Design Principles: Part 2

ENGI 5895: Software Design

Andrew Vardy

Faculty of Engineering & Applied Science
Memorial University of Newfoundland

January 29, 2018

Liskov Substitution Principle (LSP) Dependency-Inversion Principle (DIP) Interface-Segregation Principle (ISP)

Outline

1 Liskov Substitution Principle (LSP)

2 Dependency-Inversion Principle (DIP)

3 Interface-Segregation Principle (ISP)

Liskov Substitution Principle (LSP) Dependency-Inversion Principle (DIP) Interface-Segregation Principle (ISP)

The Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

- You might think that this is enforced by your language (i.e. Java, C++, ...)
  - e.g. If Gorilla inherits from Animal then I should be able to substitute a Gorilla whenever I'm asked to supply an Animal
- However, violations of the LSP are often quite subtle
- The LSP is violated whenever the reasonable expectations of a user of a base class are not met for the derived class

e.g. Rectangles and Squares

Consider the following class hierarchy:

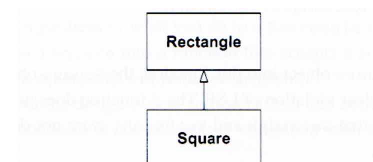


Figure 10-1 Square inherits from Rectangle

This seems quite natural. Isn't a square just a particular type of rectangle? It is hard to foresee any problems with this logic. Yet problems arise...

Consider our Rectangle class:

```
class Rectangle {
    protected Point topLeft;
    protected double width, height;

    public void setWidth(double w) { width = w; }
    public void setHeight(double h) { height = h; }

    public double getWidth() { return width; }
    public double getHeight() { return height; }

    public double getArea() { return width * height; }
}
```

The first hint of trouble comes when we consider how to maintain both width and height. Our first attempt to address this problem will utilize only the width attribute...

```
class Square1 extends Rectangle {
    @Override public void setWidth(double w) {
        width = w;
    }

    @Override public void setHeight(double h) {
        width = h;
    }
}

public class Tester1 {
    public static void testHeight(Rectangle r) {
        r.setHeight(10);
        assert r.getHeight() == 10;
    }

    public static void main(String[] args) {
        testHeight(new Rectangle());

        testHeight(new Square1());
    }
}
```

The second test fails! Square1 is not substitutable for Rectangle w.r.t. `testHeight`. Lets try again, with Square2 always keeping width and height the same...

```
class Square2 extends Rectangle {
    @Override public void setWidth(double w) {
        width = w;
        height = w; // Make the Square stay square
    }

    @Override public void setHeight(double h) {
        height = h;
        width = h; // As in 'setWidth'
    }
}

public class Tester2 {
    public static void testArea(Rectangle r) {
        r.setWidth(5);
        r.setHeight(4);
        assert r.getArea() == 20;
    }

    public static void main(String[] args) {
        testArea(new Rectangle());
        testArea(new Square2());
    }
}
```

This fails again!! Square2 is not substitutable for Rectangle w.r.t. `testArea`. The substitution principle is violated!

Design by Contract

This situation might have been avoided by adding sufficiently strong pre- and post- conditions. Consider the following postcondition for Rectangle's `setWidth(double w)` method: postcondition:

- $width' == w$ and $height' == height$

If tested (via assert) this postcondition would have detected the error in Square2 (for Square1 a corresponding postcondition for `setHeight` would have been required). Note that preconditions and postconditions for derived classes should follow Meyer's rule:

> *"A routine redeclaration [i.e. an overridden method] may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger."*

e.g. Lines and LineSegments

We want to represent both lines and line segments. Considering how much these classes have in common, we adopt Line as the base class:

```
public class Line {
    protected Point p1, p2;

    public Line(Point p1, Point p2) {/**/}
    public double getSlope() {/**/}
    public double getIntercept() {/**/}
    public boolean isOn(Point p) {/**/}
}
```

```
public class LineSegment extends Line {
    public LineSegment(Point p1, Point p2) {/**/}
    public double getLength() {/**/}
    @Override
    public boolean isOn(Point p) {/**/}
}
```

- Reasonable assumption for a user of Line:
  - For all points p lying on the infinite line, `isOn(p)` returns true
- e.g.
  - `p = new Point(0, line.intercept)`
  - We expect that `line.isOn(p)` returns true
  - However, in some cases this will not be the case for LineSegment! This is a violation of the substitution principle!

Solution

Factor out the common elements of Line and LineSegment into a new superclass that both inherit from. The common elements here includes everything but isOn. This new superclass, LinearObject, is made abstract since we do not want to instantiate it.

```
public abstract class LinearObject {
    protected Point p1, p2;

    public LinearObject(Point p1, Point p2) {/**/}
    public double getSlope() {/**/}
    public double getIntercept() {/**/}

    public abstract boolean isOn(Point p);
}
```

Concrete classes Line and LineSegment extend the abstract class LinearObject:

```
public class Line extends LinearObject {
    public Line(Point p1, Point p2) {/**/}
    @Override
    public boolean isOn(Point p) {/**/}
}

public class LineSegment extends LinearObject {
    public LineSegment(Point p1, Point p2) {/**/}
    @Override
    public boolean isOn(Point p) {/**/}
}
```

It would be easy to add further linear objects such as Ray.

Relation to OCP

- The LSP enables the OCP:
  - e.g. LinearObject is now open for extension by inheriting new derived classes from it. Behavior is added by adding code as opposed to changing existing code

The Dependency-Inversion Principle (DIP)

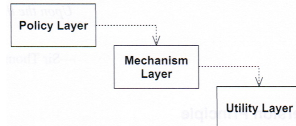
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Another way of stating this principle is this:

> *"...you should not depend on a concrete class—all relationships in a program should terminate on an abstract class or an interface."*

But take this with a grain of salt. If a concrete class is not anticipated to change much, it does little harm to depend on it.

e.g. Consider a high-level Policy Layer module and the two lower-level modules it depends on:



In applying the DIP we replace direct dependencies with interfaces:

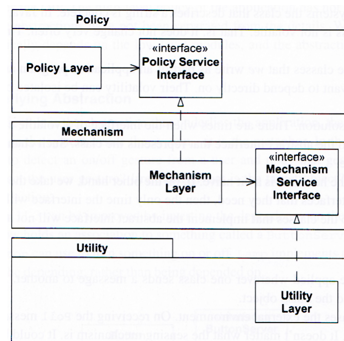


Figure 11-2 Inverted Layers

e.g. Consider a simple Button object that is able to determine if a button has been pressed. We wish to connect this Button to an object such as a Lamp to turn it on or off. We assume that some higher-level code is calling the Button's poll method repeatedly.

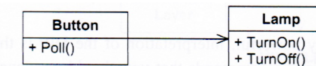


Figure 11-3 Naive Model of a Button and a Lamp

This yields the following implementation for Button:

```
public class Button {
    private Lamp lamp;
    public void Poll() {
        if (/* Button was pressed */)
            lamp.TurnOn();
    }
}
```

What's wrong with this? What if we want to use Button to control a Motor or Car class? Button.java must change.

The job of a button is to detect button presses, not talk to lamps! The current design can be viewed as violating the single-responsibility and open-closed principles as well.
The solution is to invert the dependency of Buttons on Lamps and for Button to instead depend on a new interface:

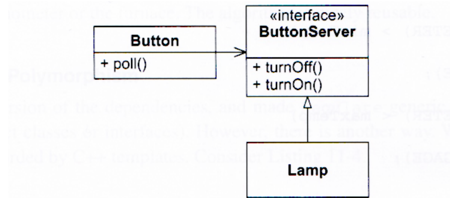


Figure 11-4 Dependency Inversion Applied to the Lamp

e.g. Furnace

Consider a simple furnace control system (written in C++). The software reads temperature from an IO channel and tells the furnace to engage or disengage through another channel.

```
#define THERMOMETER 0x86
#define FURNACE 0x87
#define ENGAGE 1
#define DISENGAGE 0

void Regulate(double minTemp, double maxTemp) {
    for (;;) {
        while (in(THERMOMETER) > minTemp)
            wait(1);    // Wait 1 sec.
        out(FURNACE, ENGAGE);

        while(in(THERMOMETER) < maxTemp)
            wait(1);
        out(FURNACE, DISENGAGE);
    }
}
```

The high-level algorithm (wait to cool, then turn on furnace, wait to heat up, turn off furnace) is mixed up with the low-level details of checking the temperature and turning on/off the furnace.

The following UML diagram shows the application of the DIP (note how stereotypes are used to alter the default meaning of symbols). Regulate now depends on interfaces, not IO channels.

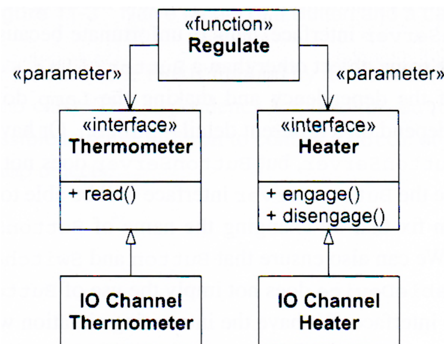


Figure 11-5 Generic Regulator

Here is the code:

```
void Regulate(Thermometer &t, Heater &h,
              double minTemp, double maxTemp) {
    for (;;) {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```

Notice that the algorithm itself is more cleanly expressed. Most importantly, we are insulated from changes to the thermometer and heater.

The Interface-Segregation Principle (ISP)

Interfaces provide an abstract way for two classes to communicate. However, an interface that provides different groups of methods to different clients is bad design. This is really just the Single Responsibility Principle applied to interfaces.
The point is to avoid "fat" interfaces, where there is little cohesion between methods.

e.g. Secure Doors

Assume that we wish to represent different kinds of doors. We define class Door:

```
abstract class Door {
    public abstract void lock();
    public abstract void unlock();
    public abstract boolean isOpen();
}
```

We wish to create a concrete implementation of Door, called TimedDoor. A TimedDoor will sound an alarm if it has been left unlocked for too long.
We decide to delegate the responsibility of managing the timing to a Timer class (application of the SRP). A TimerClient is also defined as an interface for classes who use Timer (application of the DIP):

```
class Timer {
    public void register(int timeout, TimerClient client) {
        // Trigger an internal timer that will call
        // client.timeOut() after 'timeout' seconds has
        // elapsed.
    }
}

interface TimerClient {
    void timeOut();
}
```

TimedDoor extends Door. But who implements TimerClient?

Choice 1: Door implements TimerClient.

```
abstract class Door implements TimerClient {
    public abstract void lock();
    public abstract void unlock();
    public abstract boolean isOpen();

    public void timeOut() {/**/}
}
```

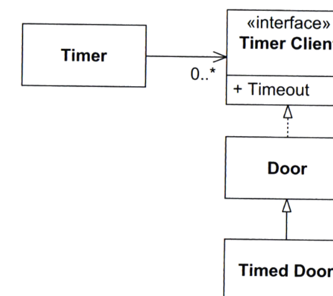


Figure 12-1 Timer Client at Top of Hierarchy

This is **bad**. Not all Door variants will need built-in timing.

Choice 2: TimedDoor implements TimerClient:

```
class TimedDoor extends Door implements TimerClient {
    @Override public void lock() {/**/}
    @Override public void unlock() {/**/}
    @Override public boolean isOpen() {/**/ return true;}

    @Override public void timeOut() {/**/}
}
```

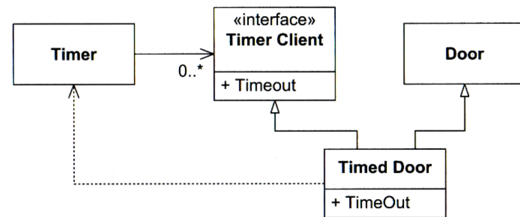


Figure 12-3 Multiply inherited Timed Door

This is much better! Doors are not polluted with timing-related stuff. It may be cleaner again to make Door an interface (this may not be possible if there is common implementation for all Doors).