Object-Oriented Programming: Class Design

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒 and Bob Myers

Object Oriented Analysis/Design

- Object-Oriented Analysis (OOA)
  - What objects do I need to implement the system?
- Object-Oriented Design (OOD)
  - How do I integrate the objects to make the system work?
- Object-Oriented Programming (OOP)
  - How do I use the programming language to create each object?

Goals of Software Design

- Simple
  - Easy-to-understand class interface and clear class relations
- Flexible
  - augmenting an existing class interface and revising implementation without affecting existing class users
- Extensible
  - capable of adding functionality to a class library
- Portable
  - recompile and run on different platforms
- Reusable
  - reuse code with relatively few modifications, e.g. automobile manufacturer.

Object-Oriented Design Process (I)

- Find the Classes
  - Extracting key nouns:

    When ordering new **foods** from a **restaurant**, the online food ordering App creates a **purchase order**, fills in the date, the restaurant's name and address, and enters **a list of videotapes** to be ordered.

Object-Oriented Design Process (II)

- Specify Class Operations
  - Foundation operations
  - Accessors and mutators
  - Conversion operations
  - Iterators
- Specify Class Dependencies
  - Inheritance
  - Composition
  - Link
- Specify Class Interfaces

Good properties for Class Design

- Strong Cohesion
- Completeness
- Consistency
- Loose Coupling

Basic Object Design: Cohesion

- A good class has *cohesion*, that is, it describes a single abstraction.
- The following example doesn't show good cohesion. Why?

```
class MailT {

  public:
    void sendMessage() const;
    void receiveMessage();
    void displayMessage() const;
    void processCommand();
    void getCommand();

  private:
    char *message;
    char *command;
    void formatString();
}
```

Basic Object Design: Completeness

- *Completeness* is obviously a necessary feature of a class.

```
class StringT {
  public:
    StringT(char *inputData);
    void displayString() const;
    char getLetter(int slot) const;
    char getLength() const;
    void concat(StringT concatString);
  private:
    char *str;
}
```

Incomplete class

```
// additional member functions
char getLetter(int slot) const;
char getFirstLetter() const;
char getLastLetter() const;
char getPrevLetter() const;
char getNextLetter() const;
char findLetter(char letter) const;
char findLetterEnd(char letter) const;
etc.
```

over-complete class

Basic Object Design: Completeness

```
class Fraction {
public:
   Fraction();                      // Set numerator = 0, denominator = 1.
   Fraction(int n, int d=1);        // constructor with parameters

   // standard input/output routines
   void Input();              // input a fraction from keyboard.
   void Show() const;         // Display a fraction on screen

   // accessors
   int GetNumerator() const;
   int GetDenominator() const;

   // mutator
   bool SetValue(int n, int d); // set the fraction's value through parameters

   double Evaluate() const; // Return the decimal value of a fraction
private:
   int numerator;         // top part (any integer)
   int denominator;       // denom must be non-zero
};
```

Basic Object Design: Consistency

- Examples of *inconsistent* class.

```
class DataT {
  public:
    DataT();
    DataT(char *name, int weight, int height);
    void setWeight(int weight);
    void putHeight(int height);
    int returnWeight();
    int getSize();
    void setValues(char *name, int height, int weight);
  private:
    char *name;
    int weight;
    int length;
};
```

```
class GraphicsT {
    ...
    void DrawLine(int x, int y);
    void MovePen(int x, int y);
    ...
};
```

absolute coordinates

relative coordinates

Basic Object Design: Reducing Coupling (I)

- Classes which have many interconnections are said to be *highly coupled*.

```
class InputT {
  public:
    double readFromFile(long &fileReferenceNum);
};
class MathT {
  public:
    double sine(InputT source, long &fileReferenceNum);
};
void main() {
    MathT mathObject;
    InputT inputObject;
    long fileReferenceNum = 0;
    cout << mathObject.sine(inputObject, fileReferenceNum);
}
```

- The high degree of coupling usually is the result of a design based on traditional programming in C.

Basic Object Design: Reducing Coupling (II)

- Good encapsulation reduces coupling.
- Encapsulation often reduces the need to pass parameters.

```
class InputT {
  public:
    InputT();                 // will set refNum to zero
    double readFromFile();    // will take care of refNum
  private:
    long refNum;
};
class MathT {
  public:
    MathT(InputT &);
    double sine()const;       // will use data as necessary
  private:
    InputT data;
};
void main() {
    InputT inputObject; //inputObject.readFromFile();
    MathT mathObject(inputObject);
    cout << mathObject.sine();
```