Object-oriented design : principles

J.Serrat, L.Gomez, E.Valveny

Advanced Programming, Data Engineering

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. 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 Python,

- by convention, we prefix an attribute or internal method, not intended to be used by client classes, with an underscore character, _
- optionally prefix them with a double underscore, __ to perform `name mangling`.
- add necessary public setters and getters

```
1  class Vehicle:
2      _speed = 0. # in Km/h
3
4      def getSpeed(self):
5          return self._speed
6
7      def setSpeed(self, s):
8          self._speed = s
```

Information hiding

```
10      def action_break(self, seconds, pressure):
11          deceleration =
        self._mpsToKmh(self._compute_deceleration(pressure))
12          while ( (seconds>0) and (self.getSpeed()>0) ):
13              newSpeed = max(0, self.getSpeed() - deceleration)
14              self.setSpeed(newSpeed)
15              delay(1)
16              seconds--
17
18      def _compute_deceleration(self, pressure): # internal method
19          # some equation relating pedal pressure to
20          # speed change in meters per second, each second
21
22      # meters/second to Km/h
23      def _mpsToKmh(self, mps): # internal method
24          return mps*36.0/10.0
```

Information hiding

Why do it so ?

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

```
1  class Vehicle:
2      _speed = 0. # in Km/h
3
4      def setSpeed(self, s):
5          if ( (s>=0.0) and (s<=MAX_SPEED) ):
6              self._speed = s
7          else:
8              raise ValueError('Incorrect␣speed␣value.')
```

Information hiding

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

```
1  class Vehicle:
2      _speed = 0. # in Miles/h
3
4      def setSpeed(self, s):
5          if ( (s>=0.0) and (s<=MAX_SPEED) ):
6              self._speed = self._kmhToMph(s)
7          else:
8              raise ValueError('Incorrect␣speed␣value.')
9
10     def _kmhToMph(self, s):
11         return s*0.62137
```

Information hiding

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

```
1  class Vehicle:
2      _speed = 0. # in Km/h
3
4      def setSpeed(self, s):
5          speed = s
6          self._automatic_change_gears()
7          self._update_wheel_revolutions()
8          self._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"

```
1  class Company:
2      _departments = []
3
4  class Employee:
5      _salary = 0.
6
7      def getSalary(self):
8          return self._salary
9
10 class Department:
11     _manager = Employee()
12
13     def getManager(self):
14         return self._manager
```

"Don't talk to strangers"

Don't :

```
# within Company
for dept in self._departments:
    print( dept.getManager().getSalary() )
    # now Company depends on Employee
```

Do :

```
class Department:
    #...
    def getManagerSalary(self):
        return self.getManager().getSalary()

# within Company
for dept in departments:
    print( 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.

```
1  class Valve:
2      _open = False
3      def open(self):
4          self._open = True
5          # do something
6
7      def close(self):
8          self._open = False
9          # do something
```

DRY: Don't repeat yourself

```
1  class PressureTank:
2      _valve = Valve()
3      #...
4      def releasePressure(self, seconds):
5          self._valve.open()
6          # launch thread so we can return exec. control at once
7          threading.Timer(seconds, self._valve.close).start()
```

DRY: Don't repeat yourself

```
1  class Boiler:
2      _inputValves = []
3      _timeToFill = 10.
4      #...
5      def fillBoiler(self):
6          for valve in self._inputValves:
7              valve.open()
8              seconds = self._timeToFill / len(self._inputValves)
9              threading.Timer(seconds, valve.close).start()
```

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

```
1  class Valve:
2      _open = False
3      def open(self, seconds):
4          self._open = True
5          # do something
6          threading.Timer(seconds, self.close).start()
7
8      def close(self):
9          self._open = False
10         # do something
```

DRY: Don't repeat yourself

```
1  class PressureTank:
2      _valve = Valve()
3      #...
4      def releasePressure(self, seconds):
5          self._valve.open(seconds)
6
7  class Boiler:
8      _inputValves = []
9      _timeToFill = 10.
10     #...
11     def fillBoiler(self):
12         for valve in self._inputValves:
13             seconds = self._timeToFill / len(self._inputValves)
14             valve.open(seconds)
```

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.

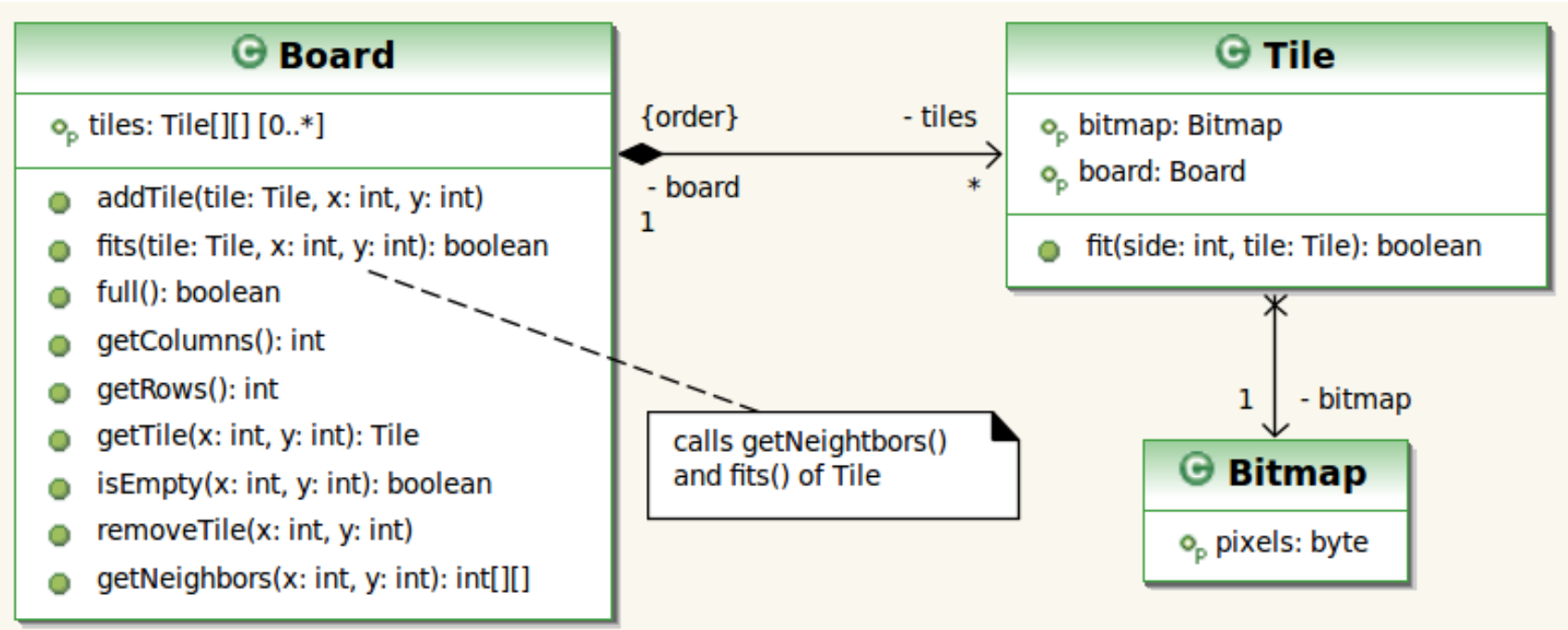
SRP: Single responsibility principle

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



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 ?

```
makeTiles(picture, rows, 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

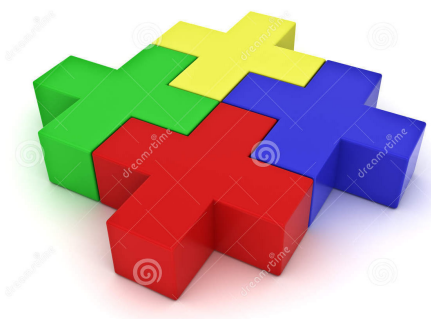
Subtypes must be substitutable for their base types.

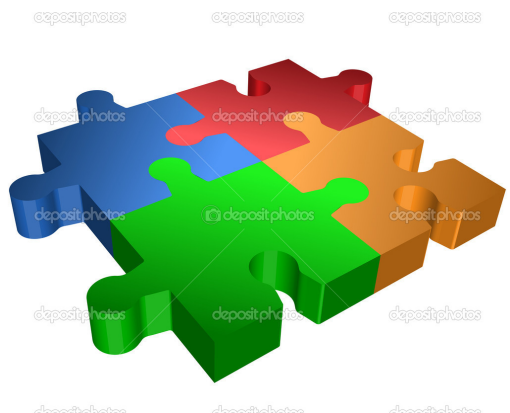
Where an object of the derived class is expected, it can be substituted by an object of the base class.

LSP: Liskov substitution principle

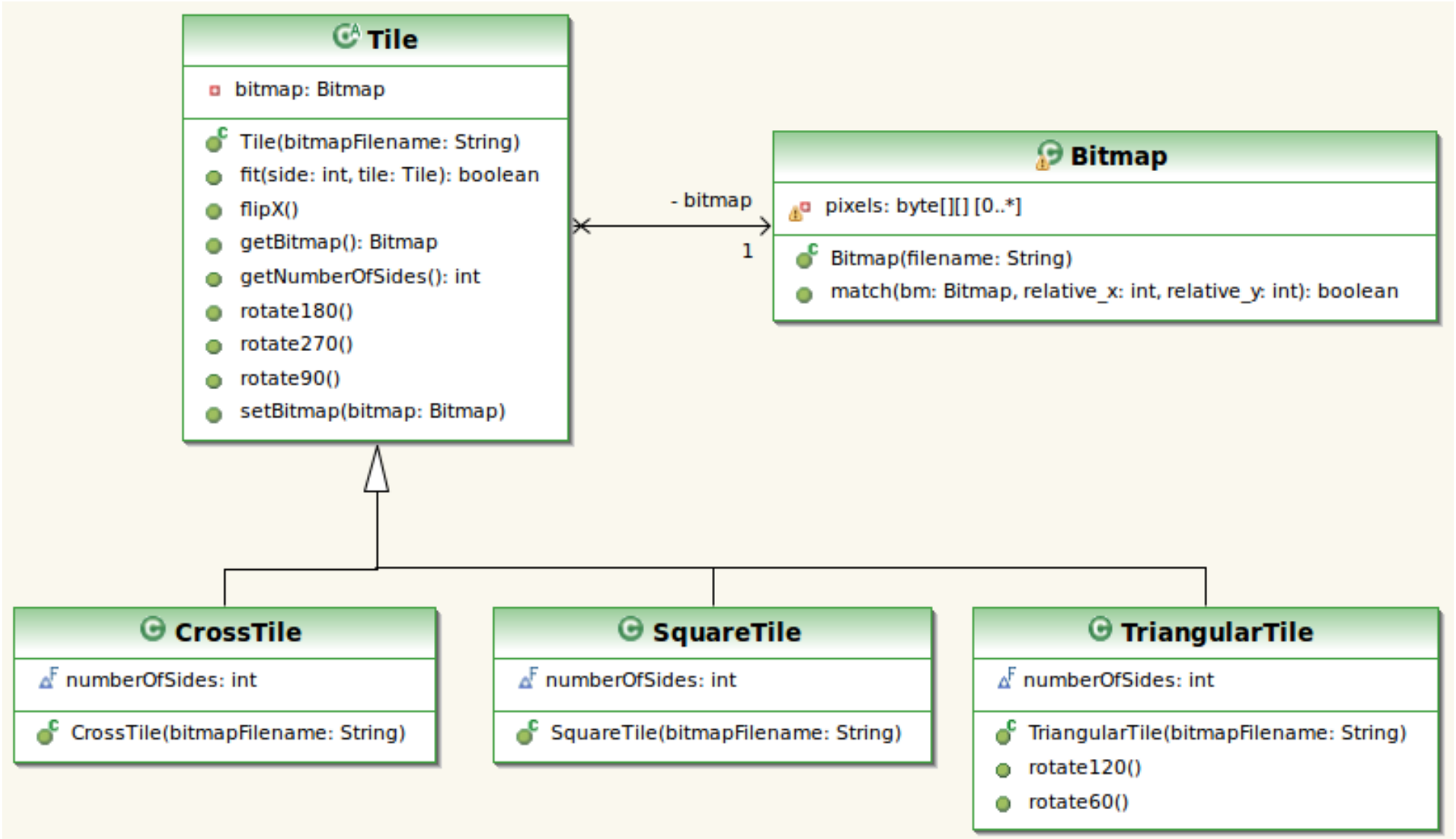
A jigsaw puzzle application

- 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



LSP: Liskov substitution principle

What's wrong here ?

```
1  def tryToAdd1(board, tile, x, y):
2      angle = 0
3      fits = board.fits(tile, x, y)
4      while ((angle<=270) and !fits):
5          # does tile t match neighbor tiles in (x, y), if any ?
6          # which are the neighbor tiles is known by board
7          angle += 90
8          tile.rotate90()
9          fits = board.fits(tile, x, y)
10     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 ?

```
1  def tryToAdd2(board, tile, x, y):
2      angle = 0
3      fits = board.fits(tile, x, y)
4      if isinstance(tile, TriangularTile):
5          while ((angle<=120) and !fits):
6              angle += 60
7              tile.rotate60()
8              fits = board.fits(tile, x, y)
9
10     elif isinstance(tile, CrossTile) or isinstance (tile,
       SquareTile):
11         while ((angle<=270) and !fits):
12             angle += 90
13             tile.rotate90()
14             fits = board.fits(tile, x, y)
15
16     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

```
1  def tryToAdd3(board, tile, x, y):
2      angle = 0
3      fits = board.fits(tile, x, y)
4      while ( (angle <= tile.getMaximumAngle())
5              # 120 for TriangleTile, 270 for Cross, SquareTile
6              and !fits ):
7          angle += tile.getUnitRotationAngle():
8          # 60 for TriangleTile, 90 for Cross, SquareTile
9          tile.rotate();
10         # rotates this unit rotation angle
11         fits = board.fits(tile, x, y)
12
13     return fits;
```

OCP: Open-closed principle

Open-Closed Principle

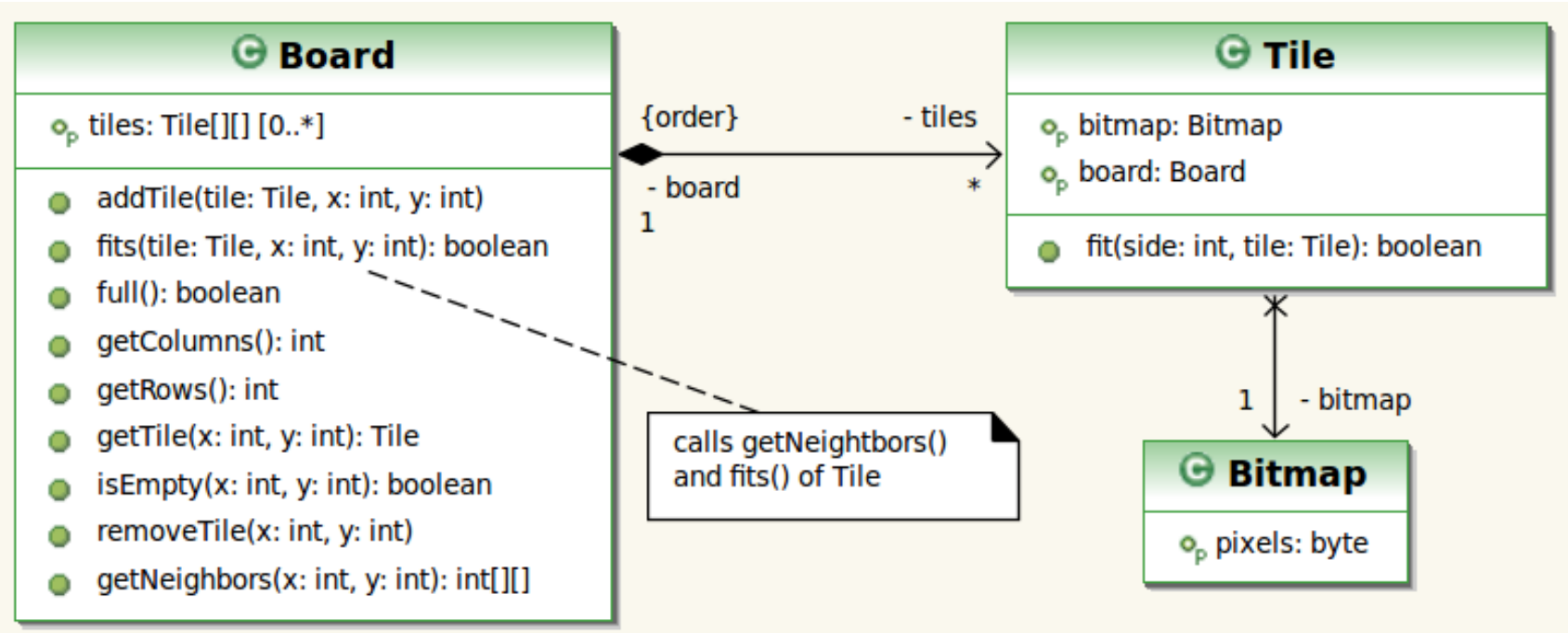
Classes should be open for extension, and closed for modification.

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

How :

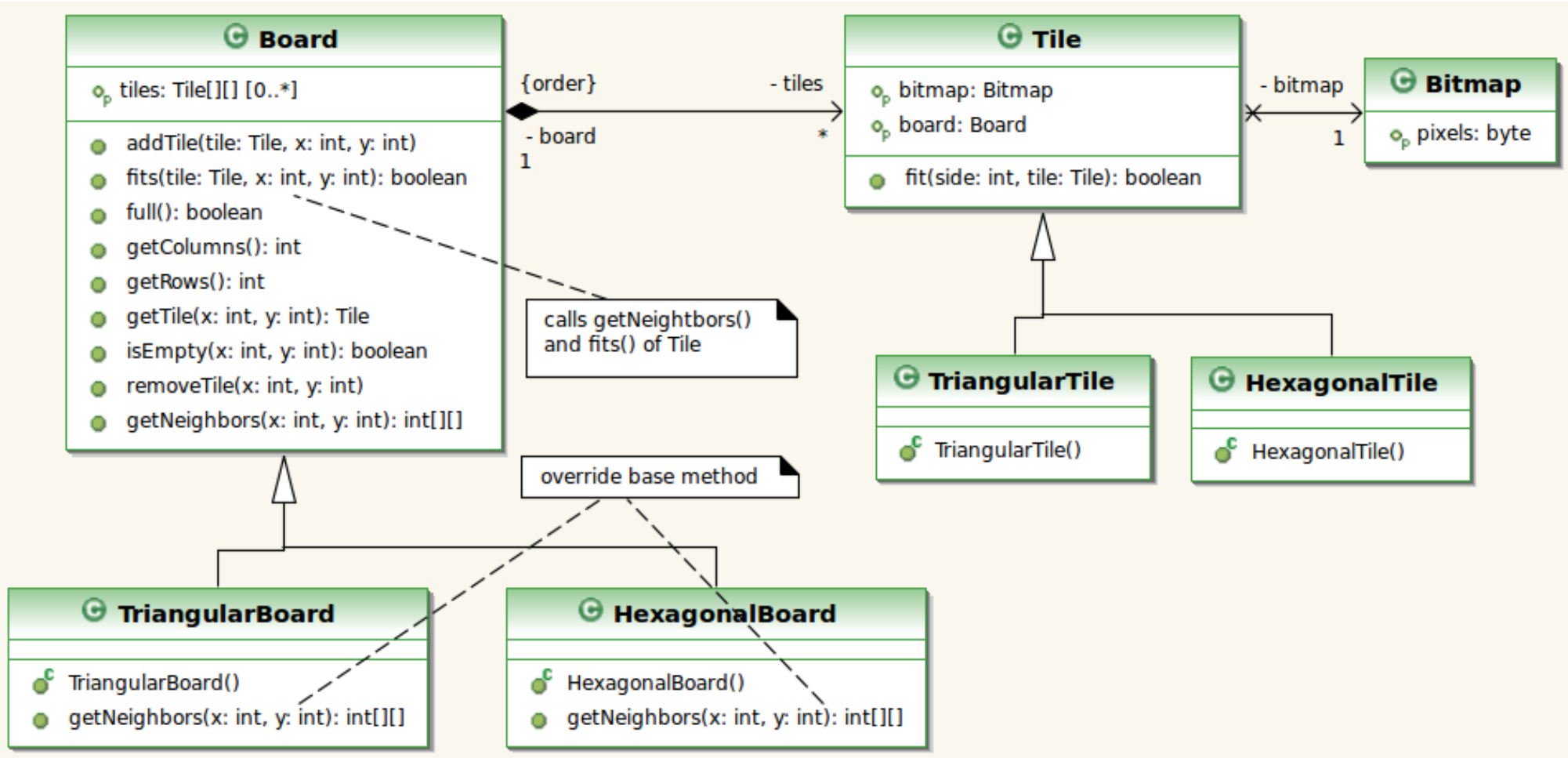
- once implemented a class, do not modify it
- if a change request comes, subclass it and override methods
- or use composition (see later "favor composition")

OCP: Open-closed principle



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

OCP: Open-closed principle



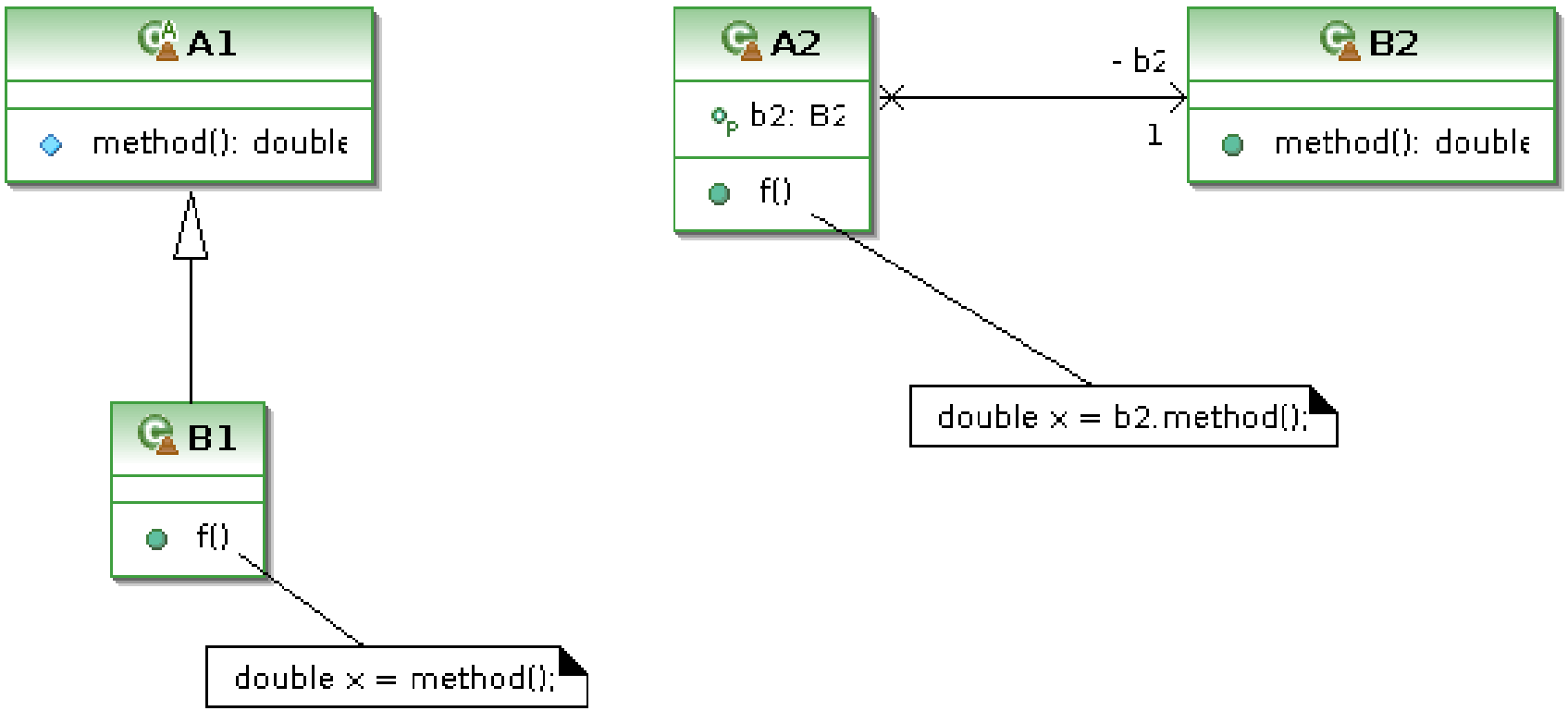
OCP: Open-closed principle

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").

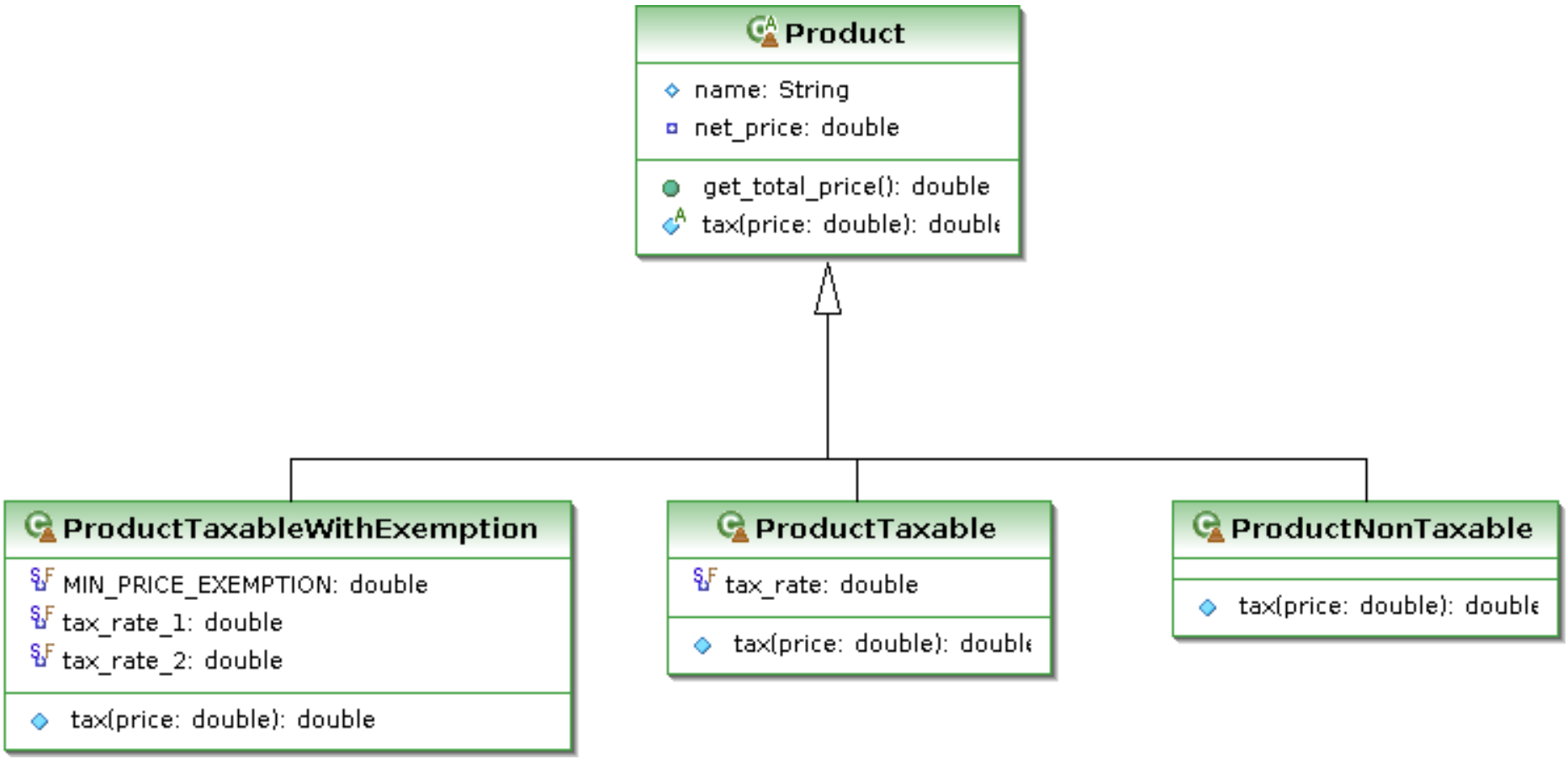
Not always possible. We would better change `Board` and `Tile` to abstract classes, and add `SquareBoard`, `SquareTile`, to increase extendability.

"Favor composition over inheritance"

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

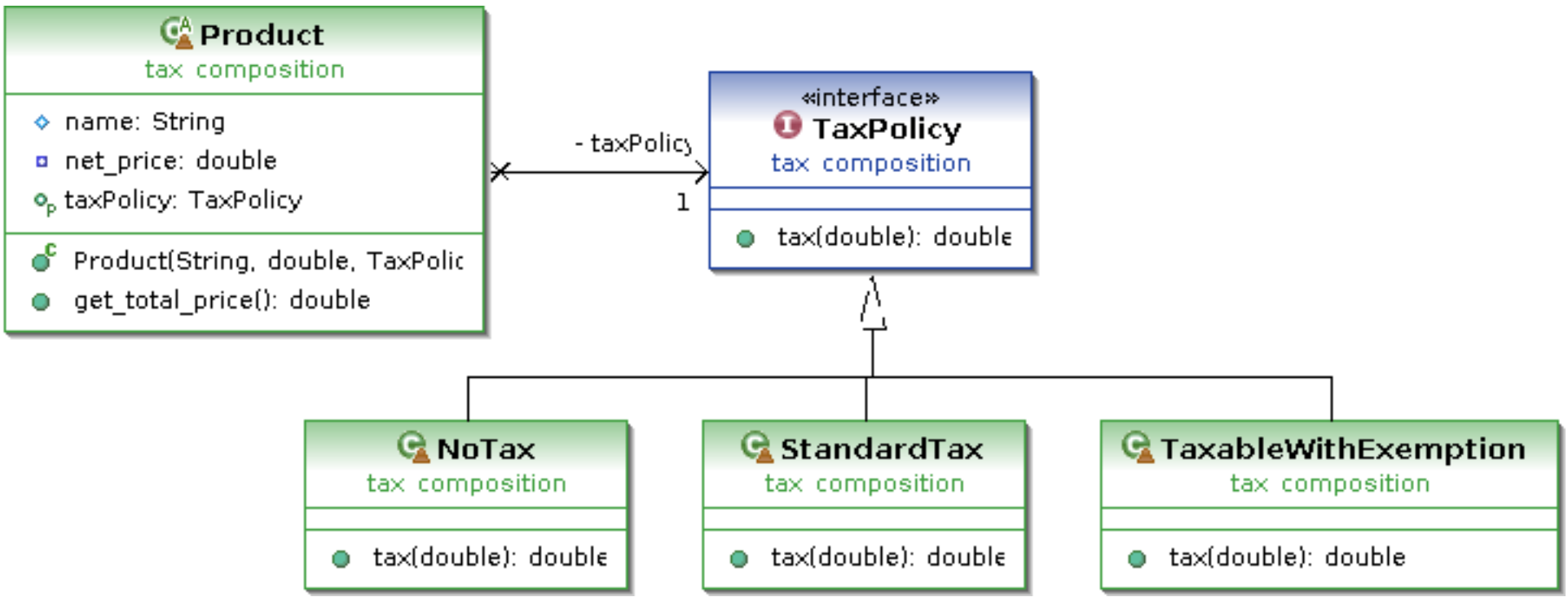


"Favor composition over inheritance"



"Favor composition over inheritance"

Another way to get it is composition:



"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"
- apply them to the list of exercises and any other object-oriented design problem you have to face