This was a 180-minute open-book test. There were a total of 101 points on this exam. If you answered more than 100% worth of questions correctly, your score was 100%.

**Question 1.** (1 point each) Answer each of the following with a word or a phrase.

(a) What principle can be described succinctly by, “Only talk to your immediate friends”? Answer: The Law of Demeter (Lec. 20)

(b) What principle says that when two elements of an interface conflict, or are ambiguous, the behavior should be that which least surprises the user? Answer: Principle of Least Astonishment (Lec. 17)

(c) Name a principle that tells which type of objects should be able to replaced by which other type of objects without undermining certain desirable properties of the program. Answer: Liskov Substitution Principle (Lec. 17)

(d) What principle means that well structured programs often require callbacks? Answer: The Hollywood Principle (Lec. 25)

(e) What is the name of the problem that is manifested when high-level components call low-level components, as well as vice versa? Answer: Dependency rot (Lec. 25)

(f) What is the assertion that evaluates to true between calls to an object of a particular class? Answer: Class invariant (Lec. 19)

(g) What principle says that the object that contains the necessary data to perform a function should be the object that manipulates the data? Answer: The Expert pattern (Lec. 20)

(h) What is the name of the problem caused by polymorphism making it hard to find the class where a called method is implemented? Answer: The yo-yo effect (Lec. 18)

(i) What is the alternative to inheritance that requires explicitly sending messages to an object of another class? Answer: Delegation (Lec. 18)

(j) What is the principle that says that a class should consist entirely of closely related functionality? Answer: High cohesion. (Lec. 20)

(k) What is the name of a small utility class that defines some useful functionality that will usually become part of something larger? Answer: Mixin. (Lec. 7)

(l) In the demanding style of enforcement, which class is expected to check whether a condition is satisfied? Answer: The client (or the calling class). (Lec. 19)

**Question 2.** (1 point each) Here are the “official” descriptions of several design patterns. Give the name of the pattern that each description refers to.

(a) Decouple an abstraction from its implementation so the two can vary independently. Answer: Bridge

(b) Provide an interface for creating families of related or dependent objects without specifying their concrete classes. Answer: Abstract Factory

(c) Provide an interface for creating families of related or dependent objects without specifying their concrete classes. Answer: Factory Method
(d) Specify the kinds of objects to create using a prototypical instance, and create objects by copying this prototype. Answer: Prototype

(e) Ensure that a class only has one instance, and provide a global point of access to it. Answer: Singleton

(f) Convert the interface of a class to one expected by clients. This lets classes work together that couldn't otherwise because of incompatible interfaces. Answer: Adapter

(g) Attach additional responsibilities to an object dynamically. This provides a flexible alternative to subclassing for extending functionality. Answer: Decorator

(h) Use sharing to support large numbers of fine-grained objects efficiently. Answer: Flyweight

(i) Provide a surrogate or placeholder for another object. Answer: Proxy

(j) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. Answer: Chain of Responsibility

(k) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Answer: Iterator

(l) Define an object that encapsulates how a set of objects interact. Promote loose coupling by keeping objects from referring to each other explicitly. Answer: Mediator

Question 3. (2 points each) Which design pattern is most useful for each of the following situations?

(a) Suppose you are writing some code that will need to search through various sets of data. You'd like to write a single search method that you can use for all the searches. However, in different data sets, you may use a linear search, binary search, hash search, etc. All of these require different ways of initializing the table, different ways of searching, and different ways of adding or removing an element. What design pattern should you use to implement search()?
   Answer: Strategy.

(b) A database transaction is performed by first gaining access to all of the needed resources, then fetching data, then operating on data, then writing it back to the database, then relinquishing the resources. How these operations are performed depends on what type of database operation it is; however, the same general sequence of steps is always used. Answer: Template method.

(c) Imagine what would happen if there were no air traffic controllers. Each and every aircraft would have to know about every other aircraft in order to avoid collisions with them. This would certainly be a cause for disaster, as it would be far too difficult for each plane to keep track of all other aircraft in the vicinity.

So, to remedy this, we have the air traffic controller. Instead of having all aircraft communicating directly with one another, they simply send the controller their flight data and the controller then decides what other aircraft need to be informed. This is clearly far more efficient, and in this case, much safer, than the alternative. Answer: Mediator. Expert or Observer, –1.

(d) Suppose you are writing a virtual-machine monitor that has to run on several different kinds of computer architecture, and implement the functionality of various OSs on each architecture. For example, you need to be able to run MacOS on Intel or PowerPC, and you need to be able to run Linux on Intel or SPARC. You could write code to implement each function of each OS (e.g., scheduling, file creation) on each OS, but there's a much easier way using this pattern. What is it?
   Answer: Bridge. Adapter, –1.
(e) The code for an MP3 player needs to read the notes for a time step and then render them to an audio oscillator. This process is performed repeatedly for each time step from the beginning to the end of the file. Answer: Interpreter. Iterator OK.

(f) JDBC (Java Database Connectivity) is an abstraction over database-access mechanisms, so that a developer does not have to be exposed to all the nitty-gritty of database-specific driver classes that connect to and read data from the database. The developer only needs to learn interfaces like Connection, ResultSet, and Statement. Their implementation classes hide all the complexities behind this simple interface. Answer: Façade.

(g) UI components are contained in containers. If a JPanel can contain other components, it must be a Container. But if we want to add it to the content pane, it must also be a Component. How can a container be a component?

Answer: Composite.

(h) In an inventory, some items go on sale and are sold at discounted prices. The getPrice() method gets the price of an Item, whether or not it is on sale. How can we send the same getPrice() message to a regular-price or sale item, and have it return the appropriate price for each?

Answer: Decorator. Have the getPrice() method for a discounted method apply the appropriate discount, and then call the getPrice() method for Item. Strategy, –1.

(i) When items are added to an invoice, the total amount shown on the GUI should change at the same time that the item is added.

Answer: Observer.

(j) A student is admitted as a freshman. Later on, the same student may become a graduate student or a TA, each of which has separate attributes and responsibilities (e.g., plan of work, pay rate). All of these attributes and responsibilities should be associated with the same Student object.


Question 4. (a) (8 points, 1/2 point per blank) Provide the missing code for CanvasStateStack from Lecture 24. On p. 12 of the notes for that lecture, the UndoRedoHandler is defined. It refers to the private inner class CanvasStateStack, given below.

private class CanvasStateStack
{
    //this class is just an Adapter for a Stack that
    //that holds Canvas.State objects
}

This class is just an Adapter for a Stack that holds Canvas.State objects. Complete the class definition below, assuming that Objects can be pushed onto and popped from the new Stack().

Answer: private class CanvasStateStack
{
    Stack canvasStateStack = new Stack();

    public void push(State canvasState) {

In addition to the code, give two distinct reasons for why the adapter pattern is used, rather than using a Stack directly.

Answer: First, the only objects we want to be put on the stack are the canvas states, and the only thing we want to return are the canvas states. Since the CanvasStateStack specifies only State objects for the return values, then any code expecting something else from a call to one of the methods will get an error. This makes sure that the user adheres to these return-value conditions.

The other reason is that the user only needs the three methods push, pop, and peek. The Stack class has these three methods, plus several others. These methods are not needed by the user, and the functionality they provide is not something the designer wants to. Thus the user is clear about the messages that a CanvasStateStack can handle.

Question 5. A Singleton can be created by returning it from a creation method the first time it is called, and on all other calls, returning null. Here is a class for a printer spooler, where only one instance can ever exist.

(a) (6 points; 1 point per blank) Fill in the blanks in the code.

```java
class iSpooler {
    static boolean instance_exists = false;
    private iSpooler() {};
    static public iSpooler Instance() {
        if (! instance_exists) {
            instance_exists = true;
            return new iSpooler(); // only callable from within class
        }
        else
            return null; // return no further instances
    }
}
```

(b) (2 points) Write a line of code that will create the spooler.

Answer: iSpooler pr1 = iSpooler.Instance();

(c) (2 points) What is the name for this style of instance creation?

Answer: Lazy initialization.

Question 6. The code below defines a Fan and a Light. Our objective is to develop a Switch that can turn either object on or off. The Fan and the Light have different interfaces, which means the Switch needs to operate independently of the actual commands (turnOn, startRotate, etc.). To achieve this, we need to parametrize each Switch with the appropriate
command. Obviously, the Switch connected to the Light will have a different command than the Switch connected to the Fan. The Command class has to be abstract or an interface for this to work.

When the constructor for a Switch is invoked, it is parametrized with the appropriate set of commands.

When the flipUp() and flipDown() operations are called, they will simply cause the appropriate command to execute(). The Switch will not have to know what happens as a result of execute() being called.

(a) (5 points, 1/2 point per blank) Fill in the blanks in the code below.

```java
public interface Command {
    public abstract void execute();
}

class Fan {
    public void startRotate() {
        System.out.println("Fan is rotating");
    }
    public void stopRotate() {
        System.out.println("Fan is not rotating");
    }
}

class Light {
    public void turnOn() {
        System.out.println("Light is on ");
    }
    public void turnOff() { ... }
}

class Switch {
    private Command UpCommand, DownCommand;
    public Switch(Command Up, Command Down) {
        UpCommand = Up; // concrete Command registers with the invoker
        DownCommand = Down;
    }
    void flipUp() { UpCommand.execute(); }
    void flipDown() { DownCommand.execute(); }
}

class LightOnCommand implements Command {
    private Light myLight;
    public LightOnCommand (Light L) {myLight = L;}
    public void execute() { myLight.turnOn(); }
}

class LightOffCommand ...
    // similar to LightOnCommand
    ...
}

class FanOnCommand implements Command {
    private Fan myFan;
    public FanOnCommand (Fan F) {myFan = F;}
    public void execute() { myFan.startRotate(); }
}

class FanOffCommand ...
```
// similar to FanOnCommand
...
}

(b) (9 points, 1/3 point per blank) Next, we need to use this functionality. Finish the following class, which creates objects of the classes defined above, and uses them when the switch is invoked.

```java
public class TestCommand {
    public static void main(String[] args) {
        Light testLight = new Light();
        LightOnCommand testLOC = new LightOnCommand(testLight);
        LightOffCommand testLFC = new LightOffCommand(testLight);
        Switch ls = new Switch(testLOC, testLFC);
        ls.flipUp();
        ls.flipDown();
        Fan testFan = new Fan();
        FanOnCommand foc = new FanOnCommand(testFan);
        FanOffCommand ffc = new FanOffCommand(testFan);
        Switch fs = new Switch(foc, ffc);
        fs.flipUp();
        fs.flipDown();
    }
}
```

Question 7. Suppose we want to use the Visitor pattern to implement “mixed-mode” arithmetic operations using double dispatching. The type of the result is the “more general” of the type of the receiver and operand. For example, if we are multiplying an integer times a float, the result will be a float.

The + method needs to send a message to the operand, which will be implemented differently depending on the operand’s type. The receiver of the original message becomes the parameter for this call. For example, the + method in class Fraction can be implemented as

```ruby
def +(aNumber)
    aNumber.sumFromFraction self
```

(a) (5 points) How would you implement the visitor methods (e.g., sumFromFraction, sumFromFloat)? Describe in general and give an example.

You may assume that there are predefined methods for converting between numeric types (like Ruby’s “to_i” and “to_f” methods, for converting the receiver to an integer or a float, respectively).

**Answer:** In general, you want to convert the “less general” of the receiver and operand to the “more general” type and then carry out the arithmetic operation. Only in the visitor method can this conversion be performed, because when this method executes, the classes of both the operand and receiver are known.

For example, the visitor method sumFromFloat would be implemented like this in the Integer class:

```ruby
def sumFromFloat(aFloat)
    aFloat + self.to_f
```

Here the parameter is not a float and therefore needs to be converted. But in the Float class, the sumFromInteger method is interpreted like this:
```ruby
def sumFromInteger(anInteger)
anInteger.to_f + self
end
```

(b) *(3 points)* Give the sequence of method calls that would take place in adding a Fraction to an
Integer, e.g., \(5 + \frac{4}{9}\).

*Answer:* \(5.\) \text{sumFromFraction}(4/9)

\[
\frac{4}{9} + 5.\text{to_Fraction}
\]

Then the + method can add \((5/1)\) and \((4/9)\).

(c) *(3 points)* What about the reverse, e.g., \(\frac{4}{9} + 5\)? [The expression was printed backwards
on the exam.]

*Answer:* \((4/9).\) \text{sumFromInteger}(5)

\[
5.\text{to_Fraction} + (4/9)
\]

Again, this has a receiver in the Fraction class, so Fraction's + method ultimately adds the values
and returns the result.

(d) *(2 points)* Why do you think this is called “double dispatching”?

*Answer:* Because two methods are always called: the original method, and the method that the
receiver subsequently sends to the argument.

(e) *(3 points)* If there are \(n\) numeric types and \(m\) operations, how many accept and visit...
methods will there be altogether?

*Answer:* For each of the \(n\) numeric operand types, there is one accept method per operation
and \(n\) visitor methods per operation. Since there are \(m\) operations, we can multiply both of these
numbers by \(m\): The total is thus \(mn\) accept methods and \(mn^2\) visit... methods. Expressed
differently, the total number of methods is \(n(m+mn)\).

(f) *(5 points)* This particular implementation of arithmetic operations is much better suited to a
dynamically typed late-binding language like Ruby than it is to a statically typed early-binding
language like Java. Explain why.

*Answer:* In a statically typed language, you usually know the operand types in advance because
of the types of variables that hold them. So there is no need to determine this dynamically by
calling visit... methods. Also, in Java, the arithmetic types are primitive types, and you can't add
methods to a primitive type.

<table>
<thead>
<tr>
<th>Grade summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
</tr>
<tr>
<td>93</td>
</tr>
<tr>
<td>Mean</td>
</tr>
<tr>
<td>73</td>
</tr>
<tr>
<td>Quartiles</td>
</tr>
<tr>
<td>(65, 76, 83)</td>
</tr>
</tbody>
</table>