CS 370
Class Design

DR. MICHAEL J. REALE
FALL 2015
We’re going to talk about designing and coding good classes...
Abstract Data Types
Abstract Data Types (ADTs)

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

- **Want to think about ADTs first (and then “classes” second)**
  - Otherwise, make classes that are “classes” in name only \(\rightarrow\) loosely connected bags of data and routines

- **ADTs \(\rightarrow\) allow you to:**
  - Manipulate real-world entities \(\rightarrow\) work in your problem’s domain!
    - E.g., think about game board, pieces, cards, etc.
  - NOT worry about low-level implementation details
Benefits of ADTs:

- (We’re going to use a Font ADT as an example)

- **Benefits of ADTs:**
  - Hide implementation details → changes don’t affect the whole program
    - `font.setBold(true)` instead of exposing HOW bold is turned on/off
  - Can make interface more informative
    - `font.setSizeInPoints(12)` and `font.setSizeInPixels(12)` instead of `font.size = 12`
  - Easier to improve performance
  - More obviously correct
    - `font.setBold(true)` rather than `font.attribute |= 0x02`
  - Program becomes self-documenting
  - Working with real-world entities rather than low-level implementation structures
What should/can be an ADT?

- Typical low-level data types
  - E.g., a stack, a list, a queue
  - Ask: “What does this represent?”
    - E.g., list of employees
  - Use highest level of abstraction you can
    - E.g., EmployeeList

- Common objects
  - E.g., a “file” is actually a lot of low-level steps that the OS handles
  - Layers of ADTs
    - E.g., Department → contains an EmployeeList

- Simple items
  - E.g., Light ADT with two functions: turnOn() and turnOff()
ADT Names

- Try to pick names that are:
  - Self-descriptive
  - Independent of low-level details, such as how it’s stored
    - E.g., don’t call an insurance-rates table “RateFile”, even if it’s stored in a file → maybe you’ll switch to a database in the future?
ADTs in Non-OOP Environments

- Usually, ADT = class
- What if you don’t HAVE classes? (E.g., pure C)
ADTs in Non-OOP Environments

- Functions would be standalone and not attached to a class (ADT services)
  - `setCurrentFontSize(sizeInPoints)`
  - `...`
- 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 each time you use ADT services
    - E.g., pass in `fontID` to each function
  - Option 2: Explicitly provide the DATA used by the ADT services
    - E.g., pass in the Font data type
      - NOTE: should only access Font data with ADT service functions → keep structure “closed”
      - *Problem*: exposes data structure to rest of program
  - Option 3: Use implicit instances (WITH GREAT CARE)
    - E.g., `setCurrentFont(fontID)`, then all services act on current font
    - *Problem*: gets really complex tracking this
Creating Good Class Interfaces

- To create a good class $\Rightarrow$ need good class interface

- To create a good class interface, need:
  - Good abstraction
  - Good encapsulation
Good Abstraction
A good class has a **central purpose**

- Name should describe central purpose
  - E.g., Employee class \(\rightarrow\) contains and acts on employee data
- All (public) routines work towards central purpose \(\rightarrow\) clear interface
  - E.g., Employee class has:
    - Fullname getName();
    - String getAddress();
    - ...

---

Central Purpose
A good class interface has a **consistent level of abstraction**

- Think of class as mechanism to implement one ADT
  - If class = ADT for generic list, fine
  - If class = ADT for list of employees, also fine
  - However, don’t want to mix the two!
- **Example of mixed levels of abstraction:**

```cpp
class EmployeeCensus : public ListContainer {
public:
    void AddEmployee(Employee employee);
    void RemoveEmployee(Employee employee);
    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
};
```

Effectively two different levels!
Why is the Preceding Example a Problem?

- Really two ADTs:
  - EmployeeCensus
  - ListContainer

- Exposing underlying implementation

- Questionable use of inheritance
  - Is an EmployeeCensus exclusively a ListContainer?
Consistent Abstraction

- A good class interface should present a **consistent abstraction**
  - Just because you decide to inherit and/or use another class, that doesn’t mean you need to expose ALL of that classes functionality!
When creating methods/services for a class, see if creating its opposite makes sense

*Examples:*
- TurnOn() and TurnOff()
- AddToList() and RemoveFromList()
Split classes if:

- Half the routines use half the data, and the other half use the other half of the data
  - Two classes masquerading as one!
- Class presents more than one abstraction
Make Interfaces Programmatic Rather Than Semantic

- **Interface has:**
  - *Programmatic part* → enforceable by the compiler
  - *Semantic part* → stuff you need to understand in order to use interface properly
    - E.g., call Method1() before Method2()

- **Where possible:**
  - Minimize dependence on semantic knowledge
  - Make the interface more programmatic
Modification of Interface

- May have to add functionality to interface over time → make sure it doesn’t break the interface’s abstraction!
  - E.g., don’t add SQL query generation to the Employee class...
Good Encapsulation
Encapsulation and Abstraction

- Encapsulation is stronger than abstraction
  - “Cannot look at details” vs. “Don’t have to look at details”

- Without encapsulation, abstraction tends to break down
Minimize Accessibility

- Should a method be public, private, protected?
  - *Key question*: “What best preserves integrity of the interface abstraction?”
    - I.e., does making this public fit with what services the class is supposed to provide?
    - *Even if the function uses only public functions* → don’t automatically make it public!
  - *Extreme strategy* → always go with the strictest privacy level possible
Don’t Make Data Public!

- Don’t expose member data in public!
  - Allows others to mess with data → class won’t know data has been changed (or whether it’s valid)
    - *Example*: Point class with x, y, z as public floats
  - Use get/set functions
    - *Example*: getX(), setX(), etc.
  - Allows you to change underlying implementation if you wish
    - External users shouldn’t know how data is stored under the hood

Watch of Semantic Violations of Encapsulation

- If you have to look at the implementation to understand what’s going on → violation of encapsulation!
  - Also, don’t USE classes based on knowledge of inner workings!
    - E.g., don’t call Init() before Operation(), because you happen to know Operation() calls Init()
Watch for Coupling That’s Too Tight

- To avoid tight coupling between classes:
  - Minimize accessibility of classes/members
  - Avoid friend classes
  - Make data private rather than protected in base class
    - Makes derived classes less tightly coupled to base class
  - Avoid exposing member data in class’s public interface
  - Be wary of semantic violations of encapsulation
Favor Read-Time Convenience

- Favor read-time convenience over write-time convenience
  - Don’t be lazy → you (or some other poor soul) will regret it later

http://www.tu-pc.com/fondos/media/4744.jpg
Containment (“has a”)
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
Introduction

• Inheritance
  ○ One class is a specialization of another class
  ○ “Is a” relationship

• If you use inheritance, you have to make decisions about:
  ○ Routines/methods
    ▫ Will each routine in base class be **visible** to derived classes?
    ▫ Will there be a **default implementation**?
    ▫ Will the default implementation be **overridable**?
  ○ Data
    ▫ Will the data be **visible** to derived classes?
When to Inherit?

- Derived class → more specialized version of base class
  - Must adhere COMPLETELY to original class contract/interface
    - If you just want the implementation (and not the interface), consider containment instead

- Adhere to Liskov Substitution Principle (LSP)
  - Don’t inherit from base class unless derived class = truly “is a” more specific version of base class
    - Can use base class interface without needing to know whether you have a base or derived class
    - I.e., routines mean the same thing
Prohibiting Inheritance

- If a class is NOT meant to be inherited from, prohibit it!
  - E.g., final in Java, non-virtual in C++, etc.

- On the flip side, DON’T “override” non-overridable functions in base class!

http://www.funnysigns.net/files/no-birds.jpg
## Kinds of Routine Inheritance

<table>
<thead>
<tr>
<th></th>
<th>Overridable</th>
<th>Not Overridable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default Implementation</strong></td>
<td>Overridable Routine</td>
<td>Non-overridable Routine</td>
</tr>
<tr>
<td><strong>No Default Implementation</strong></td>
<td>Abstract Overridable Routine</td>
<td>DEFIES LOGIC</td>
</tr>
</tbody>
</table>
Moving On Up

• **Want common:**
  - Interfaces
  - Data
  - Behavior

• ...as high as possible in inheritance tree

• Stop if it breaks abstraction
Causes for Concern

- Only one instance of class
  - Should this inherit from another class instead?
  - *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
  - Taken to ultimate conclusion, end up with ridiculously complex derived classes

- 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
Polymorphism can be used instead of type-checking:
- Instead of checking whether a shape is a Circle, Square, etc., just override draw() function

**Exception**: if overridden function gets too watered down
- E.g., DoCommand() for a Command class
Private Data

- Good idea to make data private rather than protected
  - Preserves encapsulation
  - Should have get()/set() functions if you need to access it
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*
Reasons to Create a Class
Why Create a Class?

- **Model real-world objects**
- **Model abstract objects**
  - E.g., Shape → abstraction of other specific shapes
  - Challenge to come with good abstract objects
Why Create a Class?

- **Reduce complexity**
  - Hide data/implementation so you don’t have to think about it later

- **Isolate complexity**
  - Makes it easier to debug/modify later
  - Not spread around entire program
Why Create a Class?

- Hide implementation details
  - Don’t need to worry about low-level details

- Limit effects of changes
  - Isolate things that are likely to change into their own class(es)
Why Create a Class?

- **Hide global data**
  - Makes it safer to work with

- **Streamline parameter passing**
  - Instead of passing parameter amongst several routines ➔ put all routines into same class with param as object data?
Why Create a Class?

- **Make central points of control**
  - Control can be:
    - Knowledge/data
    - Device control
    - Database control
    - Etc.
Why Create a Class?

- Facilitate reusable code
  - Well-factored classes → can be reused for easily
    - NOT talking about “gold-plating” (adding functionality you don’t need right now)

- Plan for a family of programs
  - Isolate parts you expect to change
  - May be able to reuse in related programs

- Package related operations
  - E.g., class that contains all trigonometric functions
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 Class Design Issues
Member Functions and Data

- Disallow implicitly generated member functions/operators you don’t want
  - E.g., make default constructor private → want Singleton class

- Where possible and reasonable, minimize:
  - Number of routines in class
  - Number 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