CSSE 374: Exam 2

# Policy

This is an open-book, open-note, open-Moodle exam. The work you do must be novel and your own original work. You can use Moodle, any URLs that are directly reachable from Moodle, and your eBook if you have one. You are not allowed to use any other online resources. This exam contributes **10%** to your final course grade.

Design Analysis (D)

This category explores your insights into the fundamentals of good design. It is worth 25% of the exam grade. To earn a grade on the exam, you must get at least half of the questions correct.

1. Revisit Exam 1. Here is a student's attempt at the abstract decorator in the Exam 1 code:

public abstract class Decorator extends TextProcessor{

protected TextProcessor decorated;

public Decorator(TextProcessor decorated, String text, Transformer transformer) {

super(text, transformer);

this.decorated = decorated;

}

public String transform(){

return this.decorated.transform();

}

}

Here is the student’s mostly correct code that attempts to use this decorator, but does unexpected things solely due to bugs in the student's abstract decorator code:

public class SwitchProcessor extends Decorator {

private Transformer transformerUsed;

public SwitchProcessor(TextProcessor decorated, String text, Transformer transformer) {

super(decorated, text, transformer);

this.transformerUsed = this.transformer;

}

@Override

public String transform() {

if (this.text.equals("S")) {

switchTransformer();

return "Switched transformers";

} else {

return this.transformerUsed.transform(this.text);

}

}

private void switchTransformer() {

if (this.transformerUsed == this.transformer) {

this.transformerUsed = this.decorated.transformer;

} else {

this.transformerUsed = this.transformer;

}

}

}

Recall that TextProcessor has many methods besides transform(): getText, setText, etc. The student believes that following the Decorator Pattern involves decorating only the methods explicitly mentioned by their exam’s problem specification. Explain what is wrong with the student's train of thought and give a specific test case that shows the gap in the student’s abstract decoration. Then, briefly explain how you would change the student's abstract decorator so the decoration logic works according to the conventions of Decorator.

*The issue with this code is that when you use a decorator, you want to override every method in the class. When you extend a class, you are basically promising that you will do everything that that class does. If you only decorate one method, the user of this decorator will experience unexpected behavior when they try to use methods that should work.*

*Test case:*

*make a TextProcessor with text and decorate it*

*call setText on the top level*

*call transform on the top level*

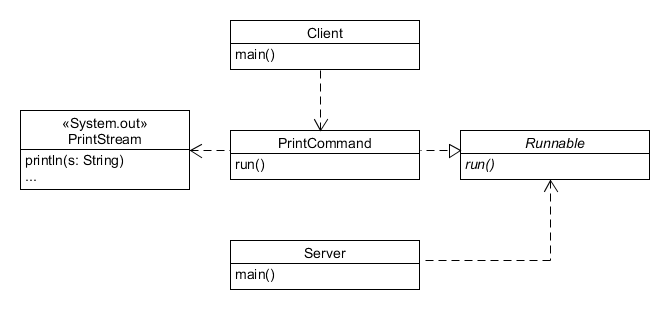
*call getText on the top level*

*This should result in an answer using the text from the original TextProcessor. Since setText is not overridden, the original TextProcessor has no way to know that there is new text to process. Since transform uses output from inner processors, it will get a bad value. To fix this, ALL methods should be implemented correctly so that everyone gets the information that it needs to do it's job*

2. Revisit the Halfway version of the command pattern live coding project. Students changed the code for the commands they were sending the server. The server printed the different messages the students were sending the server, but silently ignored the other changes to the code. Why did the server execute some, but not all, of the students' changes?

*The only changes to the code that were executed were from the run() function of the command that you sent over the network. It was just one runnable object that was done.*

3. Here is the UML for the Halfway version of the Command Pattern Live Coding:



Label the classes with the corresponding roles of the Command Pattern:

|  |  |
| --- | --- |
| Class | Role |
| Client | Client |
| PrintCommand | command |
| Runnable | command |
| Server | invoker |
| PrintStream | reciever |

*The Client is the one who sets up the object*

*PrintCommand is the real command here, runnable is just an interface for that so I think that is a command as well*

*The server gets the command and calls run on it*

*the printstream is affected bu the command*

4. Name a concrete example of a simple factory and abstract factory mentioned in this course that are designed to solve some common problem in different ways. Compare the two factories’ coupling and cohesion.

*Simple factory was used to make specific pizzas in the pizza store, it kind of doesn't really help us too much with design since is is basically moving the ugly switch statement that you will need to maintain somewhere else in the code. It kind of helps us with cohesion since we can easily find where objects are made a the cost of some coupling since we now have a new dependency in the pizza store.*

*Abstract factory is kind of like a factory fro factories. We used an abstract factory for the gui os thing. We had different factories for each operating system that would generate different widgets based on the type. The application did not really care about what operating system it was using since it just used the interface for object creation. Abstract factories remove coupling in an application by abstracting out object creation. Cohesion is a little hurt since you end up with a decent amount of complexity.*

5. Can an object be both an Invoker and Receiver in the Command Pattern at the same time? If so, give a concrete example. Otherwise, explain why not.

*Yea, it can. Say you have a command that wipes the command queue of the invoker. In this case, the invoker is both storing the command and being effected by the command.*

## Design Synthesis

Our PizzaStores have really taken off, but the NY chains are encountering problems with high turnover. The executive leadership in Chicago wants standardize the pizza production process. Their strategic plan calls for building robots to prepare pizza ingredients. The management has begun the competitive bidding process with the various international robot manufacturers, but it seems likely that the franchises will begin regionally sourcing robot production sooner rather than later.

Your very wise project manager, Emily, has interpreted all of this business speak to mean that you need to code two things:

1. Make a Command Pattern to abstract the robots' preparation of ingredients.
2. Create an abstract Decorator, similar to FilterInputStream, for the existing PizzaIngredientFactory interface to hide the instantiation and invocation of the Commands on the ingredients.

You are given Freeman's abstract factory of pizza ingredients. The current NY and Chicago concrete factories return the raw ingredients, not the prepared ingredients.

First, implement a Command Pattern to hide the details of robotic pizza ingredient preparation. Your management will give the robot manufacturers the current steps for preparing ingredients. The manufacturers will implement these steps in a method in an interface that you provide to hide the tricky code for manipulating the robots. The manufacturers needs some ingredients to act on, so the method will take in the ingredient in question as a parameter and return the processed ingredient as the same type as the parameter.

For example, a Veggie pizza needs chopped veggies of various kinds, so Samsung will make a SamsungChopVeggies command that implements your interface. They will take in Veggies as a parameter and return the chopped Veggies. This design hides the invocation of Samsung’s proprietary instructions to move the robots' servos to chop the Veggies.

Second, create an abstract decorator for Freeman's abstract ingredient factory to run Commands on the raw ingredients fetched from the underlying factory. Emily says that all of the manufacturer’s concrete decorators are interested in this feature of running Commands on ingredients, so put this logic in the abstract decorator, not the individual concrete decorators. The manufacturers will code to your abstract decorator and handle the construction of their Commands in their constructor. Your abstract decorator must offer an easy way to associate Commands with ingredients. For example, Samsung will write a SamsungDecorator, with one command per ingredient, e.g. SamsungChopVeggies. You should give Samsung some nice methods on your abstract decorator, like setVeggiesCommand, so they can associate their Commands with specific ingredients. Your abstract decorator will then call the associated Command whenever it is asked to produce a given ingredient.

Your decorator must fulfill the contract of Freeman's existing abstract factory. For example, if you construct a VeggiePizza, you know that the code for the VeggiePizza will ask your decorator for veggies. Your decorator should return the robot-prepared veggies from invoking the Command.

### C: design

Use the principles of good design to prepare a UML description of your design with the following features:

1. Draw the UML for this system. Reuse, but do not change, Freeman's factories and ingredients code.
2. Add a sample concrete decorator, PrintDecorator, to your UML. Add simple concrete PrintCommands to "prepare" (just print) the raw ingredients they receive. Manufacturers will use this UML as a template for making their own concrete decorators.
3. Add a note in your UML with a few lines of code demonstrating how to use your design to construct several types of Pizzas that use PrintDecorator-prepared ingredients processed from a NY ingredient factory.

### B: code the abstractions

1. Reuse, but do not change, Freeman's code.
2. Code the Command Pattern for this problem.
3. Code the Abstract Decorator for this problem, completely encapsulating what stays the same and meeting all of the above requirements.

### A: code a concrete example

1. Implement the sample concrete decorator, PrintDecorator, to instantiate simple Commands that just print the ingredients the robot is preparing. The manufacturers will use your code as a reference for writing their code.
2. Copy/paste the code you wrote for Part C into a main method and demonstrate that it works as-is.