Delegates & Events
Observer and Strategy Patterns

Game Design Experience
Professor Jim Whitehead
January 30, 2009
Announcements

• Work breakdown and schedule
  ► Due next Monday
  ► Team assignment
  ► Goals:
    • Develop an understanding of work needed to be performed to complete project, and time required
    • Discuss breakdown of tasks among partners

• Homework assignments are individual assignments
  ► Project deliverables (like work breakdown) are done with your team member
  ► Homework assignments (like the Wumpus redesign) are done individually
  ► Homework assignments are intended to build individual skills in game programming
Publish/Subscribe (Pub/Sub) Example

• Consider this:
  ► What if you could automatically find out when your out-of-town friend is in Santa Cruz?
  ► One could imagine your friend having a cell phone that roughly knows its position
  ► You could **subscribe** to a location service on your friend’s phone
    • In fact, many people could subscribe to this service
    • Your friend wouldn’t need to know in advance how many people this would be
  ► When your friend came into Santa Cruz, the phone would **publish** a message to you

• This is an example of a publish/subscribe (pub/sub) service
Publish/Subscribe

- In a pub/sub service:
  - A client **subscribes** to a service
  - The service provider **stores** a list of subscribers
  - When a particular event occurs, a notification message is **published** to all subscribers

- An event
  - (in the general sense – C# events are in a few slides)
  - A noteworthy change in state
  - “A timely difference that makes a difference”

- A notification
  - A message carrying the information that an event has occurred

- *In-class acting out of pub/sub information flow*
Publish/Subscribe Advantages

• Scalable
  ► Can easily add more subscribers
  ► Just add another subscriber to the list in the service provider

• Loose coupling
  ► When writing the service provider, do not need to know the complete set of potential future clients
  ► Only need to adhere to a specific interface (data passed with the notification)
  ► Service provider is completely decoupled from the clients
  ► In network-based pub/sub, clients and servers live on separate machines
• Transactional processing
  ► Client may want to treat a series of events as a conceptual whole (a transaction), but doesn’t know how many events it will receive in a row
  ► If events are being used to update a user interface, many events can lead to lots of small, jittery changes to the UI

• Complicates information flow
  ► The information a client needs is not always found in the notification message. The client then needs to make further calls to get this information.
Publish/Subscribe Implementations

• Pub/Sub is a general information flow pattern
• Can be implemented in many ways
• Direct connection
  ▶ Subscribers directly subscribe to information sources
• Event message bus
  ▶ Notifications are sent to a third party, the message bus
  ▶ Clients subscribe to the message bus
  ▶ Service providers can come and go, but the clients don’t have to keep re-subscribing
• Local/Distributed
  ▶ Pub/sub can take place inside a local program, or across a network among several distributed programs
  ▶ In local programs, pub/sub frequently implemented using the Observer design pattern
  ▶ C# has a special language feature designed specifically for local pub/sub: delegates (and events)
Delegates

• A delegate contains a list of references to a method
  ► Must state the return type and parameters of the method
  ► List can contain 0, 1, or many method references
  ► Can think of a delegate as a typed function pointer

• Once a delegate is assigned a method, it behaves exactly like that method
  ► That is, you can perform method calls, via the delegate

• In the context of pub/sub systems, a delegate holds the list of subscribers
  ► That is, the list of methods to call when an event occurs
Defining and Using Delegates

// Define delegate type
[visibility] delegate [return type] delegate_name ([params]):

// Use example

class Subject
{
    // Create delegate type "notifier"
    public delegate void notifier(string message):

    // Create instance of "notifier" type, called myNotifier
    public notifier myNotifier:

    public void update(string message)
    {
        // Check if delegate instance is null
        // Then call delegate (calls all methods currently referenced by the delegate)
        if (myNotifier != null)
        {
            // might be calling more than one method!
            myNotifier(message);
        }
    }
}

Delegate Example in Visual C# 2008

Using delegates
1. Define delegate type
   • Specifies return type, parameters and their types
2. Create instance of delegate type
3. Assign method to instance
   • Use = or += operators
4. Use delegate to perform method calls on stored method references
   • Might call more than one method!
Observer Pattern

- The name given to an object-oriented, local implementation of publish-subscribe

  ▶ Subject
  - Holds list of subscribed observers in a delegate
  - Change of state in Subject leads to call on delegate
    - Acts as a notification to observers of change of state

  ▶ Observer
  - Subscribes to subject instances it is interested in
  - Supplies method to be called upon notification
Problem: Changing AI Behavior

Consider:
- AI behavior of an opponent often changes while the game is running
  - If it gets close to the player, or some other game event occurs

How can this be accomplished in code?
- Do not want to destroy opponent object, and create new one with changed AI behavior
  - I.e., creating a separate subtype of Opponent for each separate opponent behavior isn’t dynamic enough
- However, also do not want to encode all possible behaviors inside each type of opponent
  - Ideally want to re-use AI behaviors across many types of opponent
  - I.e., putting a big switch/case statement inside each Opponent type won’t work either
    - “Switch statement” and “duplicate code” bad code smells
Strategy Pattern

- Client creates instance of IStrategy subclass
  - myStrategy = new IStrategySubclass();
  - Or, can be given subclass instance in constructor
- Inside the client, write code that relates only to IStrategy
  - myStrategy.Algorithm();
  - Will call the Algorithm method on subclass currently assigned to myStrategy

Show example of Strategy pattern in Visual C#
Design Principles

• Two design principles at play here
  ► Favor composition over inheritance
    • More flexible to compose IStrategy subclass with Client than to make lots of Client subclasses
  ► Program to Interfaces, not implementations
    • If you program to an interface, are not tied to a specific class that implements the interface
    • Can easily create another implementation of the interface, and use that instead
      – If you program to an interface, substituting a new subclass of that interface is a small change
Homework

**Read:** Chapter 12 (Delegates and Events) from pp. 255-270 in *Programming C# 3.0*

**Read:** pages 139-148 (Strategy pattern) of *C# 3.0 Design Patterns*

**Read:** pages 210-217 (Observer Pattern) of *C# 3.0 Design Patterns*