Object-Oriented Design Principles

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?

Measuring Quality of an Abstraction

- Designing Classes & Objects
  - An incremental, iterative process
  - Difficult to design right the first time

Metrics for class design

- Coupling
  - inheritance Vs. coupling
  - Strong coupling complicates a system
  - design for weakest possible coupling
- Cohesion
  - degree of connectivity among the elements of a single module/class
  - coincidental cohesion: all elements related undesirable
  - Functional cohesion: work together to provide well-bounded behavior

Law of Demeter

- "Methods of a class should not depend in any way on the structure of any class, except the immediate structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only."

First part talks about encapsulation and cohesion
Second part talks about low coupling

Tell Don't Ask

- "Procedural code gets information and then makes decisions. OO code tells objects to do things," Alec Sharp in *Smalltalk by Example*.
- Objects should take limited responsibility and rely on others to provide appropriate service
- Command-Query Separation: categorize methods as command or query

David Bock's Example on LoD/TDA

David Bocks' The Paper Boy, The Wallet, and The Law Of Demeter

Failing LoD
PaperBoy's method does:
customer.waller.totalMoney;

=> drives away shiny new Jaguar!

Honoring LoD
PaperBoy's method does:
customer.getPayment(...);

=> Customer controls amount, where it's kept (wallet, hidden in cookie jar,...)

Train Wreck Coding

- You may find code that continues to call on objects returned by methods in sequence
- customer.getAddress().getCity().getCounty()...
- This indicates that you are interested in working with objects that are farther away than those that are your close friends
- Hard to understand
- Hard to debug
- Lacks Cohesion
- Good candidate for refactoring

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

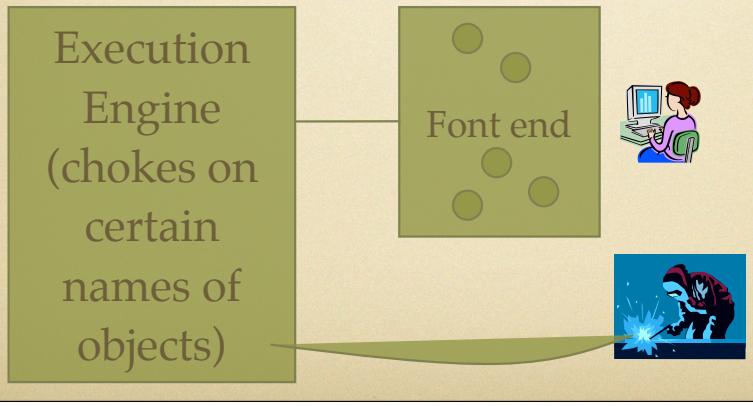
- Guiding Principles that help develop better systems
- Use principles only where they apply
- You must see the symptoms to apply them
- If you apply arbitrarily, the code ends up with Needless Complexity

YAGNI

- You Aren't Going To Need It
- You are looking out for extensibility
- You are not sure if a certain functionality is needed
- Designing what you do not know fully leads to unnecessary complexity and heavy weight design
- If you really need it, design it at that time

DRY

- Don't Repeat Yourself
- "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



Application where UI did business validation.
Start out due to flaw in Application Engine.
Once flaw was rectified, took us weeks to fix the UI due to duplication of validation logic.

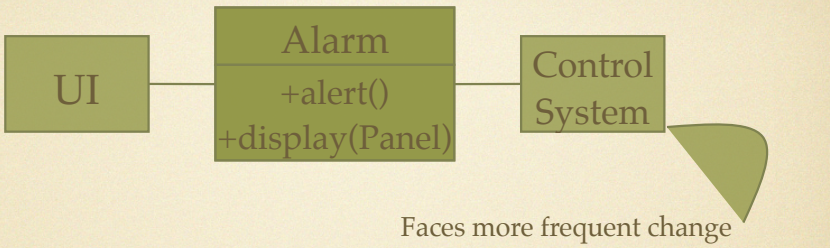
DRY

- 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

SRP

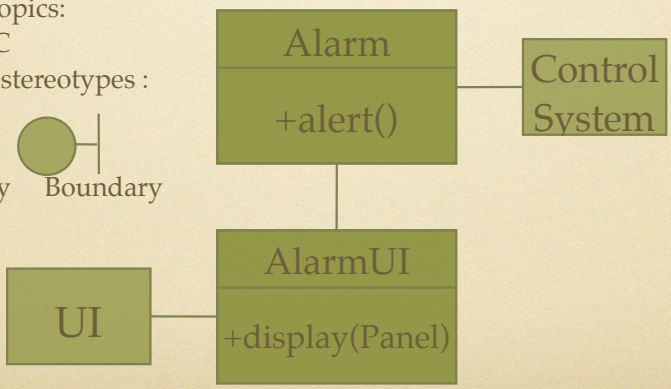
- Single-Responsibility Principle
- What metric comes to mind?
- "A class should have only one reason to change"
- Some C++ books promoted bad design
  - Overloading input/output operators!
- What if you do not want to display on a terminal any more?
  - GUI based, or web based?

SRP…



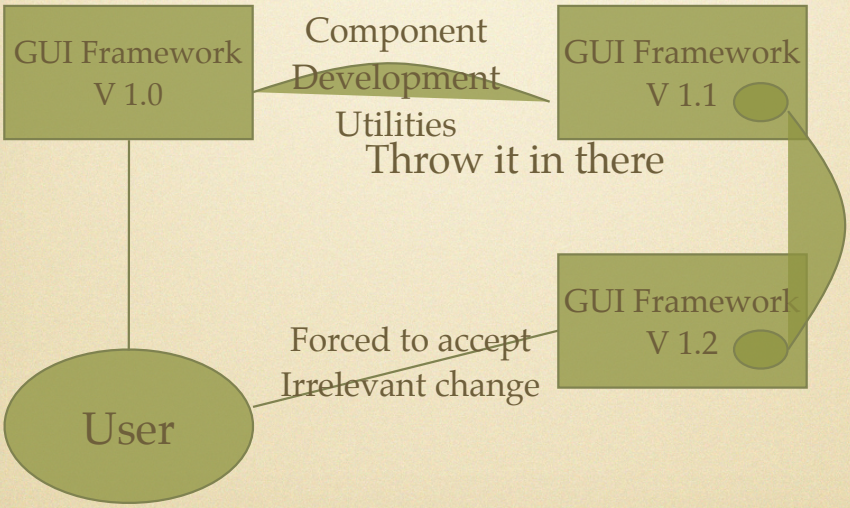
Faces more frequent change
Has greater dependency (to UI related stuff)

Related topics:
MVC
Analysis model stereotypes :



SRP at Module Level

- Can be extended to module level as well



SRP affects Reuse

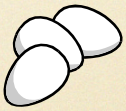
- Lower cohesion results in poor reuse
  - My brother just bought a new DVD and a big screen TV!
  - He offers to give me his VCR!
  - I have a great TV and all I need is a VCR
  - Here is what I found when I went to pickup!

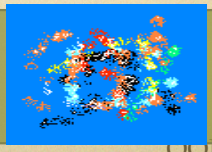
Disclaimer: This slide not intended to say anything about the brand of product shown here as an example!

Tight coupling
Poor Cohesion
Bad for reuse

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…

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

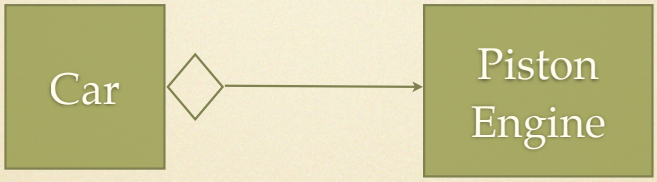
OCP…

- 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

OCP…

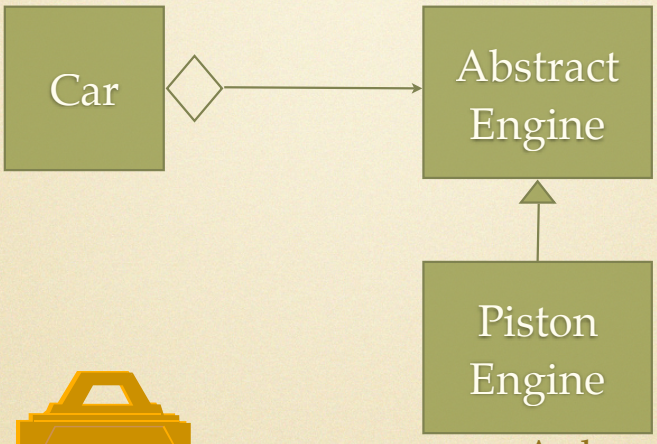
- Software Modules must
  - be open for extension
    - module's behavior can be extended
  - closed for modification
    - source code for the module must not be changed

OCP…



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

OCP…



Abstraction & Polymorphism are the Key

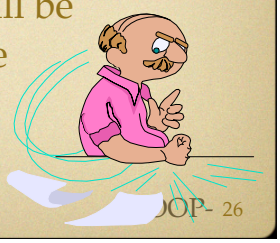
- A class must not depend on a Concrete class; it must depend on an abstract class

OCP…

- 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

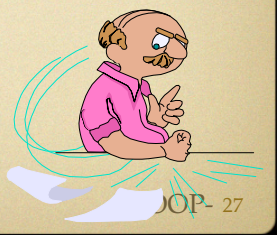
OCP…

- 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



OCP…

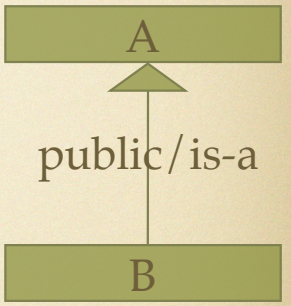
- Heuristics and Conventions that arise from OCP...
- RTTI is ugly and dangerous
  - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module
- Not all these situations violate OCP all the time



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
- What's 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"

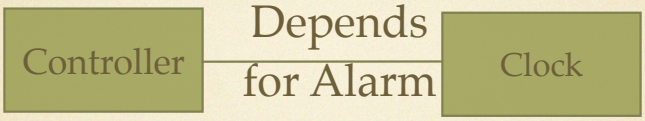
LSP in Java?

- 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

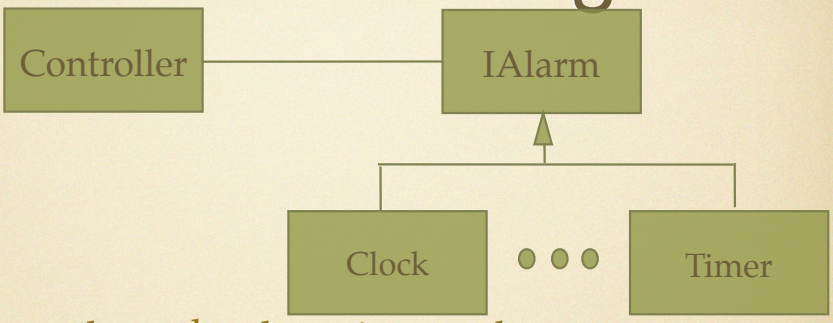
- Bad Design is one that is
  - Rigid - hard to change since changes affect too many parts
  - Fragile - unexpected parts break upon change
  - Immobile - hard to separate from current application for reuse in another

Ramifications



- 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

Inheritance Vs. Delegation

- Inheritance is one of the most abused concepts in OO programming
- Often, delegation may be a better choice than inheritance
- When should you use inheritance vs. delegation?

Inheritance Vs. Delegation...

- If an object of B may be used in place of an object of A, use inheritance
- If an object of B may use an object of A, then use delegation

DIP

- Dependency Inversion Principle
- "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."

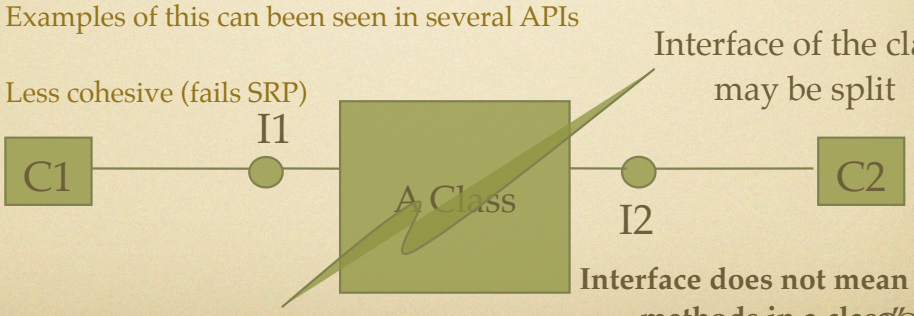
The Founding Principles

- 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

Fat Interfaces

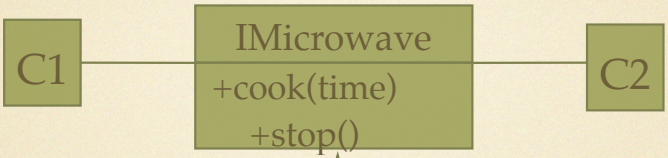
Clients should not know this as a single class
They should know about abstract base classes with cohesive interfaces

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

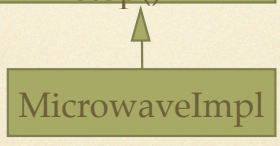


Interface does not mean "all methods in a class"

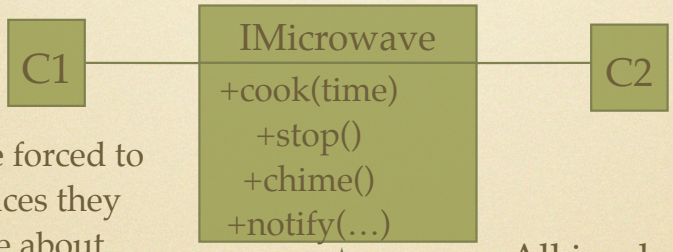
Growth of an interface



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



A few days later,
Client C2 wants it to chime

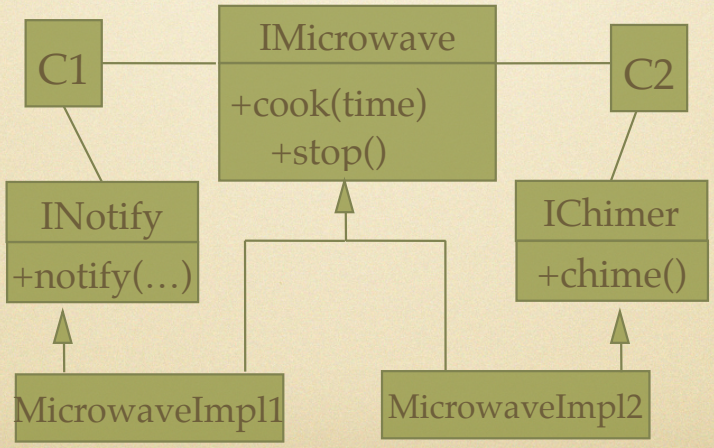


Clients are forced to use interfaces they do not care about. May result in greater coupling, dependency to more libraries

All implementations must carry the weights

ISP

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



Reuse/Release Equivalency Principle

- "The granularity of reuse is the same as the granularity of release. Only components that are released through a tracking system can be effectively reused."

Reuse/Release Equivalency Principle

- Release
  - A class generally collaborates with other classes
  - For a class to be reused, you need also the classes that this class depends on
  - All related classes must be released together

Reuse/Release Equivalency Principle

- Tracking
  - A class being reused must not change in an uncontrolled manner
  - Code copying is a poor form of reuse
- Software must be released in small chunks - components
- Each chunk must have a version number
- Reusers may decide on an appropriate time to use a newer version of a component release

Common Closure Principle

- "Classes within a released component should share common closure. If one need to be changed, they all are likely to need to be changed. What affects one, affect all."

Common Closure Principle

- A change must not cause modification to all released components
- Change must affect smallest possible number of released components
- Classes within a component must be cohesive
- Given a particular kind of change, either all classes in a component must be modified or no class needs to be modified
- Reduces frequency of re-release of component

Common Reuse Principle

- "Classes within a released component should be reused together. That is, it must be impossible to separate the component in order to reuse less than the total."

Common Reuse Principle...

- Components must be focused
- Component must not contain classes that an user is not likely to reuse
  - user may be forced to accept a new release due to changes to unused classes
- Component must be narrow

Acyclic Dependence Principle

- "The dependency structure for released component must be a Directed Acyclic Graph. There can be no cycles."

Acyclic Dependence Principle

- If there are cycles, it becomes hard to maintain
- Change ripples through
- Can't release components in small increments

Stable Dependency Principle

- "Dependencies between released components must run in the direction of stability. The dependee must be more stable than the depender."

Stable Dependency Principle

- A component can never be more stable than the one it depends upon
- Instability I = Ce / (Ca + Ce),
- where
- Ca - # of classes outside that depend upon this class
- Ce - # of classes outside that this class depends upon
- $0 \leq I \leq 1$
- 0 - ultimately stable; 1 - ultimately unstable

Stable Dependency Principle...

- Components should be arranged such that components with a high I metrics should depend upon component with low I metrics

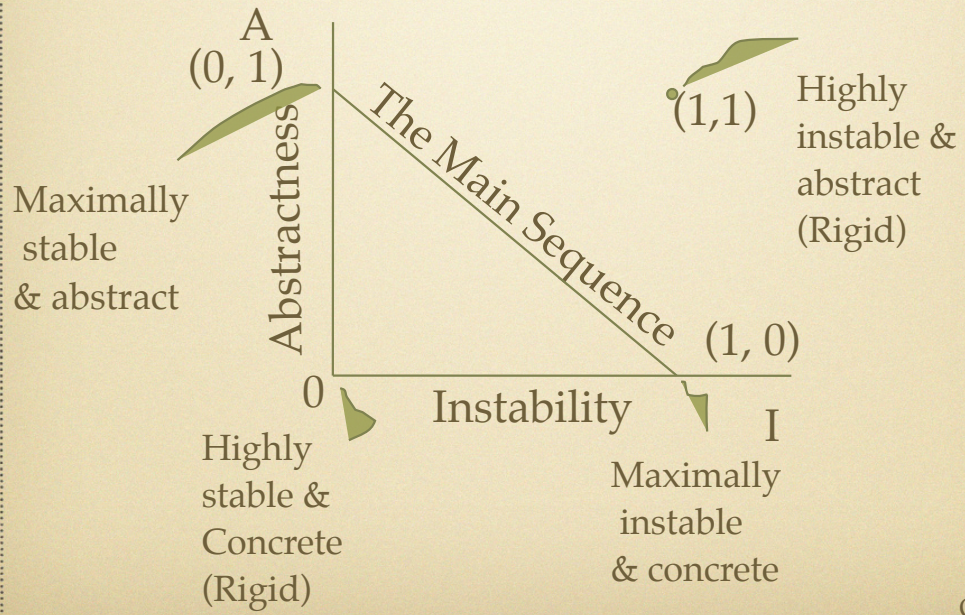
Stable Abstraction Principle

- "The more stable a component is, the more it must consist of abstract classes. A completely stable category should consist of nothing but abstract classes."

Stable Abstraction Principle

- Implementation of methods change more often than the interface
- Interfaces have more intrinsic stability than executable code
- Abstraction of a Component
- A = (# of abstract classes) / (# of classes)
- $0 \leq A \leq 1$
- 0 - no abstract classes; 1 - all abstract classes

Stability Vs. Abstractness



Distance from the main sequence

- $D = |(A + I - 1) / \sqrt{2}|$
- $0 \leq D \leq 0.707$; Desirable value of D is closed to 0
- Normalized form $D' = |(A + I - 1)|$
- Calculate D value for each component
- Component whose D value is not near Zero can be reexamined and restructured

Applying the Principles

- 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

OO Principles and Agile Development

- These principles are more relevant during iterative development and refactoring than upfront design
- As the requirements become clearer and we understand the specifics the forces behind the use of these principle become clear as well