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 `dom`
Selection of Language Typescript

1 POSSIBLE ABSTRACTIONS

1.1 Iterator

1.1.1 *Overview.*

Could be used to iterate over the stream to create the table to begin with, or to iterate / get specific rows after construction. This could be implemented as an object that tracks the next row to use. It could also be implemented trivially through simply iterating over the available rows.

1.1.2 *Advantages.*

- Our specific iterator could subsume the task of finding another row (currently done by `another()` in `lib.lua`)
- Possibly make the code more readable if the interface of the Iterator makes it clear we are retrieving a row
- If the internal representation of the table or rows changes, the caller interacting with the Iterator will not
- If using already created constructors, such as those provided for arrays in Typescript, this would simplify and minimize code as opposed to explicitly getting specific indexes

1.1.3 *Disadvantages.*

- It is unlikely we will need or discover many different ways of representing the table
- This pattern might add nothing that the language isn't already capable of doing

1.2 Builder

1.2.1 *Overview.*

The builder pattern separates the construction of a complex class from the class being constructed. JavaâĂŹs StringBuilder class is an example of this pattern in action. It is useful, because it allows for the object to be constructed in multiple stages.

This pattern could be used to split the construction of the table into stages, where each row is constructed and added to the table. Additionally, each row could be constructed in stages, where each element in the row is calculated and/or added individually.

This would not require everything StringBuilder does in Java. It wouldn't need to be able to fetch specific characters at any given time. That being said, the row builder could be used to filter which rows are returned. A row builder could be given some kind of filter to know to ignore certain indices when returning the row it has built. This filter could easily be stored as an array.

1.2.2 *Advantages.*

- Would allow for an easy expansion of the functionality of the program.
- Allows for greater control over the process of building the table.
- Reduces run time as building only happens once

1.2.3 *Disadvantages.*

- Requires a separate builder for the table and the rows.
- The information being constructed must be mutable.
- If we needed to use dependency injection, this could get in the way.

1.3 Pattern Matching

1.3.1 *Overview.*

Matching a pattern over a sequence of symbols or tokens (e.g. regex expressions) - typically replaces large procedural code blocks for parsing.

Given that dom is fairly simple, pattern matching can easily be implemented. Useful for: When we need to parse the input table for column headers, rows, and cell values. (currently done by rows.lua)

1.3.2 *Advantages.*

- Very easy to construct for simple programs. For example, a regular expression for matching a 9 or 10 digit number can be as simple as `ˆ\d{9,10}$`.
- Useful for structured data where there is little input variation.

1.3.3 *Disadvantages.*

- Does not handle edge cases well. While most data can be processed using patterns, edge cases are usually quite difficult to control for more complex tasks.

1.4 Composition

1.4.1 *Overview.*

The composition pattern would allow us to treat every row in the table uniformly. The table could be treated as being composed of rows. Each row could also be treated as if it is composed of individual slots for each column.

If this was done, each row's construction function would only need to call a constructor for each of their columns. This would allow for the construction of each row to be handled by each of its component columns.

Unfortunately, this abstraction becomes nearly meaningless when applied to data within a single row. It could be done by treating a row as a composite of data points. Each data point would be treated differently, based on which column it is in.

Given that only the dom column is actually being calculated in this example, it would be a very trivial application of composition. The function that everything else executed would just return the input data.

The biggest problem with this being applied to the calculation of the dom score is the fact that the dom calculation requires access to other elements of the row. This could destroy the entire purpose of the composite design pattern by requiring elements in the row to be calculated in a specific order.

1.4.2 *Advantages.*

- Shifts the responsibility for the logic of calculating each row to the row itself.

1.4.3 *Disadvantages.*

- Somewhat undermined by the dom score needing information from other parts of the row.

1.5 Factory

1.5.1 *Overview.*

The factory pattern allows the construction of objects of a type not specified directly by the code calling the factory. Sub-classes determine what will be constructed.

This could be applied to dom by creating a table factory, where information is passed to it, and it decides whether it should be creating the table, a row in the table, or a single element within a row. The table factory could be given all of the information read into the program, break it into rows, and use a row factory to handle the creation of each row.

This abstraction will probably not be useful, because there is very little to construct. In the best case scenario, the factory works, but saves no time or effort. Despite initially seeming to make sense, this abstraction will probably not aid the program in any way.

1.5.2 *Advantages.*

- Moves the logic for construction away from the main body of the program.

1.5.3 *Disadvantages.*

- Overcomplicated for the task being accomplished.
- Only a trivial case of a factory.

1.6 Template Method

1.6.1 *Overview.*

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

This design pattern could be used to change what it means for a given row to dominate another, or how exactly that is calculated, without touching the remaining iteration needed for the rest of the algorithm. 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.*

- Useful if optimizing just the process of determining if a given row dominates another
- If another condition rather than domination is meaningful, a subclass could implement that part of the algorithm differently

1.6.3 *Disadvantages.*

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

1.7 Pipe and Filter

1.7.1 *Overview.*

Dividing a large task into an ordered series of smaller tasks that filter the output of an incoming task and pipe to the next

We are already dealing with a pipeline, but this section of the pipeline - Dom - can still be broken up into more tasks. The internal information flow would have to slightly change but could look something like: (1) stream IO / constructing the table, (2) for a given row, determine and assign which rows it is to be compared to, (3) calculating the dom score for a row given this assigned information and appending it to the table

1.7.2 *Advantages.*

- Makes each step of the algorithm very clear and defined

1.7.3 *Disadvantages.*

- In order to make this sub-part of the pipeline itself fit this pattern, the flow of information is unnecessarily convoluted
- Information that was previously only stored / had scope within a function is now passed to another pipe, which takes up unnecessary space

1.8 State Machine

1.8.1 *Overview.*

A state machine separates logic from the program into smaller parts that express that logic simply. This allows complex to implement knowledge to be expressed in an understandable way. Writing an interpreter for a domain specific language is an example of when a state machine could be useful. You could break down the logic into states and transitions for a good amount of language rules.

A state machine could be used in dom where the states are reading in a row, processing a row, calculating a domination score, and adding the dom score to a row; and the transitions are the rows read from input and the values in each row.

1.8.2 *Advantages.*

- Useful for describing the high-level logic of a system
- Can apply formal methods to check the form of the state machine

1.8.3 *Disadvantages.*

- Not practical to use just state machines for a large system, it becomes too convoluted.
- The process of dom should always be the same - it is just the same algorithm run over a different source table. So while it is true that at any given point this system will be in a state - the transitions will always be the same. Especially for our purposes, the only real variability would be if dom was going to be used on tables with different headers.

1.9 Singleton

1.9.1 *Overview.*

The singleton pattern that maintains a single instance of an object. If it needs to be used, the class is asked for its instance.

This could work very well for the table, since there only ever needs to be a single table. This would remove any need to create a new table.

1.9.2 *Advantages.*

- Maintains a single instance of a table.
- Removes the need to construct a new table.

1.9.3 *Disadvantages.*

- Would get in the way of expanding functionality.

1.10 FlyWeight

1.10.1 *Overview.*

Minimizing memory by sharing data with other components as much as possible. This most commonly means using references to data up until the point that this data must be referenced. This most easily could be used for passing around the table, and only de-referencing when we are accessing its contents.

1.10.2 *Advantages.*

- Saves space
- Possibly could minimize run-time if functions aren't passing by reference to begin with and are wasting time creating copies of the table

1.10.3 *Disadvantages.*

- For the sizes of the table we expect to be dealing with, it is unlikely this would save noticeable time or space

2 EPILOGUE

2.1 Overview of Rewrite

We chose to rewrite dom for our project 2a. This program reads a table, calculates dom files, and returns a table updated with the dom files. We chose to do this rewrite in TypeScript for extra credit. To accomplish this, we chose to use three abstractions: iterator, builder, and pattern matching.

Execution of our program starts by parsing the header of the table towhat should be done with each column. These attributes are stored in an array, called columnsArray, for later use. This array is passed to each instance of RowBuilder on its creation. par As the program reads each line, it creates a RowBuilder, parses the columns, and stores their information in the RowBuilder that was just created. After the columns are pushed to the RowBuilder, it is added to an array that stores all of the builders.

After all of the rows have been stored in RowBuilders, their information is used to calculate their dom score. These dom scores are sent to the RowBuilders to finish off the rows. Each RowBuilder uses its columnsArray to filter the information that it returns. Each one combines the columns in its row into a single string to be added to an output table. When building that string, it adds commas to correctly format the table. This table is output for the next program in the pipe to use.

2.2 Structure of Code

2.2.1 RowBuilder. RowBuilder is a class that provides support for building rows a piece at a time. Its constructor(Array<Num>) function takes an array for use in filtering the returned rows and stores it in an array called columnsArray. Its addColumn(number|string) function stores a number or string in the builder. Its get(number) function returns th element at a given index. Its getString() function returns the entire contents ofethe builder, filtered by columnsArray, as a formatted string. This format conforms with what was expected of the original dom file.

2.2.2 IO. IO is a class that provides support for handling the input operations required by this program. Its constructor() function establishes a reader to read from process.stdin. Its getReader() method returns the constructed reader for use outside this class. The close() method closes the reader stored in this object. The getReader() function should not be relied on after the close() function is called.

2.2.3 Num. Num is a class used for keeping track of statistics for each of the columns. Upon reading a new cell, a call to numInc() updates that column's Num values. The other purpose of Num is to keep track of the type of data that any particular column is - which equates to the prefixes on the header values: "?", ">", "<", "", *or*"!".

2.2.4 DomCalculator. DomCalculator is a class that is used for calculating dom scores for each of the rows. It maintains an array of RowBuilder objects (called rowBuilders), and an array of Num objects (called columnsArray). Its constructor(Array<RowBuilder>,Array<Num>) function sets both rowBuilders and columnsArray to the given values.

DomCalculator's calculate() method calculates dom scores based on randomized row comparisons. By default the number of comparisons is 512, but in cases where the table size n is smaller than this, it will compare n-1 times. The output value is the result as a string.

DomCalculator's dom(RowBuilder, RowBuilder) function contains the actual math for comparing two rows.

DomCalcu atorl numNorm(Num, number) function

The randomized(number, number) function provides a random selection ofrows from the table. This selection is limited in size by the size value it is given. par ß'ubsectionAbstractions Used

2.2.5 *Iterator.*

Once we have built the rows and columns, we used the iterator pattern to iterate over the each row of input information to find the minimum and maximum values and calculate the domination score. We then iterate over each row and append the calculated domination score to the end of the row.

2.2.6 *Builder.*

We used the builder pattern in the form of a row builder. In our code, we called it RowBuilder, and used it to build rows over time. We were afraid that it would end up being useless as a trivial application of a builder, but it actually does have a place in the program.

RowBuilder is used to abstract some of the logic of building the table from the rest of the code. When its row is fitished, in can filter and format the information it returns as a string.

RowBuilder ended up being similar to Java's StringBuilder. RowBuilder has functions to accept new items to add to the row, return a single item in the builder, and return the entire row that was constructed. Unlike Java's StringBuilder, RowBuilder has the internal logic to ignore columns that were specified at the time of its construction.

2.2.7 *Pattern Matching.*

Our initial intention with using this pattern was to see if we could eliminate some of the procedural code related to parsing the headers and rows of the table. In the end, we ended up only matching on the separators (commas and spaces) of the rows and headers, which was used to split up the line into an array of elements. Then we would test the first character of each element in the header's current index to determine how we should treat that cell.

This is essentially the same strategy used in the initial pipeline code for dom. Matching first to separate the line and then procedurally testing on the first character just seems like the most simple and straightforward way to do things.

3 END

Since our program works, and has been finished on time, our base grade (before modifiers) is 10 marks. Our choice to replace dom didn't yield us any extra credit, but our choice of TypeScript earns us a 14 percent modifier. Additionally, our implementation of pattern matching earned us another 7 percent modifier. Since the modifiers are additive, this means we have earned 10*1.14*1.07=12.198 marks for this assignment.