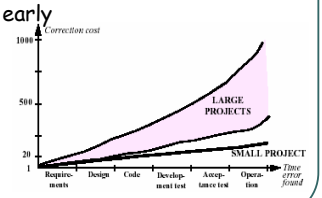
Typing Issues and LSP

David Rabinowitz

Typing

- Static typing
  - Reliability
  - Catching errors early
  - Readability
  - Efficiency
- Dynamic typing
  - Flexibility



March 28, 2006 Object Oriented Design Course 2

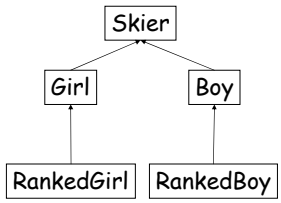
In the real world

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

March 28, 2006 Object Oriented Design Course 3

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);
}
```



March 28, 2006 Object Oriented Design Course 4

Covariance

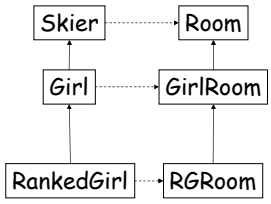
- What happens when we do the following

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

March 28, 2006 Object Oriented Design Course 5

Covariance

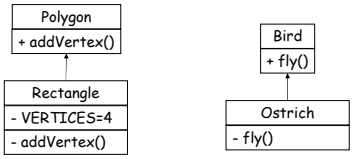
- What about multiple hierarchies?
- Does it solve the problem?



March 28, 2006 Object Oriented Design Course 6

Descendant Hiding

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



Solutions

- Some languages allow you to redeclare methods
  - 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!