Software Design
CSE 335, Spring 2006

Object-oriented programming: OO Design and Behavior

Stephen Wagner

Michigan State University

The Design Problems

- OO languages have lots of powerful features
  - Resulting complexity must be matched
  - Often there is a "right way" and many "wrong ways"
- When developing from scratch
  - Pressure to get classes right

The Design Problems

- OO languages have lots of powerful features
  - Resulting complexity must be matched
  - Often there is a “right way” and many “wrong ways”
- When developing from scratch
  - Pressure to get classes right
  - Otherwise new features may involve massive changes
  - Code will be difficult for others to understand
- Developing with reuse in mind
  - We want to develop general, essential, minimal solutions
  - 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, scientific programs
  - OOD Solution: not much, but good support for polymorphism

Problem: How to reuse behavior

- We want to develop solutions that can be reused
- Solutions need to be flexible and general
- This can be quite challenging

Example

- Suppose we are designing a GUI toolkit with widgets such as `button`, `slider`, etc.

- Consider `button`
  - Not very interesting all by itself

Example

- Suppose we are designing a GUI toolkit with widgets such as `button`, `slider`, etc.
- Consider `button`
  - Not very interesting all by itself
  - Pressing a button should invoke an operation on another object

Example

- Suppose we are designing a GUI toolkit with widgets such as `button`, `slider`, etc.

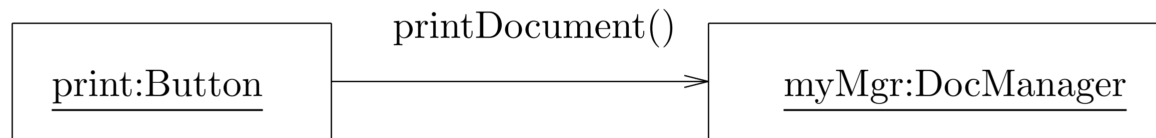
- Consider `button`
  - Not very interesting all by itself
  - Pressing a button should invoke an operation on another object
  - This requires `button` to know the class of the other object to invoke the operation

Example

- Suppose we are designing a GUI toolkit with widgets such as `button`, `slider`, etc.

- Consider `button`
  - Not very interesting all by itself
  - Pressing a button should invoke an operation on another object
  - This requires `button` to know the class of the other object to invoke the operation
  - How to make this reusable?

Example



- Collaboration between two objects
- Button press should invoke `printDocument`

(Bad) design for class button

```
class Button
{
   protected:
   DocManager *target;

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

Problem

- To invoke `printDocument()`, `button` needed to know the class (`DocManager`) of `target`
- How to make this reusable?

Problem

- To invoke `printDocument()`, `button` needed to know the class (`DocManager`) of `target`

- How to make this reusable?
  - `button` should not care what the target is
  - `button` should just send a message to the target

- How do we design `button` so that it can send messages to objects of arbitrary classes

Solution: `interface` **classes**

- Defn: abstract class with nothing but pure virtual functions

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

- Declares the `messages` that an object must handle to collaborate with `button`.

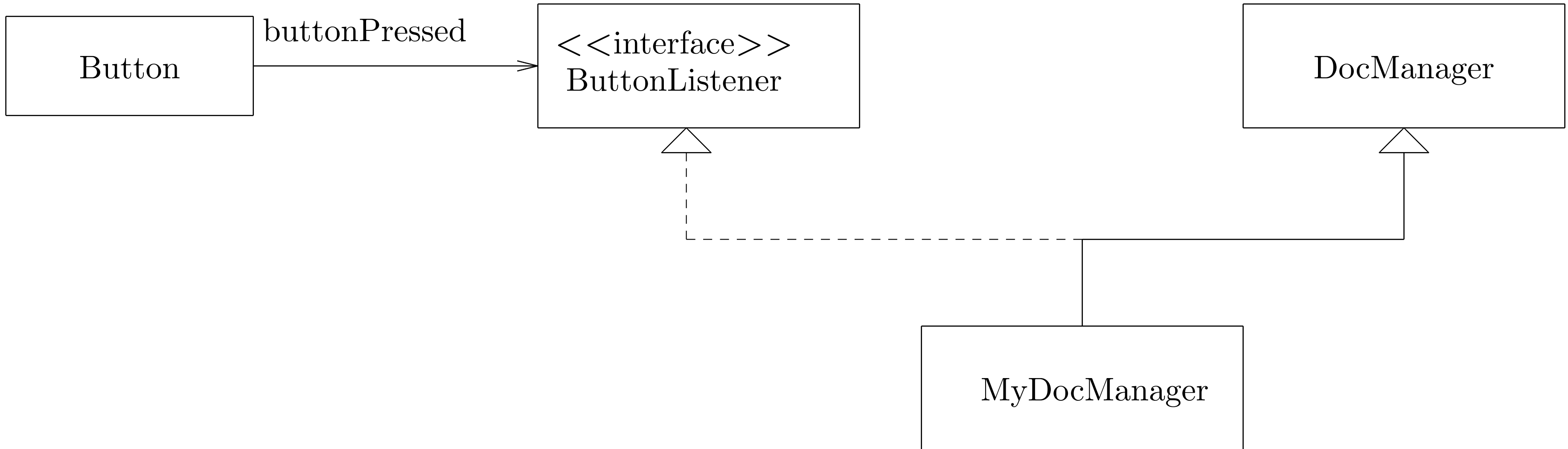
```
class Button
{
   public:
   Button (const string &lab) : label(lab), target(0) {};
   void setListener(ButtonListener *l) {target=l;};
   protected:
   string label;
   ButtonListener *target;

   void monitorMouse()
   {
       if (/* mouse click*/)
          if (target) target->buttonPressed();
   }
};
```

Implementing an `interface`

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

Example



Example

```
int main()
{
     Button          printButton("print");
     MyDocManager    docMgr(....);
     printButton.setListener(&docMgr);
     ...
     run_event_loop();  // GUI tool-kit event loop
     return 0;
}
```

Multiple Inheritance

- The same object can belong to more than one abstract category
  - A lion is a mammal
  - A lion is a carnivore
  - A shark is a fish
  - A shark is a carnivore
- There is no relationship between the separate categories
  - Not all mammals are carnivores
  - Not all carnivores are mammals

Multiple Inheritance

- C++ allows general multiple inheritance
- Java, C# support interfaces.
- General multiple inheritance is quite complicated to use and to implement

```
class A : public B, public C
{
   private:
   ....
}
int main()
{
    A a;
    B *b=&a;
    C *c=&a;
}
```

Interfaces in Java and C#

- Java supports interfaces.

```
public interface StockWatcher {
  void valueChanged(String tickerSymbol,
                    double newValue);
}
```

- C# supports interfaces.

```
interface StockWatcher {
  void valueChanged(string tickerSymbol,
                    double newValue);
}
```

Interfaces in Java

```
public class StockApplet extends Applet
      implements StockWatcher {

  public void valueChanged(String tickerSymbol,
              double newValue) {
      if (tickerSymbol.equals(sunTicker)) {
          ...
      } else if (tickerSymbol.equals(oracleTicker)) {
          ...
      }
  }
}
```

Java, C# and C++

- In `Java` and `C#` interfaces are a predefined type
  - syntax identifies what is an interface
  - syntax differentiates between inheritance and interface implementation
- In `C++` interfaces are implemented as a class
  - Both the base types and the interface are implemented as a `class`.
  - Multiple inheritance is used

Messages

- In OO design we think of objects sending and receiving messages
- In C++ we send a message by invoking an operation
- How do we send a `print` message to object `my`Object?

Messages

- In OO design we think of objects sending and receiving messages
- In C++ we send a message by invoking an operation
- How do we send a print message to object `myObject`?
  - `myObject.print(cout);`

Messages

- In OO design we think of objects sending and receiving messages
- In C++ we send a message by invoking an operation
- How do we send a print message to object myObject?
  - myObject.print(cout);
  - p− >print(cout); where p == &myObject

Messages

- p− >f(a,b,c);
- The object p points to is the receiver of the message.
- What is the message?

Messages

- p−>f(a,b,c);
- The object p points to is the receiver of the message.
- What is the message?
- Who is the sender?

Messages

- `p->f(a,b,c);`
- The object `p` points to is the receiver of the message.
- What is the message?
- Who is the sender?

Messages

```
void Button::monitorMouse()
{
    if (/* mouse click*/)
       if (target) target->buttonPressed();
}
```

- Who is the sender?
- Who is the receiver?
- What is the message?

Messages

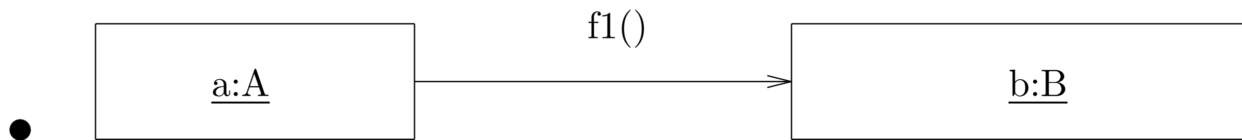
```
int main()
{
     Button          printButton("print");
     MyDocManager    docMgr(....);
     printButton.setListener(&docMgr);
```

- Who is the sender?
- Who is the receiver?
- What is the message?

Messages

- In order to send messages to an object, you must have a pointer to the object!!!
- You must specifiy the recipient of a message
- You can only send messages to an object that the object understands:
  - The object must have an appropriate method
  - This is enforced at compile time

Messages

- 

- What do we know about class `A`?
- What do we know about class `B`?