


# *Programming for extensibility – what OO really provides?*

by  
**Venkat Subramaniam**  
*venkats@agiledeveloper.com*

*July 2003*

Presentation and examples can be downloaded from  
<http://www.agiledeveloper.com/download.aspx>

## **Abstract**

- We use OO languages like Java, Smalltalk, C++, and C# for our application development. However, the code we write, is it really object-oriented? If so, how much of it is? Then again, what is object-oriented programming and why should we develop application using this paradigm? This topic addresses the fundamental question once again. We define and discuss the concept and issues of extensibility and what it takes to make a system extensible? It presents some very strong design principles, those that can change the way we develop our systems. Java examples of code that will benefit and code that applies the principles will be presented.
- Dr. Venkat Subramaniam is an agile developer who teaches and mentors software developers. He has significant experience in architecture, design and development of distributed object systems. Venkat has trained more than 2500 software professionals around the world. He is also an adjunct professor at University of Houston and teaches the Professional Software Developer Series at Rice University's Technology Education Center.
- Each page with a  has an attached example

## The Pillars of the Paradigm

- Abstraction
- Encapsulation
- Hierarchy
  - Association, Aggregation
  - Inheritance
- Polymorphism

## What is OO development?

- Algorithmic vs. OO approach
- Modeling the system
- System viewed as collection of entities
- Entities have information and behavior
- Request for service and respond to requests

## What's the benefit of the Paradigm?

- Abstraction provide modeling of system
  - Simplified model relative to perspective of viewer
- Encapsulation provides separation of concerns
  - Hides details
  - Easy to depend on and use
  - Flexibility to change
  - Localized modifications
- Polymorphism
  - Take this away and it is not OO any more
  - Provides for the extensibility
  - It is what puts the Orientation in OO

## An Assignment For Me

- There is a door which needs to be monitored
  - Assume you can interface with an API to get door status
- An alarm system needs to check
  - if the door remains open after a certain interval
- Raise an alarm if door is not secured close in time

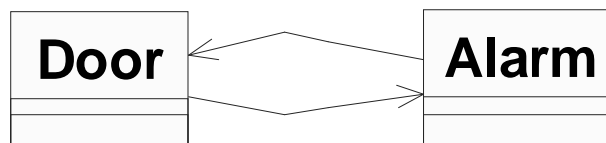
## Alarm App First Shot

```
public class Door {  
    private boolean closed = true;  
    public void open() {  
        closed = false;  
        Alarm anAlarm = new Alarm(this, 30);  
    }  
    public void close() { closed = true; }  
    public boolean isClosed() { return closed;}  
}
```

```
public class Alarm {  
    public Alarm(final Door aDoor, final int seconds){  
        Thread monitoringThread = new Thread(  
            new Runnable() {  
                public void run() { ...  
                    Thread.sleep(seconds * 1000);  
                    if (!aDoor.isClosed()) raiseAlarm();  
                }  
            }).start();  
    }  
}
```



## The classes



A couple of months later you are asked to write a program to monitor a reaction and raise an alarm if not under control within a short time!

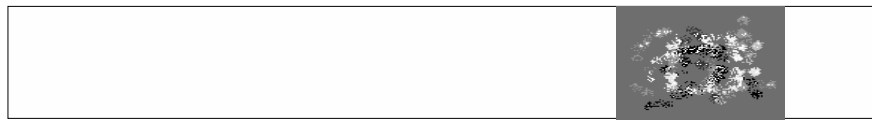
Wouldn't it be nice if you can simply use my Alarm class in your app.

## Options

- You can take my Alarm and modify
  - Poor form of code reuse
- You may inherit your Reaction from the Door!
  - that hurts

## Nature of code

- “Software Systems change during their life time”
- Both better designs and poor designs have to face the changes; good designs are stable



## Open-Closed Principle

Bertrand Meyer:

***“Software Entities (Classes, Modules, Functions, etc.) should be open for extension, but closed for modification”***

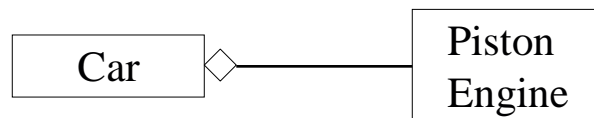
## Good vs. Bad Design

- Characteristics of a poor design:
  - Single change results in cascade of changes
  - Program is fragile, rigid and unpredictable
- Characteristics of good design:
  - Modules never change
  - Extend Module's behavior by adding new code, not changing existing code

## Good Software Modules

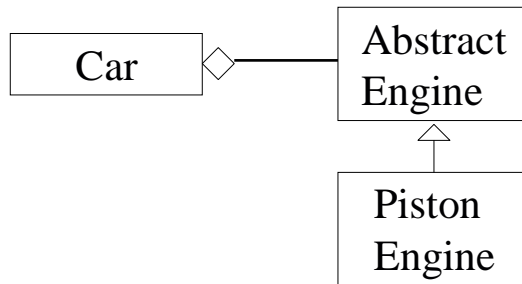
- Software Modules must
  - be open for extension
    - module's behavior can be extended
  - closed for modification
    - source code for the module must not be changed

## Looking out for OCP



- How to make the Car run efficiently with Turbo Engine ?
- Only by changing Car in the above design

## Providing Extensibility



**Abstraction &  
Polymorphism  
are the Key**

- ***A class must not depend on a Concrete class; it must depend on an abstract class***

## Strategic Closure

### **Strategic Closure:**

No program can be 100% closed

There will always be changes against which the module is not closed

***Closure is not complete - it is strategic***

Designer must decide what kinds of changes to close the design for.

This is where the experience and problem domain knowledge of the designer comes in



# Conventions from OCP

## Heuristics and Conventions that arise from OCP

- Make all member variables private
  - encapsulation: All classes/code that depend on my class are closed from change to the variable names or their implementation within my class. Member functions of my class are never closed from these changes
  - Further, if this were public, no class will be closed against improper changes made by any other class
- No global variables



# Conventions from OCP...

## Heuristics and Conventions that arise from OCP...

- RTTI is ugly and dangerous
  - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module

Not all these situations violate OCP all the time

## Usage of RTTI – *instanceof*

- Keep usage of RTTI to the minimal
- If possible do not use RTTI
- Most uses of RTTI lead to extensibility issues
- Some times, it is unavoidable though
  - some uses do not violate OCP either



## Problems for extensibility

- Developing for overrideability may not be easy

```
public class Point
{
    private final int x;
    private final int y;
    public Point(int px, int py)
    { x = px; y = py; }
    public boolean equals(Object o)
    {
        if (!(o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }
}
```

## Contract of equality

- **Reflexivity**
  - requires that for any reference x, x.equals(x) should return true.
- **Symmetry**
  - requires that for any referneces x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- **Transitivity**
  - requires that for any references x, y and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- **Consistency**
  - requires that repeated calls to x.equals(y) should consistenly return a true or consistently return a false, if no data /state has changed in either object.
- **Non-nullity**
  - requires that the for any non-null reference x, x.equals(null) should return false.

## Overriding Equals

```
public class ColorPoint extends Point {
    private Color color;
    public ColorPoint(int px, int py, Color clr)
    {
        super(px, py);
        color = clr;
    }
    public boolean equals(Object o)
    {
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) o;
        return super.equals(o) &&
            color.equals(cp.color);
    }
}
```

Fails Symmetry!



## Fixing the equals Symmetry

- ColorPoint's equals modified to

```
public boolean equals(Object o)
{
    if (!(o instanceof Point)) return false;
    // If o a normal Point, color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);
    // o is a ColorPoint; do a full comparison

    ColorPoint cp = (ColorPoint) o;
    return super.equals(o) &&
        color.equals(cp.color);
}
```

Fails Transitivity!



## Joshua Bloch's Conclusion

- In Effective Java Programming Guide

*"So what's the solution? It turns out that this is a fundamental problem of equivalence relations in object-oriented languages. **There is simply no way to extend an instantiable class and add an aspect while preserving the equals contract.** There is, however, a fine workaround. Follow the advice of Item 14, 'Favor composition over inheritance.' ...*

## Fixing the equals one last time

- Our argument: While surely consider composition over inheritance, situation may not be that bleak for this problem?

```
//Point's equals method
public boolean equals(Object o) {
    if (!(o.getClass() == getClass())) return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

- **While substitutability provides great extensibility in a system, we have to be very careful in implementing these concepts. It requires quite a bit of insight and analysis to get it done right.**

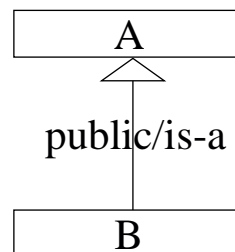


*Quiz Time*

# Liskov Substitution Principle

- Inheritance is used to realize Abstraction and Polymorphism which are key to OCP
- How do we measure the quality of inheritance?
- LSP:  
***“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it”***

## Inheritance



B publicly inherits from (“*is-a*”) A means:

- every object of type B is also object of type A
- whats true of object of A is also of object of B
- A represents a more general concept than B
- B represents more specialized concept than A
- **anywhere an object of A can be used, an object of B can be used**

# Behavior

Advertised Behavior of an object

- Advertised Requirements (Pre-Condition)
- Advertised Promise (Post Condition)

Stack and eStack example

# Design by Contract

## ***Design by Contract***

*Advertised Behavior of the*

*Derived class is Substitutable for that of the  
Base class*

Substitutability: Derived class Services Require  
no more and promise no less than the  
specifications of the corresponding  
services in the base class

# LSP

***“Any Derived class object must be substitutable where ever a Base class object is used, without the need for the user to know the difference”***

## LSP in Java?

- LSP is being used in Java at least in two places
- Overriding methods can not throw new unrelated exceptions
- Overriding method's access can't be more restrictive than the overridden method
  - for instance you can't override a public method as protected or private in derived class



## Nature of Bad Design

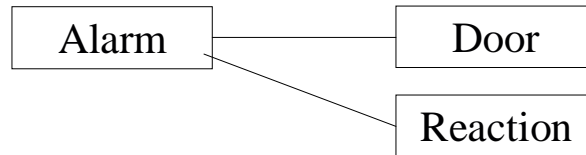
- Bad Design is one that is
  - Rigid - hard to change since changes affect too many parts
  - Fragile - unexpected parts break upon change
  - Immobile - hard to separate from current application for reuse in another

## Ramifications

### Problems:

- Modification to the Door class may require modifications to the Alarm, at least recompilation
- Alarm can not be used as is in another application which wants the Alarm to monitor say a Reaction.

## One Solution



*rigid, fragile & immobile*

*What does it take to  
use this Alarm to monitor  
yet another entity?*

## Dependencies

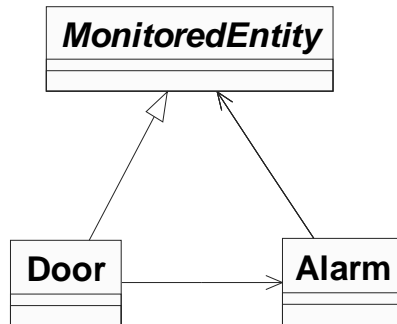
“High level modules should not depend upon low level modules. Both should depend upon abstractions.”

**“Abstractions should not depend upon details.**

Details should depend upon abstractions.”

## Alarm App Another Shot

### *Stable Dependencies:*

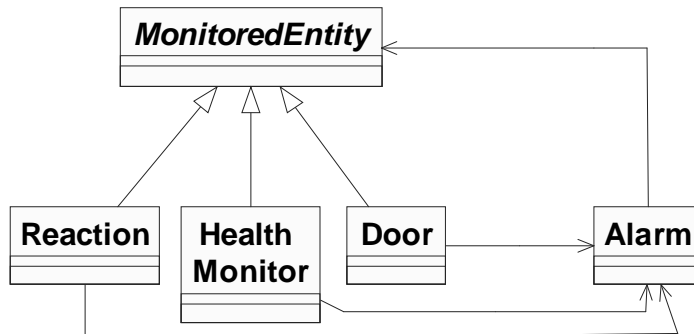


*Alarm depends on abstraction & not on details. Stable to changes to the details. Easy to reuse in other apps since does not come with a baggage.*



## Alarm App Final Shot

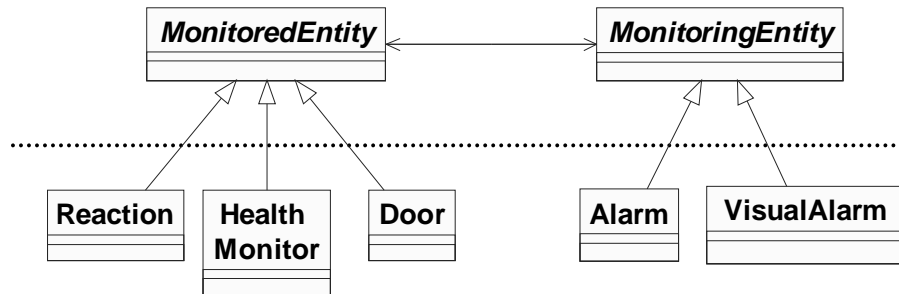
### *Better Extensibility & Reusability*



- What if a Device in other app does not inherit from **MonitoredEntity**
  - Adapter Pattern



## What about other kinds of Monitors



- A better system is one which has a layer of abstraction and a layer of concreteness
- Dependency runs vertically from concrete to abstract
- Not horizontally from concrete to concrete

## The Founding Principles

- The three principles are closely related
- Violating either LSP or DIP invariably results in violating OCP
- It is important to keep in mind these principles to get the most out of OO development

## What is Object-Oriented again?

- Not just a system that
  - has objects
  - uses C++, Java, etc.
  - uses UML
- A system built with the following in mind
  - Extensibility
  - Maintainability
  - Cost
  - Performance
- Does not compromise the fundamental principles
  - Open-Closed Principle
  - Liskov's Substitution Principle
  - Dependency Inversion Principle

*Quiz Time*

## Further Reading

1. Excellent compilation of these and more by Robert C Martin at <http://www.objectmentor.com/resources/articleIndex>  
Click on Design Principles
2. Effective Java Programming Guide, Joshua Bloch
3. Java Design, by Peter Coad, et. al.
4. <http://www.agiledeveloper.com/download.aspx>