OO design principles (I)

Improve your program

- Software design is the process of planning how to solve a problem through software.
- A software design contains enough information for a development team to implement the solution. It is the embodiment of the plan (the blueprint for the software solution).
- A design principle is a basic tool or technique that can be applied to designing or writing code so that code can be more flexible, extensible, and maintainable.

- The Don’t Repeat Yourself Principle (DRY)
- Avoid duplicate code by abstracting out things that are common and placing those things in a single location.
- It's important not to abuse it, duplication is not for code, but for functionality

- How to improve this code segment?

```
public class Report
{
   public void show(String[] data)
   {
       for (i = 0;i < data.length; i++)
       {
          System.out.println("item " + i + data[i]);
       }
   }

   public void save(String[] data)
   {
       String strToFile = "";
       for (i = 0;i < data.length; i++)
       {
          strToFile += "item "+ i + data[i];
       }
       BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));
       writer.write(str);
    }
}
```

Almost identical.

```
class Report
{
   public void show(String[] data)
   {
       print(createReport(data));
   }

   public void save(String[] data)
   {
       BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));
       writer.write(createReport(data));
   }
   private String createReport(String[] data)
   {
       String str = "";
       for (i=0,i<data.length;i++)
       {
          str += "item "+i+data[i];
       }
       return str;
   }
}
```

- DRY is good, but don’t abuse using it.
- The readability may decrease because too many logics inside.
- An alternative – Write Everything Twice (WET)
- You can ask yourself "Haven't I written this before?" two times, but never three.
- If you only have same code in two places, maybe abstraction them immediately is not a good idea because you may have to add some unique features for each in the future. At that moment, you have to unwrap the abstraction, which might be more expensive.

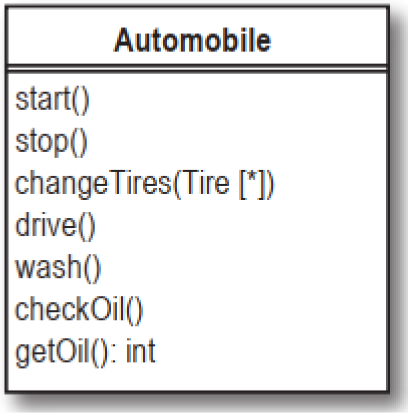
- Five basic design principles – S.O.L.I.D.
- SOLID is an acronym for the following five object-oriented design principles
- S - Single-responsiblity principle
- O - Open-closed principle
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency Inversion Principle

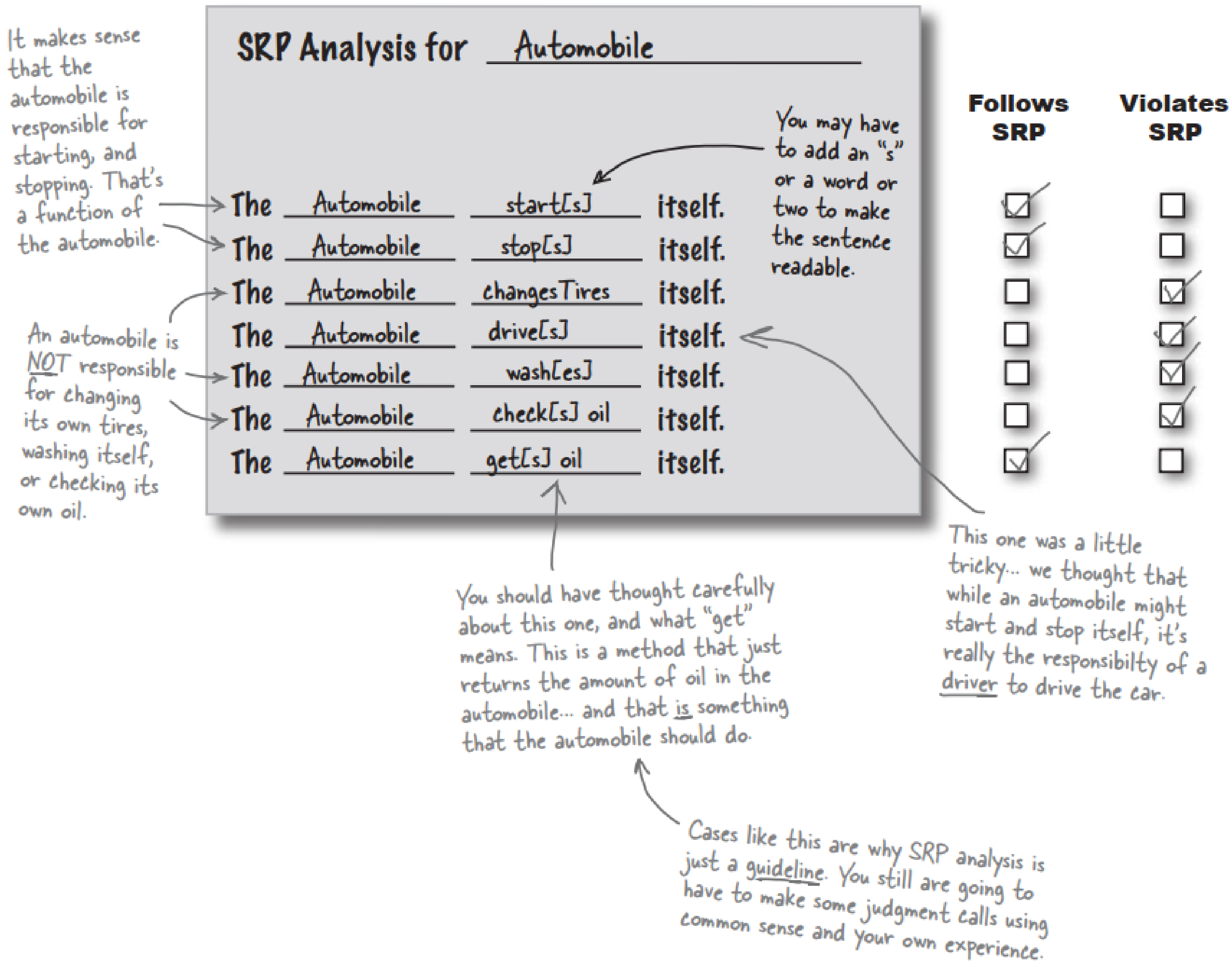
- Single-responsiblity principle (SRP)
- A class should have one and only one reason to change, meaning that a class should have only one job.
- An object can have many behaviors and methods, but all of them are relevant to it's single responsibility.
- Whenever there is a change that needs to happen, there will be only one class to be modified

```
public class AreaCalculator
{
    private float area;

    public void calArea(Circle c)
    {
        float r = c.getRadius();
        area = r*r*3.14f;
    }

    public void output()
    {
        System.out.println("the area is: " + area);
    }
}

public class AreaTester
{
    public static void main(String[] args)
    {
        AreaCalculator cal = new AreaCalculator();
        Circle c = new Circle(3);
        cal.calArea(c);
        cal.output();
    }
}
```

- Q: Anything wrong with this code segment?
- A: The output method has no relation with the AreaCalculator class

- Exercise
- Suppose you are designing a car, and the class you firstly designed is as follow. Can you improve it such that it will not violate the SRP?
- Hint: think what things can the Automobile do by itself?





SRP Analysis for Automobile

	Follows SRP	Violates SRP
The Automobile start[s] itself.	✓	
The Automobile stop[s] itself.	✓	
The Automobile changesTires itself.		✓
The Automobile drive[s] itself.		✓
The Automobile wash[es] itself.		✓
The Automobile check[s] oil itself.		✓
The Automobile get[s] oil itself.	✓	

It makes sense that the automobile is responsible for starting, and stopping. That's a function of the automobile.

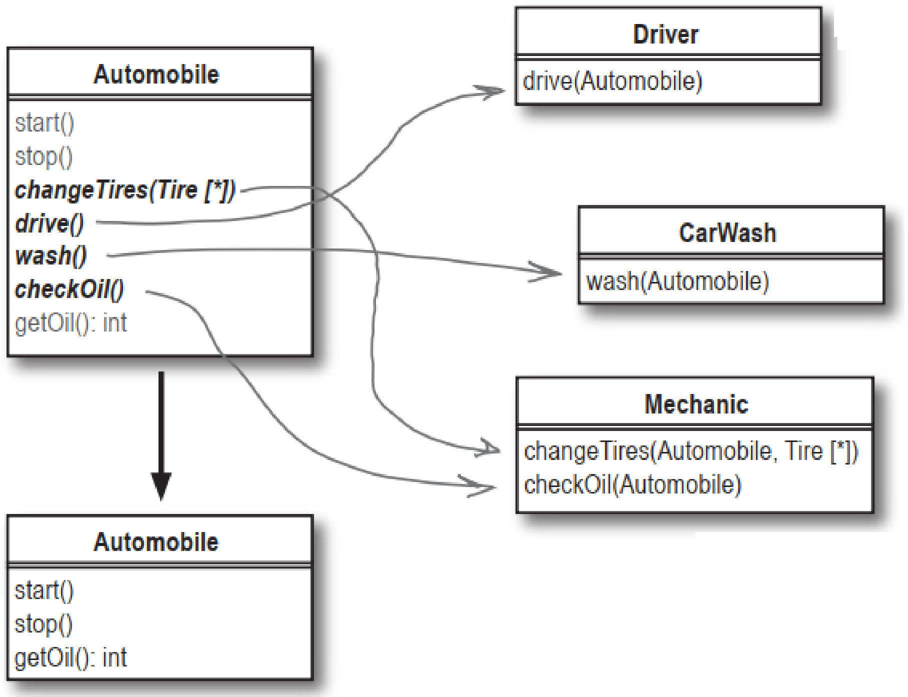
An automobile is NOT responsible for changing its own tires, washing itself, or checking its own oil.

You may have to add an "s" or a word or two to make the sentence readable.

This one was a little tricky... we thought that while an automobile might start and stop itself, it's really the responsibilty of a driver to drive the car.

You should have thought carefully about this one, and what "get" means. This is a method that just returns the amount of oil in the automobile... and that is something that the automobile should do.

Cases like this are why SRP analysis is just a guideline. You still are going to have to make some judgment calls using common sense and your own experience.


Automobile
start()
stop()
changeTires(Tire [*])
drive()
wash()
checkOil()
getOil(): int
Driver
drive(Automobile)
CarWash
wash(Automobile)
Mechanic
changeTires(Automobile, Tire [*])
checkOil(Automobile)
Automobile
start()
stop()
getOil(): int

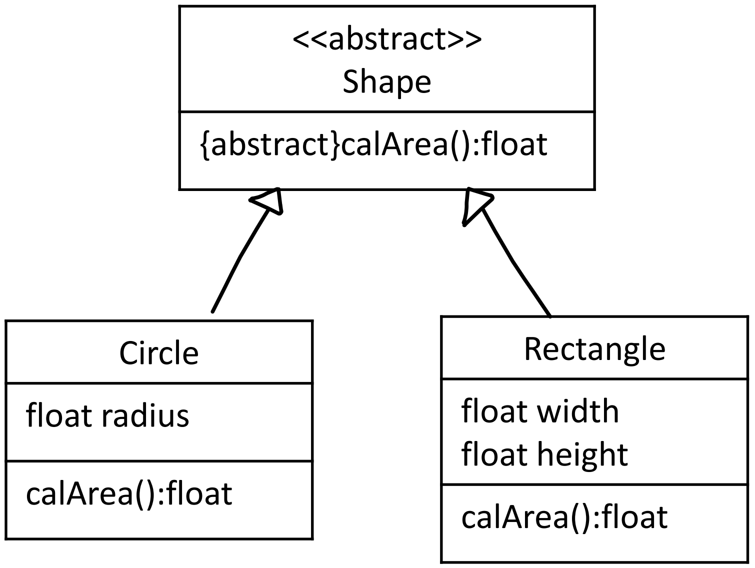
- Open-Closed Principle (OCP)
- Classes should be open for extension, and closed for modification.
- Whenever you need to add additional behaviors, or methods, you don’t have to modify the existing one, instead, you start writing new methods.
- What if you changed a behavior of an object, where some other parts of the system *depends* on it?

```java
public class AreaCalculator
{
    private float area;

    public float calArea(Circle c)
    {
        float r = c.getRadius();
        return r*r*3.14f;
    }

    public float calArea(Rectangle r)
    {
        return r.getWidth()*r.getHeight();
    }

}

public class AreaTester
{
    public static void main(String[] args)
    {
        AreaCalculator cal = new AreaCalculator();
        Circle c = new Circle(3);
        Rectangle r = new Rectangle(2,4);
        float ca = cal.calArea(c);
        float ra = cal.calArea(r);
        System.out.println("area: "+ca+","+ra);
    }
}
```

- Q: Anything wrong with this code segment?
- A: If you need to have a triangle, you have to modify AreaCalculator, which violate the OCP

- Create abstract class Shape

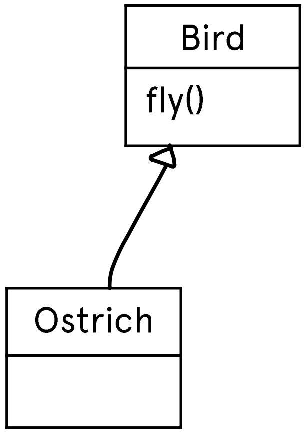
```
public class AreaCalculator
{

    public float calArea(Shape s)
    {
        return s.calArea();
    }

}
```



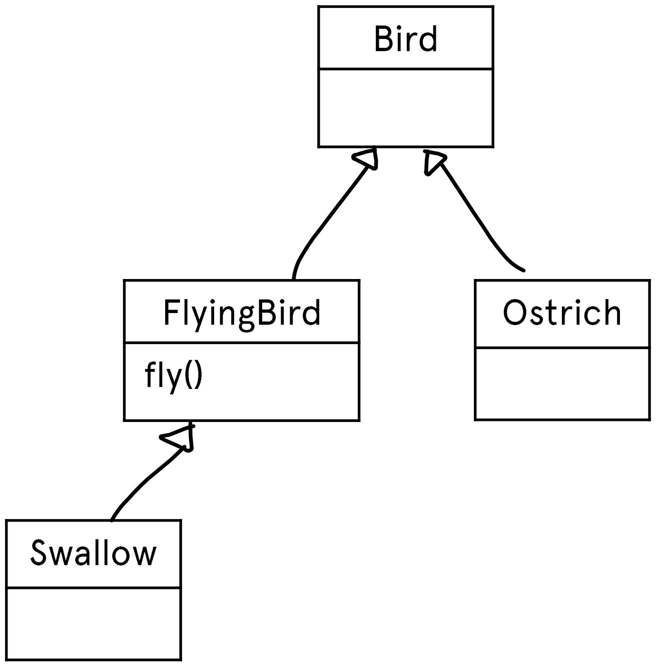
```
public class AreaTester
{
    public static void main(String[] args)
    {
        AreaCalculator cal = new AreaCalculator();
        Shape c = new Circle(4);
        float ca = cal.calArea(c);
        System.out.println("area: "+ca);
    }
}
```

- The Liskov Substitution Principle (LSP)
- Subtypes must be substitutable for their base types.
- The sub classes should extend the functionality of the super class without overriding it

- Use the IS-A test. Is this inheritance reasonable?



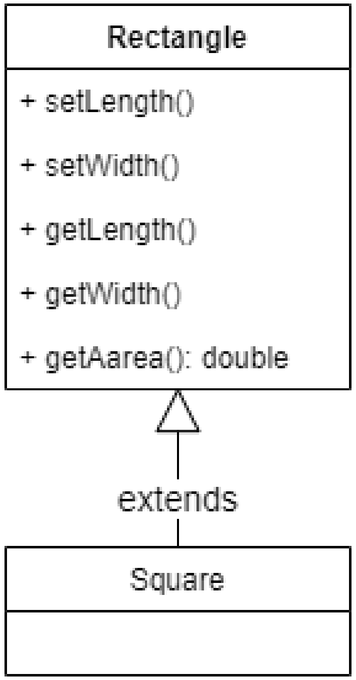
- Much better…
- When you inherit from a base class, you must be able to substitute your subclass for that base class without things going wrong
- IS-A test in fact should be "has the behavior of"



The LSP says that the code should work without knowing the actual class of the Bird object

```
public void
dropBird(~~Bird~~FlyingBird  b){
    b.fly();
}
```

- Quiz
- A square IS-A rectangle, right?



- To reduce the chance to violate LSP, some people suggested:
- Use DBC (design by contract)
- Use composition over inheritance (Use HAS-A instead of IS-A)

```
class Person {
   String title;
   String name;
   int age;
}

class Employee extends Person {
   int salary;
   String title;
}
```

```
class Person {
   String title;
   String name;
   int age;

   public Person(String title, String name, 
String age) {/*constructor*/}

}

class Employee {
   Int Salary;
   private Person person;

   public Employee(Person p, Int salary) {
       person = p;
       Salary = salary;
   }
}
```

- Interface Segregation (隔離) Principle
- A client should never be forced to implement an interface that it doesn't use
- Interfaces should be specific rather than doing many and different things
- Large interfaces should be decomposed into smaller, more specific ones

```
public interface ShapeInterface
{
    float calArea();
    float calVolume();
}


public class Circle extends Shape implements ShapeInterface
{
    private float radius;

    public Circle(float radius)
    {
        this.radius = radius;
    }


    public float calArea()
    {
        return radius*radius*3.14f;
    }

    public float calVolume()
    {
        return 0;
    }
}
```

- A little bit weird...

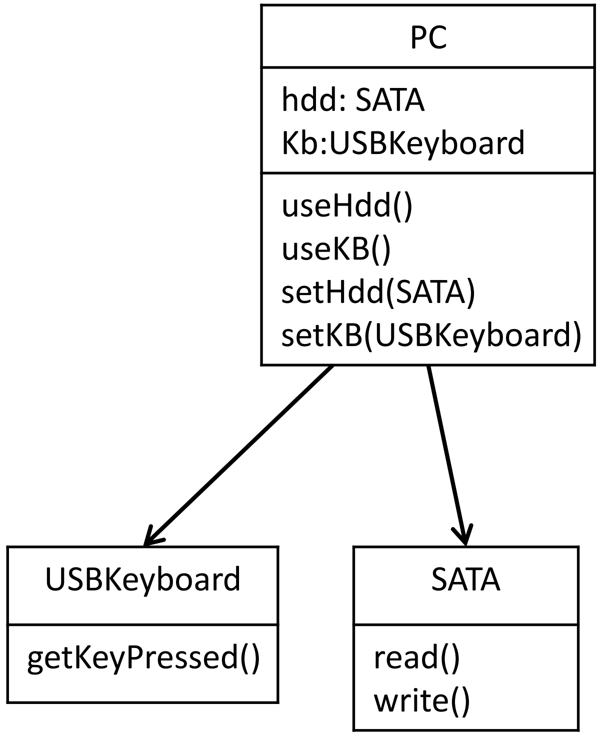
- You could use two interfaces, one has the calArea, another one has the calVolume
- For a Cube class, it should implement both interface
- For a Circle class, it only has to implement one.

- Dependency Inversion Principle
- Try to minimize the dependency between objects by using ***abstraction***
- The coupling problem – how often do changes in a class A force necessary changes in class B

```
class A {
    Dog d = new Dog();
    public void start() {
        d.bark();
    }
}

class Dog {
    public void bark() {
        System.out.println("woof!");
    }
}
```

This is a tight coupling because A directly use a concrete class Dog.

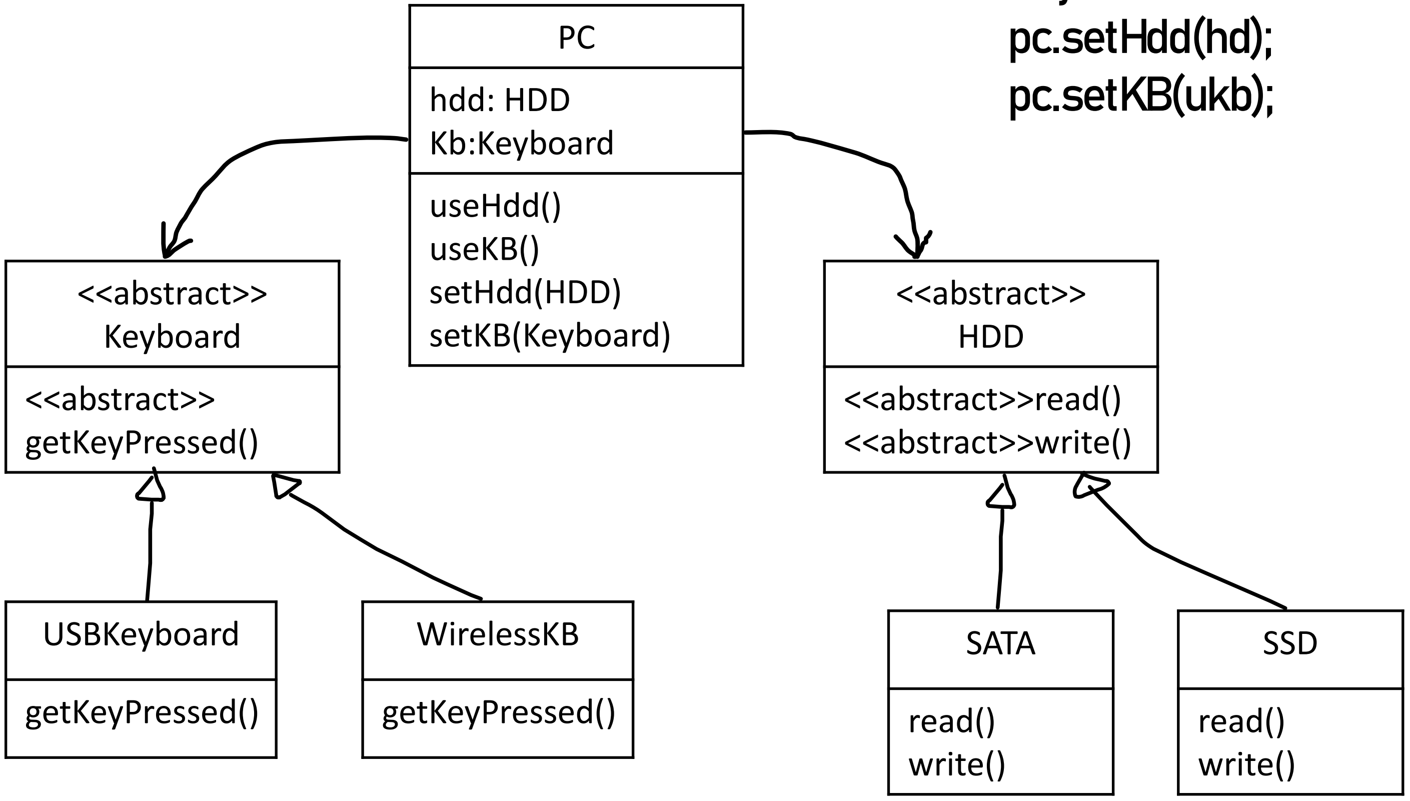
- Imaging that you're going to design an application which operates a PC
- What if I want to use a wireless keyboard and a SSD?



```
PC pc = new PC();
SATA hd = new SATA();
USBKeyboard ukb = new
USBKeyboard();
pc.setHdd(hd);
pc.setKB(ukb);
```

- Much better! (loose coupling)

```
PC pc = new PC();
HDD hd = new SSD();
Keyboard ukb = new USBKeyboard();
pc.setHdd(hd);
pc.setKB(ukb);
```



"program to an interface, not an implementation"

- These design principles might seem to be a handful at first, but with continuous usage and adherence to its guidelines, it becomes a part of you and your code which can easily be extended, modified, tested, and refactored without any problems.

- (software) design pattern
- A general, reusable solution to common occurring problems in software design
- It's a template for how to solve a problem in different situations.
- There are lots of design patterns and many of them are complicated. Here we will give you several simple but frequently used paradigms.

- Simple Factory Pattern
- Regulates how to create new objects
- Let's say we are going to design a RPG game. Firstly, we need to create some characters such as barbarians, wizards, and priests.
- An intuitive way is to write three classes: Barbarian, Wizard, and Priest.

```
class Barbarian{}
class Paladin{}
class Wizards{}
class Priest{}

class Game {
    public void create() {
        Barbarian bar = new Barbarian();
        Priest pri = new Priest();
    }
}
```

could work but not elegant. Developers have to know everything about making desired objects from scratch.

- Create a factory class whose duty is to create objects for caller.

```
class Character{}
class Barbarian extends Character{}
class Wizards extends Character{}
class Priest extends Character{}

class PlayerFactory {
    public Character createPlayer(String type) {
        switch(type) {
            case "Barbarian":
                return new Barbarian();
            case "Wizrads":
                return new Wizards();
            case "Priest":
                return new Priest();
            default:
                return null;
        }
    }
}
```

you can use static method here. That's ok.

The caller only have to tell the factory what he/she want.

```
class Game {
    public void create() {
        PlayerFactory f = new PlayerFactory();
        Character pri = f.createPlayer("Priest");
    }
}
```

- Defect of Simple Factory Pattern
- If you have new characters, you have to modify the PlayerFactory class (by adding new cases), which violates OCP.
- Solution: Factory Method Pattern
- Define an interface to define the behavior of creating an object, let concrete classes decide which class to instantiate.
- Define the PlayerFactory as an interface, create classes such as BarbarianFactory, WizardFactory, etc. to implement this interface

- Define an interface which has a method createPlayer()

```java
interface PlayerFactory {
    Character createPlayer();
}
```

- Create a concrete class BarbarianFactory

```java
class BarbarianFactory implements PlayerFactory {
    public Character createPlayer() {
        return new Barbarian();
    }
}
```

- In the main logic

```java
class Game {
    public void create() {
        PlayerFactory f = new BarbarianFactory();
        Character bar = f.createPlayer();
    }
}
```

Advantage: whenever you need a new character, simply write new class like PriestFactory and Priest then you're free to go!

- Defects of Factory Method Pattern
- For each new class, you have to write a factory method for that class (e.g. for a barbarian class, you need a barbarian factory)
- Now, think about a situation: This factory is not only create players, it also creates weapons.
- We have to put two behaviors into an interface. One is createPlayer() and another one is createWeapon().
- If we use Factory Method, we have to create a factory for new player classes and a weapon factory for new weapons.
- Now you need to have an ***abstract factory*** to manage these behaviors. This is called Abstract Factory Method

- For example, in the game we need a blacksmith factory to produce weapons and armors.
- You can define an interface with two methods:

```java
public interface EquipmentFactory {
    Weapon createWeapon();
    Armor createArmor();
}
```

- You can have several concrete classes such as MeleeFactory / RangerFactory, etc.

```java
public class MeleeFactory implements EquipmentFactory {

    public Weapon createWeapon() {
        return new Sword();
    }

    public Armor createArmor() {
        return new Chainmail();
    }
}
```

```java
public class RangeFactory implements EquipmentFactory{

    public Weapon createWeapon() {
        return new Bow();
    }

    public Armor createArmor() {
        return new LeatherArmor();
    }
}
```

- The Abstract Factory pattern is very similar to the Factory Method pattern. The main difference between a "factory method" and an "abstract factory" is that the factory method is a "method", and an Abstract Factory is an "object".
- With the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses interface and relies on a derived class to handle the desired object instantiation.
- We can say that Abstract Factory is a (logical) group of Factory Methods, i.e. the Abstract Factory is an object that has multiple factory methods on it

- ***Builder pattern***: builds a complex object using simple objects and using a step by step approach. The caller need not to know details about creating that object.

```
/* this is just a demonstration. */
class TripBuilder {

    public TripBuilder setHotel() {
        return this;
    }

    public TripBuilder setDate() {
        return this;
    }

    public TripBuilder setTransportation() {
        return this;
    }

    public Trip create() {
        return new Trip(); // create Trip object
    }

}
```

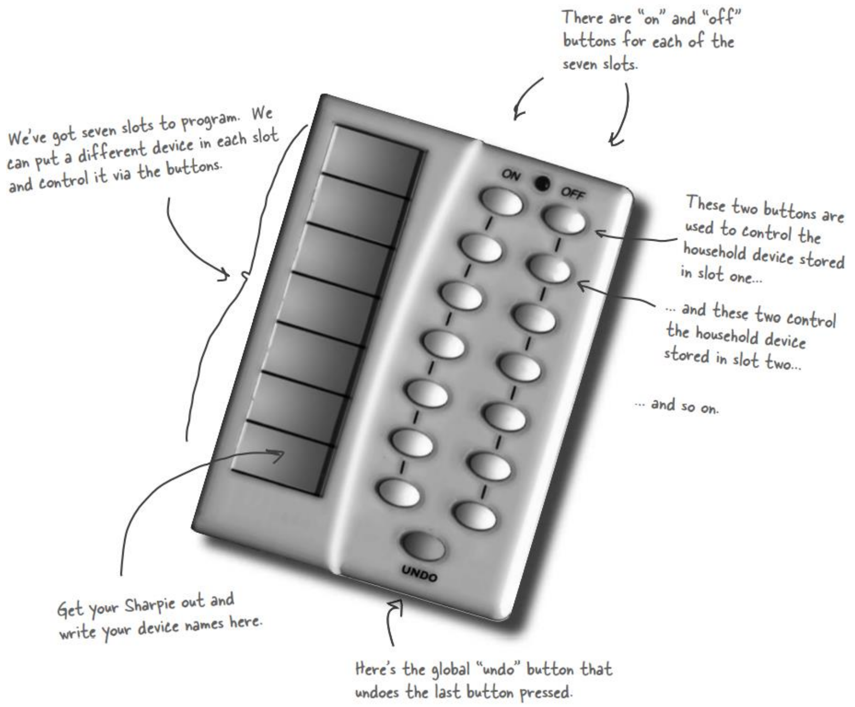
```
// this is called fluent interface
Trip t = (new TripBuilder()).setDate()
                            .setTransportation()
                            .setHotel()
                            .create();
```

the caller knows nothing about how to create a Trip instance.

- ***Singleton***: guarantee that one class will only have one instance.

```java
public class SingletenDemo {

    private static SingletenDemo instance = new SingletenDemo();

    private SingletenDemo() {}

    public static SingletenDemo getInstance() {
        return instance;
    }
}
```

- Think about a scenario of "smart home". In a house, there are plenty of IoT devices (air conditioner, robot vaccum, TV, light, auto sprinkler, etc.)
- I want to use a remote controller (or, an app) to control these things.
- For example, I can design a controller which has many slots along with corresponding on/off buttons for each.


There are "on" and "off" buttons for each of the seven slots.
We've got seven slots to program. We can put a different device in each slot and control it via the buttons.
ON
OFF
These two buttons are used to control the household device stored in slot one...
... and these two control the household device stored in slot two...
... and so on.
UNDO
Get your Sharpie out and write your device names here.
Here's the global "undo" button that undoes the last button pressed.

- A very straightforward thought: for each slot, we write the corresponding behavior for it.

```
if (slot[1] == 'light') {
    light.turnOn();
} else if (slot[1] == 'tub') {
    tub.jetsOn();
}
```

- We know this design sucks. Why?

- The remote should know how to react when it’s buttons are pressed and make requests, but it shouldn’t know a lot about how to turn on a light or hot tub.

- If the remote just knows how to make generic requests, how do we design the remote so that it can invoke an action that, say, turns on a light or opens a garage door?

- The ***Command Pattern*** allows you to decouple the requester of an action from the object that actually performs the action.

- In this example, the requester would be the remote control and the object that performs the action would be an instance of one of your IoT devices

- How is that possible? In fact, it is incredibly simple
- Introducing "command objects" in your design
- A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object)
- When the button is pressed, we just ask the command object to do some work
- The remote doesn't have any idea what the work is, it just has a command object that knows how to get the work done!

- Create a command object
- Store the object in the invoker (remote control in this case)
- The client (user) asks the invoker to execute a command.
- The whole point is that the invoker's job is very simple. It just call a specific method in that command object.

- First things first: all command objects implement the same interface, which consists of one method.
- In the TV class we may have a method named boot(); however, we typically just use the name **execute()**

```java
public interface Command {
    public void execute();
}
```

- Now we can design a LightCommand class that can turn on a light

```java
public class LightCommand implements Command {
    private Light light;

    public LightCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

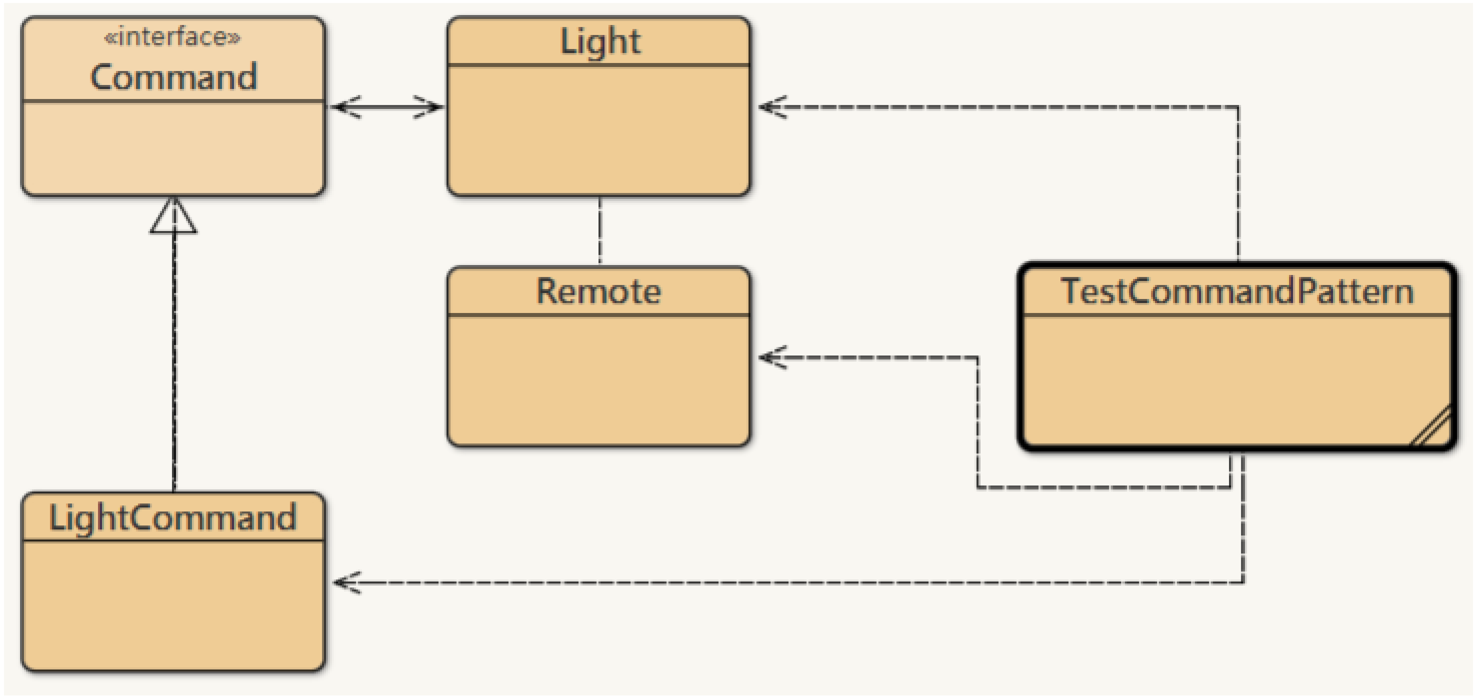
- Say we’ve got a remote control with only one button and corresponding slot to hold a device to control

```
public class Remote {

    private Command slot; // object which implements the Command interface

    public void setCommand(Command cmd){
        this.slot = cmd;
    }

    public void buttonPressed(){
        slot.execute();
    }
}
```

- Let’s make a quick test

```
public class TestCommandPattern {
    public static void main(String[] args){

        Light l = new Light();
        LightCommand lc = new LightCommand(l);
        Remote r = new Remote();
        r.setCommand(lc);
        r.buttonPressed();
    }
}
```

- Relationship between these classes



- Quiz: can you design another button that can be used to control a robot vacuum?

```
public class VacuumCommand implements Command{
    RobotVacuum rv;
    public VacuumCommand(RobotVacuum rv){
        this.rv = rv;
    }

    public void execute(){
        rv.start();
    }
}
```

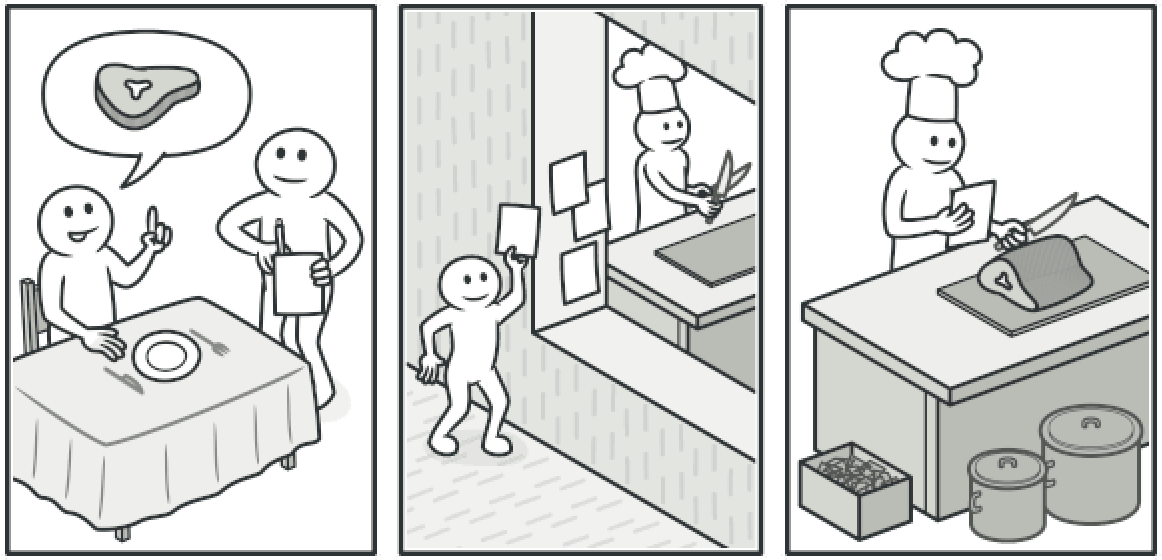
The method which performs the action of the vacuum and the light may be different, so we wrap them into the execute() method.

```
public class Remote {

    private Command [] slot; // object which implements the Command interface

    public Remote() {
        slot = new Command [2];
    }

    public void setCommand(int id, Command cmd){
        this.slot[id] = cmd;
    }

    public void buttonPressed(int id){
        slot[id].execute();
    }
}
```

- Uses of the Command Pattern: queuing requests
- The computation itself may be invoked long after some client application creates the command object
- We can take this scenario and apply it to many useful applications such as schedulers, thread pools and job queues.

- Note that the job queue is totally decoupled from the objects that are doing the computation. The job queue objects don't care what work they have to do; they just retrieve commands and call execute()

- Real-World Analogy

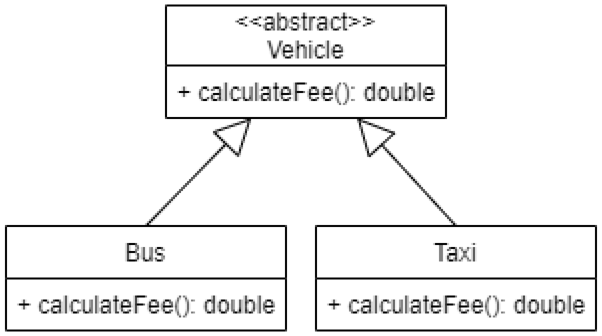


- The paper order serves as a command. It remains in a queue until the chef is ready to serve it. The order contains all the relevant information required to cook the meal. It allows the chef to start cooking right away instead of running around clarifying the order details from you directly.

- Command pattern: encapsulate a request as an object, thereby letting you parameterize clients with different requests or queue
- This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

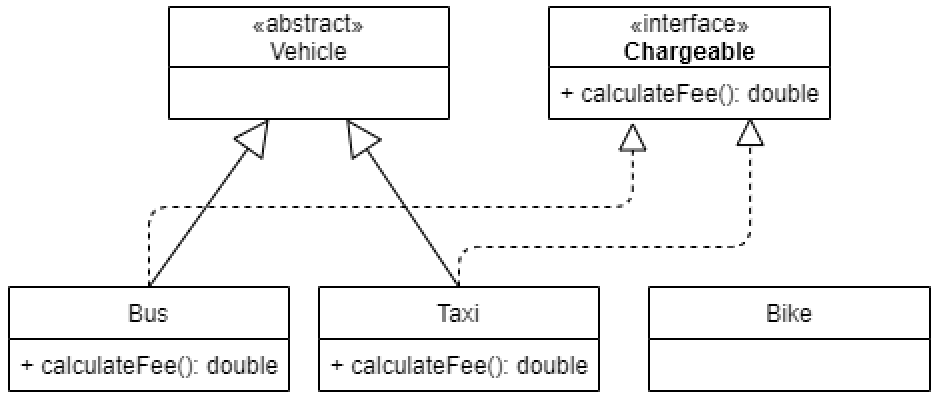
- Quiz: now you know how to execute command from the remote. Can you add an "undo" command for the remote?

- Suppose you want to develop a metropolitan transportation system application. Right now you have two vehicles: bus and taxi.
- Now you want to add a method which is able to calculate the fee of using these vehicles.



- But this design is a little bit inappropriate because different vehicles may have different pricing method. Some vehicles such as bikes may be free of charge.

- So you come up with an idea: how about using interface?



- Better but still has a problem. The calculateFee() is implemented by each concrete class. If we want to change the calculateFee() algorithm for a specific class, we have to modify that class, which violates OCP.
- Furthermore, you may need to pay when riding a public bike in some cities!

- What we want is: we have several algorithms (e.g. chargeByDistance, chargeByStops, chargeByUsage) to do the same thing (calculateFee()) and we want to switch between them very quickly.
- The ***Strategy Pattern*** is your saver. It is a behavioral software design pattern that enables selecting an algorithm at runtime.
- Chance to use: when you have similar behavior with different algorithms.

- Create an interface named, say, ChargeStrategy. Define a method named calculateFee()
- Create concrete classes for charge rules. Each class implements the calculateFee method.

```java
interface ChargeStrategy {
    double calculateFee();
}

class ByTime implements ChargeStrategy {
    private double time;

    public ByTime(double time) {
        this.time = time;
    }

    public double calculateFee() {
        System.out.println("calculating fee by time.");
        if(time < 10.0) {return 0;}
        if(time < 30.0) {return 10;}
        return 20;
    }
}
```

- Make a concrete class, for example, a Bus class. This class has a private instance variable of ChargeStrategy type.
- Users can set the strategy instance via a constructor or a setter.
- Implement a method which invokes the calculateFee of the strategy instance

```java
class Bus {

    private ChargeStrategy strategy;

    public Bus(ChargeStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(ChargeStrategy strategy) {
        this.strategy = strategy;
    }

    public void calculate() {
        double fee = strategy.calculateFee();
        System.out.println("fee:" + fee);
    }
}
```

This is a kind of delegation (i.e., you outsource the calculation job to another object)

- Advantage: users can switch between different algorithms via a setter.

```
public static void main(String[] args) {
    Bus b = new Bus(new ByDistance(15));
      b.calculate();  //calculate by distance
      b.setStrategy(new ByTime(13)); // change strategy
      b.calculate();  //calculate by time
}
```

- If you have a new strategy, simply create a new class and implements the ChargeStrategy interface.

- There are many other interesting design patterns. You are highly encouraged to discover them.

O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates