Originality is Overrated: OO Design Principles

Kenneth M. Anderson
University of Colorado, Boulder

© University of Colorado, 2007
Lecture Goals

• Review material from Chapter 8 of the OO A&D textbook
  • Object-Oriented Design Principles
    • Open-Closed Principle
    • Don’t Repeat Yourself
    • Single Responsiblity Principle
    • Liskov Substitution Principle
    • Aggregation and Composition, Revisited
  • Discuss the examples in Chapter 8
  • Emphasize the OO concepts and techniques encountered in Chapter 8
Originality is Overrated

• Corollary: “Only Steal from the Best” — various sources

• OO A&D has been performed in various forms and in various contexts for nearly 40 years
  • Over that time, designers have developed a set of principles that ease the task of creating OO designs
  • If you apply these principles in your own code, you will be “stealing” from the best that the OO A&D community has to offer
    • The same is true of Design Patterns
OO Principles: What We’ve Seen So Far

• We’ve seen the following principles in action over the past eight lectures
  • **Classes are about behavior**
    • Emphasize the behavior of classes over the data of classes
      • Don’t subclass, for instance, for data-related reasons
  • **Encapsulate what varies**
    • Provides info. hiding, limits impact of change, increases cohesion
  • **One reason to change**
    • Limits impact of change, increases cohesion
  • **Code to an Interface**
    • Promotes flexible AND extensible code
      • Code applies across broad set of classes, subclasses can be added in a straightforward fashion (including at run-time)
New Principles

• Open-Closed Principle (OCP)
  • Classes should be open for extension and closed for modification

• Don’t Repeat Yourself (DRY)
  • Avoid duplicate code by abstracting out things that are common and placing those things in a single location

• Single Responsibility Principle (SRP)
  • Every object in your system should have a single responsibility, and all the object’s services should be focused on carrying it out

• Liskov Substitution Principle (LSP)
  • Subtypes must be substitutable for their base types
Open-Closed Principle

• Classes should be open for extension and closed for modification

• Basic Idea:
  • Prevent, or heavily discourage, changes to the behavior of existing classes
    • especially classes that exist near the root of an inheritance hierarchy
  • If a change is required, create a subclass and allow it to extend/override the original behavior
    • This means you must carefully design what methods are made public and protected in these classes; private methods cannot be extended

• Why is this important?
  • Limits impact on code that follows “Code to an Interface” principle
    • If you change the behavior of an existing class, a lot of client code may need to be updated
Example

• We’ve seen one example of the Open-Closed Principle in action
  
  • InstrumentSpec.matches() being extended by GuitarSpec and MandolinSpec
Don’t Repeat Yourself

• Avoid duplicate code by abstracting out things that are common and placing those things in a single location

• Basic Idea
  • Duplication is Bad!
  • At all levels of software engineering: Analysis, Design, Code, and Test

• We want to avoid duplication in our requirements, use cases, feature lists, etc.

• We want to avoid duplication of responsibilities in our code

• We want to avoid duplication of test coverage in our tests

• Why?
  • Incremental errors that can creep into a system, when one copy is changed but the others are not
  • Isolation of Change Requests: We want to go to ONE place when responding to a change request
Example (I)

• Duplication of Code: Closing the Door in Chapter 2

We had the responsibility for closing the door automatically in our “dog door” example originally living in the RemoteControl Class.

When we added a BarkRecognizer Class to the system, it opened the door automatically but failed to close the door.

• We could have placed a copy of the code to automatically close the door in BarkRecognizer but that would have violated the DRY principle.

• Instead, we moved the responsibility to the shared Door class.
Example (II)

• DRY is really about ONE requirement in ONE place
  • We want each responsibility of the system to live in a single, sensible place

• New Requirements for the Dog Door System: Beware of Duplicates
  • The dog door should alert the owner when something inside the house gets too close to the dog door
  • The dog door will open only during certain hours of the day
  • The dog door will be integrated into the house’s alarm system to make sure it doesn’t activate when the dog door is open
  • The dog door should make a noise if the door cannot open because of a blockage outside
  • The dog door will track how many times the dog uses the door
  • When the door closes, the house alarm will re-arm if it was active before the door opened
Example (III)

• New Requirements for the Dog Door System: Beware of Duplicates
  • The dog door should alert the owner when something inside the house gets too close to the dog door
  • The dog door will open only during certain hours of the day
  • The dog door will be integrated into the house’s alarm system to make sure it doesn’t activate when the dog door is open
  • The dog door should make a noise if the door cannot open because of a blockage outside
  • The dog door will track how many times the dog uses the door
  • When the door closes, the house alarm will re-arm if it was active before the door opened
Example (III)

• New Requirements for the Dog Door System
  • The dog door should alert the owner when something is too close to the dog door
  • The dog door will open only during certain hours of the day
  • The dog door will be integrated into the house’s alarm system
  • The dog door will track how many times the dog uses the door
• Duplicates Removed!
Example (IV)

- Ruby on Rails makes use of DRY as a core part of its design
  - focused configuration files; no duplication of information
  - for each request, often single controller, single model update, single view
- But, prior to Ruby on Rails 1.2, there was duplication hiding in the URLs used by Rails applications
  - POST /people/create # create a new person
  - GET /people/show/1    # show person with id 1
  - POST /people/update/1 # edit person with id 1
  - POST /people/destroy/1 # delete person with id 1
Example (V)

- The duplication exists between the HTTP method name and the operation name in the URL
  - POST /people/create
- Recently, there has been a movement to make use of the four major “verbs” of HTTP
  - POST == create information (create)
  - GET == retrieve information (read)
  - PUT == update information (update)
  - DELETE == destroy information (destroy)
- These verbs mirror the CRUD operations found in databases
  - Thus, saying “create” in the URL above is a duplication
In version 1.2, Rails eliminates this duplication for something called "resources"

Now URLs look like this:

- **POST /people**
- **GET /people/1**
- **PUT /people/1**
- **DELETE /people/1**

And the duplication is **logically** eliminated

- Disclaimer: … but not actually eliminated… Web servers do not universally support PUT and DELETE “out of the box”. As a result, Rails uses POST
  - **POST /people/1**
  - Post-Semantics: Delete
Single Responsibility Principle

• Every object in your system should have a single responsibility, and all the object’s services should be focused on carrying it out
  • This is obviously related to the “One Reason to Change” principle
  • If you have implemented SRP correctly, then each class will have only one reason to change
• The “single responsibility” doesn’t have to be “small”, it might be, for instance, “manage units” in Gary’s Game System Framework
• We’ve encountered SRP before
  • SRP == high cohesion
  • “One Reason To Change” promotes SRP
  • DRY is often used to achieve SRP
Return to Textual Analysis

• One way of identifying high cohesion in a system is to do the following
  • For each class C
    • For each method M
      • Write “The C Ms itself”
  • Examples
    • The Automobile drives itself
    • The Automobile washes itself
    • The Automobile starts itself
  • If any one of these sentences doesn’t make sense then investigate further.
    You may have discovered a service that belongs to a different responsibility of
    the system and should be moved to a different class
    • This may require first creating a new class before performing the move
SRP in Action

• We’ve seen SRP used in several places over the last eight lectures
  • Automatically closing the door in the dog door example
  • InstrumentSpec handling all instrument-related properties in Rick’s Guitars
  • Instrument handling all inventory-related properties in Rick’s Guitars
  • Board handling board-related services in the Game System Framework
  • Unit handling all property-related functionality in the Game System Framework
• Essentially any time we’ve seen a highly cohesive class!
Liskov Substitution Principle

• Subtypes must be substitutable for their base types

• Basic Idea
  • Instances of subclasses do not violate the behaviors exhibited by instances of their superclasses
    • They may constrain that behavior but they do not contradict that behavior

• Named after Barbara Liskov who co-authored a paper with Jeannette Wing in 1993 entitled *Family Values: A Behavioral Notion of Subtyping*
  • Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.

• Properties that hold on superclass objects, hold on subclass objects
  • Return to Rectangle/Square: $\text{WidthAndHeightMayBeDifferent}(\text{Rectangle})$ equals true for Rectangles and equals false for Square
Well-Designed Inheritance

• LSP is about well-designed inheritance
  • When I put an instance of a subclass in a place where I normally place an instance of its superclass
    • the functionality of the system must remain correct
    • (not necessarily the same, but correct)
Bad Example (I)

- The book provides an example of misusing inheritance (and violating the LSP)
  - Extend Board to produce Board3D

```
Board
width: int
height: int
tiles: Tile [*][*]

getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

Board3D
zpos: int
3dTiles: Tile [*][*][*]

getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List
```
Bad Example (II)

• But this means that an instance of Board3D looks like this:

  • Each attribute and method in bold is meaningless in this object

  • Board3D is getting nothing useful from Board except for width and height!!

  • We certainly could NOT create a Board3D object and hand it to code expecting a Board object!

  • As a result, this design violates the LSP principle

• How to fix?

<table>
<thead>
<tr>
<th>Board3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>width: int</td>
</tr>
<tr>
<td>height: int</td>
</tr>
<tr>
<td>zpos: int</td>
</tr>
<tr>
<td>tiles: Tile [ ] [ ]</td>
</tr>
<tr>
<td>3dTiles: Tile [ ] [ ] [ ]</td>
</tr>
</tbody>
</table>

getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int int)
removeUnits(int, int)
getUnits(int, int): List
getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List
Delegation to the Rescue! (Again)

• You can understand why the Game System Framework thought they could extend Board when creating Board3D
  • Board has a lot of useful functionality and a Board3D should try to reuse that functionality as much as possible
  • However, the Board3D has no need to CHANGE that functionality and the Board3D doesn’t really behave in the same way as a board
    • For instance, a unit on “level 10” may be able to attack a unit on “level 1”; such functionality doesn’t make sense in the context of a 2D board
• Thus, if you need to use functionality in another class, but you don’t want to change that functionality, consider using delegation instead of inheritance
  • Inheritance was simply the wrong way to gain access to the Board’s functionality
  • Delegation is when you hand over the responsibility for a particular task to some other class or method
Board3D now maintains a list of Board objects for each legal value of “zpos”

It then delegates to the Board object to handle the requested service

```java
public Tile getTile(int x, int y, int z) {
    Board b = boards.get(z);
    return b.getTile(x, y);
}
```

Note: book gets UML diagram wrong on page 405. The “3dTiles: Tile [*][*][*] attribute is eliminated with this new design
Another Take on Composition

• The book defines composition as
  • Composition allows you to use behavior from a family of classes, and to change that behavior at runtime
    • Their definition is essentially equivalent to the Strategy design pattern
  • Delegation is useful when the behavior of the object you’re delegating to never changes
    • Delegation is still used in composition, but the object that you are delegation to can change at run-time
• Example: Unit and Weapon
  • A unit can invoke the attack() method on its Weapon; as the game progresses, the unit may switch among different weapons at will
  • The unit is composing its “attack behavior” out of a number of Weapon instances; existence dependency applies; delete unit ⇒ delete weapons
Another Take on Aggregation

• In composition, the object composed of other behaviors owns those behaviors. When the object is destroyed, so are all of its behaviors
  • The behaviors in a composition do not exist outside of the composition itself
• If this is not what you want, then use aggregation: composition without the abrupt ending
  • Aggregation is when one class is used as part of another class, but still exists outside of that other class
  • The book uses an example of a Unit that can arrive at a building and leave its weapons there in storage, the relationship between Unit and Weapon is now an aggregation
Implication: Use Inheritance Sparingly

• Delegation, composition, and aggregation all offer alternatives to inheritance when you need to reuse the behavior of another class
  
  • Only use inheritance when
    
    • an IS-A relationship exists between the superclass and the subclass
    
    • AND the subclass behaves like a superclass (i.e. maintains the properties of the superclass in its behavior)
  
• If you favor delegation, composition, and aggregation **over** inheritance, your software will usually be more flexible and easier to maintain, extend, and reuse
  
  • This was the subject of a religious war during the 90s
    
    • Unlike “emacs vs. vi”, the war is over and delegation won!
Wrapping Up

• We’ve added four new OO principles to our toolkit
  • Apply these principles and you’ll see a marked increase in the flexibility and extensibility of your OO designs
  • Indeed, one of the “secrets” of design patterns is that they invariably lead to code that exhibit these principles
• We’ve also seen that inheritance is a tool to be used sparingly
  • Favor delegation, composition, and aggregation to gain run-time flexibility
  • Use inheritance when the subclass’s semantics and behavior fit neatly with its superclass
Coming Up Next

• Lecture 14: Testing And Iterating
  • Read Chapter 9 of the OO A&D book
• Lecture 15: Putting It All Together
  • Read Chapter 10 of the OO A&D book