# Group K: Project 2

Part A

Eric McAllister†
Department of Computer Science
North Carolina State University
Raleigh, NC United States
ewmcalli@ncsu.edu

Jacob Myers
Department of Computer Science
North Carolina State University
Raleigh, NC United States
jrmyers5@ncsu.edu

Cody Nesbitt
Department of Computer Science
North Carolina State University
Raleigh, NC United States
cznesbit@ncsu.edu

## ABSTRACTIONS

### 1. Iterator

According to *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the Iterator design pattern "provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation". When the word "iterator" is mentioned, it is meant in a way that would be defined as an object used in order to "separate the responsibility for access and traversal of the elements stored in the aggregate object" and to assign it to another object.

In the "dom" filter, like in most sections of the pipe in this project, the input is received as a large table, consisting of (up to) several thousands of rows. In each row, there are a number of columns, each of which has their own properties as defined by the symbol which accompanies the column header - < for maximizing the column, > for minimizing, and so on. In dom, it would make sense to be able to traverse each one of these row/column combinations ("cells"), which is why an iterator would be useful.

In general, iterators are very easy to both implement in code as well as read what they are doing. Also, many iterators - like the standard one in Java, have the ability to call a "hasNext()" type method which allows you to check if there are more elements before trying to move to the next step. However, there are a few shortcomings when it comes to using iterators. By design, an iterator stores its position rather than traversing by using indexes. This may get in your way if you need to update the data structure in which you are traversing, since the position of the iterator could then become incorrect.

### 2. Builder

According to *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the Builder design pattern "separates the construction of a complex object from its representation so that the same construction process can create different representations". As a basic, step-by-step definition, the Builder pattern allows the programmer to have a "construction process" which stays the same yet the finished products can have different representations. The Builder design pattern can be compared to the abstract factory pattern as both can be used to abstract object creation.

When thinking about how the Builder design pattern could be useful in the "dom" part of our pipeline, we must think about what we want our finished product (dom output) to look like. Ideally, we would like the output to function as a type of spreadsheet, with each cell containing a value as well as some of the cells containing values which were computed based off of other cells (more on this in the next paragraph). The Builder design pattern can help with this vision by assisting the programmer in creating a "cell", or an object which contains both a value as well as a unique identifier which tells us how to interpret cells of a particular column (like a $ or ! in this project). So, in review, the Iterator would visit each value in the input, and pass that value as well as the unique identifier into the cell builder, which would take care of associating those two values.

If the Builder were to be compared to another design pattern, such as the abstract factory design pattern, a pro would be that rather than creating a family of related objects, the builder design pattern is about creating the object and returning it as the final result. One of the only cons that the Builder design pattern falls victim to would be the creation of a decent amount of extra code as well as losing a bit of readability.

## 3. Spreadsheet

The Spreadsheet design pattern, also known as “Dataflow” or “Active data” design pattern, contains similar properties to your basic Excel spreadsheet with rows and columns of data. However, what separates spreadsheets from text files filled with rows of numbers would be the ability to implement formulas. With formulas, the creator can set the value of a spreadsheet’s cell to be the result of some calculation done on another cell, making that resulting cell a “dependent” one while the cells that it is based off of would be classified as “independent”. In other words, when data changes, the dependent data also changes automatically.

After outlining the iterator as well as the cell builder design pattern, is it not too difficult to see how the Spreadsheet design pattern could be useful in our “dom” section of the pipeline. After all, the input data already contains several rows and columns. In this section of the pipe, an extra column is added to the input data which contains all of the “dom scores” for the lines. This dom score is calculated based on a custom function which contains the dom score formula, resulting in a number based on the order and values of earlier columns. Our code sets up the input file as a spreadsheet, with each cell containing a value and a “type”, as well as appending a “dom” column to the end of the input data which is dependent on the other data in the row.

A pro of using the Spreadsheet design pattern would be the autonomous nature of the cell update. It is easy to affect the dom value, simple change the value or the type of cells in which the dom value is dependent on. Of course, this could also be a con. Programmers must be careful when using a Spreadsheet design pattern since it is very easy to forget what cell is dependent on what value, making for less-than-ideal readability for other teammates.

## 4. Delegation

The delegation pattern is the practice of creating an interface or abstract class that expresses a certain behavior, but leaving the implementation of that behavior up to the implementing classes. This is very useful if you want to be able to call the same function on many different types of objects and have them all implemented differently.

In the dom filter, the delegation abstraction could be used to process each cell differently. Since the filter takes in a table of values and each column represents something different which needs to be handled a different way, there could be a Cell interface with a handle() method which each Cell subclass would implement differently based on its needs.

This could be beneficial because it would allow the overall program to deal with all cells in the same way, without having to worry about their data and its significance. However, it could also lead to confusing behavior and unnecessary complexity if it’s not needed or poorly implemented.

## 5. Letterbox

The letterbox pattern is when you split a program up into individual parts whose only exposed functions are to send and receive data to/from each other. This is useful when you’re writing something that always performs a series of tasks in order or has several distinct components that you want to keep separate.

This pattern could be used in dom by splitting up the tasks into smaller modules. For example, one box could read in the data and pass it to the next one, which would perform calculations and pass it to the last box, which would write the new data out. In this way it’s kind of like an assembly line.

This abstraction is beneficial because it helps break down the flow of a program and make it more clear what’s going on in what order, as well as encapsulating behavior for each chunk to keep it out of the main program. However, if changes were made that required modules to communicate more or in a different way, it could be difficult to refactor.

## 6. Visitor

The visitor abstraction allows you to define a new operation that multiple classes will implement without having to change the classes themselves. The new operation is located in a separate class that has functionality defined for each type of object it might visit.

In dom, a CellVisitor class could be implemented to visit each cell in a row, with methods like visitCellA(), visitCellB(), etc to handle each one differently as needed.

This pattern is useful for easily adding functionality to existing objects on an as-needed basis. However, it also adds complexity to the class hierarchy and can become convoluted if used too freely.

## 7. Hollywood

The hollywood abstraction involves splitting up the program into modules that can be called upon to perform specific functions and return callbacks. The modules call on each other and register callbacks, and for this reason the pattern is often called "callback heaven/hell".

For the dom filter, this could be used to split the data pipeline up into smaller chunks that can be called and used as needed, such as a cell processing chunk, a calculator chunk, and data read/write chunks.

This pattern is beneficial because it allows more flexible program structure, since the calling method determines the callbacks for each function instead of having them predetermined. However, as the term "callback hell" implies, too many custom callbacks can quickly get convoluted and make for confusing debugging.

## 8. Factory

According to the *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the Factory Method design pattern is a creational design pattern that acts as a factory to create multiple objects. This would be implemented if you had a situation where you needed to continuously make a certain kind of class many times over in a larger program. The patter would all you to innovate a call instead of making a direct call to a constructor each time. This means that you can streamline object creation. This will be useful in the case where the object has different states that it can be in and runs methods in its construction based on the state that was passed to the object.

In dom this caused be used in two different ways. The first way would be to pass an unformatted row into the factory. The factory would then reformat the row with the added '0' dom score on the end and output that object back to use. This could also do calculations while it has that row like comparing local maxes and mins. The factory pattern would allow a general run over of the row in a quicker manner. The second way you could use this pattern would be if you passed the factory two rows at a time. This would have to be after all the data has been read in so it can find a random row to pass into dom. However, once it has two different rows it could calculate the dom score, find local mins and maxes, and then go on and return the new row with adjusted values for use to use later.

The pros of using this kind of pattern would all for more general instruction making for the row objects. It would allow for other calculations to be easily added to the factory that could change the values of elements in the row. This is a big pro because it allows the code to be expanded upon. Another pro is it increase simplicity by reducing clutter. All the calculations of the row are done in one class and each row should come back from the factory exactly how you would expect a row to look.

However, there are some cons. For this program particularly there are not many calculations that need to be done on the rows. The rows also are independent until the dom score needs to be calculated. This means that passing a single instance of something wouldn't be enough you have to pass two rows for the information to be calculated. This could be enough to look at another design pattern to fix the issue of calculating rows' dom score. Also if the program's structure does not have rows as an object there could be difficulty passing the data between the classes.

I would recommend not to use this pattern for the implementation of the entire calculations of the row. However, if you are running the entire pipe this could be implemented for the first row of the program. It can set up the data specifications on what columns need to be maximized and etc. However, even then this pattern is very optional, but I would lean towards not using this pattern to solve this problem.

## 9. State Machine

The state machine design pattern allows for the user to define different states which can be represented when certain objects in the code fall under certain defined conditions that the programmer states. The state of object can be updated when it meets certain prerequisites. An example of a state machine could be a vending machine which does not vend your item until you have reached the final state (when you have inserted enough money to get your selected item assuming this vending machine has items of the same price and only that price).

In dom this could be used to implement of a system of finding the dom score and assigning the score in an organized manner. After all the data has been read in and the code goes through to have the dom score added the rows each could be an object in an array that have a state. When the program goes through to decide random rows it will choose the rows that are in an untested state. After the dom score has been calculated for a row it moves to the tested state. Then the dom score would not re-try rows and continue just comparing rows that have not yet been compared to each other. Only if it is the last singular row would it go back and compare it to an already tested row.

A con to this would be a lot of overhead that would be needed to get the state pattern working. It would require setting up the states as well as a choice algorithm for picking the open dom scores.

In my opinion I believe if someone really wanted to implement a state machine then they should. However, know that are other ways to get the code from stateA to stateB without a state machine.

## 10. Singleton

The singleton pattern is a design pattern that prevents the program from making more than one instance of a certain object. For instance, if you have code for a chess board that is a singleton then the program will only allow you to make one instance of the chess board. You don't want to have two chess boards. If the Knight moves to C6 and there are multiple boards then that could cause some confusion on which board the knight should worry about. The singleton allows for cleaner object management by restricting the usages of creation.

For dom a singleton could be used for the matrix that holds all the rows. It would allow only one matrix to be created and managed at any given time. However, that would be about the only use of a singleton in this program.

As far as pros goes, there aren't many in this case because there are not going to be multiple matrices. There should be only one matrix object and no other part of the program should attempt to make a storage unit for the cells.

A con of using a singleton would be testing as well as unnecessary encapsulation of the problem. Retrieving the values would only be from accessing the singleton and retrieving the array and passing it around for operations would become more difficult. Ultimately, I would not recommend using the singleton pattern for this assignment.

## EPILOGUE

In our code, we ended up using the Iterator, Builder, and Spreadsheet abstractions. Although it is easy to see how these abstractions could make a codebase easier to read as well as more efficient, our group ultimately decided that having to implement all three of these abstractions for the single "dom" section of the pipeline made things somewhat overcomplicated. After reading about the requirements necessary to complete "dom", we established that an Iterator provides a way to access each element, a Builder allows the programmer to have a "construction process" which stays the same yet the finished products can have different representations, and a Spreadsheet pattern allows the creator to set the value of a spreadsheet's cell to be the result of some calculation done on another cell.

With those definitions in mind, the flow of the program started to come together. We would use our iterator first to visit each line, then subsequently visit each value in that line. Next, we would send that value to the Builder, which would associate that value with its "type", defined by the symbol ($<>?!) in front of the title of that column. Lastly, we would enter those cells in a spreadsheet, with the very last column being the "dom score". This score is where the properties of a Spreadsheet pattern come in handy, since the value of the score is dependent on the values of earlier cells in the structure.

As mentioned earlier, our group thinks that we probably could have written more condense code if we were not required to implement three separate abstractions in just one section of the pipeline, since it ultimately made our code more convoluted. Moving forward, we would recommend using iterators when visiting each element is necessary, although one has to remember that iterators work by storing its relative position rather than its current index, which caused us problems a few times during development when we wanted to reference an element that was a couple of positions ahead of the iterators current position. In terms of the Builder pattern, the only thing that we noticed that was a shortcoming would be the extra code needed to implement this pattern which would in term hurt the readability of the program. Lastly, we would recommend a Spreadsheet pattern to programmers whose data could be set up in that type of 2D table format that spreadsheets are in, since it is easy to both view and edit data that way. As mentioned earlier, one issue with using the Spreadsheet pattern is that, since some cells are dependent on the values of other cells in order to calculate their final value, it is easy as a programmer to forget what cells are “linked” together, which results in a headache during the debugging process.

## MAXIMUM GRADE EXPECTED

We implemented the dom filter (1 star) in Python (1 star) using the abstractions iterator (1 star), builder (1 star), and spreadsheet (3 stars). Therefore, the maximum grade we can expect is 2 bonus marks for a total of 12.