Monte Carlo 2a

Using Java

Wil Elias
Computer Science
North Carolina State University
Raleigh, North Carolina
mwelias@ncsu.edu

Nolan McDonald
Computer Science
North Carolina State University
Raleigh, North Carolina
nrmcdona@ncsu.edu

Ian Fisher
Computer Science
North Carolina State University
Raleigh, North Carolina
imfisher@ncsu.edu

ABSTRACTIONS CONSIDERED

1 Letterbox

The letterbox style aims to break up an application into objects that expose only one procedure to other objects that allows them to accept messages. Data and process is hidden, and messages are acted on by the things that accept them. Some messages may not be understood, in which case the object may produce an error or may pass the message on to a more appropriate object to handle the message [2].

By implementing the letterbox into the Monte Carlo algorithm in Java, we could leverage Java classes to break the program up into message passing objects. Each object would have one public method, *dispatch*, and all other class methods would be private.

2 Delegation

The delegation pattern tries to move some implementation outside of some objects. There is a helper object that performs some function for the main object, given that main object as a parameter. When the main object requires that functionality it calls to the helper object, which then performs the actions for it. It may return something or it may just operate on some data for the calling object. For clarity take this example from the Wikipedia page addressing the Delegation Pattern in Kotlin

```
class Rectangle(val width: Int, val height: Int) {
    fun area() = width * height
}

class Window(val bounds: Rectangle) {
    // Delegation
    fun area() = bounds.area()
}
```

In Monte Carlo we found that it might be useful to delegate the actual generation of random numbers to a helper object. That helper object could take a seed as a parameter and generate a random number from the seed given. It could then send that random number back to the main object that called it.

3 Tantrum

In the tantrum abstraction there are two main constraints Firstly, every single procedure and function checks the sanity of arguments and will halt execution when it's arguments are determined to be unreasonable. Second, all code blocks check for all possible errors, passing errors up the function call chain and possibly logging context specific messages when encountering errors [2].

In the data mining pipeline we are developing, following the tantrum style could be useful in maintaining consistency in the data that is piped from one filter into another. Performing sanity checks on the data generated by the Monte Carlo algorithm may allow us to ensure minimum and maximum bounds are observed by the random number generation.

4 REST

REST is an architectural style generally used by interactive network-based applications. The REST style aims to provide be extensibility, decentralization, interoperability, and independent component development. To achieve these goals, a restful system must be built under the following constraints [2].

First, REST implies end-to-end interaction between an active agent and a backend. There is separation between the client and the server the two communicate in a synchronous, request-response form. Second, the server must be stateless. In other words, any requests sent by the client must contain all of the necessary data in order for the server to complete the request. The server does not store context of any outgoing communication. Finally, both the client and server handle resources with a uniform interface. The client may operate on resources with a limited interface that includes creation, modification, deletion, and retrieval [2].
In the Monte Carlo algorithm, we thought the Restful style may be useful for data generation. For example, the client could make a request for *N* number of values within a specified minimum and maximum for each independent variable with an indication for the type of distribution for the random number generator to follow. The data generation would be the sole responsibility of the server and, assuming

the server is more powerful than the client machine, this could cut down on our execution time.

5 State Machine

The state pattern intends to allow an object to alter its behavior when its internal state changes. The object itself is abstract, providing a common interface for state classes and may appear to change its class as its state changes [1]. Our hope with the state machine pattern was to be able to create discrete states representing each of the independent variables we need to generate numbers for the MonteCarlo algorithm.

4 Polymorphism

Polymorphism aims to provide dynamic binding as a resource to the programmer. Allowing anything that contains a type of interface as a parameter is an example of this. This allows a client object to make assumptions about objects beyond its scope that it may need to interact with. This allows client objects to have simpler definitions as well as decouples objects that are related to each other and allows their relationships to vary at run-time.

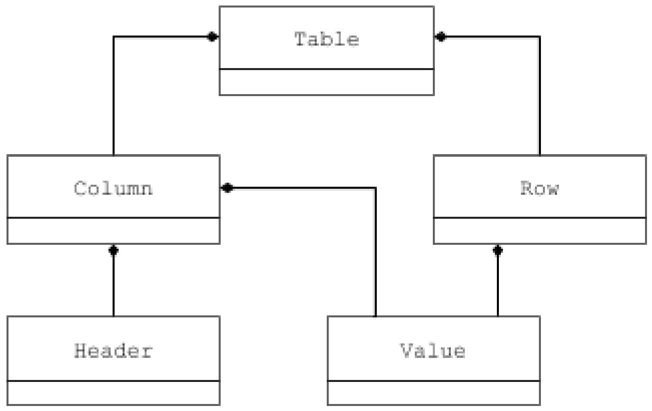
By providing a parameter object that has ubiquitous methods for setting min and max as well as generating the random values between those two variables. For each parameter we could generate in a loop an object. After that in a separate loop we could generate the number of samples per each parameter.

7 Composite

The Composite abstraction allows developers to part-whole hierarchies of objects. It also allows clients of an object or objects to ignore the differences between a single object or a composition of objects. In this way the clients will treat all objects in a composition of objects uniformly. This allows us to simplify client code as it doesn't have to worry about whether its dealing with multiple or singular objects or types of objects [1].

The important part of the Composite pattern is an abstract class that represents both the primitives and their containers. The Composite pattern also makes it easy to add more components to the system, however it can make the system overly general and limits how you can restrict what components are part of a Composite. This means you can't rely on type-checks to enforce constraints. You have to fall back to run-time checks instead [1].

We decided that it could be useful to represent the table of values in this way. Creating components for the rows, columns, values and the table as a whole. That way we could access parts of the table or all of the table without having to worry about how much we are looking at. This would also allow us to easily add functionality into the column or row classes to calculate things such as average, standard deviation, or variance.



8 Factory Pattern

The factory pattern abstraction provides an interface for generating an object but still letting the subclasses decide which class to instantiate[1].

This abstraction is useful when a class can't anticipate the class of the objects it must create. Our idea was to give each independent variable or column its own class with its own rules for generating values in that column. The MonteCarlo could create all of the column generators in the same way, even though they are different classes using the factory pattern.

9 Dataspaces

The dataspaces abstraction applies to concurrent and distributed systems and requires programming under the following constraints. First, there exists one or more units executing concurrently. There also exists one or more data spaces where concurrent units store and retrieve data. Finally, there is no communication between concurrently executing units other than through the data spaces [2].

Dataspaces are often used in parallel programming. By creating workers that operate on that dataspace you can have multiple functions all running in parallel. Here we found that it might be possible to create workers to generate the random numbers for Monte Carlo. This has the potential to accelerate the speed of the filter.

10 Lazy River

The lazy river style aims to make data available in streams that are passed from one function or object to another. The functions or objects in the application act as filters that modify or transform the data stream in some way before handing it off to another. In this style, data is processed from upstream (data sources) down to downstream (data sinks) on a need basis which eliminates problems stemming from too much data at one time [2].

Our plan to use the lazy river included separating the random number generation from the rest of the program and passing off the numbers generated as a data stream to other parts of the program that would organize the data into a table to be piped to other filters within the pipeline. In a multithreaded solution, this style has to potential to speed up the monte carlo algorithm significantly.

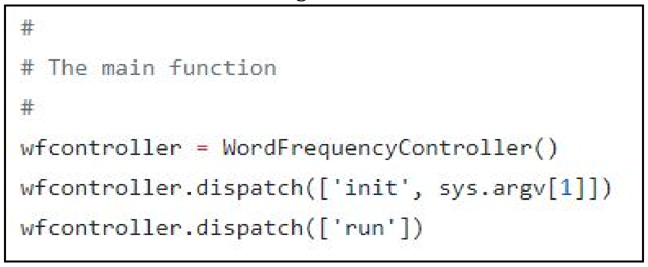
IMPLEMENTATION

After considering the replacement of several filters with a wide variety of the abstractions discussed above, we decided to pursue the Monte Carlo algorithm. This was for a few key reasons. Firstly, of all the stages in the project pipeline, the Monte Carlo one was likely the easiest to fully understand. Though it was similar in difficulty to a few others, the Monte Carlo algorithm best allowed us to brainstorm ways that we might utilize different abstractions, as evidenced by this report's previous section. Furthermore, since our team will continue to consider the filters and abstractions for two more parts of this project, it is of paramount importance to understand each piece of the pipeline. Resultantly, we agreed that starting at the beginning seemed logical, and by experimenting with the Monte Carlo algorithm, we would gain a solid understanding of the data generation that the rest of the filters are based upon.

After choosing to replace the Monte Carlo stage, we reconsidered the wide array of abstractions we had explored,. We then continued to research nuances and some examples, for each to determine which might lend themselves best to our problem. As the first stage in the pipeline, the Monte Carlo algorithm serves the purpose of generating a distribution of random data samples. For each data category, a minimum and maximum value are prescribed and then used as bounds for the generation of sample values. The algorithm also accepts arguments to determine *n*, the number of samples to generate, and *v*, a boolean value for verbosity, which is the only non-numeric attribute belonging to the fifteen data categories. The main components required for the Monte Carlo stage of the pipeline then include setting initial value boundaries, accepting and interpreting command line arguments, generating a set number of random samples, and finally printing that data to standard output. Printing to standard output is necessary in order for the following Brooks filter to receive the data, in accordance with Unix I/O standards. More specifically that data required formatting that matched the Python implementation of the Monte Carlo which was originally provided to our team.

Based upon the main components required for the Monte Carlo stage, we chose the LetterBox pattern as our primary abstraction. A fundamental of the LetterBox pattern is that a problem is broken up into separate "things" that are reasonable for the problem's domain. Additionally, each of these "things" may represent a type of data. This pattern fit our problem which required a few different "things", including argument handling, number generation, and maintaining fifteen separate data categories or attributes. Another pillar of the LetterBox pattern is that each "thing" or object has only one public method to which messages can be sent. This *dispatch* method is the only function that can be called at all for an object, though any object may have several other private or internal methods. The input and output of the *dispatch* function may be limited to a string type, to create a standardized way of inter-object interaction. The messages sent to an object may used to update those objects, or to request calls or actions to be taken by the object. From Lopez's example shown in Figure 1, this messaging system can be used to initialize an object and then invoke its run behavior. Since the Monte Carlo stage included a similar segmentation between initialization and then run behavior, the LetterBox pattern continued to prove itself as a fitting abstract.

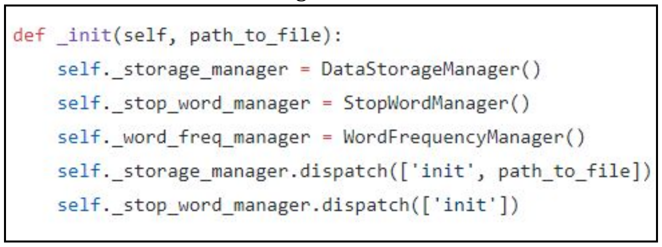
Figure 1



```
#
# The main function
#
wfcontroller = WordFrequencyController()
wfcontroller.dispatch(['init', sys.argv[1]])
wfcontroller.dispatch(['run'])
```

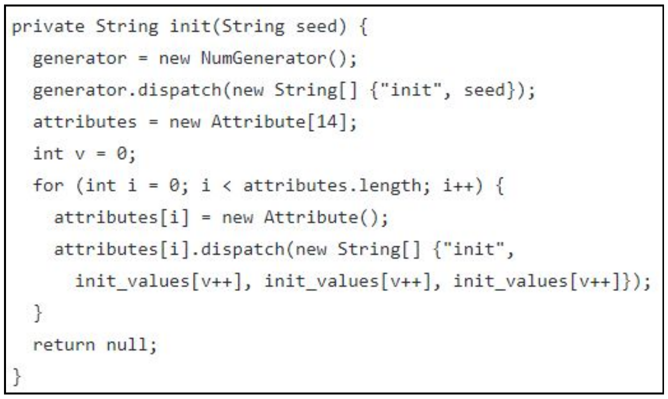
The objects we created during implementation were the *MonteCarlo*, *NumberGenerator,* and *Attribute* classes. The first handles initializing the others objects, as well as acceptancing command line arguments. Demonstrating another nuance of the LetterBox pattern, the *MonteCarlo* object defers some responsibility to those other objects, by sending them messages. This can be seen in a Lopez example in Figure 2, as well as our own implementation shown in Figure 3.

Figure 2



```
def _init(self, path_to_file):
    self._storage_manager = DataStorageManager()
    self._stop_word_manager = StopWordManager()
    self._word_freq_manager = WordFrequencyManager()
    self._storage_manager.dispatch(['init', path_to_file])
    self._stop_word_manager.dispatch(['init'])
```

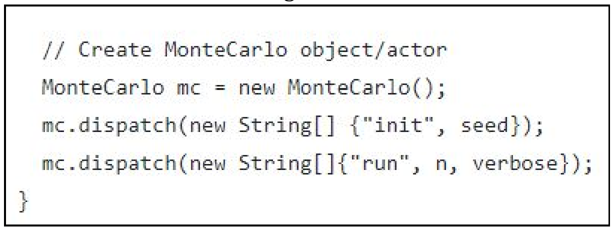
Figure 3



```
private String init(String seed) {
  generator = new NumGenerator();
  generator.dispatch(new String[] {"init", seed});
  attributes = new Attribute[14];
  int v = 0;
  for (int i = 0; i < attributes.length; i++) {
    attributes[i] = new Attribute();
    attributes[i].dispatch(new String[] {"init",
      init_values[v++], init_values[v++], init_values[v++]});
  }
  return null;
}
```

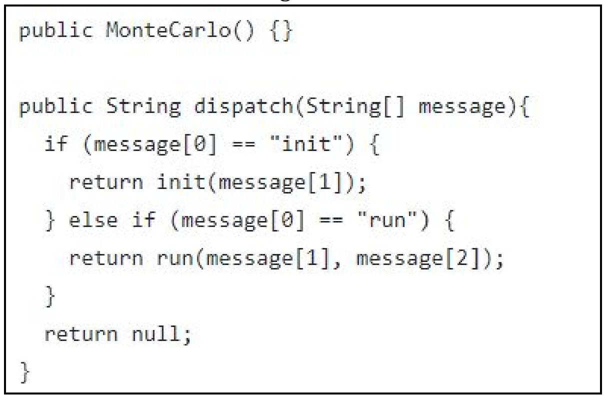
A challenge that arises when using the LetterBox pattern is that all parameters passed in a message are strings. Though this aligns well with the standard of command line arguments being in string form, it can be more difficult in the inner methods of a given object. For example, the integer number of samples to create, *n*, is prescribed as a command line argument, which the program's main method then passes to *MonteCarlo* via a "run" message. This is straight-forwad and can be seen in Figure 4. However, once the *MonteCarlo* object interprets the "run" message and passes *n* in a call to its private *run* method as seen in Figure 5, that parameter then needs to be converted to an integer. If the initial command line argument was not formatted as a proper integer, or cannot be converted to an integer at all, runtime errors will occur.

Figure 4



```
  // Create MonteCarlo object/actor
  MonteCarlo mc = new MonteCarlo();
  mc.dispatch(new String[] {"init", seed});
  mc.dispatch(new String[]{"run", n, verbose});
}
```

Figure 5



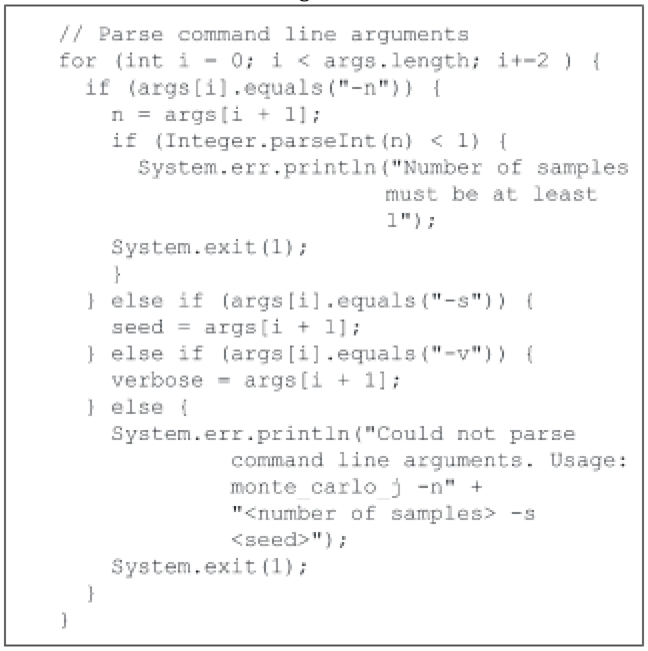
```
public MonteCarlo() {}

public String dispatch(String[] message){
  if (message[0] == "init") {
    return init(message[1]);
  } else if (message[0] == "run") {
    return run(message[1], message[2]);
  }
  return null;
}
```

Along with the LetterBox pattern we implemented and because of the chance for errors to occur from faulty command line arguments or message passing, we also implemented Tantrum style error checking. This proved easy to do in Java because of its object oriented nature. It may seem that this kind of error checking is necessary but because we are using the scripting functionality added in Java 11 along with a shebang, the type checking normally present in Java compilation is not there.

As seen in Figure 6 below, when the user adds in command line arguments that are not recognized or are unreasonable (such as giving a number but not specifying whether that number is the number of samples with -n or the random seed with -s), the program exits, and prints out a usage message for the user. This is achieved by checking the command line arguments for the -n and -s flags first, then reading in their value. If a flag is not specified or the number of samples is less than 1, all execution is halted and a message is printed to standard error.

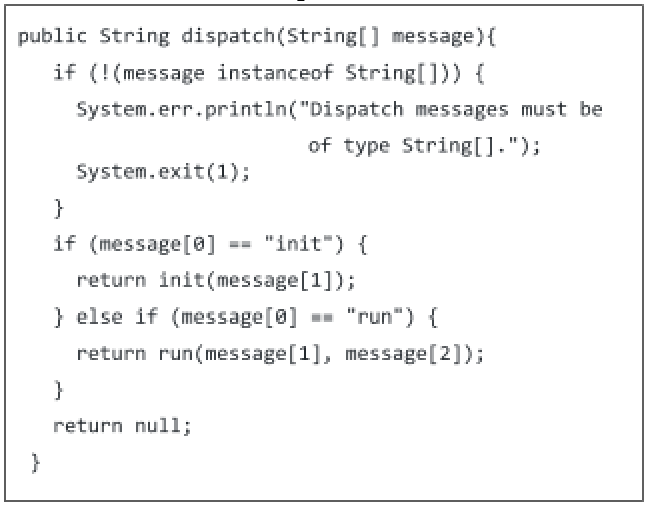
Figure 6



```
// Parse command line arguments
for (int i = 0; i < args.length; i+=2 ) {
  if (args[i].equals("-n")) {
    n = args[i + 1];
    if (Integer.parseInt(n) < 1) {
      System.err.println("Number of samples
                         must be at least
                         1");
    System.exit(1);
    }
  } else if (args[i].equals("-s")) {
    seed = args[i + 1];
  } else if (args[i].equals("-v")) {
    verbose = args[i + 1];
  } else {
    System.err.println("Could not parse
             command line arguments. Usage:
             monte_carlo_j -n" +
             "<number of samples> -s
             <seed>");
    System.exit(1);
  }
}
```

Figure 7 below shows how we applied the tantrum style to error checking letter box messages that are dispatched between the objects in MonteCarlo.

Figure 7



```
public String dispatch(String[] message){
   if (!(message instanceof String[])) {
     System.err.println("Dispatch messages must be
                         of type String[].");
     System.exit(1);
   }
   if (message[0] == "init") {
     return init(message[1]);
   } else if (message[0] == "run") {
     return run(message[1], message[2]);
   }
   return null;
 }
```

The final abstraction that we used when replacing the filter was the Delegation Pattern abstraction. This abstraction appears to be highly linked with the letterbox abstraction. The NumGenerator and Attribute classes, which were crucial to creating the LetterBox abstraction, were easy to represent as helper classes. As such the MonteCarlo object acted as the Delegator and the NumGenerator and Attribute objects it contained acted as the Delegatees.

EXPECTED GRADE

For 2a, we chose to implement the *MonteCarlo* algorithm (one star, zero bonus points) in Java (one star, zero bonus points) while utilizing the following abstractions: letterbox (two stars, one bonus point), tantrum (three stars, two bonus points), and delegation (one star, zero bonus points). Assuming that we receive the full 10 points for assignment 2a, with the maximum 2 bonus marks from abstractions, we expect to receive 12 marks.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software. Boston: Addison-Wesley.

[2] Lopes, C. V. (2014). Exercises in Programming Style. Boca Raton, FL: Taylor & Francis Group, LLC.