Université Nice Sophia Antipolis

Université Nice Sophia Antipolis

Advanced Inheritance

Philippe Collet

With elements from a Roger Rousseau's lecture

Master 1 IFI International
2013-2014
http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1314

Agenda

- Subtyping versus factorization
- Study on the possibilities of inheritance

A reminder…

- Inheritance (in the broad sense) in OO languages aims at two objectives to establish relationships of

1. Of compatibility for polymorphic substitutions : **subtyping**

2. Of inclusion to reuse, completely or partially, implementations (data structures and algorithms): **factorization**

With two OO languages

- In Eiffel, integrated in one single inheritance mechanism, supporting multiple inheritance with covariant redefinitions

- In Java, separated among two explicit mechanisms:
  - Compatibility with multiple interface inheritance (implements)
  - Code reuse of one single implementation (extends)

Compatibility: subtyping

- Polymorphism, ie dynamic substitutions of objects

  var v : T;

  v= **new** T1 ; v = **new** T2; *// T1 and T2 subtypes of T*

- Multiple implementation of the same **interface**, with possibility for dynamic loading of classes
  - Typing guarantees the service existence
  - Lookup chooses the implementation that is best suited
- Constraints for subtyping: a subtype of A must
  - Provide **at least** all services of A (same methods and attributes, following compatibility rules)
  - Add new services

Subtyping: inversion principle

Imperative programming: if the data structure evolves,

- *All functions must be modified,*
- *Or discrimination must be done explicitly*

```
void f1(reservation r){
 switch (r)
case r1 : action 1.1
case r2 : action 1.2
…
case rn : action 1.n
}
```

```
void f2(reservation r){
 switch (r)
   case r1 : action 2.1
   case r2 : action 2.2
   …
   case rn : action 2.n
 }
```

```
void fm(reservation r){
 switch (r)
case r1 : action m.1
case r2 : action m.2
…
case rn : action m.n
}
```

Subtyping: inversion principle (2)

- The inversion principle [Meyer 88] says:

> Instead of testing data types in functions,
>
> Put functions inside your data…

- If a lot of functions have the same type of parameters:
  - f1(State s…), f2(State s…), … fn(State s…)

- i.e., transform functions in class methods, with as much classes as situations:
  - Class State …
    void f1() …
    void fn() …

Subtyping: inversion principle (3)

- Of course, there are always n*m algorithms to write…

- But explicit discriminations (**if**, **switch**, **case**…) are replaced by implicit discriminations made by the dynamic binding *(lookup)*

➨ Code already developed remains **stable**, even if new cases are added, handled by new classes

Subtyping: inversion principle (4)

```
class R1 extends R{
 void f1() { action 1.1 }
 void f2() { action 1.2 }
 …
 void fn() { action 1.n }
}

class Rm extends R{
 void f1() { action m.1 }
 void f2() { action m.2 }
 …
 void fn() { action n.n }
}
```

```
class R2 extends R {
 void f1(){ action 2.1 }
 void f2(){ action 2.2 }
…
 void fn(){ action 2.n }
  }

  void usage(R: r, …){
  r.f1() ;
  r.f2() ;
   …
  r.fn();
}
```

Code reuse: factorization

- Maintains unicity of parts to be included several times
- Propagates automatically changes in factorized parts to all entities that reuse them
- Reduces software costs (time, money)
- Reduces memory footprint (code more compact)

Example :

Class `Vector<Monoid>` **extends** `ArrayList<Monoid>` …

All services of ArrayList, present and future, can be automatically reused in Vector

- Only constraints : unicity (no subsitution needed a priori)

Inheritance: an adaptive technique

- Both inheritance relationships, subtyping and factorization, aims at mastering evolution in an **adaptive** way:
  - Possibility to choose what is reused, to modify names or algorithms (redefinitions), visibility, abstraction...
  - While preserving stability and unicity zones
  - With guarantess of typing (static and/or dynamic)
- So one single relationship?
  - Eiffel and C++ : yes
  - Java : no

Unicity of inheritance relationships

- Possible pros:
  - One single concept: class, concrete or abstract
  - Multiple inheritance
  - Covariance in Eiffel, no-variance in C++
  - Possibilities of integration with genericity

- Possible cons:
  - Confusion in the intentions
  - Multiple inheritance leads to conflicts of interest that must be solved by hand
  - Inefficiency during lookup ? NO
  - Static typing problem due to covariance ? YES but can be easily mitigated

Separation of inheritance relationships (Java)

- Possible pros:
  - A construction for each intention: **class** and **interface**
  - No conflict of interest? Not really
  - Efficiency gain???
- Possible cons:
  - Redundancy of concepts (interface and abstract class)
  - Simple inheritance of classes, multiple for interfaces
  - The two relationships are actually both subtyping…

Simple or multiple inheritance?

- Multiple inheritance was said to be inefficient but :
- A static, complete system can be optimized:
  - lookup in constant time
  - Useless attributes and methods removed
  - Inlining when dynamic binding is useless
- Multiple inheritance is said to be too complex:
  - Not for library design
  - But certainly for the average application developer

Separation or unification of relationships?

- Unification can lead to bad usage:
  - A plane is not an inheritance of wings and fuselage, but a real aggregation of parts
  - There is no interest in attaching by polymorphism a plane to a variable of type wing!
- Separation (class and interface in Java) is clearer to handle subtyping with interface as often as possible, but the **extends** link is weak with single inheritance only

Conclusions on inheritance

- Inheritance is powerful to reuse simple structures and for subtyping
- But it is not well suited to reuse complex software architectures:
  - With several interfaces ➔ components
  - With architectural canvas ➔
    - Design patterns,
    - Service oriented architectures, component frameworks…