Various Design Implementations of the Monte-Carlo Method

Adam Davis
North Carolina State University
agdavis5@ncsu.edu

Logan McMillan
North Carolina State University
ltmcmill@ncsu.edu

Jack Schaffner
North Carolina State University
jwschaff@ncsu.edu

ABSTRACT

This document explores the potential design patters which can be used to re-implement the Monte-Carlo portion of the Duo pipeline. It also explores the findings resulting from implementation of `monte_carlo` with the REST, Subject/Observer, and Pipe and Filter design patterns.

KEYWORDS

design patterns, abstractions, monte-carlo

ACM Reference Format:
Adam Davis, Logan McMillan, and Jack Schaffner. 2019. Various Design Implementations of the Monte-Carlo Method. *CSC417*. 5 pages.

1 INTRODUCTION

Whenever a developer is added to a project, there is an associated cost with on-boarding that developer. Not only does the new developer need to dedicate time to learning about the project via documentation and existing software artifacts, but they must also seek help from developers who would otherwise be spending their time implementing the product.

For this project, we have been provided a data mining pipeline that attempts to model a project development cycle. By doing this, we can determine at what point in a project's life cycle it makes sense to stop adding new developers to the project.

The code that was provided to us is implemented using the pipe and filter pattern, and is thus made up of a series of discrete programs. Each program in the pipeline accepts a `.csv` file as input (with exception to the first program in the pipeline) and prints a `.csv` file as output.

Our goal in this project is to replace the first program in the pipeline, `monte_carlo`, with a program of our own creation. This program generates rows of random parameter values using a uniform distribution and minimum and maximum bounds for each parameter.

This paper discusses several possible design patterns that could be used for this program, which patterns we decided to use, and our experiences in writing using those patterns for this project.

2 DESIGN PATTERNS

This section will give a detailed overview of various design patterns that could be applied to the `monte_carlo` program. Each subsection will begin with an overview of the pattern and its common uses, and then follow with a potential way to apply the abstraction to `monte_carlo` program.

2.1 Restful

2.1.1 Background. The Restful design pattern allows for a separation between client and server, and is commonly used in web development. It promotes stateless interactions and uses a uniform interface. The most common restful "verbs" are as follows:

- `GET`
- `POST`
- `PUT`
- `DELETE`

`GET` is used for retrieving information. `POST` is used for creating a record or performing an action. `PUT` is used to edit existing records. `DELETE` is used to delete records.

While there are additional verbs, and an application does not necessarily need to use all of the above verbs, these are the most common.

2.1.2 Implementation Potential. For the Restful design pattern, we see implementation potential in separating the part of the system doing the output from the part of the system doing the table's row generation. The table's row generation can first post its output to the server. Then the part of the program that performs output can ask the server for what it is supposed to write to standard out. One problem here is that the output part of the program needs to know when it should ask the server for its output. That's where another design pattern would have to help.

2.2 Subject / Observer

2.2.1 Background. The subject observer pattern is a pattern which allows dependents of an object to be notified when the another object changes state. Generally, this pattern is used in cases when changing the state of one object requires that you change the state of many other objects. It can also be used when objects need to perform actions when another object changes state.

2.2.2 Implementation Potential. In the monte-carlo problem, it is possible to separate row generation from the output. One way to achieve this is by using the subject/observer pattern. We see potential here to register the output portion of the program with the generation portion of the program. Then the generation portion can notify the output portion every time it has generated a new row of output. This way, it is not necessary to worry about calling the output function in the main script. It will be done automatically once the output portion has been registered.

2.3 Composite

2.3.1 Background. The composite design pattern is a pattern where a whole object is composed of instances that inherit from the same parent class. There are two types of objects that inherit from the parent class. One is an object that is composed of other objects. The other is a leaf object. A leaf object is like a primitive in a

programming language. It cannot be further subdivided. This means that each of the composite objects can be viewed as a sub tree with the outermost object being the root of the tree.

2.3.2 Implementation Potential. In the monte-carlo problem, the composite pattern could be most easily seen as a way to format the output. For instance, the total output could be viewed as a composite object that is made up of several composite row objects. Then the row objects can be seen as being made of leaf objects where each leaf object is an individually generated value.

2.4 Map Reduce

2.4.1 Background. The map reduce pattern is a way of breaking up a problem into smaller chunks, mapping an identical operation to each piece of data, and then combining the output from the operations. This pattern is used primarily for processing large data sets, and is thus common within the fields of data mining and machine learning.

The primary benefit of the map reduce pattern is that it allows a program to run in parallel, reducing the total amount of time required for the program to run. However, the threads required by the program must be supported by the hardware in order to gain these performance benefits. Additionally, using too large of a number of threads in proportion to the size of the data set has potential to slow down the program. Thus, a balance must be struck between the size of the data and the number of threads.

2.4.2 Implementation Potential. In the context of the `monte_carlo` program, the map reduce pattern could be used to parallelize the process of generating the rows of randomized data. Separate worker threads would handle the generation of different rows of data, and then they would be combined at the end into a single data set to be output.

2.5 Lambda Calculus

2.5.1 Background. The lamdba pattern is a way of handling functions as first-class objects that can be passed around and referenced in the same ways as other data types. This provides a dynamic framework for performing functions in various contexts.

This practice is common in javascript, as shown below:

```
var timesTwo = function(num) {
    return num * 2;
};
```

Here, a function is stored as a variable. It can be stored in lists or passed into other functions just like any other data type.

2.5.2 Implementation Potential. The lambda pattern would apply to the random number generation of the `monte_carlo` program by storing functions to generate numbers instead of having a single function that generates a random numbers based on each field's minimum and maximum values. Through this method, the program would become more extensible by allowing different fields using different strategies. For example, instead of using a uniform distribution for every field, some fields could take advantage of other distributions, like the normal distribution.

2.6 Macros

2.6.1 Background. Macros are a form of programming in which the programmer is able to create a meta syntax for writing code. This allows the programmer to create domain-specific programming constructs, but at the cost of hiding the real code behind a layer of pre-processing.

An example of this can be seen in the Julia snippet below:

```
macro addToNum(num)
    return :(function(x)
        return x + $num
    end)
end

f = @addToNum 5

f(0)
```

In this example, the macro `addToNum` returns a function that takes in parameter `x` and adds it to another number `num`. However, `num` is a value determined macro, meaning that `addToNum` is able to create any function in which a static number is added to the variable `x`. From the snippet above, the number `5` would be returned.

2.6.2 Implementation Potential. Macros would fit nicely with the proposed implementation of lambda calculus. However, instead of writing out the entire lambda function for each property, a macro could be used that takes in the minimum and maximum values, as well as details about the type of distribution.

2.7 State Machine

2.7.1 Background. State machines include a set states with specific paths for transitioning from one state to another. Typically, there is one entry state, one or more final states, and some number of intermediary states. This abstraction is commonly found in safety-critical environments where it is more beneficial to be able to model every possible state and transition of a piece of software to prevent unexpected outcomes.

2.7.2 Implementation Potential. To implement the `monte_carlo` program with a state machine pattern, the program would begin in the state in which no values have been generated. Then the first value would be generated and the program would transition to the state in which one value for the row has been generated. The program would then generate the second value and proceed to the next state. This process would repeat for all values in the row. Once all values for a row have been generated, the program would transition to the first state if there are more rows to generate. Otherwise, the program would transition to the end state.

2.8 Pipe and Filter

2.8.1 Background. The pipe and filter pattern involves solving a problem by using a series of discrete programs. In this context, each program acts as a utility that performs a step in a process. The programs communicate with one another using a common interface: files. The output of the first program is piped into the second program, whose output is piped into the third program and so forth.

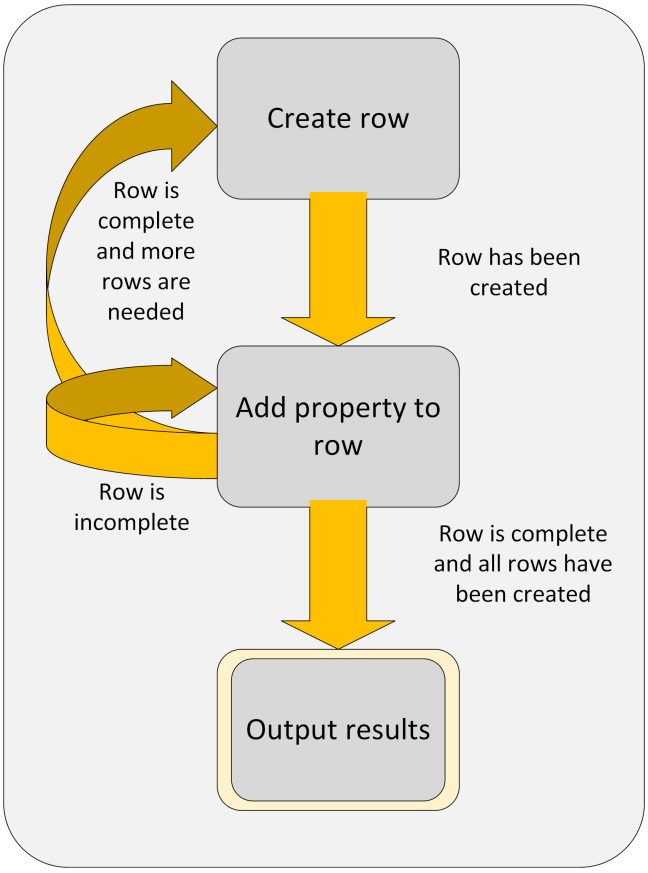

Figure 1: A potential state machine design for our program.

This abstraction is commonly used on the Linux command line. For example, if a user wanted to view all of the `.java` files in the current directory using `less`, they could use the following command:

```
ls | grep .java | less
```

This approach takes advantage of three different programs that each have their own individual purpose while leveraging a common interface, making it incredible modular. For example, if the user wanted to instead save the output to a text file it would only require swapping out `less` with a redirect:

```
ls | grep .java > output.txt
```

2.8.2 Implementation Potential. The provided implementation of the data mining pipeline already uses the pipe and filter pattern. This section, however, explores what it would look like to break the functionality of the `monte_carlo` file into its own pipe and filter design.

Generating all of the output can be viewed iteratively as generating a single row several times. Generating a single row can be further broken down into generating several columns that all have their own attributes. Therefore, the generation of a single row can be viewed as a pipeline that can be reused as many times as necessary to get the appropriate amount of output. This pipeline would be composed of a filter for each of its columns. Each filter would know the name of its attribute as well as the information necessary to generate a single instance of itself. The filters would pass either the random number generator that they are using in order for seeding functionality to work appropriately and/or its output appended to the output of the previous filter. By the end of the pipeline, an entire row has been generated and is ready for output.

2.9 Delegate

2.9.1 Background. Delegation is an object-oriented design pattern in which an object passes its context to another object that performs methods on its behalf. The pattern is used to help promote code reuse in the context of object composition.

2.9.2 Implementation Potential. To implement delegation in the `monte_carlo` program, there would be a class that stores a minimum and maximum value and has a function for generating a random number within that range. An instance of this class would exist for each column in the `.csv` output file. These objects would be stored in a list, and then called upon when each object's generate number function is needed.

2.10 Rule-Based Programming

2.10.1 Background. In rule-based programming, there are four different steps:

(1) Match
(2) Select
(3) Act
(4) Repeat

When an action needs to be performed, the program begins by searching for all applicable rules. Once this has been completed, the program uses a set of criteria to determine which rule to select. For the act step, the program carries out the action specified by the selected rule. Following the completion of the action, the entire process is repeated.

Rule-based programming is similar to a state machine, but is less precise and more difficult to model. While a small set of rules can be easy to manage, a program can become incredibly difficult to maintain once it has a large enough number of rules.

2.10.2 Implementation Potential. A possible implementation for this abstraction would involve a rule class that has a field to store a rule's precedence level, a function that checks whether the rule is applicable, and function that performs the rule's action. Using instances of this class, the following rules would need to be defined:

- If there are no more rows of data to produce, terminate the program.
- If none of the values for the next row have been generated, generate the pomposity.
- If only one of the values for the current row has been generated, generate the learning curve.
- Repeat rules that are similar to the two above for the remaining properties that need to be generated.
- If the row has been completely generated, begin creating the next row.

3 DESIGN PATTERN RANKINGS

What follows is a ranked list of the design patterns listed in the previous sections based on how well we think the abstraction fits the context of the program. Our implementation uses the top three ranked abstractions.

(1) Restful
(2) Subject / observer
(3) Pipe and filter
(4) Composite
(5) Lambda calculus
(6) Macros
(7) Map reduce
(8) State machine
(9) Delegate
(10) Rule-base programming

4 EPILOGUE

Our final implementation ended up including the Restful, subject/observer, and pipe and filter abstractions. Our subject contained the pipeline that generated an entire row, and it also added appropriate formatting to the beginning and end of a table row so that our output would fit into the pipeline appropriately. Each filter contained its attribute name and the minimum and maximum values for that attribute. An individual filter took in the row string that was generated up until that point and added its generated attribute to the end of its output. Once the entire pipeline is done, the subject posts the entire row to the REST API. Once the row is posted, the subject notifies the observer. The observer then retrieves the output from the server with a get request and writes it out to standard out. This entire process can be repeated n times in order to achieve the necessary amount of output.

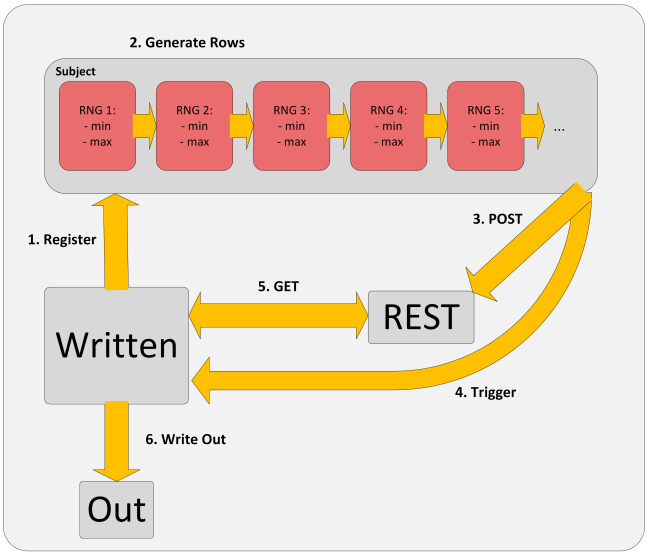

Figure 2: A diagram of the interaction between each part of our application.

4.1 Our Experience

As a broad reflection for all the abstractions used, it's necessary to consider the part of the pipeline we implemented. The monte carlo part of the pipeline is relatively simple. ItâĂŹs easy to imagine how the program could be implemented with just a dictionary and a nested pair of loops. Even if it were implemented in this manner, it would still be relatively easy to understand. Because of the simplistic nature of this part of the pipeline, all of these abstractions tend to seem like overkill.

The subject and observer abstraction was fairly uneventful, and it easily worked with how we used it. We registered a writer with the container for the pipe and filter abstraction. Once that pipe and filter abstraction was done with the row generation, the writer was notified so it could get the text to output from the REST API. One convenient thing about the subject and observer abstraction was that we didnâĂŹt have to repeatedly call the writerâĂŹs method from our main script for it to write out the next rows. Instead, we registered it once at the beginning and just had to repeatedly tell the subject to create the next row.

Our experience with the pipe and filter abstraction shifted as we completed the project. At first, we planned that the pipe and filter abstraction would work throughout a table row in order to pass the random number generator from one filter to the next. This would have made sure that the seeding functionality worked appropriately even though the filters were so not have direct access to each othersâĂŹ internal states. This would have worked if we had used a language like Java whose random number generator is an object and takes a seed in the constructor. In Typescript and JavaScript, however, the Math.random() function cannot be seeded normally. Instead, we must use NodeJS to seed the entire programâĂŹs execution environment. Since this seeds the environment, we have no need to pass the seed in the pipe and filter. So instead of passing the random number generator through the pipes and filters, we passed the accumulating row that was being generated. This means that we only made one post call to the REST API at the end of the pipes and filters rather than making a call at each filter as we had initially planned.

The REST abstraction utilized the post and get aspects of the REST API. When a subject had created it's row, the row was posted to the API. Inside the API, the row was stored until a get was used to retrieve it. When the Observer is notified of a complete row, it uses a get request to retrieve the row from the API. In order to maintain the correct execution order for our code, extensive use of the async/await system implemented by typescript was needed. Most of this was done in the Subject and Observer class where the post and get requests are made. In the Subject class, the post request needed to complete before the Observer was notified. In the Observer class, the row needed to be retrieved and printed before the notification could be considered complete. Since the makeRow() function included await, the function was required to be declared async. This posed a problem for us because top level await is not supported by Node. To work around this, we used a recursive function call that utilized the callback of the promise returned from the makeRow() function to call makeRow() again the correct amount of times, while also maintaining correct execution order.

4.2 Recommendations

For the subject and observer abstraction, one recommendation would be to use it when an event may or may not occur. Our pipe and filter structure will always return. Since it always returns, it would be easy to just call the pipe and filter abstractionâĂŹs function then pass its output to the observer from the main script. On the other hand, if the observer only needed to be notified if there were a mouse click, it would be undefined as to whether that event would occur. This would make the subject and observer pattern necessary as opposed to seemingly extraneous.

Our pipe and filter abstraction ended up feeling excessive. One recommendation in order to avoid an excessive pipe and filter structure would be to make sure that each filter is reasonably complex. If each portion of the pipeline is reasonably complex on its own, it makes sense to break that up into several separate programs when maintenance and modularity become more important. Similarly to how rule sets for production systems should become large prior to segmenting the rule base, the filter/program should be large and complex enough to justify breaking it into smaller pieces. If the filters are too small, it becomes more burdensome than useful to separate functionality into filters that must be explicitly defined.

REST can be extremely useful when wanting to offload processing or storage to another application, the downside of this pattern is that there is no control over the process by which the application completes the request or the time it takes to complete the request. This makes the REST pattern to use when attempting to make grantees on performance or process of execution.

5 ANTICIPATED GRADE

For our implementation, our team has done two three star parts and a one star part. We extensively applied the three star language Typescript, which is a language that is transpiled to JavaScript. We significantly applied its object-oriented functionality as well as its typing that both do not exist in JavaScript alone. We also applied the three star REST pattern in order to transfer our row output from one part of the program to the other. Finally, the part of the pipeline that we replaced is the monte carlo program, which is only a one star segment. With these parts, we expect our maximum grade for 2a to be 10*1.14*1.14*1 = 13.