LSP: the Liskov Substitution Principle

Subtypes must be substitutable for their base types

Originally stated as: (*Barbara Liskov, 1988)*

" What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."

LSP: the Liskov Substitution Principle

Corollary: *Functions that use pointers or refernces to base classes must be able to use objects of both existing and future derived classes without knowing it*

- in other words*: Inheritance must be used in a way that any property proved about supertype objects also holds for subtype objects*

LSP and OCP

- Motivation for LSP comes from OCP (at least partly)
  - Abstraction and polymorphism are used in standard OO-designs to achieve OCP, but how to use them?
    - in statically typed languages, the key mechanism to support this is inheritance
  - LSP *restricts* the use of inheritance, in a way that OCP holds (in the basic OCP design)
  - LSP addresses the questions of
    - what are the inheritance hierarchies that give designs conforming to OCP
    - what are the common mistakes we make with inheritance regarding OCP?

LSP and OCP

- Example: how does non-LSP inheritance break OCP?
  - Suppose that function *f* (the client code) takes as an argument a reference to an object of baseclass *B*. Assume that there is a derivative class *D of B* which, when passed to *f* as an object of baseclass *B* causes *f* to misbehave. Then *D* obviously violates LSP.
  - Now the author of *f* has to fix the problem by putting in some kind of test of the real type of argument, misbehaving *D* or well behaving some other derivative of *B*.
  - The client *f* has become aware of the derivatives of *B*, thus closure wrt. this kind of variation is broken.

→ *violation of LSP is a latent violation of OCP !*

There are limits to inheritance

```
class Bird {
public: virtual void fly();  // Bird can fly
};

class Parrot : public Bird {     // Parrot is a bird
  public: virtual void speak();  // Can Repeat words...
};

Parrot mypet;
mypet.speak();   // my pet being a parrot can Speak()
mypet.fly();     // my pet “is-a” bird, can fly

class Penguin : public Bird {
   public: void fly() {    // Penguins can’t, lets model it here
     error (“Penguins don’t fly!”); }
};
// client code
void PlayWithBird (Bird& abird) {
    abird.fly();    // OK if Parrot.
    // run time error if abird is a Penguin...OOOPS!!
}
```

... so what went wrong?

- The previous did not model: “*Penguins can’t fly*”
- It models “*Penguins may fly, but if they try it is error*”
  - Run-time error if attempt to fly
  - It does not help if you give an empty method *fly()* to penguin
- ***Think about Substitutability – this fails LSP, but why?***
  - because there is a program *playWithBird* defined in terms of baseclass *Bird* that will behave differently when an object of derived class *Penguin* is substituted for baseclass object *Bird*. Thus *Penguin* is not a subtype of *Bird (*it is a subclass however*)*.
  - a property *assumed* by the client about the base type did not hold for the subtype
- The author of subtypes must respect what the client of the base class can reasonably expect about the base class
- How can we anticipate what some client will expect ?
  - this leads us directly to ....

Design by Contract

- A class declares its behavior :
  - **Requirements** (Pre-conditions) that must be fulfilled
  - **Promises** (Post-conditions) that will hold afterwards
- Pre- and postcondition form a contract between the class and client using its services
  - this tells explicitly what the client may expect
- In the case of inheritance the following must hold:

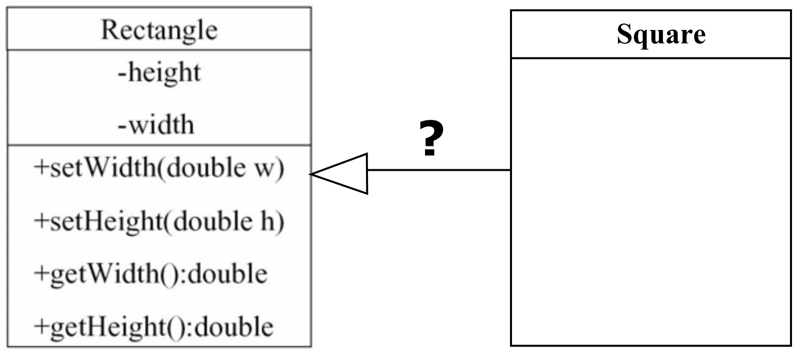
> *When redefining a method in a derivate class, you may only replace its* precondition by a weaker one*, and its* postcondition by a stronger one
>
> **B. Meyer**, 1988

  - Weaker here means that at least one constraint is dropped, stronger that all constrains remain and possibly new ones are added

Design by Contract

- In other words a derived class should *require no more,* and *provide no less* than the base class.
- Who is to blame, the author of Bird, Penguin or playingWithBirds?
  - Documented or not, Class Penguin in fact adds a precondition ”does not fly”, thus the precondition is stronger, derived class Penguin requires more.
  - It is reasonable for clients to expect birds to fly?

Square IS-A Rectangle?



- Is it wise to inherit Square from Rectangle?

Probably not

- You can do it by overriding **`setHeight`** and **`setWidth` `to behave "in sync"`**
  – static binding (in C++) so the following will not work if *f* is passed a square
    - **`void f(Rectangle& r) { r.setHeight(5); }`**
  – To fix this need to change base class to set methods virtual
    - Indication that something is not right! Why should the designer of rectancle allow redefinition of these very basic methods?
- The real problem – pass *square* to *g* …

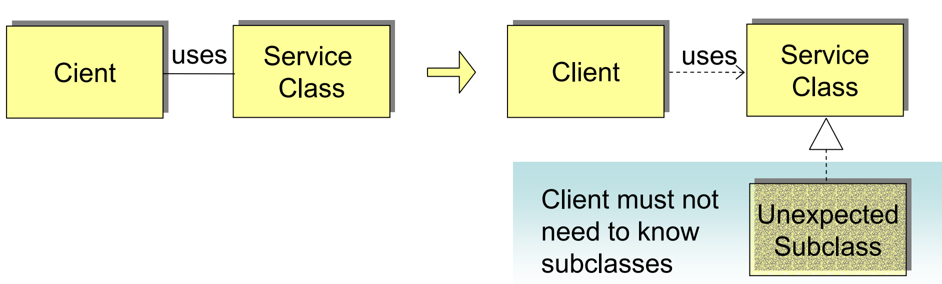
```
void g(Rectangle& r) {
  r.setWidth(5); r.setHeight(4);
  // How large is the area?
  assert(r.area()==20);
}
```

LSP is about Semantics and Replacement

- IS-A relationship is about *behavior*, not conceptual sub-types
- The *meaning* and *purpose* of every method and class must be *clearly documented*
  - Lack of user and programmer understanding will induce de facto violations of LSP
- Replaceability is crucial
  - Whenever any class is referenced by any code in any system, any future or existing subclasses of that class must be 100% replaceable
  - Because, sooner or later, someone **will** substitute a subclass;
    - it's almost inevitable.

LSP and Replaceability

- Any code which can legally call another class's methods must be able to substitute any subclass of that class without modification:



LSP Related Heuristic

It is illegal for a derived class, to override a base-class method with a method that does nothing

- Degenerate functions in derived classes is a strong telltale sign of LSP violation
- Solution 1: Inverse Inheritance Relation
  - if the initial base-class has only additional behavior
- Solution 2: Extract Common Base-Class
  - if both initial and derived classes have different behaviors
  - for **`Penguins`** $\rightarrow$ **`Birds`**, **`FlyingBirds`**, **`Penguins`**
- However, we do not always have the luxury of being able to touch the base class
- Throwing exceptions from derived classes is a strong telltale sign of LSP violation

LSP Conclusion

- Rectangle – square design was perfectly self-consistent and valid when isolated from its clients, however in the context of its clients it was clearly not valid
- *A model, viewed in isolation, cannot be meaningfully validated !*
- IS-A is about behavior, not about one problem domain concept being a special type of another
  - beware: this is the type of class hierarchy you'll most likely get from requirement analysts
- Design by contract explicitly states what to expect from a class and informs the author of client code about the behaviors that can be relied on
- OCP is at the hart of good oo-desing. LSP is its primary enabler.