# Object-oriented design : principles

J.Serrat

102759 Software Design
`http://www.cvc.uab.es/shared/teach/a21291/web/`

August 6, 2014

## Index

1. Information hiding
2. Don’t talk to strangers
3. DRY: Don’t repeat yourself
4. SRP: Single responsibility principle
5. LSP: Liskov substitution principle
6. OCP: Open-closed principle
7. Program to an interface
8. Favor composition

# References

1. *Head first object-oriented analysis and design*. B.D. McLaughlin, G. Pollice, D. West. O'Reilly, 2006. Chapter 8.
2. Articles on design principles, LSP, SRP, OCP from `objectmentor.com` at course web page

# Principles

**Design principle**

Technique or advice to be applied when designing or writing code to make software more maintainable, flexible, or extensible under the inevitable changes.

Extend GRASP patterns.

Where do they come from ? Years of experience in OO design and implementation.

# Information hiding

### Information hiding

Minimize the accessibility of classes and members.

Classes should not expose their internal implementation details.

A component (class, package, API. . . ) should provide *all and only* the information clients need to effectively use it.

Benefits:

- protect clients from changes in the implementation
- also, protect the provider from undue use of internal variables by clients

# Information hiding

In Java,

- set all attributes `private` or `protected`
- add necessary public setters and getters
- set to `private` internal methods, not intended to be used by client classes

```java
class Vehicle {
    private double speed; // in Km/h

    public double getSpeed() {
        return speed;
    }
    public void setSpeed(double s) {
        speed = s;
    }
```

# Information hiding

```
    public action_break(int seconds, double pressure) {
        deceleration = mpsToKmh(
            compute_deceleration(pressure));
        while ( (seconds>0) && (getSpeed()>0) ) {
            double newSpeed = max(0, getSpeed() - deceleration)
            setSpeed(newSpeed)
            delay(1);
            seconds--;
        }
    }
    private double compute_deceleration(double pressure) {
        // some equation relating pedal pressure to
        // speed change in meters per second, each second
    }
    // meters/second to Km/h
    private double mpsToKmh(double mps) {
        return mps*36.0/10.0;
    }
}
```

# Information hiding

Why do it so ?

You can put constraints on values. If clients of `Velocity` accessed speed directly, then they would each be responsible for checking these constraints

```
class Vehicle {
    private double speed; // in Km/h

    public void setSpeed(double s) {
        if ( (s>=0.0) && (s<=MAX_SPEED) ){
            speed = s;
        } else {
            throw SpeedOutOfRangeException();
        }
    }
```

# Information hiding

You can change your internal representation without changing the class interface (i.e. what's public, exposed to the outside)

```
class Vehicle {
    private double speed; // in Miles/h

    public void setSpeed(double s) {
        if ( (s>=0.0) && (s<=MAX_SPEED) ){
            speed = kmhToMph(s);
        } else {
            throw SpeedOutOfRangeException();
        }
    }
    private kmhToMph(double s) {
        return s*0.62137;
    }
```

# Information hiding

You can perform arbitrary side effects. If clients of `Velocity` accessed `speed` directly, then they would each be responsible for executing these side effects.

```
class Vehicle {
    private double speed; // in Km/h

    public void setSpeed(double s) {
        speed = s;
        automatic_change_gears();
        update_wheel_revolutions();
        update_fuel_consuption();
    }
```

# "Don't talk to strangers"

## Don't talk to strangers

An object *A* can request a service (call a method) of an object instance *B*, but object *A* should not "reach through" object `B` to access yet another object *C* to request its services.

Another name for loose coupling.

"Just one point" : in A don't do `getB().getC().methodOfC()`

# "Don't talk to strangers"

```
class Company {
    Collection departments = new ArrayList<Department>();
}
class Department {
    private Employee manager;
    public Employee getManager() {
        return manager;
}
class Employee {
    private double salary;
    public double getSalary() {
        return salary;
    }
}
```

# "Don't talk to strangers"

Don't :

```
// within Company
for (Department dept : departments) {
    System.out.println( dept.getManager().getSalary() );
    // now Company depends on Employee
}
```

Do :

```
class Department {
    //...
    double getManagerSalary() {
        return getManager().getSalary();
    }
}

// within Company
for (Department dept : departments) {
    System.out.println( dept.getManagerSalary() );
}
```

# DRY: Don't repeat yourself

## Don't Repeat Yourself

Avoid duplicate code by abstracting out things that are common and placing those things in a single location.

*One rule, one place.*

DRY is also about responsibility assignment : put each piece of *information* and *behavior* is in a unique, *sensible* place.

Cut and paste [code] is evil.

# DRY: Don't repeat yourself

A software for chemical plant control has a `Valve` class.

Each time a valve is opened, it must automatically close after *n* seconds.

Both `PressureTank` and `Boiler` objects have an output valve.

```
class Valve {
    private open = False;
    public open() {
        open = True;
        // do something
    }
    public close() {
        open = False;
        // do something
    }
}
```

# DRY: Don't repeat yourself

```
class PressureTank {
    private Valve valve = new Valve();
    //...
    public void releasePressure(seconds) {
        valve.open();
        // launch thread so we can return exec. control at once
        final Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            public void run() {
                valve.close();
                timer.cancel();
            }
        }, seconds);
    }
```

# DRY: Don't repeat yourself

```java
class Boiler {
    private List<Valve> inputValves = new ArrayList<Valve>();
    private int timeToFill;
    //...
    public void fillBoiler() {
        for (valve : inputValves) {
            valve.open();
            final Timer timer = new Timer();
            timer.schedule(new TimerTask() {
                public void run() {
                    valve.close();
                    timer.cancel();
                }
            }, (int) (timeToFill/inputValves.size()));
        }
    }
```

What if we wanted later to change how to close the valve ? Or add additional effect like record the opening and closing events ?

# DRY: Don't repeat yourself

```java
public class Valve {
    public void open(int seconds) {
        open = true;
        // do something
        final Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            public void run() {
                close();
                timer.cancel();
            }
        }, seconds);
    }
}
```

# DRY: Don't repeat yourself

```java
class PressureTank {
    private Valve valve = new Valve();
    //...
    public void releasePressure(seconds) {
        valve.open(seconds);
    }
}

class Boiler {
    private List<Valve> inputValves = new ArrayList<Valve>();
    private int timeToFill;
    //...
    public void fillBoiler() {
        for (valve : inputValves) {
            valve.open((int) (timeToFill/inputValves.size()));
        }
    }
}
```

# SRP: Single responsibility principle

## Single Responsibility Principle

Every object in your system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.

One class should have only one reason to change.

SRP is another name for *cohesion*.

Why ? because each responsibility is an axis of change.

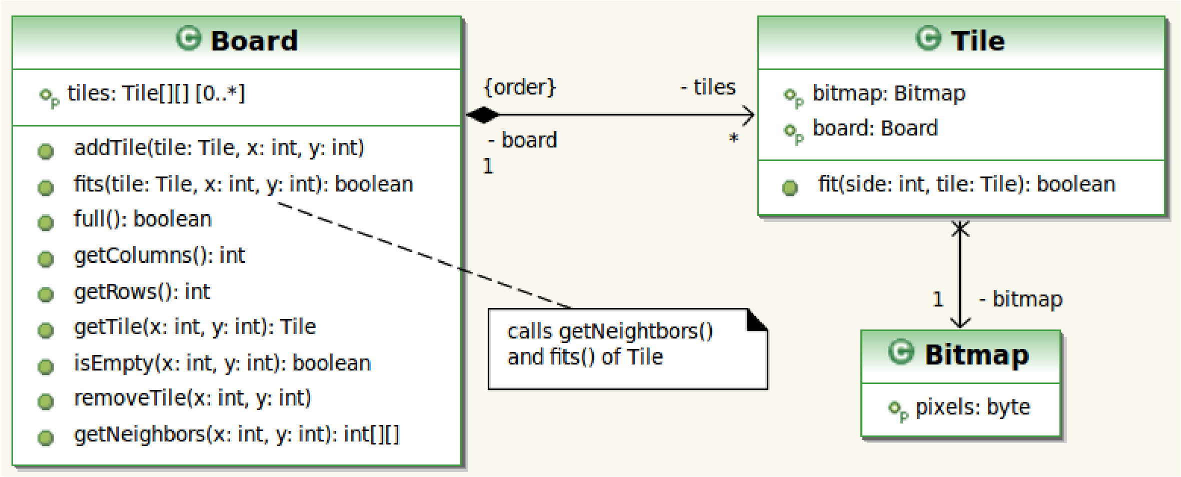
# SRP: Single responsibility principle

The "one responsibility" of a class can be lot of different small tasks, but all related to a single big thing.

The Board of a puzzle game application holds the tiles placed by the user and can

- initialize the board to no tiles
- return the number of rows, columns, tiles in a row, column
- add, remove a tile in a certain position
- check if a tile fits into a position
- check whether all tiles have been placed

# SRP: Single responsibility principle



Board

tiles: Tile[][] [0..*]

addTile(tile: Tile, x: int, y: int)
fits(tile: Tile, x: int, y: int): boolean
full(): boolean
getColumns(): int
getRows(): int
getTile(x: int, y: int): Tile
isEmpty(x: int, y: int): boolean
removeTile(x: int, y: int)
getNeighbors(x: int, y: int): int[][]

{order}
- tiles
- board
1
*

Tile

bitmap: Bitmap
board: Board

fit(side: int, tile: Tile): boolean

1
- bitmap

Bitmap

pixels: byte

calls getNeightbors() and fits() of Tile

All Board methods manage the board one way or another.

## SRP: Single responsibility principle

Would it be ok to put in `Board` a method which takes a picture and generates a grid of tiles ?

```
Tiles[][] makeTiles(Bitmap picture, int rows, int columns)
```

According to GRASP creator, since `Board` contains and uses tiles, yes. But `Board` would loose cohesion

- different number of sides in a tile: square, triangular, hexagonal
- different types of tile profile (difficulty)
- pictures can be in different formats
- tiles can be created from synthetic images, video frames . . .

Better make a `TileFactory` class with this responsibility. And a `Jigsaw` object gives tiles to `Board` constructor.

## LSP: Liskov substitution principle

### Liskov substitution principle

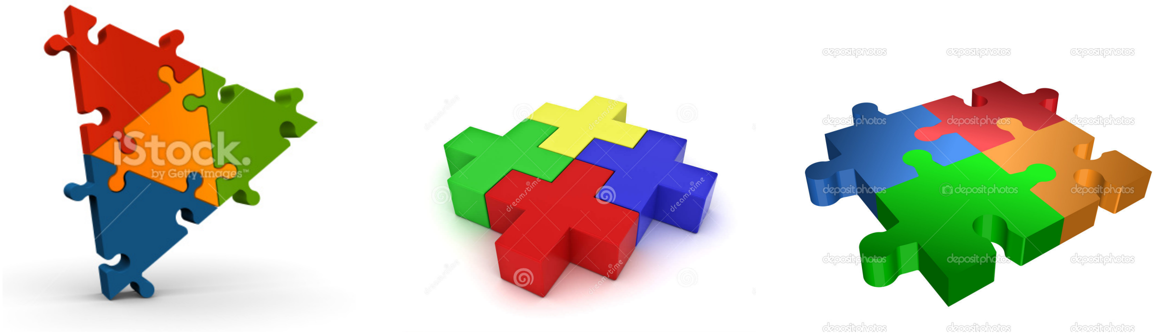
Subtypes must be substitutable for their base types.

Where an object of the derived class is expected, it can be substituted by an object of the base class.

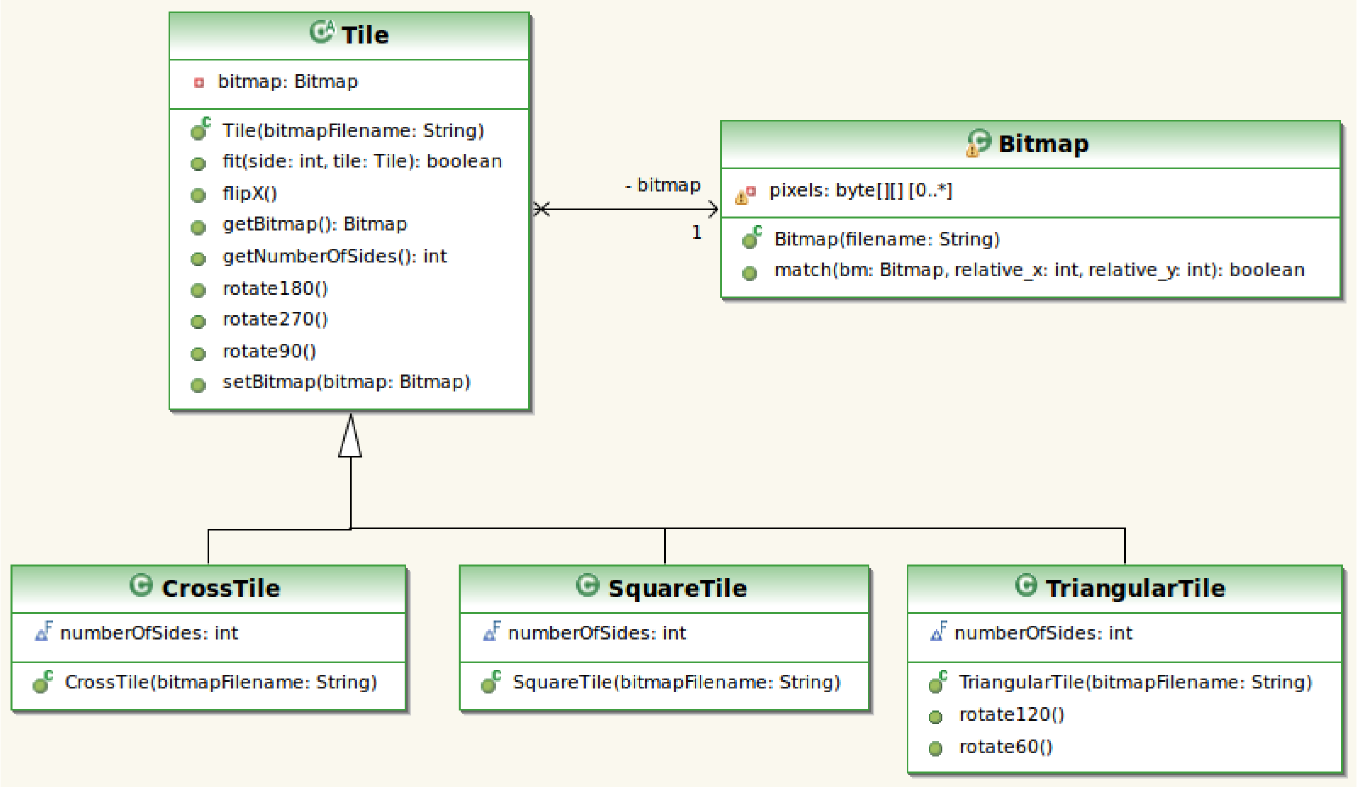
# LSP: Liskov substitution principle

A jigsaw puzzle application

- lets you choose among cross, square and triangular tiles
- tiles have two different faces
- can be rotated and flipped to fit the jigsaw
- one can check whether a piece fits or not in a certain place



# LSP: Liskov substitution principle



**Tile**

bitmap: Bitmap

Tile(bitmapFilename: String)
fit(side: int, tile: Tile): boolean
flipX()
getBitmap(): Bitmap
getNumberOfSides(): int
rotate180()
rotate270()
rotate90()
setBitmap(bitmap: Bitmap)

- bitmap
1

**Bitmap**

pixels: byte[][] [0..*]

Bitmap(filename: String)
match(bm: Bitmap, relative_x: int, relative_y: int): boolean

**CrossTile**

numberOfSides: int

CrossTile(bitmapFilename: String)

**SquareTile**

numberOfSides: int

SquareTile(bitmapFilename: String)

**TriangularTile**

numberOfSides: int

TriangularTile(bitmapFilename: String)
rotate120()
rotate60()

# LSP: Liskov substitution principle

What's wrong here ?

```
boolean tryToAdd1(Board board, Tile t, int x, int y) {
    int angle = 0;
    boolean fits = board.fits(t, x, y);
    while ((angle<=270) && !fits) {
        // does tile t match neighbor tiles in (x, y), if any ?
        // which are the neighbor tiles is known by board
        angle += 90;
        t.rotate90();
        fits = board.fits(t, x, y);
    }
    return fits;
}
```

# LSP: Liskov substitution principle

What about moving `rotate60()`, `rotate120()` from `TriangularTile` to the base class `Tile` ?

Misuse of inheritance: again not all base methods apply to all subclasses. Would make design and implementation confusing.

# LSP: Liskov substitution principle

Is this any better ?

```java
boolean tryToAdd2(Board board, Tile t, int x, int y) {
    int angle = 0;
    boolean fits = board.fits(t, x, y);
    if (t instanceof TriangularTile) {
        while ((angle<=120) && !fits) {
            angle += 60;
            ((TriangularTile) t).rotate60();
            fits = board.fits(t, x, y);
        }
    } else if ( (t instanceof CrossTile) | (t instanceof SquareTile) ) {
        while ((angle<=270) && !fits) {
            angle += 90;
            t.rotate90();
            fits = board.fits(t, x, y);
        }
    }
    return fits;
}
```

# LSP: Liskov substitution principle

We already saw in GRASP this was a bad idea: redundant code, problem if we add a new tile class (hexagonal).

How to solve it ?

Replace all `rotateX()` methods for

- base method `rotate()` , which rotates the tile by
- `int unitRotationAngle`, different for each subclass

# LSP: Liskov substitution principle

```java
boolean tryToAdd3(Board board, Tile t, int x, int y) {
    int angle = 0;
    boolean fits = board.fits(t, x, y);
    while ( (angle <= t.getMaximumAngle())
            // 120 for TriangleTile, 270 for Cross, SquareTile
            && !fits ) {
        angle += t.getUnitRotationAngle();
        // 60 for TriangleTile, 90 for Cross, SquareTile
        t.rotate();
        // rotates this unit rotation angle
        fits = board.fits(t, x, y);
    }
    return fits;
}
```

# OCP: Open-closed principle

**Open-Closed Principle**

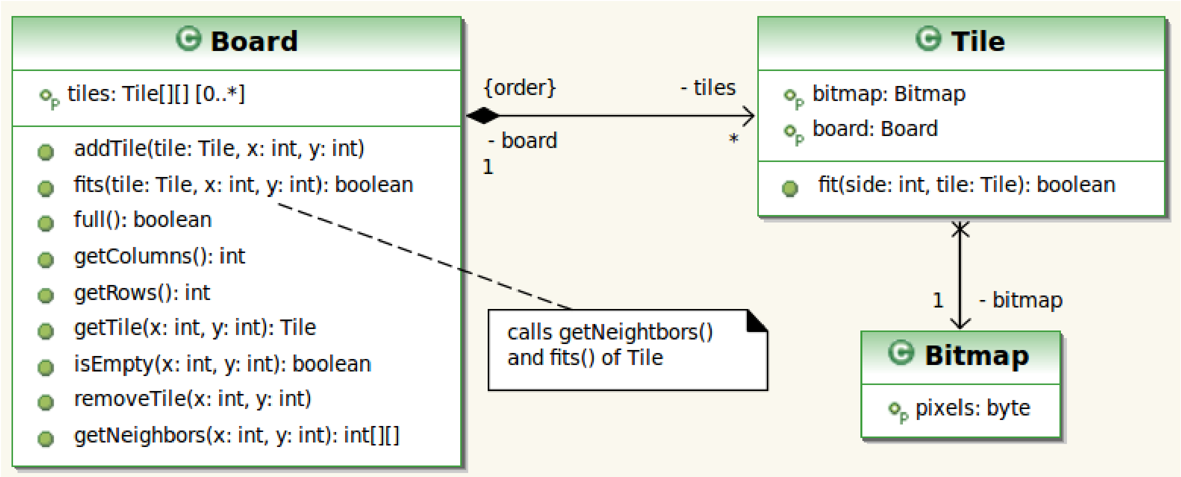
Classes should be open for extension, and closed for modification.

Intent: allow change, but doing it without requiring to modify existing code.

How :

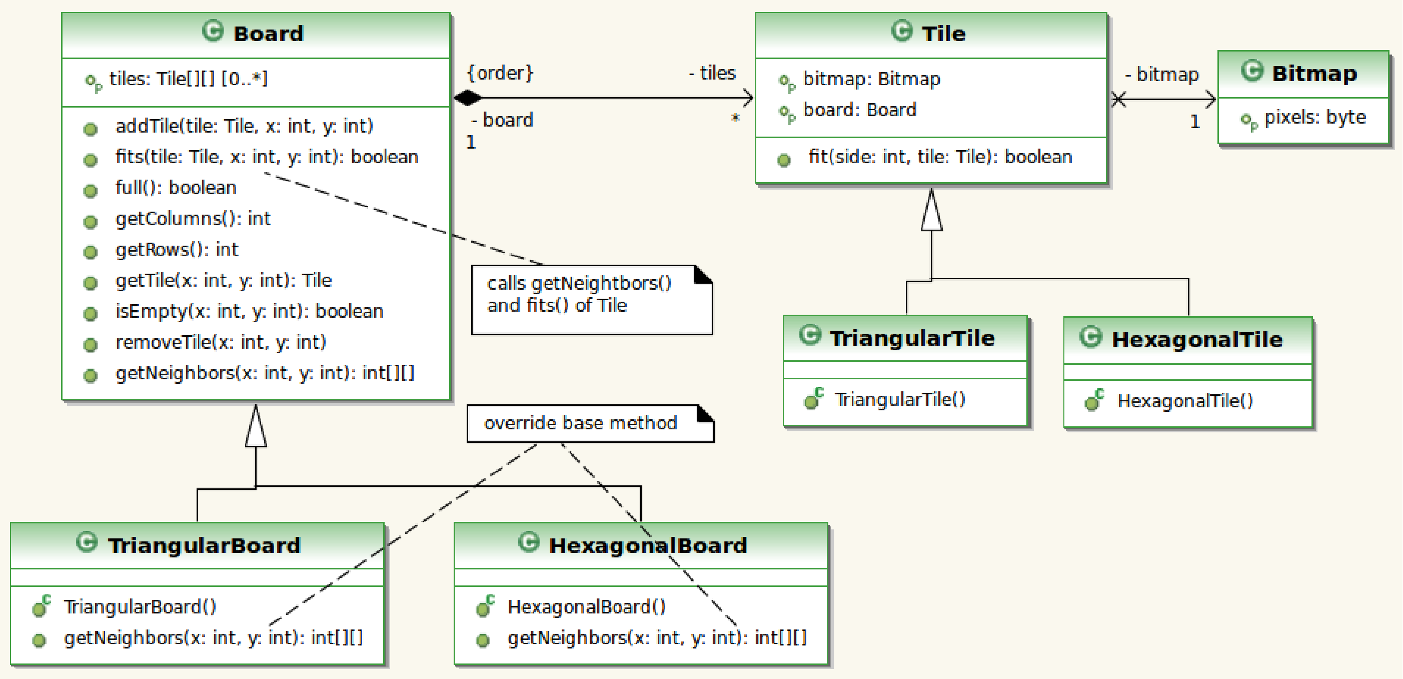
- once implemented a class, do not modify it
- if a change request comes, subclass it and override methods
- or use composition (see later "favor composition")

# OCP: Open-closed principle


Board
tiles: Tile[][] [0..*]
addTile(tile: Tile, x: int, y: int)
fits(tile: Tile, x: int, y: int): boolean
full(): boolean
getColumns(): int
getRows(): int
getTile(x: int, y: int): Tile
isEmpty(x: int, y: int): boolean
removeTile(x: int, y: int)
getNeighbors(x: int, y: int): int[][]
{order}
- tiles
- board
1
*
calls getNeightbors()
and fits() of Tile
Tile
bitmap: Bitmap
board: Board
fit(side: int, tile: Tile): boolean
1
- bitmap
Bitmap
pixels: byte

Now we need also to represent jigsaws with triangular and hexagonal tiles.

# OCP: Open-closed principle



# OCP: Open-closed principle

Ideally, make changes and avoid client classes to be affected by them because they rely on the interface of the base class (see "program to an interface, not an implementation").

Not always possible. We would better change `Board` and `Tile` to abstract classes, and add `SquareBoard`, `SquareTile`, to increase extendability.

# "Program to an interface, not an implementation"

Suppose your application must manage a *sequence* of students enrolled in a course. Which is the best choice in Java ?

(a) an array `Student employees[MaxNumStudents]`

(b) `ArrayList<Student> students = new ArrayList<Student>();`

(c) `LinkedList<Student> students = new LinkedList<Student>();`

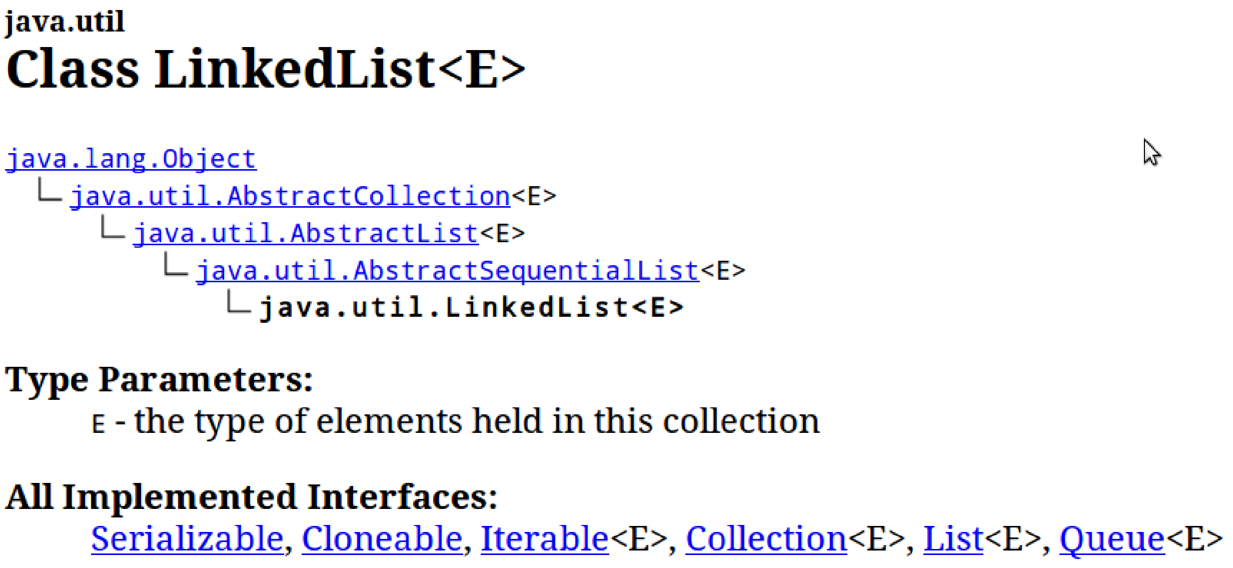
(d) `Vector<Student> students = new Vector<Student>();`

(e) none of them

(a) is surely a bad choice: it can not be resized.

`ArrayList`, `LinkedList`, `Vector` are specific classes of lists derived from `AbstractList` and implementing the `List` interface.

# "Program to an interface, not an implementation"



**java.util**

## Class LinkedList<E>

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.AbstractSequentialList<E>
              └ java.util.LinkedList<E>
```

**Type Parameters:**

E - the type of elements held in this collection

**All Implemented Interfaces:**

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, Queue<E>

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/LinkedList.html

# "Program to an interface, not an implementation"

What's the difference ?

`ArrayList`

*Resizable-array implementation of the List interface plus methods to manipulate the size of the array that is used internally to store the list.*

`LinkeList`

*Linked list implementation of the List interface plus provides uniformly named methods to get, remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue (deque).*

*The class implements the* `Queue` *interface, providing first-in-first-out queue operations for push, pop, etc.*

# "Program to an interface, not an implementation"

`Vector`

*Implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.*

*Roughly equivalent to ArrayList, except that it is synchronized.*

# "Program to an interface, not an implementation"

| | `LinkedList<E>` | `ArrayList<E>` | `Vector` | `List` | `Queue` |
|---|---|---|---|---|---|
| `add(E el)` | ● | ● | ● | ● | |
| `add(int index, E el)` | ● | ● | ● | ● | |
| `addElement(E el)` | | | ● | | |
| `addFirst(E el)` | ● | | | | |
| `addLast(E el)` | ● | | | | |
| `offer()` | ● | | | | ● |
| `remove()` | ● | | | | ● |
| `remove(int index)` | ● | ● | ● | ● | |
| `remove(Object ob)` | ● | ● | ● | ● | |
| `removeFirst()` | ● | | | | |
| `removeLast()` | ● | | | | |
| `poll()` | ● | | | | ● |
| `get(int index)` | ● | ● | ● | ● | |
| `getFirst(E el)` | ● | | | | |
| `getLast(E el)` | ● | | | | |
| `firstElement()` | | | ● | | |
| `lastElement()` | | | ● | | |
| `set(int index, E el)` | ● | ● | ● | ● | |

# “Program to an interface, not an implementation”

| | ArrayList<E><br>Vector<E> | LinkedList<E> |
|---|---|---|
| get(int index) | O(1) | O(n) |
| add(E el) | O(n) * | O(1) |
| add(int index, E el) | O(n) * | O(n) |
| remove(int index) | O(n−index) | O(n) |
| Iterator.remove() | O(n−index) | O(1) |
| ListIterator.add(E el) | O(n−index) | O(1) |

* worst-case since the array must be resized and copied

**Answer**: it depends. If frequent insertions/deletions not in the end of the list, use LinkedList.

# “Program to an interface, not an implementation”

Then should we do like this ?

```
class Course {
    LinkedList<Student> students = new LinkedList<Student>();
    //...
    public void enrollStudent(Student st) {
        students.addLast(st); // exists only for linked lists
    }
}

class Listing {
    //...
    public void printStudents(LinkedList<Student> stlist) {
        // sort list
        Iterator it = stlist.Iterator();
        while (it.hasNext()) {
            student = it.next();
            printStudent(st);
        }
    }
}
```

## "Program to an interface, not an implementation"

This is "programming to an implementation": the code depends on a concrete class, the `LinkedList` subclass, which is one implementation of the `List` supertype.

What if later on we need to change the type of list to `ArrayList` to speed up list traversal or sorting ? We would need to replace *everywhere*

- `LinkedList` ⟶ `ArrayList`
- `addLast()` ⟶ `add()`
- `addFirst()`, `addlast()`, `removeFirst()`, `removeLast()`, `getFirst()` . . . by something else

But we could change our mind again and switch to a `Vector` !
`add()` ⟶ `addElement()` . . .

## "Program to an interface, not an implementation"

It is better to "program to an interface" which means a supertype, *exploiting polymorphism so that code does not depend on the type of the actual runtime object*

In Java : an interface, abstract or super class.

In our case: write code that is ok as long as he student list is of a type implementing the Java interface `List`

# “Program to an interface, not an implementation”

```java
class Course {
    List<Student> students = new LinkedList<Student>();
    // could be also ArrayList or Vector
    public void enrollStudent(Student st) {
        students.add(st); // to the end
    }
}

class Listing {
    //...
    public void printStudents(List<Student> stlist) {
        // sort list
        Iterator it = stlist.Iterator();
        while (it.hasNext()) {
            student = it.next();
            printStudent(st);
        }
    }
}
```

# “Program to an interface, not an implementation”

Suppose your application must manage a queue of student teams (`StudentTeam` objects) which present their project to the assistant lecturer. What would be the best class to represent them ?

(a) an array `StudentTeam teams[MaxNumStudentTeams]`

(b) `ArrayList<StudentTeam> students = new ArrayList<StudentTeam>();`

(c) `LinkedList<StudentTeam> students = new LinkedList<StudentTeam>();`

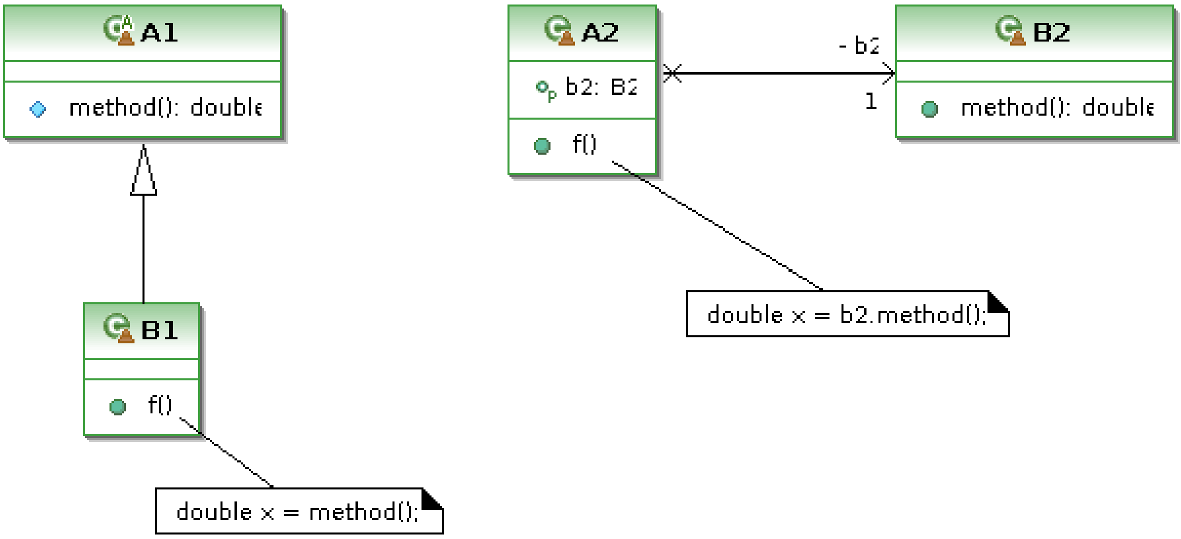
(d) `Vector<StudentTeam> students = new Vector<StudentTeam>();`

(e) none of them

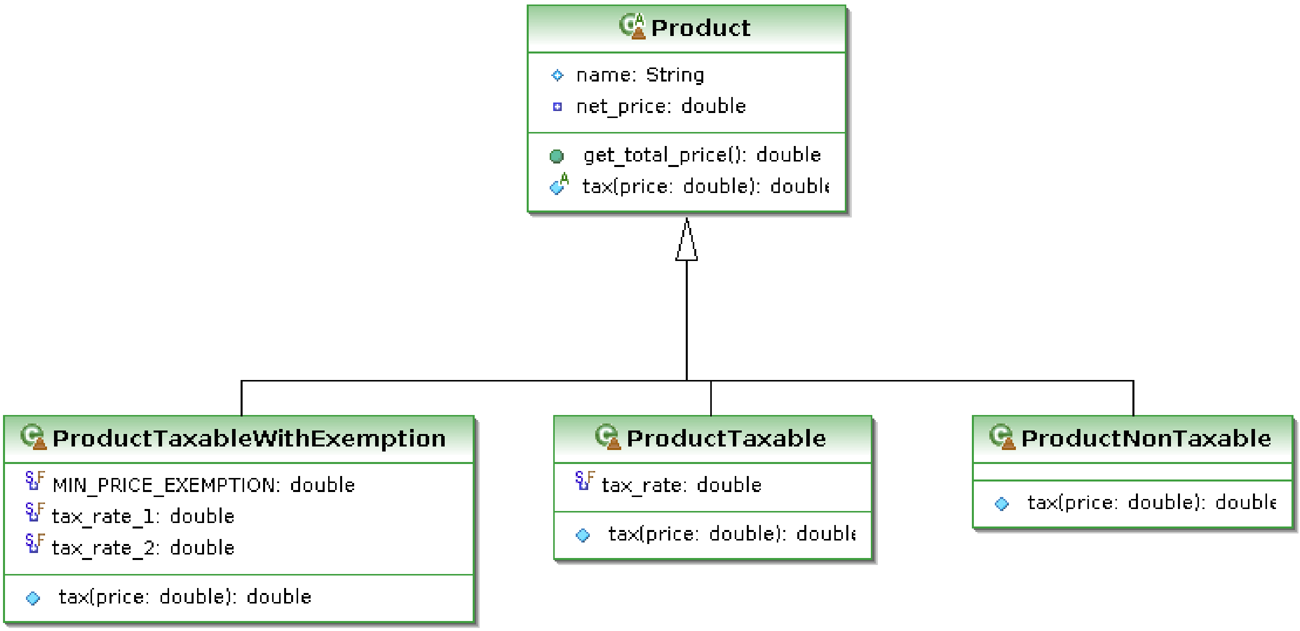
`Queue<StudentTeam> students = new Queue<StudentTeam>();` because of this

# "Favor composition over inheritance"

Composition and inheritance are two ways of getting some functionality from another class.



# "Favor composition over inheritance"


Product
name: String
net_price: double
get_total_price(): double
tax(price: double): double
ProductTaxableWithExemption
MIN_PRICE_EXEMPTION: double
tax_rate_1: double
tax_rate_2: double
tax(price: double): double
ProductTaxable
tax_rate: double
tax(price: double): double
ProductNonTaxable
tax(price: double): double

# “Favor composition over inheritance”

```java
abstract class Product {
    protected String name;
    private double net_price;
    //... setters and getters
    public double get_total_price() { // net price plus tax
        return net_price + tax(net_price);
    }
    protected abstract double tax(double price);
}

class ProductTaxable extends Product {
    private static final double tax_rate = 0.21; // 21% VAT
    @Override
    protected double tax(double price) {
        return tax_rate*price;
    }
}
```

# “Favor composition over inheritance”

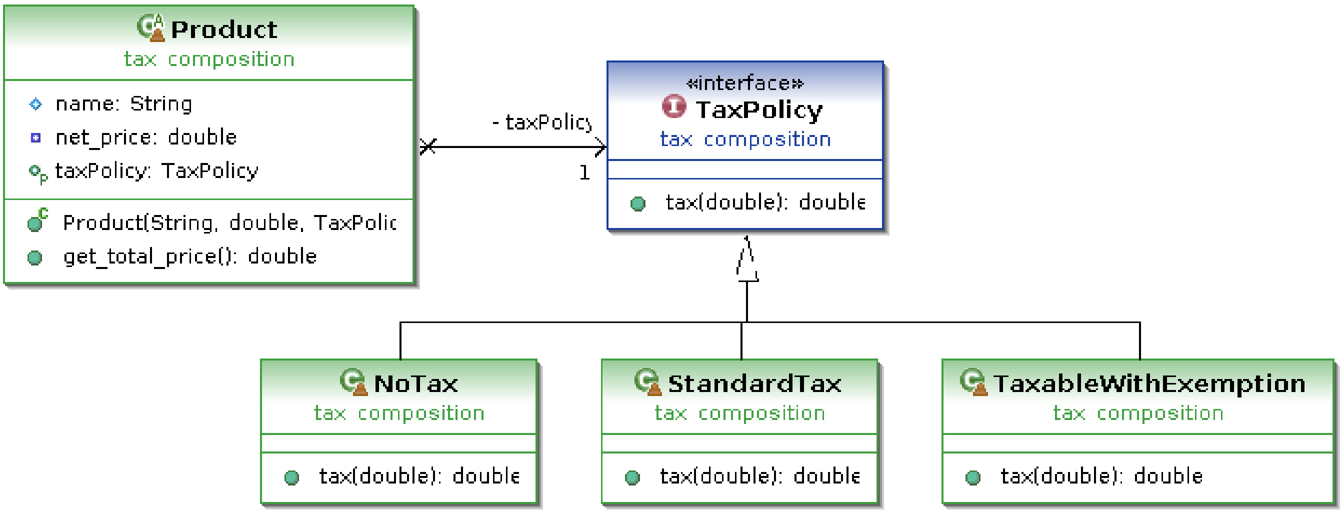
```java
class ProductNonTaxable extends Product {
    @Override
    protected double tax(double price) {
        return 0.0;
    }
}

class ProductTaxableWithExemption extends Product {
    private static final double MIN_PRICE_EXEMPTION = 1000.0;
    private static final double tax_rate_1 = 0.21; // VAT
    private static final double tax_rate_2 = 0.07; // reduced VAT
    @Override
    protected double tax(double price) {
        return (price < MIN_PRICE_EXEMPTION ?
                tax_rate_1*price : tax_rate_2*price);
    }
}
```

Who gets which functionality ?

# “Favor composition over inheritance”

Another way to get it is composition:



# “Favor composition over inheritance”

```java
abstract class Product {
    protected String name;
    private double net_price;
    private TaxPolicy taxPolicy;
    //... setters and getters
    public Product(String str, double netp, TaxPolicy tp) {
        name = str;
        net_price = netp;
        taxPolicy = tp;
    }
    public double get_total_price() { // net price plus tax
        return net_price + taxPolicy.tax(net_price);
    }
}
```

# “Favor composition over inheritance”

```
interface TaxPolicy {
    public double tax(double price);
}
class StandardTax implements TaxPolicy {
    private static final double tax_rate = 0.21; // 21% VAT
    public double tax(double price) {
        return tax_rate*price;
   }
}
class NoTax implements TaxPolicy {
    public double tax(float price) {
        return 0.0;
    }
}
class TaxableWithExemption implements TaxPolicy {
//...
```

Which way is better ?

# “Favor composition over inheritance”

Advantages of composition

- contained objects are accessed by the containing object solely through their interfaces $\Longrightarrow$ “black-box” reuse, since internal details of contained objects are not visible
- fewer implementation dependencies than with inheritance
- each class is *focused* on just one task
- the contained object can be set dynamically at run-time

Problems

- we have more *objects* : each `Product` has a *different* contained `Tax` object

## "Favor composition over inheritance"

Advantages of inheritance

- new implementation is easy, since most of it is inherited
- easy to override or extend the implementation being reused

Disadvantages

- exposes implementation details of superclass to its subclasses, "white-box" reuse
- subclasses may have to be changed if the implementation of the superclass changes
- implementations inherited from superclass can not be changed at run-time: a product instantiated as `ProductNonTaxable` will always be like this.

## "Favor composition over inheritance"

Coad's Rules : use inheritance only when all of the following criteria are satisfied

- a subclass expresses "is a special kind of" and not "is a role played by a"
- an instance of a subclass never needs to become an object of another class
- a subclass extends, rather than overrides or nullifies, the responsibilities of its superclass

## "Favor composition over inheritance"

- `ProductTaxable`, `ProductNonTaxable`, `ProductTaxableWithExemption` "are a special kind of" and not "are a role played by a" `Product` ? No
- a `ProductTaxable` never needs to transmute into an `ProductNonTaxable` ? No, it may depend on tax law changes or buyers nationality
- `ProductTaxable` . . . extend rather than override or nullify `Product` ? No, simply override tax computation

Therefore, better use composition.

## "Favor composition over inheritance"

- `NoTax`, `StandardTax`, `TaxWithExemption` "are a special kind of" `TaxPolicy` ? Yes
- a `NoTax` role never needs to transmute into an `StandardTax` etc. ? Yes
- `NoTax` . . . roles extend `TaxPolicy` rather than override or nullify it ? Yes, they implement `tax` method in the interface

## Summary

You should

- know what's the intent of each principle :
  - information hiding
  - "Don't talk to strangers"
  - DRY
  - SRP
  - LSP
  - OCP
  - "Favor composition over inheritance"
  - "Program to an interface, not an implementation"
- recognize them in the *Snake & Ladders* game
- apply them to the list of exercises and any other object-oriented design problem you have to face

## Next

A number of *design patterns*, almost all based on this principle:

**Encapsulate what varies**

Identify the aspects of your application that vary and separate them from what stays the same.

Take the parts of your design that vary and encapsulate them so that later you can extend or change them without affecting those that don't.

As an example, in the taxable products design we applied the *Strategy* pattern.