Answer each of the questions below, demonstrating a full understanding of the concepts involved in each question. While you may discuss the questions with your classmates, your responses must be in your own words and your designs must be your own.

1. (5 points) Complete traversals of binary tree structures are commonly used for such varied purposes as arithmetic expression interpretation (e.g., infix and postfix), graphical hidden surface determination (e.g., BSP trees), and sorting algorithms (e.g., binary insertion trees). In view of the recursive nature of the binary tree structure, identify which design pattern lends itself to binary tree traversal. Include a UML diagram illustrating this application of the identified design pattern.

2. (5 points) Aviation schools often employ flight simulators and air traffic control (ATC) simulators in a networked configuration that allows pilot and ATC trainees to interact with each other during simulations. Thus, for example, dozens of pilot trainees may use separate flight simulators, all modeling the same airport at the same time, while being monitored by several ATC trainees using their own simulators to coordinate the flights around the airport being modeled. Explain which design pattern lends itself to the difficulties associated with this application scenario. Suggest ways in which the use of this pattern facilitates the extension of each type of simulator’s functionality.

3. (5 points) There are two principal implementation approaches for the Observer pattern. In the first approach, all Observers are automatically notified whenever the Subject experiences any state changes and each Observer determines whether or not to retrieve any of the new state information from the Subject. In the second approach, the Subject automatically transmits all state-change information to every Observer. Discuss the relative advantages of each approach, providing an example for which each approach would be appropriate.

4. (5 points) The Bridge pattern appears to have much in common with the Strategy pattern. Note the similarity between their GOF intent statements:

   **Bridge Intent:** “Decouple an abstraction from its implementation so that the two can vary independently.”

   **Strategy Intent:** “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”

Also note the similarity between their UML diagrams:

Specify the deciding factors that determine which of these two patterns should be applied in a particular situation, emphasizing the reasons why the Bridge pattern is considered to be structural while the Strategy pattern is considered to be behavioral. Supply examples to demonstrate this distinction.
5. (5 points each) Consider the development of a software package that must account for the different procedures for handling various commercial transactions (sales tax computation, shipping costs, fund transfers, leasing restrictions, legal liability, credit regulations, etc.) in each of the fifty states.

   a) One possibility would be to use switch statements to handle these procedures. For example, when determining a state’s sales tax rate, a routine could look like this:

   ```c
   float getSalesTaxRate(state s, merchandiseType m)
   {
     switch (s)
     {
       case Alabama:
         if (merchandiseType == PrescriptionDrugs)
           return 0.0;
         else
           return 0.04;
         break;
       case Alaska:
         return 0.0;
         break;
       case Arizona:
         if ((merchandiseType == Food) ||
             (merchandiseType == PrescriptionDrugs))
           return 0.0;
         else
           return 0.056;
       break;
     }
     // And so forth...
   }
   ```

   Explain what’s wrong with using this approach in this project, specifying which aspects of the traditional software development lifecycle (requirements gathering, analysis, design, implementation, testing, deployment, and maintenance) would be adversely affected by this switch-based approach.

   b) A second possibility would be to exploit inheritance, with each state’s way of handling commercial transactions encapsulated within the subclass associated with that state. For instance, the subclasses could look like:

   ```c
   class AlabamaCommerce: public StateCommerce
   {
     public:
       float getSalesTaxRate(merchandiseType m);
       float getShippingCosts(merchandiseType m, float weight, priority p);
       float getFundTransferLimit(fundTransferType f);
       // And so forth...
   };
   
   class AlaskaCommerce: public StateCommerce
   {
     public:
       float getSalesTaxRate(merchandiseType m);
       float getShippingCosts(merchandiseType m, float weight, priority p);
       float getFundTransferLimit(fundTransferType f);
       // And so forth...
   };
   // And so forth...
   ```

   Again, explain what’s wrong with this inheritance-based approach with respect to the traditional software development lifecycle.

   c) Explain how the Strategy pattern, which favors aggregation over inheritance, effectively addresses the problems that you observed in the two previous questions. Include a UML class diagram that illustrates how the Strategy pattern would be applied here.
6. (5 points) Notice that the State pattern (illustrated at left below) and the Strategy pattern (illustrated at right below) have exactly the same class diagram. Explain, then, the fundamental difference between these two patterns with respect to their intent.

```
\begin{center}
\includegraphics[width=\textwidth]{state_strategy.png}
\end{center}
```

7. (5 points each) The Template Method pattern and the Strategy pattern are frequently confused by some software developers. The principal difference between these two patterns is that the Strategy pattern allows for varying entire algorithms, while the Template Method pattern dictates the basic structure of the algorithm, while allowing the implementation of the details to vary.

a) In C++ STL, an array-sorting algorithm is available:

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

void main()
{
    int a[7] = {87, 25, 93, -16, 4, 4, 10};
    sort(a, a+7);
    for (int i = 0; i < 7; i++)
        cout << a[i] << " ";
}
```

If you were creating several new classes of objects that would be employing this array-sorting algorithm as part of a larger program, explain how you would be using the Template Method pattern (and not the Strategy pattern) to accomplish this implementation.

b) The Dependency Inversion Principle (DIP) states that high level or low level modules should not depend upon each other, instead they should both depend upon abstractions. Explain whether the Strategy and Template Method patterns adhere to the DIP.

This assignment is due on your drop-box by 9 AM on Thursday, April 9, 2009.