CS 494
Object-Oriented Analysis & Design

Evaluating Class Diagrams

- **Topics include:**
  - **Cohesion, Coupling**
  - **Law of Demeter (handout)**
  - **Generalization and specialization**
  - **Generalization vs. aggregation**
  - **Other aggregation issues**

Cohesion

- How diverse are the things inside an “entity”
  - A what? Module, function,… In OO a class.
- What’s this mean?
  - Class should represent a single abstraction
  - Or, it should address a single general responsibility

Problems Created by Bad Cohesion

- Hard to understand the class
- If two abstractions grouped into one class, that implies a one-to-one relationship
  - What if this changes?
- Often we specialize a class along a dimension
  - This new thing is like the existing one except we extend it in one area (dimension)
  - Problems arise when each of the several abstractions need such specialization

The “Multiplicity” Problem

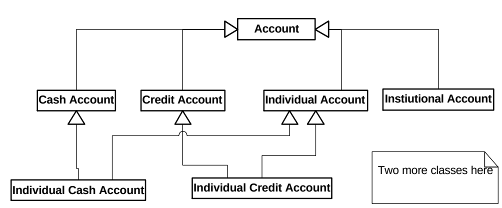
- Consider an Account class that holds:
  - Customer name, address, tax ID, Account status, etc.
- What if one customer needs two accounts?
  - Two Account objects, but each stores name and address
- What if one account has two owners?
  - You can’t do this, unless you create a collection in each Account to hold owner info

Specializing along Dimensions

- Let’s say we need variations on class Account
  - First, based on account type: Cash Account, Credit Account
  - Second, based on customer type: Individual Account, Institutional Account
- These are two dimensions, but are they mutually exclusive?
  - We often compose along two dimensions
  - E.g. Individual Cash Account, Individual Credit Account, etc.
- Specialization often implemented as inheritance: Do we want multiple inheritance?

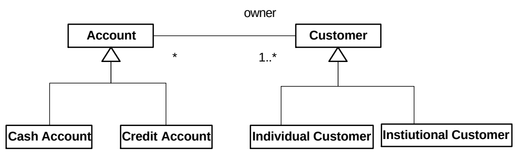
Inheritance Diamonds

- **Structures like this cause messy problems!**



Separating Abstractions

- Composition across dimensions achieved by aggregation
- You can see how this improves earlier problem too



How to Achieve Better Cohesion

- Some of this is just good OO experience
- We can learn from database normalization
  - Eliminate redundancy
  - Attributes should have a single value and should not have structure (repeating groups of things)
  - Attributes always describe an instance of its containing class
    - That's what attributes are all about! State values that define a particular instance
- Note: there are always tradeoffs! Sometimes we combine abstractions into one class for efficiency.

Coupling

- How dependent an object/class is on the world around it
  - How many connections
  - Nature of the connections
  - Will changes cause a "ripple effect"?
- Our goals:
  - Reduce coupling if possible
  - Improve nature of necessary coupling

Forms of Coupling (from Richter)

- Identity Coupling
  - An object contains a reference or pointer to another object
  - Eliminate associations or make them one-way
- Representational Coupling
  - An object refers to another through that object's interface
    - How it does this affects the degree of coupling

Forms of Coupling (cont'd)

- Subclass Coupling
  - Object refers to another object using a subclass reference for that object
    - Not the more general superclass
  - A client should refer to the most general type possible
    - Why? Subclasses may be added later, possibly by someone else
  - Try to write code that minimizes dependencies on subclass details
    - Instead rely on the common interface defined in the superclass
    - Factory patterns for creation

Interfaces

- Java's interfaces; C++ classes with pure virtual functions and no data members
- Interfaces define a role not a class-abstraction
  - Many classes can pay that role
- We can define a function parameter or pointer in terms of the role (interface) instead of the class type

Forms of Coupling (cont'd)

- Inheritance coupling
  - A subclass is coupled to its superclass at compile-time
    - In general, prefer late to early
  - Seems like the only way to do things, but ask:
    While the program executes, does an object need to change its subclass?
  - Aggregation is supported at run-time
  - Examples:
    - PARTS: subclasses for Manager and Employee?

Generalization/Specialization: When?

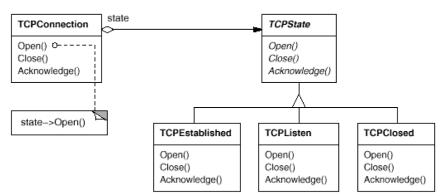
- Why <u>might</u> you choose to introduce a super-class?
  - A true "Is-A" relationship in the domain model
  - Two or more classes share a common implementation
    - Rule: "Write once!"
  - Two or more classes share a common interface
    - Can use super-class as parameter type
    - <u>But</u> interfaces solve this problem too
- A subclass specializes its super-class by one of:
  - Adding state info in terms of an attribute
  - Adding state info in terms of an association
  - Adding behavior in terms of a new method
  - Replacing behavior by overriding a method

Generalization/Specialization: When Not?

- **Avoid distinctions based on state of an instance**
- **Why? Objects change state!**
- **Solutions:**
  - **Replace with aggregation. How?**
  - **Factor out state-specific "extensions" to attributes and operations into a second object**
  - **Attach that object as needed**
- **Example: the *State design pattern* from the Gang of Four book**

The State Design Pattern

- **A connection can be in various states**
  - **Handles requests differently depending on state**
- **Connection delegates requests to its state object**
  - **Which changes dynamically**



Generalization/Specialization: When Not? (2)

- **Concrete super-classes are often a bad idea**
  - **Consider example of Manager as a sub-type of Employee**
  - **Implies Managers inherit <u>every</u> property of Employee**
    - **Nothing can be unique to non-Managers!**
- **Reminder: Specialization along multiple dimensions is often better done with aggregation**
  - **See earlier example of Account and Customer**

Generalization/Specialization: When Not? (3)

- **Where to place properties that are common to some but not all subclasses?**
  - **This can get ugly! (Example in Richter's textbook, Sect 4.3.2)**
  - **Intermediate subclasses? Mix-in classes? Helper classes?**
- **Do not (repeat, do not) use multiple inheritance when it's really aggregation**
  - **Ask the "Is-A" question.**
  - **Liskov substitutability principle:**
    - **An instance of a child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent class if substituted in a similar situation.**

Coad's Five Criteria for When to Inherit

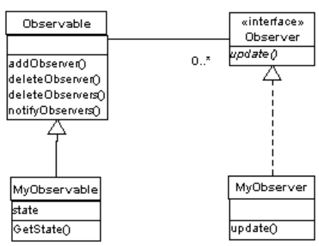
- **Peter Coad in book *Java Design***
- **Encapsulation is weak within a class hierarchy**
- **Only use inheritance when:**
  - **"Is a special kind of", not "Is a role played by".**
  - **Never need to transmute an object to be in some other class**
  - **Extends rather than overrides or nullifies**
  - **Does not subclass what is merely a utility class**
  - **For problem domain (PD) objects, it is a special kind of role, transaction, or thing**

Example: Java's Observer/Observable

- **Coad argues Java's implementation of this design pattern is poor.**
- **Observer interface:**
  public interface Observer {
  void update (Observable observed,
  Object argument); }
- **Observable superclass. Has operations:**
  addObserver(), deleteObserver(),
  deleteObservers(), notifyObservers()

Class Diagram: Observable/Observer

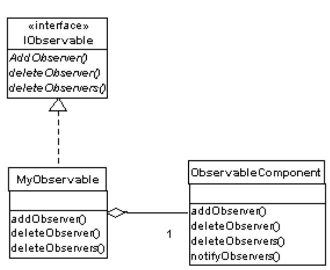
- **Top two classifiers from java.util**



Java's Observable Superclass

- **Does it meet Coad's criteria? No!**
  - **We're subclassing a utility class**
- **A better implementation:**
  - **Make Observable an interface (not a superclass)**
  - **Factor out the "observable-related" stuff into an ObservableComponent object**
    - **This is a reusable utility class**
    - **Implements storing and notifying observers**

Observable Interface, ObservableComponent Class



Coad's Observer Interface

- **Java's Observer interface won't work with this model:**
  public interface Observer {
  void update (Observable theObserved,
  Object argument); }
- **First parameter is an Observable**
  - **We need it to be anything that implements our IObservable interface**
- **Coad's solution: Replace with a new Observer interface that looks like this:**
  public interface IObserver {
  void update (Object theObserved,
  Object argument); }

  Or, should it be IObservable theObserved **???**