Team A

Austen Adler
Bryan Arrington
Casey Belcher
Christopher Haynes
agadler@ncsu.edu
bjarring@ncsu.edu
cjbelch2@ncsu.edu
cehayne2@ncsu.edu
North Carolina State University
Raleigh, 27695, United States

ACM Reference Format:
Austen Adler, Bryan Arrington, Casey Belcher, and Christopher Haynes. 2019. Team A. In *Proceedings of* . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SELECTIONS

Selection of Pipeline `super`
Selection of Language Elixir

1 POSSIBLE ABSTRACTIONS

1.1 Servant

1.1.1 Overview.

A servant pattern is where the programmer uses helper methods or helper classes to abstract code from where a behavior is needed. This is often used by programmers to simplify condition statements in the portion of code where they are used by moving the actual logic to a separate section of code.

In our code, this could be used to simplify the logic for calculating a range of values by moving the actual calculation out of the way of the overall logic.

This abstraction can result in more legible functions, because of the ability to use a concise function name in place of potentially many lines of code. The same benefit is available for classes that use helper classes to abstract some of their functionality or state.

The Servant pattern also helps programmers make self documenting code. This is because programmers can take long chunks of code, such as "if thing is not None and thing >= someNumber:" and condense it to code like "if valid(thing):" by moving the logic of the calculation into a separate function.

In addition to the already stated benefits, this abstraction can make debugging code easier. This is because the actual behavior is only written in a single place, and referenced everywhere else. As such, a programmer would only have to fix the code once to resolve bugs everywhere this code is used.

Unfortunately, this benefit comes with a cost. It moves some of the behavior of a piece of code into a separate part of the code base. This means a programmer must take the time to find another section of code to read while attempting to figure out what the original secion of code does. This can make debugging, or even learning a new code base, significantly harder.

1.1.2 Advantages.

- Increases legibility.
- Helps with self-documenting code.
- Potentially easier to debug.

1.1.3 Disadvantages.

- Increases the number of places a programmer must look to follow the logic of a program.

1.2 Singleton

1.2.1 Overview.

A Singleton involves using a class that can only ever have a single instance. Generally, the class is asked for the instance of itself. That instance is then told to do something to its state. This is useful for maintaining a single source of truth throughout a body of code.

Despite the advantages offered by using a singleton in this situations, there are many criticisms of the pattern in general. It is often considered an anti-pattern, and its use appears to be falling out of fashion among computer scientists. It is effectively just a more complicated global variable.

Singletons also make unit testing an application vastly more difficult. They make it very hard to guarantee the state of the system during testing. Additionally, it adds another potential source for error in multi-threaded applications.

1.2.2 Advantages.

- Provides a single source of truth.

1.2.3 Disadvantages.

- Generally considered an anti-pattern.
- Effectively just a fancy global variable.
- Makes unit testing hard.

1.3 Facade

1.3.1 Overview.

A Facade pattern is where one class functions as an interface for another class. The class functioning as an interface is used to hide the complexity of the interface of the hidden class.

In our code, we could use this to hide some of the details of the calculations by creating an extra layer between that and the main code.

Despite all of that, the simplicity of this program makes this little more than a normal class with normal functions.

1.3.2 *Advantages.*

- Can hide some complexity from the high level logic.

1.3.3 *Disadvantages.*

- Not much of a distinction between this and normal classes in this project.

1.4 Visitor

1.4.1 *Overview.*

The Visitor pattern abstracts some of the capability of a class from its implementation. It can be used to add functionality to a number of classes at the same time.

Using the Visitor pattern helps conform to the open/closed principle. This is the idea that classes should be open to additional functionality, but closed to modification.

This abstraction could be used in our code by telling each element in the table to visit itself, and update itself. Unfortunately, this abstraction might not be able to contribute much to a project of this size.

1.4.2 *Advantages.*

- Follows the open/closed principle.
- Elements in the table would know their own part of any calculation.

1.4.3 *Disadvantages.*

- Might not actually add anything to a project of this size.

1.5 Pipe and Filter

1.5.1 *Overview.*

Pipe and Filter is an abstraction that is used primarily in situations where calculations can be easily serialized. When calculations always happen in a specific order, the programmer can separate parts of the calculations into multiple chunks. The output of each chunk of code would be fed into the next part as its input.

When code is abstracted into discrete chunks and set up in a row, it is easy to write self-documenting code. It is obvious what the logic is at a high level.

The Pipe and Filter abstraction could be used to serialize the creation of the output table. This could be done by creating a separate function to handle each column of the table. The entire input table would be passed to the first function. The same table, but with its first column changed into a range of values, would be output. That output would be used as the input for the second function. This process would continue until the every part of the table that needs to change has been recalculated.

In effect, this pattern would be splitting the calculation of each column into its own step within the broader project. They could even be split into different files, such as atleast, done_percent, learning_curve, nprod, optimism, pomposity, productivity_exp, productivity_new, r, to, ts, and klass. Columns that were not in that list do not require calculations in this part of the code base, so they would not require any code to calculate changes for them.

The Pipe and Filter abstraction would make super more granular from a project-wide perspective. This granularity could allow for each portion of the algorithm to be focused with a clear intent and logic.

While this abstraction seems perfect for the rewrite of super, it has some drawbacks. This abstraction was already used by the overall project. Because of this, it would almost be too easy to implement. This abstraction would potentially lead to duplication of code, because of how similar the calculations performed on each column are.

1.5.2 *Advantages.*

- Abstracts actual implementation from the high level logic.
- Encourages self-documenting code.
- Increases the granularity of the overall project.
- Makes each step of the algorithm very clear and defined.

1.5.3 *Disadvantages.*

- Already used for the overall project.
- Could lead to duplication of code.

1.6 Template Method

1.6.1 *Overview.*

The Template Method abstraction allows certain steps or parts of an algorithm to be implemented by deferring to subclasses, while defining these steps abstractly.

This design pattern could be used to abstract the calculation of the value range of each group of cells to a function dedicated to it. The adaptability of this approach will not benefit such a small program much, but it could be quite beneficial in larger projects.

1.6.2 *Advantages.*

- Each section of a column could be conveniently delegated to a function.

1.6.3 *Disadvantages.*

- For our purposes, we are unlikely going to need much optimization or deviation from the standard way of calculating the value ranges.

1.7 State Machine

1.7.1 *Overview.*

A Finite State Machine is a logical abstraction where a program, or part of a program, is viewed as a finite set of possible states that model every possible situation it could find itself in. Each of these states contains information about its possible state changes. These state changes would be selected based on some input received.

States in a State Machine can often be easily represented by an object in an object oriented language. Their transitions can often be easily modeled as a switch statement, where each possible transition is a separate case.

The use of State Machines is good for splitting the logic in the program into smaller parts that are easier to conceptualize. This allows complex behaviors to be easily expressed and modeled. State

Machines can be used when writing an interpreter for a domain-specific language. The logic could be broken down into states and transitions.

For our rewrite of super, a State Machine could be used to calculate the new values for each cell in the table. The transitions between stats could depend on the new value encountered in each cell.

Unfortunately, this would be difficult to model in a state machine, because the calculation of each cell in a column is dependent on other rows in the column. This could be done, but its complexity would be vastly expanded by the nature of the problem.

1.7.2 *Advantages.*

- Effective at breaking parts of a job into discrete chunks of logic.

1.7.3 *Disadvantages.*

- Difficult to implement for this specific problem.

1.8 Adapter

1.8.1 *Overview.*

The Adapter pattern is when a programmer uses a single interface for multiple classes. This is generally used to make sure that different classes can work with each other.

Since super interacts primarily with columns, we could wrap the normal table representation, which is a list of rows, to instead be a list of columns without having to rewrite it entirely. The underlying representation would not change, but instead of repeatedly calling rows[index][column_Of_Interest], one call could be made like columns[column_Of_interest], which returns the actual list of values we are operating on.

Unfortunately, this reuse of a data structure in a different way could become very confusing.

1.8.2 *Advantages.*

- This would allow for the reuse of code for the table.

1.8.3 *Disadvantages.*

- The reuse of the same data structure to store data differently may cause confusion.

1.9 Decorator

1.9.1 *Overview.*

The Decorator design pattern is used to dynamically add to the behavior of a class or function. In Python, the @ symbol can be used to wrap a function inside of another function to add to its functionality. One of the more notable examples is using @contextmanager to ensure that code can be used as part of a "with" command.

In our code, this could be used to create a wrapper around the code that evaluates each cell. This wrapper could serve to track the state of the section of code being parsed.

Unfortunately, this abstraction can make code more difficult to interpret if it is used incautiously.

1.9.2 *Advantages.*

- Can add to the behavior of a function.

1.9.3 *Disadvantages.*

- Can be more difficult to read.

1.10 REST

1.10.1 *Overview.*

A REpresentational State Transfer (REST) API is useful for allowing programmers to create a stateless server to interact with clients of a web application. This is important, because it allows an arbitrary number of users to connect to a server.

For our rewrite of super, we could use a RESTful interface. We would at least be able to separate the calculations for the ranges of rows into a section of code that does not need to maintain a state.

Unfortunately, this may end up being a trivial implementation of a REST API. While this abstraction can be used to separate the logic of the code into layers, the effort might be on a small program. The effectiveness of the abstraction would depend on how much of the program could be performed statelessly.

1.10.2 *Advantages.*

- Separates the logic of calculating ranges from the state of the program.

1.10.3 *Disadvantages.*

- May be a trivial implementation of a REST API.
- Effectiveness of the abstraction depends on how many actions can be performed statelessly.
- Is a somewhat forced abstraction.

2 EPILOGUE

2.1 Overview of Rewrite

We chose to rewrite super for our second part of project 2. This program reads a table, calculates value ranges, and returns a table updated with ranges of values in columns marked for processing. We chose to do this rewrite in Elixir for extra credit. To accomplish this, we chose to use three abstractions: Servant, Singleton, and Facade.

When we were initially considering what patterns we wanted to use in this project, we planned to use REST and Marker Interface. We discarded both of these abstractions during our research into their feasibility.

We had planned to use a restful interface between our high level logic and the calculations performed on the table. Below the restful interface, we would have maintained no state between calls. Unfortunately, the structure of the language made this extremely difficult to implement, because only a single argument can be passed into a map function. In the end, we decided to replace the REST pattern with the Servant pattern.

We planned to use a Marker Interface pattern to store meta-data about the table, such as the list of ranges found that are printed above the table in the output. This was rejected for effectively the same reason as the REST interface was. The map function in Elixir only takes a single argument. In the end, we decided to replace the Marker Interface pattern with the Singleton pattern.

2.2 Structure of Code

The first thing our program does is parse the input table. When the input table is being parsed, it is stored in a list of columns, where each column is a list of cells in a specific column of the table.

For each column tagged with "$", the program establishes cuts in the column, and establishes value ranges for that slice of the column. These ranges are recorded and stored in every cell in that range of rows for the column being calculated.

After the value ranges have been calculated, and the list of columns has been updated, the table is printed row-by-row. This process is complicated by the structure of the language, and the fact that the table is stored by columns instead of rows.

2.3 Abstractions Used

2.3.1 Servant.

We used the Servant pattern to assist in outputting the table. This ended up making printing our table far easier than it otherwise would have been. Because we stored the code as a list of columns, it increased the complexity of printing the table row by row. Within our code, printTableHelper and printRowHelper are servants for printTable and printRow respectively.

These helper functions are used to get around the fact that Elixir restricts programmers to a single instruction when specifying a conditional execution of code through "cond do".

The printTable function is used to crawl over the table (row-by-row). If the row exists within the table, printTableHelper is called. The printTableHelper function calls printRow and then calls printTable on the next row of the table. If the row does not exist in the table, the printTable function does nothing.

The printRow function is used to crawl over a row (cell-by-cell). If the cell exists within the table, printRowHelper is called. The printRowHelper function prints the value in the given cell, and then calls printRow for the next cell in the row. If the cell does not exist within the table, printRow prints a newline character instead of calling its helper function.

2.3.2 Singleton.

In our code, the Storage module functions as a singleton. It maintains a single source for truth for the code to use in various places.

Storage is used by initializing it with the begin() function, setting values with the set(key, value) function, and getting stored information with the get() function.

2.3.3 Facade.

We implemented the Facade abstraction as a wrapper around an Agent function. Instead of calling Agent directly, the rest of the code goes through Storage.

3 END

Since our program does not work, but we put in considerable effort with a wonderful, yet difficult language choice, and has been finished on time, our base grade (before modifiers) is 8 marks. Our choice to replace super gives us a 14 percent modifier, and our choice of Elixir earns us an additional 14 percent modifier. Our choice of patterns did not yield us any extra credit. Since the modifiers are additive, this means we have earned 8 * 1.14 * 1.14 = 10.39 marks for this assignment.