Patterns Continued and Concluded

July 26, 2017

Review Quiz

What is the purpose of the Singleton pattern?

A. To advertise to other developers that the object should only be modified by `main()`

B.To prevent a system from creating multiple instances of an object.

C. To make it more convenient to create an object instance

D. To manage threading related consistency issues with a class

Which is **not** a danger of the Singleton pattern?

A. Singletons are global variables, making it difficult to track down modifications

B. Singletons are shared across threads, causing possible correctness issues

C. Singletons use static methods, which cause performance issues

Which is the purpose of the Builder pattern?

A. To allocate memory more efficiently when constructing large objects

B. To break a larger program up into smaller, more atomic units

C. To make it easer to instantiate objects that require a large number of arguments at construction

D. To safely build objects across threads

Which **is** a downside of the Builder pattern?

A. You need to implement a large number of constructors to manage all possible cases

B. You have to spread the implementation of a single concept across two classes

C. Extra function calls can cause substantial performance issues

D. Not using constructors in the target class reduces type safety

Which is the purpose of the Lazy Initialization pattern?

A. To avoid allocating resources for expensive method calls unless / until they're really needed

B. To schedule an object's creation until a future turn on an event loop

C. To reduce the amount of boiler plate code needed to define getter / setter methods

D. To safely synchronize events across threads

Which **is** a downside of the Lazy Initialization pattern?

A. The pattern can make it difficult to predict the cost of calling a method

B. The pattern can result in redundant memory allocations / large variables

C. The pattern can cause null pointer errors, due to methods returning inconsistent results

D. The pattern can cause security issues, due to loss of type information

Done!

House Keeping

Course Reviews

- Please complete, helps me involve the class!
- Due by August 2nd
- Check your email

Remaining Course Schedule

- **Friday, July 28**
  Final lecture

- **Monday, July 31**
  Final projects pt 1, final review

- **Wednesday, August 2**
  Final projects pt 2, final review

- **Friday, August 4**
  Final Exam

More Patterns

- Dependency injection pattern
- Object pool / Pool pattern
- Singleton pattern
- Builder pattern
- Lazy initialization pattern
- Factory pattern
- Adapter pattern

Factory Pattern

Problem Scenario

- We want to create different types of objects, depending on input
- We need to handle a lot of different cases (ie we provide lots of different classes)
- How to make this convenient for users?

Problem Example

- We're building a package for dealing with audio formats
- We create classes representing MP3, AIFF, FLAC, WAV, etc files
- Creates redundant, fragile, tightly bound code

factory/AudioParser.java –>

Problems in Example

- Redundant if / else-if / switch code
- Tight binding between implementation and client code
- Difficult to update going forward

Factory Pattern Solution

- Create a new class, to create instances
- Couple client code to new object and an interface
- <Interface> and <Interface>Factory

factory/AudioParser.java –>

Factory Pattern

- New "factory" class for creating other classes
- Typically for when types need to be determined at run time
- Prevent tight binding
- Related patterns:
    - Factory Method Pattern (ex: Path#toString())
    - Abstract Factory Pattern: (factories of factories)

Adapter Pattern

Problem Scenario

- We want to incorporate third party code
- Different author / interface, similar functionality
- Avoid subclassing
    - target class could be final
    - subclassing is more tightly bound than necessary

Problem Example

- We want to add MIDI support to our audio library
- Parsing MIDI can be complicated, so we'll use a third party
- Third party code does similar things to our code, but has its own interface / methods

adapter-pattern/miditools –>

Problem in Example

- Code has the functionality we want it too
- Incompatible method names, etc
- Difficult to use with our existing system

Adapter Pattern Solution

- Create a class in our system
- Have that class implement interface we control
- Pass method calls onto the "wrapped" / "adapted" object

adapter-pattern/miditools,
factory-pattern/audiotools –>

Adapter Pattern

- Wrap existing code we don't control, in an interface we do control
- Prevents overly tight binding / subclassing
- Good for creating common interfaces of disparate types
  - Ex: Web audio
  - Ex: Real time audio