**15-214 Design Pattern Practice Andrew IDs \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

A server receives messages from a client containing operations that the server should do. The server is supposed to execute these operations in the order received. What design pattern should be used in the server’s implementation to accomplish this?

**Command – represents an operation explicitly so it can go in a queue**

What design pattern is being implemented here? **Template Method pattern**

class Parent {

public final void aBigOperation() {

littleOperation1();

littleOperation2();

littleOperation3();

}

protected abstract void littleOperation1();

protected abstract void littleOperation2();

protected abstract void littleOperation3();

}

class Child extends Parent {

protected void littleOperation1() { … }

protected void littleOperation2() { … }

protected void littleOperation3() { … }

}

What are two benefits of the above design pattern?

1) **Superclass decides order, fixed**

2) **Reuse overall order of behavior in superclass - can modify behavior in subclass**

You want to add the ability to write text to Java standard output (System.out.println()) or a file in ALL CAPS. You want this to work with clients who can write text to a PrintStream, but who write text in lowercase that will need to be converted. What pattern should you use and why?

**Decorator – changes behavior without changing interface**

Cross out some old code and add some new code to implement the factory method pattern in the code below:

**abstract** class World {

public final void simulateAnAnimal() {

Rabbit rabbit = **makeRabbit()**;

rabbit.move();

rabbit.eat();

}

**protected abstract Rabbit makeRabbit();**

}

class AnotherWorld extends World {

**protected Rabbit makeRabbit() { return new Rabbit(); }**

}

class DeadlyWorld extends World {

**protected Rabbit makeRabbit() { return new KillerRabbit(); }**

}

What pattern is being implemented here? **Composite**

class SlimeMold {

abstract void eat();

abstract void move();

}

class SlimeMouldCell extends SlimeMold {

void eat() { /\* much some yummy debris \*/ }

void move() { /\* go find something to slime \*/ }

}

class SlimeMouldGlob extends SlimeMold {

Set<SlimeMold> subGlobs = …;

void addGlob(SlimeMold m) { subGlobs.add(m); }

void eat() { for (SlimeMold m : subGlobs) { m.eat(); } }

void move(){ for (SlimeMold m : subGlobs) { m.move(); } }

}

You’ve been thinking about writing a better Rabbit AI. You’d like rabbits to behave differently when they’re hungry vs. when they’re full, and you want to switch between the AIs easily depending on the contents of the rabbit’s stomach. What design pattern would you use, and why?

**State (better than Strategy)**

Draw a UML diagram of the Adapter pattern

**(see class notes)**

You’re implementing a mobile phone, and you have created an object to keep track of whether the phone is in airplane mode. Some other applications probably need to know when the phone goes into airplane mode and when it comes out, but you don’t know exactly what those applications are. What design pattern should you use to keep them up to date?

**Observer**

There is only one airplane mode status object. How do you make sure that stays the case, and let clients get it when they need it? Name the pattern and sketch code that implements it.

**Singleton (see textbooks and/or online resources for code)**

You’ve written a framework for mobile phone surveys. People can use it to write mobile phone apps that ask the user 7 different kinds of questions, show the user the questions with the aid of fancy fonts, pictures, and videos, and analyze the answers with sophisticated statistical analyses. But some programmers just want a simple way to ask a few questions. What pattern would you use to provide this simple interface to your survey framework?

**Façade**