Various Design Implementations of the Dom Program

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 dom portion of the Duo pipeline. It also explores the findings resulting from implementation of `dom` with the Composite, Lambda, and Map/Reduce design patterns.

KEYWORDS

design patterns, abstractions, dom

ACM Reference Format:
Adam Davis, Logan McMillan, and Jack Schaffner. 2019. Various Design Implementations of the Dom Program. *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 third program in the pipeline, `dom`, 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 `dom` 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 `dom` program.

2.1 Inheritance

2.1.1 Background. Inheritance is an object-oriented design pattern in which one class uses another class as a template, or parent, and only needs to specify its own properties in relationship to another class.

2.1.2 When To Use. Inheritance is helpful because it allows for code re-use and sub-classifications. It is practical in situations where data fits neatly into hierarchical structures. However, it has the drawback of making the child class dependent on the parent class. If at some point it becomes necessary for the parent class to have a property that the child class should not have, then some fairly serious refactoring will need to take place.

2.1.3 Implementation Potential. To apply inheritance to the dom program, the table would need to be implemented in an object-oriented manner. There would be a parent class for for properties, and each type of property would inherit from this class.

2.2 Polymorphism

2.2.1 Background. Polymorphism is a common Object oriented design pattern that involves abstracting away the implementation of a function or object behind a common appearance. Not to be confused with inheritance, which allows object to use the functions and variables of a parent, polymorphism allows entire implementations to be rewritten leaving the interface for those implementations the same.

2.2.2 When To Use. Polymorphism is most helpful when in cases where an objects behavior may change from instance to instance, but the way that the object exposes itself and is used is similar to other objects of it's type. For example, this would be useful in the case of a destroy function, where each child of a destroyable parent class may have different behavior when it should be destroyed. Polymorphism is not helpful when it is useful to know implementation details or when side effects of a function need to be fully understood. It is also not particularly helpful if there will only ever be one implementation.

2.2.3 Implementation Potential. Polymorphism may be useful in the implementation of this stage in the pipeline when comparing numbers. A comparitor class could be created which could have a common compare function which would take in two numbers and return one. The implementation for how the returned number was chosen could be different for each child of that comparitor class.

2.3 Blackboard

2.3.1 Background. The blackboard design pattern is a pattern that allows many clients to share a common memory space and arbitrates the use of that space between clients.

2.3.2 When To Use. The blackboard pattern is very useful when you have many clients wanting to access and update shared information. One common use case for blackboard architecture is in websites like forums or any website that stores and provides

information to clients. The blackboard architecture is not helpful in cases where flow of control is consistent. In cases where data needs to be processed by multiple functions, it is much more efficient to simply pass the information as a parameter than have a blackboard arbitrate the access to the data.

2.3.3 Implementation Potential. The blackboard architecture could be used as a way of storing and accessing table data as it is read and processed. The blackboard could be used as a common space for functions to independently produce the desired data value at each cell in a row without worrying about concurrency.

2.4 Restful

2.4.1 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.

2.4.2 When To Use. Restful is very useful when trying to offload processing or information storage to another system. The Restful architecture has exceptional facilities for distributing these behaviors and making access to information easy between all systems therefore it is commonly used for web APIs. Restful is not useful when local data and processing is sufficient to complete the task required. Restful generally has significant overhead, especially in web based implementations, which should be avoided if possible.

2.4.3 Implementation Potential. For the dom implementation, Restful could be used in the place of the map reduce functions, where the data could be processed externally through and API and returned to the client as a printable table. This would allow the processing of the data to potentially occur on a more powerful system externally and be returned faster than what our local machine would be able to do.

2.5 Subject / Observer

2.5.1 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.

2.5.2 When To Use. Subject observer can be useful when attempting to manage the results of multiple threads within an application, or more generally when you are not sure of when data will be produced, but you want to perform and action when it does.

2.5.3 Implementation Potential. The subject observer pattern could be used in our dom implementation to notify of the completion of rows being read. When each row is finished, the observer could add the processed row into a common table abstraction.

2.6 Composite

2.6.1 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.

2.6.2 When To Use. Composites are commonly useful when there are clear hierarchies of common types. A two common cases are menus and directories. Composites are not useful when there are no clear hierarchies, or the hierarchies to not share common functions.

2.6.3 Implementation Potential. In the dom program, the composite pattern is very useful for converting the input from the brooks program into a table format. The table is a composite of rows. Rows are composites of columns. Then columns are leaves with their numeric value and other related meta-data. This format is useful for indexing a table by row then indexing rows by columns. This type of indexing will allow the dom program to access individual rows and columns neatly and logically. It will also allow dom values to be easily appended as the last column in the table. Finally, the composite pattern will be useful for outputting the table data with a single output call to the table that will cascade down to the leaves through the composite elements.

2.7 Map Reduce

2.7.1 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.

2.7.2 When To Use. Map reduce is extremely useful when there are common functions that should be applied to data in parallel. Big data applications like processing logs for useful information is a common case. Map reduce can become overly complex when each data elements processing is tied to other data in the set, as processing should be done in parallel to take full advantage of map reduce.

2.7.3 Implementation Potential. In the context of the dom program, the map reduce pattern could be used to parallelize the process of calculating dom scores. For instance, its first map function could map N other rows to each row's id/index key. The reduction for each key would then calculate the dom score for the row at its key index with respect to all of the rows that were mapped to its index. The reduction would then output the cumulative dom score for its row. Once the reduction is done, its total output could be coerced into the table as the final column. One thing to note here is that it would be challenging to figure out which row should map to which keys using the map reduce pattern alone as there would be some required state to make sure at least/only N rows are mapped to every row's key. One possible solution would be to pre-generate which rows should be mapped to which other rows' keys. This is definitely a major limitation of the map/reduce pattern as it would not be as easy to parallelize this portion of the program.

2.8 Lambda Calculus

2.8.1 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.

2.8.2 When To Use. Lambdas are useful in many cases. Most commonly, lambdas are used as a system for allowing dynamic processing based on information at runtime. Callbacks are extremely easy to implement with lambdas. Lambdas are not always useful when clarity in the behavior of a function is desired as they do not have good mechanisms for restricting what kinds of things a received lambda will do.

2.8.3 Implementation Potential. In the case of dom, lambda calculus, when used in conjunction with map/reduce, would allow us to simplify the interface for our map and reduce layers. Rather than having to create different map and reduce interfaces, we can use a single interface that accepts lambda functions to be able to handle different functionality.

2.9 State Machine

2.9.1 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.

2.9.2 When To Use. State machines are 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.

They can be used in many situations that would use production rules, but state machines have the advantage of spending much less time in the match stage.

State machines should be avoided in situations where the application should be stateless. For example, state machines tend to be a poor fit for restful web applications, which

2.9.3 Implementation Potential. To implement the dom program with the state machine pattern, the machine could start at an initial state that receives a row for input. Once it receives that row, it can transition to the next state. The next state receives a different row where that row is the next row to compare against the row received by the start state. Every time the second state receives another row, it adds to the start row's dom score and increments a counter. This process continues until the counter is equal to N. Once the counter is equal to N, it transitions to the next state. The next state outputs the row received at the initial state with its dom score appended to the end. If there are more rows to process, the machine uses the next row as the next start row. If all rows are processed, then the program is ready to terminate.

2.10 Pipe and Filter

2.10.1 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.

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
```

2.10.2 When To Use. Pipe and filter is best applied to stateless situations where the functionality can be broken up into multiple discrete parts that complete in a specific order. It is ideal if every part of the pipeline performs only a single, specific functionality so that it may be modular for use in other contexts.

This pattern should be avoided in GUI applications where the user is able to select one of many possible actions, which would lead to a complicated pipeline. It should also be avoided in situations where two different filters would need to communicate with one another.

2.10.3 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 dom file into its own pipe and filter design.

When checking if one row dominates another, it is possible to break the process into a pipe and filter pattern. Each column of the row could be processed by a separate filter. Then the values of s1 and s2 would be piped into each filter then incremented with the

next column's value. At the end of the pipeline, the two s values would be compared and the final filter could output whether or not the row in question dominates the other. This same pipe and filter structure could be reused to perform the same process every time a different pair needs to be processed.

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) Composite
(2) Lambda calculus
(3) Map reduce
(4) Restful
(5) Polymorphism
(6) Blackboard
(7) State Machine
(8) Subject / observer
(9) Inheritance
(10) Pipe and filter

4 EPILOGUE

Our final implementation used the composite, map/reduce, and lambda patterns in the language Julia. The composite pattern was used for handling the application's input and output data. We used the same table structure as implemented in Lua in the source code, but we converted the structure to Julia. Once the input data is converted to a tabular format, we handle data manipulation using the map/reduce pattern. We use several map and reduce layers in order to convert rows to have the appropriate dom scores appended. Having several different map and reduce layers would not have been nearly as easy if we did not apply the lambda pattern. The lambda pattern allowed us to pass traditional map/reduce functions to the same map/reduce layer infrastructure in order to achieve different functionality.

4.1 Our Experience

The composite pattern was nearly a logical necessity for the dom problem, but implementing it was also a lot of work compared to solving the rest of the problem. It would have been very inconvenient to manipulate the input and output data in the format provided without having some concept of a table, but we are not sure it was worth the effort for a one time implementation of a table that will likely never be used again. On top of the work required to implement a composite table, instead of the object-oriented view of the composite pattern, we had to use nested dictionaries since Julia isn't object-oriented. Fortunately, we had the Lua source code to reference that also used the nested dictionary view of the composite pattern. Overall, we had a love-hate relationship with the composite pattern, since it made the rest of the problem a lot easier at the expense of a lot of infrastructural overhead.

The map/reduce pattern ended up fitting the dom problem quite well despite lacking parallelism in our implementation. The entire dom score calculation could be broken down into map and reduce functions fairly easily. Simulating the environment for map and reduce functions could be performed with a dictionary of lists. The input to and the output from the map/reduce layers were both dictionaries with their keys mapped to lists of values pushed with those keys. That means the layers could be arbitrarily strung together in order to perform a complex operation. Inside an individual layer's map/reduce function, it is just necessary to push key/value pairs to a similar dictionary structure and then return that structure once it's done executing. The layer adds all the function call's new key/value pairs to the overall map/reduce environment which is then passed to the next layer.

The lambda abstraction was really convenient for this problem, especially when using the map/reduce pattern. We used lambda bodies to ensure that we only ever had to create the environment for the map/reduce layers once. That made our map/reduce operations very modular in that we could just plug in whatever individual map or reduce function we wanted into its corresponding layer. This reduced unnecessary code as we did not have to make separate environmental functions to call separate, low-level map and reduce functions. The same environment just calls whatever lambda body map/reduce that was passed in as a parameter.

4.2 Recommendations

While the composite pattern is highly fitting for the dom problem, we would definitely suggest that the pattern be applied more readily in an object-oriented environment as the dictionary syntax is rather overbearing and confusing from time to time. Nested dictionaries make the composite pattern fairly verbose to use and declare, but that could be taken with a grain of salt as all of us are originally object-oriented programmers. Another thing to keep in mind when applying a generic composite/table pattern is to ensure the problem is complex enough to require a generic solution as the table we implemented ended up being a lot more work than the rest of the program.

The most obvious advice for the application of the map/reduce pattern is to only use it when there is a ton of data that can be processed in parallel with a lot of resources. Parallelism obviously comes with a lot of implementation complication, however. That is especially the case when operating on a cluster of machines, but there are pre-existing technologies like Hadoop or Apache Spark that should be applied in that situation. We are not sure there is a situation where a modern programmer should logically be reimplementing their own map/reduce infrastructure on a scale where it makes sense to use the pattern. Running map/reduce on a single thread like we are does not make a lot of sense practically speaking, since it becomes similar to the pipe/filter abstraction but has the added overhead of handling the key/value environment.

Based on our experiences, we think lambda bodies are extremely applicable in many different problems. There is really no overhead to speak of in relation to their use. In our experience, they actually reduced overhead, since they made a modular approach to map/reduce a viable option. Other than use more lambda bodies, one possible recommendation would be to keep lambda bodies as simple as possible. This is especially the case if they are being created anonymously inline when calling a function. The declaration of the lambda body can sometimes make code harder to read and understand. This is somewhat of a two-edged sword, however,

since declaring the lambda as a function elsewhere and passing as a function pointer makes it necessary to search elsewhere for the function definition.

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 Julia. None of our source code is written in anything else. We also applied the three star map/reduce pattern in order to make the actual dom calculations. We would consider this the core logic of the dom application despite the composite/table pattern taking more time to implement. Finally, the part of the pipeline that we replaced is the dom program, which is only a one star segment. With these parts, we expect our maximum grade for 2b to be 10*1.14*1.14*1 = 13.