Exam Questions
Chain of Responsibility & Singleton Patterns

Game Design Experience
Professor Jim Whitehead
February 4, 2009
Announcements

• Homework #1, Revisited (Improved Design)
  ▶ Due next Monday, February 9
  ▶ Make sure you ask in section this week whether your design is procedural, object-oriented, or other
The midterm exam is this Friday
- Will cover material up to and including Monday’s class (does not include today’s lecture)
- Will give list of potential questions on following slides
- Mostly short answer questions
Potential Exam Topics

• As Univ. of California students, you are expected to be able to assess complex material and make judgments concerning its relative importance.

• That said, it can be helpful to have some input from the Professor to help focus studying activity.

• The following are questions/topics that are likely, but not guaranteed to appear on the exam.

• Anything covered in class or in the assigned readings may appear, even if not explicitly mentioned today.
Potential Exam Questions

• C# language
  ▶ Basic syntax
  ▶ Foreach
  ▶ Properties
  ▶ Delegates
  ▶ Lists
  ▶ Arrays
  ▶ Visibility rules
  ▶ Enumerations
  ▶ Interfaces and Abstract classes
    • Know how to use these, and difference between these
    • Abstract methods
  ▶ Be prepared to read and understand C# code examples, and to write short segments of C# code
Potential Exam Questions

• Object-oriented design
  ▶ Know the difference between procedural and object-oriented design
    • Know the indicators of a pure procedural design
  ▶ Know the nouns/verbs rule of thumb for determining classes and methods from a problem description

• Refactoring
  ▶ What is a bad code smell? How is it related to refactoring?
  ▶ Know the bad code smells covered in class
    • Duplicate code, long method, long class, switch, feature envy
    • Be able to define these, and identify these in a code sample
    • Know how to refactor these to remove them
Potential Exam Questions

• Be able to define classification
  ► Know how classification relates to the is-a relationship
  ► Know how abstraction relates to classification

• UML structure diagrams
  ► Know how to represent a class in UML
  ► Know how to represent inheritance (is-a) and aggregation (contains) relationships among classes
  ► Given some C# code, be able to draw a UML diagram for it
Potential Exam Questions

• XNA
  ► What is a spritebatch?
  ► What is a clock tick? How long is the default clock tick in XNA?
  ► What is the difference between Update() and Draw() methods?
  ► Be able to understand code that receives input from an Xbox controller

• Design patterns
  ► Know UML diagrams for each of the design patterns covered in class
    • Strategy, Observer, Decorator
    • Need to memorize these, and be able to reproduce them
    • Need to understand how they work
    • Given an example of one these patterns, be able to identify the pattern
  ► Know what these mean:
    • Favor composition over inheritance
    • Write to an interface, not an implementation
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)

```
<table>
<thead>
<tr>
<th>castle</th>
<th>HP = 500</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td></td>
</tr>
<tr>
<td>fighter</td>
<td>HP = 100</td>
</tr>
<tr>
<td>HP = 50</td>
<td></td>
</tr>
</tbody>
</table>

Fireball spell handed to castle for processing first
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
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)

![Diagram showing the chain of spell handlers with objects castle, car, fighter, and fireball. Castle hands off the Fireball spell to the car, reducing its 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

Demonstration of Chain of Responsibility in Visual C#
Implementation details

- 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
Approach: Singleton Pattern

- 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.
Singleton Implementation

```csharp
public sealed class Singleton {
    // Private Constructor
    Singleton() { }

    // Private object initialized by calling private
    // constructor
    static readonly Singleton instance =
        new Singleton();

    // Public static property is used to retrieve
    // reference to the Singleton instance
    public static Singleton UniqueInstance {
        get { return instance; }
    }
}
```

- Constructor is private by default
- In C#, the first call on any methods, properties or constructors of a class causes its class variables to be set up.
  - First access to the UniqueInstance property causes variable “instance” to be initialized by calling constructor
  - Hence, first access to UniqueInstance causes new instance to be created, then returns a reference to it
  - Clever!
- Sealed keyword means no subclasses are possible

Code from *C# 3.0 Design Patterns*, p. 117.
Originally from:
Singleton Advantages

• 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
Homework

- Read in *C# 3.0 Design Patterns*
  - pages 110-122 (Factory Method and Singleton patterns)