Université Nice Sophia Antipolis

Université Nice Sophia Antipolis

Genericity

Philippe Collet

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

Agenda

- Introduction
- Principles of parameterization
- Principles of genericity
- Genericity in OO languages
- Static and dynamic genericity
- Generics in Java 5

Université Nice Sophia Antipolis

On software evolution

- Mastering evolution is one of the major challenges of software engineering: *maintenance cost, qualities deterioration, oversight, bugs…*
- **Mastering evolution** consists in:
  - Introducing needed modifications
  - Only needed ones
  - Wherever this is needed
  - Without impairing initial qualities
  - At the lower cost…

Difficult Problem

especially if the software is large and complex

On software evolution (cont'd)

- A lot of software techniques aim at facilitating evolution management:
  - Avoid duplication (uniqueness in definition)
  - Hide what is not useful (encapsulation)
  - Locate and explain dependencies

- Two families of techniques to manage evolution:
  - **Planned techniques**
    - Anticipate possible evolutions, to make them easier and safer
  - **Adaptive techniques**
    - « lazy » change handling, one at a time, but in the most automatic way

Planned vs. Adaptive

- Planned
  - Parameterization, factorization, genericity, formal models, model-driven engineering
  - Raising the level of abstraction, overhead of creation should be compensated by ROI
- Adaptive
  - Rewriting, inheritance, separation of concerns, dynamically reconfigurable components

- Each technique borrows a small part of characteristics from the other:
  - There is some planning in inheritance
  - There is some adaptiveness in genericity
- Both techniques complement themselves and can combine with eachother

Principles of parameterization

- Define a formal parameter of an entity, abstract and general, which displays what is necessary and sufficient
- Substitute in an automatic way formal by effective parameters, with no impact on the use of the formal ones

- Verify that:
  - Formal parameters are well used (typing)
  - Effective parameters conform to formal ones
  - Usages of generated entities are coherent

Parameterization on values

- A value is a simple concept : value + type
- The formal parameter is typed and **hides its potential value**
- One can use it and verify its type
- Effective values can be automatically substituted on all occurrences in **any document**
  - Simple substitution, macrogeneration, cpp, sed
- Checking *intensity depends on available knowledge:*
  - Nothing: macrogeneration
  - Classic typing: static checking at compilation time, or at binding and run times
  - Assertions, formal specifications: dynamic checking, proofs…

Principles of genericity

- A lot of things are generic: *medicines…*
- In programming, genericity is a form of type genericity
- In languages:
  - Imperative (pioneer): *CLU, LPG, Euclide, Ada*
  - OO: *Eiffel, then C++, C#, Java*
- This is always a constant search for tradeoffs between:
  - Flexibility in derivations
  - Checking safety
  - Cost at compilation and run times
  - Simplicity, readability

Université Nice Sophia Antipolis

Generic OO languages

- Eiffel (pioneer)
  - Powerful and complex multiple inheritance
  - Primitive types as expanded objects (no limitations for genericity)
  - No genericity information at runtime (a simple technique of static typing)
  - Genericity very readable and very simple
  - Constrained genericity through abstract classes
  - Automatic constraints with **like** object / **like** current
  - No introspection on effective types

Generic OO languages (2)

- C++
  - Macro-generation at compilation or linking times
  - Checking done through the linkers => error messages are *strange*
  - No constrained genericity (implicit genericity in signatures)
  - No guarantee that the introspection of effective types will be possible at runtime

Université Nice Sophia Antipolis

Generic OO languages (3)

- C#
  - Instantiation of generic types at *runtime*
  - Genericity constrainted by class and interface
  - Type checking by the compiler and possible introspection at runtime
- Java
  - Ten years to put chunks of genericity inside…
  - Compatibility at the code level between generic/non-generic software
  - Very sophisticated, but complex
  - Inconsistencies and performance loss due to compapility constraints (cast, type erasure…)
  - Some problems with primitive types and arrays

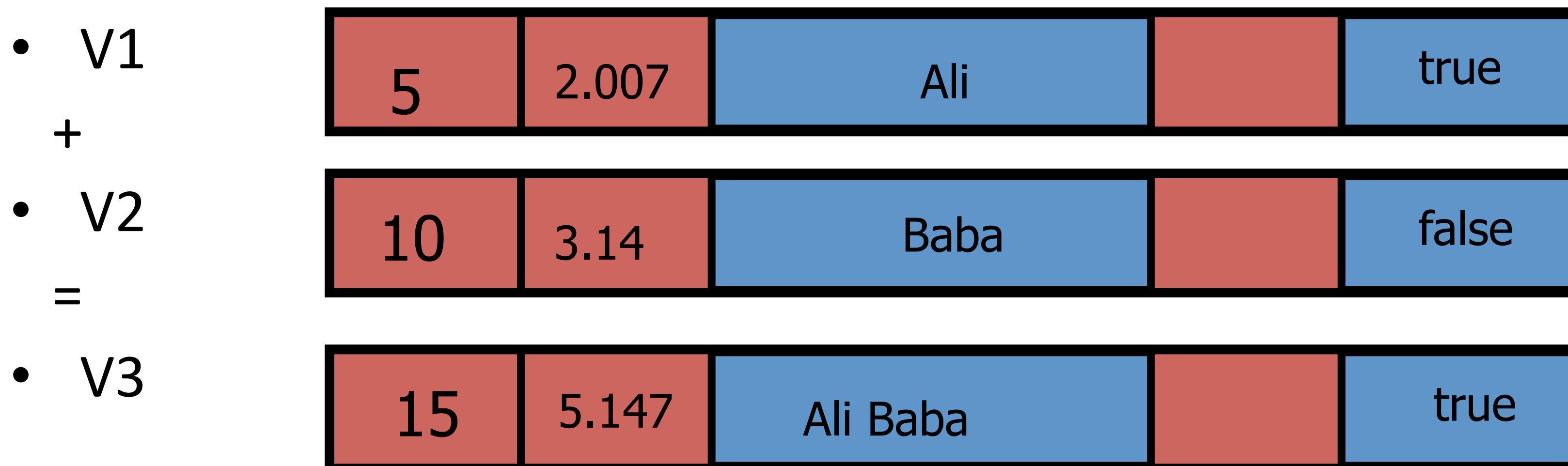
Principles of dynamic genericity

- An object carries information on its properties (through reflexivity) and on its co-instances' properties

- The technique is similar to the **prototype** design pattern
- The formal parameter is an attribute of type T
- All properties of T can be used in the model
- Substitution is dynamic, by a simple (polymorphic) assignment
- Possible checking: by dynamic reflexivity, very flexible but costly…

Example: polymorphic vector

- Addition of two polymorphic vectors:



- Advantages:
  - Get more flexibility than with static typing (with which all elements have the same type)
  - No concession on rigorous typing
  - While doing dynamic checking

Solution #1 : No static checking

- All elements are of type Object
- One checks by introspection that two elements of the same index are compatible for the addition
- The method « plus » of the first element is used to add the second one

- This is possible in all OO languages with introspection

Université Nice Sophia Antipolis

Solution #2: mixed checking, static and dynamic

- The necessary and sufficient static typing is added to the previous solution: *All elements are of type Monoid*

```
class PolymorphicVector <Monoid>
        extends ArrayList<Monoid> {
...

}
```

Solution #2: mixed checking, static and dynamic (cont'd)

- One can statically constrain:

    PolymorphicVector <Number>

- One can dynamically constrain:

```
class DynamicMonoVector
        extends ArrayList<Monoid> implements Monoid {
        Monoid prototype ; // fournit le +
        Void DynamicMonoVector(Monoid  type){
            super(); prototype = type;
}
// Usage
DynamicMonoVector pvi = new DynamicMonoVector(new Integer(0));
```

Other possibilities

- By combining static and dynamic approaches, one has various possibilities to choose:
  - The degree of polymorphism: all of the same type, same sur-type, same type two by two…
  - When the constraint is defined and checked : at compile or run times

Université Nice Sophia Antipolis

Generics in Java 5

The Collection Class :

```
Public interface Collection<E> extends Iterable<E> {
…
int size();
boolean isEmpty();
boolean contains(Object o);
Iterator<E> iterator();
boolean add(E o);
boolean remove(Object o);
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
void clear();
boolean equals(Object o);
int hashCode();
}
```

Java 5: generic parameter

- Declaration of a formal parameter that will be used throughout the class
  - Use the notation <T> just after the class name
  - Use T in the rest of the code
  - Convention: use a capital letter for the parameter id
  - T will be replaced by a given class at instantiation time
- Genericity can be limiter to a specific method:
  - Use the notation <T> just before the method signature
  - Use T in the rest of the method code
  - T will be **infered** from the method argument types

Kinds of generic declarations

- Novariance = Covariance ∩ Contravariance : only the specified class is accepted
  - <A> ➨ a class A
- Bivariance = Covariance ∪ Contravariance : All classes are accepted
  - <?> joker/unknown ➨ any class
- Covariance (all sub-classes are acceptable)
  - <A extends B> any class A that specializes B (extends, implements)
  - <? extends B> any class that specializes B (extends, implements)
- Contravariance (all **super-classes** are acceptable)
  - <? super B> any class that generalizes B

Illustrations

```
// Novariance
List <Integer> l = new ArrayList <Integer>();
// compilation error: default is novariance
// List <Number> l1 = l;

// possible : covariance
List <? extends Number> l1 = l;

// bivariance : typing is lost
List <?> l2 = new ArrayList <Object>();

// contravariance
List <? super Number> l3 = new ArrayList <Object>();
for (Number n : l1) { // compilation error: l2 could have been a
   list of Double!
   // l2.add(n);
   l3.add(n); // OK
}
```

Illustrations (2)

```
// Covariance and contravariance

public static <T> void copy(Collection <? extends T> src,
                Collection <? super T> dest) {
  for(T t : src) { dest.add(t); }
}

List <Integer> l1 = new ArrayList <Integer>();
  for(int i=0; i<10; i++) { l1.add(i); }

List <Number> l2 = new ArrayList <Number>();

copy(l1,l2);
copy (l2,l1); // Compilation error
// Number(s) cannot be put in a list of Integer
```

Université Nice Sophia Antipolis

Survival guide: PECS principle

- **PECS: Producer extends, Consumer super**
  – Use Foo<? extends T> for a *producer* of T
  – Use Foo<? super T> for a *consumer* of T
  – Only applicable to method parameters
  – No joker in return types

PECS : application

```
public static <T> void copy(Collection<T> src,
                            Collection<T> dest)
```

- In copy:
  - src provides elements of type T
  - dest consumes elements of type T

```
public static <T> void copy(
               Collection <? extends T> src,
               Collection <? super T> dest)
```

PECS : application (2)

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

- In union:
  - s1 and s2 are producers of elements E !

```
public static <E> Set<E> union(Set<? extends E> s1,
   Set<? extends E> s2)
```

Type erasure

- The bytecode *understood* by a JVM v5 does not contain any trace of genericity due to the constraint of backward compatibility
- A Java 5+ compiler:
  - Replace parameters by Object (case of novariance or bivariance) or by the bounded type (co/contra-variance)
  - Create methods with compatible signatures for the other cases
  - The problem of return types is solved by the covariance

Université Nice Sophia Antipolis

Conclusions on genericity

- Genericity allows for factorization and reuse of knowledge and know-how with guaranteed quality
- Can everything be made generic ? NO!
- The reuse of *architectures or canvas* to solve a general problem cannot be, most often, described by a generic unit.
- *Solutions:*
  - Design patterns
  - Frameworks
  - Domain Specific Languages
  - Dedicated systems (DB, IDE…)
  - Software Product Lines