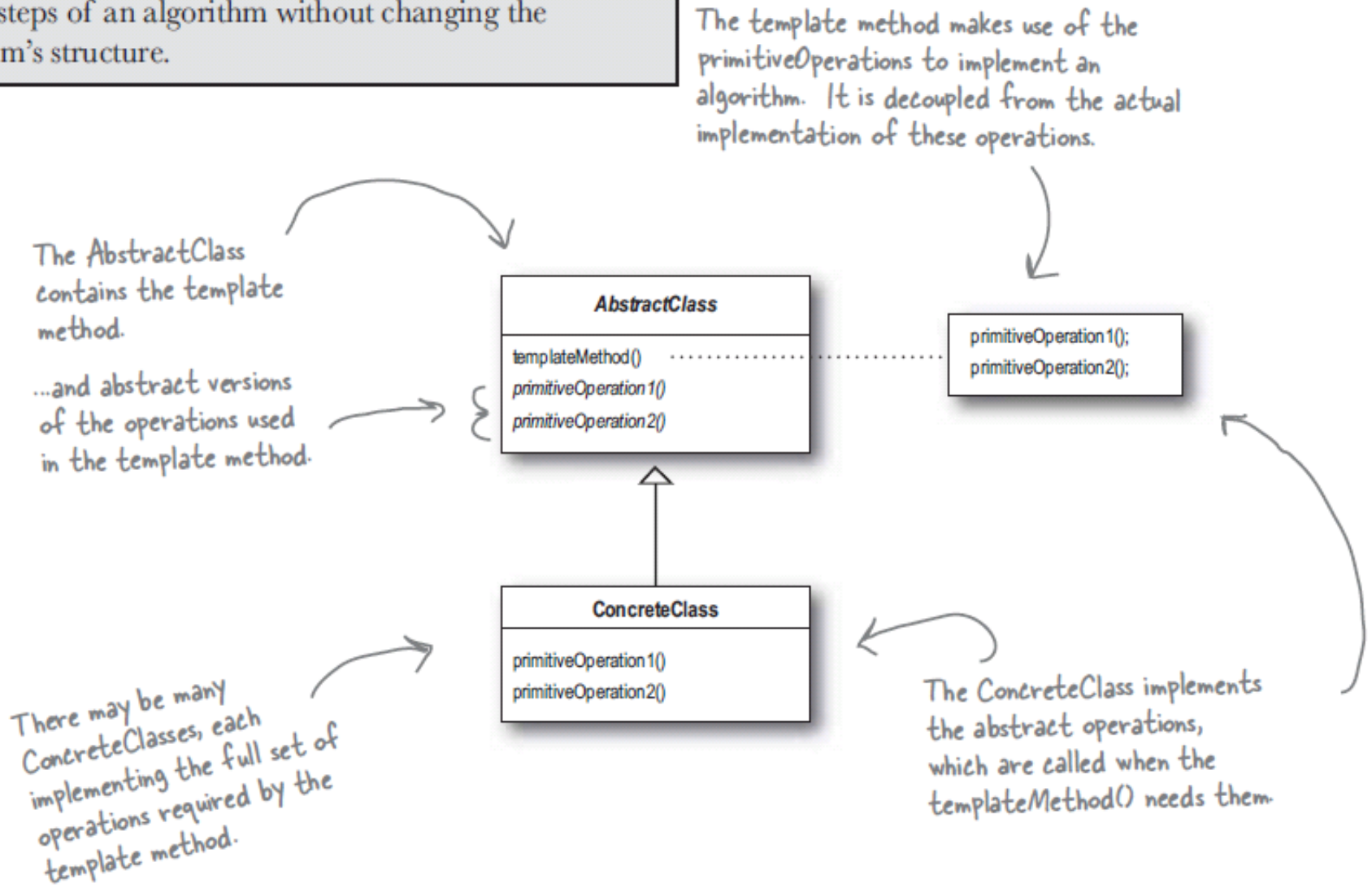
```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Template structure



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.



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 lettu
whole wheat
2.99
BLT
Bacon with lettuce & tor
Soup of the day
A bowl of the soup of the
a side of potato salad
Hot Dog
A hot dog, with saurkrau
topped with cheese
Steamed Veggies and Bro
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

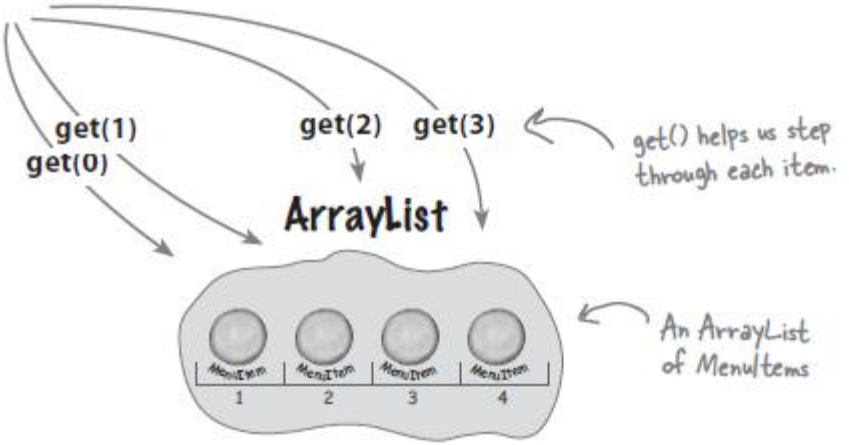
Access Menu Spec

Java-Enabled Waitress: code-name "Alice"

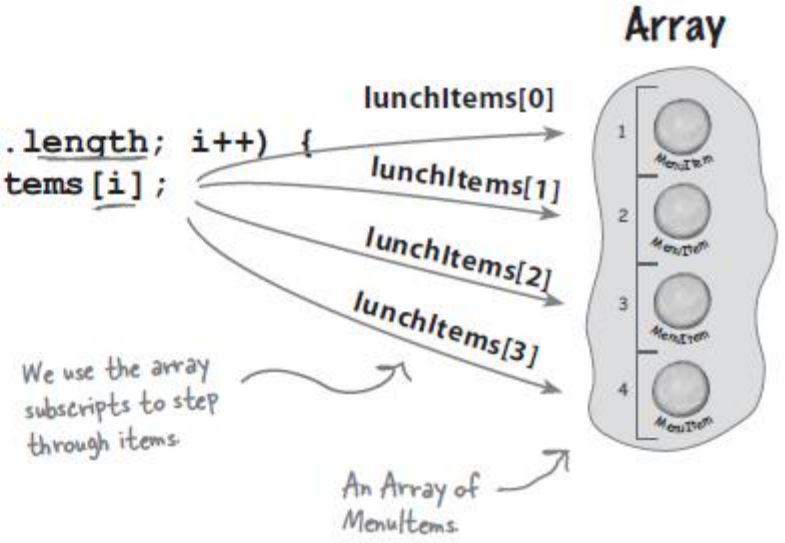
```
printMenu()
   - prints every item on the menu

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 items is vegetarian, otherwise,
     returns false
```

Individual Access

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

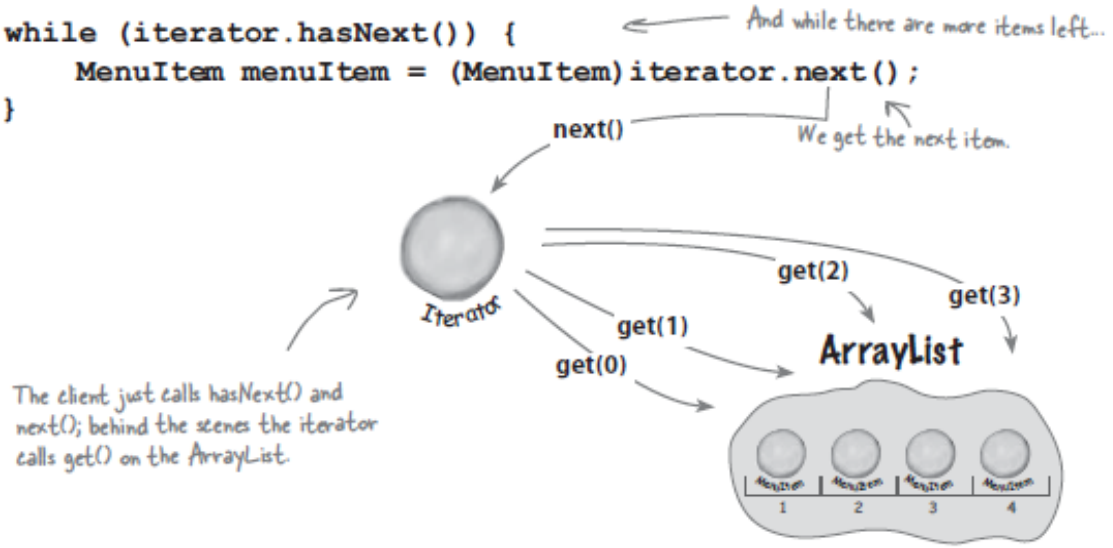



```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

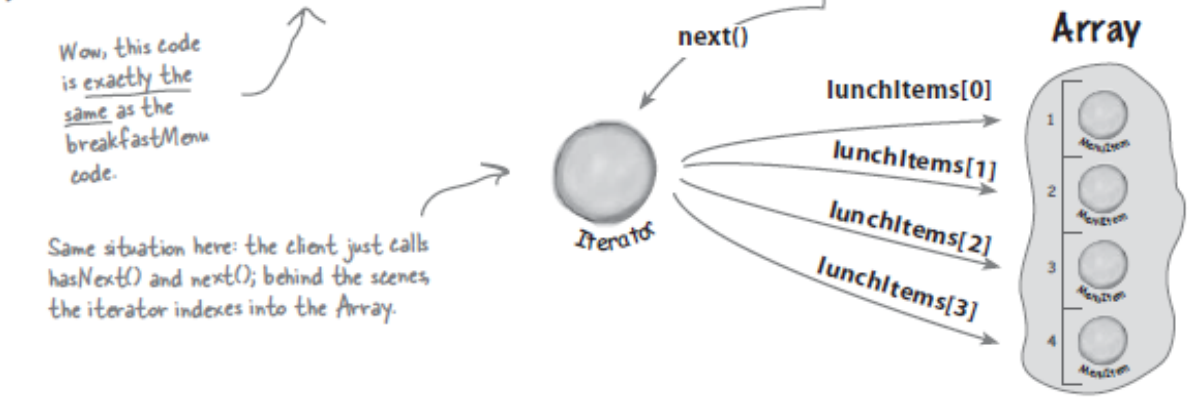


Iterator Access

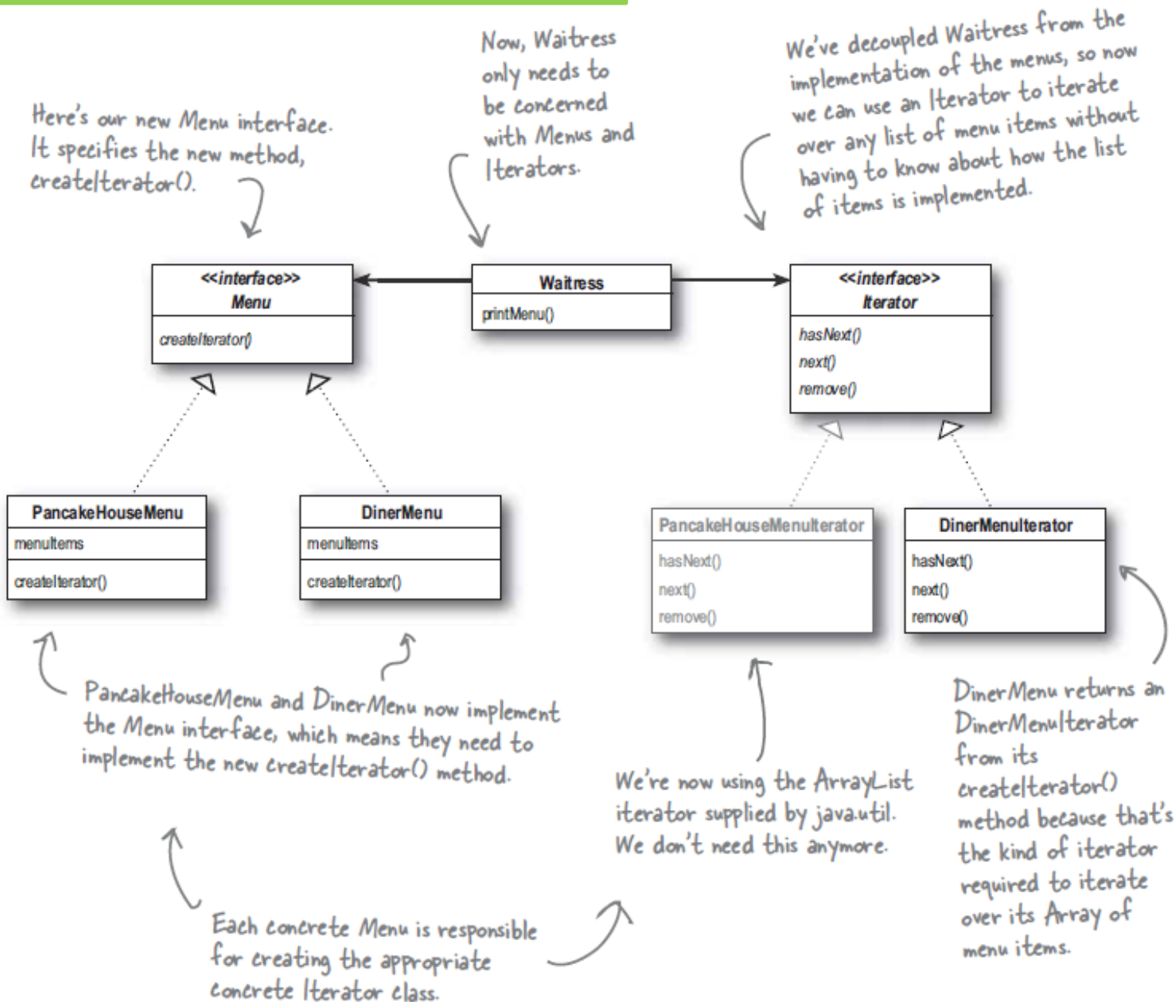
```
Iterator iterator = breakfastMenu.createIterator();

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




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

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

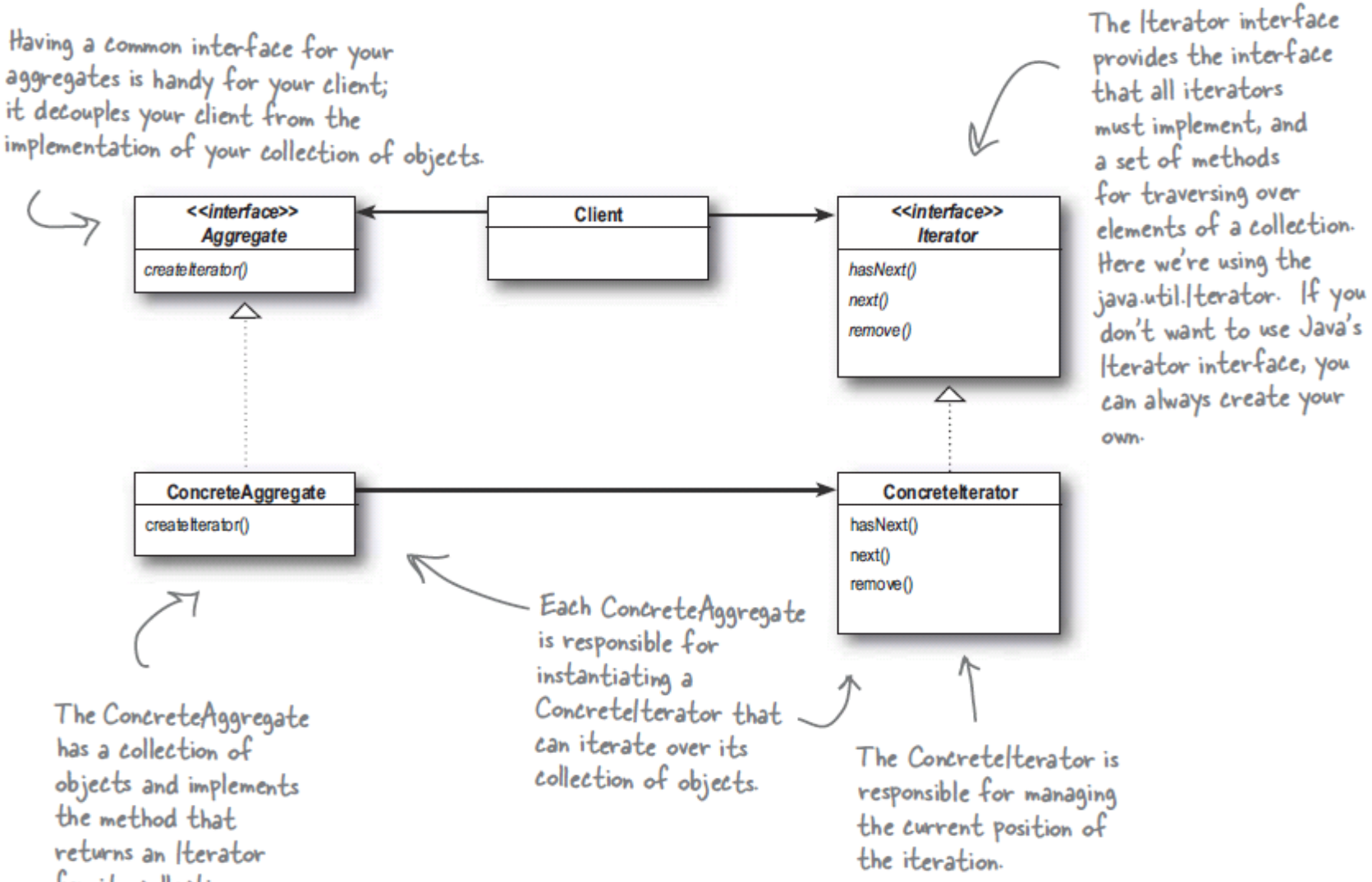


What did we do?

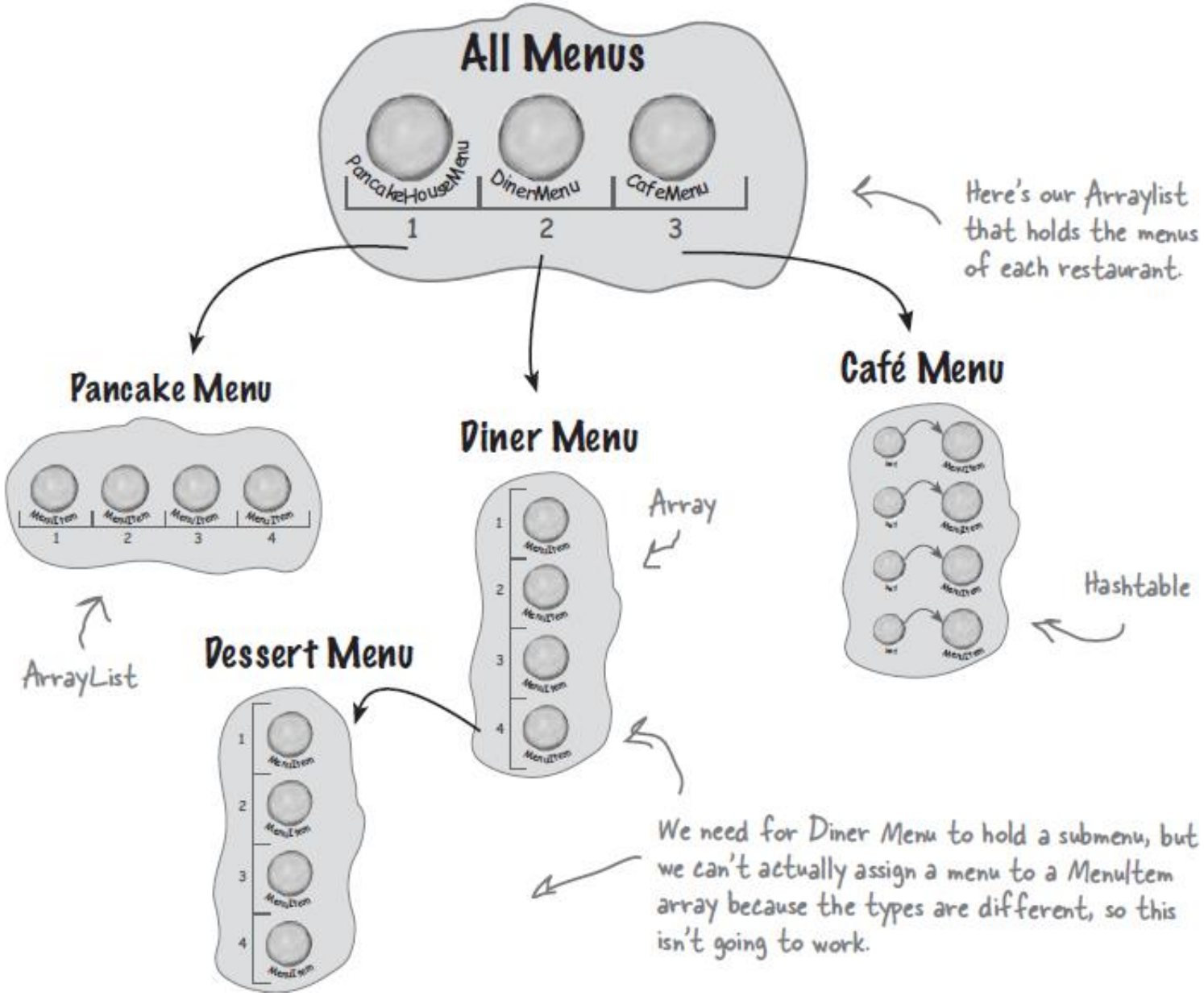


Iterator Pattern

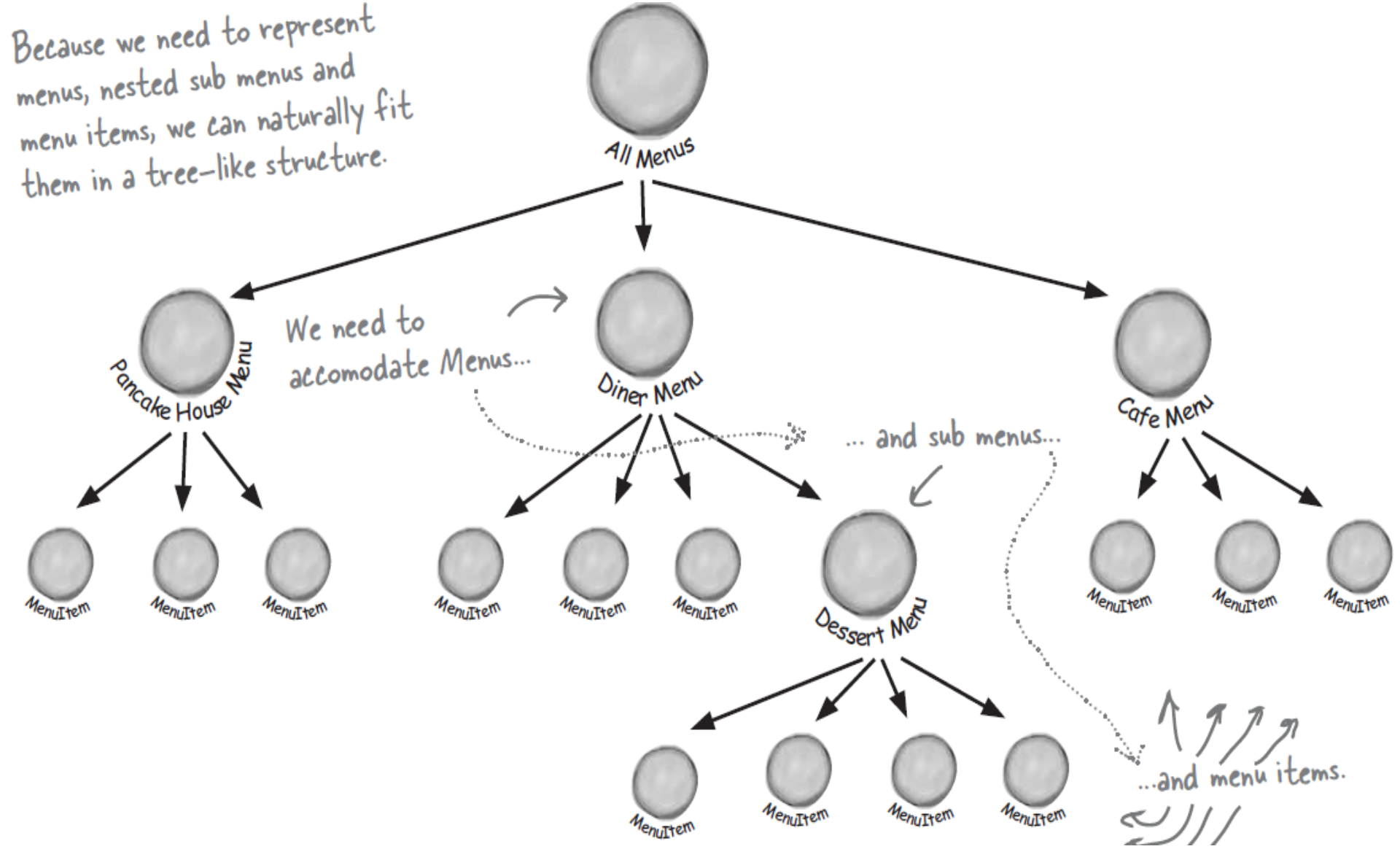
The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



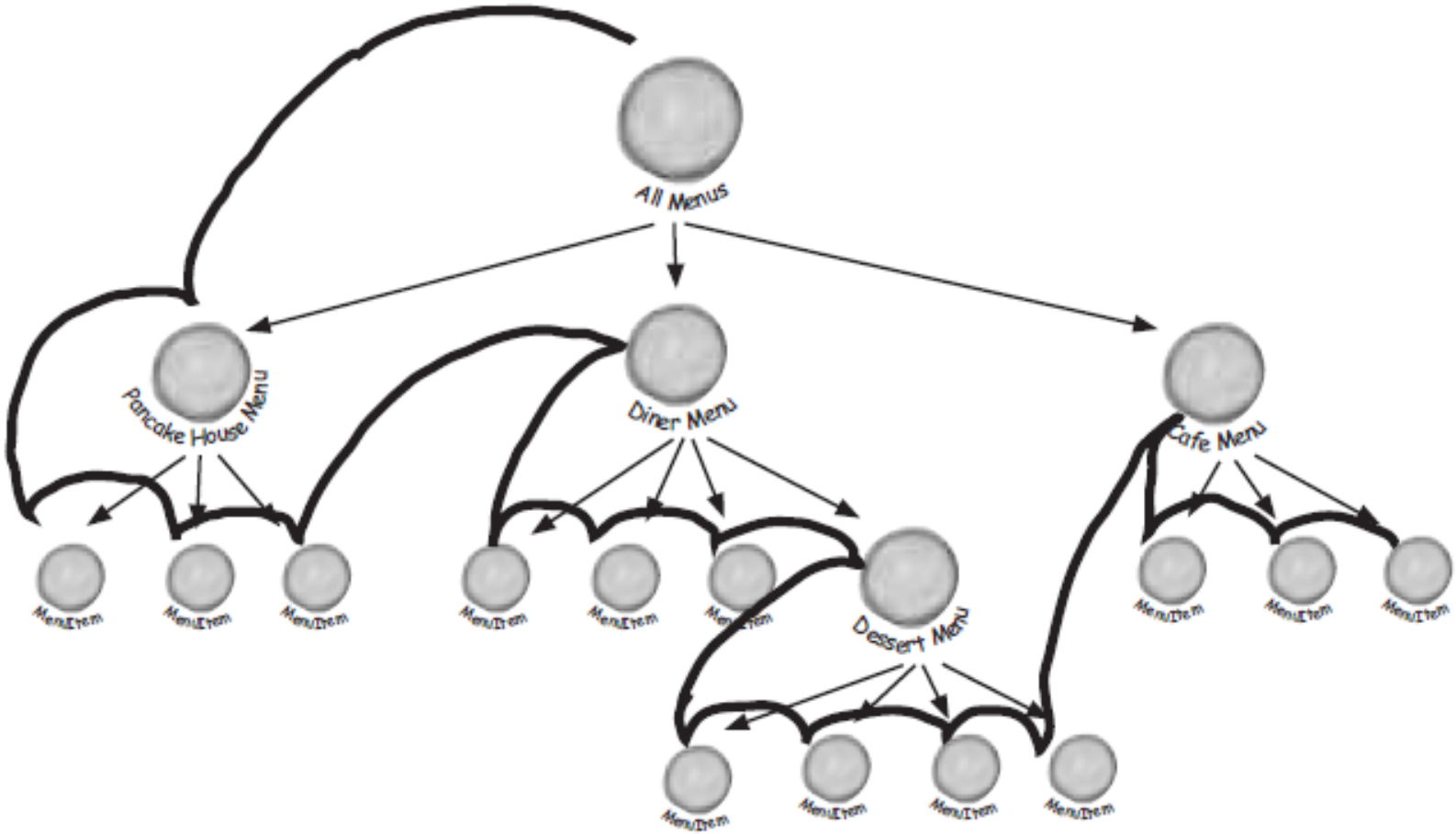
Pattern 9: Change



Representation

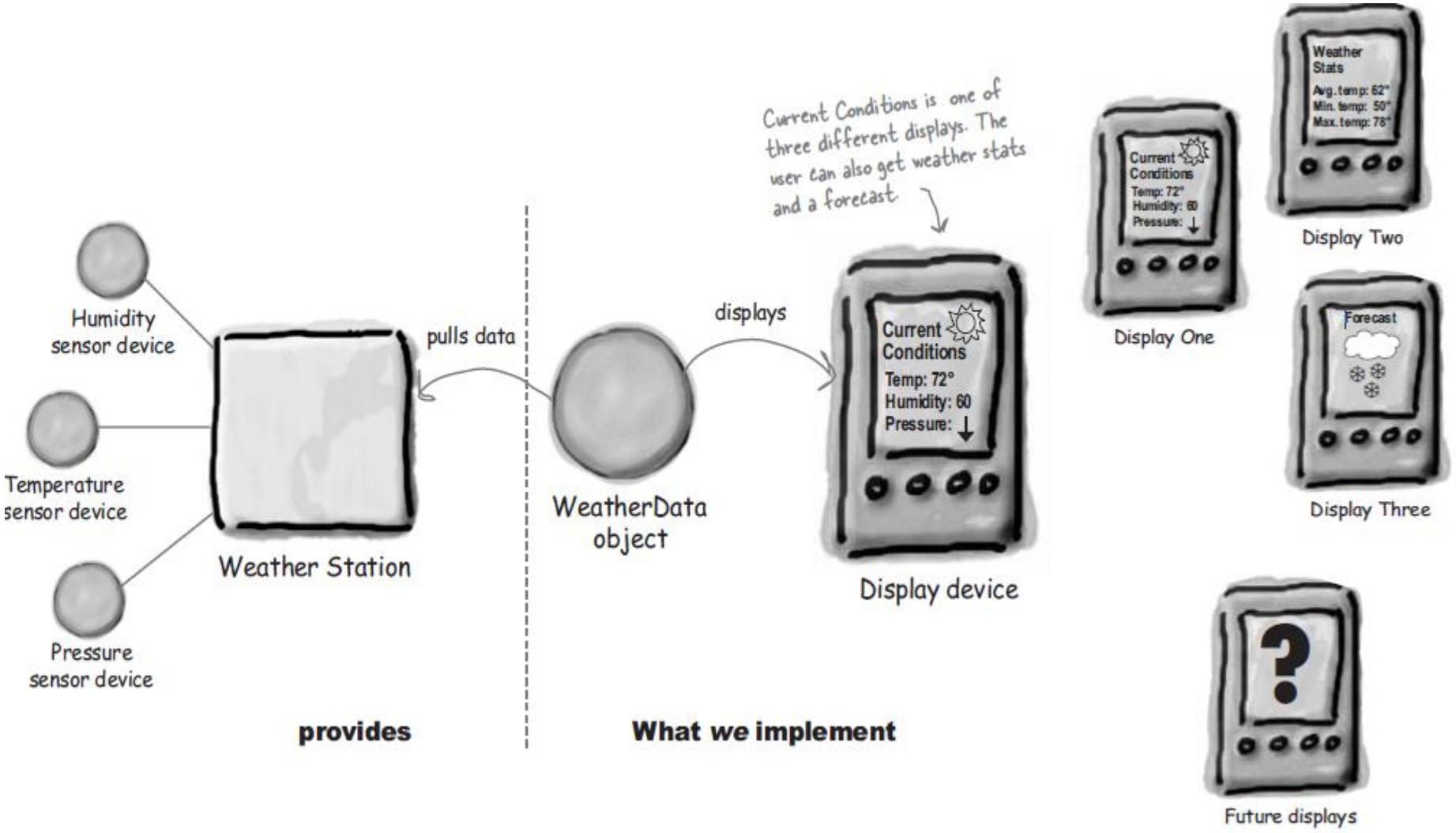


Traverse


All Menus
Pancake House Menu
Diner Menu
Cafe Menu
Dessert Menu
MenuItem

Pattern 10

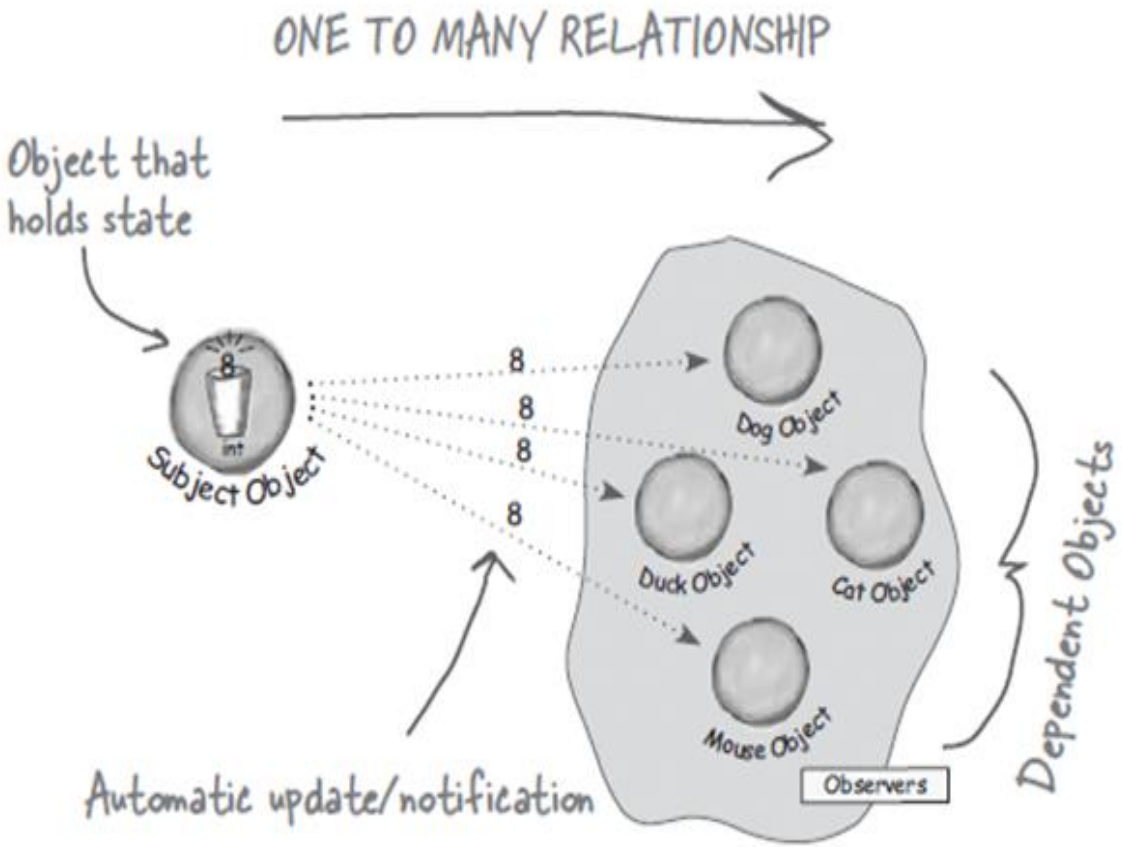
The Weather Monitoring application overview



Observer Pattern

Like
News Paper

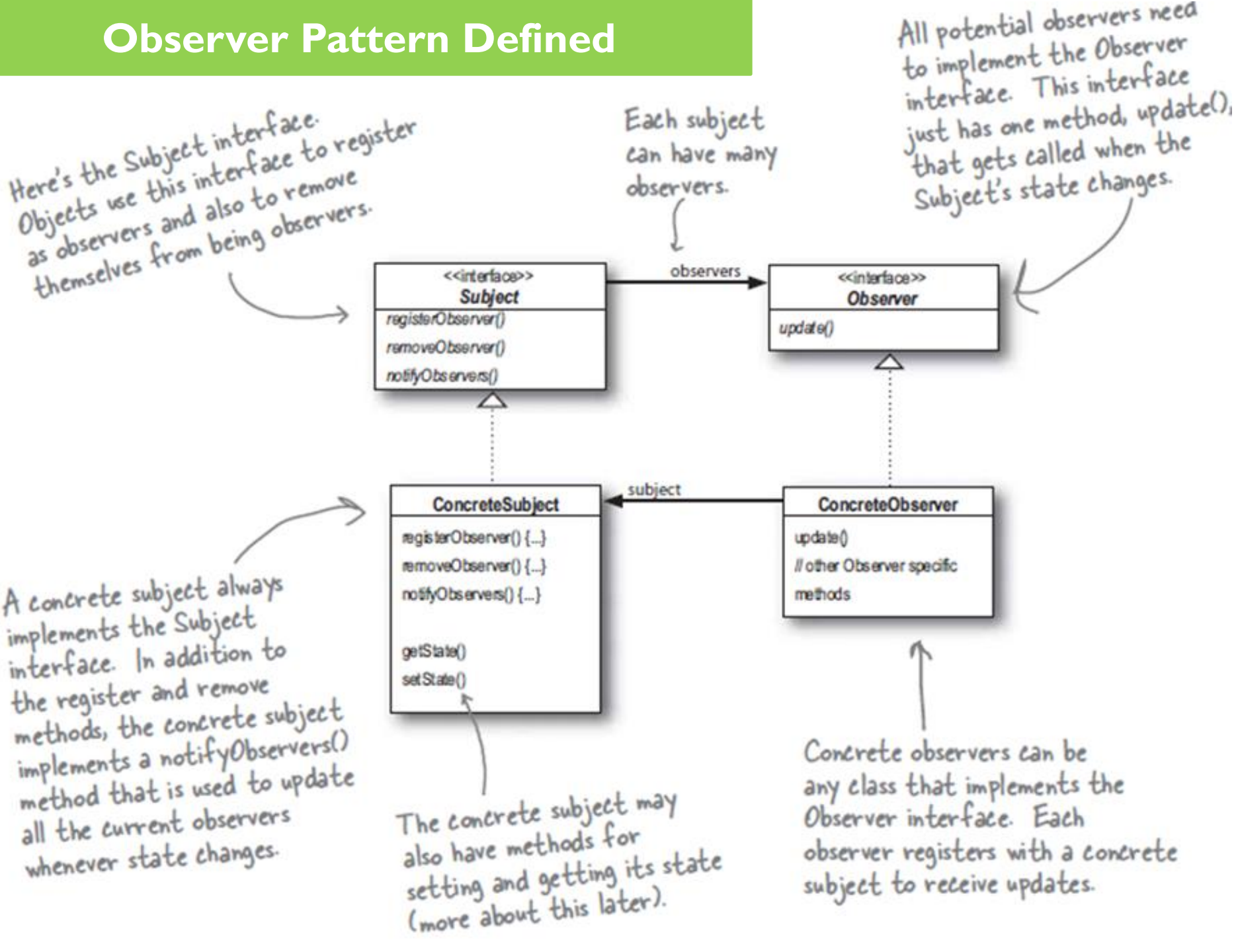
Publishers + Subscribers = Observer Pattern



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

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

Observer Pattern Defined



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.

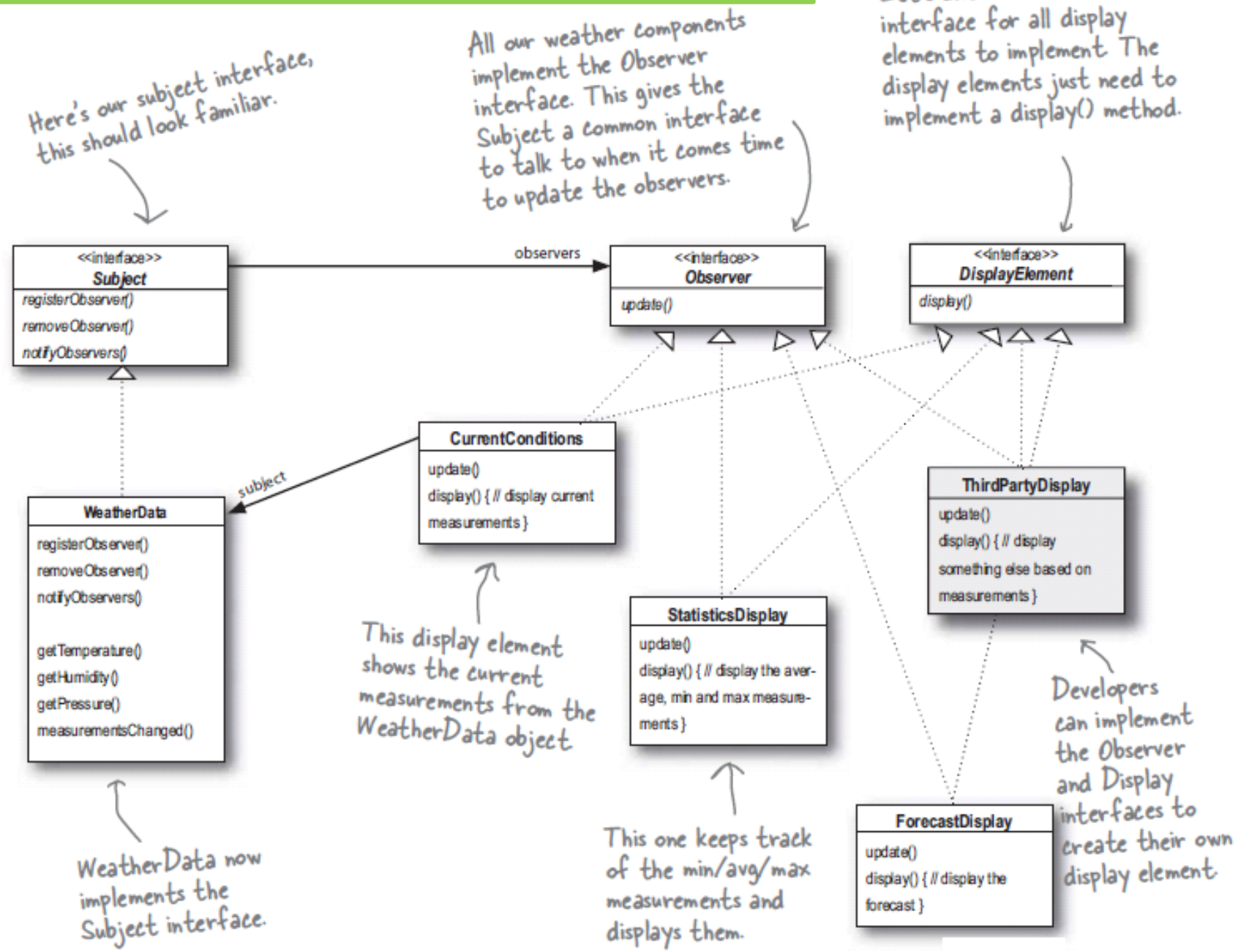
Changes to either the subject or an observer will not affect the other.

Design Principle

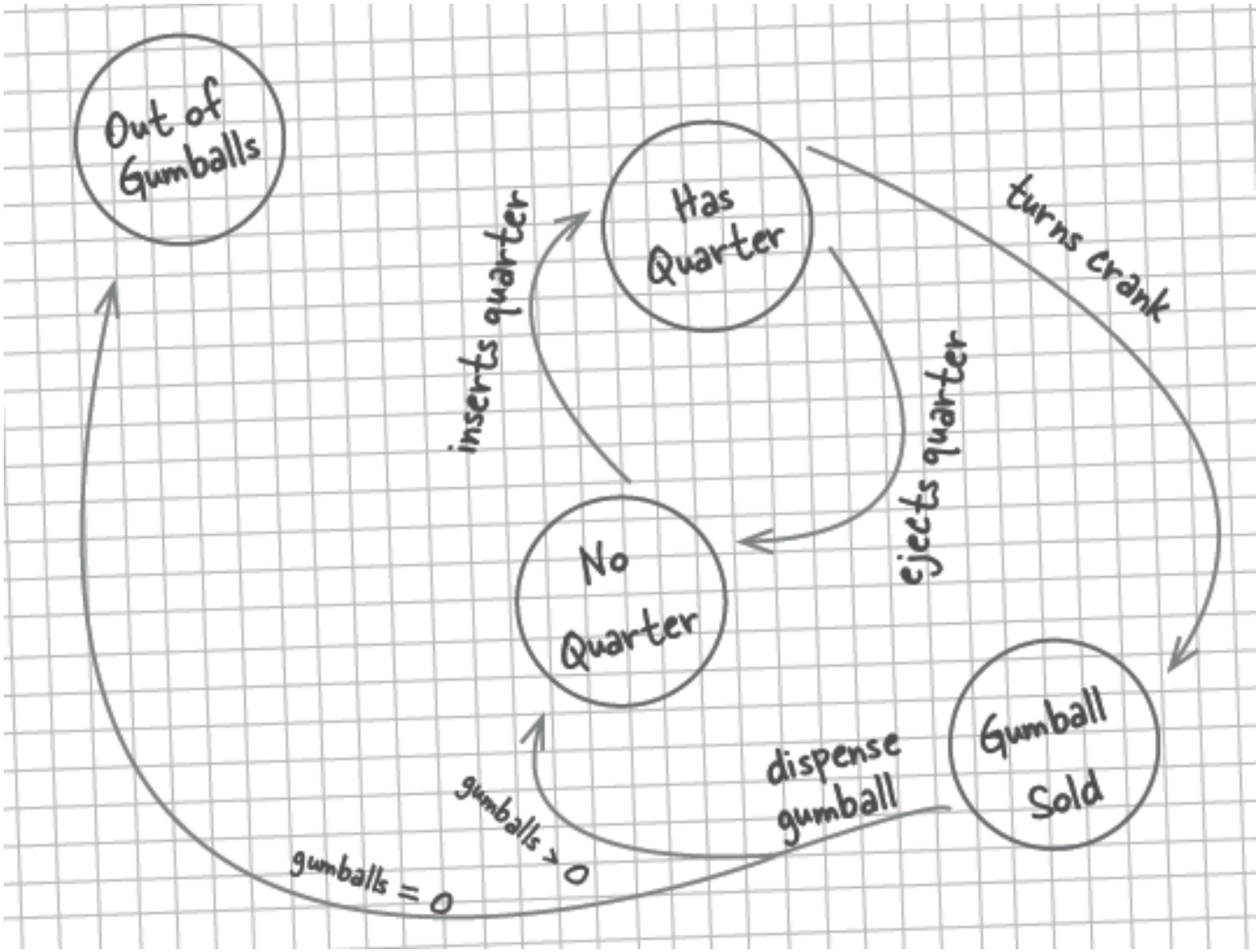
Strive for loosely coupled designs between objects that interact.

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

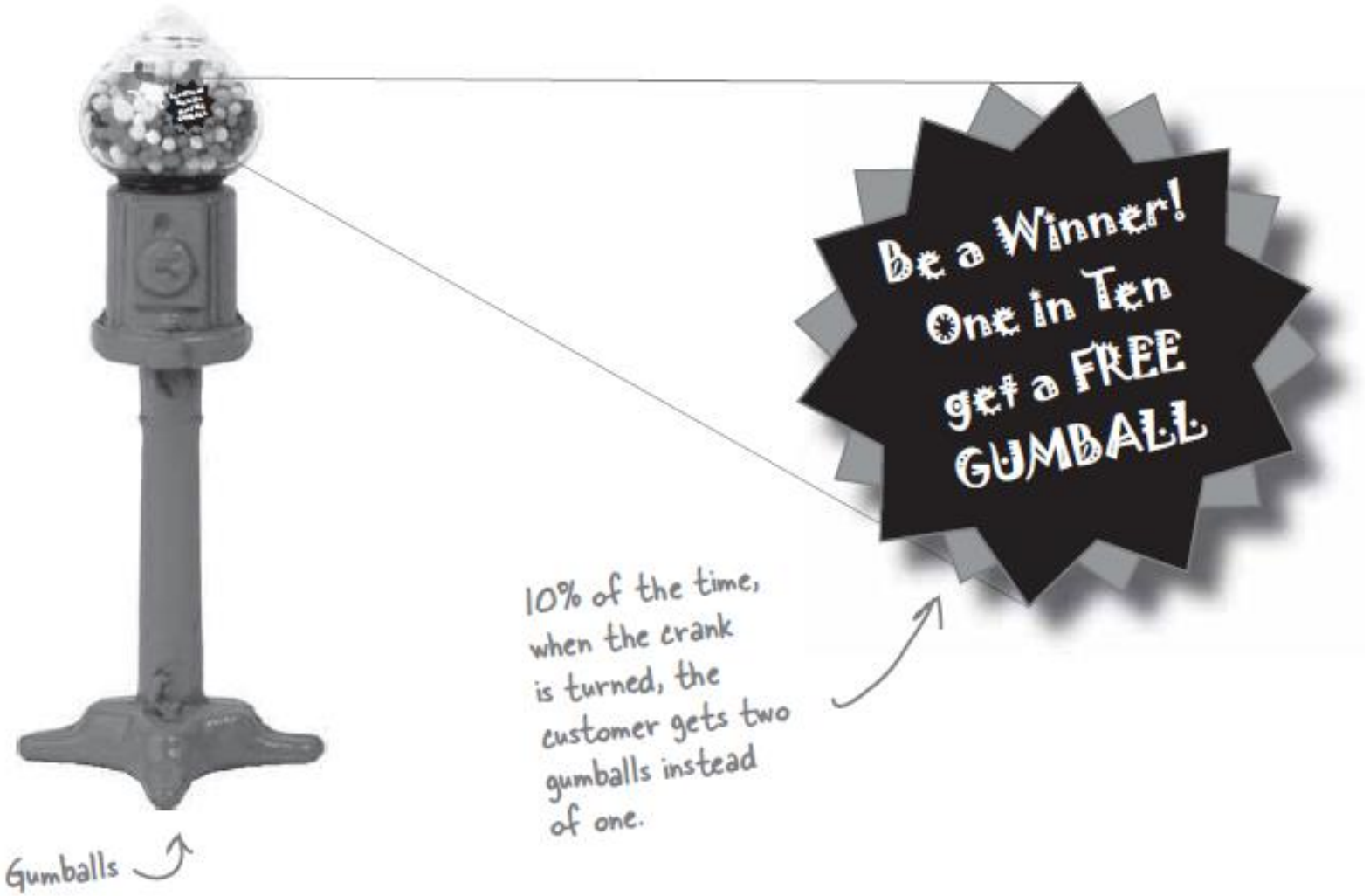
Observer Pattern: Implementation



Pattern 11

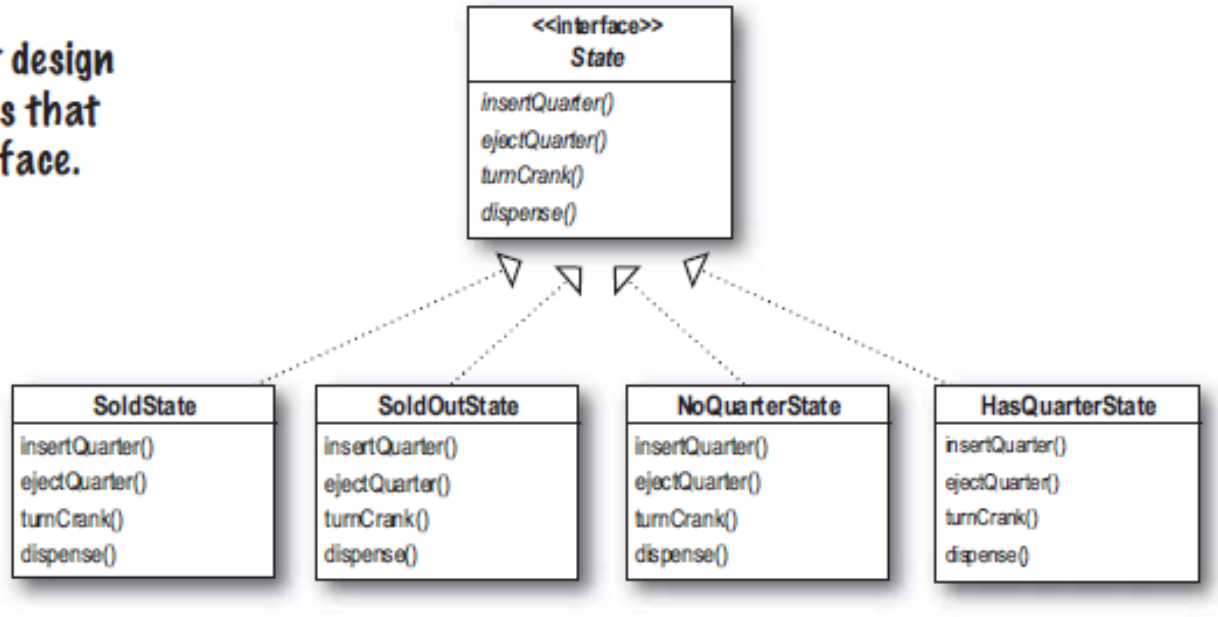

Out of Gumballs
Has Quarter
No Quarter
Gumball Sold
inserts quarter
turns crank
ejects quarter
dispense gumball
gumballs > 0
gumballs = 0

Change


Be a Winner!
One in Ten
get a FREE
GUMBALL
10% of the time, when the crank is turned, the customer gets two gumballs instead of one.
Gumballs

What did we do?

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



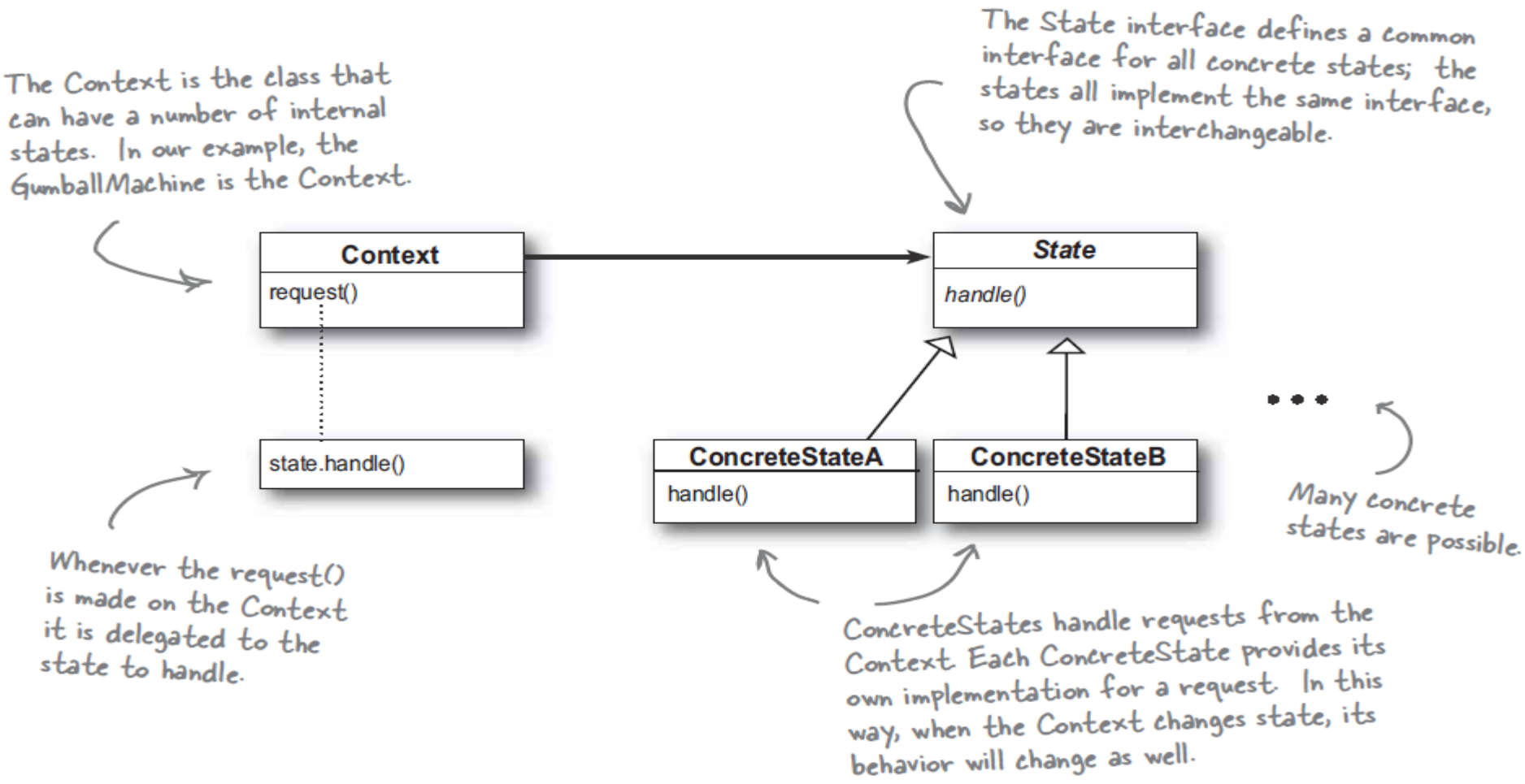
To figure out what states we need, we look at our previous code...

```
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.

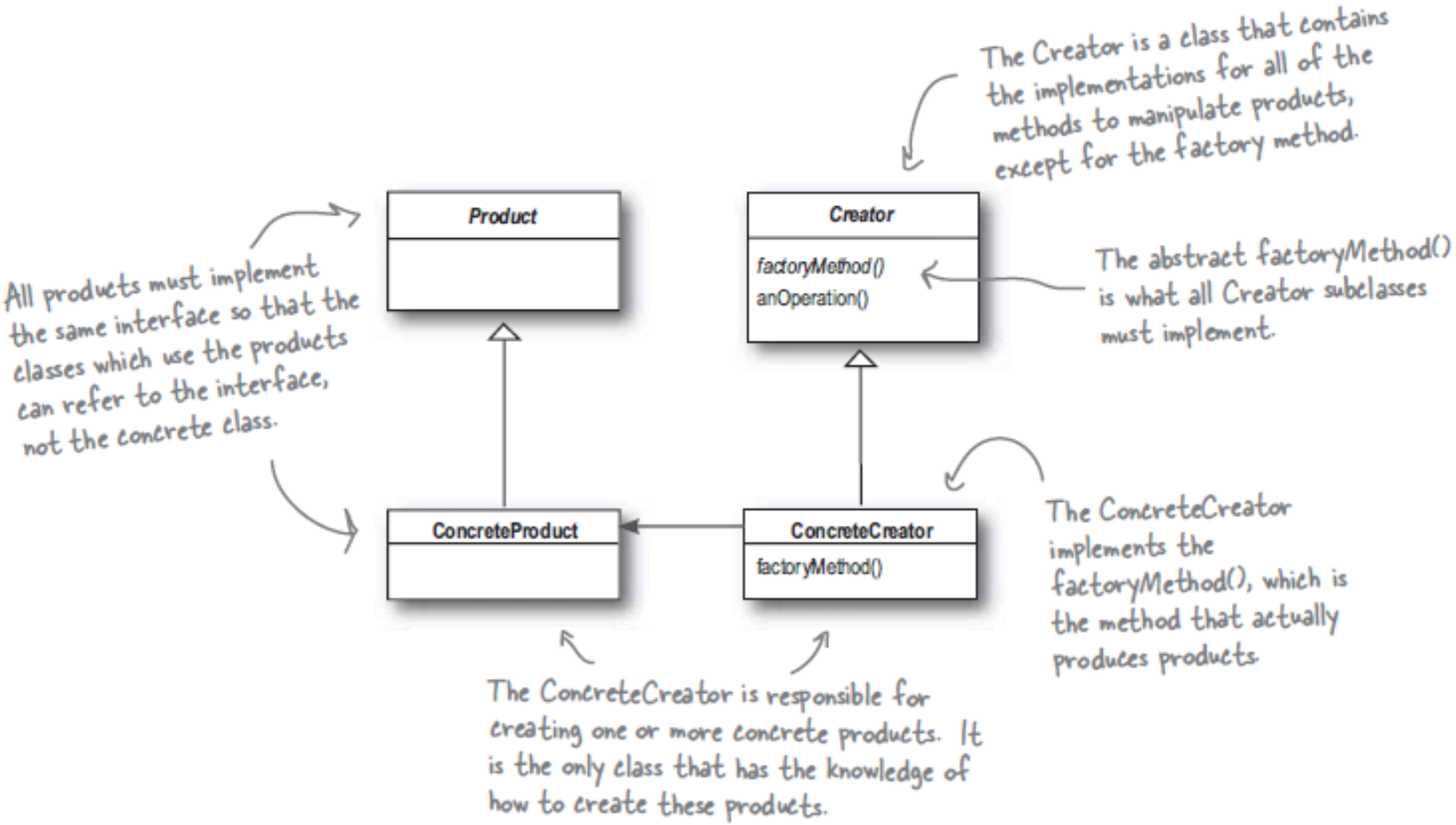
State Pattern

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



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.

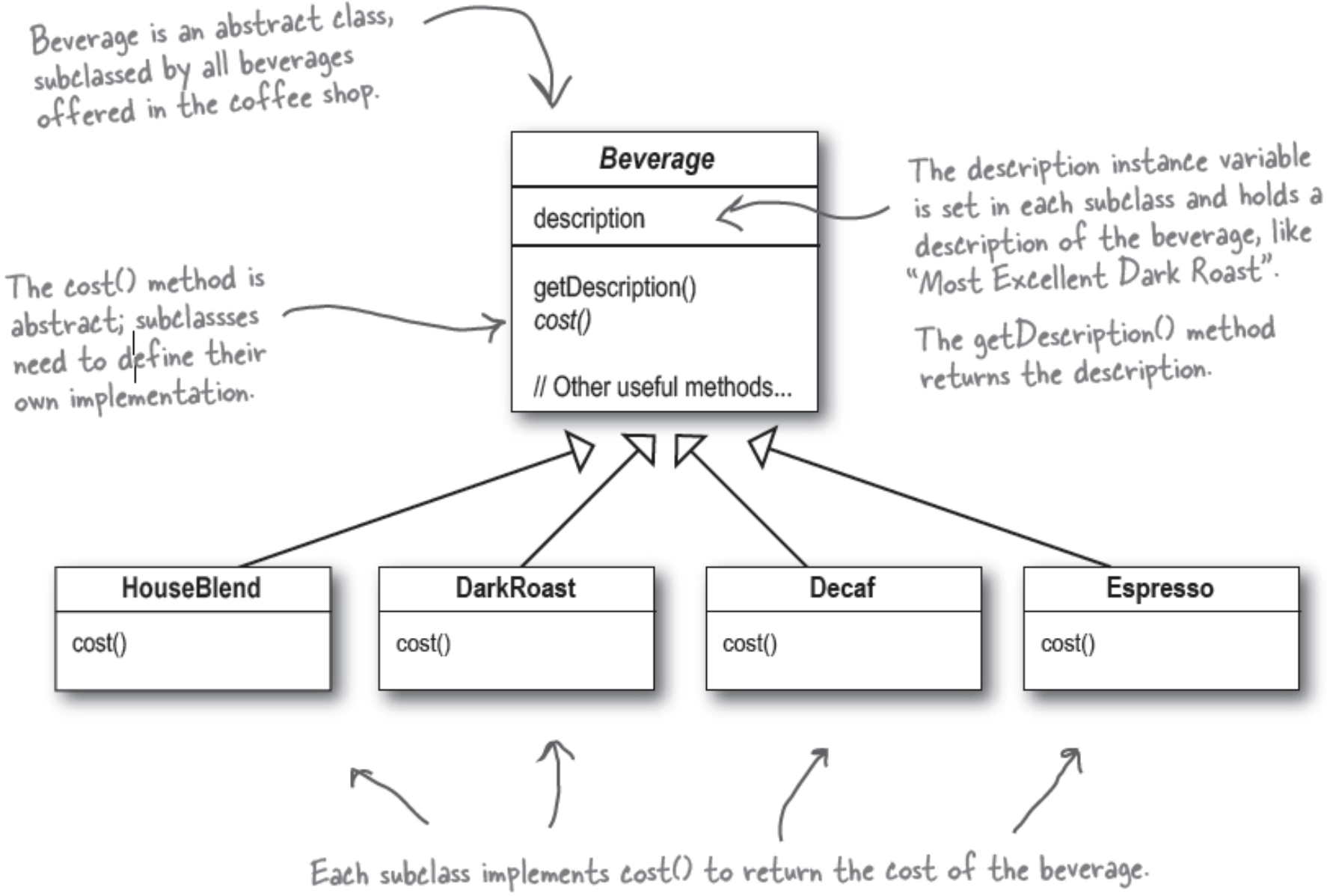


Welcome to Starbuzz Coffee

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

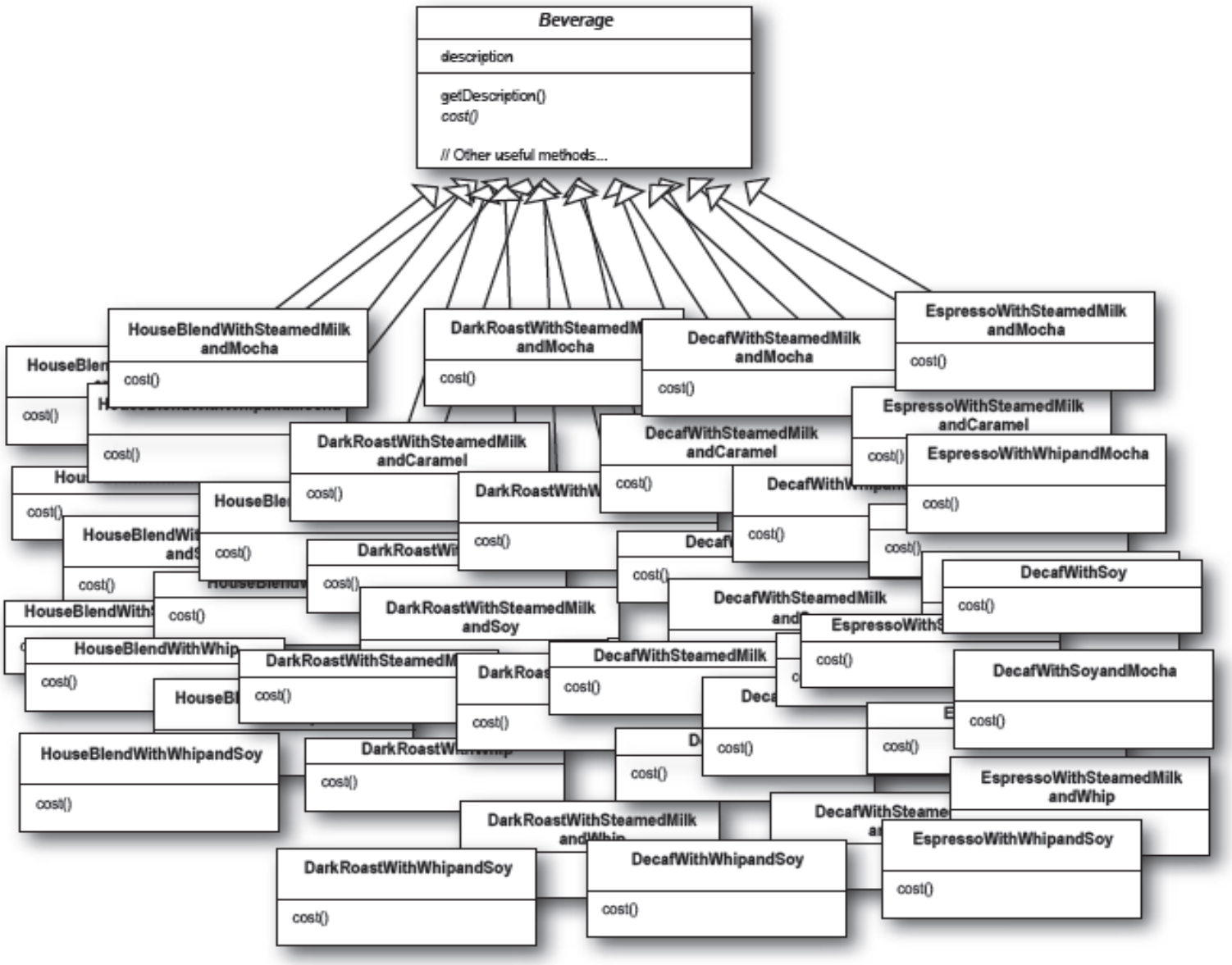


CHANGE – Add Condiments

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?

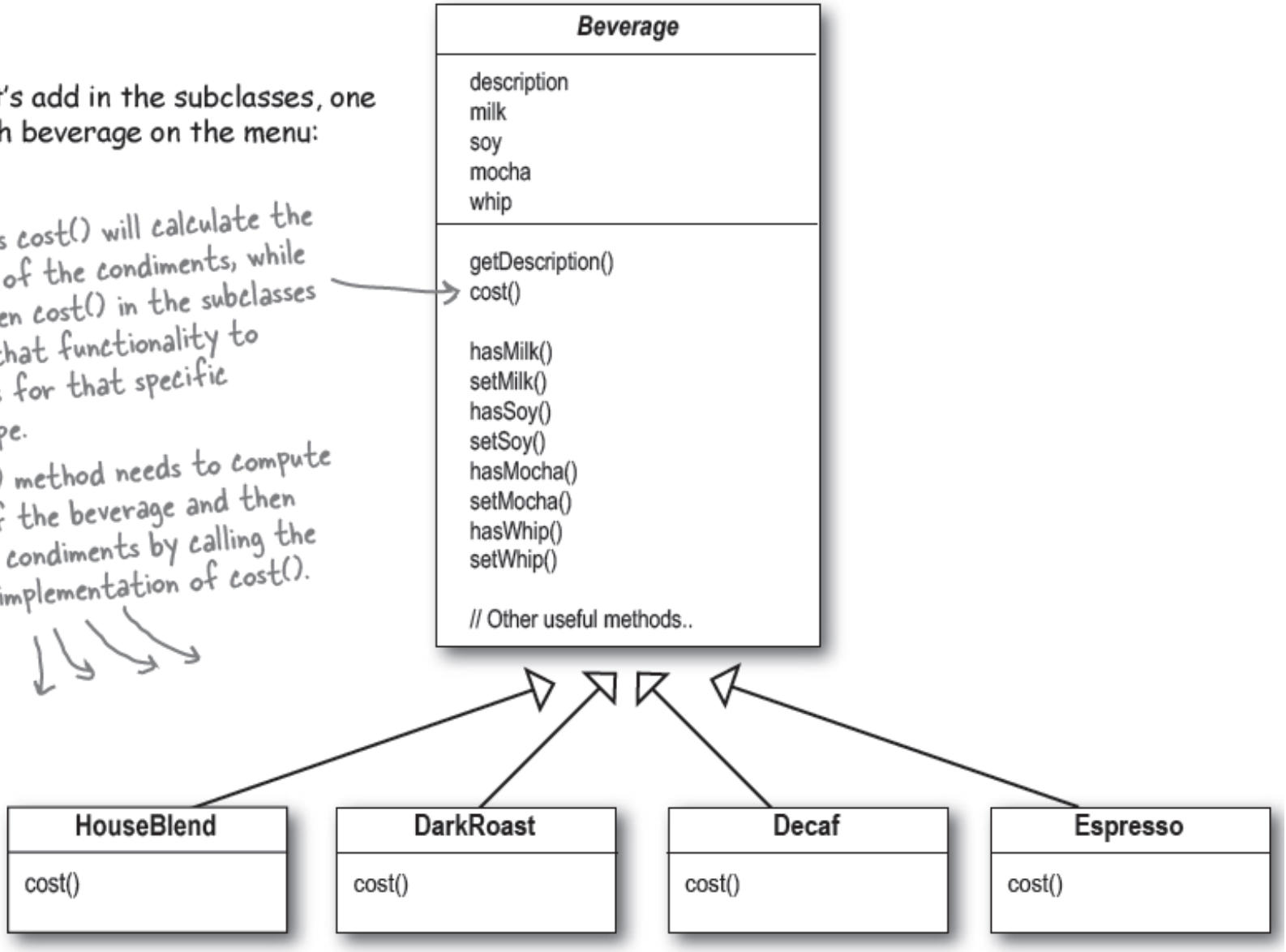

Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

How about this?

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

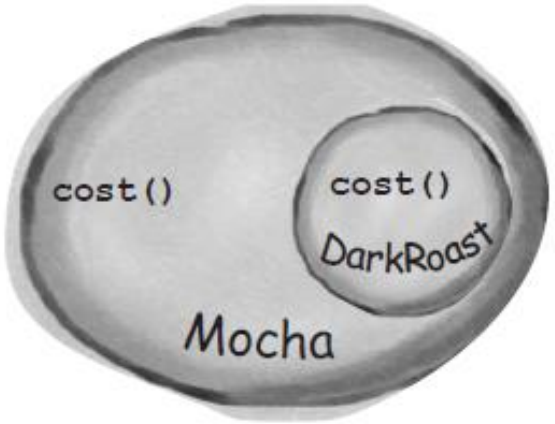
The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

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



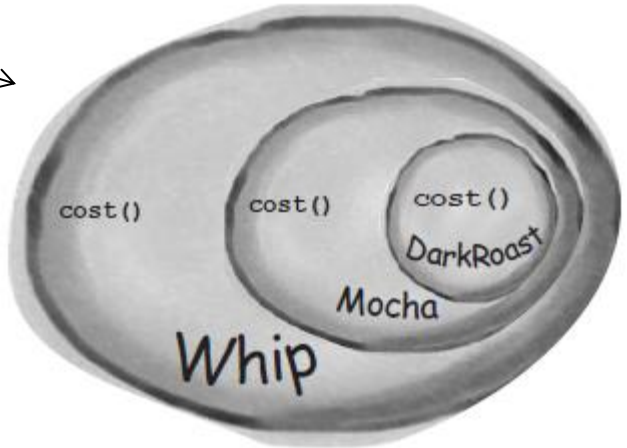
Decoration

1. Take a DarkRoast object



2. Decorate it with a Mocha object

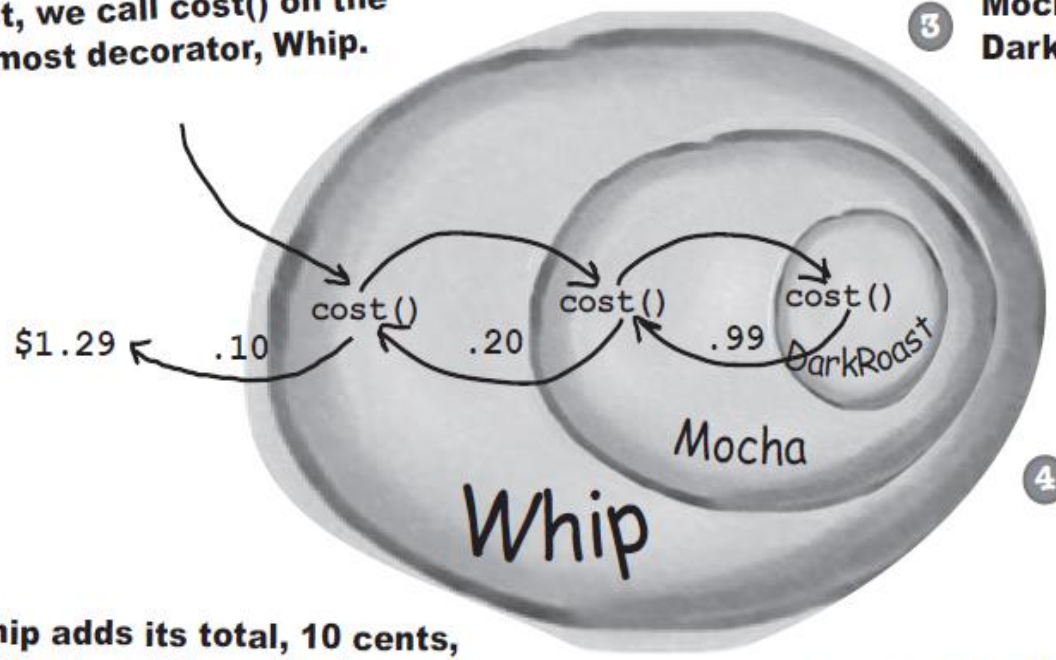
3. Decorate it with a Whip object



4. Call the cost() method and rely on delegation to add on the condiment costs

Decorator Used

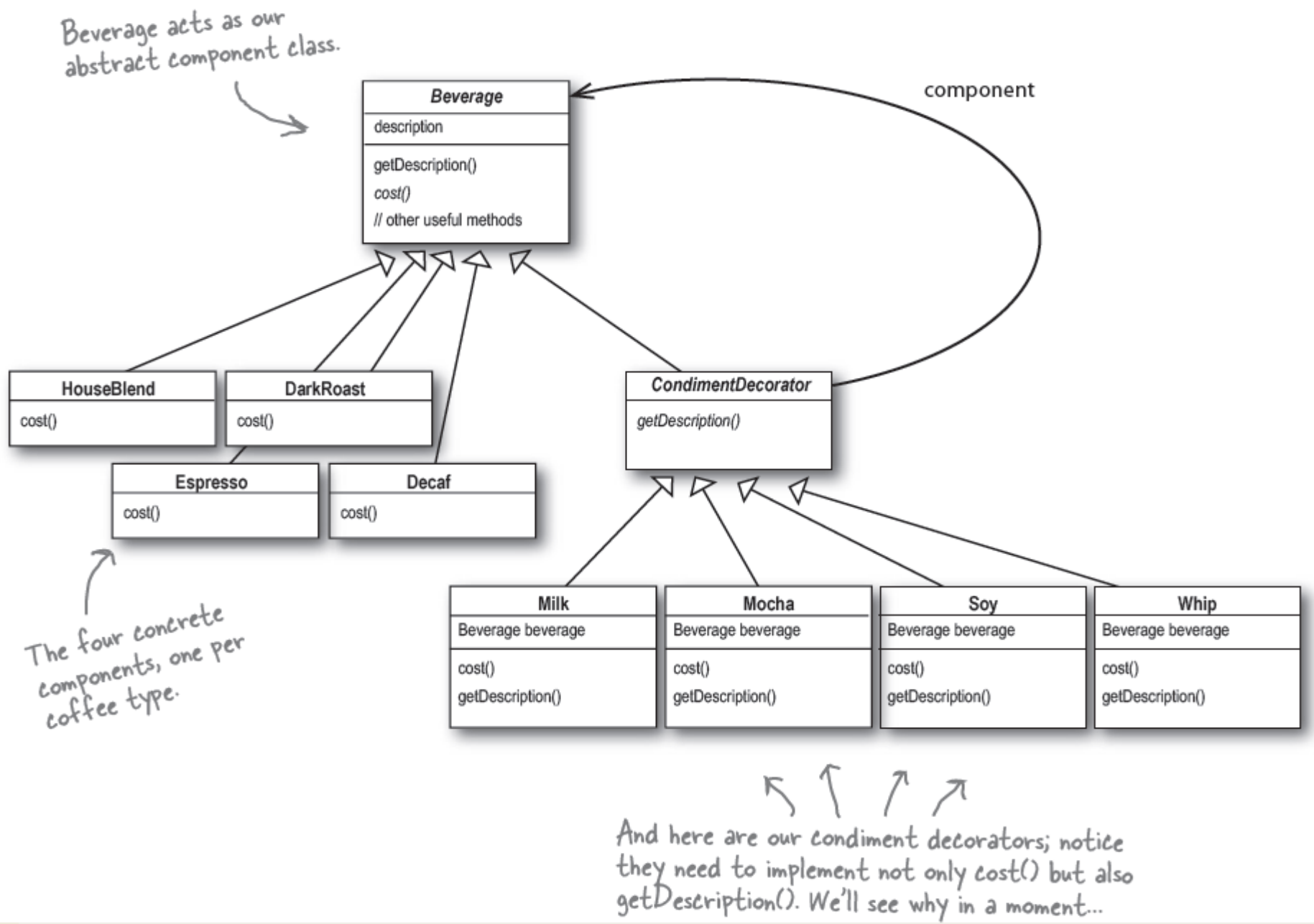
1. Take a DarkRoast object
2. Decorate it with a Mocha object
3. Decorate it with a Whip object
4. Call the cost() method and rely on delegation to add on the condiment costs



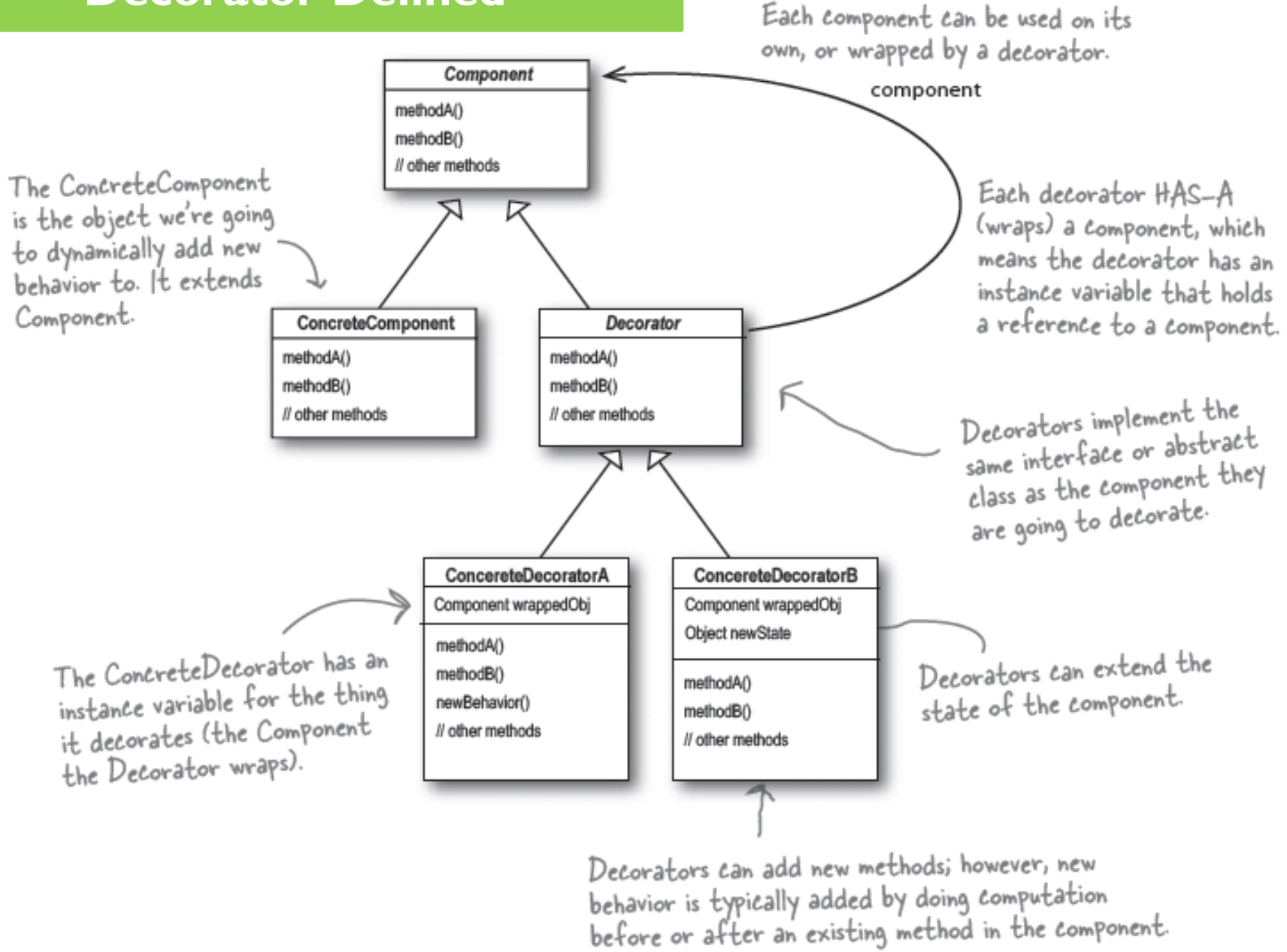
Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

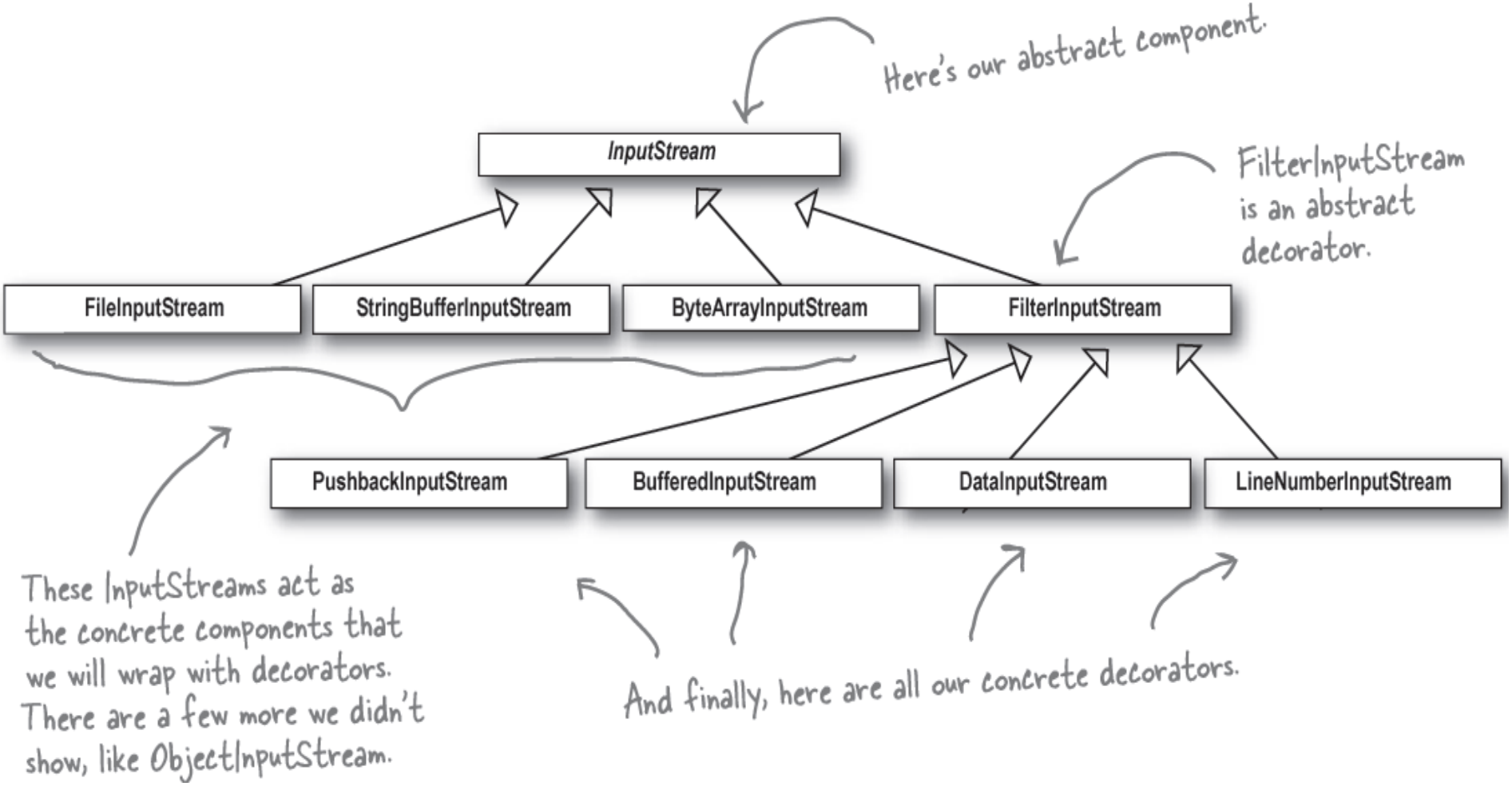
Apply to Star Buzz



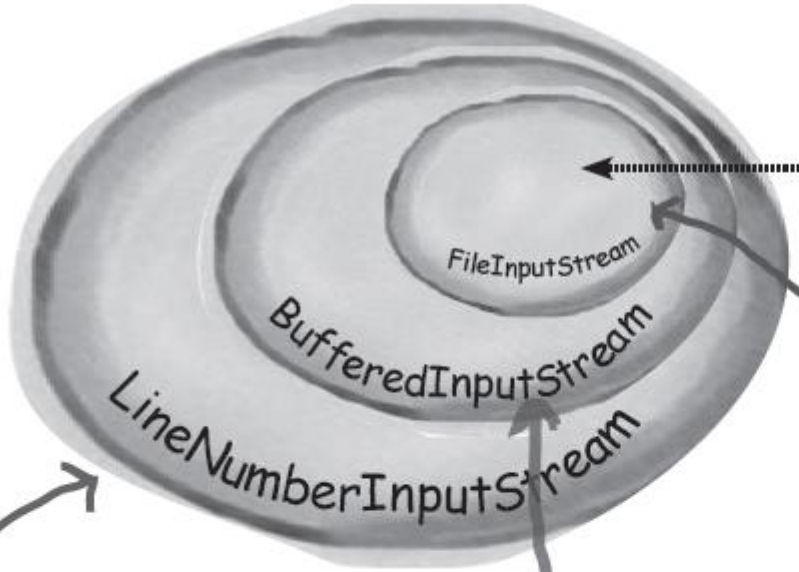
Decorator Defined



File I/O Decorators in Java



File I/O Decorators



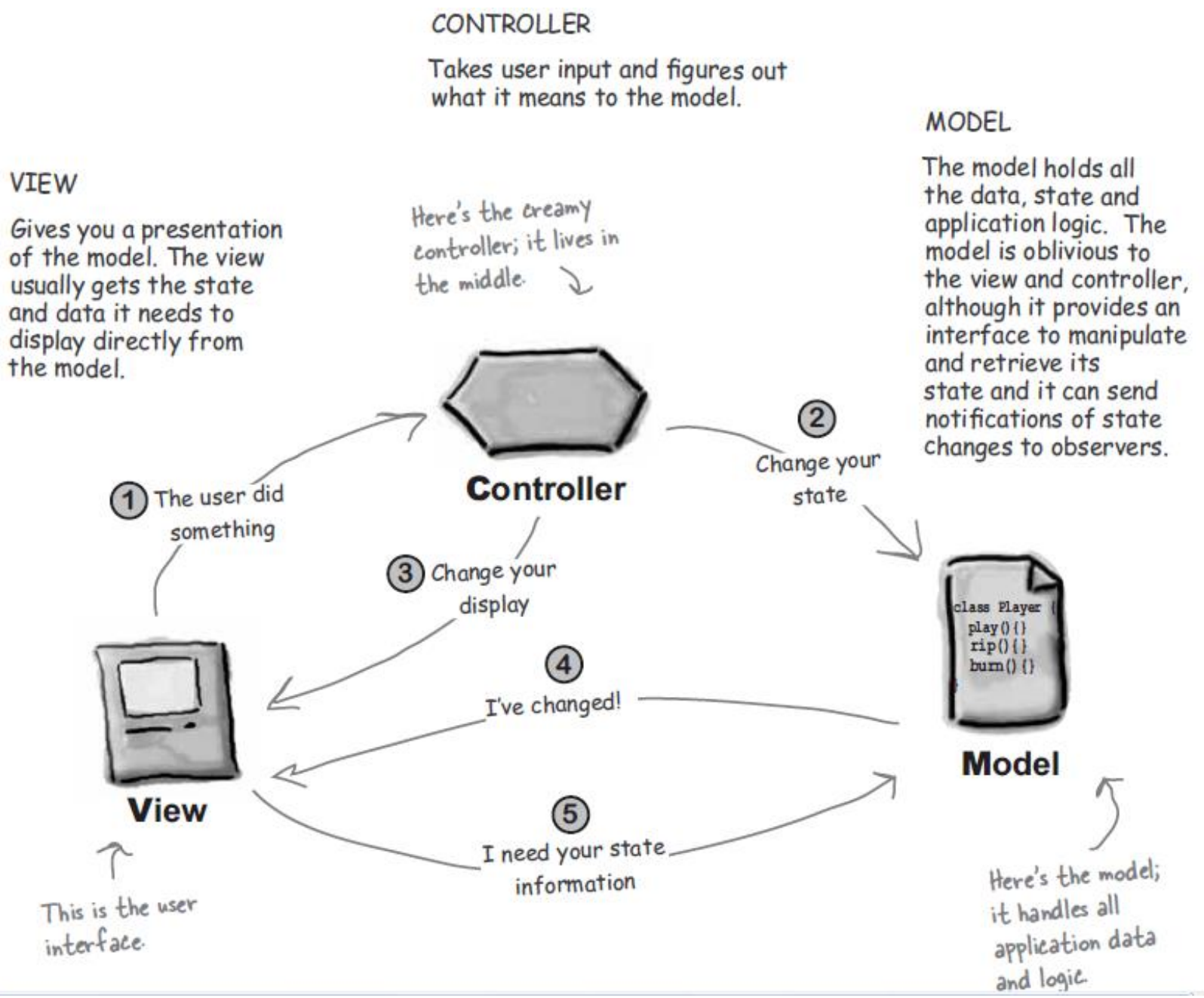
A text file for reading.

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.

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.

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

Pattern of Patterns

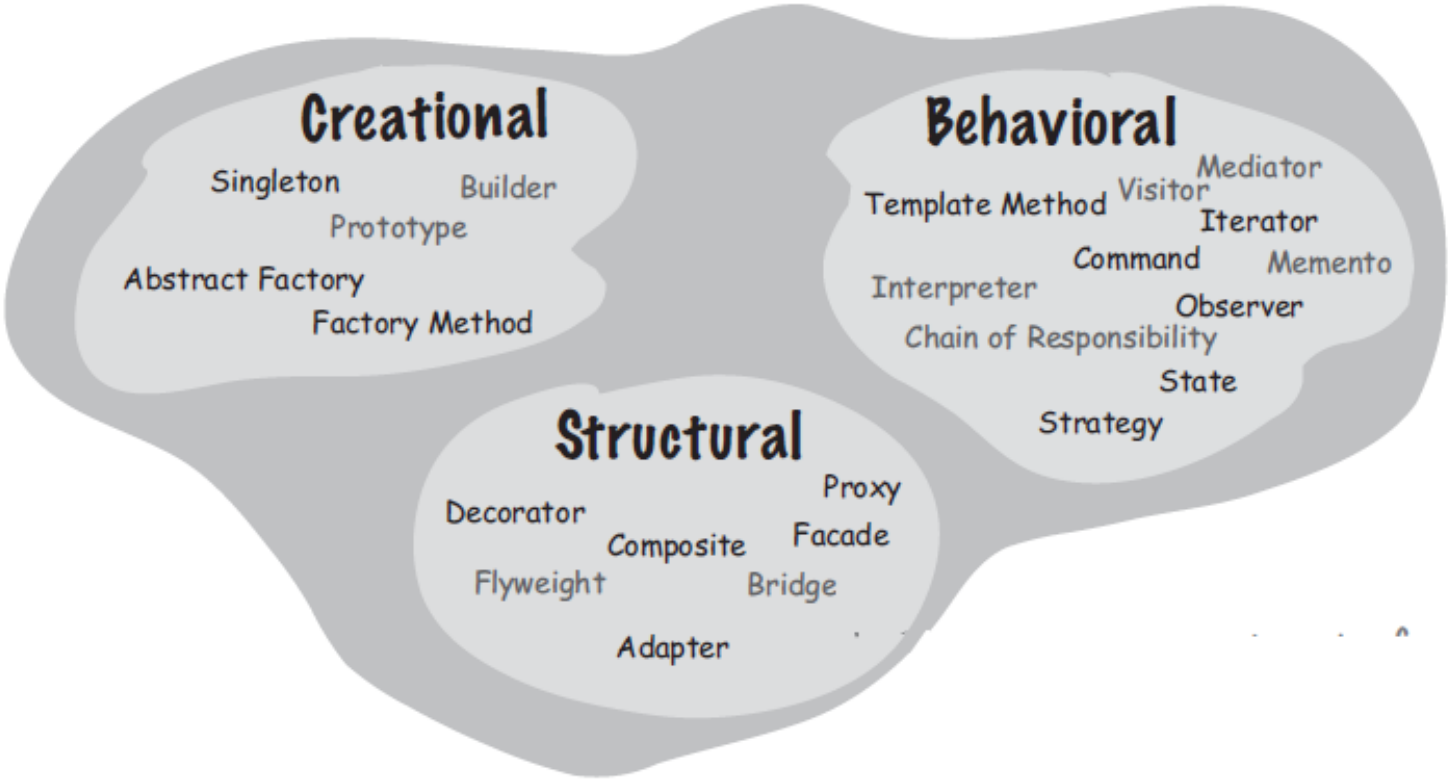


Summary…

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.

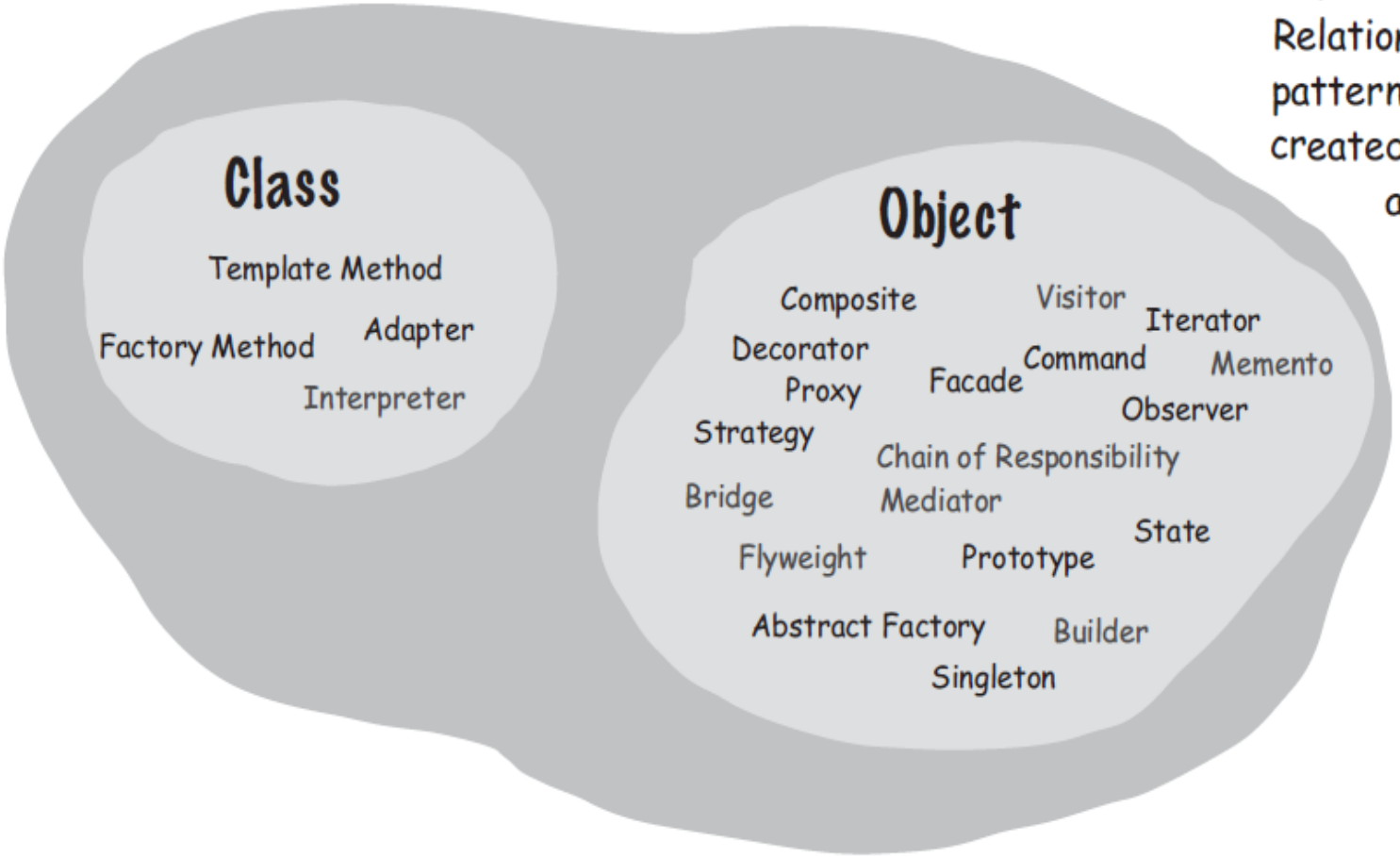


Structural patterns let you compose classes or objects into larger structures.

Class and Object Patterns

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

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

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.

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

INTERMEDIATE MIND

"Maybe I need a Singleton here."

ZEN MIND

"This is a natural place for Decorator."

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.

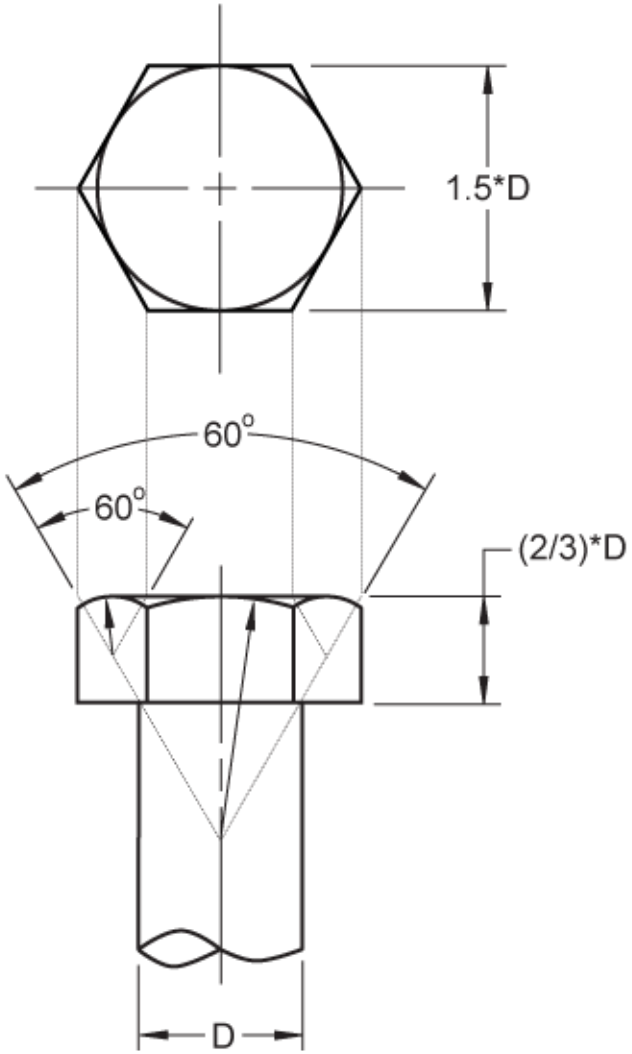
UML

With StarUml

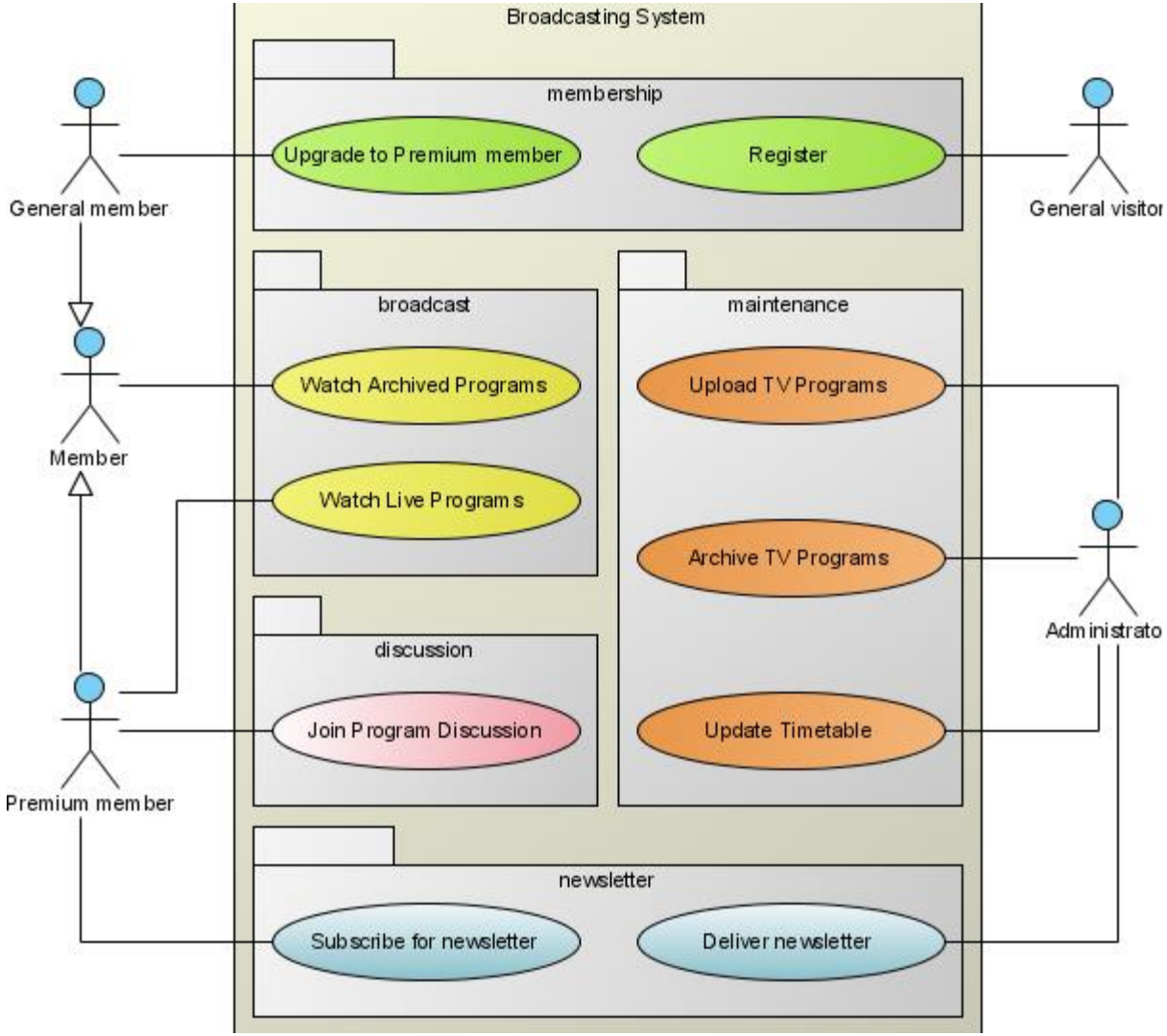
Credits:

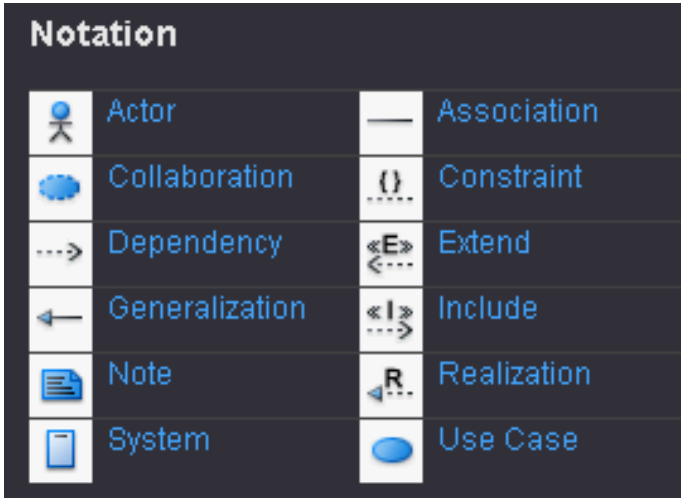
http://www.uml-diagrams.org
http://www.visual-paradigm.com/support/documents

The need for UML

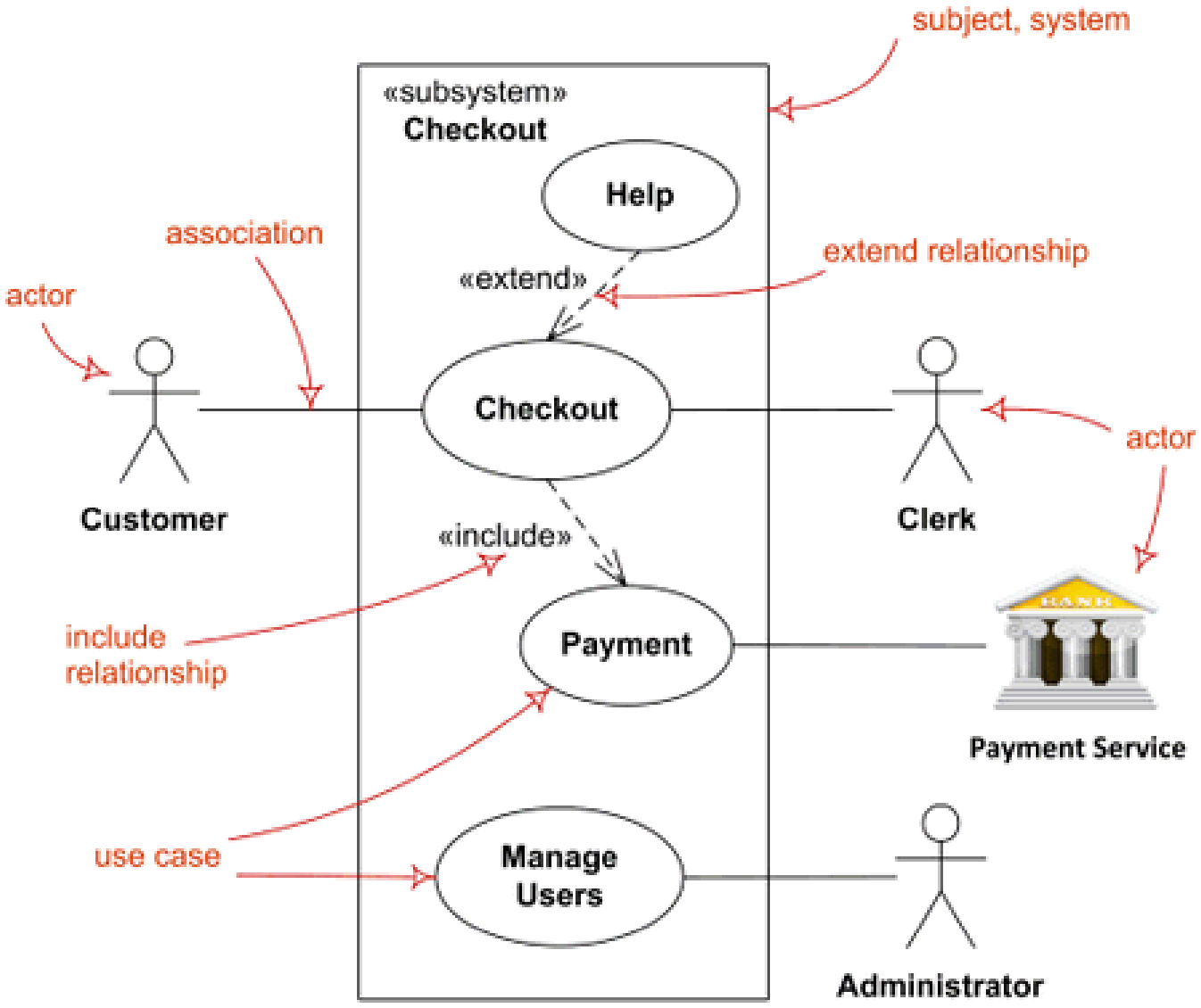


Use Case Diagram

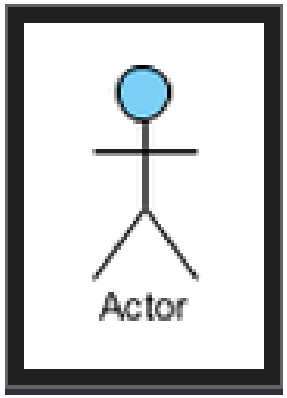


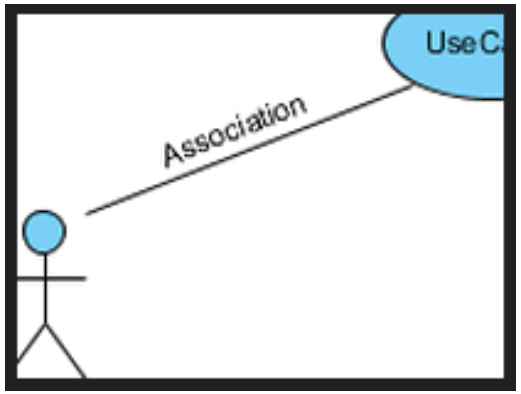


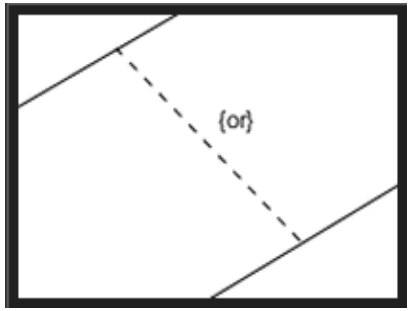
Use Case Diagram Reference

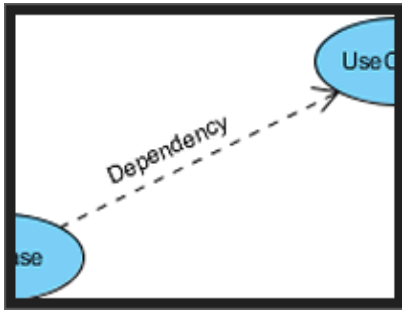


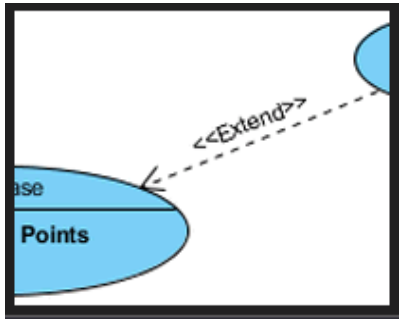
Use Case Diagram (Notations)

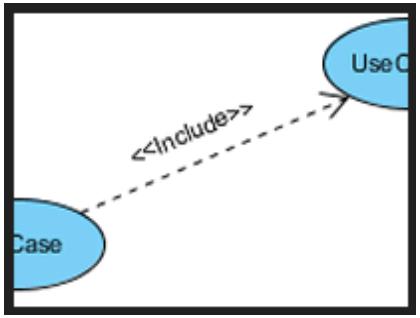


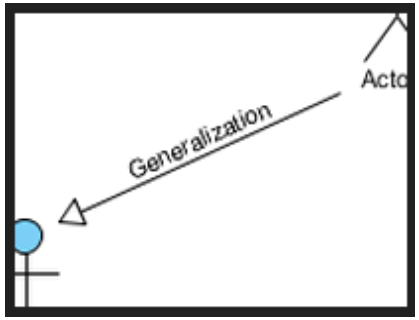


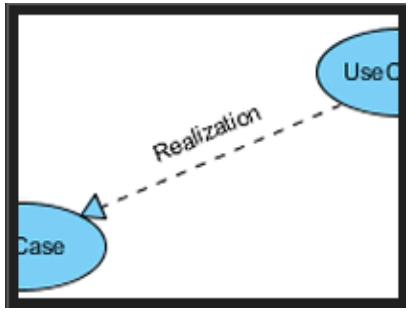


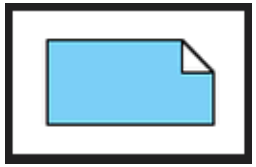


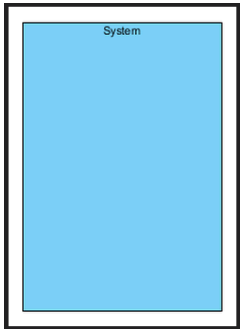


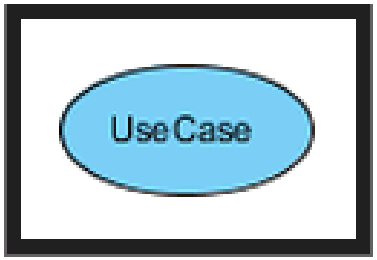




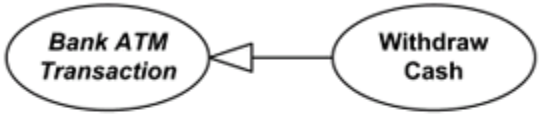
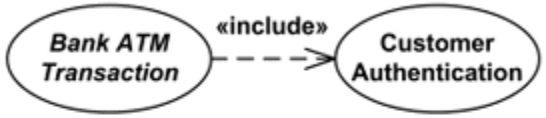


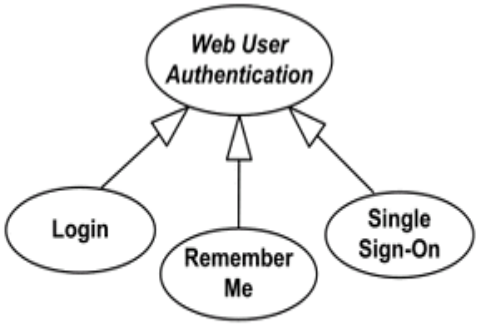




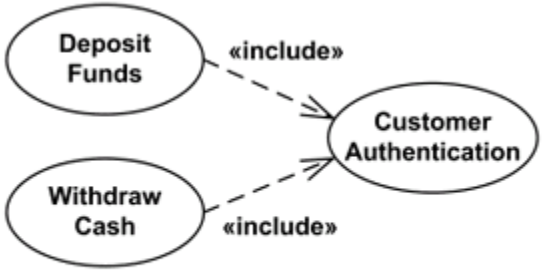


Use Case Diagram (Relationships)

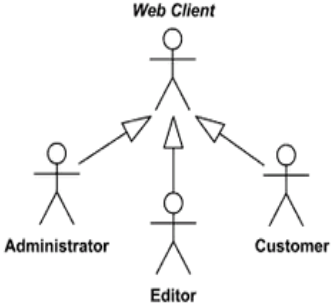
Generalization	Extend	Include
		
Base use case could be **abstract use case** (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (**abstract use case**).
Specialized use case is required, not optional, if base use case 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.



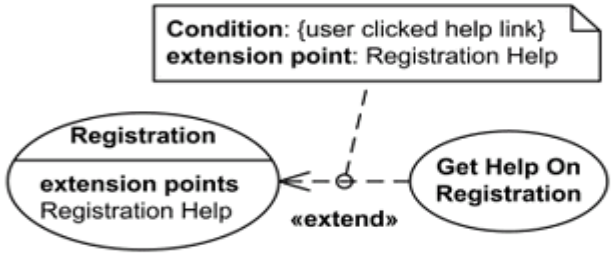
*Web User Authentication use case is **abstract use case** specialized by Login, Remember Me and Single Sign-On use cases.*



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

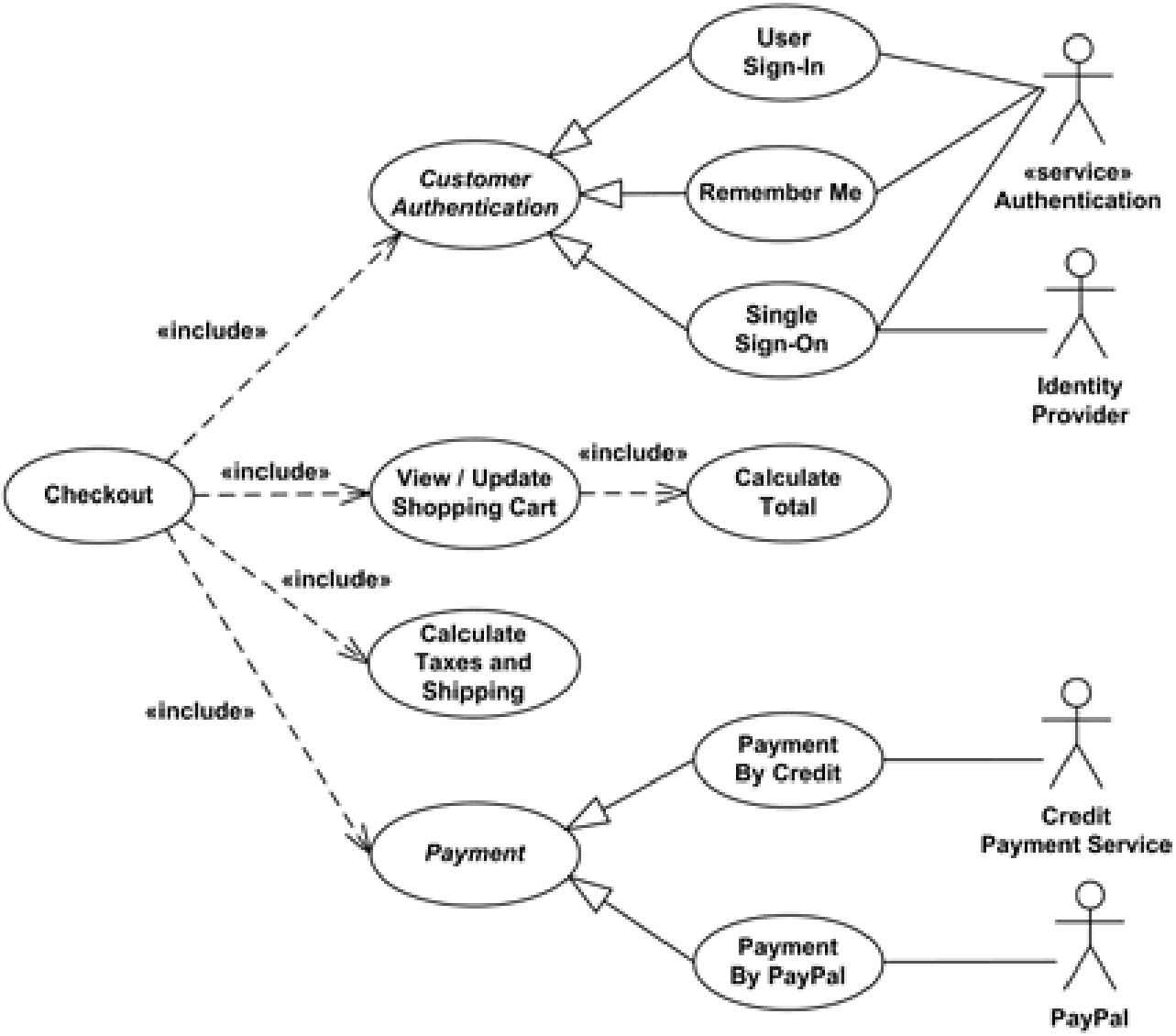


Web Client actor is abstract superclass for Administrator, Editor and Customer



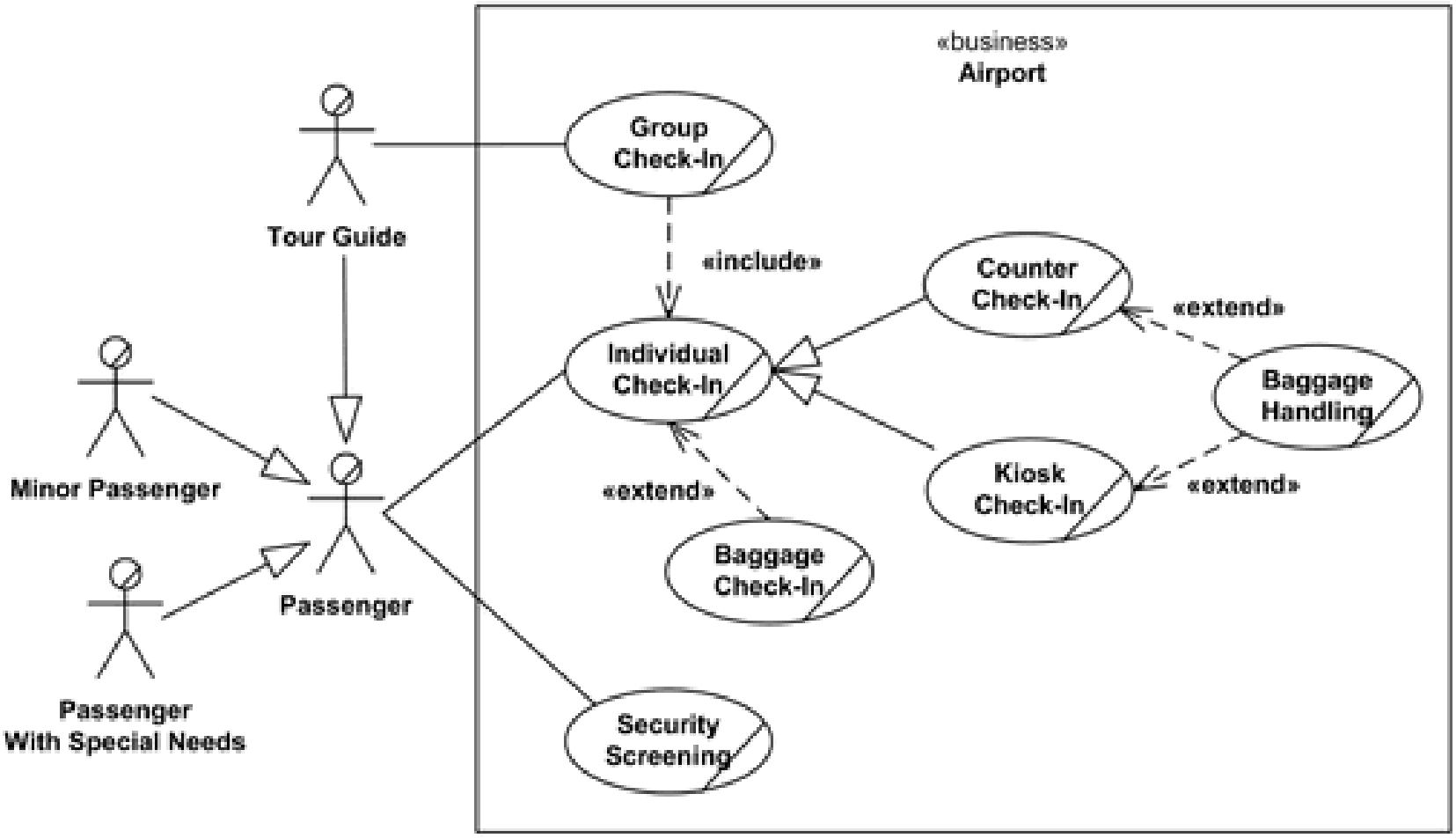
Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

Online Shopping Checkout



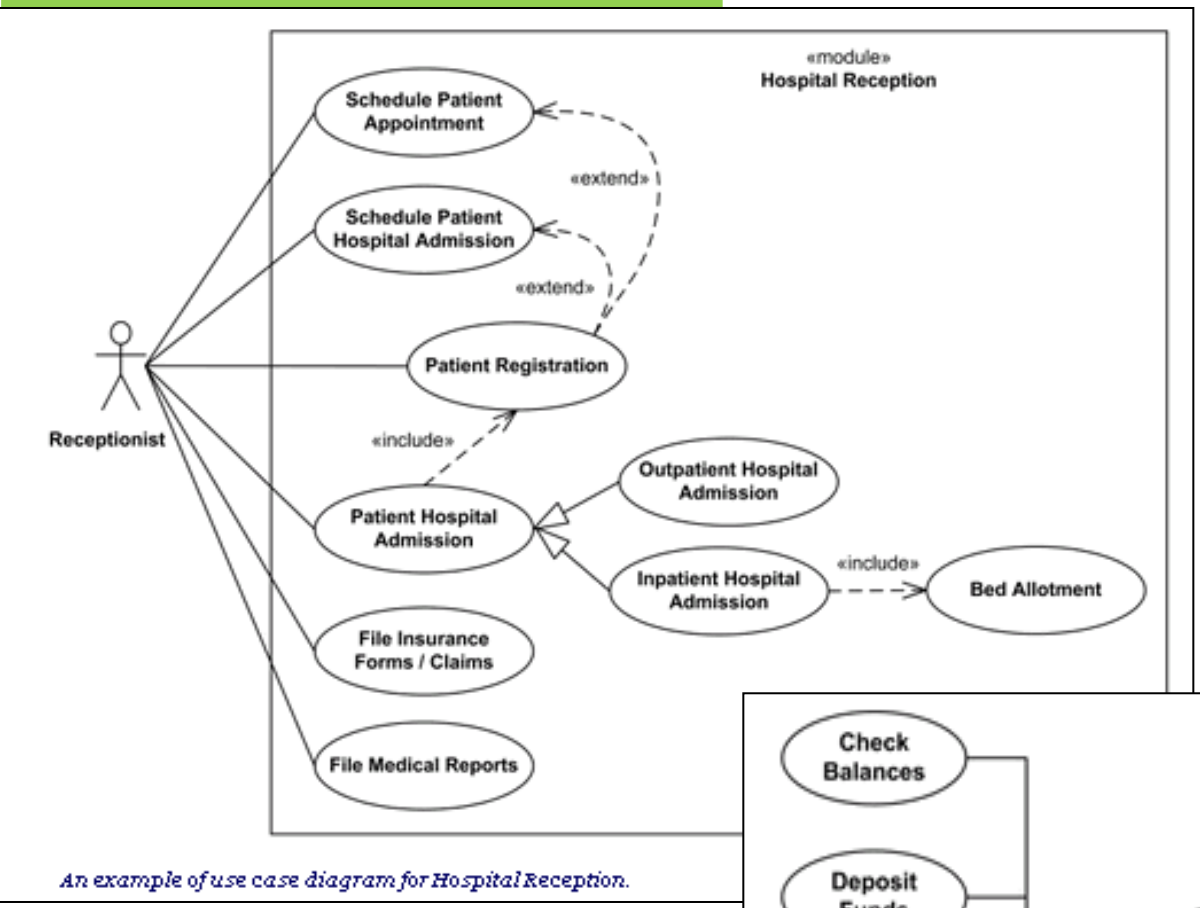
Online Shopping - Checkout, Authentication and Payment Use Cases

Airport Check-in

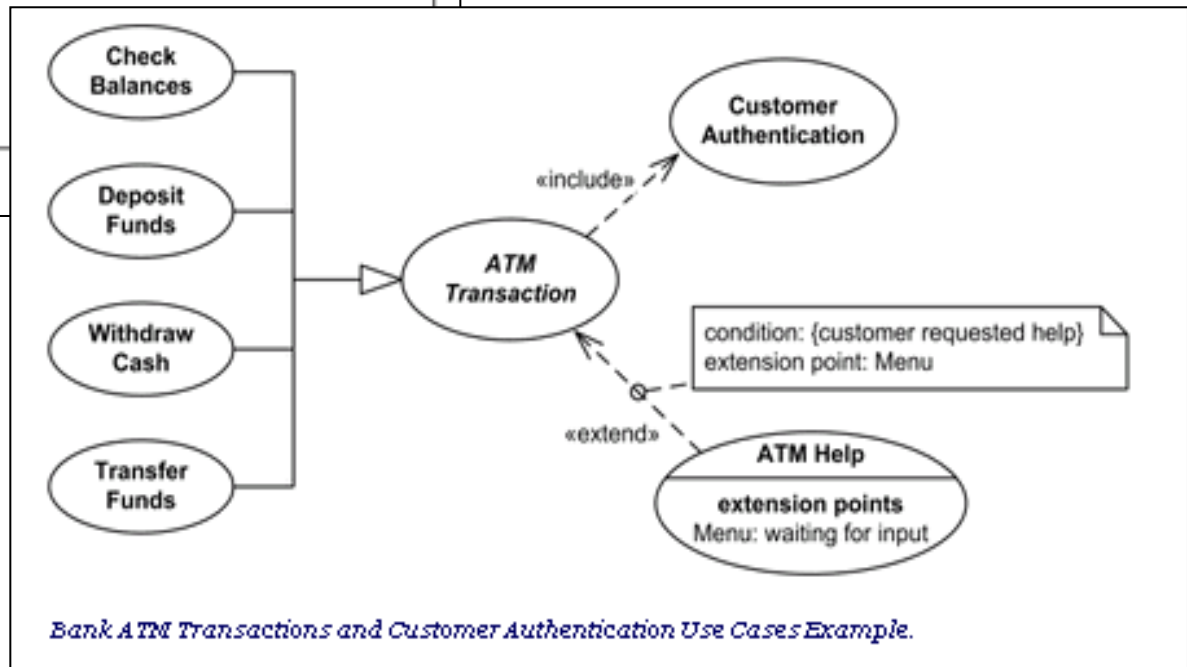


An example of use case diagram for airport check-in and security screening

More Examples

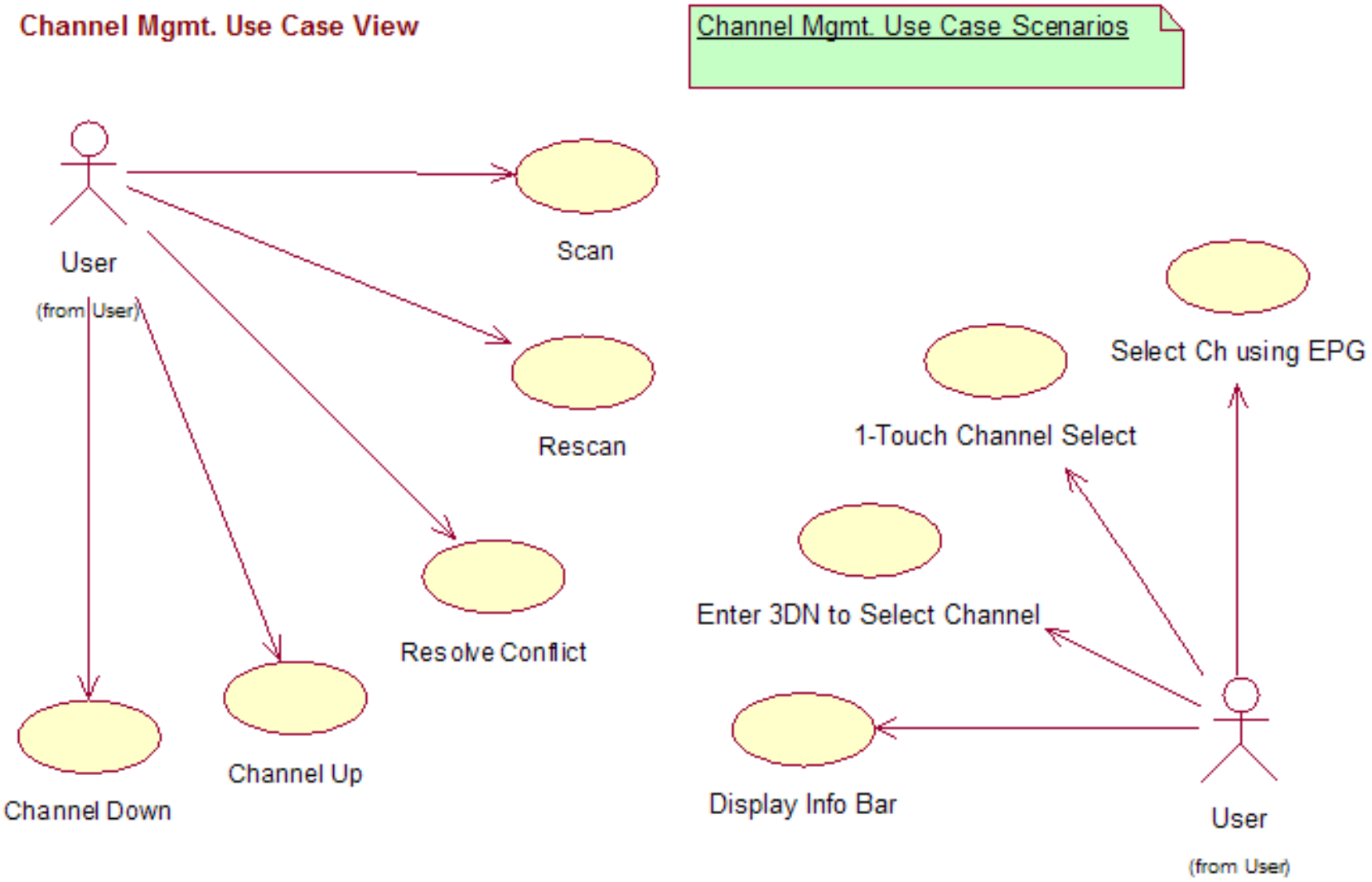


An example of use case diagram for Hospital Reception.

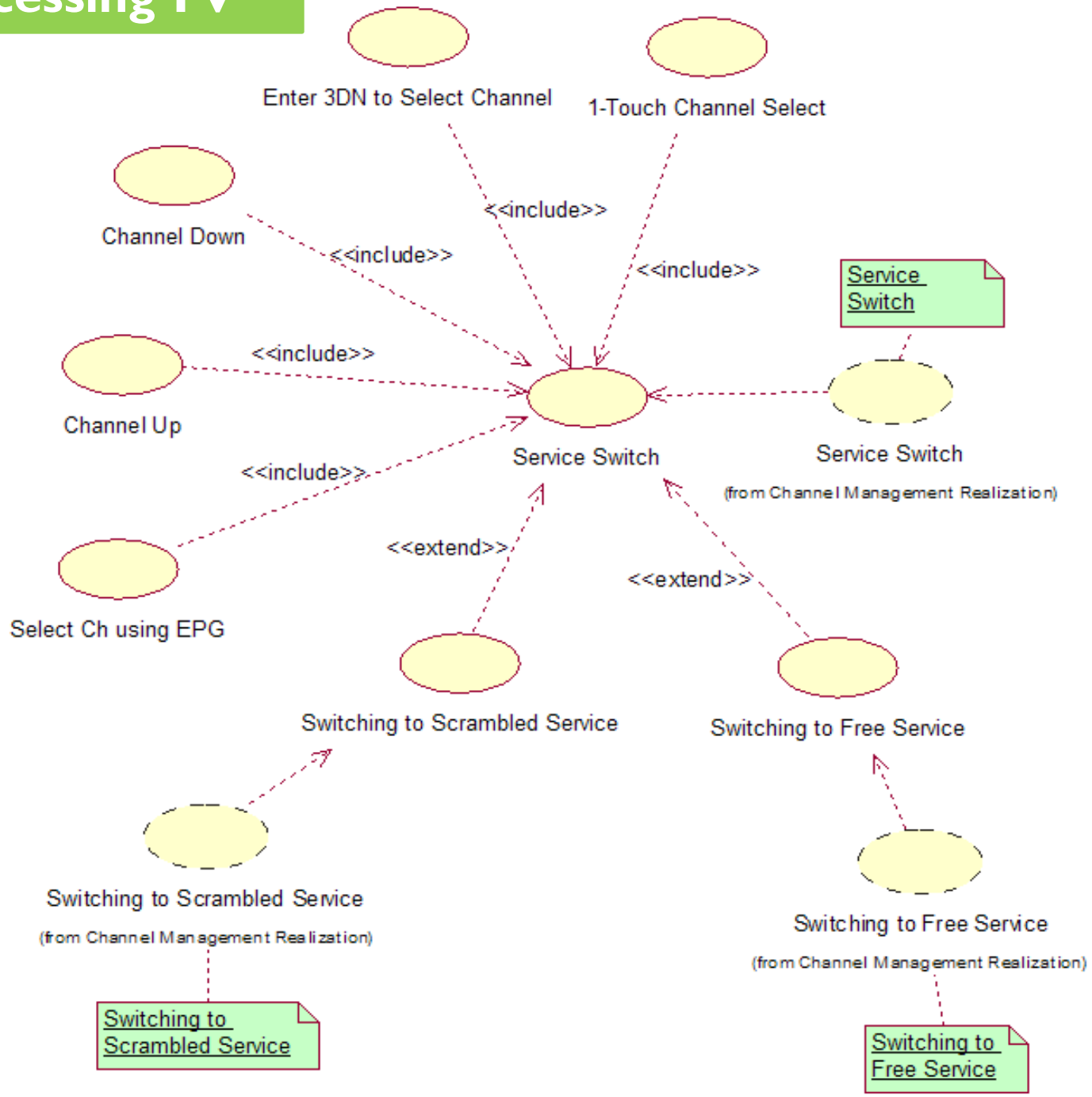


Bank ATM Transactions and Customer Authentication Use Cases Example.

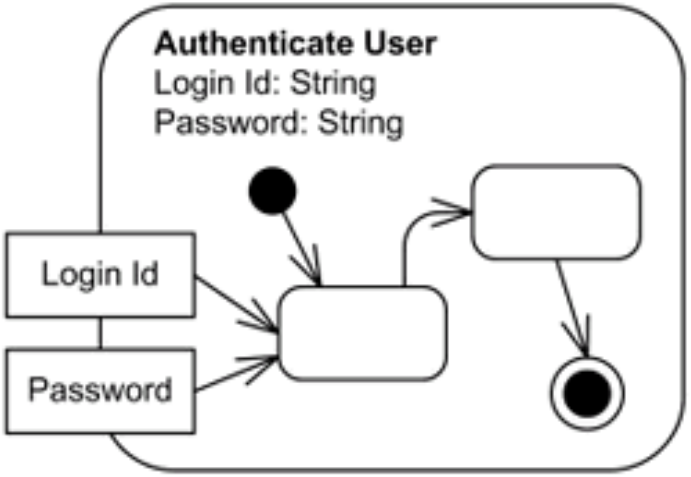
User Accessing TV

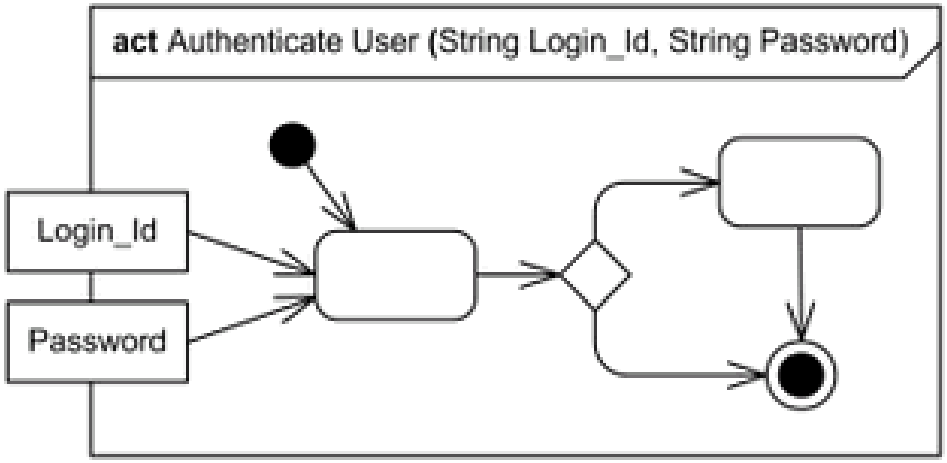


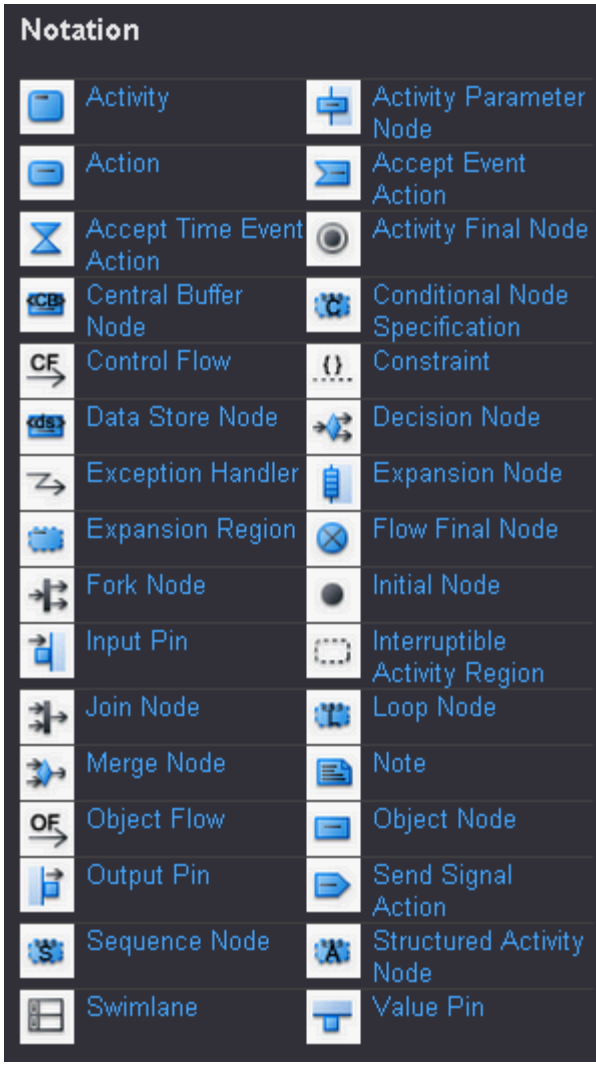
User Accessing TV



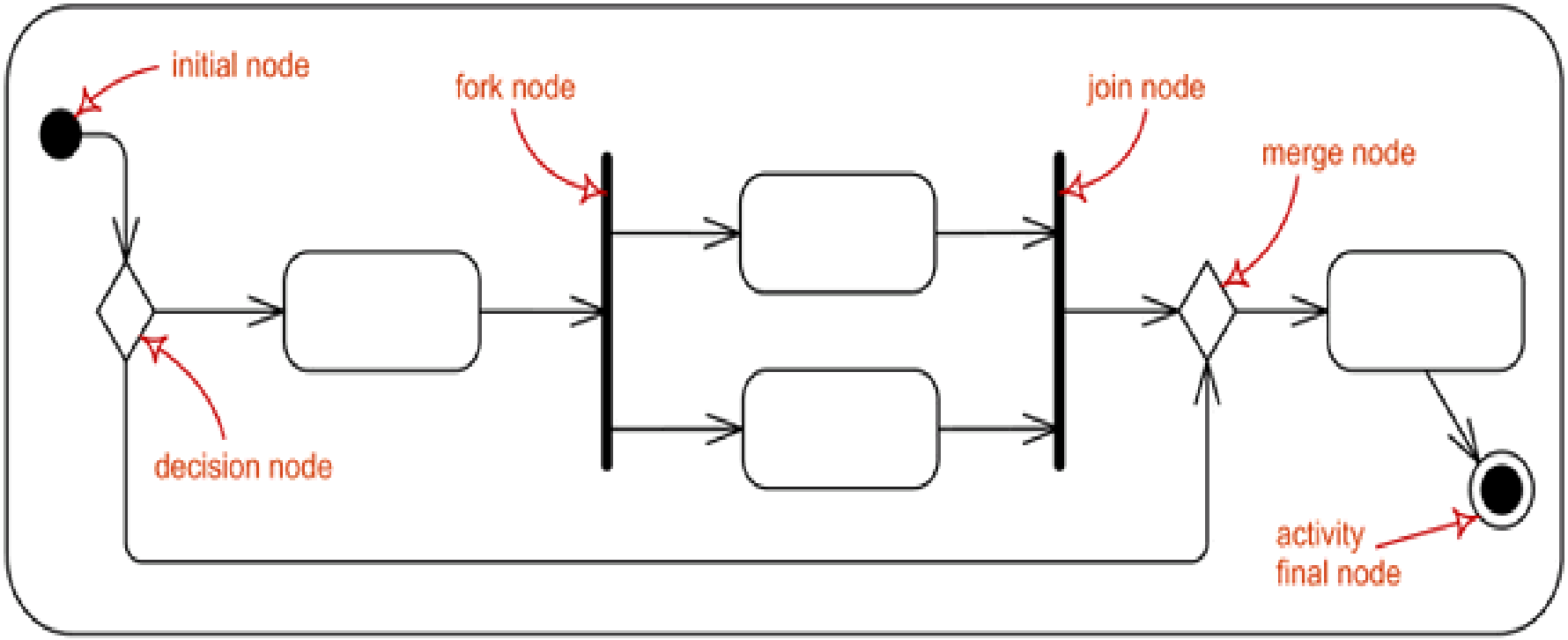
Activity Diagram



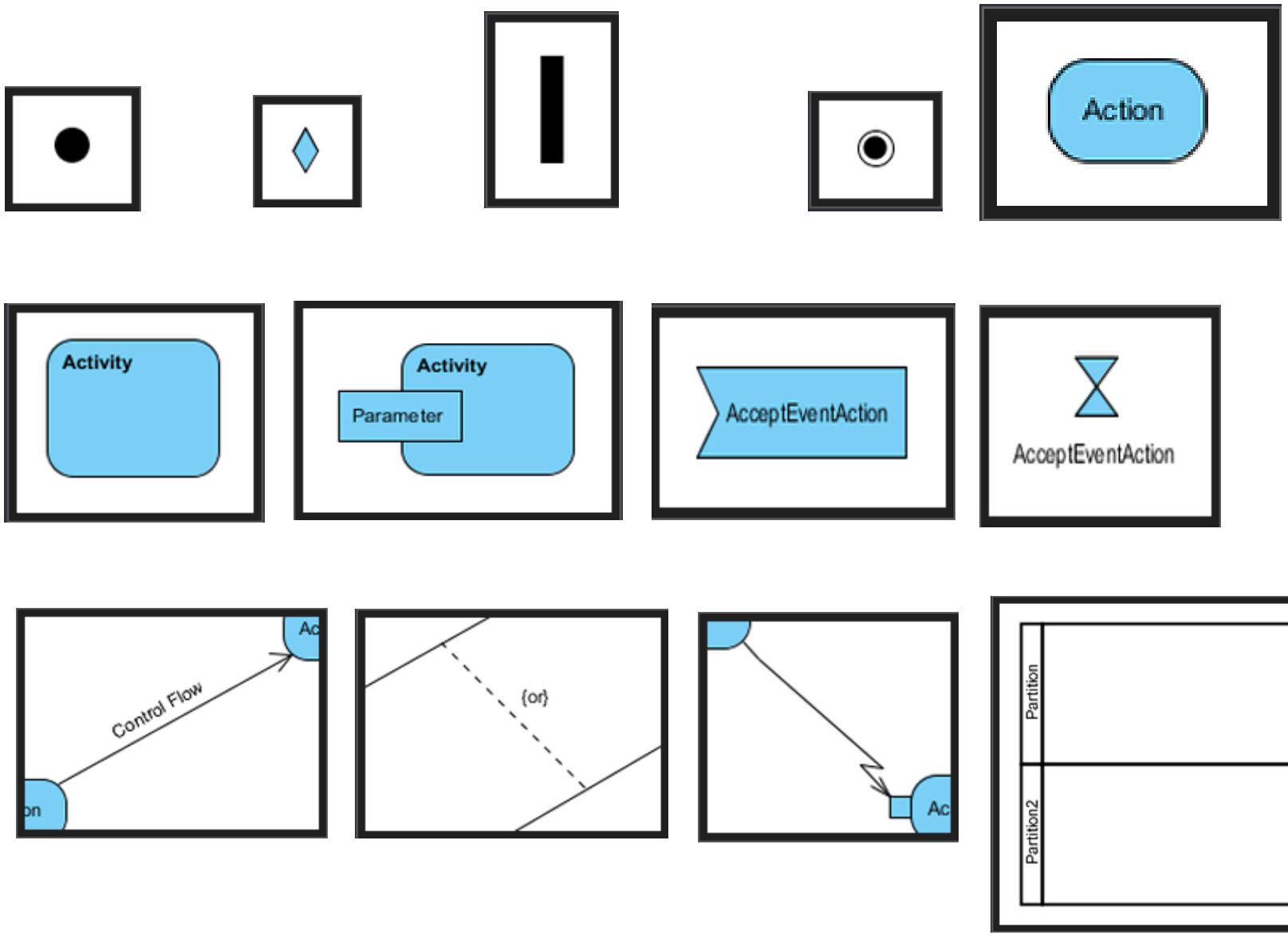




Activity Diagram



Activity Diagram (Notations)

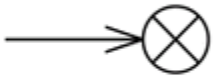


Activity Diagram (Notations)

Initial Node

Activity initial node.

Flow Final Node

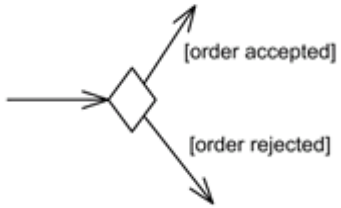


Flow final node.

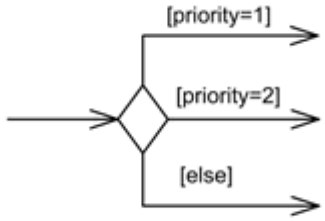
Activity Final Node

Activity final node.

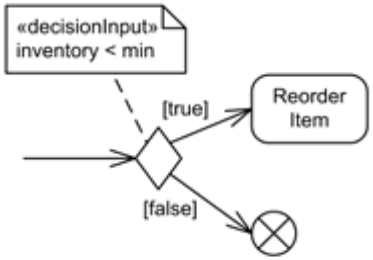
Decision



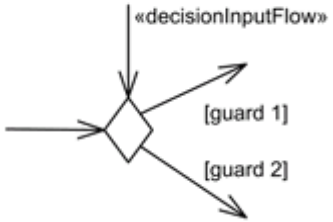
***Decision node** with two **outgoing edges** with guards.*



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

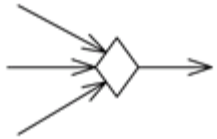


***Decision node** with **decision input behavior**.*



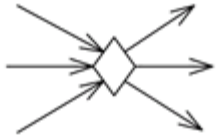
***Decision node** with **decision input flow**.*

Merge



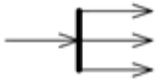
***Merge node** with three incoming edges and a single outgoing edge.*

Merge and decision combined



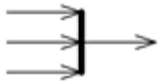
***Merge node** and **decision node** combined.*

Fork

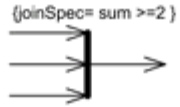


***Fork node** with a single activity edge entering it, and three edges leaving it.*

Join Node



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

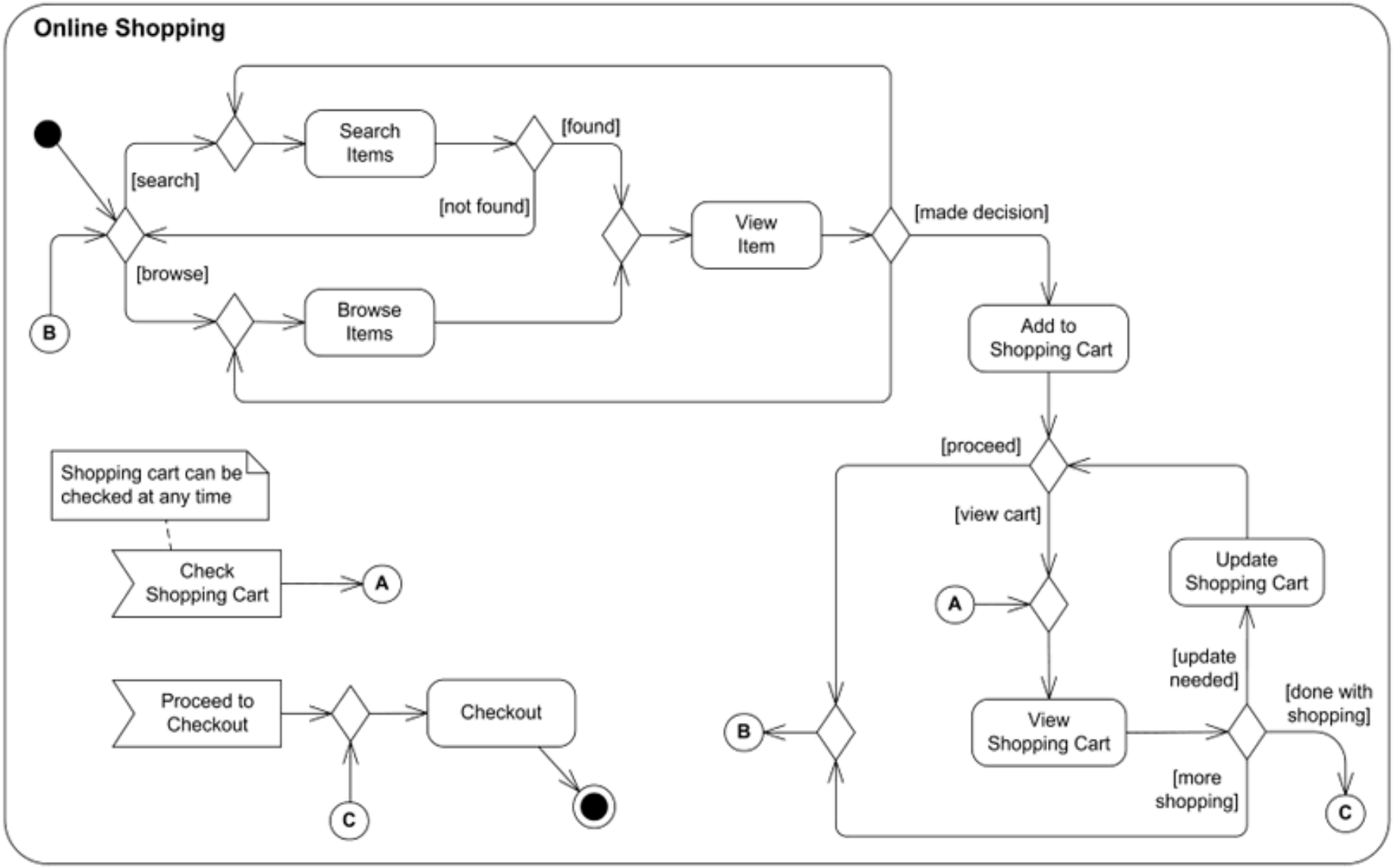


***Join node** with **join specification** shown in curly braces.*

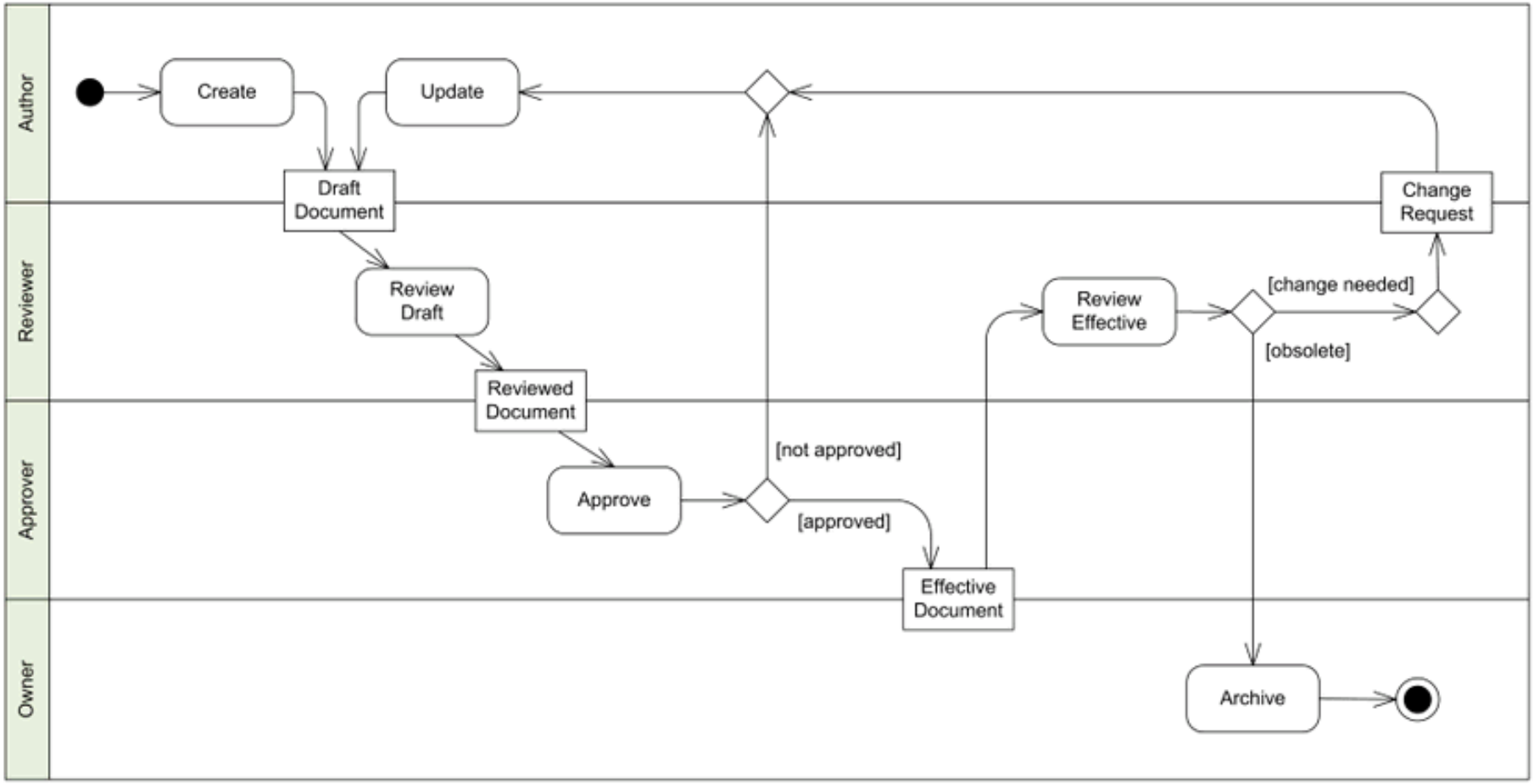
Join and fork combined

*Combined **join node** and **fork node**.*

Activity Diagram - Online Shopping

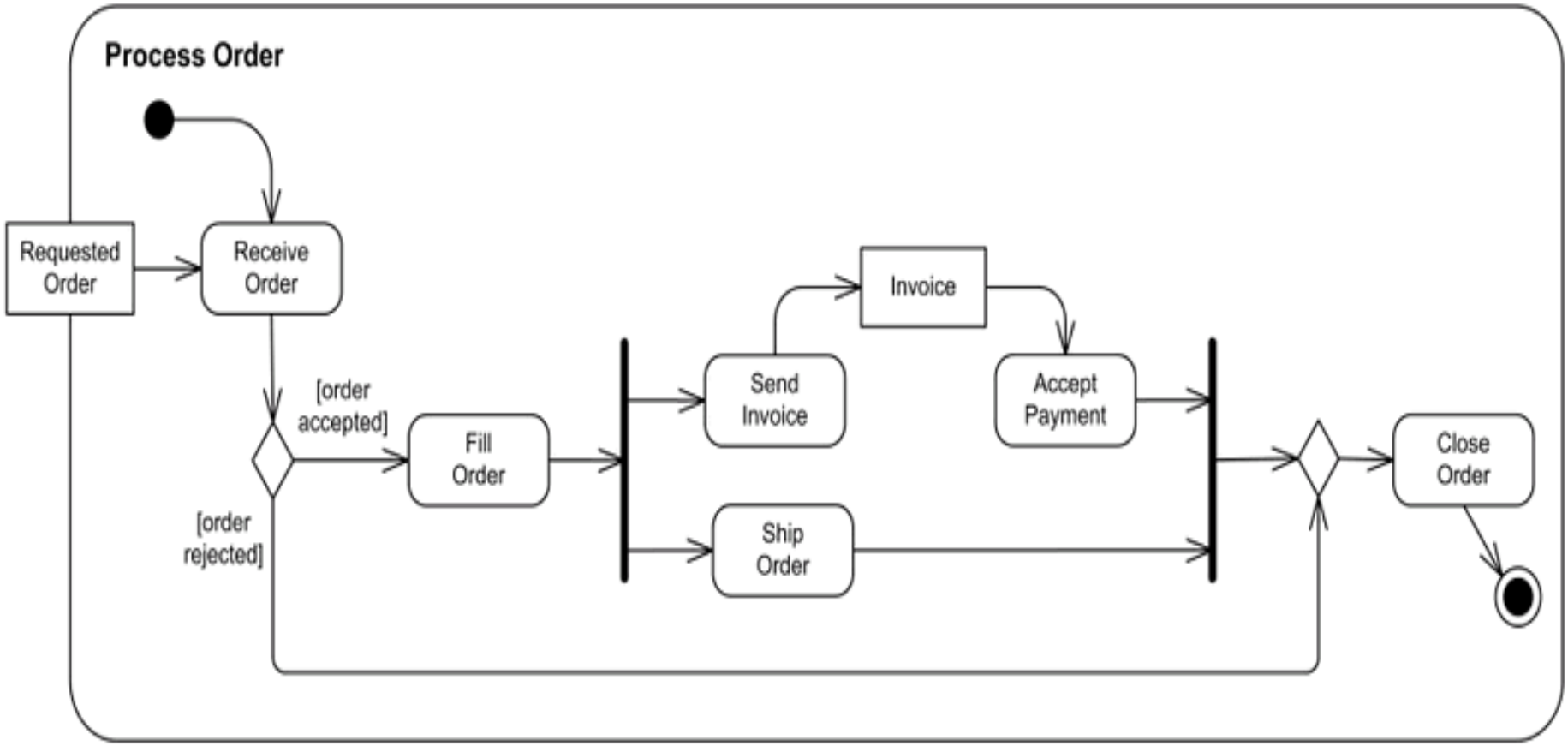


Activity Diagram with Swim Lanes

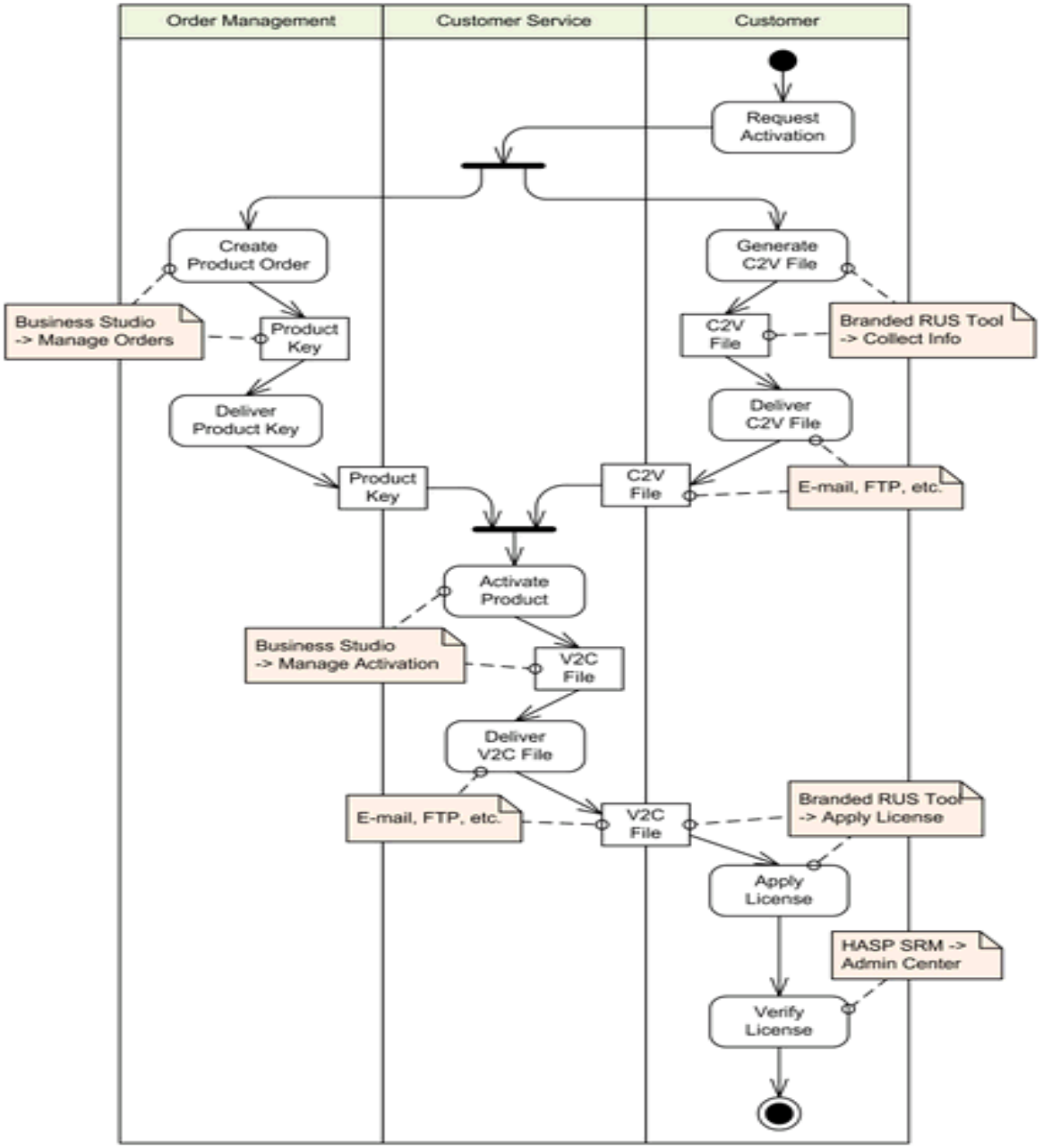


An example of Document Management Process activity.

Activity Diagram with Fork

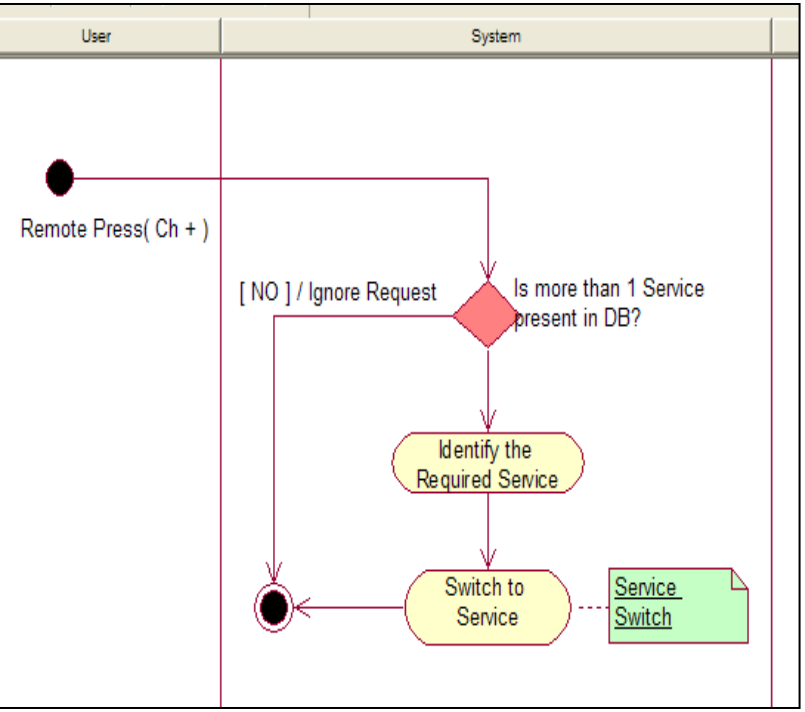


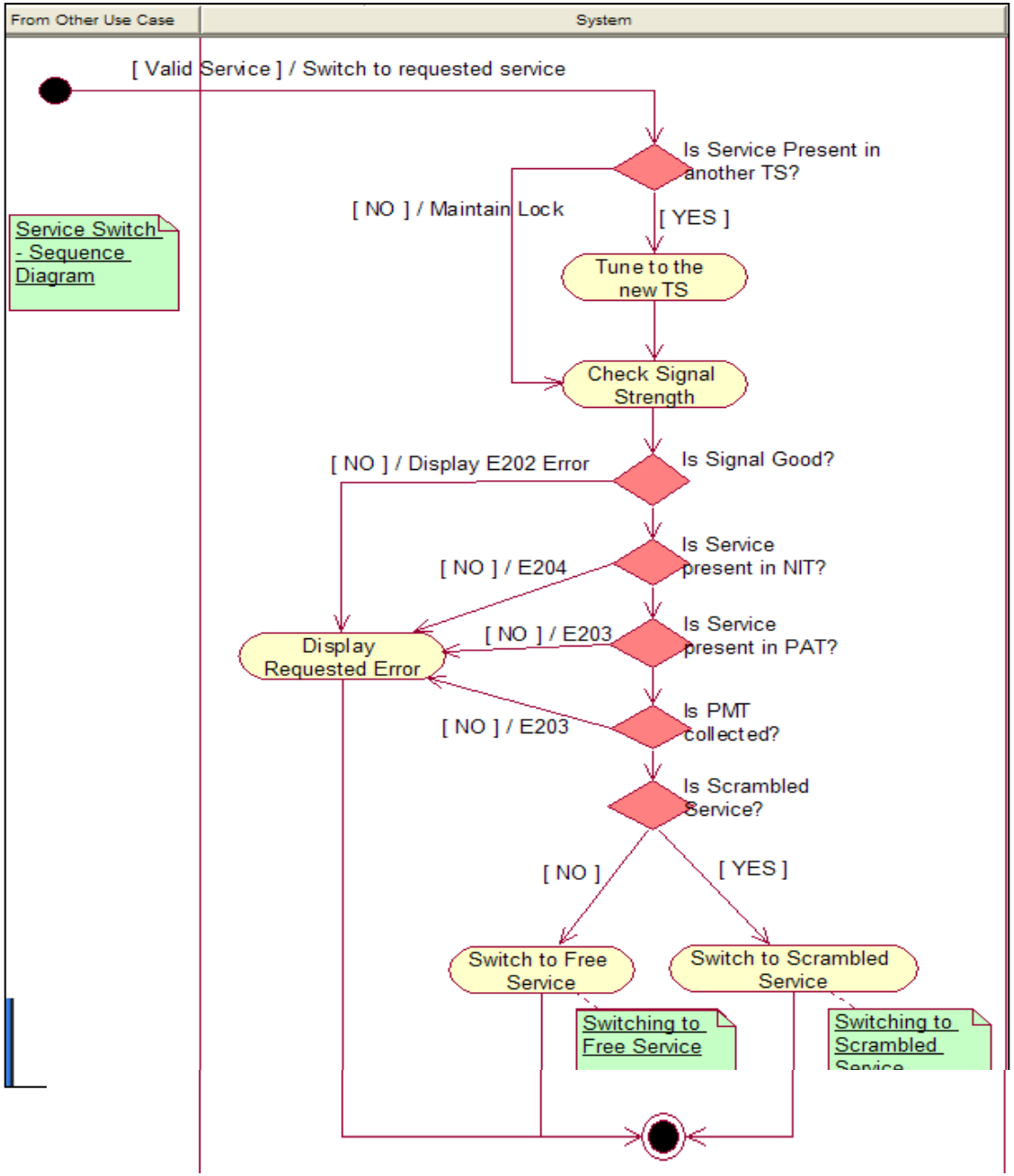
Activity Diagram with Fork



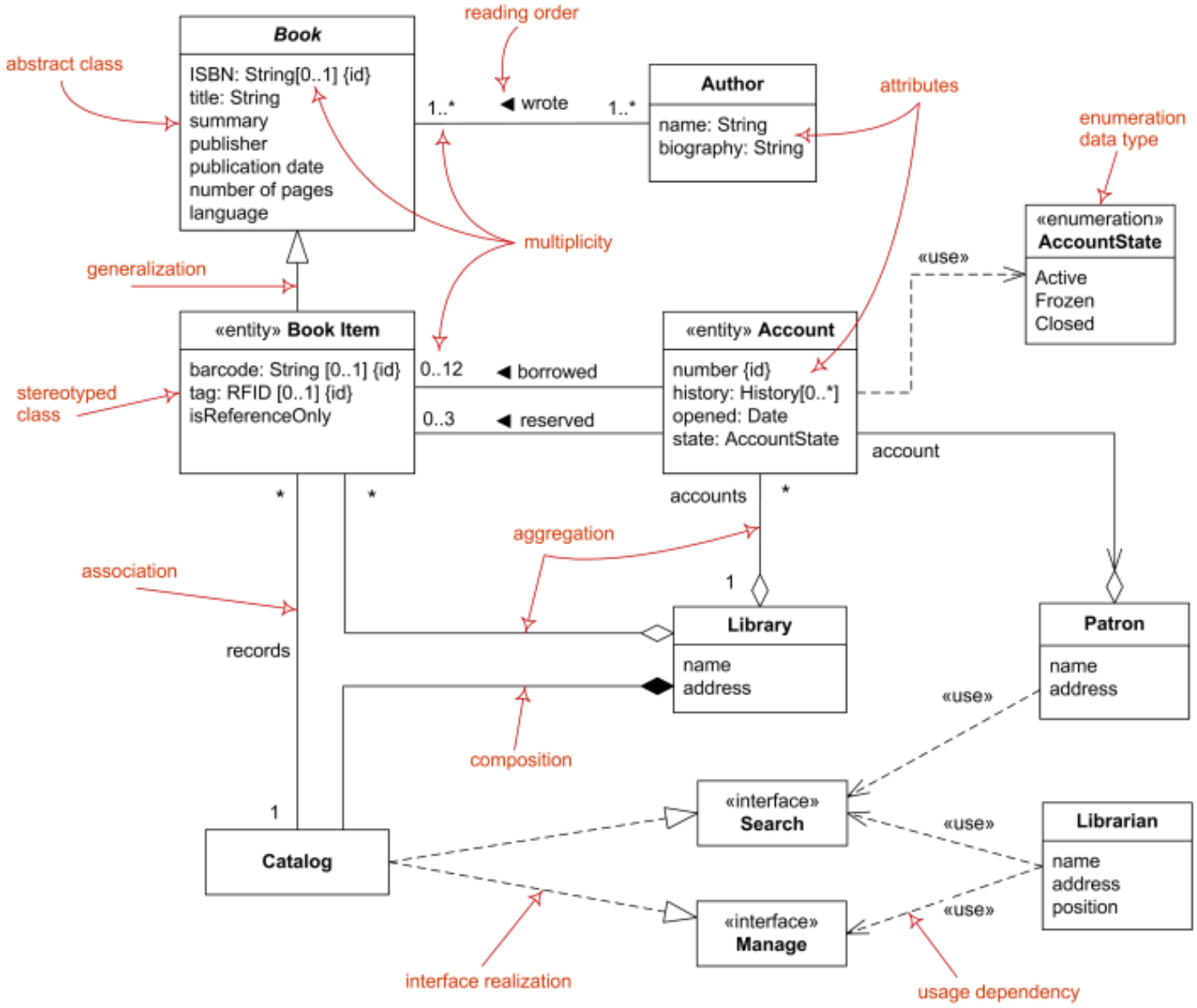
An example of activity for manual activation of trial product protected by HASP SL.

Activity Diagram



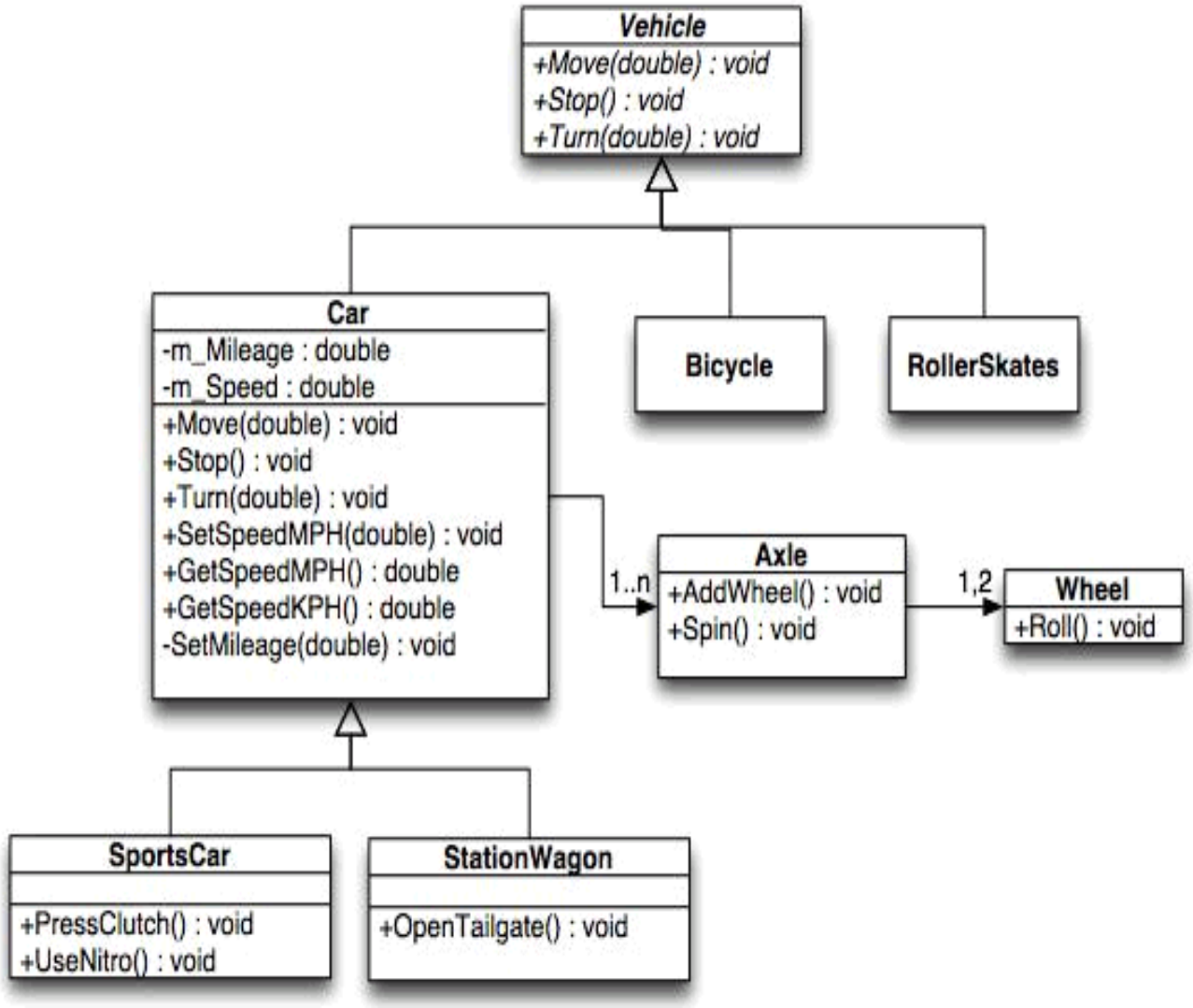


Domain Model Diagram

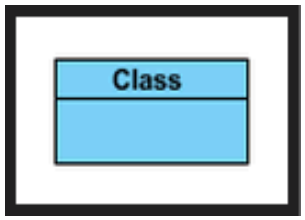


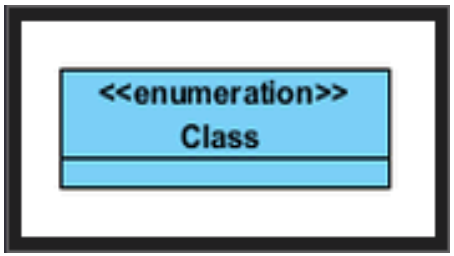
Domain diagram overview - classes, interfaces, associations, usage, realization, multiplicity.

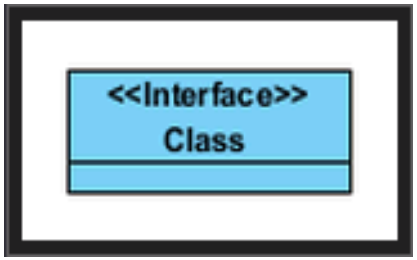
Class Diagram

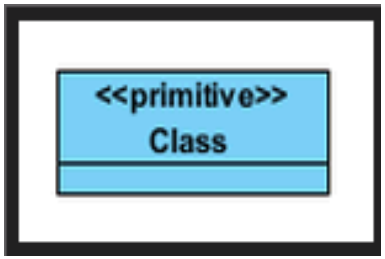


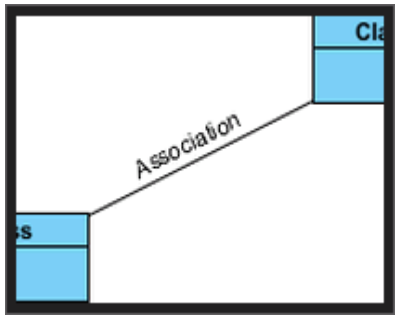
Class Diagram (Notations)

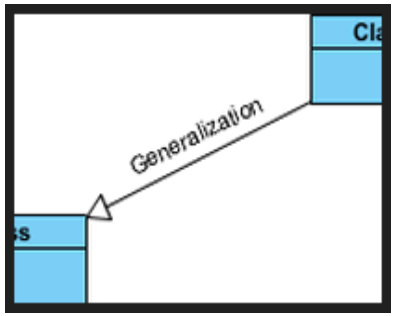


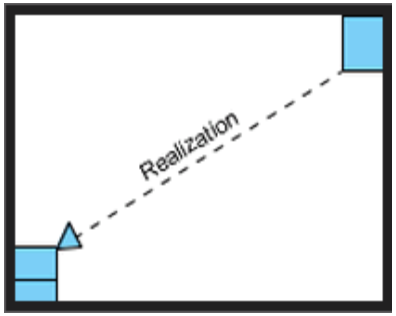


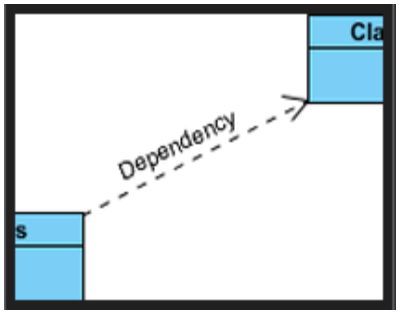


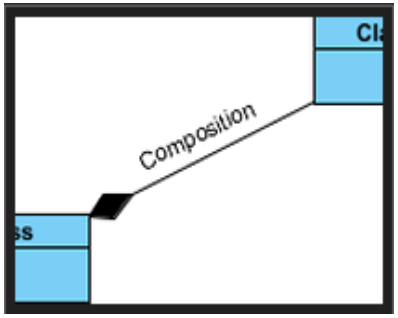


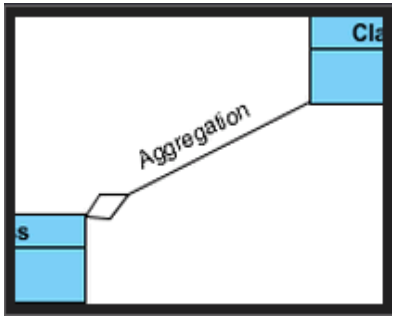


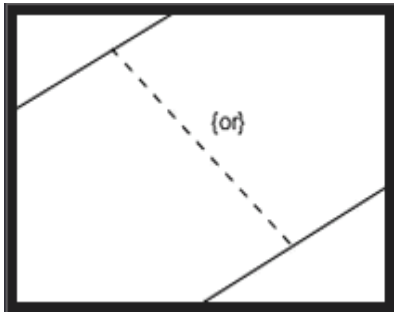


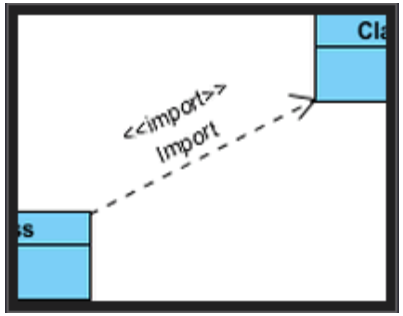




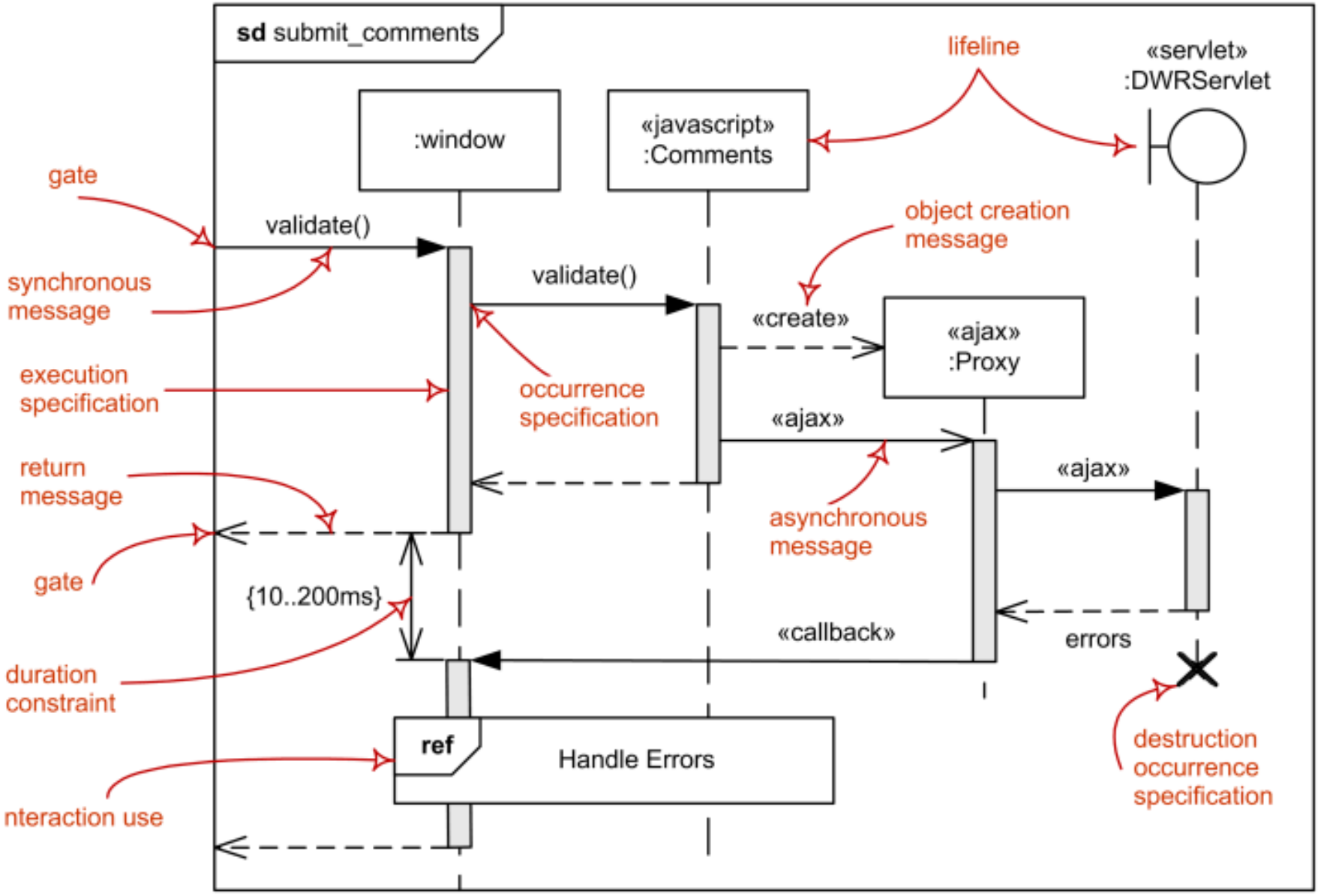








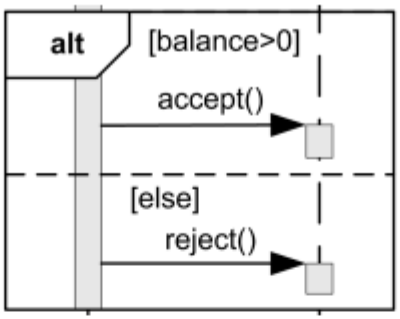
Sequence Diagram



Sequence diagram major elements.

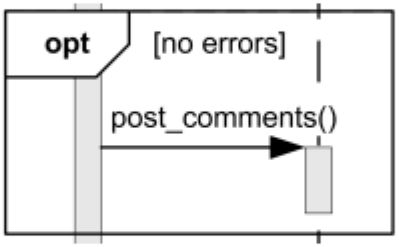
Sequence Diagram (Notations)

Alternatives



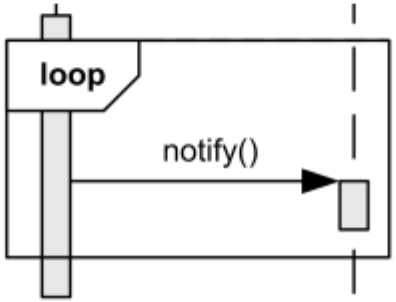
Call accept() if balance > 0, call reject() otherwise.

Option



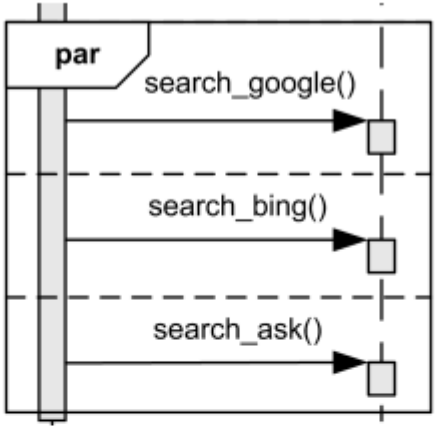
Post comments if there were no errors.

Loop

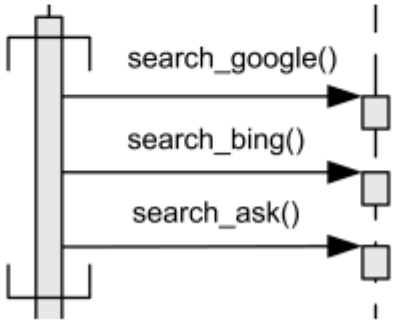


Potentially infinite loop.

Parallel

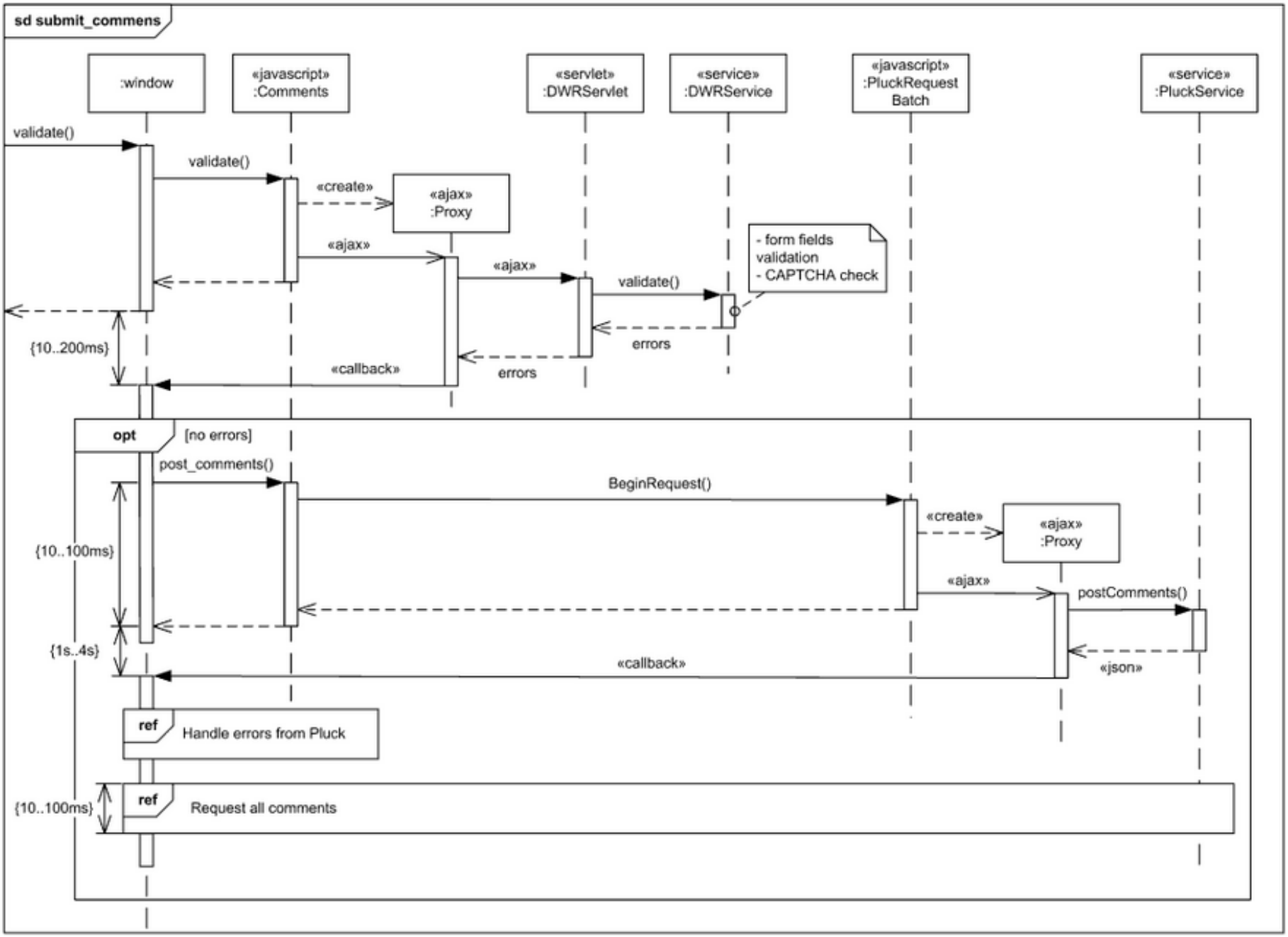


Search Google, Bing and Ask in any order, possibly parallel.

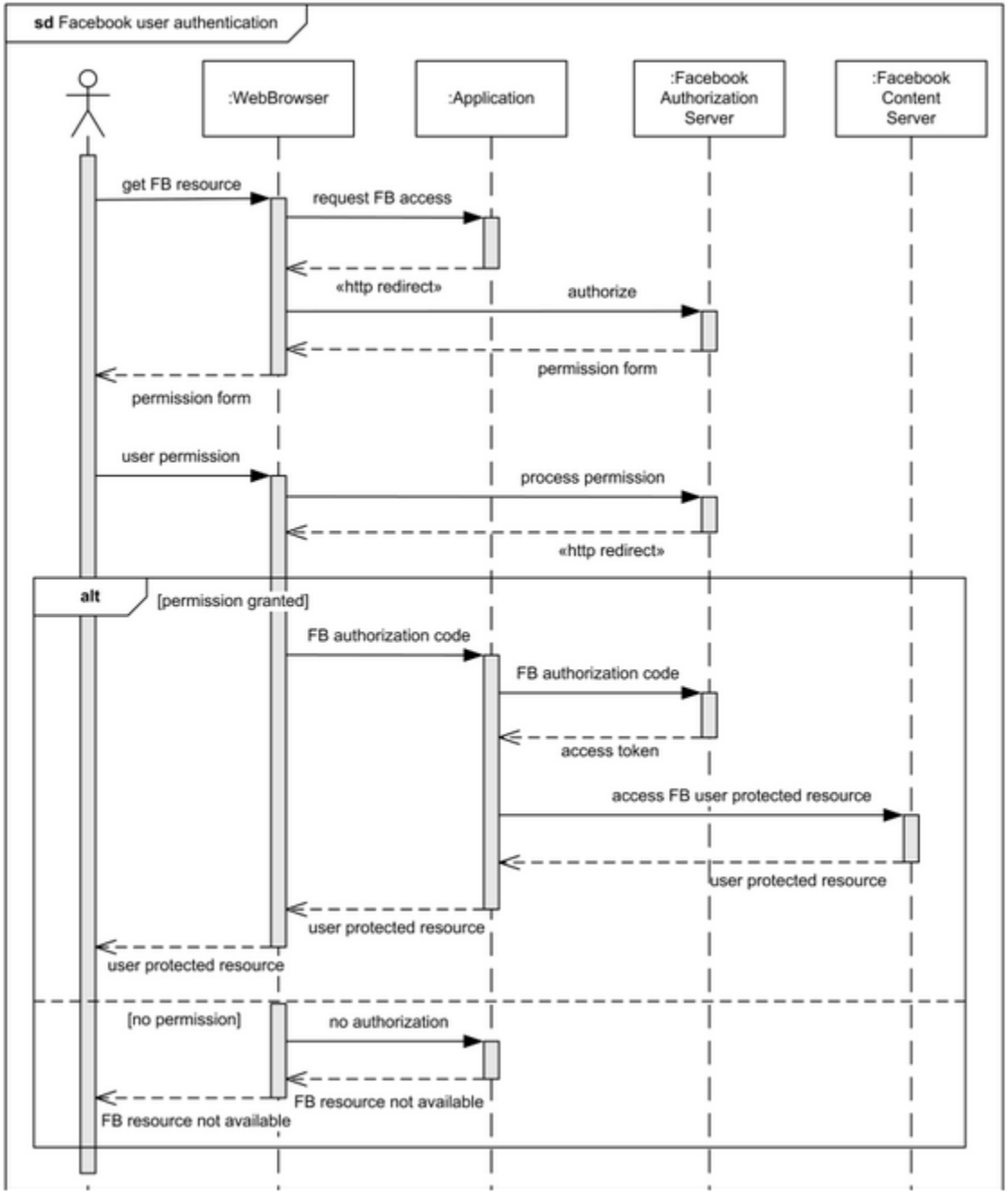


Coregion - search Google, Bing and Ask in any order, possibly parallel.

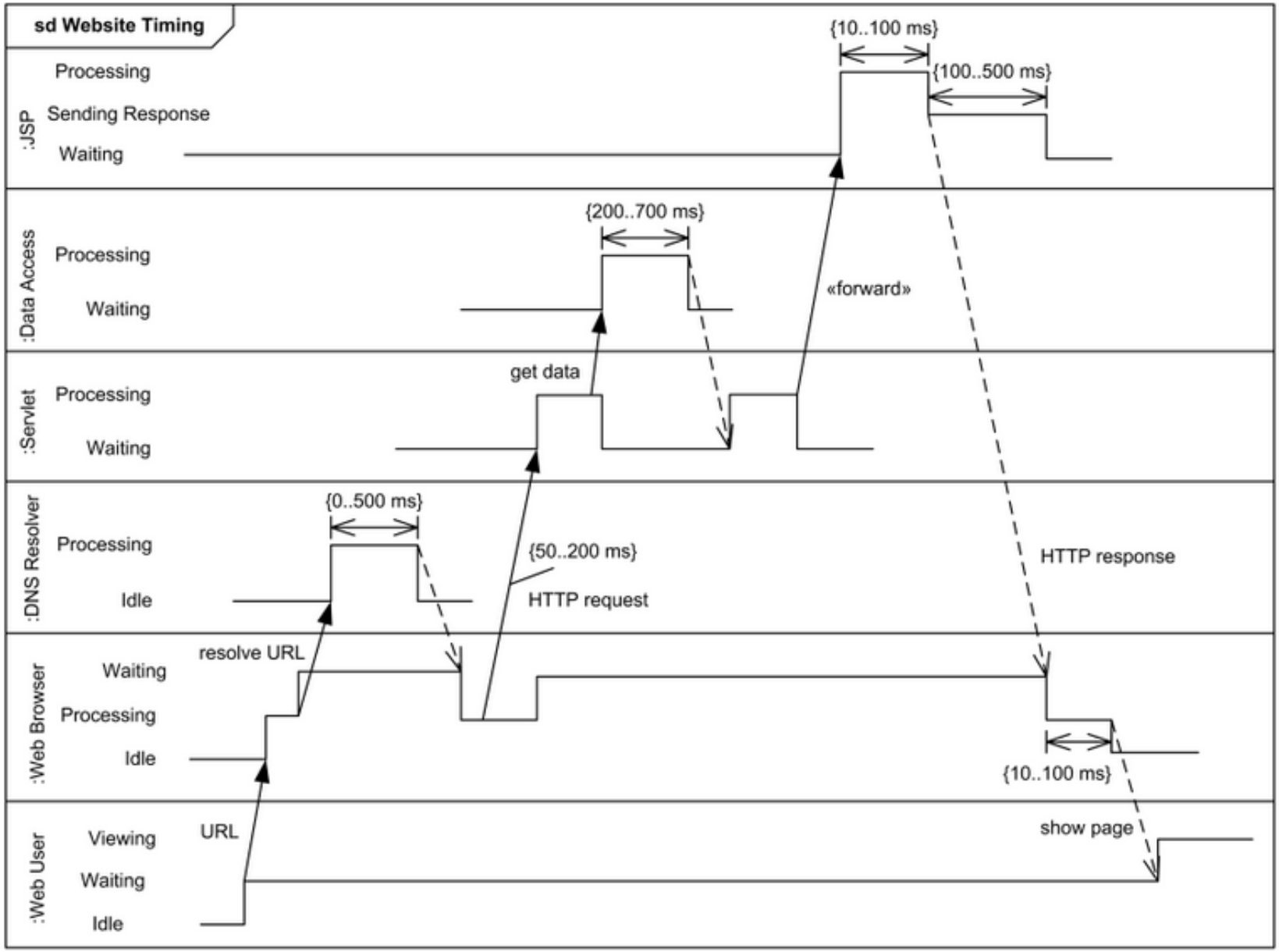
Sequence Diagram (Example)



Sequence Diagram (Cont…)



Timing Diagrams

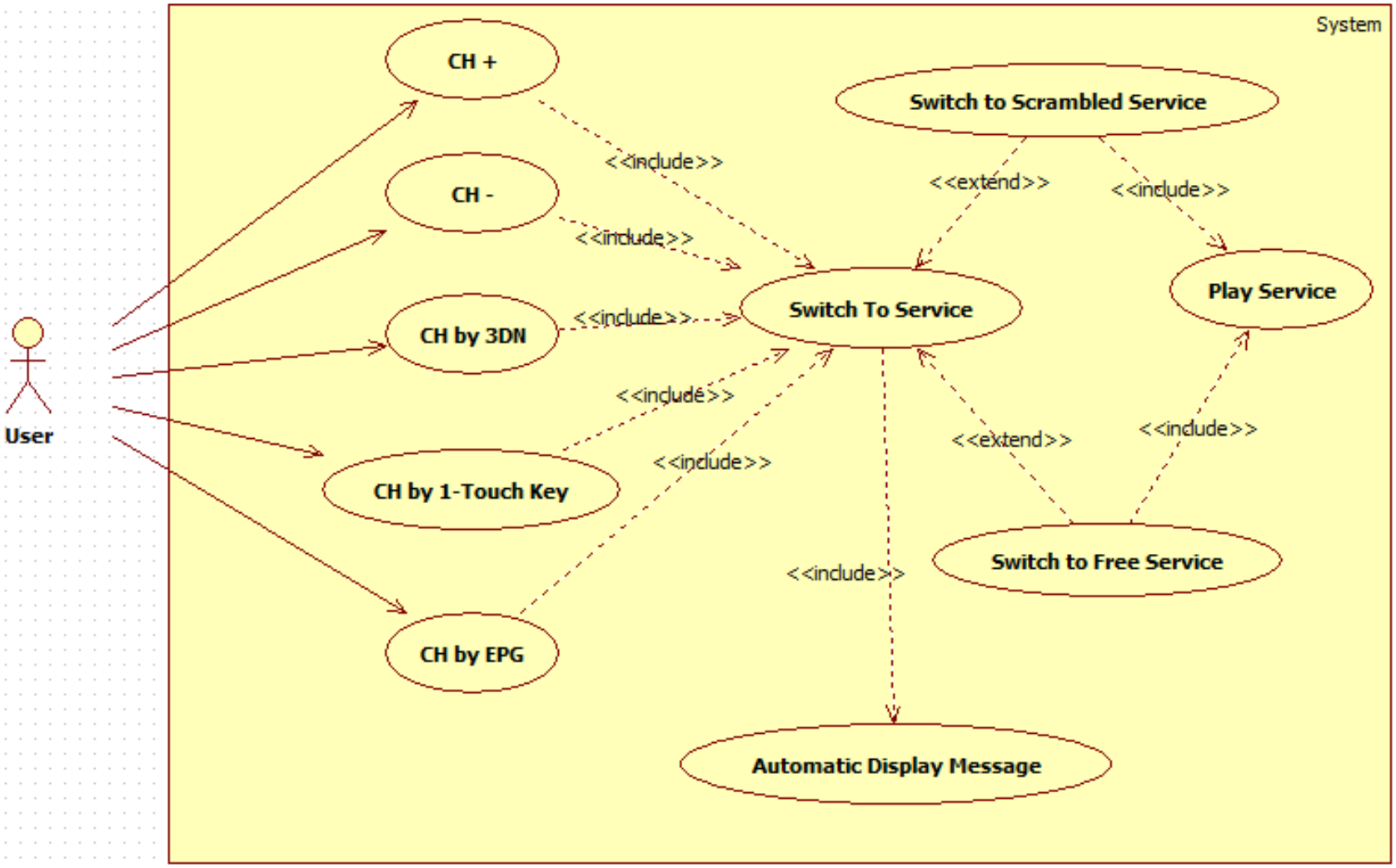


Timing Diagram Example - User Experience Website Latency

UML Approach

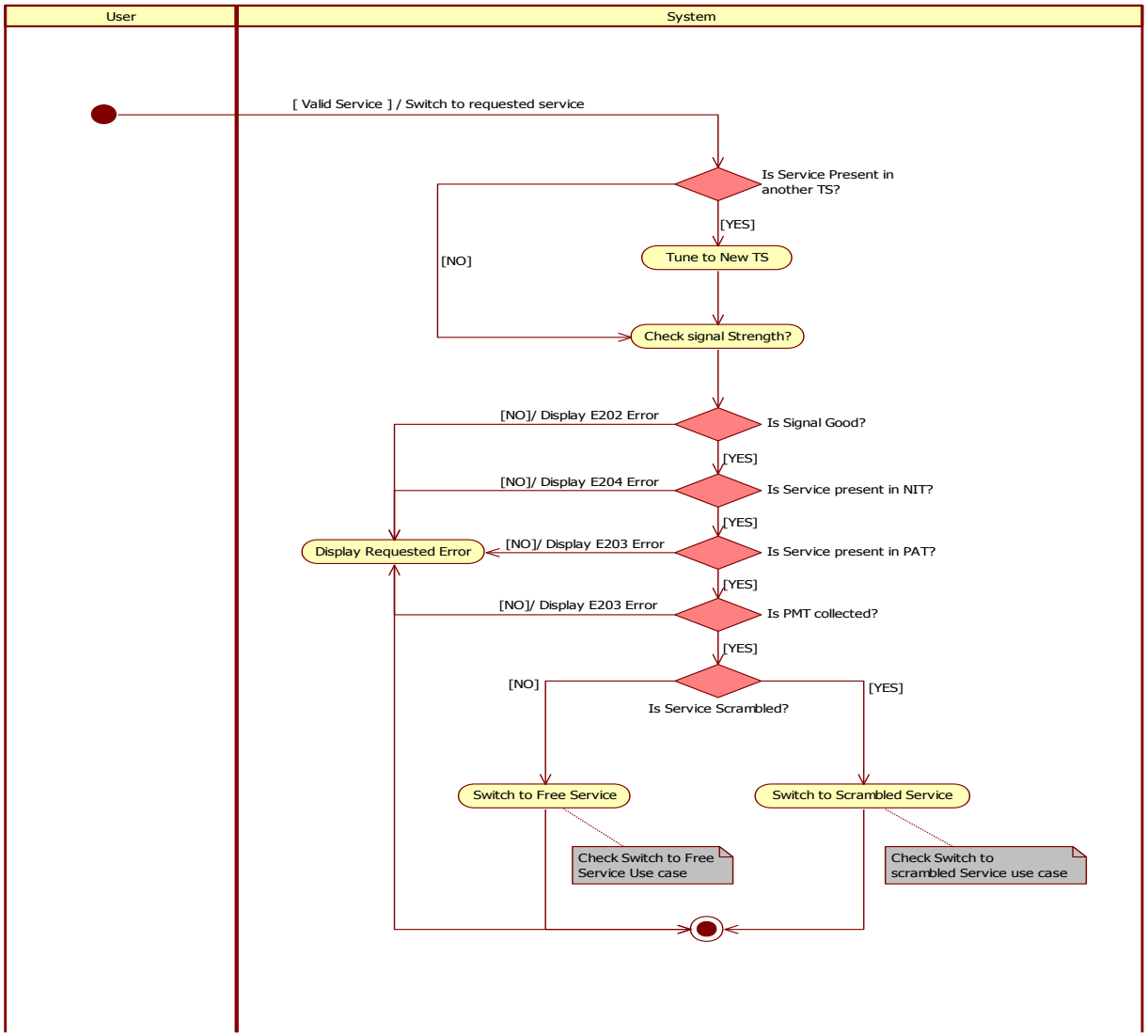
UML Approach

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



UML Approach (Cont..)

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

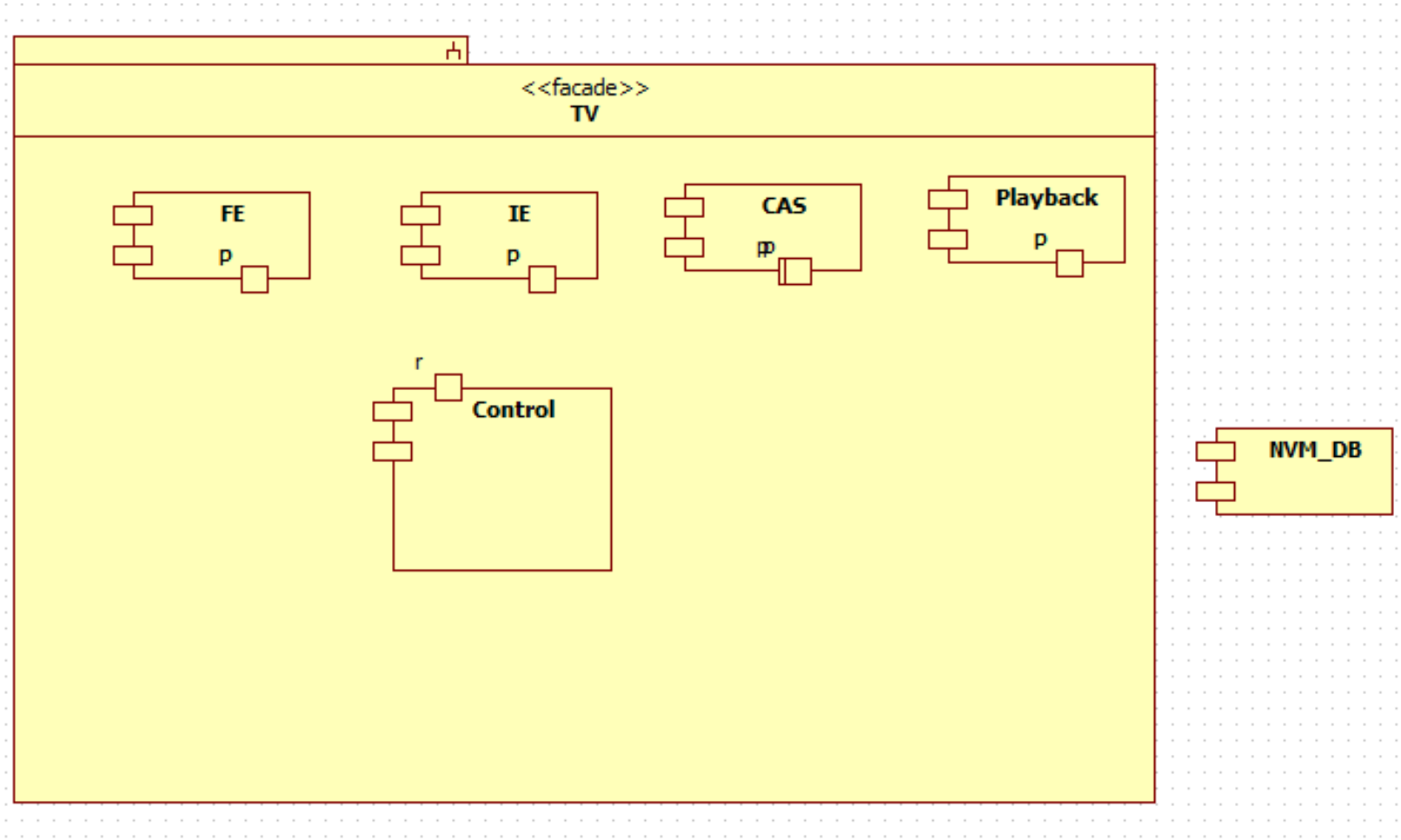


❖ NOTE

Think how to Achieve RE-USABLE Diagrams

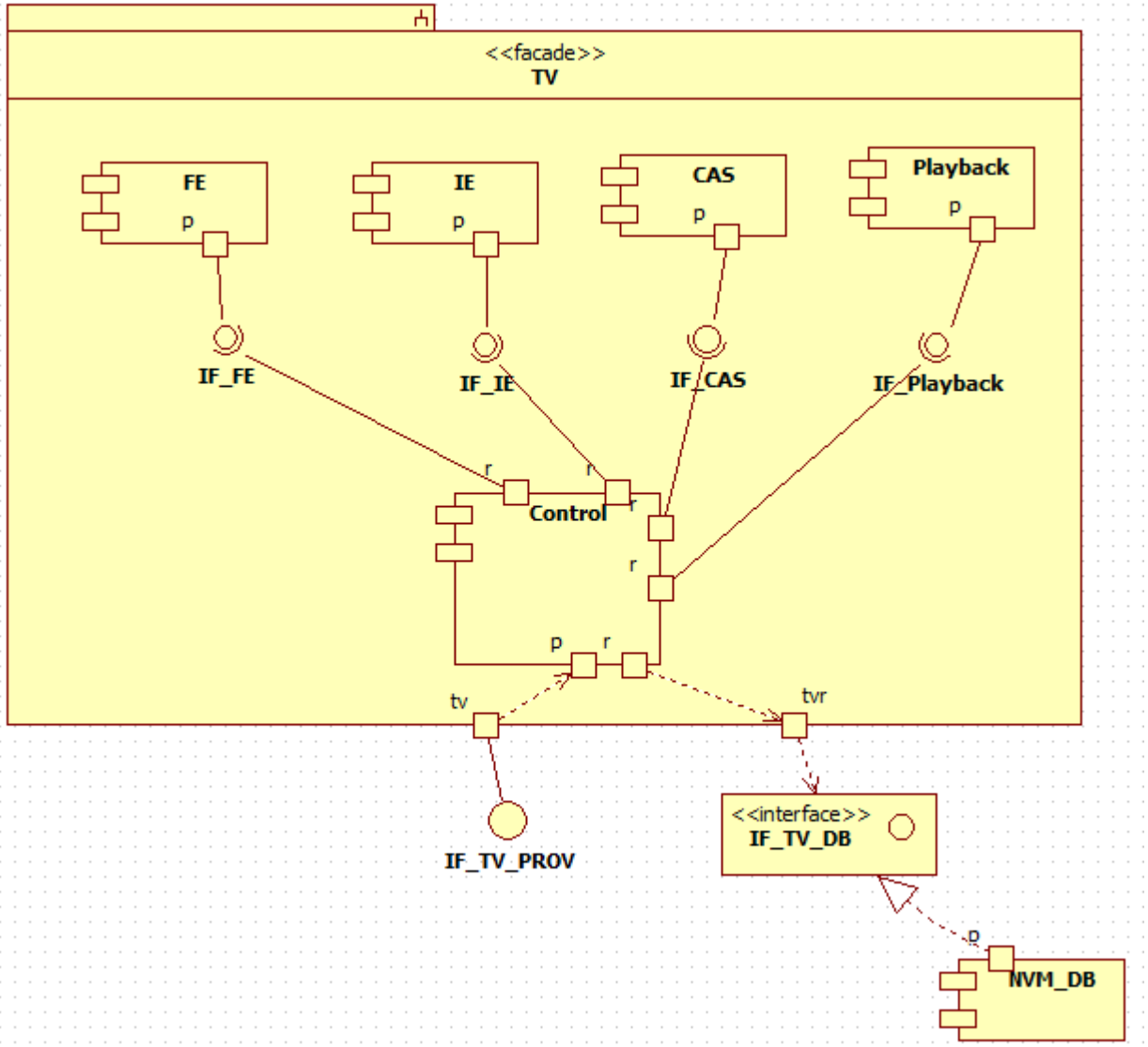
UML Approach (Cont..)

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



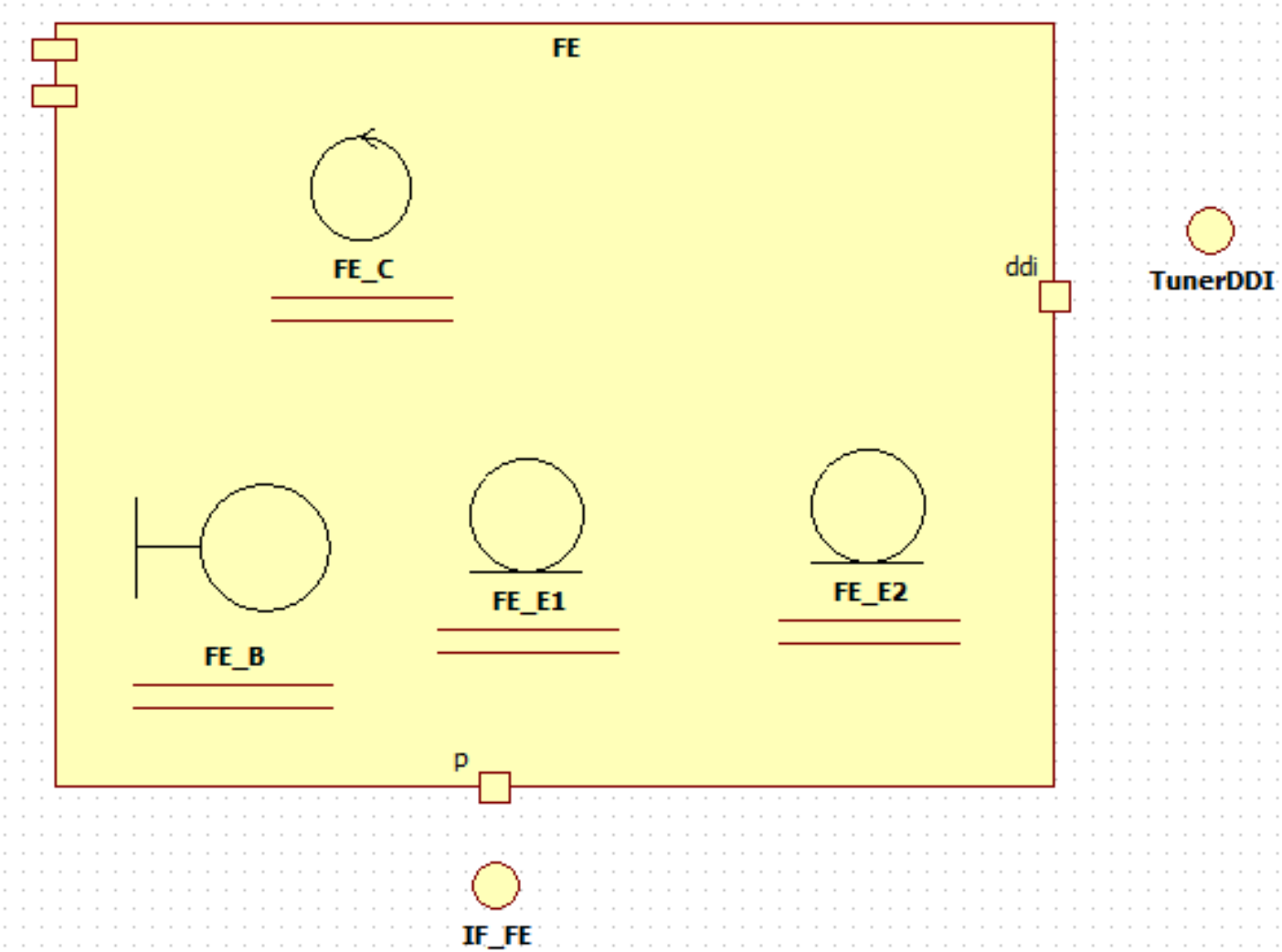
UML Approach (Cont..)

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



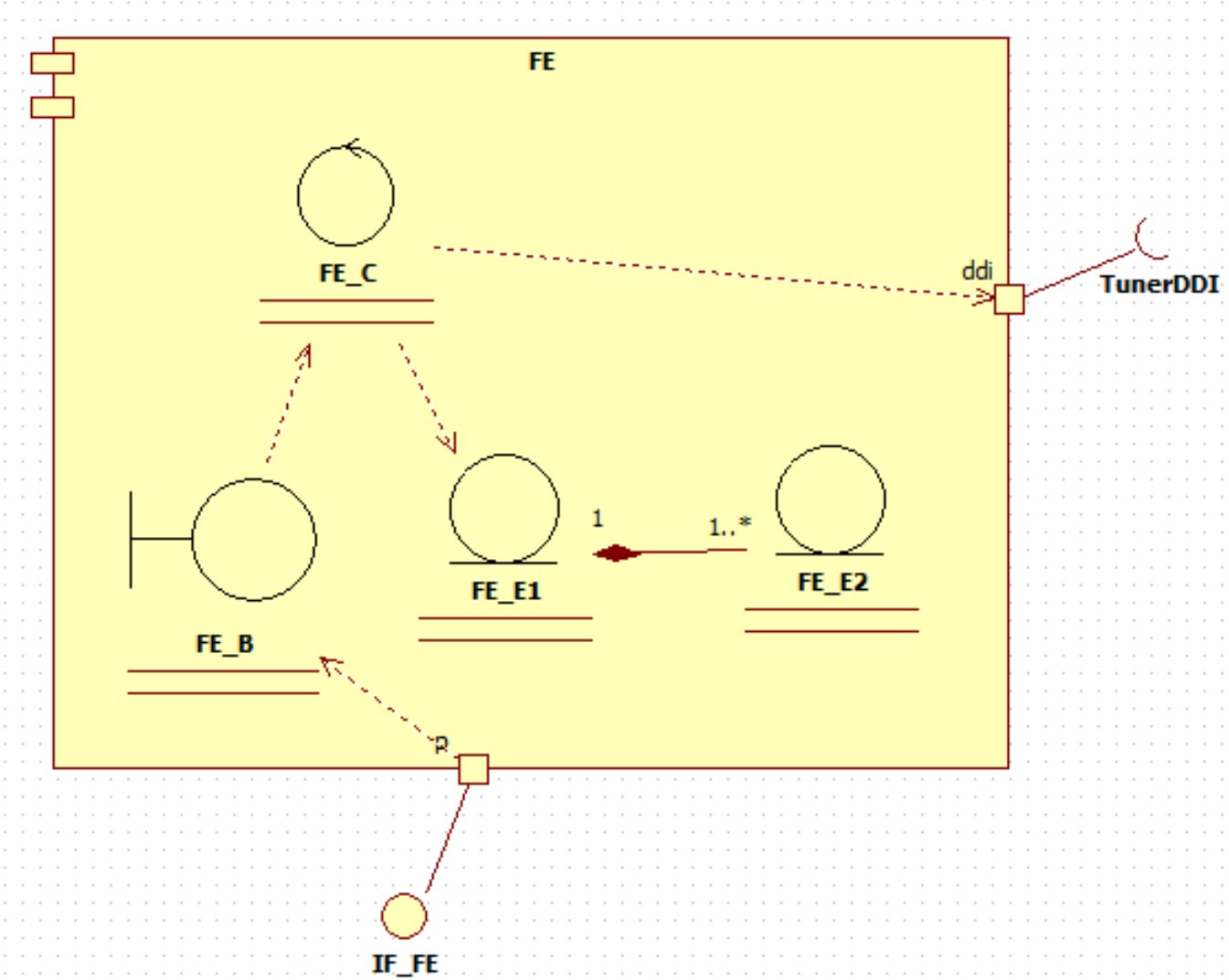
UML Approach (Cont..)

- <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



UML Approach (Cont..)

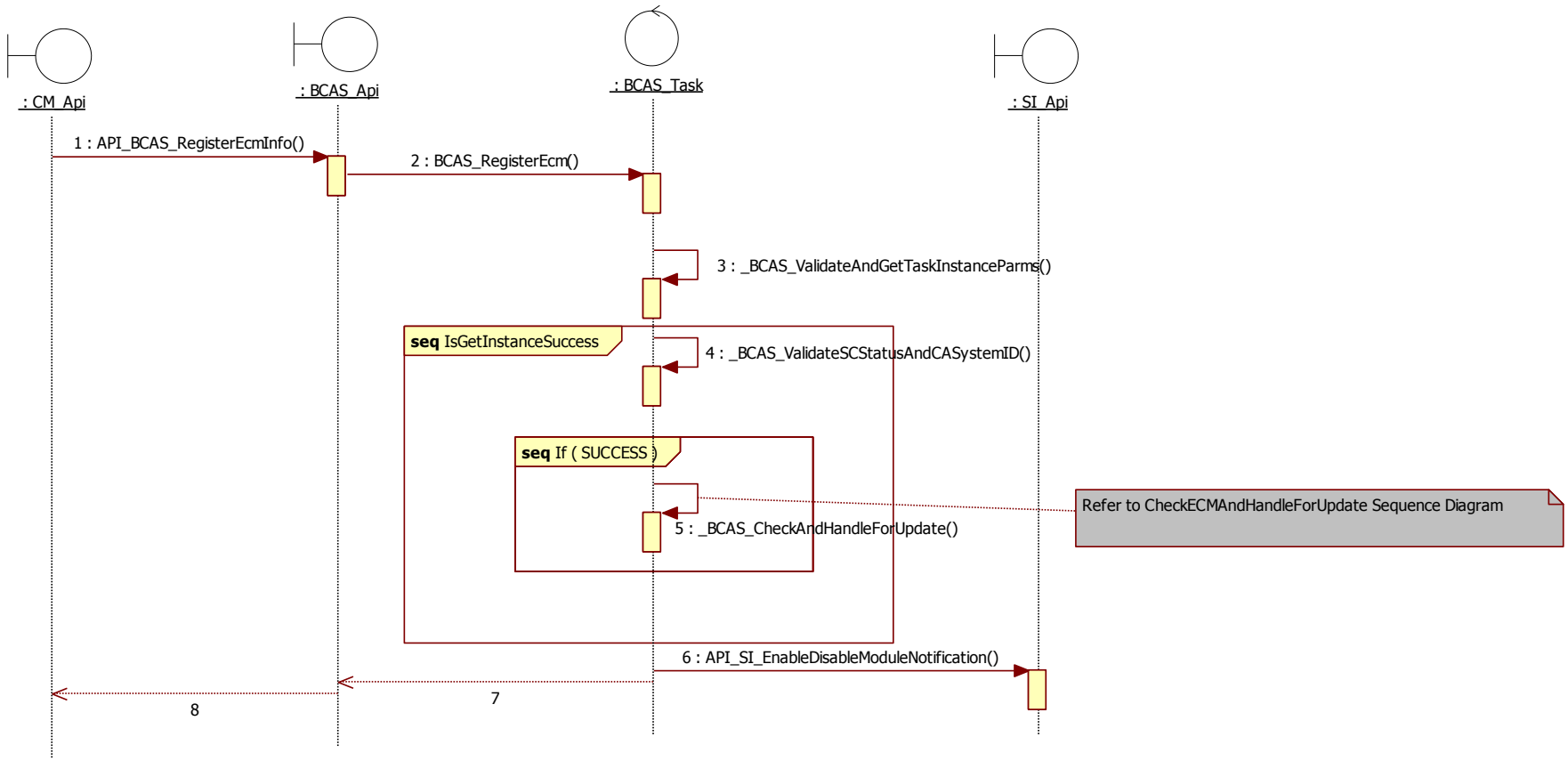
- <Logical View>(8) add following to components
  - State Diagram for a component – if required
  - Activity Diagram for each State – if required

- <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

UML Reference (Cont..)

- 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
  - www.cplusplus.com
  - C++ Primer : Stanley, Josee and Barbar

Credits

- Books:
  - Head First OOAD & Design patterns

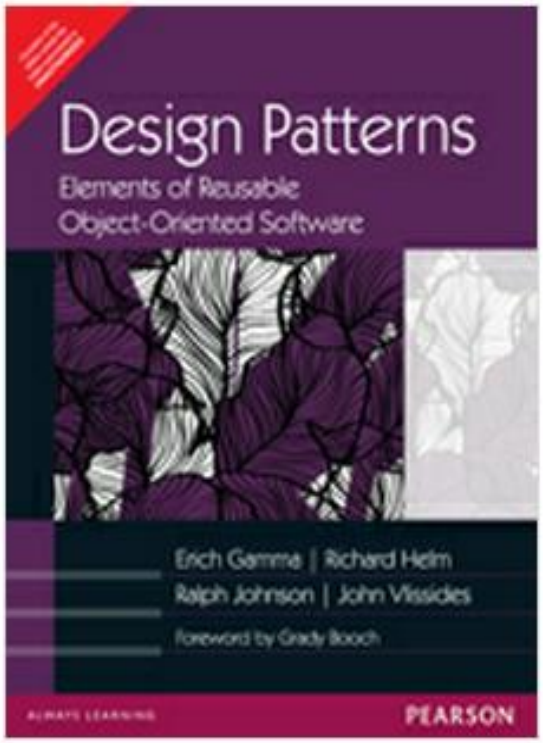
- Web Links:
  - http://www.c-sharpcorner.com/UploadFile/e881fb/simplest-way-to-learn-object-oriented-programming/
  - http://www.slideshare.net/nishithshukla/jump-start-to-oop-ooad-and-design-pattern?from_search=1
  - http://www.c-sharpcorner.com/UploadFile/e881fb/object-oriented-programming-using-C-Sharp-part-10/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

Pattern Reference Cards

Reference Card

Reference Card
(Short)

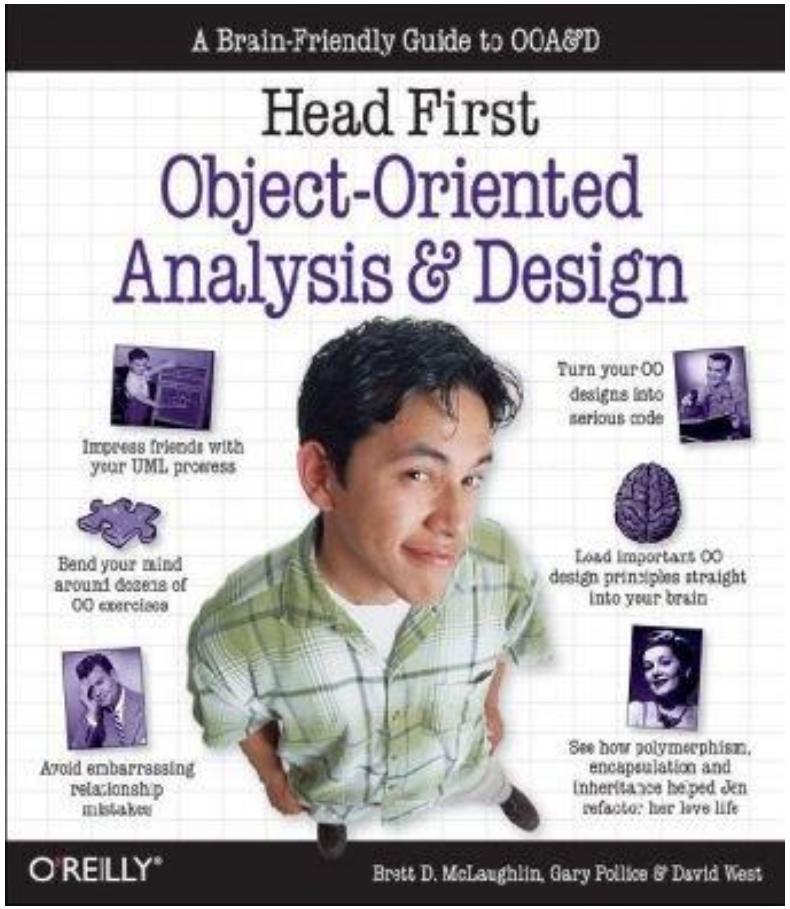
Books (Must Have)



Design Patterns: Elements of Reusable Object-Reusable Object- Oriented Software

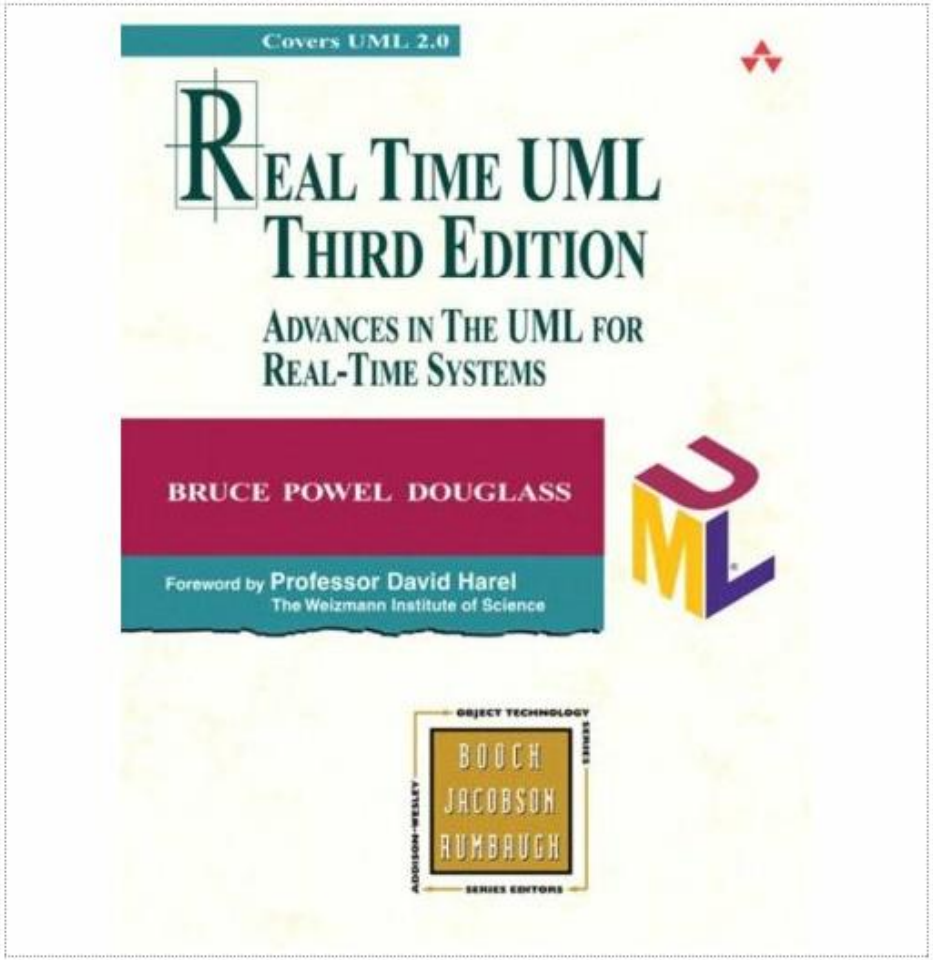
by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Books (Must Have)

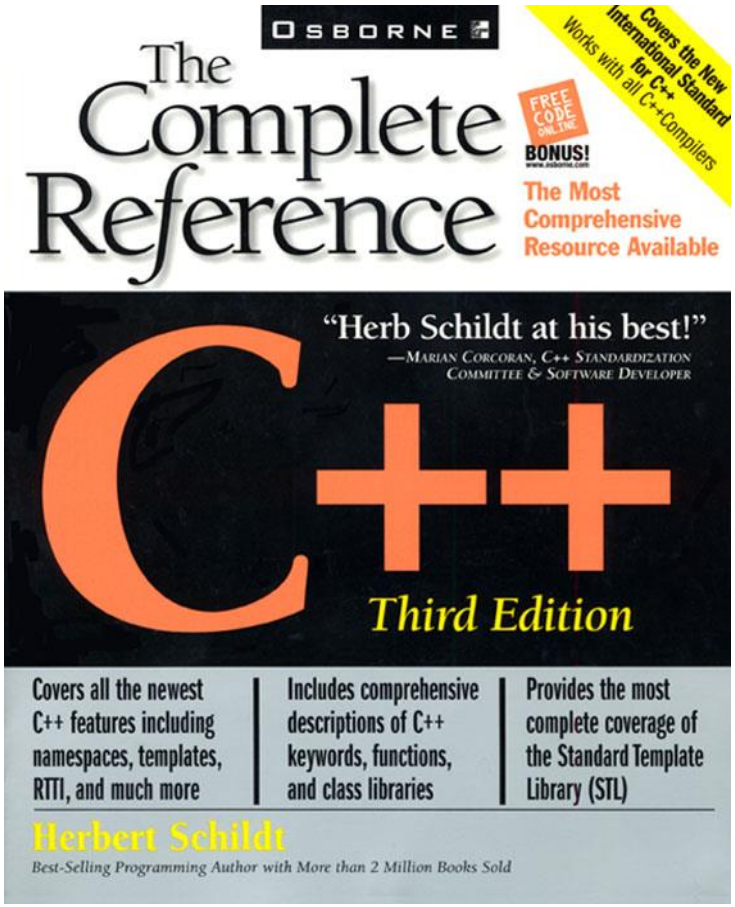


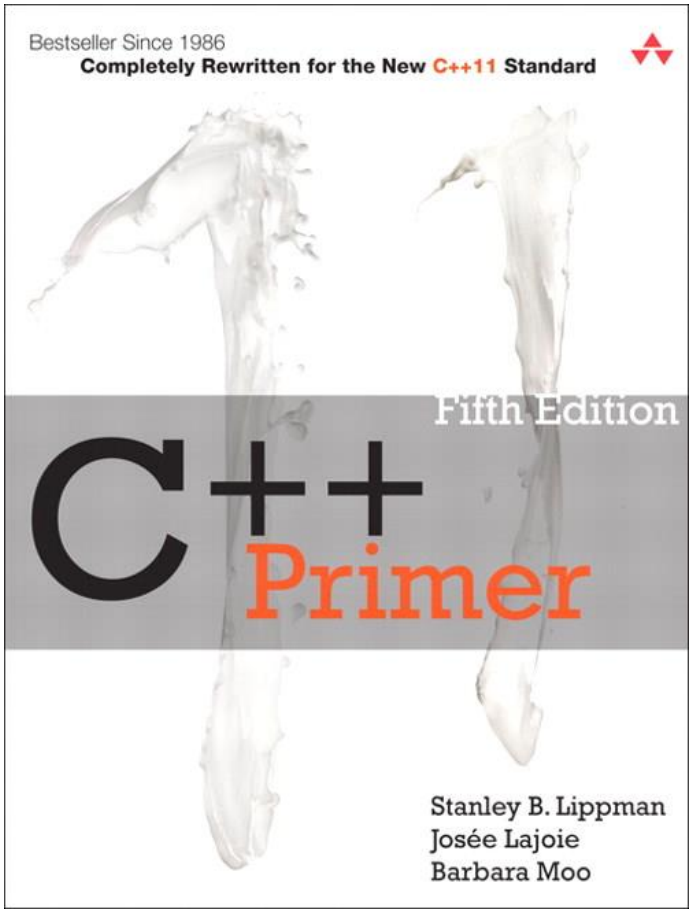
Books (Must Have)

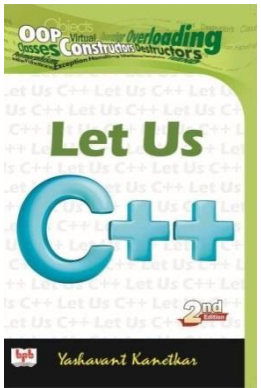
Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)



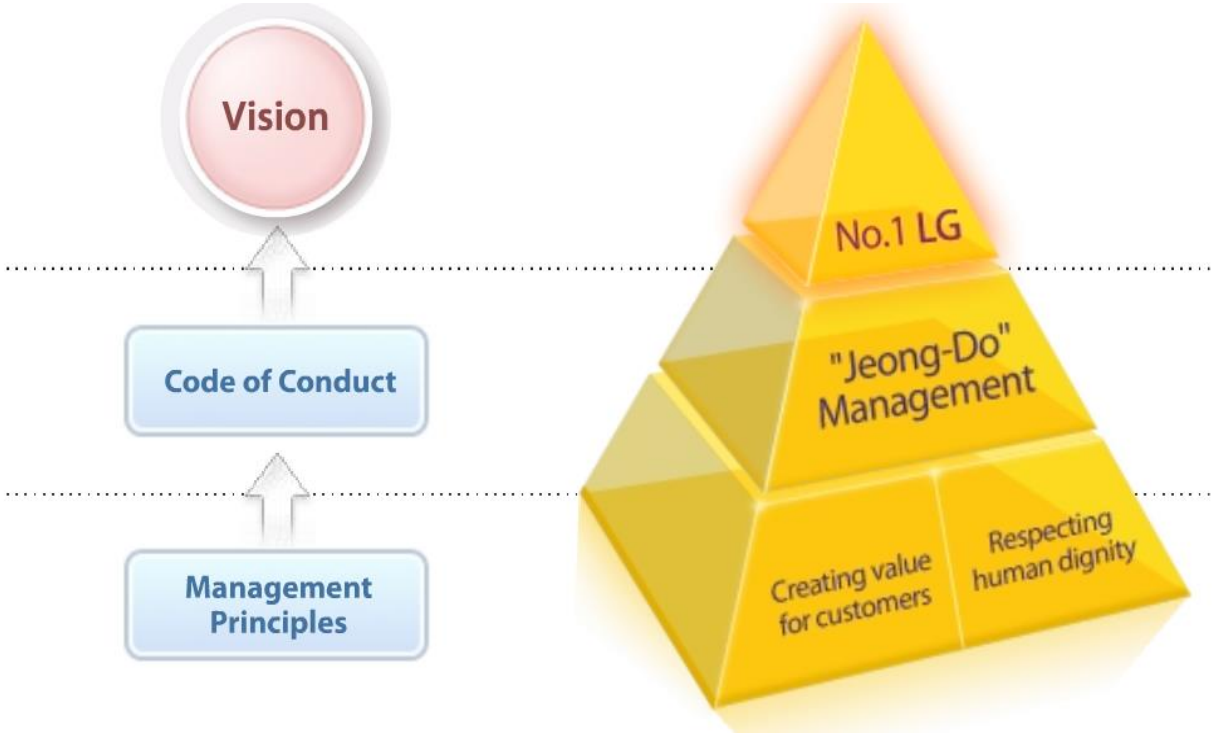
Books (Must Have)







Thank You



LG
Life's Good