Object-oriented programming: OO design and behavior

Topics:
- Different flavors of OO design
- Interface classes

The design problems

OO languages have lots of powerful features
- Resulting complexity must be managed
- Often there is a “right way” and several “wrong ways” to accomplish the same goal

When we develop an application from scratch:
- Pressure to "get the classes right"
- Otherwise:
  - new features might involve massive changes to code; and
  - code will be difficult for new programmers to understand

We want to reuse code in later applications
- Pressure to invent general, essential, and minimal solutions
- Boils down to how to "get the classes right"
3 dimensions of software complexity

Data:
- Dominant source of complexity of information systems
- OOD Solution: Static data modeling and design

Behavior:
- Dominant source of complexity in reactive, interactive, and client-server systems
- OOD Solution: Role—collaboration-based design

Function:
- Dominant source of complexity in compilers, operations-planning, and scientific programs
- OOD Solution: Not much, but good support for polymorphism

Problem: How to reuse behavior

Suppose we want to design a GUI toolkit with widgets, such as Button, Slider, etc.

Problem:
- Class Button not very interesting by itself
- Pressing the button should cause some operation to be invoked on some other object
- But Button must know the class of this other object in order to invoke that operation
- So how do we make Button reusable?
Example: “Print” button

Collaboration between two objects:
- “print button” of class Button
- “document manager” object of class DocManager

Button press should invoke printDocument() operation

(Bad) design for class Button

class Button {
  protected:
  DocManager* target;

  void monitorMouse() {
    ...
    if ( /* mouse click */ ) {
      target->printDocument();
    }
    ...
  }
  ...
};

Question: Why is this a bad solution?
Problem

Observe: To invoke `printDocument()` operation, `Button` needed to know the class (`DocManager`) of the target object.

However:
- Button should not care that target is a `DocManager`
- Not requesting information from `target`
- More like sending a message to say:
  
  “Hey, I’ve been pressed! Go do something!”

Question: How can we design `Button` to send messages to objects of arbitrary classes?

Solution: *Interface class*

Defn: abstract class w/ nothing but pure virtual functions

```cpp
class ButtonListener {
public:
  virtual void buttonPressed( const string& )=0;
};
```

Declares the messages an object must be able to handle to collaborate with a `Button`
Better design of class Button

class Button {
   public:
   Button( const strings& lab ) : label(lab), target(0) {} 
   void setListener( ButtonListener* l ) { target=l; }
   protected:
   string   label;
   ButtonListener*  target;
   void monitorMouse() {
       ...
       if( /* mouse click */ ) {
           if(target) target->buttonPressed(label);
       }
       ...
   }
};

Collaborator must register interest in press events

Collaborator: any object that implements the ButtonListener interface

Implementing an interface

class MyDocManager : public DocManager,
   public ButtonListener {
   public:
   void buttonPressed( const string& s )
      { if(s == "print") { DocManager::printDocument(); } }
};
Example use

```c
int main(void)
{
    Button printButton("print");
    MyDocManager docMgr(...);
    printButton.addActionListener(&docMgr);
    ...
    run_event_loop();  // GUI-toolkit event loop
    return 0;
}
```