CSC 417 - Project 2

Maanasa Thyagarajan
mthyaga@ncsu.edu

Bodhit Chattopdhyay
bchatto@ncsu.edu

Sanchit Razdan
srazdan@ncsu.edu

1. Abstract:

This document talks about the different abstractions that can be used for the program written (dom filter of the pipe). The document discusses 10 abstractions that could be useful for the dom filter. The 10 abstractions are discussed based on their ranks, with the most useful abstraction discussed first and the least useful abstraction discussed at the very end. The document later discusses the team's experience with the abstractions with respect to the filter, and their recommendations for the use or avoidance of the listed abstractions.

2. Introduction:

Pipes and filters are one of the most commonly used patterns in Unix based Operating Systems. They help break down large projects into manageable chunks and do them in stages. Pipes connect one filter to another (the output for the first filter becomes the input for the next filter).
Filters are programs which help transform the input data into some useful output which would be used as input by the next filter in the pipe.

The main rules of thumb of pipes and filters include:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

This document talks about one of the programs (dom) in the whole pipe (monte_carlo | brooks2 | dom | bestrest | super | rank) and the way different abstractions could be used in the program or how the program could use different abstractions.

3. Program:

The domination program helps to summarize our problem to a larger problem. It helps figure out dominant objects based on some dependent factors with respect to other objects. There are 2 measures for achieving domination, namely Binary Domination and Indicator Domination.

4. Abstractions:

Abstraction is a technique for arranging complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. The following abstractions have been taken into consideration for the program:

1. **Composite**:
   The composite pattern focuses on the composition of objects in a tree-like hierarchical structure so that the client can treat objects in a uniform manner. It is used when a objects are to be represented in a hierarchical structure.

   The input of the dom program is a csv file which includes a table of data. The table itself is made up of rows, which are in turn made of dependent and independent variables. The dependent and independent variables have cells inside them. This whole structure resembles a tree-like hierarchical structure, hence showing a composite pattern.

   The csv file is of the following hierarchy:

(See next page)

Table
- Row
  - Independent Variable
- Cell
  - Dependent Variable
- Cell

This shows how the composite pattern has been used in the program. Since the input for the program is of this nature, this abstraction is of importance for the program.

Intent:

- Create objects in tree structures to show whole-part relationships between them
- There are many "has-a" relationships going up the "is-a" hierarchy
- Use recursion in composition

Rules of thumb:

- Relies on recursion in creating objects so that there can be an open-ended number of objects
- Composite pattern is able to utilize Chain of Responsibility to let components gain access to their parent's global properties

2. **Visitor**:
The visitor pattern focuses on the operations to be done on the elements of the structure. It lets you define new operations without changing the classes of the elements. It is used when an object structure includes many classes and you want to perform an operation on the elements of that structure that depend on their classes.

In the context of the dom program, the element is the row and the operation done is done on two subsequent rows. For example, the first and the second row of the table are used and the operation is done on them. After the operation is done, the program goes on to the next two subsequent rows and performs an operation on them. In this case, even though the object structure may create other classes or elements, the operation can be changed without having an effect on the actual element.

This shows the use of the visitor pattern in the program. Since the operation is performed on the various elements of the objects (rows in this case), the abstraction is an important abstraction used in this program.

Rules of thumb:

- This pattern is a great technique to restore any lost information without the use of dynamic casting
- Visitor pattern can apply an operation over a composite pattern

3. **Iterator**:
The iterator pattern provides a way to access elements of an aggregated objects sequentially without exposing how they are internally stored. It is used in situations where the program wishes to access the object's contents without knowing how it is internally represented.

For the dom program, the program accesses the elements of the row object, which are the dependent variables, without knowing how they are stored in the table and how they are internally represented in the csv file. The program only worries about the content of the element, rather than its representation. For example, the dom program only worries about the contents of the weight, acceleration and mpg of each row. It does not care about the way the row is internally represented.

This shows the use of the iterator pattern in the program. Since with each iteration in the program, the contents of the elements of the object (row in this case) are accessed by the program, the abstraction is of importance and is used in the program.

Intent:

- Allow access to elements of an aggregate object without revealing that object's internal state/data
- This abstraction makes it possible to decouple collection classes

Rules of thumb:

- Iterator can traverse a composite pattern
- Often used with memento, which can help in snapshotting the state of an iteration

4. **Blackboard**:
The blackboard pattern provides a loosely coupled system, perhaps using different programming languages, where agents react to observations by making conclusions that define higher-level observations. It is used when a complete search on a data structure is not feasible or there is a need to experiment with different algorithms in the same sub-task.

   In the case of the dom project, this pattern can be implemented in a way where the data can be stored in a centralized database and many users can add more data, row by row. The data can be reviewed in terms of accuracy and duplicacy.

   With the increase in the data itself, the program will get slower. Also, since there is only one prefilled database and one thing to figure out for each row, different algorithms would not be used or would not be even required in this program. Hence it does not make sense to use this pattern as an abstraction for the program. Using this pattern as an abstraction would rather complicate the program rather than simplifying it.

5. **State**:
The State Pattern is a behavioral design pattern in which the behavior of an object varies based on what its internal state it. This pattern can be an alternative in scenarios where many if/else or switch statements are used to alter the behavior of an object.

   With regard to the dom program, this pattern can be implemented so that the result of comparing two columns at a time is stored as the state, and that state dictates how that object is altered going forward or what other states it travels through to reach its final state. This way, once the final state is reached (meaning dom has done its job), it can move through the pipe.

   Though this pattern would be viable for the above reasons, we will not be implementing this in our program. Since we are only manipulating one part of the pipe mechanism, using state transitions will only complicate our program.

   Intent:

   - Allow an object to automatically alter its behavior upon an internal state change
   - Can be represented by an object-based state machine

   Rules of thumb:

   - State objects are many times implemented as singletons objects
   - State pattern builds on the strategy pattern

6. **Observer**:
The observer pattern is intended for use with objects that are dependent on one another. With the use of "observers," "listeners," and "notifications," when a change occurs in one object, all other objects that are dependent on that one also change accordingly.

   In context of the dom program, since the outcome of this program will act as the input of another program through pipe-and-filter, we could implement the observer pattern by creating multiple instances of the manipulated object before passing it through the pipes. That way, only the object that is intended to be manipulated will be as it goes through the pipes.

   After considering all options, we will not be moving forward with the observer pattern. Since the pipe-and-filter mechanism will only take in the object we feed it, we won't need multiple instances and therefore don't

have to notify any objects upon changes.

Intent:

- Pattern defines one-to-many relationship/dependencies between objects
    - So that when one object changes internal state, all dependent objects are notified an automatically updated
- Often used as the "view" component of the Model-View-Controller (MVC) pattern

Rules of thumb:

- Communication occurs through "observer" and "subject" objects, along with a "mediator" to moderate the relationship/communication
- Utilizes Chain of Responsibility to deal with object dependencies

7. **Memento**:
The memento pattern is useful in storing the external state of an object in case that state needs to be restored later. This is done through the use of "originator," "caretaker," and "memento" objects. The originator is the original object with its internal state. The caretaker then asks the originator for a memento before manipulating it, so if the original state needs to be restored, the caretaker can return the memento to the object.

   Because the Dom program is part of a pipe-and-filter mechanism that passes the output of one program into the input of another, the memento pattern could be useful in case one of the programs affects the passing object/value in a way we don't want it to, so a memento object could be stored in another class to restore a value that has been incorrectly manipulated

   Though this pattern could be helpful in case of functionality failures, we decided that it would be more viable to test and perfect our solution in the dom program rather than spending time on potential errors.

8. **Factory**:
The factory pattern allows a class to define creating a new object, but delegates the task of instantiating the appropriate object to its subclasses. In this creational pattern, the user can call a factory method rather than a constructor to create an object.

   In context of the Dom program, since the dom score is determined after the comparison of multiple rows and columns, the class containing the factory method can determine the necessary scores/outcomes from the column. The factory method can then pass the result to the appropriate subclass, which returns the dom score based on that passed in result.

   We will not be utilizing the factory pattern. It will be simpler to have a method contained within one class that calculates and determines the dom score rather than going through multiple classes and dealing with encapsulation.

9. **Flyweight**:
The flyweight pattern allows the data or state of an object to be shared with other objects, thus eliminating memory usage. This pattern is commonly used with data structures that store parts of the object's state and are passed to an object when necessary.

   With regard to the dom program, since memory is stored and passed through pipes, it may be helpful to implement the program in a way that reduces memory usage. This could speed up the program and allow data to flow easily between pipes.

   We will not be implementing this pattern in our program. Creating external data structures to story states of an object would only complicate our problems, since we can just utilize simple methods and return values.

10. **Proxy**:
The proxy pattern can be useful when wanting to control access to an object or when wanting to

provide additional functionality to an object. There are 3 types of Proxy patterns:

a. Remote Proxy: A local object can be used for manipulation rather than the original or remote object itself. By making changes to the local object, the state of the original object can also be changed.

b. Virtual Proxy: When an object is complex or heavy, virtual proxy can provide a skeleton representation for the object rather than having to retrieve and load a larger object

c. Protection Proxy: Works to control access to the object in question

In context of the dom program, a remote proxy pattern could be useful so that instead of having to pass the actual value or object through pipes as input, a proxy can be passed along. This might save runtime, and it would ensure that the original object is still manipulated as necessary.

Though this pattern could be useful for pipes-and-filters in general, we are not passing any heavy or complex objects/values, so there is no need to utilize proxy objects.

5. Epilogue:

The abstractions used in the program were Composite, Visitor and Iterator pattern abstractions. All 3 abstractions are of importance to the program since all use abstractions are used in the program in some form or the other. Our experience, as a team, with the abstractions was a good experience. Understanding the pros and cons of each of the above 10 abstractions helped the whole team to understand the different abstractions. We also understood the relativity of each abstraction to the program and hence could figure out which 3 were most relative according to us. The research about the abstractions also helped us understand more about the abstractions, their structure, uses, rules of thumb and the type of problems each abstraction could solve. Replicating the program in another language helped us understand the application of the abstraction in different situations. It also helped us understand the practical use of the abstractions.

Following are some of the situations where each abstraction should be used:

1. Composite:

a. When application has hierarchical structure and needs generic functionality across the structure.
b. When application needs to aggregate data across a hierarchy.
c. When application wants to treat composite and individual objects uniformly.

2. Visitor:

. When you want to reduce proliferation of code which is only slightly different when data structures change.
a. When you want to apply the same computation to several data structures, without changing the code which implements the computation.
b. When you want to add information to legacy libraries without changing the legacy code.

3. Iterator:

. When you want to access the contents of a collection without exposing its internal structure.
a. When you want to support multiple simultaneous traversals of a collection.
b. When you want to provide a uniform interface for traversing different collection.

6. Maximum Marks Expected:

The following is the expectation of marks from this project:

1. Completion of project: 10 points
2. Language used: Python (0 extra points)
3. Program done: Dom (0 extra points)
4. Abstractions used: Iterator, Visitor and Composite Pattern (0 extra points)

Hence, from this section of the project, the maximum marks expected are 10.