Conformance

Object-Oriented Programming

236703

Winter 2015-6
What’s Conformance?

- **Overriding**: replace method body in sub-class
- **Polymorphism**: subclass is usable wherever superclass is usable
- **Dynamic Binding**: consequence of overriding + polymorphism
  - Select right method body

**Conformance**: overriding method and overridden method should be equivalent
Conformance and Overriding

• Ideally, an overriding method should be “semantically compatible” with the overridden one
  • All versions of “draw” should draw the object
  • Not well-defined
  • Impossible to guarantee!

• We must content ourselves with conformance in the following aspects:
  • **Signature**: input arguments, output arguments, input-output arguments, return type, exceptions
  • **Access**
  • **Contract**: pre- and post-conditions, invariants
“Same or better” Principle
Semantics vs. Syntax

**Same or better:** An overriding method should have at least the same functionality as the overridden method.

Namely:

Overriding should not pose surprises to client

---

**The Liskov Substitution Principle:**
Functions that use pointers or references to base classes must be able to use objects of derived classes* - *without knowing it!*

* Given **subtyping** (Derived *is a* Base, not just Derive : Base, AKA **subclassing**)
“Same or better” Principle
Semantics vs. Syntax

class Rectangle {
    virtual void setW(int);
    virtual void setH(int);
}
class Square : public Rectangle {
    void setW(int) override;
    void setH(int) override;
}

void foo(Rectangle* r) {
    r->setW(20);
    r->setH(10);
}

Can Square replace a Rectangle?
LSP doesn’t apply! Remember:

Overriding should not pose surprises to client
“Same or better” Principle
Semantics vs. Syntax

Ideally, an overriding method should be “semantically compatible” with the overridden one.

**Realistically:** All that can be checked is static typing. The overriding method should be compatible with the overridden one (and possibly allow more – we’ll discuss the options soon).

The overriding method should have the same or more “calling patterns” as the overridden one.

Can semantics be determined in the general case?
Ask Turing…
Signature Conformance

Three alternatives:

- **No-variance**: The type in signature cannot be changed.

```cpp
class Base {
    virtual void f(Base);
};

class Derived : public Base {
    void f(Base) override;
};
```

We start the discussion with arguments, but variance is a general term.

To override, argument type must not be changed.
Signature Conformance

Three alternatives:

- **No-variance**: no change
- **Co-variance**: *Change of type in the signature is in the same direction as that of the inheritance.*

```cpp
class Base {
    virtual void f(Base);  
};
class Derived : public Base {
    void f(Derived) override;
};
```

Note: this is not legal C++ code

To override, argument type can be a subtype
Signature Conformance

Three alternatives:

• **No-variance**: *no change*
• **Co-variance**: *change downwards*
• **Contra-variance**: *Change of type in the signature is in the opposite direction as that of the inheritance.*

```cpp
class Base {
    virtual void f(Base);  
}
class Derived : public Base {
    void f(Object) override;  
}
```

Note: this is not legal C++ code

To override, argument type can be a supertype
## Type of Arguments

- For simplicity, we consider only input arguments
  - Arguments can be used for output or both
- Suppose that:
  - M is a method of a base class
  - M takes an argument of class A
  - M is overridden by M’ in a derived class
- What is the type of the corresponding argument of M’ in the derived class?

<table>
<thead>
<tr>
<th>Variance Type</th>
<th>Argument Type</th>
<th>Programming Language Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-variance</td>
<td>Must be A</td>
<td>C++, Java, C#</td>
</tr>
<tr>
<td>Contra-variance</td>
<td>A or base class thereof</td>
<td>Sather</td>
</tr>
<tr>
<td>Co-variance</td>
<td>A or derived class thereof</td>
<td>Eiffel</td>
</tr>
</tbody>
</table>

What about Squeak?
Arguments Covariance is Natural

More examples:

Base: Graph and Node. Derived: MyGraph and MyNode
Base: Animal and Food. Derived: Lion and Meat
class Food{}
class Meat extends Food{}

class Animal {
    void feed(Food f) {
        //...
    }
}

class Lion extends Animal {
    void feed(Meat m) {
        //...
    }
}

void f(Animal a) {
    Food f = new Food();
    a.feed(f);
}

f(new Lion()) ?!
Contra-variance is Safe

```java
class Food{
class Meat extends Food{}

class Animal {
    void feed(Food f) {
        //...
    }
}

class Pig extends Animal {
    void feed(Object o) {
        //...
    }
}

void f(Animal a) {
    Food f = new Food();
    a.feed(f);
}
```

Is Object as useful as Food ?!
Co-variance and Static Typing

- We have seen that co-variance is type unsafe
  - It may generate run-time type errors
- Q: How can co-variance exist in Eiffel?
  - Given Eiffel has static type checking
- A: Eiffel's type system has “holes” (*catcalls*)
  - Some messages will not have a compatible method in the receiver
  - The developer should be aware of that
    - Similar to programming in a dynamically typed language
- **A subclass with a covariant overriding is not a proper subtype of its superclass**
  - The compiler will allow assignability
  - But, you may get a “method-not-found” runtime error
Co-variance and Static Typing

• We have seen that co-variance is type unsafe.
  • It may generate run-time type errors.
• Q: How can co-variance exist in Eiffel?
  • Given Eiffel has static type checking
• Possible solution: A whole-world-analysis
  • The compiler tries to find all possible assignments to a variable
    • => Tries to predict the dynamic type of all variables
  • Cannot be achieved in the general case
Binary Methods

• A special case of a method with a covariant argument
  • The argument is of the same type as the receiver
  • Frequently occurs in many domains: comparison & sorting, arithmetics, recursive data-structures (graphs) etc.

```java
class Point {
    int x, y;
    boolean same(Point p) {
        return x == p.x && y == p.y;
    }
}

class ColorPoint extends Point {
    Color color;
    boolean same(ColorPoint cp) {
        return x == cp.x && y == cp.y && color == cp.color;
    }
}
```
Binary Methods (cont.)

• Partial solutions
  • Eiffel – **like current** (unsafe – possible runtime error)
  • Java – Recursive genericity (an interface can be implemented once)
  • LOOM – Self type (eliminates subtyping – not very useful)

```java
// Pseudo-code: LOOM
class Point {
    int x, y;
    boolean same(ThisClass p) {
        return x == p.x && y == p.y;
    }
}
class ColorPoint extends Point {
    Color color;
    boolean same(ThisClass cp) {
        return x == cp.x && y == cp.y && color == cp.color;
    }
}
Point p = new ColorPoint(); // Error – Not a subtype!
```
Return Type

• Suppose that:
  • M is a method of a base class
  • M returns a value of class A
  • M is overridden by M’ in a derived class

• Q: What should be the return type of M’ in the derived class?

• A: Covariance of return type is both natural and safe
class Food{}
class Meat extends Food{}

class Animal {
    Food whatDoIEat() {
        //...
    }
}

class Lion extends Animal {
    Meat whatDoIEat() {
        //...
    }
}

void f(Animal a) {
    Food f = a.whatDoIEat();
}
struct Employee {
    virtual Employee* clone() {
        return new Employee(*this);
    }
};

struct Manager : Employee {
    Manager* clone() override {
        return new Manager(*this);
    }
};

void f() {
    Employee* e1 = new Employee();
    Employee* e2 = new Manager();
    Employee* e3;
    e3 = e1->clone(); // e3 points to an Employee
    e3 = e2->clone(); // e3 points to a Manager
    Manager* m = (new Manager())->clone();
}
Variance in C++ and Java

• Return value: Co-variance is allowed!
  • C++: Must be pointer or reference
  • A later addition to both languages
  • Absolutely type safe
  • Quite useful
  • Painful C++ implementation... (in a few weeks)

• Arguments: No Variance

• If a method has a covariant argument, It does not override the “old” method
  • Java: The new method **overloads** the old one
  • C++: The new method **hides** the old one
Covariance + Overloading = Headache

- This is a legal Java program
  - Recall: Covariance of arguments => Overloading

```java
class A {
    int f(A a) { return 1; }
}
class B extends A {
    int f(A a) { return 2; }
    int f(B b) { return 3; }
}

void f() {
    A a = new A(), ab = new B();
    B b = new B();
    a.f(a); a.f(ab); a.f(b); // 1, 1, 1
    ab.f(a); ab.f(ab); ab.f(b); // 2, 2, 2
    b.f(a); b.f(ab); b.f(b); // 2, 2, 3
}
```

What would happen if we remove one at a time? And if this was C++?
Hiding in C++

• A similar program in C++
  • Recall: Covariance of arguments => hiding

```cpp
class A {
    public: virtual int f(A* a) { return 1; }
};
class B : public A {
    public: virtual int f(B* b) { return 3; }
};

void f() {
    A *a = new A(), *ab = new B();
    B* b = new B();
    a->f(a); a->f(ab); a->f(b);  // 1, 1, 1
    ab->f(a); ab->f(ab); ab->f(b); // 1, 1, 1
    b->f(a); b->f(ab); b->f(b); // compilation error!
}
```
Conformance and Arguments Number

• Suppose that:
  • M is a method of a base class taking N arguments
  • M is overridden by M’ in a derived class.

• Then, how many arguments should m’ have?
  • Exactly N:
    • Most current programming languages.
  • N or less:
    • It does not matter which arguments are omitted as long as naming is consistent.
  • N or more:
    • The BETA programming language (daughter of Simula).

• Note that adding arguments in an overriding method violates the “same or better” principle! (recall the LSP)
Conformance of Fields

• Usually, fields can be read-from and written-to
  • Similar to input-output arguments of a method
  • Thus, an overriding field should not change the field's type (No-variance)
  • But, in most languages there's no overriding of fields...

• What about final fields?
  • They can only be read from – just like return values
  • Thus, covariance of final fields is OK
Conformance of Exceptions

• Suppose that:
  • M is a method of a base class that throws exceptions of types \{E_1...E_N\}
  • M is overridden by M’ in a derived class
  • M’ throws exceptions of types \{X_1...X_L\}

• What is the relationship between \{E_1...E_N\} and \{X_1...X_L\}? 

Java
• Recall: a calling function should handle the callee’s exceptions – catch or declare
• Therefore, for each \(X_i\) \((1 \leq i \leq M)\) we require that there is \(E_j\) \((1 \leq j \leq L)\) where \(X_i\) is a subtype of \(E_j\)

C++
• exceptions are not part of the signature
• C++ throws clause is checked at run-time
Access Conformance

All versions of a method should have at least the same visibility

- Smalltalk: All methods are public
- Java: Subclass can only improve the visibility of a method
- C++: Not enforced, may lead to breaking of encapsulation

```c++
struct X {
    virtual void f();
};

struct Y : X {
    private:
    void f() override;
} y, *py = &y;

X* px = py;
py->f();   // Error! Y::f is private.
px->f();   // OK! (but breaks encapsulation)
```
Friendship and Overriding

A friend of base class is not necessarily a friend of the derived class.

class X {
    friend void amigo();
    virtual void f();
};
class Y : public X {
    void f() override;
};

void amigo()
{
    Y* y = new Y();
    y->f();   // Error! Y::f is private
    X* x = y; // Simple up casting
    x->f();   // OK! now the same Y::f is accessible
}
Design By Contract

• A method can declare **pre-conditions**
  • Expected arguments values
• A method can declare **post-conditions**
  • Guaranteed return value
• A class can declare **invariants**
  • Object’s state (when stable)
• Contract is checked at run time
  • Like an assert – exception thrown upon failure
• **Allows enforcing semantic compatibility!**
  • But only at run time 😞
Language Support

• Introduced in Eiffel, later adopted by others
• Use case: Contracts for Java
  • A library providing Design by Contract abilities

```java
@Invariant("balance >= 0")
class Account {
    int balance, opsCtr;
    @Requires("sum > 0")
    @Ensures("result = old(balance) + sum")
    int deposit(int sum) {
        balance += sum;
        opsCtr++;
        return balance;
    }
}
```
Pre-condition Conformance

Overriding method must demand the **same or less** from its client (preconditions are or-ed):

```java
class X {
    @Require("i > 0") void f(int i) { ... }
}
class Y extends X {
    @Require("i >= 0") void f(int i) { ... }
};
void f() {
    Y y = new Y();
    X x = y;
    x.f(1);  // Ok: >0 is expected
    y.f(0);  // Ok: 0 is allowed
}
```
Post-condition Conformance

Overriding method must ensure the **same or more** to its client (post-conditions are **and**-ed):

```java
class X {
    @Ensures("result > 0") int f() { ... }
}
class Y extends X {
    @Ensures("result > 5") int f() { ... }
};
void f() {
    Y y = new Y();
    X x = y;
    assert(x.f() > 0); // Ok: >0 is expected (got >5)
    assert(y.f() > 5); // Ok: >5 is expected
}
```
Derived must **keep** Base invariants; can **add** invariants guarding new fields:

```java
@Invariant("m > 0")
class X {
    int m;
}

@Invariant("n % 2 == 0")
class Y extends X {
    int n;
}
```

What would changing Base invariants cause?
- Reduce requirements: might surprise client
- Increase requirements: might get Base methods to break new invariants