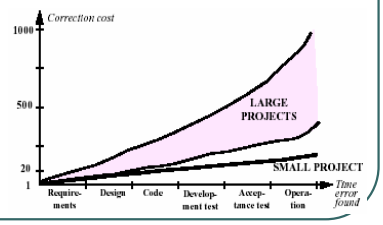
Typing Issues and LSP

Amit Shabtay

Typing

- Static typing
  - Readability (Java vs. Pearl)
  - Catching errors early
  - Reliability
  - Efficiency
- Dynamic typing
  - Flexibility
  - Speed



Weak Typing Flexibility (Python)

```
class Cat:
        def speak(self): print "meow!"
class Dog:
        def speak(self): print "woof!"
class Bob:
        def speak(self): print "hello, world!"

def command(pet):
        pet.speak()

pets = [ Cat(), Dog(), Bob() ]
for pet in pets: command(pet)
```


Typing

- "Programmer cycles are expensive, CPU cycles are cheap" (Bruce Eckel)

  Pro or Con static typing?
- How do we add flexibility to static typing?
  - Genericity – C++ templates, Java Generics
  - Inheritance (including multiple inheritance)

Covariance, Contravariance and NoVariance

- Covariance: The policy allowing a feature *redeclaration* to change the *signature* so that the new types of both arguments and result *conform* to the originals (inherited).
- Contravariance: The policy allowing a feature *redeclaration* to change the *signature* so that a new result type will *conform* to the original but the original argument types conform to the new.
- Novariance: The policy prohibiting feature *redeclaration* from changing the *signature*.

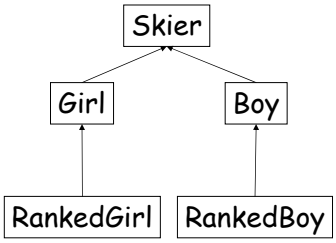
Covariance, Contravariance and NoVariance

- Covariance in method arguments: the ability to strengthen the type of an argument of a method.
- Contravariance: the ability to weakening the type of an argument.
- Novariance: the type can neither be strengthened nor weakened.

- class Parent {
  void test (covar : Mammal, contravar : Mammal) : boolean
  }
- class Child extends Parent {
  void test (covar : Cat, contravar : Animal) : boolean }

Covariance

```
Class Skier {
   Skier roommate;
   void share(Skier s);
}
Class Girl {
   Girl roommate;
   void share(Girl g);
}
Class RankedGirl {
    RankedGirl roommate;
   void share(RankedGirl rg);
}
```



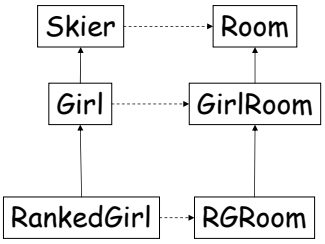
Covariance

- What happens when we do the following

```
Skier s; Boy b; Girl g;
b = new Boy(); g = new Girl();
s = b;
s.share(g);
g.share(b);
```

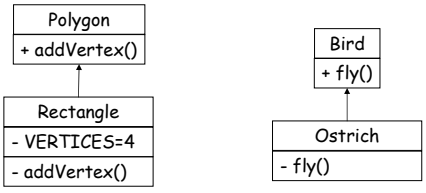

Covariance

- What about multiple hierarchies?
- Does it solve the problem?



Descendant Hiding

- How can a class hide its parent's method?



Solutions

- Some languages allow you to redeclare methods (Eiffel)
  - No Girl.share(Skier)
- Java and C++ do not allow this
  - Need to check the validity at runtime
  - if(skier instnaceof Girl) {…}
  - if(skier instnaceof Girl) {…} else throw…

The Liskov Substitution Principle

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

Barbara Liskov, 1988

LSP in plain English

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it

What's wrong with this?

```
void DrawShape(const Shape& s) {
   if (typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
   else if (typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}
```


Things Are Not Always That Simple

Consider the following class:

```
class Rectangle{
public:
   void SetWidth(double w) {_width=w;}
   void SetHeight(double h) {_height=w;}
   double GetHeight() const {return _height;}
   double GetWidth() const {return _width;}
private:
   double _width;
   double _height;
};
```


Square

- We want to add a Square object
  - Naturally derives Rectangle
- And the trivial implementation is:

```
void Square::SetWidth(double w) {
   Rectangle::SetWidth(w);
   Rectangle::SetHeight(w);
}
void Square::SetHeight(double h) {
   Rectangle::SetHeight(h);
   Rectangle::SetWidth(h);
}
```

- Do you see any problem ?

LSP is broken!

```
void g(Rectangle& r) {
   r.SetWidth(5);
   r.SetHeight(4);
   assert(r.GetWidth()*r.GetHeight())==20);
}
```

- A Square object is not Rectangle object!
  - Their behavior is different

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
- Use inheritance carefully!

Design By Contract

- A way to define the behavior of classes in hierarchy
  - A derived class must obey the contract of it's parent class
  - A contract can be "among gentlemen" or can be enforced using assertions and reflection.