CMPS 20 – Game Design Experience

Winter 2013

Week 4/5: Design Patterns

Arnav Jhala
TA: Sarah Harmon
Graders: Eric Lipshutz and Andrew Duensing, Thomas Deeb
**Pattern**: Reusable code structure that can be applied to commonly occurring solutions to software design problems

– They are proven solutions and are widely used and discussed

– Usually specified in a way that can be applied to many contexts

– Described using vocabulary that makes it easy to identify target problems

– Sometimes different patterns can be used to solve the same issue
Structure of Patterns

Context,
issues with handling code in this context,
configuration that resolves the issues.

– Pattern name
– Outline of context
– Problem statement
– Solution
– Design
– Implementation
– Example code
– Relations
Problem: Adding exhaust, shield to spaceship

• Vertically scrolling 2D space shooters have a spaceship that:
  – Banks left or right, depending on motion
  – Typically have three animations:
    • Banking left, straight ahead, banking right

• Also want to
  – Add particle effect for exhaust if the ship is moving up the screen
    • But turn this off if it’s moving down the screen
  – Add a shield in front of the ship if the player picks up a shield powerup

• Could accomplish all of these visuals using a series of combined animations
  – Bank left, bank left and shield, bank left and exhaust, bank left and shield and exhaust, etc..
  – But, leads to combinatoric expansion

• Isn’t there a better way? Why, yes!
Decoring the spaceship with shield & exhaust

• Instead of pre-defining all possible animations...
• Draw the shield, exhaust, and spaceship separately
  – Can think of “decorating” the spaceship with a shield, or an exhaust particle effect, as needed
  – No longer need large combinations of animations
Decorators using object-oriented code

• Basic idea is to create a linked list of objects
  – Each object draws one part of final scene...
    • E.g., the exhaust, the shield, etc.
  – ... then calls the next object on the list
  – The final object is the main object being decorated
    • E.g., the spaceship

Exhaust, shield, and spaceship are object instances
**Decorator pattern**

- **IComponent interface** gives just the decoration `Operation()`
  - Can mix in this Interface with a wide range of classes
- **Component** is terminal element (spaceship)
- **Decorator** provides decorating elements (shield, exhaust)
  - Diamond-headed arrow indicates that Decorator instances have a reference to an instance of `IComponent` (another Decorator, or a Component)

*Example of Decorator pattern*
The essence of the Decorator pattern

- Linked list of object instances, first one calls next one, etc.
- Each object adds its particular computation (often, but not exclusively, visual) to the output before passing to next object
- It is possible to have multiple chains, each ending at the same terminal (or decorator) object
  - Can pre-arrange commonly used decorator chains
Implementation details

- Need to know next object in chain
  - Typically passed into the constructor
    - Could have a separate method instead (would need to add this to IComponent)
- May need to know details about terminal object
  - Need to know position of spaceship to put shield directly in front, and exhaust directly in back
  - Each object in list can be given reference to terminal object
    - Use reference to spaceship object to retrieve its current location
- May need to remove a decorator
  - E.g., shields no longer active
  - Will need to write code that removes a decorator instance, and repairs the links among remaining ones
  - If the removed decorator is the first in the chain, removal is easy
Decorator Pattern: Pros and Cons

• Pro: Favors composition over inheritance
  – Decorator chains created at runtime, can be dynamically changed
  – Avoids combinatoric expansion of subclasses

• Pro: Avoids feature-rich parent classes
  – New decorations are “pay as you go”
  – Avoids need for adding all features into large, customizable parent class

• Con: Decorator and component aren’t identical
  – Requires more complex handling of situations where you really want direct access to terminal component

• Con: Many little objects
  – Can result in designs that have lots of little objects that are very similar
  – Can be hard to learn and debug, since so many classes affect output
Problem: Magic system with variable effects

• Consider a game that has castles, fighters, wizards, and magic cars
  – Fighters can be inside castles
  – Wizards have spells:
    • Some only affect the castle, weakening its walls (lightening bolt)
    • Other, more powerful spells, weaken the walls and do damage to the people inside (fireball explosion)
    • If the people inside the castle are inside a magic car, the car absorbs some, but not all, of the damage

• Need some way to have the applied damage vary depending on who is inside what
  – E.g., damage depends on containment
An approach: Chain of Spell Handlers

• Create objects that
  – Represent a cast spell
  – Represent castles, fighters, cars

• Create a chain of object references that represent containment
  – Castle –references--> Car –references--> Fighter

• Pass each spell object along the containment chain
  – Castle handles spell first, absorbs some damage, reduces power of spell, and passes it along to Car
  – Car handles spell, reduces power of spell (no damage – it’s magic), and passes along to Fighter
  – Fighter handles spell last, applying damage
Chain of Spell Handlers

- Chain of object references looks like Decorator chain
  - Containment of castle(cars(fighters))
- Separate object represents request (spell)

Fireball spell handed to castle for processing first
• Chain of object references looks like Decorator chain
  – Containment of castle(cars(fighters))
• Separate object represents request (spell)

Fireball spell handed to castle for processing first
Castle reduces its HP, and reduces power of fireball
Chain of Spell Handlers

- Chain of object references looks like Decorator chain
  - Containment of castle(cars(fighters))
- Separate object represents request (spell)

Castle hands Fireball spell off to car (next in chain), which reduces spell power
Chain of Spell Handlers

- Chain of object references looks like Decorator chain
  - Containment of castle(cars(fighters))
- Separate object represents request (spell)

Car hands Fireball spell off to fighter (next in chain), which applies remaining spell power as damage. Fighter is end of chain, so no further handoffs.
**Chain of Responsibility Pattern**

**IHandler interface**
- Successor: reference to next object in the chain
- Request(): method that receives request object (e.g., spell), and takes action on it

**Handler1, Handler2**
- Specific handler implementations (e.g., castle, car, fighter)
- Each handles requests in a different way

---

© Judith Bishop, C# 3.0 Design Patterns, O’ Reilly, 2008

**Figure 8-2. Chain of Responsibility pattern UML diagram**
• Request is often represented as an object instance
  – In this case, need to add IRequest interface and Request1...RequestN implementations to UML diagram

© Judith Bishop, C# 3.0 Design Patterns, O’Reilly, 2008
Chain of Responsibility: Pros and Cons

• Benefits
  – Reduces coupling between requests, and objects that handle requests
    • Would be very easy to add a new character class (say, a thief) and have it handle spells in a class-specific way, without modifying the spell class
  – Dynamic modification of request handling
    • Can change the chain of request handlers at runtime
    • For example, as fighters enter/exit cars, castles, etc.

• Drawback
  – Handling isn’t guaranteed
    • Since it is up to each object in the chain whether to take action, there is no global guarantee that anyone will handle a request
Problem: Representing Game State

• In most computer games, there is some state associated with the entire game
  – Current level number
  – Pointer to level data
  – Time elapsed while playing
  – High score during this play session

• Want one and only one instance of the object holding this state

• Also want to easily gain access to this state anywhere in the game
  – Avoid need for long data passing chains
• Singleton pattern ensures only one instance is ever made of a class
  – Wait.. How is this possible? Can’t I have my code just call “new” as many times as I want?
  – Not if the constructor is private!
    • Ensures that only code within the class can call the constructor
  – OK, so now I can’t call new, which means there is no way to ever create an instance, right?
  – Wrong – a private class variable (“instance”) in the Singleton is initialized by calling the Singleton’s constructor
  – Great, but how do I get access to “instance” if it’s private?
  – Use a public static property (“UniqueInstance”). You can always call this since it’s static.
• Pro:
  – Ensures only one instance of class is ever created
  – Can get access to data in this class from anywhere
• Con:
  – Acts like global variables
  – Can cause methods to have unintended side-effects, due to data sharing via state in the singleton
Different ways to do this:

One way is:

```javascript
var Singleton = {
    property1: 1,
    property2: 'name',
    method: function1()
};
```
Categories of Patterns

• Creational
  • Factory
  • Builder
  • Prototype
  • Singleton

• Structural
  • Adapter
  • Bridge
  • Composite
  • Decorator
  • Façade

• Behavioral
  • Chain of responsibility
  • Strategy
  • Visitor
  • Observer
Mid-Term Preparation

• Review on Monday
  – Readings R1-R6
  – Slides
  – Alchemy and Snake code
  – Basic javascript (data types, differences from other languages, mostly from the reading)
  – NO Brine specific questions

• Don’t have a sample exam (new class structure)

• Will try to provide some sample questions on Monday
• 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
Publish/Subscribe Disadvantages

• 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
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
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
• 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
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