Prudent OO Development

Venkat Subramaniam
venkats@agiledeveloper.com

September 2003

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

Abstract

Abstract Developing with objects involves more than using languages like Java, C#, C++ or Smalltalk for that matter. How object-oriented is our code? From time to time, the OO paradigm can stump even expert developers. In this presentation the author will present some of the challenges that are fundamental in nature. Then he will present some principles and good practices for prudent development of OO code.

Speaker Dr. Venkat Subramaniam, founder of Agile Developer, Inc., has taught and mentored more than 2,500 software developers around the world. He has significant experience in architecture, design, and development of distributed object systems. Venkat is an adjunct professor at the University of Houston and teaches the Professional Software Developer Series at Rice University's Technology Education Center. He may be reached at venkats@agiledeveloper.com.

Examples Any page with a  has an example attached
Download from http://www.agiledeveloper.com/download.aspx

Prudent OO Development

- **Basics**
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- ISP
- Conclusion

The Pillars of the Paradigm

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

What's OO?

- Is it using Objects?
- Is it using C++, Java, C#, Smalltalk?
- No, its got to be using UML?! ☺

- What makes a program OO?
- How do you measure good design?

Prudent OO Development

- Basics
- **Metrics**
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- ISP
- Conclusion

Metrics for class Design

- Cohesion
  - The object is focused on doing one thing well
- Coupling
  - Number of classes that your class depends on
- A class is forced to change more often if
  - it does more than one thing – Low Cohesion
  - It depends on a number of classes – high coupling
- We should strive for
  - **High cohesion**
  - **Low coupling**

Prudent OO Development

- Basics
- Metrics
- **Object Copying**
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- ISP
- Conclusion

Object Copying

- Let's first look at some thing simple and fundamental
- How do we copy an object
- Consider an example of a Car with an Engine

Venkat's past recommendation

- Before reading Bloch's Effective Java!
- Writing a Copy Constructor is a bad idea
- Why?
- Leads to extensibility issues

Bloch's Recommendation

- Cloning comes with its own problems
  - No constructor called when object cloned
  - If a class has final fields, these can't be given a value within close method!
- Bloch's recommendation:

"... you are probably better off providing some alternative means of object copying or simply not providing the capability." He goes on to say **"A fine approach to object copying is to provide a copy constructor."**

- I agree with the part "simply not providing the capability"
- But providing a copy constructor has problems mentioned earlier?
- What's the solution?

Further Reading: [1]

A combined approach

- Implement the clone method
  - but not the way it is generally done in Java
- Write a protected copy constructor
- From the clone method, invoke the protected copy constructor

Further Reading: [5]

Prudent OO Development

- Basics
- Metrics
- Object Copying
- **Bad Design**
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- ISP
- Conclusion

Bad design

- Perils of a bad design
  - Rigidity
    - Hard to change, results in cascade of changes
  - Fragility
    - Breaks easily and often
  - Immobility
    - Hard to reuse (due to coupling)
  - Viscosity
    - Easy to do wrong things, hard to do right things
  - Needless Complexity
    - Complicated class design, overly generalized
  - Needless Repetition
    - Copy and Paste away
  - Opacity
    - Hard to understand

Principles

- Guiding Principles that help develop better systems
- Use principles only where they apply
- You must see the symptoms to apply them
- If you apply arbitrarily, the code ends up with *Needless Complexity*

Prudent OO Development

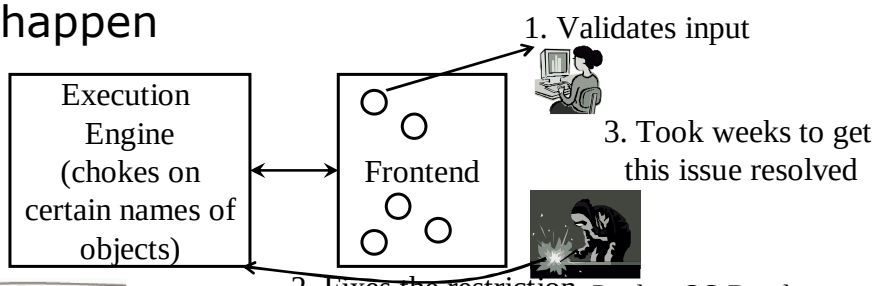
- Basics
- Metrics
- Object Copying
- Bad Design
- **DRY**
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- ISP
- Conclusion

DRY

- Don't Repeat Yourself

Further Reading: [3]

- "Every Piece of Knowledge must have a single, unambiguous, authoritative representation within a system"
- One of the most difficult, but most seen
- How many times have you see this happen



DRY

- Some times hard to realize this
- It is much easier to copy, paste and modify code to get it working the way you want it, isn't it

- Duplicating code results in
  - Poor maintainability
  - Expensive to fix bugs/errors
  - Hard to keep up with change

Prudent OO Development

- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- **Composition vs. Inheritance**
- SRP
- OCP
- LSP
- DIP
- ISP
- Conclusion

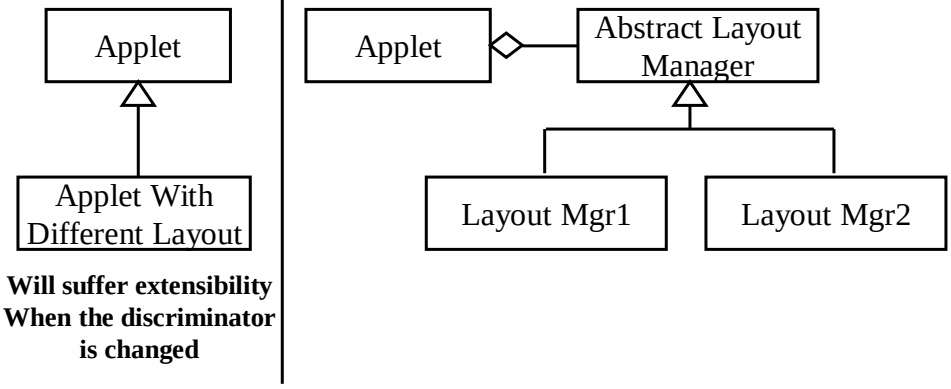
Composition vs. Inheritance

- Inheritance represents *is-a* relationship
- Inheritance increases coupling

  Further Reading: [4]
  - Stronger binding to base class
- Inheritance is not necessarily for code reuse
- It has more to do with Substituitability
- "Use composition to extend responsibilities by delegating work to other more appropriate objects"

Strategy

- Example: Use of strategy pattern in Java applet layout manager – instead of inheritance



Prudent OO Development

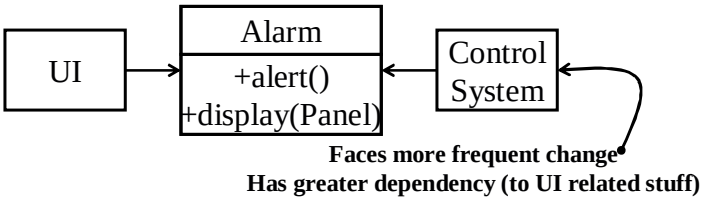
- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- **SRP**
- OCP
- LSP
- DIP
- ISP
- Conclusion

SRP

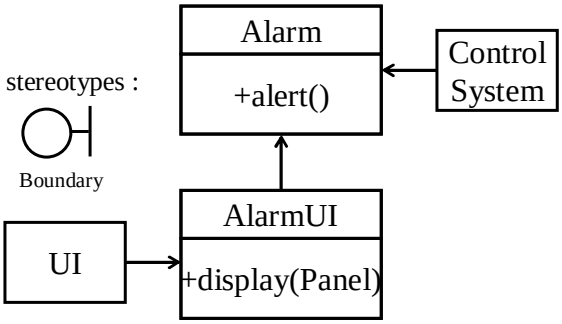
- Single-Responsibility Principle
- What metric comes to mind?
- "A class should have only one reason to change"
- Some C++ books promoted bad design
  - Overloading input/output operators!
- What if you do not want to display on a terminal any more?
  - GUI based, or web based?

Further Reading: [2]

SRP...

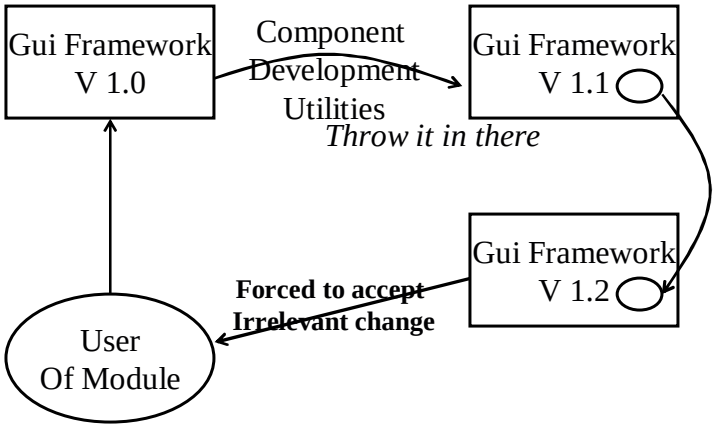


Related topics:
MVC
Analysis model stereotypes :



SRP at Module Level

- Can be extended to module level as well



SRP affects Reuse

- Lower cohesion results in poor reuse
  - My brother just bought a new DVD and a big screen TV!
  - He offers to give me his VCR!
  - I have a great TV and all I need is a VCR
  - Here is what I found when I went to pickup!

Tight coupling
Poor Cohesion
Bad for resuse

Disclaimer: This slide not intended to say anything about the brand of product shown here as an example!

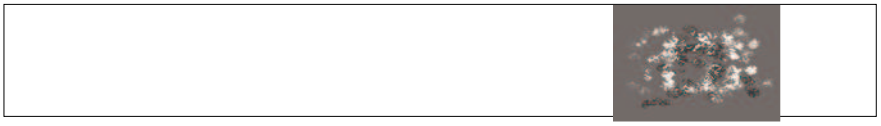
Quiz Time

Prudent OO Development

- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- **OCP**
- LSP
- DIP
- ISP
- Conclusion

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



OCP

Open-Closed Principle

Further Reading: [2]

Bertrand Meyer:

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

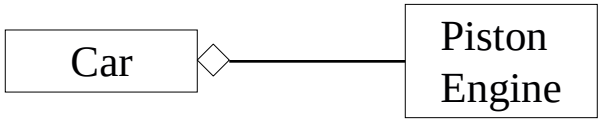
Good vs. Bad Design

- 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

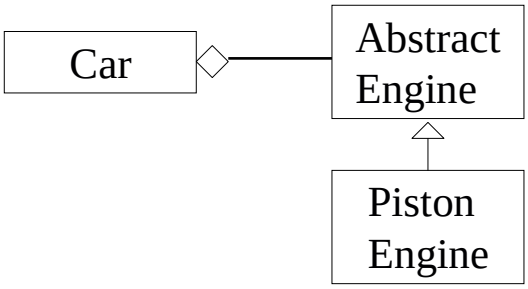
- Software Modules must
  - be open for extension
    - module's behavior can be extended
  - closed for modification
    - source code for the module must not be changed

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

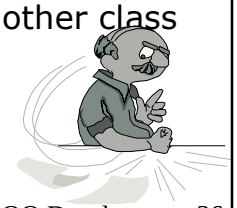
Designer must decide what kinds of changes to close the design for.

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

Usage of Reflection

- Reflection allows use to invoke methods and access objects without compile time dependency
- Great, let's use reflection for all calls?!

- Better to depend on an interface rather than using reflection
- Avoid use of reflection, except for
  - Dynamic object creation (abstract factory)
  - And in cases where the benefit outweighs cost and clarity

Prudent OO Development

- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- **LSP**
- DIP
- ISP
- Conclusion

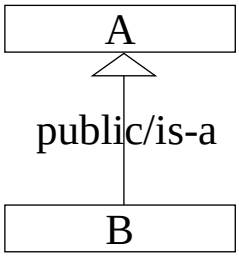
Liskov Substitution Principle

Further Reading: [2]

- 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

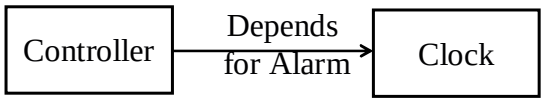
Prudent OO Development

- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- **DIP**
- ISP
- Conclusion

Nature of Bad Design

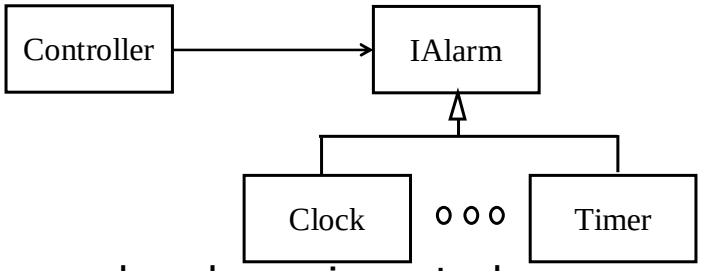
- Bad Design is one that is
  - Rigid - hard to change since changes affect too many parts
  - Fragile - unexpected parts break upon change
  - Immobile - hard to separate from current application for reuse in another

Ramifications



- Controller needs an alarm
- Clock has it, so why not use it?
- Concrete Controller depends on concrete Clock
- Changes to Clock affect Controller
- Hard to make Controller use different alarm (fails OCP)
- Clock has multiple responsibilities (fails SRP)

Alternate Design



- Dependency has been inverted
- Both Controller and Clock depend on Abstraction (IAlarm)
- Changes to Clock does not affect Controller
- Better reuse results as well

DIP

- Dependency Inversion Principle

Further Reading: [2]

"High level modules should not depend upon low level modules. Both should depend upon abstractions."

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

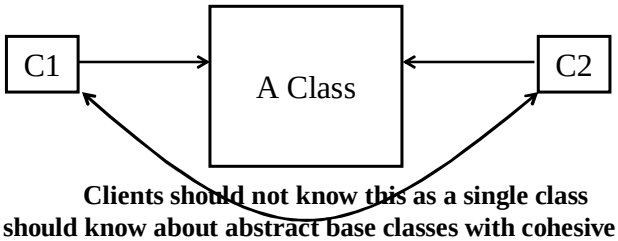
Details should depend upon abstractions."

Prudent OO Development

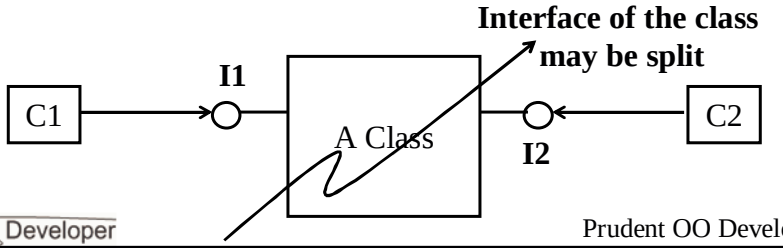
- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- **ISP**
- Conclusion

Fat Interfaces

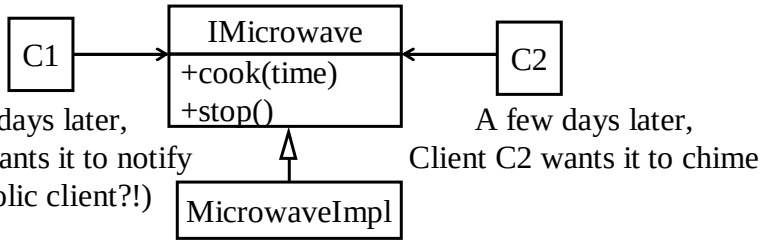
- Classes tend to grow into fat interfaces
- Examples of this can been seen in several APIs
- Less cohesive (fails SRP)



Clients should not know this as a single class
They should know about abstract base classes with cohesive interfaces

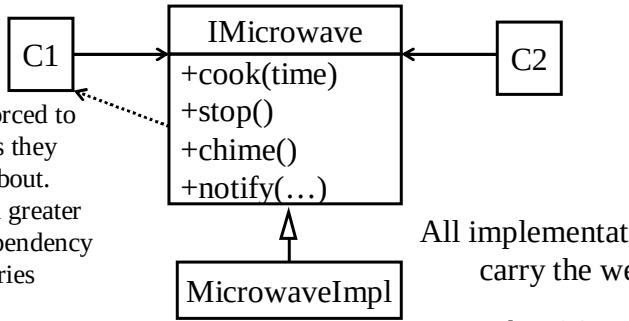


Growth of an interface



A few days later,
Client C1 wants it to notify
(workaholic client?!)

A few days later,
Client C2 wants it to chime



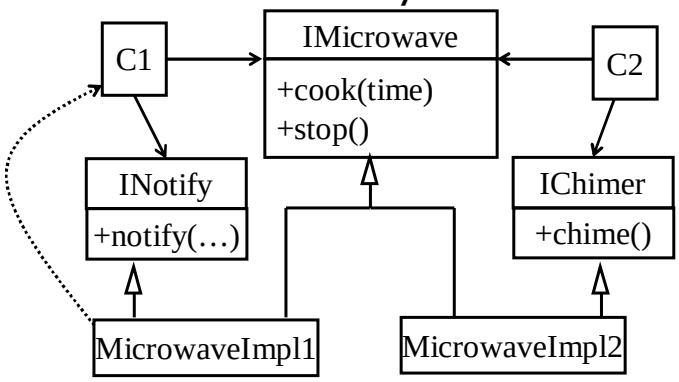
Clients are forced to
use interfaces they
do not care about.
May result in greater
coupling, dependency
to more libraries

All implementations must
carry the weights

ISP

- Interface Segregation Principle
- "Clients should not be forced to depend on methods that they do not use"

Further Reading: [2]



Quiz Time

Prudent OO Development

- Basics
- Metrics
- Object Copying
- Bad Design
- DRY
- Composition vs. Inheritance
- SRP
- OCP
- LSP
- DIP
- ISP
- **Conclusion**

Conclusion

- Developing with OO is more than
  - Using a certain language
  - Creating objects
  - Drawing UML
- It tends to elude even experienced developers
- Following the principles while developing code helps attain agility
- Use of each principle should be justified at each occurrence, however

References

1. Effective Java, Joshua Bloch

2. Agile Software Development, Robert Martin with James Newkirk & Robert Koss

3. The Pragmatic Programmer, Andrew Hunt & Dave Thomas

4. Java Design, Peter Coad & Mark Mayfield with Jonathan Kern

5. "Why copying an object is a terrible thing to do?" (downloadable from URL below)

6. http://www.AgileDeveloper.com/download.aspx