CS 370
REVIEW: Class Design

DR. MICHAEL J. REALE
FALL 2015
Abstract Data Types (ADTs)

- Abstract Data Type (ADT)
  - Collection of data
  - Operations that work on data

- ADTs → allow you to:
  - Manipulate real-world entities → work in your problem’s domain!
    - E.g., think about game board, pieces, cards, etc.
  - NOT worry about low-level implementation details
What should/can be an ADT?

- Typical low-level data types
  - E.g., a stack, a list, a queue
- Simple items
  - E.g., Light ADT with turnOn()/turnOff()
- Common objects
ADTs in Non-OOP Environments

• Have to have functions to create, delete, (and maybe set current) instance of ADT
  ○ createFont(fontID)
  ○ deleteFont(fontID)

• Three strategies:
  ○ Option 1: Explicitly identify instances → pass in ID
  ○ Option 2: Explicitly provide the DATA → pass in object
    ▶ Problem: exposes data structure to rest of program
  ○ Option 3: Use implicit instances (WITH GREAT CARE) → set current ID
    ▶ Problem: gets really complex tracking this
Creating Good Class Interfaces

- To create a good class → need good class interface

- To create a good class interface, need:
  - Good abstraction
  - Good encapsulation
Good Abstraction

- **Good Abstraction**
  - Class has **central purpose**
  - Interface has **consistent level of abstraction**
    - E.g., not generic list AND list of Employees → pick one interface
    - Otherwise:
      - Making one ADT out of two
      - Exposing implementation
      - Questionable use of inheritance
  - **Split** classes if they represent **more than one abstraction**!
  - Make interface **more programmatic** (and less semantic)
Good Encapsulation

- Encapsulation is stronger than abstraction
  - “Cannot look at details” vs. “Don’t have to look at details”
- Without encapsulation, abstraction tends to break down

Good Encapsulation

- Public methods → preserve integrity of interface abstraction
  - Otherwise, make private or protected
- Don’t make member data public!
  - Prevents unknown/invalid changes to data from outside
  - Allows change to implementation
- Avoid tight coupling between classes
- Favor read-time convenience over write-time convenience
Containment

- Containment
  - Class contains a primitive data element or object
  - Implementation of “has a” relationship
  - Less dangerous than inheritance

- ONLY implement “has a” through private inheritance as a last resort!
  - Used to allow access to protected functions/data
  - BUT violates encapsulation

- If have more than (7 +/- 2) data elements, consider whether class should be split.
Inheritance

- **Inheritance**
  - “Is a” relationship
  - One class is a specialization of another class
    - Must adhere COMPLETELY to original class contract/interface
      - Can use base class interface → don’t need to know whether base or derived → routines mean same thing
    - Adhere to **Liskov Substitution Principle (LSP)**
      - derived class = truly “is a” more specific version of base class
Using Inheritance

- Prohibit inheritance if you don’t want it!
- Want common interfaces/data/behavior as high as possible in inheritance tree
  - Stop if it breaks abstraction
- Polymorphism can be used instead of type-checking
  - *Exception*: if overridden function gets too watered down
- Should make data private rather than protected
Inheritance: Causes for Concern

- Only one instance of class
  - Exception: Singleton

- Only one derived class
  - “Designing ahead” → not a good idea to add extra inheritance

- Overriding a routine, but then do nothing inside
  - Violates interface contract

- Deep inheritance trees
  - DEFINITELY shouldn’t have more than (7 plus/minus 2) levels
    - Probably shouldn’t have more than (7 plus/minus 2) subclasses
Multiple Inheritance?

- **Multiple inheritance** → dangerous, but can be useful for “mixins”
  - “mixins” = simple classes that add set of properties to classes
  - E.g., Drawable, Serializable, etc.

- C++ → gives you interface AND implementation
  - VERY dangerous

- **Java**
  - Single inheritance
  - Can implement multiple **Interfaces**
    - Only gives you interface without implementation
    - Much safer
Containment vs. Inheritance

**Containment**
- Common data but not behavior → contain common object
- *Want current class to control interface*

**Inheritance**
- Common behavior but not data → inherit from common base class defining routines
- Common behavior AND data → inherit from common base class defining routines and data
- *Want base class to control interface*
Why Create a Class?

- Model real-world or abstract objects
- Reduce and/or isolate complexity
- Hide implementation details
- Limit effects of changes
- Hide global data
- Streamline parameter passing
- Make central points of control
- Facilitate reusable code
- Plan for a family of programs
- Package related operations
Classes to Avoid

- Avoid creating god classes
  - Don’t make all-knowing, all-powerful classes that effectively manage everyone else’s data

- Eliminate irrelevant classes
  - E.g., class has data but no behavior → move data to other classes?

- Avoid classes named after verbs
  - Only behavior but no data → generally not a class → should be a routine in another class
Other Good Practices

• **Member functions and data**
  - *Where possible and reasonable*, minimize:
    - # of routines in class
    - # of different routines called by class → minimize “fan-out”
    - Indirect routine calls to other classes
      - **Law of Demeter**
        - If A makes B, A can call B’s routines
        - However, A shouldn’t call routines from objects provided by B
          - E.g., avoid account>ContactPerson().PhoneNumber()
    - Extent to which class collaborates with other classes

• **Constructors**
  - Initialize all member data in all constructors (if possible)
  - Enforce singleton property by using private constructor
  - Prefer deep copies to shallow copies until proven otherwise
    - **Deep copy** = copies all member data
      - Easier to code
      - Safer
      - Usually not a big performance impact
    - **Shallow copy** = usually just a reference