CSE 143

Object-Oriented Design

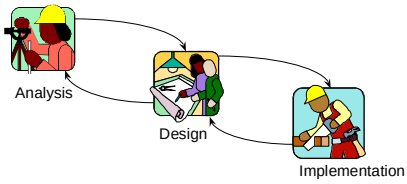
[Chapters 1, 8]

Design Process

- The traditional development model consists of three phases:
  - Analysis - defining the problem (what)
  - Design - the solution (high level)
  - Implementation - the solution (code)

Design Process

- In reality, these phases don't usually proceed sequentially: later phases often reveal changes needed in earlier phases
- This is sometimes called the *waterfall model*.



Traditional Development

- In this course, the problem is usually pretty well defined. We often suggest a design and ask you to provide an implementation.
- Most of the time, development isn't so easy…
  - Analysis or design may be poor
  - Goals may change during development
  - Implementation may be too hard or take too long
  - Need to be able to test components & final assembly
  - Social, political, economic factors

Design Methodology

- Changes may result in lots of wasted work! How to minimize their impact?
  - Use a good *design methodology*
  - Procedure and structure by which a design is created
- Top-Down Design, aka structured design
  - Focus on overall control flow rather than data.
  - Think of problem in terms of functions and algorithms and how they need to interact
  - Often have a layered or hierarchical approach: make successively more detailed refinements to design
  - Traditional design method for C and similar procedural languages

Top-Down Design

- Advantages and disadvantages?
  - Usually gives very structured approach to problem

```
First input is read, then processed, then results
  are printed, etc.
```

  - Adapts well to analysis, design, implementation phases
  - If right breakdown is chosen, everything works OK

```
Each function does one thing and is separated from
  others
```

  - What happens if a bad decision was made?

```
Not easy to fix without rewriting a lot of code
```

  - What happens if design needs to be changed?

Example: Payroll

- Suppose we are implementing a payroll program. We want to represent employees and provide appropriate operations for employee records
- Traditional design
  - Representation: data structure for each employee with hire date, salary, exempt or non-exempt, hours worked, …, and *kind* of employee (programmer, writer, pointy-hair boss, …)
  - Operations: set salary, set hours worked, compute pay, hire, fire, …

Employee Representation

- Represent employee data using a struct or class.

```
struct Employee {
   int kind;   // employee kind; 0=writer,
               // 1=programmer, 2=boss, ...
   Bool exempt; // = "exempt-not hourly"
   int pay;     // base pay
};
```


Employee Class (cont)

- Operations do something appropriate depending on employee kind.

```
int getPay(int hoursWorked) {
  switch(this->kind) {
    0: calculate pay for hourly writer
       return pay;
    1: calculate pay for exempt programmer;
       return pay;
    2: calculate pay for pointy-hair boss;
       return pay;
       ...
  }
}
```

- Even if implemented as a class, member functions would have the same structure

New Kind of Employee

- Now, what happens if we want to add a new kind of employee? Probably have to modify most of the functions involved.

```
void getPay(int hoursWorked) {
  switch(this->kind) {
       ...
    3: calculate pay for employee with
         stock options;
       return pay;
       ...
  }
}
```

- Not easy to do right, without introducing bugs

Object-Oriented Design

- Alternate design philosophy.
- Instead of control flow and functions, concentrate on different *kinds* of entities ("objects") in the problem (*data-driven* approach)
- Object = Collection of data and operations on that data
- All phases of design are in terms of objects
- Often easier to prototype a design or adapt to changing conditions

OO Design (2)

- What about changes?
  - Should be localized to a single object, or operations of objects, not spread out across lots of functions
- What about bad decisions?
  - Hopefully, changes will be restricted to objects instead of functions
- Ideally, objects are more reasonable way to think about problem than "flow control" is
- Which approach is better?
  - Depends on the problem, the people, etc...

Designing in the OO Style

- Identify the objects in the problem, and the operations they should have
  - What are the objects in the problem?
- Determine organization of objects and operations
  - How do the objects relate to one another?
  - Are some contained inside another object, or need to organize other objects?
  - Drawing an *object hierarchy diagram* might help
- Implement objects (C++ classes, or off-the-shelf)
- Test and refine

What is an object?

- Metaphor: robot, daemon
- In C++
  - A class instance
  - A variable of a built-in type
  - Examples:

```
ItemList I;
int c;

Bus b;                  // traffic simulation
DieselBus d;
ElectricBus e;
DualModeBus f;
```


What is an operation?

- Metaphor: a message which is sent to an object, requesting an action or service
- Any action that manipulates data
- Examples:

```
i++;  // built-in operation

list.insertItem(. . .);   // member function

strcmp(a, b);   // global function
```


Objects and operations

- From the problem statement:
  - *nouns* may suggest *objects*
  - *verbs (*and often *adjectives)* may suggest *operations*
- Determining *which* nouns and verbs are important is difficult
- Examples of nouns from the payroll package

```
programmer, writer, boss, ...
```

- Example verbs:

```
hire, setPay, getPay, fire, …
```

  - Adjectives:

```
hasStockOptions, knowsWindows, …
```


Writer Object

- Characteristics
  - name
  - address
- Operations
  - hire
  - set pay rate (hourly)
  - calculate pay for pay period given hours worked
  - change address given new address

Programmer Object

- Characteristics
  - name
  - address
- Operations
  - hire
  - set pay rate (exempt)
  - calculate pay for pay period
  - change address given new address

Objects and Operations

- What is the key here? We have many different kinds of objects and different operations.

	Writer	programmer	boss	...
hire	x	x	x	…
set pay	x	x	x	…
calc pay	x	x	x	…
change adr	x	x	x	…

  - Traditional design has function for each operation (row)
  - Object design has a class (object) for each column
- Which is better?
  - Often objects, particularly if expected changes will require new kinds of objects with similar operations

Employee Objects

- The different kinds of employee objects have much in common
  - Characteristics (name, address)
  - Operations (hire, set pay rate, calculate pay, change address given new address)
- But different kinds of employees require different specific versions of some operations
  - e.g., pay calculation is different for exempt and hourly employees
- Idea: Factor common features into one place (class); derive specialized versions from that one

Employee Class

- Create one class with information common to all employees
  - Give implementation of operations that are the same for all employee objects
  - Give interface of operations that have different versions for different kinds of employees
  - Define data common to all employees.

Employee Class

- (in pseudo-C++ - crucial details omitted for now)

```
class Employee {
public:
  void hire( );
  void setPay(int pennies);
  int getPay(int hoursWorked);
  void changeAddress(char *newAddr);
  ...
private:
   Bool exempt; // = “exempt-not hourly”
   int pay;     // base pay
   char *address; // home address
};
```


Employee Implementation

- Provide implementations for operations common to all employees

```
void Employee::changeAddress(char *newAddr)
{
  assert(length(newAddr) < maxAddr);
  strcpy(address, newAddr);
}
```


Employee Subclasses

- For each specific kind of employee, provide a derived class that extends (modifies, adds to) the basic Employee class.

```
class Writer : public Employee {
public:
  void setPay(int pennies);
  int getPay(int hoursWorked);
  ...
private:
   int hoursWorked;
};
```

new class → Writer; base class → Employee

operations with special versions for hourly employees

extra data for hourly employees

Employee Subclasses

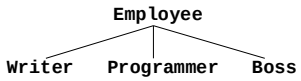
- Provide implementations of subclass functions (methods) in a .cpp file as usual

```
void Writer::setPay(int pennies){
  pay = pennies;
}
int Writer::getPay(int hoursWorked)
    { return hoursWorked * pay; }
```

- Create similar classes for Programmer, Boss, etc.

Subclasses Heirarchy

- The extended classes form a heirarchy



- Key idea: every object of class Programmer (Boss, Writer) can also be treated as an Employee object.
- Or, to put it another way, every Programmer (Boss, Writer) object *is-a* Employee object.

Employee Subclasses

- Since every Programmer (Writer, Boss) object is-a Employee object, any operation declared in class Employee can be used on those objects.
  - If the extended class (Programmer, …) doesn't provide a specialized version of an operation, the version of the operation from Employee will be used instead (*inherited*)

```
Writer homer;
Programmer dilbert;
homer.hire( );         // Employee::hire
homer.setPay(4712);    // Writer::setPay
dilbert.hire( );       // Employee::hire
dilbert.setPay(4712);
                    // Programmer::setPay
```


Summary

- Object-oriented design typically results in layered software.
- Data types (classes) play a vital role in these layers of abstraction.
- OO design helps in problem decomposition and aids in reuse.
- Still to consider:
  - polymorphism
  - C++ implementation details