Lecture 30: OO Design Heuristics

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2003

Credit is where Credit is Due

- Some material for this lecture is taken from
  - Object-Oriented Design Heuristics
    - by Arthur J. Riel
    - ISBN: 0-201-63385-X
  - as such, it is copyright, © 1999, by Addison Wesley

This Lecture

- Cover OO Design Heuristics
  - Classes and Objects
  - Topologies of Procedural versus Object-Oriented Applications
  - Relationships between Classes and Objects
  - The Inheritance Relationship
  - Multiple Inheritance

Typical Problem

- I have created an OO design for my system
  - Is it good?
  - Bad?
  - Somewhere in between?
- Ask an OO “guru”
  - A design is good when “it feels right”
- So, how do we know when it feels right?

One Approach: Design Heuristics

- “The guru runs through a subconscious list of heuristics, built up through his or her design experience, over the design. If the heuristics pass, then the design feels right, and if they do not pass, then the design does not feel right”
- from Object-Oriented Design Heuristics
  - by Arthur J. Riel

Riel’s Take

- We would be in a sorry state if we depended on designers to gain heuristics only through experience
- Riel’s book documents 61 heuristics that
  - he has developed working as a faculty member at Northeastern University
  - and as a consultant on real-world OO A&D software development projects
- Lets take a look at a some of these heuristics

Note on Heuristics

- Not all heuristics work together
  - Some are directly opposed!
- This occurs because there are always trade-offs in analysis and design
  - Sometimes you want to make a change to reduce complexity…this may have the consequence that it also reduces flexiblity
  - You will have to decide which heuristic makes the most sense for your particular context

Classes and Objects

- Heuristics
  - *All data should be hidden within its class*
    - When a developer says
      - “I need to make this piece of data public because…”
    - They should ask themselves
      - “What is it that I’m trying to do with the data and why doesn’t the class perform that operation for me?”
  - *Users of a class must be dependent on its public interface, but a class should not be dependent on its users*
    - Why?

Classes and Objects, continued

- Heuristics
  - *Minimize the number of messages in the protocol of a class*
    - The problem with large public interfaces is that you can never find what you are looking for…smaller public interfaces make a class easier to understand and modify
  - *Do not put implementation details such as common-code “helper” functions into the public interface of a class*
    - Users of a class do not want to see operations in the public interface that they are not supposed to use

Classes and Objects, continued

- Heuristics
  - *Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class*
- This resonates with what we have seen before on coupling earlier in the semester
  - nil coupling: no coupling
  - export coupling: make use of public interface
  - overt coupling: make use of private details

Classes and Objects, continued

- Heuristics
  - *A class should capture one and only one key abstraction*
    - e.g. a class should be cohesive; Riel defines “key abstraction” as an element of the problem domain
  - *Keep related data and behavior in one place*
    - Similar to the “Move Method” refactoring pattern
  - *Spin off non-related information into another class*
    - Similar to the “Extract Class” refactoring pattern (not covered)
  - *Most of the methods defined on a class should be using most of the data members most of the time*
- All of these heuristics deal with class cohesion

Topologies of Procedural vs. OO Applications

- These heuristics help you identify the use of non-OO structures in OO Applications
  - Procedural topologies break an application down by functions, which then share data structures
    - while it is easy to see which functions access which data structures, it is difficult to go the other way, to see which data structures are used by which functions
    - The problem: a change to a data structure may have unintended consequences because the developer was not aware of all the dependencies on the data structure

Typical problems

- There are two typical problems that arise when developers familiar with procedural techniques try to create an OO design
  - The God Class
    - A single class drives the application, all other classes are data holders
  - Proliferation of Classes
    - Problems with modularization taken too far

OO Topologies

- Heuristics (God Class)
  - *Distribute system intelligence horizontally as uniformly as possible, that is, the top-level classes in a design should share the work uniformly*
  - *Do not create god classes/objects in your system. Be very suspicious of a class whose name contains "Driver", "Manager", "System", or "Subsystem"*

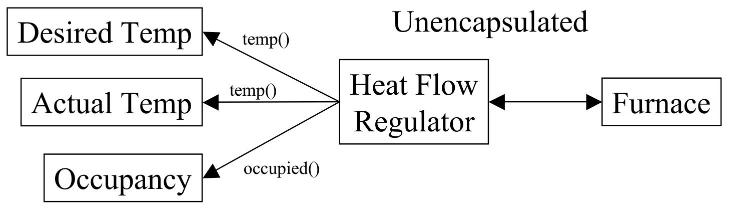
OO Topologies

- Heuristics (God Class)
  - *Beware of classes that have many accessor methods defined in their public interface. Having many implies that related data and behavior are not being kept in one place*
  - *Beware of classes whose methods operate on a proper subset of the data members of a class. God classes often exhibit this behavior*

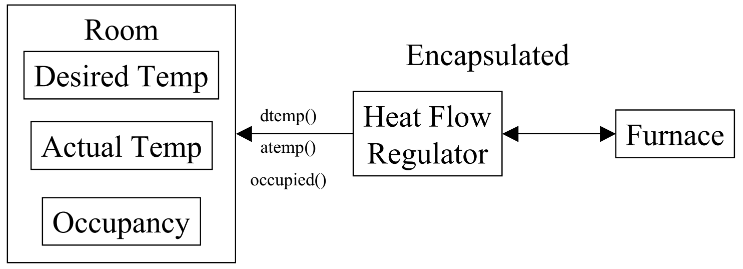
OO Topologies, continued

- God Class Example
  - A heat flow regulator needs to decide when to activate a furnace to keep a room at a certain temperature
  - Consider the following three designs
    - Unencapsulated
    - Encapsulated
    - Distributed Intelligence

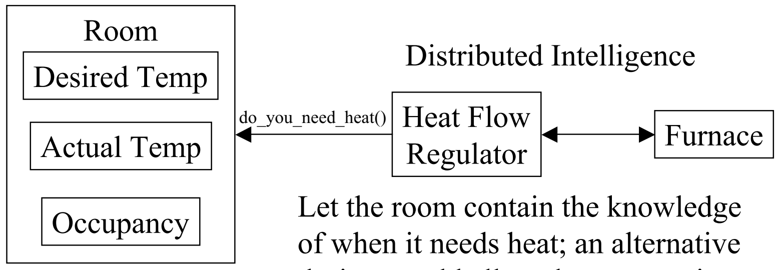
God Class Example



God Class Example, cont.



God Class Example, continued



Let the room contain the knowledge of when it needs heat; an alternative design would allow the room to just ask the regulator for heat when do_you_need_heat() is true

OO Topologies

- Heuristics (Proliferation of Classes)
  - *Eliminate irrelevant classes from your design*
    - principle of domain relevance
    - often only have get, set, and print methods
  - *Eliminate classes that are outside the system*
    - principle of domain relevance again
  - *Do not turn an operation into a class*.
    - Be suspicious of any class whose name is a verb or is derived from a verb, especially those that have only one piece of meaningful behavior.
    - Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class

Relationships between Classes and Objects

- Heuristic
  - *Minimize the number of classes with which another class collaborates*
    - Look for situations where one class communicates with a group of classes; Ask if its possible to replace the group with a class that contains the group
  - This heuristic is obviously related to coupling and its supporting what we have said earlier this semester: aim for systems whose component parts are highly cohesive and loosely coupled

Relationships between Classes and Objects, continued

- Heuristic
  - *If a class contains objects of another class, then the containing class should be sending messages to the contained objects*
    - that is a containment relationship should always imply a uses relationship
- Related
  - *Classes should not contain more objects than a developer can fit in short-term memory.*
  - *A class must know what it contains, but it should not know its container (do not depend on your users)*

Inheritance Relationship

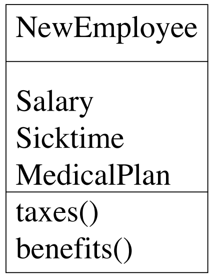
- Important not to confuse inheritance and containment
- Heuristics
  - *Inheritance should be used only to model a specialization hierarchy*
    - Containment is black-box
    - Inheritance is white-box
  - *Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes*
  - *All data in a base class should be private; do not use protected data*

Inheritance Relationship, continued

- Heuristics
  - *In theory, inheritance hierarchies should be deep—the deeper, the better*
    - In practice, inheritance hierarchies should be no deeper than an average person can keep in short-term memory.
  - *All abstract classes must be base classes*
    - You can’t make instances of an abstract class, so you need subclasses in order to access any functionality provided by the abstract class
  - *Factor the commonality of data, behavior, and/or interface as high as possible in a class hierarchy*
  - *All base classes should be abstract classes*

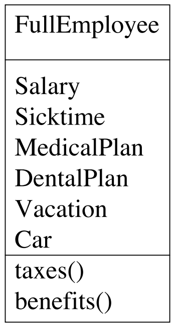
Example (to explain last heuristic)

- Consider a start up company…
  - they need a class to store information about employees



Six Months Later

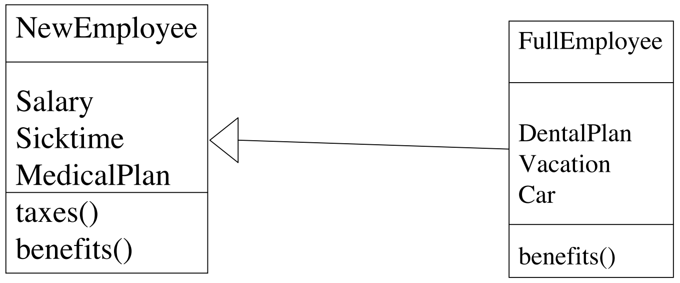
- The company decides to make a distinction between new employees and employees that have been with the company for six months



We notice that the full employee is just a special case of the new employee

so…

Lets use inheritance

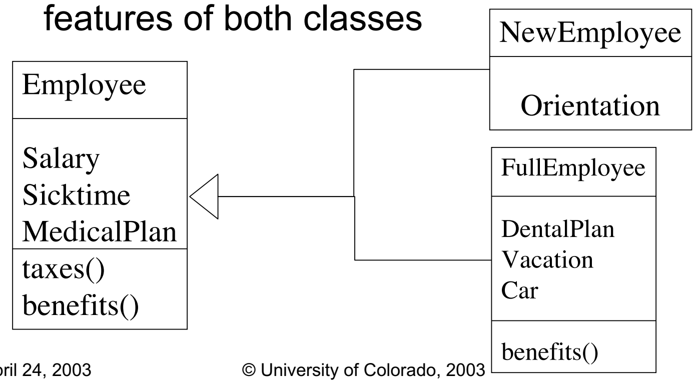


Adding to NewEmployee

- Assume we decide that all new employees should go to an orientation session
  - we want to add an attribute to track whether an employee has attended the session
  - Can we add this attribute without adding it to the FullEmployee class? (Full Employees either do not need the orientation session or already had it)
    - The answer is no! (because full employee is a subclass of new employee)
    - This is the danger of inheriting from a concrete class
      - (which is the fear that the specialization link between the two classes will not hold up under extension or refinement of the design)
      - earlier this semester, we referred to this as the “fragile base class” problem

The solution

- Have both classes inherit from an abstract base class, that captures the common features of both classes



Ramifications

- If you violate this heuristic, as we did with this example, you may (probably will) end up in a situation where you need to shift to the abstract base class design
  - Then, you need to introduce a new class, refactor, and change NewEmployee references to Employee references, except when access is needed to the new “orientation” attribute
- Note, also, that this problem of an employee being in two different states, is perhaps better solved using the State design pattern

Multiple Inheritance

- Riel does not advocate the use of multiple inheritance (its too easy to misuse it). As such, his first heuristic is
  - *If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise!*
- Most common mistake
  - Using multiple inheritance in place of containment

Multiple Inheritance

- A Second Heuristic
  - *Whenever there is inheritance in an object-oriented design, ask yourself two questions:*
  - *1) Am I a special type of the thing from which I’m inheriting? 2) Is the thing from which I’m inheriting part of me?*
- A yes to 1) and no to 2) implies the need for inheritance; A no to 1) and a yes to 2) implies the need for composition
  - Is an airplane a special type of fuselage? No
  - Is a fuselage part of an airplane? Yes

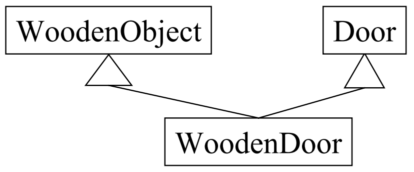
Multiple Inheritance

- A third heuristic
  - *Whenever you have found a multiple inheritance relationship in an object-oriented design, be sure that no base class is actually a derived class of another base class*
- Otherwise you have what Riel calls accidental multiple inheritance
  - Consider the classes “Citrus”, “Food”, and “Orange”; you can have Orange multiply inherit from both Citrus and Food…but Citrus is-a-kind-of Food, and so the proper hierarchy can be achieved with single inheritence

Multiple Inheritance

- So, is there a valid use of multiple inheritance?
  - Yes, subtyping for combination
    - It is used to define a new class that is a special type of two other classes where those two base classes are from different domains

Multiple Inheritance Example



Is a wooden door a special type of door? Yes
Is a door part of a wooden door? No
Is a wooden door a special type of wooden object? Yes
Is a wooden object part of a door? No
Is a wooden object a special type of door? No
Is a door a special type of wooden object? No
All Heuristics Pass!

What’s Next?

- Possibly one more (short) lecture on OO Heuristics
  - for the first half of Tuesday’s lecture
- Then, Review for Final
  - Final is cumulative
  - For in-class students, next Saturday, May 3rd at 4:30 PM in this class
  - For CATECS students, I’ll be sending the exam to your test proctor next week; your exam needs to be postmarked by May 10th