Dom 2b

Using CoffeeScript

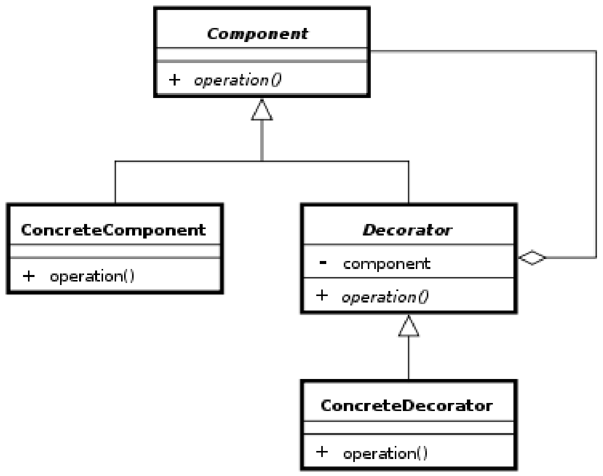
Wil Elias
Computer Science
North Carolina State University
Raleigh, North Carolina
mwelias@ncsu.edu

Nolan McDonald
Computer Science
North Carolina State University
Raleigh, North Carolina
nrmcdona@ncsu.edu

Ian Fisher
Computer Science
North Carolina State University
Raleigh, North Carolina
imfisher@ncsu.edu

ABSTRACTS CONSIDERED

1 Decorator



In our case, this abstraction could have been used to create a type-checking decorator. This decorator, AcceptArguments, would act on all existing methods, and be passed an argument list of parameter types. The decorator would examine the list of types, as well as the wrapped function's arguments, to determine if the function's arguments matched the expected types and could be executed [1].

Advantages: A decorator allows actions to be applied to other functions at run time, possibly imposing new behavior before, after, or even within, the regular code execution.

Disadvantages: Not all languages support a straightforward syntax for using decorators (as Python does). In the case that decorators aren't a key mechanism of a given language, such as with CoffeeScript, a decorator may require clunky nesting of functions and odd manipulation of variable arguments, resulting in a debugging cost that outweighs the efficiency of decorators.

2 Constructivist

The Constructivist abstraction could prove to be quite useful with CoffeeScript as parameters in CoffeeScript do not have a declared type. As such type checking could be a powerful tool for ensuring that the pipe runs in its totality. [2]

Advantages: The Constructivist pattern ensures that no matter the given input to a function, the function will run and produce some kind of behavior. This is helpful for ensuring that the input that functions receive are of the right types.

Disadvantages: Since the code won't fail due to parameter issues it can make debugging code more difficult. This pattern can also lead to snowballing issues if used in a system such as the dom filter, in which data is modified and referenced many times. That snowballing can eventually lead to false results if inputs are ignored due to typing errors.

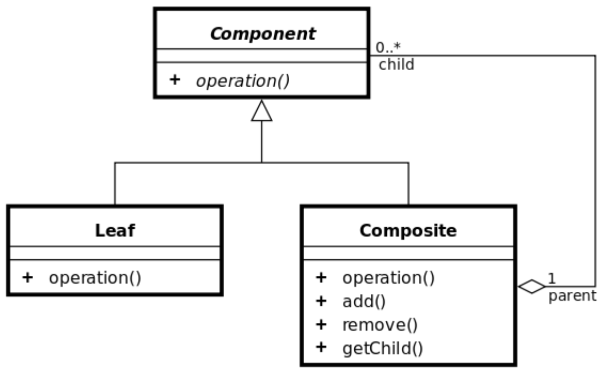
3 Declared Intentions

The declared intentions pattern aims to reduce type mismatch errors that have been problematic to programmers since the early days of computer programming. It includes a strict type enforcer, requires that procedures and functions declare what types of arguments they expect. Finally, if a caller sends arguments of unexpected types, type errors are raised and the procedures or functions called do not execute [2].

Advantages: Can narrow down accepted types more than most existing type systems that are built into languages and can help to make debugging easier down the line when maintaining or developing code further.

Disadvantages: Does not do any checks related to the size of arguments or whether arguments are empty. It also can be redundant to implement declared intentions in strongly typed languages which can increase development time and reduce code visibility.

4 Composite



The Composite abstraction allows developers to represent part-whole hierarchies for objects. It also allows clients of an object or objects to ignore the differences between a single object or a composition of objects. In this way, the clients will treat all members in a composition of multiple objects, uniformly. This allows us to simplify client code as it doesn't have to worry about whether its dealing with multiple or singular objects or types of objects [1].

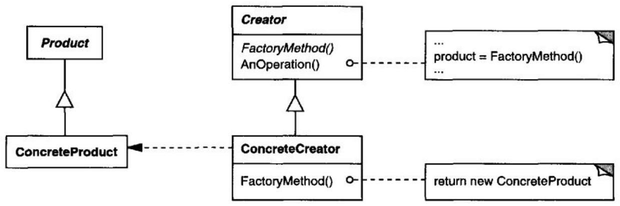
The important part of the Composite pattern is an abstract class that represents both the primitives and their containers. The Composite pattern also makes it easy to add more components to the system, however it can make the system overly general and limits how you can restrict what components are part of a Composite. This means you can't rely on type-checks to enforce constraints. You have to fall back to run-time checks instead [1].

Advantage: The composite pattern is great when you want to represent part-whole hierarchies within objects or want to build objects procedurally piece by piece. It is also useful when you want clients to be able to ignore differences between compositions of objects and individual objects.

Disadvantages: You can't rely on a type system to enforce constraints on the components added to a composite when you want such a composite to only contain some number of a certain type of component.

5 Factory

The factory pattern allows the programmer to define an interface for creating an object without explicitly knowing the type of object it needs to create. The example depicted below shows a program utilizing a creator class that calls upon a ConcreteCreator to handle the creation of the ConcreteProduct [1].



Advantages: Knowledge of the object type needed to create is not needed. This provides more flexibility in object creation making it easier to add additional object types later on and also reduces coupling between logical and creational portions of the program.

Disadvantages: Because the type of an object is decided by the factory class and its subclasses, the program may have trouble performing a type check on object later on.

6 Quarantine

The quarantine pattern aims to 'quarantine' all IO functionality and seperate it from the core program functions to reduce side effects of any kind. Implementing this pattern requires that all IO actions are computation sequences are clearly separated from the pure functions and all IO must be called from the main program [2].

Advantages: The quarantine pattern reduces side effects of asynchronous IO and creates a clear distinction between parts of the program that may halt waiting for IO and parts that can execute freely without IO.

Disadvantages: This pattern often falls short of minimizing IO in programs that rely on IO heavily. It also can negate any performance benefits from asynchronous IO execution where one process can continue writing or reading data while another process executes other functionality.

7 Pattern Matching

Pattern matching is the act of checking a given sequence of tokens against a pattern specifier. A common use case for pattern matching involves using regex expressions to manipulate strings and in dom, we can see this being useful to parse the input from the previous filter to separate inputs into cells that can be put into a table and to determine if column headers represent dependant or independent variables, symbols or numbers.

Advantages: Implementing pattern matching often allows you to eliminate large chunks of procedural code by replacing it with a declarative pattern matching code.

Disadvantages: While pattern matching can reduce code size, it also inhibits code readability because it requires knowledge of how the pattern descriptor is used.

8 Data Spaces

The data spaces abstraction applies to concurrent and distributed systems and requires programming under the following constraints. First, there exists one or more units executing concurrently. There also exists one or more data spaces where concurrent units store and retrieve data. Finally, there is no communication between concurrently executing units other than through the data spaces [2].

Advantages: Using the table in dom as a dataspace, the IO could potentially run concurrently with the rest of the program forcing the program to process rows, columns, and headers as the data becomes available from the input processor.

Disadvantages: Data spaces are not a good fit when concurrent units need to address each other. Before calculating a dom score for each row in the table, dom requires that the entire table has been processed and stored which would require communication between the concurrently running input processor and dom score calculator.

9 Lazy Rivers

The lazy river style aims to make data available in streams that are passed from one function or object to another. The functions or objects in the application act as filters that modify or transform the data stream in some way before handing it off to another. In this style, data is processed from upstream (data sources) down to downstream (data sinks) on a need basis which eliminates problems stemming from too much data at one time [2].

Advantages: A lazy river pattern would make it easy to handle asynchronous IO to and from the filter. In the case of the dom filter, processing all of the input could be line by line while other functions in the program construct the rows of the table as each line is read in rather than trying to process it all at once. Because CoffeeScript inherits Javascripts asynchronous read line behavior, this abstraction could prove to be very useful and could speed up execution time because the main functions don't need to wait for all of the input to be processed.

Disadvantages: Implementing a lazy river pattern requires restructuring the code so that inputs can be processed without knowledge of input that has not yet been read in from the input stream. While this may be possible in the early parts of Dom where the table is constructed row by row, calculating the dom scores for each row requires that all other rows have been processed.

10 The One

The One abstraction acts to solve the problem in a pipe and filter fashion. It contains a list of functions that it needs to run and a value that it should run those functions on. It contains the function to bind functions into its list of functions to run and the function to execute those functions in the order they were bound. We believed that The order in which the dom process was executed could have been boiled down further into a smaller pipe and filter setup where we could have functions run the different parts of the dom function. These functions could be bound to a The One abstraction and run in the order that achieves a Dom behavior.

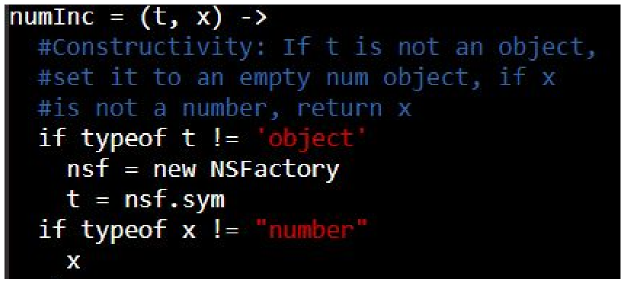
Advantages: By explicitly stating the order in which the function calls are done it gives us much more control over how the data will be handled in the pipe. It also helps to make debugging simpler as we can identify easily which part of the program an error occurred in.

Disadvantages: The style itself does not add many distinguishable properties that make it desirable when compared to similar patterns like the kick forward style [2].

Epilogue

After exploring how the above ten abstractions could be leveraged to implement the Dom filter, we decided to utilize the Constructivist, Factory and Composite patterns. It is worth noting however, that the implementation as a whole reflected knowledge we had gained of other patterns. Many other patterns that we have learned in class or researched outside of it were not applicable at all, due to the nature of Dom and our usage of CoffeeScript.

Constructivist: Constructivist was an excellent pattern for use with CoffeeScript. Since CoffeeScript is a language where parameters and variables are not strongly typed it meant that we were working on some assumptions when we were passing variables between functions. By implementing this pattern into our Dom code we found several bugs that had gone unnoticed and we were thus able to fix them. If you are working with a language where variables are not strongly typed or you don't want errors propagating through the program, then the constructivist pattern is very good at making the code more consistent and making sure that the types of the variables you are passing are the types that you assumed they were. One of several of the instances in which we used it can be seen in this snippet:

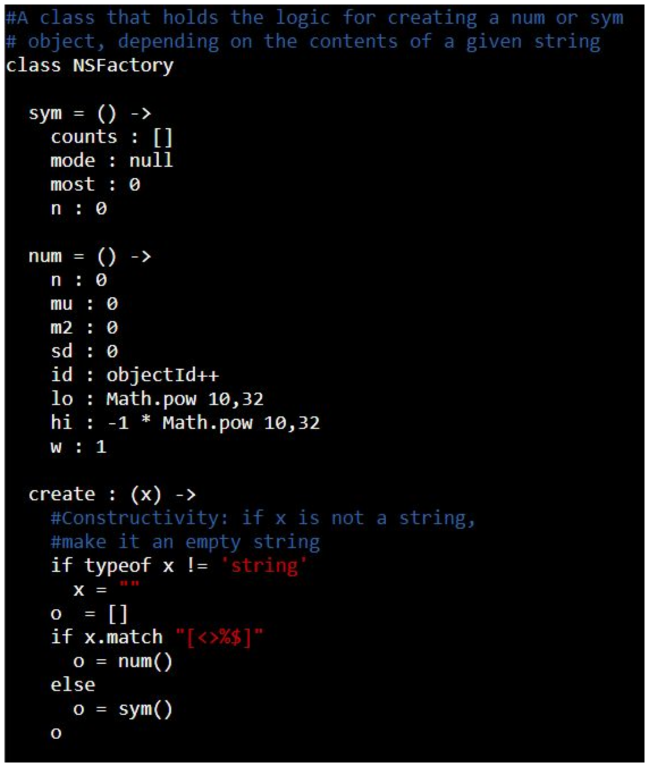


```
numInc = (t, x) ->
  #Constructivity: If t is not an object,
  #set it to an empty num object, if x
  #is not a number, return x
  if typeof t != 'object'
    nsf = new NSFactory
    t = nsf.sym
  if typeof x != "number"
    x
```

.

Composite: The composite pattern proved to be very useful in the construction of the table used in the dom filter. Our implementation included a table that consisted of rows, columns, independent variables, symbols and numbers. The symbols and numbers themselves were composite structures that stored things like counts, standard deviations, mode and other statistical information related to their columns of data. The composite made it easy to construct the table piece by piece and with CoffeeScript's lax type checking, we were able to define a table that could hold an array of an unknown type that we could later fill with cells. We recommend using the composite pattern when dealing with complex data structures whose individual pieces may not be constructed all at the same time and especially in languages like CoffeeScript where adding new components to a composite object is straightforward.

Factory: Factory was semi-useful in this case. It helped to abstract the creation of the num and sym objects away from the header function and thus made that code easier to read. However since there was only one place where objects were getting created in that way it was not incredibly powerful for our program. In the case where there are many more types of objects that can be created it would be very useful to have a factory pattern as it could save a great many conditional statements and type comparisons. It can be seen below how this pattern could be leveraged to encapsulate many more classes, and then let a client or main method, in our case the header function, easily delegate creation to the factory:



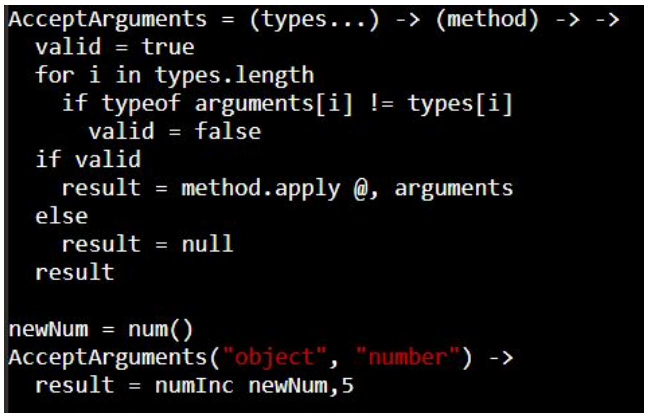
```
#A class that holds the logic for creating a num or sym
# object, depending on the contents of a given string
class NSFactory

  sym = () ->
    counts : []
    mode : null
    most : 0
    n : 0

  num = () ->
    n : 0
    mu : 0
    m2 : 0
    sd : 0
    id : objectId++
    lo : Math.pow 10,32
    hi : -1 * Math.pow 10,32
    w : 1

  create : (x) ->
    #Constructivity: if x is not a string,
    #make it an empty string
    if typeof x != 'string'
      x = ""
    o  = []
    if x.match "[<>%$]"
      o = num()
    else
      o = sym()
    o
```

Decorator: While decorator is a powerful pattern when coding in Python and similar languages, it becomes more clunky and difficult to use when written in CoffeeScript. We spent a large amount of time attempting to implement decorators, so that we could leverage them to create a Declared Intentions type of behavior, but to avail. The syntax that allows for decorator-like behavior in CoffeeScript does not align very cleanly with more typical code, and it even involves a mechanism (apply @, arguments) which is seen in no other uses and seldom demonstrated in documentation or examples. A code snippet of our attempt to leverage the pattern, in order to create an AcceptArguments Decorator can be seen below:



```
AcceptArguments = (types...) -> (method) -> ->
  valid = true
  for i in types.length
    if typeof arguments[i] != types[i]
      valid = false
  if valid
    result = method.apply @, arguments
  else
    result = null
  result

newNum = num()
AcceptArguments("object", "number") ->
  result = numInc newNum,5
```

Declared Intentions: Declared intentions is very similar to the constructivist pattern in that it attempts to manage the types of variables that get passed to functions. What made it difficult to use was that it required a decorator function to be implemented. As stated above the decorator pattern is difficult to implement in CoffeeScript and so the development cost far outweighed the perceived benefit. Declared intentions creates parameter type enforcement for functions by restricting imperfect calls to them. If the call doesn't have the correct parameter types then the decorator does not allow that function to be run.

This abstraction could be very useful with non-strongly typed languages, such as CoffeeScript. However, for our specific filter, it was not a perfect fit since we wanted the functions to be run even if the parameters are of the wrong type. That is why we used the constructivist pattern. In a safety critical program where functions data or states should not be affected if invalid types are encountered, then Declared Intentions would be quite useful. Especially, as we encountered, when I/O or other input streams are piped to an application, data may have unexpected corruption or null values, due to separate modules and components.

Quarantine: The quarantine pattern had advantages that were initially attractive to us as we began to implement dom. When dealing with the asynchronous behavior of CoffeeScript's *readline*, we ended up using some of the methodology behind the quarantine style. We needed to make sure that all of the input had been processed before other parts of the dom program tried to reference them and put them into the table so we made sure that the readline.on 'close' function started the rest of the dom program and subsequently quit the IO process spawned by readline. We recommend using the quarantine style in similar situations where the entirety of a programs input must be read in before it can be processed and when performing synchronous IO using a language with inherently asynchronous IO functions. Ultimately, because quarantine is an expansion upon The One style, we did not end up implementing the quarantine abstraction.

Pattern Matching: Pattern matching is very useful when you want to eliminate large swaths of declarative code with a procedural alternative to make code more succinct. In our case we did not have large amounts of declarative code that could have been rewritten in a procedural manner. As such while pattern matching was present in our program it did not have a large impact on the structure or design of it. We used pattern matching to parse string input to extract the data we needed from it. Past that it was not very useful for us. In a program where you have a huge amount of patterns that are declared or value comparisons that are made it would be useful to replace the declarations with a pattern matching scheme where you simply parse the input and compare for valid patterns within it. Overall a powerful pattern that was not very applicable to our problem.

Data Spaces: The dataspaces style did not end up working well for our implementation of dom because calculating when dom scores for a row, it requires that all other rows have been read in from IO and put into the table. We do not recommend using this style when data read in early relies on other data that may be read in later.

Lazy Rivers: Similar to data spaces, the lazy river style did not end up working in our favor in implementing dom. Because calculating dom scores relies on other rows in the table, it was impossible to structure our whole program to accept IO piece by piece and perform process it along the way because before we start calculating the dom score, we need to wait for all of the rows to be processed and inserted to the table.

The One: The one gives an incredible amount of control over the way a program is executed. However in our case the order in which functions in the program are executed was not imperative to its outcome. As such while it would have been possible to write the program in The One format it was very unnecessary and most likely would have restricted how we could use other abstractions. In a program where you are acting on the data that is fragile and performing actions on it out of order could end up with corrupted or unusable data it would be very useful to implement your program in The One format as you can make sure things happen in the order you want them to and only the order you want them to.

EXPECTED GRADE

For project 2b we implemented the dom filter * (one star) in CoffeeScript *** (three stars), with the following patterns: constructivist, factory, and composite. We expect to receive 12 points for project 2b and possibly 13 depending on what difficulty the constructivist abstraction is deemed to be.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software. Boston: Addison-Wesley.

[2] Lopes, C. V. (2014). Exercises in Programming Style. Boca Raton, FL: Taylor & Francis Group, LLC.