Synthetic OO Design Concepts & Reuse
Lecture 1: Program fragments and abstract classes

CSE335
Spring 2008
E. Kraemer
adapted from K. Stirewalt

Topics
– Economics of software development
– Design for change and program families
– Abstract operations (and pure virtual functions)
– Abstract classes
– Composite object structures

Synthetic OO concepts and reuse
Foundational OO concepts:
– Identity
– Classification
– Inheritance
– Polymorphism

Synthetic concepts: New design patterns and abstractions built by composing existing concepts in some novel way
Often motivated by desire to produce assets of value to an organization

Economics of software development

Problem: Financing development of software systems
– Big software is notoriously expensive over long term
– Many software projects run out of money before they ever yield a product

Question: What can a project manager do to minimize cost and maximize value?

What the manager can do:
– Outsource development to cheaper programmers

Economics of software development

Problem: Financing development of software systems
– Big software is notoriously expensive over long term
– Many software projects run out of money before they ever yield a product

What the manager can do:
– Outsource development to cheaper programmers
– “Buy vs. build”
Buy vs. build [Brooks § 16]

Dominant cost of software has always been development cost, not replication cost.
- Sharing that cost among even a few clients is a win.
- Use of \( n \) copies of a system multiplies the productivity of the programmer by \( n \).

Any existing product is cheaper to buy than to build from scratch:
- Even at $100K, a product costs only about as much as one programmer year.
- Delivery is immediate.
- Such products tend to be much better documented.

Key issue: Applicability
- Can I use an off-the-shelf product to do my task?

Economics of software development

Problem: Financing development of software systems
- Big software is notoriously expensive over long term.
- Many software projects run out of money before they ever yield a product.

What the manager can do:
- Outsource development to cheaper programmers.
- “Buy vs. build”
- Amortize costs over long term: Design for change.
- Create assets that have intrinsic value and that pay dividends: Reusable libraries.

Note: Assumes level of design skill and discipline among developers.

Design for change

Idea: Amortize development costs over the long term of a project or organization.

Model for thinking about how to design for change based on the notion of program families:

“We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from this set first by studying their common properties and then determining the special properties of the individual members.”

D.L. Parnas [TSE’1976]

What is a “Program Family”?

Is Netscape a “program”?

Mosaic

Netscape 1.1

Netscape 3

All of these are internet browsers that do about the same thing = family.

Navigator 2.0

Communicator 3.0

Netscape 6.0

Netscape 4.0

Netscape 2.0
**How are families developed?**

"Program fragments": incomplete programs; exist only in mind of the developer

**Problems with model**

Much of the important "design knowledge" is not represented [may even be obscured] in the code.

**Thus:** Very expensive to produce new members of the family

- Involves reverse engineering to infer designer's intent
  - Expensive
  - Difficult to do without introducing errors
- No easy way to put a boundary around what needs to be tested

---

**Example**

```plaintext
input: x, ε : double
returns double
is
if (x = 0.0) return 0.0;
else
  y := x / 2;
  while ( abs(x - y*y) ≥ ε * ε ) do
    y := (x/y + y) / 2;
  end while
  return y;
end if
end
```

**Question:** What does this program compute?

**Example**

```plaintext
input: x, ε : double
returns double
is
if (x = 0.0) return 0.0;
else
  y := x / 2;
  while ( abs(x - y*y) ≥ ε * ε ) do
    y := (x/y + y) / 2;
  end while
  return y;
end if
end
```

**Task:** Modify this program so that it computes the correct answer in half the time

**Example fragment**

```plaintext
input: x, ε : double
returns double
is
if (x = 0.0) return 0.0;
else
  y := x / 2;
  while ( abs(x - y*y) ≥ ε * ε ) do
    y := z such that abs(√x - z) < abs(√x - y);
  end while
  return y;
end if
end
```

**Note:** Method for choosing new value for y replaced with an abstract operation

---

**Design knowledge: a terrible thing to waste!**

**Idea:** Reduce cost by recording and using these intermediate program fragments

- Dramatically reduces costs involved in reverse engineering from code to determine designer's intent
- Dramatically reduces likelihood of introducing faults (recall that testing consumes up to 50% of development costs)
- May increase cost in developing first member of the family

"A savage finds his way skillfully through a wilderness by reading certain obscure indications; civilized man builds a highway which shows the road to all."  

-- John Dewey

---
Abstract operations
Invented by designer of a program fragment
– Signify designer’s intentional deferment of detail
– Named and documented so as to declare “what” is to be done without defining “how” that goal is to be accomplished
Subsequent refinements must flesh out the operation

Example fragment (more abstract)
input: \( x, \varepsilon : \text{double} \)
returns double
is
if \( (x = 0.0) \) return 0.0;
else
\( y := z \) such that \( \text{abs}(y \sqrt{x} - z) \in [0, \varepsilon] \);
return \( y \);
end if
end

Abstract decisions model of program-family evolution
Program fragments:
• retained; refined to produce new fragments and programs
• defer implementation/design decisions to fragments or programs “down the tree”.
• abstract operations specify “what”, not “how”

Abstract operations in OOP
Recall: A base class might declare an operation for which there is no reasonable default method
Example: Class Shape
– Defines an area() operation
– But there is no general method for computing area of a shape
Such an operation should be declared abstract
In C++, abstract operations are called pure virtual functions

Recall: virtual Functions
• Imagine a set of shape classes such as Circle, Triangle, etc.
  – Every shape has its own unique draw function but it is possible to call them by calling the draw function of base class Shape
  – Compiler determines dynamically (i.e., at run time) which to call
  – In base-class declare draw to be virtual
  – Override draw in each of the derived classes

Virtual functions, cont’d
– virtual declaration:
  • Keyword virtual before function prototype in base-class
    virtual void draw() const;
  • A base-class pointer to a derived class object will call the correct draw function
    Shape->draw();
  • If a derived class does not define a virtual function, the function is inherited from the base class
Virtual Functions

ShapePtr->Draw();
- Compiler implements dynamic binding
- Function determined during execution time

ShapeObject.Draw();
- Compiler implements static binding
- Function determined during compile-time

Pure virtual function

**Defn:** Mechanism by which a class can declare an operation w/o providing a method

**Syntax:**
```cpp
class BaseClass {
    public:
        virtual void pvf() = 0;
};
class DerivedClass : public BaseClass {
    public:
        void pvf() { ... }
};
```

Example: Class Shape

```cpp
class Shape {
    public:
        virtual unsigned area() = 0;
};

class Rectangle : public Shape {
    public:
        Rectangle(unsigned l, unsigned h) :
            length(l), height(h) {}
        unsigned area() { return length * width; }
        protected:
            unsigned length, height;
};
```

Abstract Class

**Defn:** A class that cannot be instantiated

**Illegal**
- Shape var;
- void f(Shape x)
- Shape g(); Shape* x = new Shape;
- Shape& var;
- void Foo(Shape & x)
- Shape* Bar(); Shape* x = new Rectangle(...);

**Legal**
- void Foo(Shape x)
- Shape* Bar(); Shape* x = new Rectangle(...);

Declaring an abstract class

In C++, a class is abstract if it:
- declares (or inherits) a pure-virtual function; or
- has a protected constructor

**Example:**
```cpp
class GUIElement {
    public:
        void move(unsigned x, unsigned y);
    protected:
        unsigned xPosition, yPosition;
    GUIElement(unsigned x=0, unsigned y=0) :
        xPosition(x), yPosition(y) {}
};
```

Abstract and Concrete Classes

- Abstract classes
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects (i.e., TwoDimensionalShape)
    - Can have pointers and references
Concrete Classes

- Concrete classes
  - Classes that can instantiate objects
  - Provide specifics to make real objects (i.e., Square, Circle)

Making a class abstract

- Declare one or more virtual functions as “pure” by initializing the function to zero
- Example of a pure virtual function:
  ```cpp
  virtual double earnings() const = 0;
  ```

Uses of abstract classes

Defining an abstract “placeholder” that can hold objects of various types
- E.g., Shape
- Useful for building composite object structures
Factoring common code into an abstract concept
Serving as a program fragment in the design of a family of programs
Definition of role-classes for use in collaboration-based designs

Case Study: A Payroll System Using Polymorphism

- The following example is a payroll system
  - Uses virtual functions and polymorphism to perform payroll calculations based on the type of an employee

```cpp
1. Employee
   earnings: declared pure virtual because implementation will depend on which derived class it will be used in:
   Employee is an abstract base class.
   ```
```
// Member function definitions for class Boss
1.1 Function Definitions

// Boss class derived from Employee

class Boss : public Employee {
1. Boss Definition (derived class)

// Constructor function for class Boss

Boss(const char*, const char*, double = 0.0);

// Get the Boss's name

const char* getFirstName() const;

const char* getLastName() const;

// Return a pointer to the first name

const char* firstName;

// Return a pointer to the last name

const char* lastName;

// Set the Boss's salary

double setWeeklySalary(double);

// Get the Boss's pay

double earnings() const;

// Print the name of the Employee

void Employee::print();

// Print the Boss's name

void Boss::print();

// Notice the overridden function definitions.
They were declared virtual in the base class.

// Notice the overridden function definitions.
They were declared virtual in the base class.

// Boss class derived from Employee

class Boss : public Employee {

// Constructor function for class Boss

Boss(const char*, const char*, double = 0.0);

// Get the Boss's name

const char* getFirstName() const;

const char* getLastName() const;

// Return a pointer to the first name

const char* firstName;

// Return a pointer to the last name

const char* lastName;

// Set the Boss's salary

double setWeeklySalary(double);

// Get the Boss's pay

double earnings() const;

// Print the name of the Employee

void Employee::print();

// Print the Boss's name

void Boss::print();

// Notice the overridden function definitions.
They were declared virtual in the base class.

// Notice the overridden function definitions.
They were declared virtual in the base class.
1. PieceWorker 
Definition 
(derived class)

1. HourlyWorker 
Definition 
(derived class)

1.1 Function Definitions
New Classes and Dynamic Binding

- Polymorphism and virtual functions
  - Work well when all classes are not known in advance
  - Use dynamic binding to accommodate new classes being added to a system
- Dynamic binding (late binding)
  - Object's type need not be know at compile time for a virtual function
  - virtual function call is matched at run time

Case Study: Inheriting Interface and Implementation

- Extension of point, circle, cylinder hierarchy
  - Use the abstract base class Shape to head the hierarchy
    - Two pure virtual functions printShapeName and print
    - Two other virtual functions volume and area
  - Point is derived from Shape and inherits these implementations
1. Circle Definition (derived class)

```cpp
#include <iostream>

using namespace std;

class Circle : public Point {
private:

    int radius;

public:

    Circle( int r, int x, int y ) { radius = r > 0 ? r : 0; }

    double getRadius() const { return radius; }

    void setRadius( double r ) { radius = r; }

    void print() const { cout << "Circle: [" << x << "," << y << "] Radius = " << radius; }

    double area() const { return 2 * 3.14159 * getRadius() * getRadius(); }

    double volume() const { return 2 * getRadius() * getRadius() * getRadius() / 3; }

};
```

1. Cylinder Definition (derived class)

```cpp
#include <iostream>

using namespace std;

class Cylinder : public Circle {
private:

    int height;

public:

    Cylinder( int h, int r, int x, int y ) { height = h > 0 ? h : 0; }

    int getHeight() const { return height; }

    void setHeight( int h ) { height = h; }

    void print() const { cout << "Cylinder: [" << x << "," << y << "] Height = " << height; }

    double area() const { return 2 * area() + 2 * 2 * 3.14159 * radius * height; }

    double volume() const { return 2 * area() * height; }

};
```

1. Load headers

```cpp
#include <iostream>

using namespace std;

class Circle {
public:

    Point print() const {
        cout << "Circle: [" << x << "," << y << "] Radius = " << getRadius();
        return *this;
    }

    double area() const {
        return 2 * PI * getRadius() * getRadius();
    }

    double volume() const {
        return 2 * PI * getRadius() * getRadius() * height;  // Volume = base area * height
    }

};
```

1. Initialize objects

```cpp
Cylinder cylinder( 10, 10, 10, 10 ); Radius = 3.30; Height = 10.00
Circle circle( 22, 8, 8 ); Radius = 3.50
Point point( 7, 12 ); // create a Point
```
Collaborative Exercise

Design classes for arithmetic expression trees. Each arithmetic operator class should provide operations for retrieving operand expressions. Define at least the following classes:

- Variable
- Literal
- Negate
- Add, Subtract, Multiply, Divide

**Hint:** You will need to invent some abstract classes

Exercise

Extend Expr hierarchy with polymorphic operation:

```cpp
void print( ostream& )
```

It should be possible to execute code such as:

```cpp
Expr* l = new Literal(5);
Expr* v = new Variable("x");
Expr* e = new Add( l, v );
e->print(cout);
v->print(cout);
```
### Composite pattern (idealized)

- **Client** → **Component**
  - `Operation()`
  - `Add(Component)`, `Remove(Component)`, `GetChild(int : Component)`

- **Composite**
  - `Operation()`
  - `Add(Component)`, `Remove(Component)`, `GetChild(int : Component)`

- **Leaf**
  - `Operation()`

### Composite pattern (most general)

- **Client** → **Component**
  - `Operation()`
  - `Add(Component)`, `Remove(Component)`, `GetChild(int : Component)`

- **Composite**
  - `Operation()`
  - `Add(Component)`, `Remove(Component)`, `GetChild(int : Component)`

- **Leaf**
  - `Operation()`

### Quick detour: UML associations

- Associations between instances of two classes:
  - Between Client and Component and between Composite and Component.
  - Latter is a special kind of association called an aggregation (diamond points to container).

- End points of the association called roles.

- Association multiplicities describe constraints on how many instances of a class playing one role in the association may be linked to instances of the class playing the other role.

### Design virtues of composites

- Makes client code simple:
  - Clients treat composite structures and individual objects uniformly.
  - **Design principle**: Abstraction through use of the abstract class `Component` and the identification of abstract operations that apply to many types of objects.

- Makes it easy to add new kinds of components without modifying any client code:
  - **Design principle**: Incrementality.
  - **Design principle**: Anticipation of change.

### Exercise

Draw a UML class diagram that illustrates the composite nature of the Expr class hierarchy. Said another way: Draw a UML class diagram that illustrates how the Expr class hierarchy implements the Composite Pattern.

### Question

Can you come up with another example use of the Composite pattern?