Project 2 - Part B : Design Patterns Report

Gage Miller
dgmille2@ncsu.edu
NCSU

Daniel C. Mills
dcmills2@ncsu.edu
NCSU

Daniel E. Mills
demills@ncsu.edu
NCSU

Ian Smart
irsmart@ncsu.edu
NCSU

ABSTRACT

In this paper, we rank ten design patterns according to their utility in the refactoring of individual filters of a six-filter data pipeline into a different programming language. Additionally, we chose a single filter to refactor, utilizing our top three design patterns in the refactoring. At the end of this paper is a reflection on how our choices of design pattern and programming language affected the refactoring process.

KEYWORDS

pipeline,monte carlo,design pattern

ACM Reference Format:
Gage Miller, Daniel C. Mills, Daniel E. Mills, and Ian Smart. 2019. Project 2 - Part B : Design Patterns Report. In *Proceedings of CSC417 (Project 2B).* Raleigh, NC, USA, 5 pages.

1 INTRODUCTION

Our task was to select one filter of the provided data pipeline and refactor it into a different programming language, using three design patterns to guide our implementation. We started by collaboratively working to understand each filter. During this process, we began brainstorming potential ways that filters might be refactored. As a result of our deliberations, we had a handful of design patterns that could be applied to one or several of the filters. Then, we selected a filter and three design patterns based on the practicality of refactoring it using the chosen patterns. These choices then informed our choice of programming language, Haskell. The rest of this paper details these choices, why we made them, and how they affected the refactoring process, for better or worse.

1.1 Pipeline Overview

Brooks Law describes the tendency of late software projects to take even longer to complete when more programmers are added. The "law" was described by its namesake, Fred Brooks, as an "outrageous oversimplification" of how software projects function in the real world. To examine whether Brooks Law does in fact hold true, the provided pipeline generates simulated data describing software project environments and feeds this data into a compartmental model of Brooks Law. The output from the compartmental model is then filtered several times to produce meaningful results. The pipeline as a whole is an example of a Monte Carlo simulation, where a problem is solved by generating randomized input data and then analyzing the results.

The existing pipeline consists of the following six filters, ordered by their sequence in the pipeline:

- *monte_carlo*: Generates randomized data describing collaborative software projects
- *brooks2*: Runs data through a Brooks Law compartmental model. Calculated dependent variables are attached to the records along with whether they should minimized or optimized.
- *dom*: Attaches to every record a *dom* score quantifying the degree to which a record dominates.
- *bestrest*: Partitions the records into two groups, best and rest, according to how well they dominate others.
- *super*: Groups records into low-variance partitions for each attribute. Labels the partition a record belongs to for each attribute.
- *rank*: Ranks which partition of records for an attribute value best predicts membership of a record to the "best" class.

Each filter is an independent program connected by standard Unix-like, anonymous pipes. In practice, this looks like:

```
monte_carlo | brooks2 | dom | bestrest| super | rank
```

1.2 Filter Replacement

Our group decided to replace the *brooks2* filter. The *brooks2* filter defines the compartmental model used to evaluate Brooks Law. The current filter takes as input lines of serialized Python dictionaries defining parameters for one instance of the model. Each parameter is classified as a Stock, Aux(iliary), Flow, or Percent attribute and sorted in this order. The type, and therefore order indicates the flow of information

2 ABSTRACTION RANKINGS

The following subsections detail our considerations of the applicability of design patterns to the refactoring of different filters. Each subsection also includes a short description of our understanding of the pattern. The patterns are ranked from most suitable to least suitable, with the top three being those chosen for our implementation.

Project 2B, April 2019, Raleigh, NC

2.1 Adapter Pattern

2.1.1 Intent . The Adapter design pattern, in the context of object-oriented programming, is used to create an alternative, but functionally equivalent, interface for an existing class, referred to as the *adaptee*, without modifying the class itself. To implement the pattern, an *adapter* class is created to be used by a *target* class. This *adapter* class acts as a translator through which the *target* can make calls to the *target* class. Although *officially* an object-oriented design pattern, the Adapter pattern is also applicable in procedural programming contexts. For example, an adapter could be written that translates the whitespace-delimited output of the Unix command, 'ls -l', into JSON format for consumption by a Javascript program.

2.1.2 Rules of Thumb . The Adapter design pattern is similar to several other design patterns related to interfaces between classes, including the Facade, Proxy, Bridge, and Decorator patterns. The Facade pattern, for example, also modifies an interface, To determine if the Adapter pattern is appropriate for a problem, ensure that the following criteria are met:

- The problem requires the modification of a target class's interface, for use by a client.
- The modification of this interface does not change the underlying functionality available to the client.
- The target class itself will not be modified.

2.1.3 Examples . In the existing pipeline, each filter uses formats its input and output as comma-separated values, except for *monte_carlo*. *monte_carlo*́s output is a Python dictionary, which makes the succeeding filter, *brooks2*, language-dependent. To enable *brooks2* to be refactored in a different programming language, the Adapter pattern could be utilized *monte_carlo*. The programmer simply needs to implement an adapter class that translates the Python dictionaries into comma-separated value records. This would save the programmer the trouble of refactoring both *monte_carlo* and *brooks2*. Additionally, if *monte_carlo's* output is used by some other classes, where Python dictionary format is expected, the adapter pattern allows both *monte_carlo* and these other classes to remain unmodified.

2.2 Recursion

2.2.1 Intent . Recursion is a pattern which involves breaking a problem down into its base elements and then continuously solving the problem. In functional programming this pattern is commonly implemented by creating a function with a base case and then having the function call itself until it hits the base case. It is commonly used for handling data of unknown size.

2.2.2 Rules of Thumb . There are several trade offs to keep in mind when using recursion. The first is that it uses a large amount of stack memory. Recursive designs rely on using the system stack in order to keep track of execution. The second is that recursion is usually slower than in-lining functions. This is caused by the overhead from using stack memory. However, in exchange recursion is extremely straight forward to write and understand. It also simplifies tree traversal algorithms.

2.2.3 Examples . Recursion is a pattern which could easily be applied to most of the filters in this project. For instance, recursion could be applied to how brooks2 computes multiple steps in the flow chart. This would be accomplished by designing a step function to handle all of the possible step, passing in the number of steps to complete, and then having the step function recursively call itself until the steps are complete.

2.3 Aspect

2.3.1 Intent . The Aspect design pattern is a paradigm which helps increase modularity without increasing code complexity. It works by adding additional features on top of existing code, without modifying the underlying code. This is mainly done by grouping logically distinct aspects into distinct parts and then only having those aspects intersect at specific *Pointcut* locations. This locations should be uniform and reoccurring if possible. The Aspect design pattern is commonly used for businesses which want to streamline their solutions and for combining multiple programs together.

2.3.2 Rules of Thumb .

- The control flow is often obscured when using the Aspect design pattern. This can sometimes make it difficult to debug.
- The Aspect design pattern can make sections of code very reliant on each other. This can make developing code independently very difficult, if not impossible.
- The placement of *pointcuts* is very important and often times incredibly volatile. They can break down very quickly and are very sensitive to any changes in the overall program.

2.3.3 Examples. The Aspect design patter has a large number of potential use cases. The first example is adding logging to the brooks2 filter. This would allow us to get extra feedback when debugging and running the filter. It could be implemented by separating each step into its own function, and then adding a function to log any and all function calls. Another example is turning the whole project into a single program. It would be possible to break the whole project down into a single program and turn each filter into a separate aspect. This would allow the same level of code separation while also making each section logically distinct.

2.4 MapReduce

MapReduce is a popular programming model that leverages cluster computing to process huge datasets (i.e. *Big Data*). It is useful when a parallelizable computation can be separated into sequential phases of *map*, *shuffle*, and *reduce*. In the *map* phase, a dataset is partitioned and distributed to a cluster of nodes, each of whom apply a *map* function to key-value pair records of their partition. This *map* function performs some transformation and associates its output records with new keys. In the *shuffle* phase, *map*-phase outputs are redistributed such that records with the same key are collected and sent to the same node, where a *reduce* function is applied to the collection. This *reduce* function outputs a value or list of values that constitutes the desired, meaningful output.

2.4.1 Rules of Thumb.

- MapReduce is a Big Data processing technique and is typically inappropriate for datasets smaller than 1GB.

- MapReduce is not a catch-all pattern for all parallelizable, distributed computing problems. It should only be used when splitting the problem into map, *shuffle*, and *reduce* phase is useful.
- When processing on a single node, MapReduce implementations should be multi-threaded. Single-threaded MapReduce implementations do not realize the performance benefits typically used to justify using MapReduce.

2.4.2 Examples. In the existing pipeline, the MapReduce pattern can applied to refactor the *rank* filter. The dataset would randomly partitioned by rows and passed to worker nodes. On each node, a *map* function could iterate across each row and produce a key-value of pair consisting of a key "attribute:symbol" like "nprod:3.83..4.97" and a value, "!klass". The *shuffle* function would then redistribute all records with the same symbol for an attribute to different worker nodes. The *reduce* function would then count the number of relative frequencies of "best" and "rest" records and output the same key with the metrics (see lines 72 to 77 in rank.lua) calculated. From each node, these key-value pairs are then collected and sorted, with the top 10 values printed.

2.5 Enumeration

2.5.1 Intent . An enumeration is an abstract data type enabling a programmer to define a finite, and often ordered, set of values as the range of possible values for a custom *enumerated type*, or *enum*. Similar to how declaring a variable as a "char" in C restricts its possible values to the range 0 to 255, declaring an enumerated type variable restricts its possible values to those defined in the enumeration. For example, an enumeration, *MatterPhase*, defined as $\langle SOLID, LIQUID, GAS \rangle$ would enable the use of *MatterPhase* types restricted to one of these three values. Also, the ordering of the phases by temperature is implied by the order of values in the enumeration. Without enumerations, programmers usually have to declare a constant integer for every value in every enumeration they wish to use. Continuing with the previous example, a programmer would instead have to define three integer constants $SOLID = 0$, $LIQUID = 1$, and $GAS = 2$. Using integer constants is potentially dangerous because an integer variable might be set to a value not defined by the constants. The use of an enum variable provides an assertion that the variable always has a valid value. Additionally, enum variables are self-documenting, in that they communicate to a programmer their allowed range of values. Consider the readability of a function signature "listProperties(int element, int phase)" compared to "listProperties(Element, Phase)". Enumerated types prove useful in implementations of the Strategy and State patterns, Finite State Machines, and in innumerable other contexts.

2.5.2 Rules of Thumb .

- Enumerated types function like categorical and ordinal attributes.
- Although they have integer values, they should not be used for storing integer constants.
- Remember to initialize enum variables. A null value for an undefined enum might break your program when switched on. If an initialization to an enum value does not make sense in some context, an *UNDEFINED* value could be added to the enumeration to initialize enum variables with.

2.5.3 Examples . The *Thing* class hierarchy, used in the *brooks2* filter and defined in *dsl2.py*, contains four subclasses that implement a function *rank()*. This function is the only function defined in each subclass and its only purpose is to return the subclass's order relative to the other subclasses. To remove needless complexity, these subclasses could be replaced by defining the enumeration $ThingType = \langle Stock, Aux, Flow, Percent \rangle$, then adding a *ThingType* instance variable to the Thing class.

2.6 Information Hiding

2.6.1 Intent . Information hiding is a design principle that recommends the decoupling of software components whose implementations are expected to change. This is achieved by defining a common interface between components, so when one changes, its clients can continue to interact with it with little, and ideally no, refactoring. Details, like a text encoding or underlying data structure, are hidden from clients directly, but exposed through an stable interface. The application of this strategy increases maintainability by keeping components loosely coupled. In object-oriented programming, information hiding is achieved by encapsulating details within an object. Additionally, the use of an abstract class or interface definition can be used to strictly ensure an implemented class conforms to the expected interface.

2.6.2 Rules of Thumb .

- A well-designed interface will likely save development time in the long run.
- For large software systems, adherence to the principle of information hiding should be a primary goal during the design process.
- Information hiding is practically synonymous with encapsulation in object-oriented programming.

2.6.3 Examples . In the existing pipeline, *monte_carlo* and *brooks2* are tightly coupled. Parameters generated by *monte_carlo* (e.g. "newPersonnel" and "experiencedPeople") have associated behavior hardcoded into *brooks2* and its dependency *dsl2.py*. The addition of a parameter to the Brooks compartmental model requires the refactoring of at least three separate classes. The issue here is that the model's schema is not well encapsulated. These details would need to be encapsulated within *monte_carlo* with a more informative string format output. This string format would serve as the interface between the two filters, communicating information like whether an attribute should be minimized or maximized. Then *brooks2.py* and *dsl2.py* would need to be generalized to handle a variable number of attributes of unknown type. Since the system has already been implemented, the aforementioned changes along with others would need to be made to thoroughly decouple these classes. The difficulty of such a task post-implementation highlights the importance of considering information hiding early in the development process.

2.7 Lambda Calculus

2.7.1 Intent . Lambda Calculus is a framework which aims to take specific calculations and expresses them as generalized abstractions.

It performs functional computation through abstraction, substitution and variable binding instead of through numerical means. This allows the calculations to be broken down into their key aspects and then redistributed in a wide variety of places. This design pattern is incredibly helpful and is widely used in fields such as mathematics, linguistics, artificial intelligence design.

2.7.2 Rules of Thumb .

- Due to the mathematically nature of the framework, it can be difficult to break a function down to its main aspects.
- An expression can be much simpler to understand after converting it to use lambda calculus.
- Improper understanding of the underlying mechanics can cause the expression to become incredibly volatile/unstable.

2.7.3 Examples. Lambda Calculus is applicable to almost every single filter in the pipeline. For example, it would be incredibly valuable for creating macros in brooks2. By using lambda calculus in macros, we could easily replace large sections of the step code with generalized macros. This would help reduce the complexity of our code and make it easier to understand and debug.

2.8 Compartmental Model

2.8.1 Intent . The Compartmental Model is a commonly used design pattern which aims to separate logic from programming. This helps simplify the program and make it easier to understand. It accomplishes this by by modelling a system in which data flows easily between modules.

2.8.2 Rules of Thumb .

- By separating the logic from the programming, it makes higher level understanding easier.
- Compartmentalizing the logic makes modelling/simulating various problems a lot simpler than using other design patterns.

2.8.3 Examples. The Compartmental design pattern could be used in the *monte_carlo* filter when creating the random number generator. We could attempt to make the numbers generated as random as possible by defining a number of different data flows, all of which calculate a random number using a different method, and then combine all of them. We could also pass a set seed to each of them during debugging so that we could properly analyze the result.

2.9 Generator Pattern

2.9.1 Intent . The Generator Pattern describes the usage of generator function abstraction for efficient iterator over large sets. A generator function is used like a conventional iterator to return a series of outputs within a loop. A generator is distinct from a conventional iterator in that outputs are produced and returned on-demand. A conventional iterator might load an entire linear structure into memory before iteration. A generator, however, will only produce a value when called. This is a form of lazy evaluation, and as a result, generators are incredibly memory efficient compared to traditional iterators. A generator function only takes an argument once, after which it progresses until reaching a *yield* statement, where it returns control to its containing loop. When the generator function is called again, execution begins below the *yield* statement, rather than at the top of the function definition. Generators are particularly useful for iteration across excessively large or infinite streams, where memory would otherwise be a limiting factor. It also enables the production and consumption of inputs in parallel.

2.9.2 Rules of Thumb .

- Generators are useful for processing large or infinite size datasets.
- Since a generator function will continue yielding values until some base condition is met, a counter or range of values can be passed as its argument(s). This enables generator functions to be used like a traditional bounded for-loop.
- Generators are memory efficient iterators.

2.9.3 Examples. This pattern could be used in the *monte_carlo* filter. A generator function could be defined taking a random number seed and an iteration counter. It could then be called to generate new random numbers for specification of parameters to be fed to *brooks2*. Additionally, every filter besides *monte_carlo* could have a generator function for efficiently reading rows of input from standard input.

2.10 Macros

Macros are used to mapping a snippet of code to series of characters. Then, when the compiler encounters that series of characters, it replaces the series of characters with the snippet of code. This helps make code easier to read and allows programmers to write less code when repeatedly performing the same operation. Macros are used all over the place, especially in languages like Lisp, C, Clojure, Elixr, and Prolog.

2.10.1 Rules of Thumb .

- Works well with Lambda Calculus and abstraction
- Makes high level understanding a lot easier
- Can sometimes make code hard to debug

2.10.2 Examples. A few specific examples of where macros might be used includes assigning a large number to a short name (such as PI to represent 3.1415...), finding the average of two numbers, or even creating a macro which includes multiple macros. This pattern would be very useful in the rank filter, as it requires multiple uses of long numbers and several identical function calls. Those could be cut and replaced with macros, which would make the filter infinitely easier to understand. Here's an example of defining and using a macro in Lisp:

```
(defmacro mulsq (x y) "multiplies the squares" `(* (* ,a ,a) (* ,b ,b)))
```

The macro can now be used to multiply the squares of two numbers, for example `(mulsq 4 5)` results in 320.

3 EPILOGUE

For this project we decided to try and implement the brooks2 filter in Haskell. The three abstractions that we had chosen to implement for the brooks2 filter were Recursion, Adapter, and Macros. The first two we managed to implement fairly easily, however we struggled a bit with properly applying the third. The brooks2 filter readily adapts itself to recursion. The filter is designed in such a

way that involves continuously searching and creating lists. This meant that we could easily create a number of functions which would recursively search an association list and recursively create the association list. The adapter pattern was similarly easy implement. We simply used it to translate the output of the previous filter into an object that was a lot easier for us to work with. Macros, on the other hand, were a lot harder to implement than we originally planned. We just couldn't find a solid place to use them or a valid reason to justify their existence. Also, macros aren't very well supported in the language we were using, Haskell. We decided that it would be a much better idea to switch instead to the Aspect pattern. The Aspect design pattern allowed us to easily add features, like logging, without having to change our code, and it made a lot more sense in the context of the filter.

4 EXPECTED GRADE

Completion of the project with expected output and meeting all requirements contributes 10 marks to our total grade. Our choice to refactor the *brooks2* filter using the *Haskell* programming language and, in part, the *Aspects* design pattern, contribute another 6 marks. In total, we expect to earn 16 marks for the project.

Table 1: Expected Marks

Bonus	Stars	Marks
Haskell	3	2
Aspect	3	2
brooks2	3	2
Project	NA	10
Total	NA	16