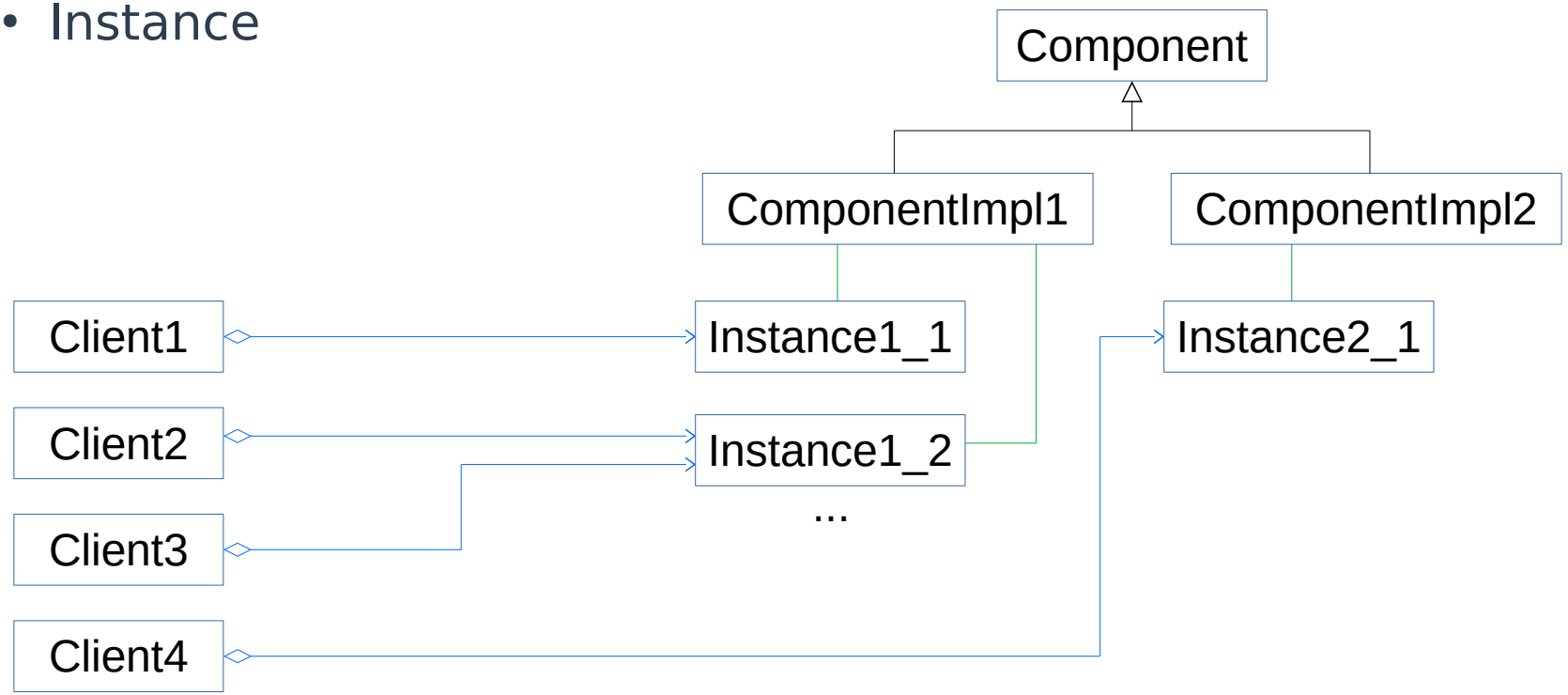
Tools for today programmers

Jean-Luc Delarbre

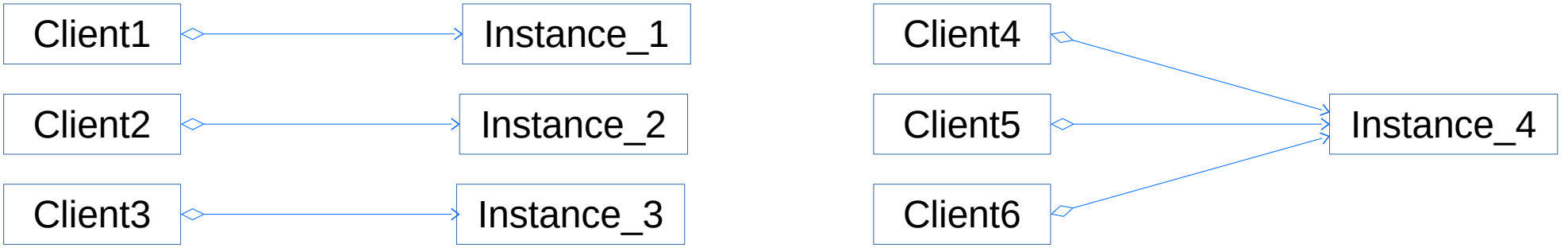
Wiring

- **Operation may be complex**
  - Involves factories, choose:
    - Implementation
    - Instance



Common cases of wiring and dependency injection framework

- **Usage of dependency injection framework**
  - Automatized wiring
  - Less code to write
  - Less cluttered code
  - Clean management of singleton...
- **List:**
  - Google Guice
  - Spring
  - J2EE
  - Google Fruit (C++)
  - ...



Singleton (GOF)

Singleton (GOF)

- **DON’T DO IT**
  - Singleton considered harmful / evil
  - Not testable

Singleton

- **Tests and application wiring are different**
  - Even for singleton
  - Good DI framework helps for this

Compiler: Object mechanism

- **Behind the scene**
  - How objects are made ?
  - How the compiler makes them work ?

Structure of simple object

Object of type ObjectType

Data structure	Memory layout	Methods

`ObjectType {`
`  type1 field1`
`  type2 field2`
`  …`
 	
`  void m1() {…}`
`  OutType m2(…) {…}`
`}`
	`Data segment:`
	
	`field1`
	`field2`
	`...`
	
	`Code segment:`
	
	`m1 code`
	`m2 code`
		`ObjectType object;`
		`object.m1();`
		⇔
		`m1(object);`
		
		`void m1(ObjectType this) {`
		`  // …`
		`  this.field1 = …;`
		`}`
		
		`OutType m2(ObjectType this, …)`
		`{`
		`  // …`
		`  this.field1 = …;`
		`  return …;`
		`}`
		
		`m2(object, params);`
		⇔
		`object.m2(params);`

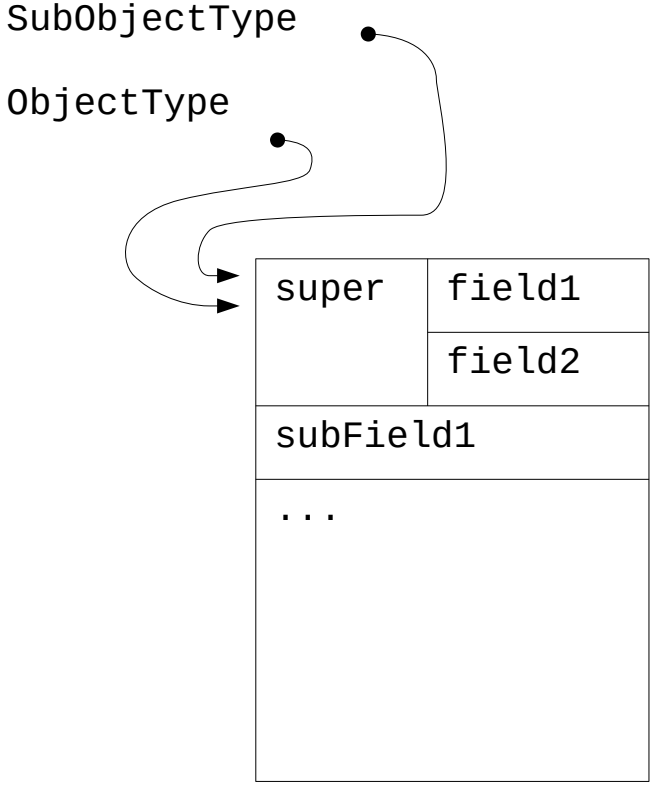
Structure of inherited object

Object of type SubObjectType extends ObjectType

Data structure

```
SubObjectType {
    ObjectType super
    type subField1
}
```

Memory layout



Methods

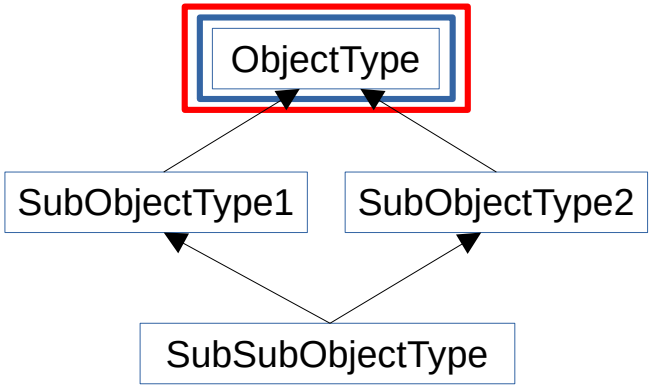
```
T mSub(SubObjectType this) {
    // …
    this.super.field1 = …;
    This.subField1 = …;
}

SubObjectType obj;
mSub(obj); ⇔ obj.mSub();

m1(obj); ⇔ obj.m1();
```

Structure of object with multiple and diamond inheritance

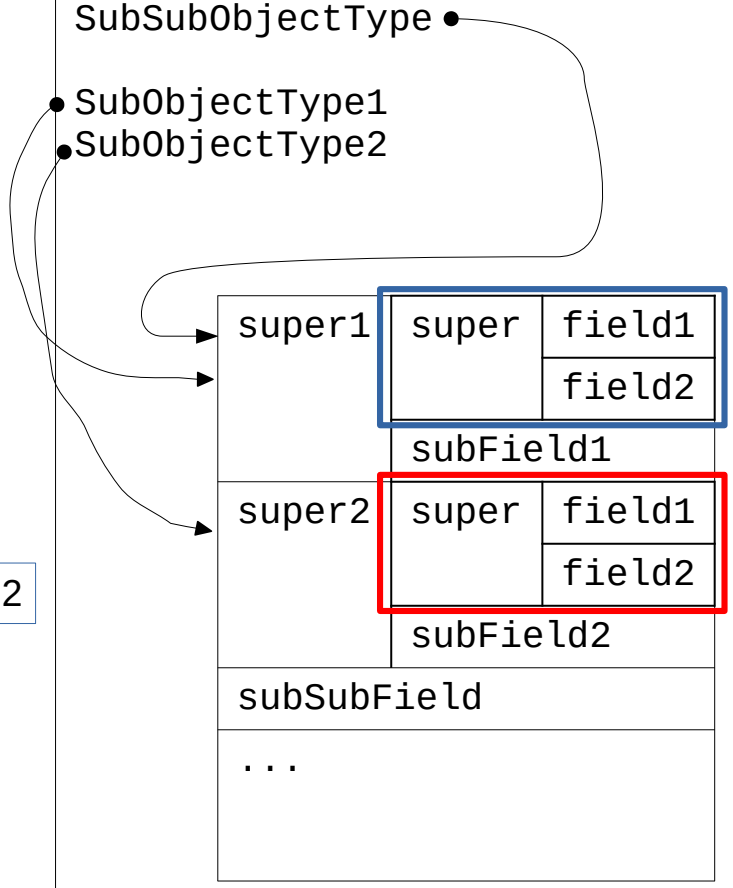
Object of type SubSubObjectType extends SubObjectType1 and SubObjectType2

Data structure

```
SubSubObjectType {
    SubObjectType1 super1
    SubObjectType2 super2
    type subSubField
}
```



Memory layout



Methods

```
T mSubSub(SubSubObjectType this){
    // …
    this.super1.super.field1 = 0;
    this.super2.super.field1 = 5;

    this.super1.super.field1
        !=
    this.super2.super.field1

    this.super1.subField1 = …;
    this.super2.subField2 = …;
    this.subSubField1 = …;
}
```

Diamond inheritance leads to multiple instance for common base class.
Very hard to manage.
No multiple inheritance in Java.

Virtual methods

Object of type ObjectType

Data structure

```
ObjectType {
    …

    void m1() {…}
    …
}
```

Virtual method

Java

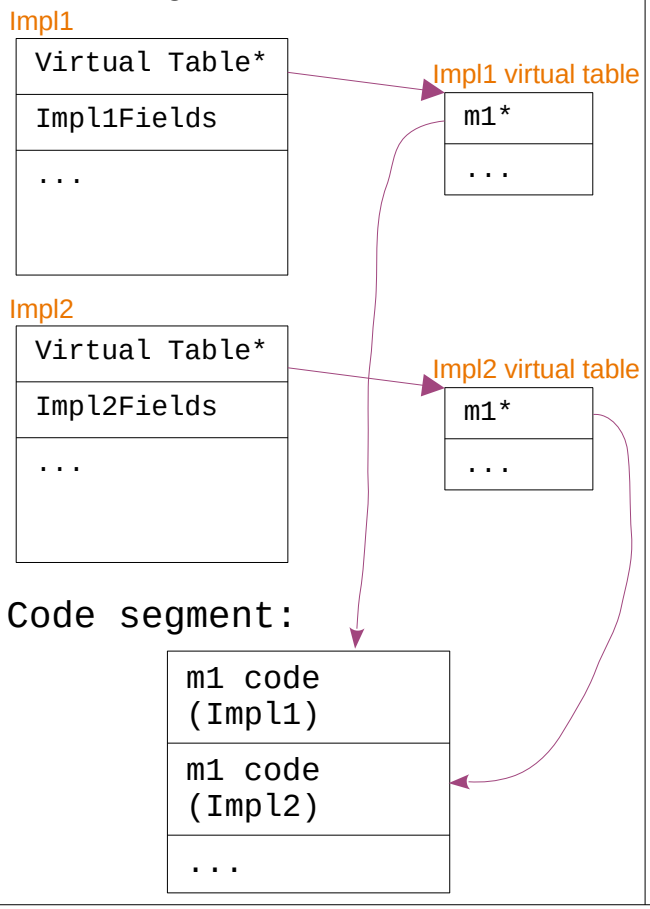
```
interface ObjectType {
    void m1();
}
class Impl1 implements ObjectType {
    // data fields …
    void m1() {…}
}
class Impl2 implements ObjectType {
    // data fields …
    void m1() {…}
}
```

C++

```
Class ObjectType {
    virtual void m1() = 0;
}
Class Impl1 : ObjectType {
    void m1() {…}
} …
```

Memory layout

Data segment:



Methods

```
ObjectType object;
Object = new Imp1();
    or
Object = new Imp2();

object.m1();
                    ⇔
m1(object);
                    ⇔
(*(object.vTable[Ind_m1]))(object);
```

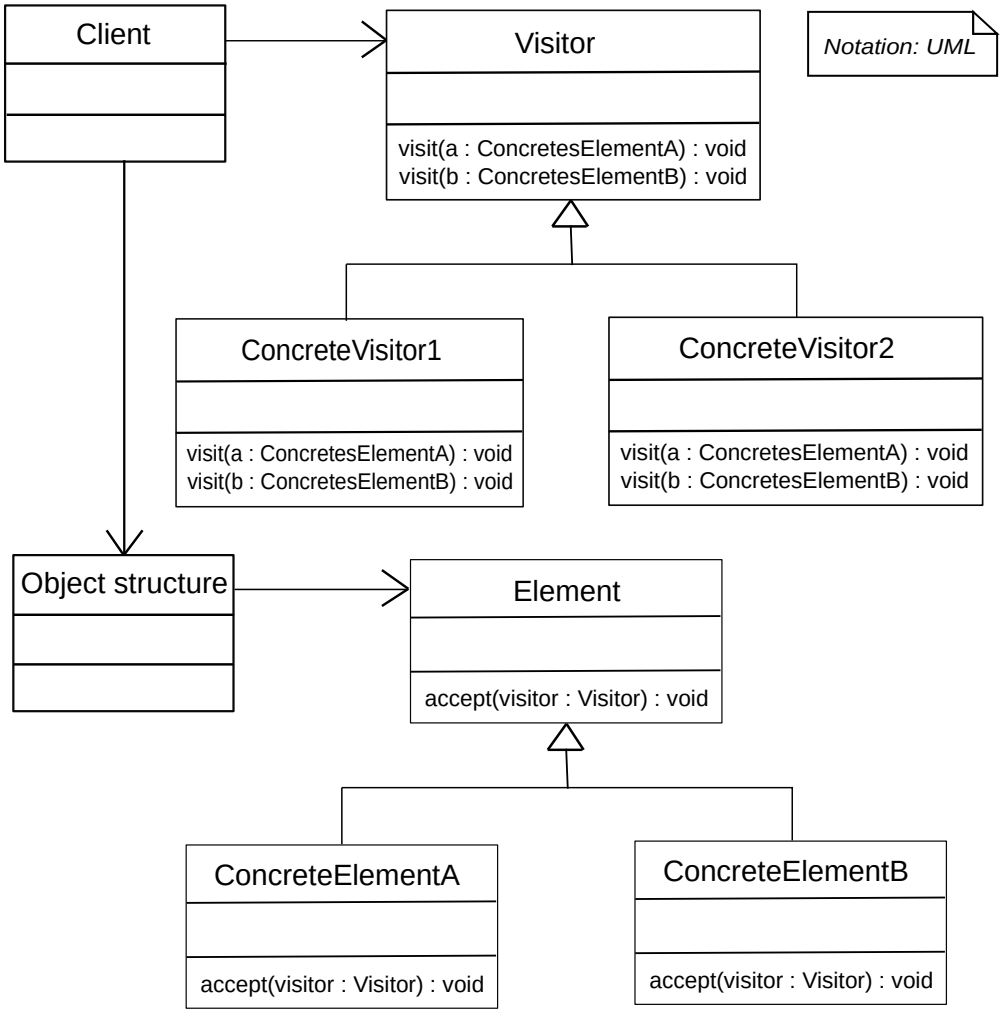
Virtual methods = late binding

Polymorphism:
Abstract object involves a mechanism of dispatching to call appropriate code to execute

Virtual methods cost

- **Virtual methods call is slower**
  - Indirection
  - Pipeline flush
- **Prefer them**
  - Optimization comes last in a development
  - Optimization necessary only if shown by a profiler
- **Cost almost negligible**
  - Reduced by processor optimization (branch prediction)

Visitor pattern



Source: wikipédia

Visitor pattern

- **Behavioral pattern**
- **Separate object structure and algorithms**
  - Add independently implementation with new algorithm
  - Add independently implementation with new structure
  - Open/close principle
- **Implemented using double dispatch**
- **Double and multiple dispatch natively supported by some languages (C#, Groovy, Lisp...)**

Visitor pattern

- **SRP – Separate code that change for different reasons**
  - Visitor code vs Element (data structure) code
- **OCP – Open for extension but closed for modification**
  - Extend element behavior without having to changed them
- **LSP – Subclasses should be substitutable for their base classes**
  - Inheritance satisfies data abstraction (subtyping)
    - Every ConcreteElement and ConcreteVisitor shall be substitutable as Element or Visitor
- **ISP – Many client specific interfaces are better than one general purpose interface**
  - Visitors split interfaces in dedicated extensions and let base element clean
- **DIP – Depend upon abstractions, do not depend upon concretions**
  - Extended polymorphism with multiple dispatch: Client only knows abstraction, visitor mechanism dynamically links to implementations
- **More…**
  - Common closure principle

Next objective

- **Software architecture with Object oriented paradigm**

Reference

- **Google guice documentation:**
  - https://github.com/google/guice/wiki/Motivation
- **Google search: singleton consider evil, harmful**
- **Agile Principles, Patterns, and Practices in C# - Martin C. Robert, Martin Micah**
- **Clean Code, A Handbook of Agile Software Craftmanship - Martin C. Robert**
- **http://blog.cleancoder.com/uncle-bob/2019/06/16/ObjectsAndDataStructures.html**
- **Object programming in C**
  - https://chgi.developpez.com/c/objet/
  - Object Oriented Programming with ANSI-C - Axel-Tobias Schreiner
  - Méthode pour écrire du C Orienté Objet - Cédric Cellier