CSC 417 Homework 2b

"Absurd Awful Audacious Appalling Antiquated Automated Adventures in Almost all Aspects, round 2"

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: Monte_Carlo
Language: CoffeeScript

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.

Rules of Thumb
- Good for modeling high-level logic
- Enables formal analysis
- Poor for very large systems

Example
We might choose to code a state for each possible parameter being randomized as part of the monte-carlo simulation. Each state would then define their own transition with a side-effect that generates its random value between the min and max for that parameter. The state machine would then progress through the states, each state would perform its random number calculation, then transition to the next state. For example, when the 'randomize' transition is called from the " pomposity" state, the program would calculate its value between 0 and 1 and then transition to the "learning curve" state which has its own behavior defined for generating its value. This would provide a means to distinguish the different behaviors for each parameter by explicitly defining the behavior (i.e. limits) for the state.

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.

Rules of Thumb
- Adds overhead of data transmission
- Good for separation of components
- Won't work if an operation requires information about past requests

Example
A simple way to make use of a restful interface would be to separate out our random number generator into the server and make a call to the interface whenever we needed a random number. The client could make a GET call to /random to get a random number between 0 and 1. To get a random number with a seed, the seed could be placed in the path variable. This would relieve the burden of random

number generation from the client and allow for a flexible and interchangeable implementation of the pseudo-random algorithm on the server side.

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.

Rules of Thumb
- Break up parts of functionality
- High level understanding

Example
We could use a compartmental model to build a random number generator needed for the monte carlo filter. In order to make the generator as random as possible, we could define a number of different flows that modify data in typical pseudorandom number generators such as multiplying a value by a large number and then calculating the modulus of it with the value that you want to be the upper limit of the pseudorandom output value. We could even use the initial state of the model effectively as a seed, to make testing of our model easier.

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.

Rules of Thumb
- Hides complex implementation behind simple function calls
- Broad abstraction

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.

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.

Rules of Thumb
- Creates overall object without specifying every part
- Interface hides smaller parts of object

Example
We could use the Factory pattern to create a random number generator for each column of the pipeline data, where you'd pass in the minimum and maximum values for the random number generator to the factory function and it would return a generator for that column. You could hypothetically then run the random number generators in parallel to speed up execution of the generation of pseudorandom numbers. This could be quite helpful on large inputs.

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.

Rules of Thumb

- Steps through group of elements
- Hides layout of group

Example

We could make a very basic iterator for stepping through each column of our input and processing the required numeric range for each of our variables. In conjunction with the above example for using a factory, each call to this iterator's next() method would then be passed to the factory to create a pseudorandom number generator for that particular range of numeric values.

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.

Rules of Thumb

- Update data across system

Example

Spreadsheets would be a useful abstraction for many of the future filters in this pipe, however for monte carlo, there is not much benefit. For this filter, it could potentially could be used to spread the same random generating formula across each attribute in the filter.

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.

Rules of Thumb

- Keep stages simple
- Good for a series of sequential operations
- Not good if stages depend on each other

Example

We might be able to construct a pseudo-random number generator using a pipe and filter strategy. Each filter would enact some mathematical operation on the number as it flows through the pipe until something pseudo-random is produced at the output. For example, we could apply the Von-Neumman "Middle Square" method by having a filter stage that squares the seed number that pipes to another which bitshifts it. Having these stages as separate components would make it easy to increase the complexity of the mathematical jumbling function simply by adding more stages.

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.

Rules of Thumb

- Each layer only knows its functionality
- Data can be shared without functionality being shared

Example

In general, layers could be used to split up work that doesn't really have overlapping code. The values can be passed between layers, but the code is separate. For Monte_Carlo, we could have one layer that generates the inputs and another that does the deterministic computation. The inputs would be the data passed between layers, while the two layers do not need to know how the other layer works. It should work, but may not be super helpful.

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.

Rules of Thumb

- Agents have access to board of data
- Lower level agents notify upper level agents of updates to data

Example
In general, blackboards are used when the higher levels of the program are waiting on lower levels. We could have lower levels generating the inputs and posting them to a “blackboard”. Then the upper layer(s) would be notified about the update so they could do the deterministic computation.

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.

Rules of Thumb

- Packages of data that are evaluated

Example
We could use this one to break up the deterministic computation. Each capsule could be a different computation. The computations could then be done concurrently, making the filter run faster.

Singleton

Intent
Enforce that only one instance of a class is allowed. There is a single instance over the whole program and the class cannot be re-initialized.

Structure
Once the class is defined, it is encased in a way that only allows the class to be instantiated once. Regardless of the number of times it is called, the class only has one instance.

Rules of Thumb

- Only one instance of the object
- Control access to data by one instance

Example
In general, a singleton pattern would be used when you only need one instance of the class. It is a class that controls enough of the program to where you only need one instance. For this project, we can use a singleton pattern for our random number generator. It doesn’t matter how many times we call from the generator, there is only one.

Epilogue

State machine

The state machine abstraction worked for reading in the config file. Each state was a different element of the columns for the data in the pipeline. By keeping track of the state of reading in the config file, the program could maintain what data it had. This abstraction is not necessary for this task, but it is a nice way to lay out the config file reader, both for the program and for the programmer. SInce each column of data for monte carlo has the same elements of name, max, and min, it is easily understood as different states for a state machine.

While the implementation of this state machine is not necessary for monte carlo, it is a nice addition. We would recommend using this in the future since the layout of the config file lends itself to a state machine. Reading in files with complex data layouts can be more easily understood within a state machine than on their own.

In general, state machines are good for any time you want to track the status of a process. As the status updates, the state instance updates, letting the rest of the program know where the process is. This can be helpful in tracking tickets in a system or keeping track of input to something like a vending machine.

Pipe and Filter

We were able to use a pipe and filter to generate random numbers. Each part of the pipe added another layer to the random number. This way, we could change out different possibilities for random number generation without having to change the entire process.

We would recommend using this abstraction. It helps with generating random numbers since the process of generating random numbers has different parts which are fairly distinct from each other. Each part adds to what the previous one did for the random number generation, so it is perfect for a pipe and filter. This abstraction is not absolutely necessary,

but it is a great way to implement a random number generator.

In general, pipe and filter abstractions are good for any system that progressively updates data. Whenever the data is passed on to the next step, something has changed, and each stage updates a successive part of the output. Any time you need an “assembly line” in the code, a pipe and filter abstraction will do the job.

Singleton

The Singleton was one of the most helpful abstractions in our project. We created a singleton class for the generator, with just one get method and one generate method. This allowed the single instance of the random number generator to effectively maintain a global variable for the random number seed. This way, even if the random number generator was requested multiple times, the same running seed would be used. This abstraction was not strictly necessary as the same parameters could just be passed in directly or kept global, but it did provide a neat way to enforce the singular-ness of the random generator state.

Although our abstraction of singleton was helpful yet unnecessary, we would still recommend using this abstraction in more applicable contexts. It is a great structure to ensure only one of that type is created with data that will remain consistent throughout the implementation.

In general, singleton abstractions are good when only one instance is needed. This one instance maintains control of the object for the system, keeping access to certain parts of the data to itself. This can be helpful for manager classes or times when only one part of the program needs access to some part of the data at a time.

Bonus

Filter:
Monte_Carlo - no bonus

Language:
CoffeeScript - 2 bonus

Abstractions:
Singleton - no bonus
State machine - 1 bonus
Pipe and Filter - 1 bonus

Total:
4 bonus - 2 from language and 2 from abstractions