

Agile Design Principles: The Liskov Substitution Principle

Based on Chapter 10 of Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003 and on Barbara Liskov and Jeannette Wing, "A behavioral notion of subtyping", ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 16, #6, 1994.

1

The Liskov Substitution Principle (LSP)

- "If S is a declared subtype of T, objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T"
- Note that the LSP is all about *expected behaviour* of objects. One can only follow the LSP if one is clear about what the expected behaviour of objects is.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 2

subtypes and instances

- For Java S is a **declared subtype** of T if
 - S is T,
 - S implements or extends T, or
 - S implements or extends a type that implements or extends T, and so on
- S is a **direct subtype** of T if
 - S extends or implements T

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 3

subtypes and instances

- An object is a **direct instance** of a type T
 - if it is created by a "new T()" expression
- An object is an **instance** of T
 - if it is a direct instance of a declared subtype of T.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 4

The Liskov Substitution Principle (LSP)

- "If S is a declared subtype of T, objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T"

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 5

Example of the LSP

```
void someClientCode( Bag t )
{
    Assertion.check( t.isEmpty() );
    t.put( new Range(0,N) );
    while( !t.empty() ) {
        Range r = (Range) t.take();
        if( r.size() > 2 ) {
            int m = part( r );
            t.put( new Range( r.low(), m ) );
            t.put( new Range( m, r.high() ) );
        }
    }
}
```

- Clearly the designer has some *expectations* about how an instance of Bag will behave.
- Let S be any declared subtype of Bag .
- If we pass in a direct instance of S, this code should still work.
- The expectations we have for instances of Bag should hold for instances of S.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 6

Expectations about behaviour

- Behavioural specification
 - The behavioural specification of a class explains the “allowable behaviours” of the instances of a class.
- S is a behavioural subtype of T if
 - an instance of type S behaves only as allowed of type T objects
- The LSP then says
“declared subtypes should be behavioural subtypes”

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 7

Expectations about behaviour

- So where do these *expectations about behaviour* live?
- In most language only a part of the expectations can be encoded in the language (for example types of parameters and results)
- The rest of our expectations have to be expressed in the documentation.
- [Time for an example.]

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 8

Bags and Stacks

the Bag class

- State: a bag (i.e. multiset) of Objects b
- isEmpty() : boolean
- take() : Object
 - Precondition: ! isEmpty()
 - Postcondition: result is an arbitrary member of b. The final value of b is the initial value with the result removed.
- put(ob : Object)
 - Precondition: true
 - Postcondition: The final value of b is its initial value with ob added.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 9

Bags and Stacks

the Stack class

- State: a sequence of Objects s
- isEmpty() : boolean
- take() : Object
 - Precondition: ! isEmpty()
 - Postcondition: result is the first item of s. The final value of s is the initial value with the first item removed.
- put(ob : Object)
 - Precondition: true
 - Postcondition: The final value of s is its initial value with ob prepended.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 10

Bags and Stacks

- Stacks are more constrained than Bags.
- A Stack object could be used where a Bag object is expected without violating our expectations of how a bag should behave.
- Thus Stack is a behavioural subtype of Bag.
- By the LSP, it is reasonable that Stack should be a declared subtype of Bag.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 11

Expectations about behaviour

- [Now where were we?]
- So where do these *expectations about behaviour* live?
- They have to come (in part) from the documentation.
- Such expectations can not come from the code, as
 - method implementations may be abstract
 - even if not abstract, method implementations can be overridden

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 12

The “counterfeit” test.

- Here is a way to think about behavioural subtypes:
 - Suppose I promise to deliver you an object of class T, but instead I give you an object x of class S.
 - You can subject x to any series of method calls you like (chosen from T’s signature).
 - If x behaves in a way that is not expected of a T object, then you know it is a counterfeit, x has failed the test.
 - If all S objects always pass the counterfeit test, then S is a behavioural subtype of T.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 13

Your turn

- the Square class
 - state: x, y, size
 - getX() returns x
 - getY() returns y
 - getWidth() returns size
 - getHeight() returns size
- the Rectangle class
 - state: x, y, width, height
 - getX() returns x
 - getY() returns y
 - getWidth() returns width
 - getHeight() returns height

Is Square a behavioural subtype of Rectangle?
Is Rectangle a behavioural subtype of Square?

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 14

Your turn

- the MutSquare class
 - state: x, y, size
 - getX() returns x
 - getY() returns y
 - getWidth() returns size
 - getHeight() returns size
 - setWidth(int w) size := w
 - setHeight(int h) size := h
- the MutRectangle class
 - state: x, y, width, height
 - getX() returns x
 - getY() returns y
 - getWidth() returns width
 - getHeight() returns height
 - setWidth(int w) width := w
 - setHeight(int h) height := h

Is Square a behavioural subtype of MutRectangle?
Is Rectangle a behavioural subtype of MutSquare?

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 15

LSP and syntactic signatures

- As we’ve seen. Semantic signatures for subtypes can be more specific compared to the supertype.
- The same applies to syntactic signatures (in many languages). Consider two Java classes:

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 16

LSP and syntactic signatures

```
class T {  
    Object a() { ... }  
}  
  
class S extends T {  
    @Override String a() { ... } ✓  
}
```

- This is allowed in Java.
 - More specific classes may have more specific return types
 - This is called “covariance”

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 17

LSP and syntactic signatures

- The same applies to exceptions

```
class T {  
    void b() throws Throwable { ... }  
}  
class S extends T {  
    @Override void b() throws IOException { ... } ✓  
}  
class U extends S {  
    @Override void b() { ... } ✓  
}
```

 - Every exception declared for the subtype’s method should be a subtype of some exception declared for the supertype’s method.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 18

LSP and syntactic signatures

- Logically it “could” be allowed for parameters to be “contravariant”

```
class T {  
    void c( String s ) { ... }  
}  
class S extends T {  
    @Override void c( Object s ) { ... } X  
}
```

- However this is actually not allowed (in Java), as it would complicate the overloading rules

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 19

Some cases where the LSP is difficult

- Like a healthy diet the LSP is obviously good for you, but it can be tempting to cheat a little.

- Consider a class

```
public class Point2D {  
    protected double x ;  
    protected double y ;  
    ... }  
}
```

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 20

Some cases where the LSP is difficult

- Consider Java’s toString method (inherited from Object)

```
class Point2D { ...  
    /** Return a string representation of the point.  
     * Postcondition: result is a string of the form  
     * ( xrep, yrep )  
     * where xrep is a string representing the x value and  
     * yrep is a string representing the y value */  
    @Override public String toString() {  
        return "(" + Double.toString(x) + ", "  
            + Double.toString(y) + ")";  
    }  
    ... }
```

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 21

Some cases where the LSP is difficult

- And Java’s “equals” method.

```
class Point2D { ...  
    /** Indicate whether two points are equal.  
     * Returns true iff the x and y values are equal. */  
    @Override public boolean equals(Object ob) {  
        if ( ob instanceof Point2D ) {  
            Point2D that = (Point2D) ob ;  
            return this.x == that.x && this.y == that.y ;  
        }  
        else return false ;  
    }  
    ... }
```

- So far so good.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 22

Some cases where the LSP is difficult

- Now consider extending Point2D to Point3D

```
public class Point3D extends Point2D {  
    protected double z ;  
    ... }
```

- We define toString as

```
@Override public String toString() {  
    return "(" + Double.toString(x) + ", "  
        + Double.toString(y) + ", "  
        + Double.toString(z) + ")";  
}
```

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 23

Some cases where the LSP is difficult

- Consider:

```
void printPoint( Point2D p ) {  
    p.setX(1.0) ; p.setY(2.0) ;  
    System.out.println( p.toString() ) ;  
}
```

- The behaviour will not be as expected if a Point3D is passed in.
- Surely there is no problem with our code though!
- The problem is with our expectations.

© 2007—9 T. S. Norvell

Engineering 5895 Memorial University

Liskov Substitution Principle
Slide 24

Two solutions

1. Lower expectations

```
/** Return a string representation of the point.  
 * Postcondition: result is a string indicating at least  
 * the x and y values. */  
@Override public String toString() {...as on slide 14...}
```

2. Prevent overrides

- It would be poor practice to prevent an override of `toString()`, so I use another name.

```
/** Return a string representation of the point.  
 * Postcondition: ... as on slide 14...*/  
public final String toString2D() {... as on slide 14 ... }
```

What about equals?

- Naturally, `equals` is also overridden in `Point3D`.

```
@Override public boolean equals(Object ob) {  
    if( ob instanceof Point3D ) {  
        Point3D that = (Point3D) ob ;  
        return this.z == that.z && super.equals( that ) ; }  
    else return super.equals( ob ) ; }
```

- (By the way, the reason for not just returning `false`, when the other object is not a `Point3D`, is that “`equals`” should be symmetric when neither object is null. I.e. `p2.equals(p3) == p3.equals(p2)`)

What about equals?

- So the code

```
void thisOrThat( Point2D p, Point2D q ) {  
    p.setX( x0 ) ; p.setY( y0 ) ;  
    q.setX( x1 ) ; q.setY( y1 ) ;  
    if( p.equals( q ) ) { ...do this... } else { ...do that... } }
```

may not behave according to our expectations.
(Consider the case where `x0 == x1`, `y0 == y1`, and `p.z != q.z`.)

- Again we have violated the LSP.

Two Solutions

1. Reduce Expectations.

- We should reword the documentation of `equals` for `Point2D` to be more flexible
- /** Do two `Point2D` objects compare equal by the standard of their least common ancestor class? <p> At this level the standard is equality of the x and y values.*/

2. Prevent overrides

- We wouldn't want to prevent overrides of `equals`. We are better off providing a new name
/** Are points equal as 2D points? */
public final boolean equals2D(Point2D that) {
 return this.x==that.x && this.y==that.y ; }

Lesson 0

- Every class represents 2 specifications
 - One specifies the behaviour of its direct instances
 - And this can be reverse engineered from the code.
 - One specifies the behaviour of its instances
 - And this can only be deduced from its documentation.
- It is important to document the behaviour that can be expected of all instances.
- It is less important to document the behaviour that can be expected of direct instances.
- However sometimes it is useful to do both.

Lesson 1

- When documenting methods that may be overridden,
- one must be careful to document the method in a way that will make sense for all potential overrides of the function.

Lesson 2

- One should document any restrictions on how the method may be overridden.
- For example consider the documentation of “equals” in Object. It consists almost entirely of restrictions on how the method may be overridden and thus it describes what the clients may expect.

Documentation of Object.equals(Object)

- Indicates whether some other object is “equal to” this one.
- The equals method implements an equivalence relation on non-null object references:
 - It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
 - It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
 - It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
 - It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
 - For any non-null reference value x, x.equals(null) should return false.
- The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Lesson 3

- It is particularly important to carefully and precisely document methods that may be overridden
- because one can not deduce the intended specification from the code.
 - For example, consider the implementation of equals in Object
 - `public boolean equals(Object ob) {`
 - `return this == ob ; }`
 - compared to the documentation on the previous slide.
- There may not even be any code.

In Summary

- The Liskov Substitution Principle
 - demands that subtyping (**extends** and **implements**, in Java) really lives up to its “is a” billing.
 - prevents breaking client code when objects of a declared subtype are passed to it.
 - is thus good practice.
 - can not be practiced without careful and precise documentation of object behaviour.

By the way

- Martin's description is pithier:
“Subtypes must be substitutable for their base types”
- But, I have no idea what this means.
- For example, if S is a subtype of T and I have a statement
`if(x instanceof T) doThis() ; else doThat() ;`
it would be a mistake to replace T with S.
And in the statement
`T x = new T() ;`
it would definitely be a mistake to replace the first T with S,
and it could be a mistake to replace the second with S.