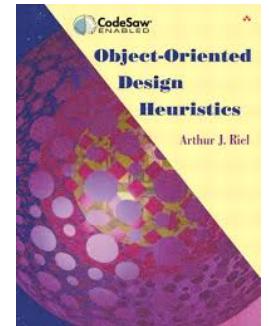
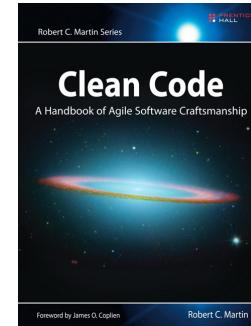
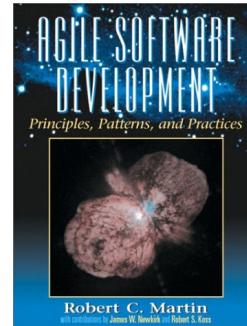


Principles

of object-oriented design



[https://labs.cs.upt.ro/~ooose/pmwiki.php/Resources/
ObjectOrientedDevelopment](https://labs.cs.upt.ro/~ooose/pmwiki.php/Resources/ObjectOrientedDevelopment)

1

Object vs Structured
design

2

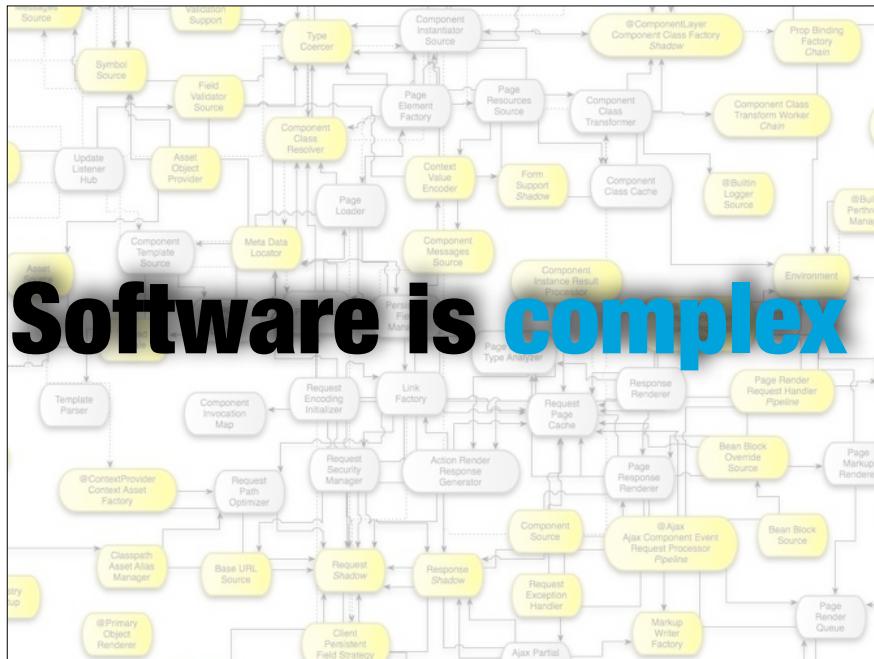
S.O.L.I.D
design principles

1

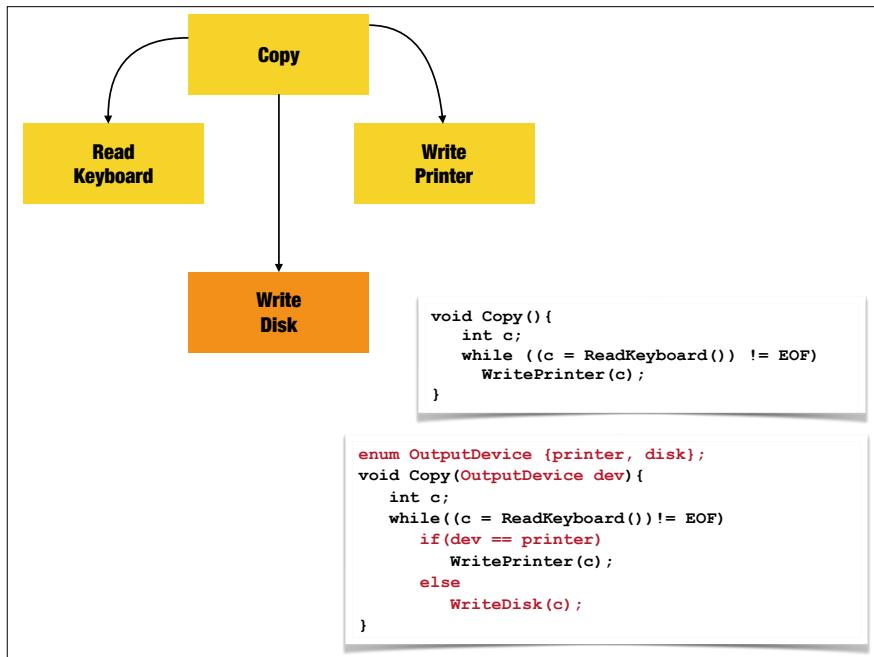
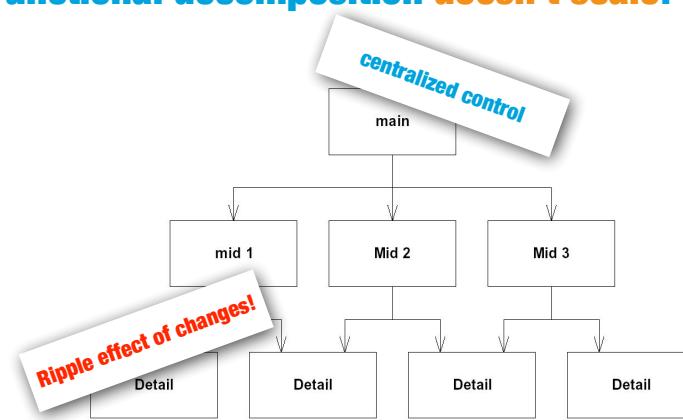
Object vs Structured
design

2

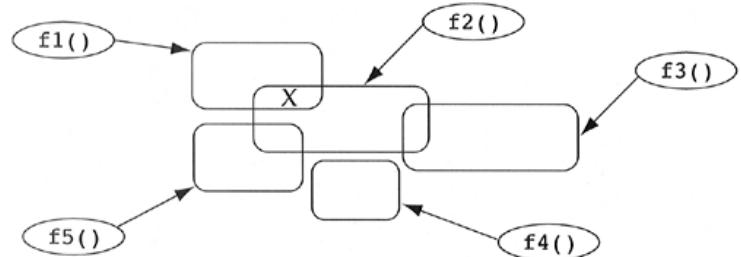
S.O.L.I.D
design principles



Functional decomposition doesn't scale!

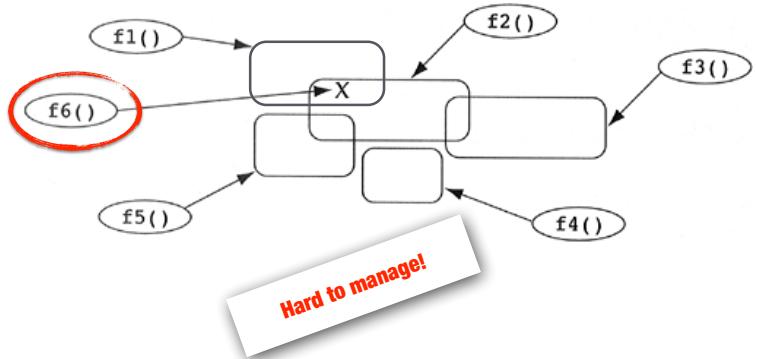


Hang procedures on spaghetti data structures



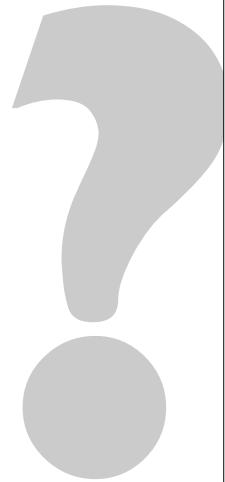
from A. Riel - Object-Oriented Design Heuristics, 1996

Unidirectional relation between code and data



from A. Riel - Object-Oriented Design Heuristics, 1996

How is **object-oriented** different



The power of **encapsulation**



Can you **use** a phone?



Can you **build** a phone?

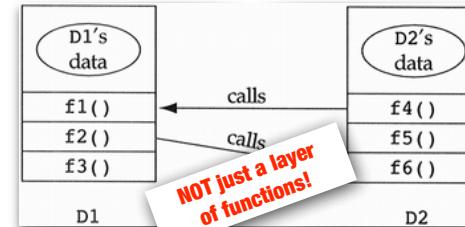


© Todd McLellan TODD MCLELLAN
<http://www.toddmclellan.com/thingscomeapart>

Let's illustrate the point...

Hiding implementation is about abstraction

Expose **interfaces** that allow users to manipulate the **essence of the data**, **without knowing** its implementation.



from A. Riel - Object-Oriented Design Heuristics, 1996

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.1415;

    public double area(Object shape) throws NoSuchShapeException {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

✓ easy to add `perimeter()`

from R.C. Martin - Clean Code, 2008

✗ hard to add `Triangle`

```

interface Shape {
    double area();
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    private final double PI = 3.1415;
    public double area() {
        return PI * radius * radius;
    }
}

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;
    public double area() {
        return height * width;
    }
}

public class Square implements Shape {
    private Point topLeft;
    private double side;
    public double area() {
        return side*side;
    }
}

✓ easy to add Triangle
✗ hard to add perimeter()

```

from R.C. Martin - Clean Code, 2008

It's a trade-off...

Object-Oriented

add new data types rather than new functions

versus

Procedural

add new functions as opposed to data types

Data/Object Anti-Symmetry

Data structures expose data and have no significant behavior.

easy to add **new behaviors** to existing data structures

hard to add **new data structures** to existing functions.

Objects expose behavior and hide data.

easy to add **new kinds of objects** without changing existing behavior

hard to add **new behaviors** to existing objects.

from R.C. Martin - Clean Code, 2008

Hybrid Structures

Functionality



CONFUSING!

Public variables



hard to add new functions.

hard to add new data structures.

from R.C. Martin - Clean Code, 2008

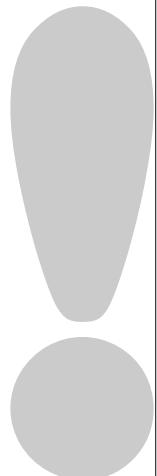
Law of Demeter

Inside of a method M of a class C, you can only **access data**, and **call functions** from the following objects:

- i. **this** and **base-object**
- ii. **data members** of class C [and its ancestors] (in weak form of LoD)
- iii. **parameters** of the method M
- iv. **objects created** within M
- v. **global variables**

Principle of Least Knowledge

An object A can call a method of an object instance B, but object A **cannot “reach through” object B** to access yet another object, to request its services.



Law of Demeter (corollaries)

A module should **not know about the innards** of the objects it manipulates

An object should **not expose its internal structure** through accessors

Law of Demeter example

```
getUserManager().getUser(123).getProfile().isAdmin()
```



```
Manager m = getUserManager();
User u = m.getUser(123);
Profile p = u.getProfile();
p.isAdmin()
```

still breaks
Demeter's law!

from R.C. Martin - Clean Code, 2008

Example

“Context” objects are a strong smell

```

class Mechanic {
    Engine theEngine;
    Mechanic (Context context) {
        this.engine = context.getEngine();
    }
};

```

✗ Mechanic does not care about Context

✗ Can't reuse Mechanic without Context

✗ Mechanic inherits transitively the coupling of Context

✗ The JavaDoc is lying! It **hides the true dependency of Mechanic**

✗ Testing requires to create the entire objects' graph

✗ Testing pain is transitive: how do I test Shop that needs Mechanic ?

<http://misko.hevery.com/2008/07/18/breaking-the-law-of-demeter-is-like-looking-for-a-needle-in-the-haystack/>

Don't dig into collaborators

```

class SalesTaxCalculator {
    TaxTable taxTable;
    SalesTaxCalculator(TaxTable taxTable) { this.taxTable = taxTable; }
    float computeSalesTax(User user, Invoice invoice) {
        Address address = user.getAddress();
        float amount = invoice.getSubTotal();
        return amount * taxTable.getTaxRate(address);
    }
}

```

```

class SalesTaxCalculator {
    TaxTable taxTable;
    SalesTaxCalculator(TaxTable taxTable) { this.taxTable = taxTable; }
    float computeSalesTax(Address address, float amount) {
        return amount * taxTable.getTaxRate(address);
    }
}

```

✗ mixes object lookup with calculation

✗ API is lying: method just needs an amount and an address

✗ reuse is difficult: needs to supply Invoice and User classes

from J.Wolter, R.Ruffer, M.Hevery - Guide: Writing Testable Code, <http://misko.hevery.com/code-reviewers-guide/>



Example #2

service object directly violating
Law of Demeter

Exercise - is this allowed?

```

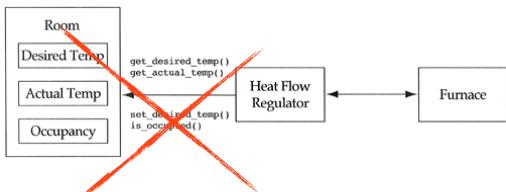
public void showBalance(BankAccount acct) {
    Money amt = acct.getBalance();
    printToScreen(amt.printFormat());
}

```

Exercise - is this allowed?

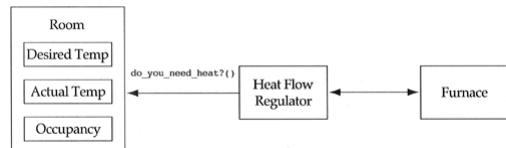
```
public class Colada {  
    private Blender myBlender;  
    private Vector myStuff;  
    public Colada() {  
        myBlender = new Blender();  
        myStuff = new Vector();  
    }  
    private void doSomething() {  
        myBlender.addIngredients(myStuff.elements());  
    }  
}
```

A brilliant question...



When using handles to the internals of a class (getters/setters) ask yourself:

What is it I'm doing with this data, and why doesn't the class do it for me?



from A. Riel - Object-Oriented Design Heuristics, 1996

Example: hot water, oil-fired home heating system.

One portion of the requirement specification discusses the contents of each room in the house as consisting of:
a desired temperature input device
an actual temperature sensor
an occupancy sensor.

A heat flow regulator is responsible for sensing when each room needs heat. If any of the rooms needs heat, the heat flow regulator turns on the furnace, waits for the water to heat up, and then tells the room that heat is available. The method of determining if a room needs heat is to find the difference between the desired and actual temperatures. If there is a person in the room, then heat should be provided as soon as the actual temperature is less than the desired temperature. If there is no person in the room, then the actual temperature is allowed to drop to five degrees less than the desired temperature before heat is supplied.

from A. Riel - Object-Oriented Design Heuristics, 1996

God Classes

capture the **central control mechanism** within an object-oriented design.

performs **most of the work**,

leaving **minor details to a collection of trivial classes**.



Avoiding God Classes

Distribute system **intelligence horizontally as uniformly** as possible.

The top-level classes in a design should **share the work uniformly**.

Be very suspicious of a class whose name contains

Driver, Manager, System, or Subsystem.

Beware of classes that have many **accessor methods**,
as this implies that **related data and behavior** are not kept in **one place**.

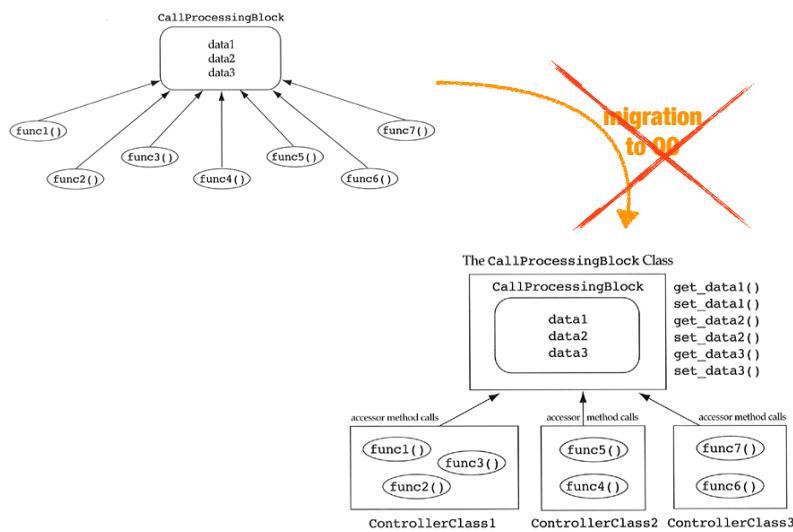
Beware of **non-cohesive classes**, where many methods operate on a proper subset of the data members of a class.

God classes often exhibit much **noncommunicating behavior**.

from A. Riel - Object-Oriented Design Heuristics, 1996



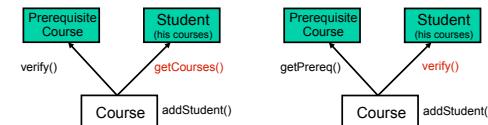
God Class (Data Form)



from A. Riel - Object-Oriented Design Heuristics, 1996

When are accessor methods OK?

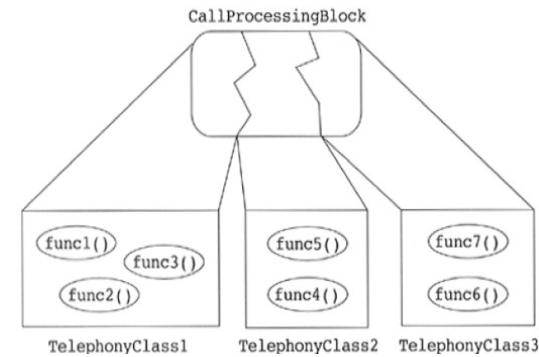
1 Classes defining a collaboration policy



2 Decoupling model from UI

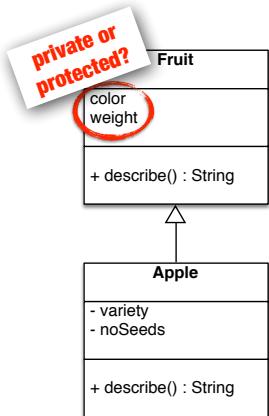


Proper migration strategy



from A. Riel - Object-Oriented Design Heuristics, 1996

What about protected data?



Protected data members are **breaking encapsulation**.

Try to avoid them. If absolutely needed **use a protected getter** method.

Three frequent lies about hierarchies...

Data have a highly volatile nature!

1

These data will never change

Hierarchies grow!

2

This hierarchy is so small

Reuse is about interface!

3

Inheritance is about reuse

The Yo-Yo Problem

... a **long and complicated inheritance graph**

forces the programmer to keep **flipping between many different class**

definitions in order to follow the control flow of the program.

Class hierarchies should not be **very deep**.

The depth should not be more than 6.



from D. Taenzer, David, M. Ganti, S. Podar - Problems in Object-Oriented Software Reuse, ECOOP, 1989



Inheritance is white/glass-box reuse

language supported

easy to use

static bound

mixture of physical data representation



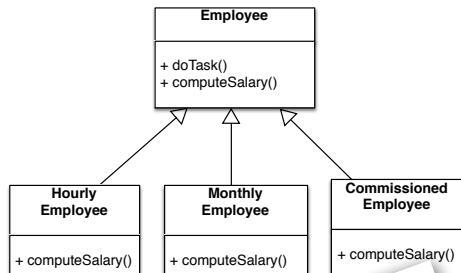
Composition is black box reuse

encapsulation is preserved

dynamic nature

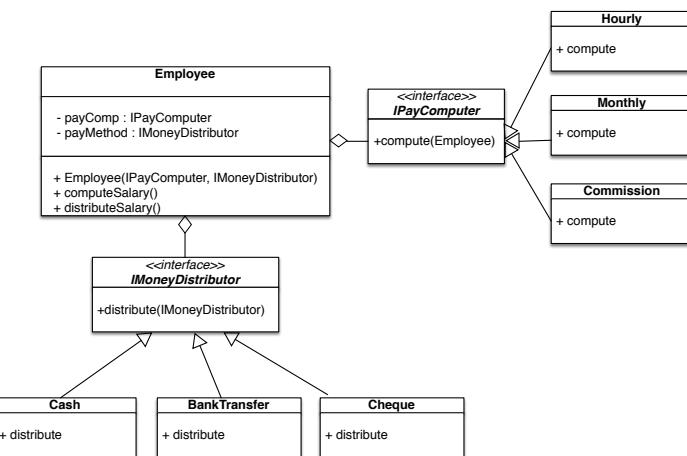


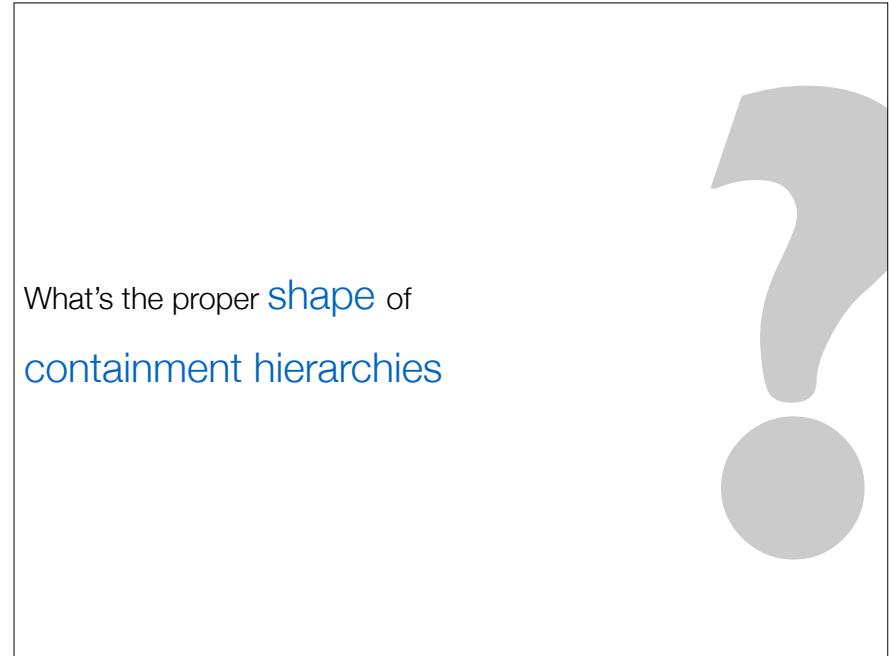
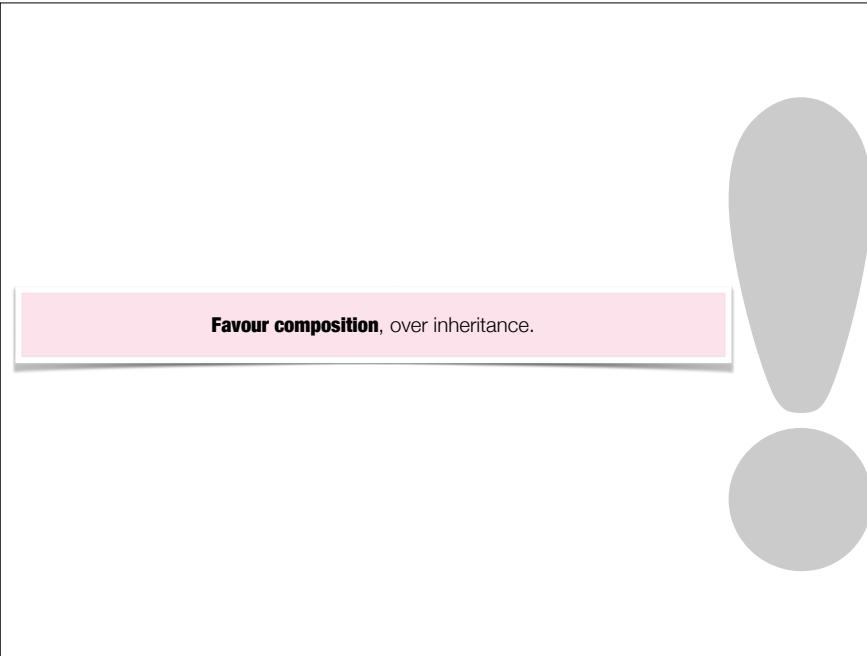
Inheritance is static and rigid



What if salaries could be paid in different ways (bank, cash etc)?

Composition is dynamic and flexible





Which Meal class would you prefer to use?

IT DOESN'T MATTER!

from A. Riel - Object-Oriented Design Heuristics, 1996

Which Meal class would you prefer to maintain?

from A. Riel - Object-Oriented Design Heuristics, 1996

Containment hierarchies should be **deep and narrow**.

