CSC 417 - Project 2

Maanasa Thyagarajan
mthyaga@ncsu.edu

Bodhit Chattopdhyay
bchatto@ncsu.edu

Sanchit Razdan
srazdan@ncsu.edu

ABSTRACT

This document talks about the different abstractions that can be used for the program written (monte_carlo filter of the pipe). The document discusses 10 abstractions that could be useful for the monte_carlo 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.

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 (monte_carlo) 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.

PROGRAM

The goal of monte-carlo simulation is to randomly generate a set of samples to run an algorithm over. monte_carlo methods vary, but tend to follow a particular pattern:

1. Define a domain of possible inputs
2. Generate inputs randomly from a probability distribution over the domain
3. Perform a deterministic computation on the inputs
4. Aggregate the results

There are two important points:

- If the points are not uniformly distributed, then the approximation will be poor.
- There are a large number of points. The approximation is generally poor if only a few points are randomly placed in the whole square. On average, the approximation improves as more points are placed.

In other words, the monte_carlo program takes a range of minimum and maximum values for different attributes and produces random values between those ranges for each. Those values are then output similar to JSON style output.

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 objects are to be represented in a hierarchical structure.

   The monte_carlo program creates an object called Attribute, which contains field values of minimum and maximum values, as well as a name value indicated by the original code provided for this project. For example, "pomposity" has a minimum value of 1 and a maximum value of 10.

   The MonteCarlo instance created upon running the program contains a list (AttributeList) of Attributes, which are accessed to obtain their minimum and maximum values.

   Hierarchy in MonteCarlo:

   MonteCarlo
   AttributeList
   Attribute
   (fields: min, max, name)

This shows how the composite abstraction is used in monte_carlo. By creating types of Attribute objects and storing them as a part of the MonteCarlo instance, we can access the various values from each attribute necessary to produce the desired output.

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. **Generator**

The generator pattern sees the use of a "generator" to control iterations within a loop. Moreover, all iterators are considered to be generators. Generators look like functions but behave like iterators- that is to say, generators can be called with parameters like a function, but they control iterations through a loop to return or extract one value at a time.

In the monte_carlo program, there is a function called "generator()." Java does not have a built in generator, so it is actually yielded by the iterator object created within the program. The iterator is used in the function to traverse the AttributeList (similar to an ArrayList) and generates one value at a time corresponding to each attribute.

Intent:

- Control iteration behavior within a loop
- Yield one value or return object at a time, which decreases storage costs

Rules of Thumb:

- Should be able to be called like a function
- Should be able to accept parameter

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.

In the MonteCarlo patterns, an iterator is created and used to access objects within an AttributeList, which is similar to an ArrayList. The iterator has overridden methods such as next() and hasNext() that are used by the MonteCarlo instance to traverse the AttributeList and extract values from the objects in that list.

This abstraction is used in the patterns since values of every attribute are accessed by traversing the list data structure in the program.

Intent:

- Used to traverse elements of a collection
- Traverse elements without exposing underlying representation of those elements/objects

Rules of Thumb:

- Must implement same standard interface
  - next() and hasNext() methods
- Iterator object should keep track of current position and length of the collection

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.

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

In the case of this 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. 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 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. Singleton

The singleton pattern is a very simple design that involves only one class to instantiate itself. It makes sure that it creates not more than one instance of itself and at the same time also provides a global point of access to that instance.

Intent:

- Ensures that a class only has one instance.
- Provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use.4

Rule of Thumb:

- State objects are often singletons.
- The advantage of using Singleton patterns that one is always sure of the number of instances being used. One can also change and manage the number of instances in the Singleton.
- Singletons are intended to be used when a class must have only one instance, not more or less.

Cons of Singleton Pattern:

- Singletons hinder unit testing: It is hard to write tests for Singleton as the methods can be very tightly coupled.
- Singletons create hidden dependencies: The singleton can be overused. Moreover, since its reference is not completely transparent while passing to different methods, it becomes difficult to track.

We can use the singleton abstraction by making a single instances of the MonteCarlo object for different algorithms. The variables can be used as global variables for the instance that will be created.

We will not be using this abstraction for this project. As the singleton makes only one instance of the MonteCarlo object it makes the program very implicit, therefore not making it a very explicit abstraction to be used in the program.

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.

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-ViewController (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

In the context of this 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.

7. Command Design Pattern

Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Intent:

- Encapsulates a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and also support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object Oriented call back.

Rule of Thumb:

- Chain of responsibility can use command to represent requests as objects.
- Command and Memento act as magic tokens to be passed around and invoked at a later time.
- Command can use Memento to maintain the state required for an undo operation.
- MacroCommands can be implemented with Composite.

Cons of Command Pattern:

- Using command design pattern may require more effort on implementation,since each command requires a concrete command class, which will increase the number of classes significantly

In the context of this program we can use this abstraction to wrap the data coming in from the file for the monte carlo and make it an object to be used for the algorithms.

We decided not to use this abstraction for this monte-carlo program. As this will require more time and also be harder to implement. This will also increase the number of classes and that is not required.

8. **Decorator**

Decorator pattern allows us to add new functionality to an already existing object without changing its structure. This pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods and their signatures intact.

Intent:

- The intent of this pattern is to add additional responsibilities dynamically to an object.

Rule of Thumb:

- Provides an enhanced interface.
- Decorator enhances an object's responsibilities, which makes it more transparent for the client.
- Designed to let you add responsibilities to objects without subclassing.

Cons for decorator:

- Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
- It can be complicated to have decorators keep track of other decorators.

In the context of this program, we can use the decorator pattern to add new functionality to the monte carlo program. We can add different algorithms and data that can be used to compute more problems through the program.

We decide not to use this program as we are not adding extra functionality to our program right now. It can also get very complicated and we could lose track of the new functionalities that are added.

9. **Private Class Data**

The private class data design pattern seeks to reduce the exposure of attributes by limiting their visibility. It reduces the number of class attributes by encapsulating them in single Data object. It allows the class designer to remove write privilege of attributes that are intended to be set only during construction, even from methods of the target class.

Intent:

- Control write access to class attributes
- Separate data from methods that use it
- Encapsulate class data initialization

Rule of Thumb:

- Private class data design pattern prevents that undesirable manipulation.

Cons for Private Class data:

- A class may expose its attributes to manipulation when manipulation is no longer desirable

We can use this abstraction in our project by adding all the global variables to a private data class. This will separate the data from the actual method for the algorithm and also be a convenient place to store the data.

We decided not to use this abstraction for this project. This is because we don't want to make a separate class for our program as it reduces the visibility of the variables and it will be harder to manipulate the data.

10. **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.

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

The Visitor pattern was considered as a main abstraction for monte_carlo, because data of various objects are accessed without actually changing that data. For example, the MonteCarlo instance access the data in the Attribute object and utilizes that data to generate some other values, but the internal state of the accessed object is never altered.

Though the idea of the Visitor abstraction is used in our program, we didn't include this as the main abstraction because we don't have an explicit Visitor method or object.

EPILOGUE

The patterns used in our implementation of monte_carlo are: Composite, Generator, Iterator. The program utilizes hierarchical objects to store instances of the Attribute within an AttributeList, all encompassed in the created instance of the MonteCarlo object. The Attributes are then accessed in the list through an Iterator object and passed through a Generator to access the values stored within each Attribute. The use of these abstractions should save storage space and reduce runtime because of the use of an iterator and generator.

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. **Iterator**
   a. When you want to access the contents of a collection without exposing its internal structure.
   b. When you want to support multiple simultaneous traversals of a collection.
   c. When you want to provide a uniform interface for traversing different collection.
3. **Generator**
   a. When you want to control the iteration behavior within a loop for an instance of an object.
   b. When you want to yield one value or return object at a time, which decreases storage costs

MAXIMUM MARKS EXPECTED

The following is the expectation of marks from this project:

1. Completion of project: 10 points
2. Language used: Java (0 extra points)
3. Program done: monte_carlo (0 extra points)
4. Abstractions used: Composite, Generator, Iterator

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