1. MResearch the Kazaa file-sharing service. Diagram and discuss its software architecture. Be sure to consider how the “supernodes” fit into the architecture.

Question asked for a diagram. I was looking for a diagram like what we've learned about (e.g., UML); class diagram seemed most appropriate. Supernode inheriting from regular node seemed like (part of) a good model.

2. A number of people said Client/Server was a specialized form of Repository. The textbook made that statement about the one example they were discussing (a client/server database). However, in general a Client/Server architecture is not restricted to database applications only. So then, Client/Server is more general than Repository since Repository is only data-oriented.

Also, answers were expected to indicate that Repository architecture has limited and targeted set of services it provides (data-oriented ones); whereas Client/Server had no restrictions (indeed, the diagram just says Service1(), etc.).

3. Which software architecture do you think most closely resembles your Wildfire or Drought system and why?

Many people seemed to only consider Client/Server vs. Repository (previous question should not have limited your consideration). I agree that Client/Server is more appropriate than Repository but expected some discussion of at 3-tier architecture in your consideration since that seems applicable (client tier, web service tier, NIFC/NOAA raw data tier).

4. Some feel that the Observer object pattern is actually a client/server relationship by a different name. Discuss how these are similar and/or dissimilar in the context of two applications. First, an X windows system is comprised of an Xserver that controls the display and communicates to various Xclients (like xterms, xclocks, etc.) events like “your window is not obscured anymore, repaint it”, etc.. Second, an Excel spreadsheet containing a table of values and two charts (a barchart and a piechart) simultaneously depicting the values. If a value in a cell is changed, the charts are automatically updated.

Looking for a commitment about your view of these being genuinely different or the same with only different names. Description/analysis should cover both contexts. It is hard to believe that cells in Excel are “servers” and that charts are clients, so the described behavior must due to something like the Observer pattern. Normal client/server relationships like file servers or webservers involve requests that are initiated by the clients and responded to by the servers. Xwindows basically follows this model. Xclients request that the Xserver change their display in some way. There is an element of Observer involved in that the Xserver sends Xclients information about keyboard/mouse input. But then the Xclients followup by making additional requests. So, it seems – to me – that Observer and Client/Server are different, but that some Client/Server architectures include an Observer aspect (but
5. Explain when the Singleton pattern is useful. Write out the code for the pattern and explain how the code achieves the goal of the pattern. Explain how this pattern is or is not applicable in a multi-threaded situation.

Most people got the code right, a couple forgot that the instance reference MUST be static and the getInstance method should be static. Too many people forgot to mention anything about the pattern in a multi-threaded context. Some simply said that the idea of a Singleton was good in a multi-threaded context, but the question was asking more if THIS pattern was viable. The one as written in class is NOT thread-safe.

6. (Student submitted question (also in text)) What are six design patterns and what do they do?

Chapter 8 of your text describes these.

7. You have been given the task of designing a computer system to display and print shapes. The type of resolution to use to display and print the shapes depends on the computer that they system is currently running on: the speed of the CPU and the amount of memory that it has available. Your system must be careful about how much demand it is placing on the computer. The challenge is that your system must control the drivers that it is using: either a family of low-resolution drivers (print and display) in a less-capable machine or a family of high-resolution drivers in a high-capacity machine.

The job of determining which specific driver to create can be simplified by employing a design pattern. Name the applicable pattern and draw the class diagram for this particular context.

The most appropriate is the Abstract Factory pattern. It is a creational pattern and we're needing to create drivers. Also, we are creating drivers from different “families”. The diagram is like the textbook example; just make sure that the factory is where you distinguish between HighRes and LowRes.

A number of people said Strategy is best. If the machine power changed frequently then I would agree, indeed then a combination of Strategy and Factory would be good. But given that this decision of machine capability is really only made once per system startup then Strategy is less appropriate.

8. (Question from text.) Apply the appropriate forward engineering transformations to the associations below. Assume that all associations are bidirectional and that they can change during the lifetime of each object. Write the source code to manage the associations, including class, field, and method declarations, method bodies, and visibility.

Too many people didn't implement bidirectional associations. So, for the Folder class for example, there must be a collection field for Messages, but also a reference to the Mailbox that this folder is contained in. Then, there needs to be code to set this Mailbox reference and change it as necessary. In particular, the associations need quite a bit of managing in the event of removing something. See textbook and slides for how to do this specifically.

9. (Question from text.) Consider the following design goals. For each of them, indicate the candidate
pattern(s) you would consider to satisfy each goal.

- Given a legacy banking application, encapsulate the existing business logic component.
  
  Adapter: “legacy” is the key here.

- Given a chess program, enable future developers to substitute the planning algorithm that decides on the next move with a better one.
  
  Bridge: Future implementations of same abstraction(algorithm)

- Given a chess program, enable a monitoring component to switch planning algorithms at runtime, based on the opposing player's style and response time.
  
  Strategy: changing implementations on the fly due to changing circumstances

- Given a simulation of a mouse solving a maze, enable the path evaluation component to evaluate different paths independently of the types of moves considered by the mouse.
  
  Command: Encapsulate information about moves for further processing/analysis (redo,replay,undo,etc.).

10. Reverse engineer the following code to produce a class diagram. (You don't need to diagram attributes or methods, just classes and relationships). Then identify which design pattern is being utilized.

```java
class Client {
    public static void main(String[] argv) {
        Shape r1, r2;
        Drawing dp;

        dp = new V1Drawing();
        r1 = new Rectangle(dp, 1, 1, 2, 2);

        dp = new V2Drawing();
        r2 = new Circle(dp, 2, 2, 3);

        r1.draw();
        r2.draw();
    }
}

abstract class Shape {
    abstract public draw();
    private Drawing _dp;

    Shape(Drawing dp) { _dp = dp; }  
    protected void drawLine(double x1,double y1, double x2, double y2) {
        _dp.drawLine(x1,y1,x2,y2);
    }
    protected void drawCircle(double x, double y, double r) {
        _dp.drawCircle(x,y,r);
    }
}
```
abstract class Drawing {
    abstract public void drawLine(double x1, double y1, double x2, double y2);
    abstract public void drawCircle(double x, double y, double r);
}

class V1Drawing extends Drawing {
    public void drawLine(double x1, double y1, double x2, double y2) {
        DP1.draw_a_line(x1,y1,x2,y2);
    }
    public void drawCircle(double x, double y, double r) {
        DP1.draw_a_circle(x,y,r);
    }
}

class V2Drawing extends Drawing {
    public void drawLine(double x1, double y1, double x2, double y2) {
        DP2.draw_a_line(x1,y1,x2,y2);
    }
    public void drawCircle(double x, double y, double r) {
        DP2.draw_a_circle(x,y,r);
    }
}

class Rectangle extends Shape {
    private double _x1, _y1, _x2, _y2;
    public Rectangle(Drawing dp, double x1, double y1, double x2, double y2) {
        super(dp);
        _x1 = x1;  _y1 = y1;  _x2 = x2;  _y2 = y2;
    }
    public void draw() {
        drawLine(_x1,_y1,_x2,_y2);    drawLine(_x2,_y1,_x2,_y2);
        drawLine(_x2,_y2,_x1,_y2);    drawLine(_x1,_y2,_x1,_y1);
    }
}

class Circle extends Shape {
    private double _x, _y, _r;
    public Circle(Drawing dp, double x, double y, double r) {
        super(dp);
        _x = x;  _y = y;  _r = r;
    }
    public void draw() {  drawCircle(_x,_y,_r);  }
}

class DP1 {
    static public void draw_a_line(double x1, double y1, double x2, double y2) {
        // some concrete implementation omitted for space here
    }
    static public void draw_a_circle(double x, double y, double r) {
        // some concrete implementation omitted for space here
    }
}

class DP2 {
static public void draw_a_line(double x1, double y1, double x2, double y2) {
     // some other concrete implementation omitted for space here
}
static public void draw_a_circle(double x, double y, double r) {
     // some other concrete implementation omitted for space here
}

Client connects to Drawing and Shape.

Circle and Rectangle inherit from Shape.
V1Drawing and V2Drawing inherit from Drawing.

Shape has an association with Drawing.
V1Drawing has an association with DP1.
V2Drawing has an association with DP2.

This is an example of the Bridge pattern. There are multiple “versions” of how to do drawings.