CSC 417 Homework 2

"Absurd Awful Audacious Appalling Antiquated Automated Adventures in Almost all Aspects"

Jeremy Schmidt
North Carolina State University
Raleigh, NC, United States
jpschmid@ncsu.edu

Justin Schwab
North Carolina State University
Raleigh, NC, United States
jgschwab@ncsu.edu

Mandy Shafer
North Carolina State University
Raleigh, NC, United States
anshafer@ncsu.edu

Tony Niverth
North Carolina State University
Raleigh, NC, United States
ahnivert@ncsu.edu

Filter: Dom
Language: Python 3

Abstractions

State Machine

Intent
Simplify program logic by breaking it up into states and rules. Aims to achieve simpler debugging because program actions are deterministic based on the current state.

Structure
Define a *machine* which holds several *states*, defines *transitions* between the states, and maintains the current state. When an action occurs, e.g. a certain type of input is read by a parser, decide what procedure to follow based on the current state of the machine and, if the action matches a transition condition, change the machine state.

Example
For processing rows of input to DOM, we might define a state machine which has a state for each type of input it may be parsing, e.g. reading a row vs. reading the contents of a cell. We would then define transitions such as 'when a new line is read, transition from reading the last cell to reading the next row' or 'when finished processing the contents of cell 3, transition to processing the contents of cell 4'. This would make it clear when certain actions are happening while representing the states of the program in a readable way.

RESTful Interface

Intent

To decouple server and client logic and provide true encapsulation by creating a uniform, stateless interface.

Structure
An interactive interface exists where a client program can make requests to the server program to retrieve or manage resources. The request must include all the information the server needs to fulfill it. This ensures the server does not have to remember any past transactions or information about the client. The "resource" passed back and forth must be something both the server and client understand (such as json or xml data). The interface (http) is standardized so at any time the server or client could be replaced.

Example
We could create a mock "server" program to separate out the work of calculating DOM score from the job of parsing input data. The server would take a request with the "POST" verb along with a list of data rows as

json data. This could be described as a request to create a “dom-scored-data” resource object. The server code would then perform DOM calculations on that batch of data and return the result - again in json format. This standardized interface would provide us with the flexibility to modify or replace the client or server without affecting the other. This encapsulation would also allow us to separately apply additional different abstractions to the server and client programs.

Compartmental Model

Intent
Separate logic from program, making the system more understandable from a high level; utilize a paradigm of modelling data in the system as flows between modules (or compartments)

Structure
A container object that holds the other system state at the current time mark, as well as another for holding the state at the next time mark. The state is updated with each time mark, and contains stocks (pools of data to be used/consumed), flows (a description of how data moves from one stock to another), and perhaps auxiliary constants that affect how flows operate.

Example
We could use a compartmental model to process the rows of the table, where each row is an object to be consumed from a stock. The processing of that row into a dom score for that row may be represented as a flow. We may be able to go deeper and have, for example, two rows be stocks that flow into each other that determine binary domination as an outflow.

Facade

Intent
To create a simple interface for a more complicated system, making it faster, easier, and more efficient to use.

Structure
The facade pattern typically specifies simple functions, which when called, deal with the more complicated system under the hood, abstracting away the need for the caller to use the complicated system in any direct way.

Example
This pattern is essentially just the concept of general abstraction and interfacing, so we could create an interface for dealing with the components of the input/output table. For example, creating functions that deal with file operations related to reading and writing the table could be an effective use of this pattern. Also, creating a facade for determining if a row dominates another would be a useful interface to easily call, as the process of determining domination is non-trivial.

Factory

Intent
The factory pattern is intended to simplify the construction of different objects but of similar types. This makes the creation of related objects easier because one does not have to specify the entire schema for the new object, just the type that they want.

Structure
This creational-style pattern involves a function that can be called to return a new object, the type of which is specified by the parameter sent to the factory function. Vehicle objects, for example, could all be constructed in the same factory, but the parameters sent to the factory would determine the type of vehicle returned.

Example
We could use the Factory with the Iterator to create different types of iterators. The type of input file could be specified as a parameter to the factory function and then return an iterator for that input type. For example, you would pass in the parameter ‘csv’ to the factory, and it would give you an iterator for a CSV file. You could also, perhaps, use it to create an iterator for a tsv (tab-separated values) file.

Iterator

Intent
Allow a caller to access elements in an iterable object, usually sequentially, without needing to understand or expose the object's actual representation.

Structure
Similar to a facade, an iterator gives an interface for iterating through elements of an object. A popular paradigm is the getNext() method, which accesses the next element in the object (whatever that object is, specifically), and advances the iterator forward one element such that when getNext() is called again, the element after the previously returned element is returned.

Example
We could use this for parsing out each field of the csv input, where getting the 'next' element is getting the field after the next comma, and keeping track of where the iterator is in the row of data so that it sequentially accesses the fields. There may be some special return value that indicates all elements in the row of data have been consumed. Traditionally, this would be an EOF.

Spreadsheet

Intent
Spreadsheets are used to update data in real time. Instead of recalculating data for each value in an array, dependent values will update whenever a change is made to the matrix.

Structure
A Spreadsheet is built by a matrix of lambda bodies columns of data and formulas. When data changes in one cell of a matrix, the dependent functions also update to change the values in other cells.

Example
Spreadsheets could be extra useful for this project. As we continue to calculate the Dom of each new row of data, the spreadsheet and recalculate to determine its rank amongst the other rows.

Pipe and Filter

Intent
The purpose of this architecture is to simplify the flow of your program as the data undergoes multiple transformations. It also allows many programmers to work and test on seperate functionalities at the same time and combine the filters to complete the pipe.

Structure
The structure is made of both filters and pipes. The Filters will alter the data in any chosen way, and then the pipe will carry that data to a new filter. A pipe can be any length and will continue to move data along the filters until it reaches the end.

Example
We could separate the functionality of dom into computing the dom score, and appending a column to the original table with the dom score. We can funneling the data and the dom scores into a second filter where it will append the data in a format appropriate for bestrest.

Layers

Intent
Separate logic within the program to allow for different parts to function without the others. The stacked groups allow parts of the program to function without seeing how the other parts function.

Structure
The different layers would typically not be able to see the processes from the other layers. Data can be passed back and forth between the layers, but shared processes don't really exist. In some instances, a layer can only interact with the layer directly above or below it, whereas others allow data to be shared across layers with multiple layers between them.

Example
We could use layers to separate evaluating the different rows. The ranking of one row doesn't necessarily affect the ranking of another, so we could separate there, or have different layers for the different rows we are comparing the main row against.

Blackboards

Intent
Allow multiple agents to access information as it is available. Lower levels trigger actions in higher levels. Allows for separate processes for different parts of the program.

Structure
Each agent would watch the blackboard for updates. When certain data is updated, the respective agent would be triggered to act on the update. The lowest levels would just start their own processes. As these finish up, the higher level agents would be triggered to use the updated data.

Example
We could use a blackboard for the rankings. The lower level would execute the ranking of the different rows of data. The higher level would then add the ranking in a new column in the data.

LetterBox

Intent
Break the program down into different segments for execution. This could be for concurrency, saving time with multiple CPUs running the different capsules for the job.

Structure
Each part of the abstraction is a capsule of data to be evaluated. Little packages of data become jobs that combine to make the big job overall. One job finishing could lead to a message being sent to another job. Eventually, all parts of the program are executed.

Example
We could use this one to break up the rows. Each capsule could be a different row to be analyzed with its random rows to be compared to. After each row is evaluated for a ranking, the rankings are added to the data.

Epilogue

Iterator

The python iterator class we wrote is largely unnecessary because the standard csv file operations libraries in python are more than enough to easily handle csv-like files at a reasonably high level, with more stability and correctness than our implementation. Despite this, it still makes working with the input file relatively easy, which is indeed the purpose of an iterator object.

I would generally recommend that one use existing libraries for trivial things like reading csv input, so our implementation is largely redundant and therefore not generally recommended.

Furthermore, this kind of approach may actually be useful in parsing unusual or non-standard types of input files that don't already have a (probably highly optimized) library for dealing with that file type.

It was useful practice (not really "learning" per se) in text processing in python, however. It could be a useful learning exercise for new computer science students.

Factory

We used factory, a creational pattern, to create an iterator for the text input to DOM. You can give the factory a couple different types of input file (csv and tsv), and get an iterator for parsing input of that type. We only use this for getting a CSV iterator, though. This wasn't the most useful abstraction to add, but it would indeed be more useful if our filter required multiple different types of file input to be supported. We could then easily create an iterator for each input type we needed to support. I would really only recommend using this pattern if you indeed had multiple types of input. Otherwise, it just seems a bit unnecessary.

This pattern was, however, helpful for code organization because it adds another layer of abstraction onto our iterator class, making it more extensible and easier to use. It also gave me a bit of experience with python classes, something I haven't really used until now.

RESTful Interface

In the case of this project, applying a RESTful architecture was wildly inefficient. Because we used a mock-server rather than a fully decoupled client and server, the benefit gained from separating the components was not very helpful. Additionally, the way we setup the DOM calculation with RESTful calls required a very high volume of requests that would suffer significant slowdown on real network. I would not recommend a RESTful architecture in a scenario such as this one where many calls to the server are used to achieve a relatively simple calculation.

One area where the RESTful server-client separation was helpful was in team collaboration. Having server and client code relatively de-coupled made it easier for our team of four to coordinate on implementation.

An instance where the RESTful design would be more valuable would be if the computation was expensive enough to justify the network overhead and if we wanted to offload that work to an actual server. As it is now, the simple DOM score calculation did not justify this heavyweight abstraction.

Bonus

Filter:
DOM - No bonus

Language:
Python - No bonus

Abstractions:
REST interface - 2 bonus marks
Iterator - No bonus
Factory - No bonus

Total:
Max of 2 bonus points from abstractions