Singleton (& Object Pool)

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 16

Acknowledgement & Materials Copyright

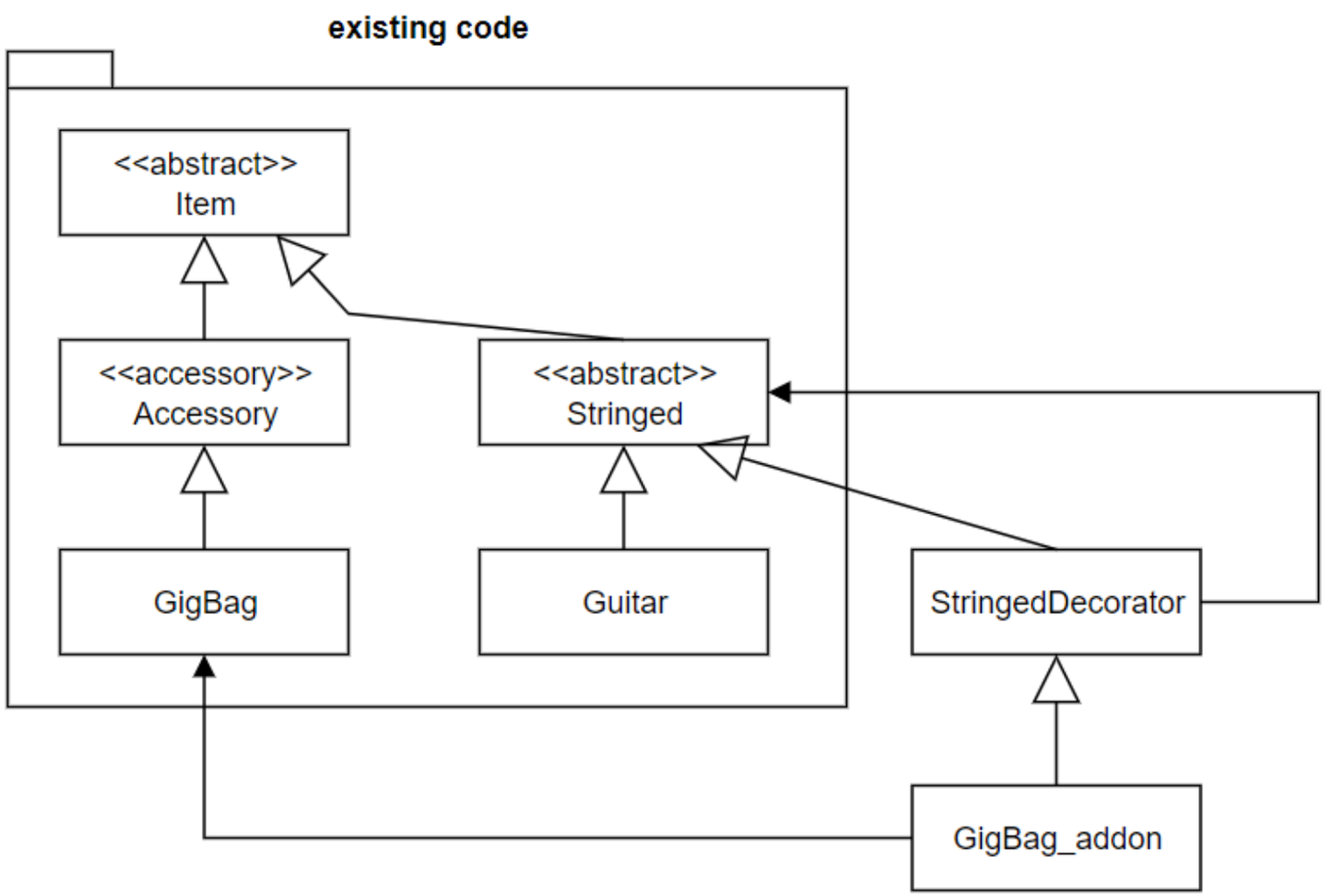
- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Before we start: Decorator in Project 3

- I think folks were a little thrown off because the decorator for stringed instruments used already existing class for accessories to decorate with
- Lots of folks had questions in office hours, but I hadn't really considered a solution
- So I went back to my Project 2 code and looked at a couple of different implementations
- The goals for implementing a decorator-like pattern
  - Don't change existing code
  - Make it easy to wrap or extend an existing transaction to add the decorating items
- **How you decide to implement the decorator depends on your design**
- My example code can be found in Files/Class Files under Decorator-Ex.zip

Decorator Example 1

- This one tries to get close to what the book does for a decorator
- If you disconnect the accessories from the decorators themselves, it gets a little easier to envision how you could apply the decorator and go get the actual items from inventory with them



Decorator Example 2

- Moving away from the book and more to how my design works, I created a custom abstract class called DecoratingItem
- It has reference to the Item inventory and an internal list of items added to an order
- It has custom methods for toString, totalCost, and putting an item from inventory into the list for an order
- I inherited DecoratingItem into custom subclasses for AddGigBag and AddStrings
- Those routines can have the logic in them for deciding what gets added, checking to see what's in inventory, etc.
- What I get after using DecoratingItem is a list of everything in the order and custom summary descriptions and total cost

Notes from the Graders

- Please use a .gitignore file to reduce the number of non-source files being pushed into repos – there's a post on this in Piazza (https://piazza.com/class/ky3q1sooafc1w3?cid=71) and more info here (https://docs.github.com/en/get-started/getting-started-with-git/ignoring-files) – you can usually find a good .gitignore for each language/IDE
- Avoid "god" classes and procedural code – large if then or switch blocks for basic logic
- Consider how your classes would extend if more subclasses or more methods were added – it helps in the base design to increase their independence of function (cohesion)
- Even though Java is generally strongly typed, using enums for known and limited parameter sets is recommended. (https://stackoverflow.com/questions/277364/java-enum-vs-string-as-parameters#:~:text=If%20your%20set%20of%20parameters,at%20compile%20time%2C%20use%20strings)

Singleton Pattern

- Used to ensure that only one instance of a particular class ever gets created and that there is just one (global) way to gain access to that instance
- Let’s derive this pattern by starting with a class that has no restrictions on who can create it

Deriving Singleton (I)

```
public class Ball {
    private String color;
    public Ball(String color) { this.color = color; }
    public void bounce() { System.out.println("boing!"); }
}
Ball b1 = new Ball("red");
Ball b2 = new Ball("green");
b1.bounce();
b2.bounce();
```

Problem: Universal Instantiation

- As long as a client object "knows about" the name of the class Ball, it can create instances of Ball
  - `Ball b1 = new Ball("orange");`
- This is because the constructor is public
  - We can stop unauthorized creation of Ball instances by making the constructor private

Deriving Singleton (II)

```
public class Ball {
    private String color;
    private Ball(String color) { this.color = color; }
    public void bounce() { System.out.println("boing!"); }
}
// next line now impossible by any method outside of Ball
Ball b2 = new Ball("red");
```

Problem: No Point of Access

- Now that the constructor is private, no class can gain access to instances of Ball
  - But our requirements were that there would be **at least one way** to get access to an instance of Ball
- We need a method to return an instance of Ball
  - But since there is no way to get access to an instance of Ball, the method can **NOT** be an *instance method*
  - This means it needs to be a *class method*, aka a static method

Deriving Singleton (III)

```
public class Ball {
    private String color;
    private Ball(String color) { this.color = color; }
    public void bounce() { System.out.println("boing!"); }
    public static Ball getInstance(String color) {
        return new Ball(color);
    }
}
```

Problem: Back to Universal Instantiation

- We are back to the problem where any client can create an instance of Ball; instead of saying this:
  - `Ball b1 = new Ball("blue");`
- they just say
  - `Ball b1 = Ball.getInstance("blue");`
- Need to ensure only one instance is ever created
  - Need a static variable to store that instance
  - No instance variables are available in static methods

Deriving Singleton (IV)

```
public class Ball {
    private static Ball ball;
    private String color;
    private Ball(String color) { this.color = color; }
    public void bounce() { System.out.println("boing!"); }
    public static Ball getInstance(String color) {
        return ball;
    }
}
```

Problem: No Instance!

- Now the getInstance() method returns null each time it is called
  - Need to check the static variable to see if it is null
  - If so, create an instance
  - Otherwise return the single instance

Deriving Singleton (V)

```
public class Ball {
   private static Ball ball;
   private String color;
   private Ball(String color) { this.color = color; }
   public void bounce() { System.out.println("boing!"); }
   public static Ball getInstance(String color) {
      if (ball == null) { ball = new Ball(color); }
      return ball;
   }
}
```

Variant: First Parameter Wins

- The code on the previous slide shows the Singleton pattern
  - private constructor
  - private static instance variable to store the single instance
  - public static method to gain access to that instance
  - this method creates object if needed; returns it
- But this code ignores the fact that a parameter is being passed in; so if a "red" ball is created all subsequent requests for a "green" ball are ignored
- The solution to this problem is to change the private static instance variable to a Map
  - `private Map<String, Ball> toybox = new HashMap…`
- Then check if the map contains an instance for a given value of the parameter
  - this ensures that only one ball of a given color is ever created
  - this is an acceptable variation of the Singleton pattern
  - indeed, it is VERY similar to the Flyweight pattern

Singleton Pattern: Structure

Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods (not shown)

The class has a static variable that points at a single instance of the class.

The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance

World's Smallest Java-based Singleton Class

```
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13 }
14
```

Meets Requirements: static var, static method, private constructor

Thread Safe?

- The Java code just shown is not thread safe
  - This means that it is possible for two threads to attempt to create the singleton for the first time simultaneously
  - If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!)

How to Fix?

```
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static synchronized Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13
14 }
15
```

In Java, the **easiest** fix is to add the **synchronized** keyword to the getInstance() method

Synchronized prevents other threads from entering once a thread acquires a lock. Can be applied to objects, methods, or parts of methods.

Alternative, just go ahead and make it

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it

If your code always has a Singleton, create a static instance. The JVM will create the unique instance before any threading issues even come up.

This is so-called **"eager" instantiation** – instantiating before needed
(vs. **lazy instantiation** – delaying instantiation until needed the first time)

Singletons

- Ensures one instance of a class in an application
- Provides a global access point
- Java’s version has a private constructor
- Consider alternatives for multi-threaded use

Python Singletons

- Don’t look…
- There are so many ways to do this in Python it generates great debate
  - https://stackoverflow.com/questions/31875/is-there-a-simple-elegant-way-to-define-singletons
- This definition is from the language creator

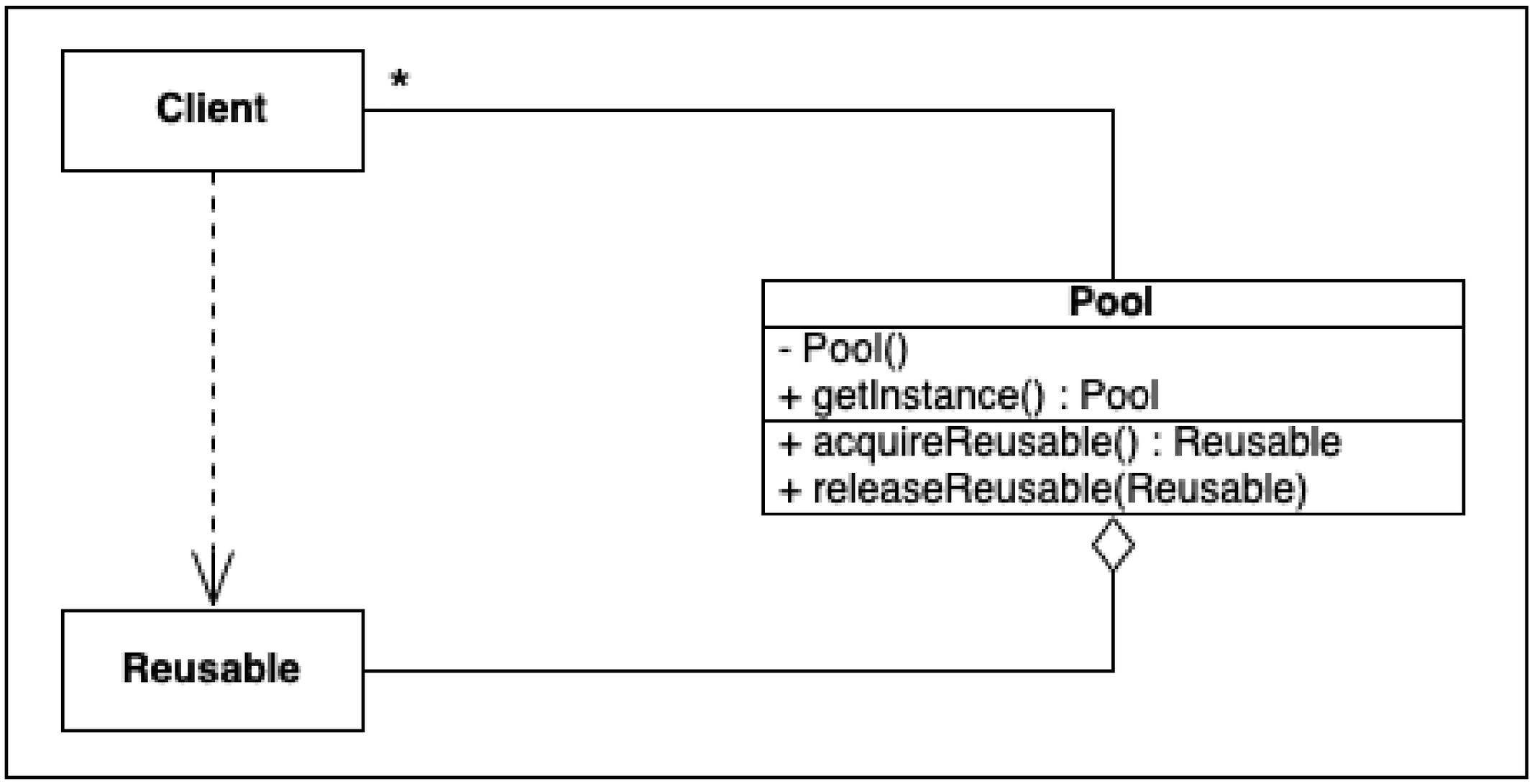
```
class Singleton(object):
    def __new__(cls, *args, **kwds):
        it = cls.__dict__.get("__it__")
        if it is not None:
            return it
        cls.__it__ = it = object.__new__(cls)
        it.init(*args, **kwds)
        return it

    def init(self, *args, **kwds):
        pass
```

Object Pool

- A variant of the Singleton Pattern is known as the Object Pool pattern
- This allows some instances (x) of an object to be created up to some maximum number (m)
  - where $1 < x \leq m$
- In this case, each instance provides a reusable service (typically tied to system resources such as network connections, printers, etc.) and clients don't care which instance they receive

Object Pool Structure Diagram



Common Use: Thread Pool

- One place in which the Object Pool pattern is used frequently is in multithreaded applications
  - where each thread is a “computation resource” that can be used to execute one or more tasks associated with the system
  - when a task needs to be done, a thread is pulled out of the pool and assigned the task; it executes the task and is then released back to the pool to (eventually) work on other tasks

Object Pool in Java

```
public interface Pool {

  // Return one of the pooled objects.
  T get();

  // Object T to be returned back to pool
  void release(T object);

  // Shuts down the pool, should release all resources.
  void shutdown();
}

public interface ObjectFactory {

  // Returns a new instance of an object type T.
  public abstract T createNew();
}
```

Many different implementations. This one is using templates to allow it to be easily used with other classes. It is built with two interfaces to implement in the object pool itself.

This one is from https://www.admfactory.com/how-to-create-object-pool-in-java/

Object Pool in Java

This implementation uses a Java Blocking Queue. A blocking queue blocks when you try to dequeue from it when its empty, or if you try to add queue items to it when its full.

This one is from https://www.admfactory.com/how-to-create-object-pool-in-java/

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public abstract class ObjectPool implements
ObjectFactory, Pool {

  private int size;
  private boolean shutdown;
  private BlockingQueue objects;

  public ObjectPool(int size) {
    this.size = size;
    shutdown = false;
    init();
  }

// Initiate the pool with fixed size
private void init() {
  objects = new LinkedBlockingQueue();
  for (int i = 0; i < size; i++) {
    objects.add(createNew());
  }
}
```

Object Pool in Java

Overrides of Pool methods here using the objects blocking queue.

https://www.admfactory.com/how-to-create-object-pool-in-java/

```
@Override
 public T get() {
   if (!shutdown) {
     T t = null;
     try {  t = objects.take();  }
     catch (Exception e) { e.printStackTrace(); }
     return t;
   }
   throw new IllegalStateException("Object pool is
        already shutdown.");
 }

 @Override
 public void release(T t) {
     try {  objects.offer(t); }
     catch (Exception e) { e.printStackTrace();  }
   }

   @Override
   public void shutdown() {
     objects.clear();
   }

   public int size() {
     return objects.size();
   }
 }
```

Typical Example – Thread Management

- ThreadPool implemented as a blocking queue of Thread subclasses called Processors
- Goal is to calculate the number of primes between 1 and 20M (20,000,000).
- Producer creates tasks to calculate primes between a subset of numbers, say 1 to 250,000; 250,001 to …
- Processor calculates in separate thread
- Consumer joins with Processors and merges the results

Object Pool in Python

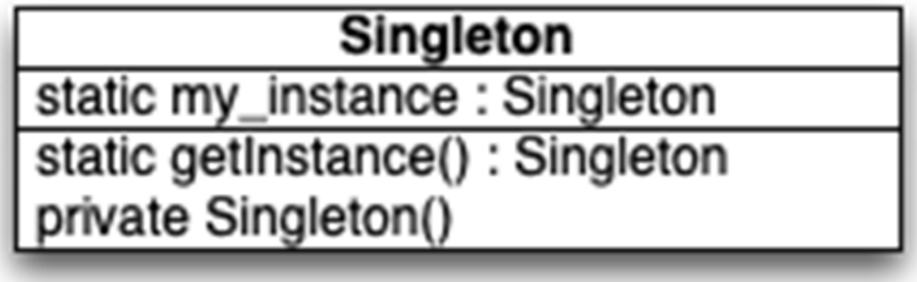
```
class ReusablePool:
#    Manage Reusable objects for use by Client objects.
    def __init__(self, size):
        self._reusables = [Reusable() for _ in range(size)]
    def acquire(self):
        return self._reusables.pop()
    def release(self, reusable):
        self._reusables.append(reusable)

class Reusable:
    # Collaborate with other objects for a limited time, until no longer needed for that collaboration.
    pass

if __name__ == "__main__":
    reusable_pool = ReusablePool(10)
    reusable = reusable_pool.acquire()
    reusable_pool.release(reusable)
```

https://sourcemaking.com/design_patterns/object_pool/python/1

Summary



- Singleton
  - Used to ensure that only one object instance of a class is instantiated and accessed
  - Award for simplest UML diagram
  - Don’t talk about in Python
- Object Pool
  - Used to ensure that only N object instances of a class are instantiated and accessed
    - Example: thread management
- Eager vs. Lazy Instantiation
  - Eager – creating an object and having it ready to use before it is needed
    - An alternative to using a singleton is an eager instantiation of the single object
  - Lazy – creating an object at the point it is needed

Next Steps

- If you're not on a team for projects, you should be! Use Piazza!
- Assignments
  - New participation topic up now on Piazza (CS references)– keep up with your responses
  - Project 3 part 2 on Wed 2/23
  - Next quiz goes up by Sat noon, due Thur
  - The Graduate Research Outline is due today! Draft Presentation is up next.
  - Readings from the Head First Design Patterns textbook
    - Lecture 13 on Decorator – Chapter 3
    - Lecture 14 on Factory – Chapter 4
    - Lecture 16 on Singleton – Chapter 5
    - Lecture 17 on Command – Chapter 6
- Coming up
  - Conceptual Design, Singleton/Object Pool, Command, Façade/Adapter
  - In class exercise on UML of patterns…
- Please come find us for any help you need or questions you have!