CMPS 115 Winter 04

Class #10 (2004/02/05)

- Changes/Review
- Programming Paradigms
- Principles of OOD
- Design Patterns
Changes & Lecture 9 Takeaway

- Changes/Notices
  - 2/26 may be guest after all, on design-with-frameworks
  - New Final policy: 75% cutoff, is 100 pts

- Takeaway lecture 8 - Modularization
  - Module is named block
  - Module should hold together (*cohesion*) (7 levels)
    - OO/ADT basis is “informational cohesion”
      Set of actions, each with own entry/exit,
      independent code for each, but all operate on same data structure
  - Modules should stand apart (*coupling*) (5 levels)
    - OO/ADT should use “data coupling”
    - Objects-as-params can hide tramp data, make “stamp” into “data”
    - Common coupling happens at DB in business systems
Coupling Example

<table>
<thead>
<tr>
<th>number</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aircraft type</td>
<td>status flag</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>list of aircraft parts</td>
</tr>
<tr>
<td>3</td>
<td>function code</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>list of aircraft parts</td>
</tr>
<tr>
<td>5</td>
<td>part number</td>
<td>part manufacturer</td>
</tr>
<tr>
<td>6</td>
<td>part number</td>
<td>part name</td>
</tr>
</tbody>
</table>

p, t, and u access the same database in update mode

Wouldn't this make a great exam question?
Programming Paradigms

- **Procedural**
  - stepwise refinement, seek optimal local algorithms

- **Modules**
  - hide data or design; functional abstractions
  - modularize as you go (top down) or when at statement level (bottom-up)

- **ADT**
  - form types with complete set of ops

- **Object** - organize the ADTs
  - make commonality explicit
Programming as an art (Knuth)

while (programming == art) {
    incr( pleasure );
    decr( bugs );
    incr( portability );
    incr( maintainability );
    incr( quality );
    incr( salary );
} // live happily ever after

• No inherent state
• Heavily type-dependent
Modules As Early Paradigm

- As programs get more complex, need data abstraction and firewalls to design changes.
- Modules introduced as syntactic mechanism for separating interface from implementation.
- No semantics, no “how to define” guidance … no expression of meaning.
- No module relationship other than “uses”.
Abstract Data Type

- Module that represent one “type” of data
  - characterized by set of operations
  - amenable to formal analysis
- Encapsulation allows
  - initialization of ADT instance
  - protection of ADT instance sanity
  - safe coercions of range and domain (stack of int, stack of long, …)
- Starts to get at “meaning” but turns out to be inflexible
Object Orientation

- Similar to ADT, being a set of operations ("class") with state encapsulation
- Add inheritance to express "what’s the same" between classes
- Inherited operations in different subclasses need to differ in implementation (polymorphism)
- Instances of class are "objects" and have own state as well as (some languages) shared state
- operations are invoked by messages with dynamic binding for polymorphism
OOP vs OOA/OOD

- OO Analysis is modeling the problem in terms of communicating objects.
- OO Design is modeling the problem (as defined by the OOA) in terms of behavior (specific methods, grouped into interfaces) and essential data allocated to *classes*.
- Object Oriented Programming (OOP) is implementing the OOD ... preferably in a language with good support for OO mechanisms (encapsulation, inheritance, polymorphism).
Some OOA
from Coad/Nicola, OOP

- “I’m alive!” Principle …
  - talk in first person; does it make sense to say “I know my own ___ and I can ___ myself.”
  - “I’m a Counter; I know my own value and I can increment myself. Tell me increment, and I’ll do it”

- “Read it again, Sam!”
  - does it sound funny read aloud?
  - a “counter” works on a value - it is a value manager

- In object-think, avoid managers
  - creates functional decomposition
The “-er-er” Principle

- During analysis, challenge any class name that ends in “-er.”
  - if it has no parts, change the name of the class to what each object is managing
  - if has parts, put as much work in the parts that the parts know enough to do themselves (match to data)

- Manager objects are just part of intermediate object think

- More advanced object think OOA stays closer to domain
The “amount of think” Hierarchy

- Low
  - applying functional decomposition rather than an object-oriented decomposition

- Intermediate
  - managers and data encapsulators; all work in manager, little for subordinate to do

- High
  - *Objects* at work! Putting each action in the object that knows enough to carry it out
  - group essential data; allocate behavior accordingly. Responsibility follows capability.
Say it in the First Person

- I increment and decrement myself. What am I?
- I am a *Count*!
  - increment, decrement - these are what I do; they are the *services* I provide
  - the value I have know, the value I reset to; these are my *attributes*
- This habit really helps develop good OO instincts
Some OO Design Principles

- Open-Closed (open for extension; closed for modification)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Law of Demeter (LoD)
- Design By Contract (DBC)
Open-Closed Principle (Bertrand Meyer)

*Software entities should be open for extension, but closed for modification.*

- Open for extension: behavior can be changed to suit new requirements
- Closed for modification: module itself doesn’t change

- **Corollary: Single Choice Principle**
  - *Whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives*
Open-Closed Mechanisms

- How to change behavior without editing source?
  - Creating abstractions and deriving concrete classes from those abstractions so abstract modules are open for extension, yet closed to modification
  - Using abstract modules in other code, so can change other code behavior by supplying different concrete module
  - See also
    http://www.objectmentor.com/resources/articles/ocp.pdf
Liskov Substitution Principle (LSP)

If for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$ then $S$ is a subtype of $T$.

$\Rightarrow$ Modules that use references to base types must be able to use references to derived types without knowing the difference

class Rect {
    width, height, area   // attributes
}

class Square extends Rect{}
LSP: Is a Square a Rectangle?

Rect r = new Rect();
    setWidth = 4;
    setHeight=5;
    assert(20 == getArea());

class Square extends Rect{
    // Square invariant, height = width
    setWidth(x) {setHeight()=x}
    setHeight(x) {setWidth(x)}
} // violate LSP?
Single Responsibility Principle

- There should never be more than one reason for a class to change.

Modem.java -- SRP Violation
interface Modem
{
public void dial(String pno);
public void hangup();
public void send(char c);
public char recv();
}
Interface Segregation Principle

- Some classes with good informational cohesion support many many many services/operations
- Most clients only need subset of services; this is like “tramp data!”
- So, split class interface into multiple interfaces such that most clients use all the operations they see, and don’t see operations they don’t need
Law of Demeter Defined

*Only talk to your immediate friends*

For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:

1. The argument classes of M (including C).
2. The instance variable classes of C.

(Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.)
Two Forms of LoD

- **Strong LoD:**
  - The Strong LoD defines instance variables as being ONLY the instance variables which make up a given class. Inherited instance variable types may not be passed messages.
    - Changes to data structure only affect methods on changed classes
    - Tends to have more methods

- **The Weak LoD:**
  - The Weak LoD defines instance variables as being BOTH the instance variables which make up a given class AND any instance variables inherited from other classes.
LoD Programming Style

- In class PaperBoy, do not use
  `customer.getWallet().getCash(due);`
rather, use
  `customer.getPayment(due);`
then in `Customer.getPayment (due), use`
  `wallet.getCash(due)`

- Benefits - easier analysis (structural induction)
- Tradeoff - more methods, but smaller/easier
- Experiments have show significantly improved maintainability
**BREAK**

W/E 10 Feb:
W: 2 Astrix for sharing paper proto

<table>
<thead>
<tr>
<th></th>
<th>27-Jan</th>
<th>3-Feb</th>
<th>10-Feb</th>
<th>17-Feb</th>
<th>24-Feb</th>
<th>2-Mar</th>
<th>9-Mar</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrix</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>BattleSnake</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Caesar's</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td><strong>CLASS TOT</strong></td>
<td><strong>9</strong></td>
<td><strong>17</strong></td>
<td><strong>19</strong></td>
<td><strong>19</strong></td>
<td><strong>19</strong></td>
<td><strong>19</strong></td>
<td><strong>19</strong></td>
<td><strong>19</strong></td>
</tr>
</tbody>
</table>

Instructor awarded 4/6
TA awarded 4/9
Averaging 8.5/wk * 7 weeks = 60 BP … need 80 BP

**NO FREE PIZZA AT THIS RATE**
OO Design Guidelines
Favor Composition Over Inheritance

- Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called *aggregation* or *containment*, although some authors give special meanings to these terms. For example:
  - Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
  - Aggregation - when one object has a collection of objects that can exist on their own (UML)
  - Containment - a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object (Coad)
Coad’s Rules

- Use inheritance only when all of the following criteria are satisfied:
  - A subclass expresses "is a special kind of" and not "is a role played by a" (no Manager isa Employee)
  - An instance of a subclass never needs to become an object of another class
  - A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
  - A subclass does not extend the capabilities of what is merely a utility class
  - For a class in the actual Problem Domain, the subclass specializes a role, transaction or device
Inheritance/Composition Example 1

Diagram:
- Person
  - Name
  - Address
- Passenger
  - Frequent Flyer ID
  - Reservation
- Agent
  - Password
  - Authorization Level
- Agent Passenger
I/C Example 1 Problems

1. "Is a special kind of" not "is a role played by a"
   - Fail. A passenger is a role a person plays. So is an agent.

2. Never needs to transmute
   - Fail. A instance of a subclass of Person could change from Passenger to Agent to Agent Passenger over time

3. Extends rather than overrides or nullifies
   - Pass.

4. Does not extend a utility class
   - Pass.

5. Within the Problem Domain, specializes a role, transaction or device
   - Fail. A Person is not a role, transaction or device.

   Inheritance does not fit here!
I/C Example 1 Fixed

Composition to the rescue!
I/C Example 2
Inheritance/Composition Example3

"Is a special kind of" not "is a role played by a"

Fail. A reservation is not a special kind of observable.

Never needs to transmute

Pass. A Reservation object stays a Reservation object.

Extends rather than overrides or nullifies

Pass.

Does not extend a utility class

Fail. Observable is just a utility class.

Within the Problem Domain (PD), specializes a role, transaction or device

Not Applicable. Observable is a utility class, not a PD class

*Inheritance does not fit here!*
Program To An Interface, Not An Implementation (class)

- An object can have many interfaces. (Essentially, an interface is a subset of all the methods that an object implements).

- A type is a specific interface of an object

- Different objects can have the same type and the same object can have many different types

- An object is known by other objects only through its interface

- In a sense, interfaces express "is a kind of" in a very limited way as "is a kind of that supports this interface"

- Interfaces are the key to pluggability!
Design Patterns and Idioms

- Three-part rule with context, problem/forces, and solution that resolves forces

- Good patterns are:
  - Problem solutions, not abstract principles
  - Proven (3 successes track record; not theories)
  - Non-obvious
  - Relationship descriptions (capture collaborations)

- Catalogs (GoF) for codifying vocabulary

- Nice “Nutshell” -
Patterns and Idioms

- **Software Patterns** - “Descriptions of communicating objects and classes that are customized to solve a general design problem, in a context.” - [Gamm1]
  - Intent - “A vehicle for conveying expertise” - [Plop2]

- **Language Idioms** - The realisation of a particular pattern in a given language environment.

- “Idioms were an early precursor of patterns, conveying key design insights for a specific language or technology. Today idioms retain the original sense of language dependence...” [Plop2]
Patterns

- GoF (Gamma, Helm, Johnson, Vlissedes)
  - mid-range in abstraction; not “frameworks” or “architectures”

- Resources
  - patterndigest.com
  - Hillside.net (see http://hillside.net/patterns/onlinepatterncatalog.htm)
  - Nice Nutshell:
  - Free Book (lots of code & text, no UML)
### Design Patterns from Gang of Four

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Command</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td>Factory</td>
<td>Composite</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Prototype</td>
<td>Decorator</td>
<td>Iterator</td>
</tr>
<tr>
<td>Singleton</td>
<td>Facade</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Template</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>

*Command is a common pattern*

*Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides (Addison Wesley)*
Command Pattern (example)

Encapsulate commands in objects so that you can control their selection, sequencing, queue them, undo them and otherwise manipulate them.
Mapping Pattern to Idiom

Java idioms: http://c2.com/cgi/wiki?JavaIdioms