2C Extra Credit

Bryan Arrington
bjarring@ncsu.edu
North Carolina State University
Raleigh, 27695, United States

ACM Reference Format:
Bryan Arrington. 2019. 2C Extra Credit. In *Proceedings of* . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SELECTIONS

Selection of Pipeline `monte carlo`
Selection of Language Kotlin

1 POSSIBLE ABSTRACTIONS

1.1 Singleton

1.1.1 Overview. The singleton pattern is a creational design pattern that maintains a single instance of an object and ensures only one instance is ever created. All further references to objects of the singleton class refer to the same underlying instance.

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

1.1.2 Advantages.

- Easy to controls access to a sole instance.
- Maintains a single instance of a generator.
- Reduces the clutter in the global name space.

1.1.3 Disadvantages.

- Would get in the way of expanding functionality.
- Can cause bad side effects when used in a large system.

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 attributes into stages, where each attribute and range is constructed and added to the list of attributes. Additionally, each attribute could be constructed in stages, where min and max of each attribute 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 attribute builder could be used to filter which rows are returned. A attribute builder could be given some kind of filter to know to ignore certain indices when returning the attributes 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 attributes and the ranges.
- 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 monte_carlo is fairly simple, pattern matching can easily be implemented. Useful for: When we need to parse the command line arguments for the number of iterations to run, the seed of the random number generator, and the verbose flag.

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 `^\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 Decorator

1.4.1 Overview.

The decorator pattern is a structural design pattern that is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This pattern provides an alternative to using inheritance to modify behavior.

With Kotlin you can easily extend the functionality of any class by adding methods. This could be useful for writing monte_carlo when performing the rounding of each statistic's calculation.

1.4.2 Advantages.

- It is more flexible than inheritance since inheritance adds responsibility at compile time, while decorator adds at run time.
- Can have any number of decorators and in any order.
- Extends functionality without adding another object.

1.4.3 *Disadvantages.*

- Can make code hard to maintain since this creates similar classes which are hard to maintain and distinguish.

1.5 Strategy

1.5.1 *Overview.*

The strategy pattern is a behavioral design pattern that is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

This could be used in monte_carlo to create multiple algorithms that are chosen based on different command line arguments. This would allow for different types of data to be generated or different min's and max's to be used when generating each attributes' value.

1.5.2 *Advantages.*

- Allows families of related algorithms and with inheritance can factor out common functionality of the algorithms.
- Can eliminate conditional statements.
- Can provide different implementations of the same behavior where the client can choose among strategies with different trade-offs.

1.5.3 *Disadvantages.*

- Client must know about the different strategies in order to chose the correct one.
- Increased communication overhead between the strategy and the context.
- Increases the number of objects in the application.

1.6 Visitor

1.6.1 *Overview.*

The visitor pattern is a behavioral design pattern that is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

1.6.2 *Advantages.*

- Makes adding new operations easy. Visitors can add operations over an object structure by simply adding a new visitor.
- Can visit across class hierarchies, even with objects that don't have a common parent class.
- Visitors can accumulate state, so that state does not have to be passed around as augments or global variables.

1.6.3 *Disadvantages.*

- Can break encapsulation since visitors may make an element's internal state public.

1.7 Adapter

1.7.1 *Overview.*

The adapter pattern is a structural design pattern that is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

1.7.2 *Advantages.*

- Allows classes to be incorporated into existing systems that might expect different interfaces to the class.
- Low cost for the benefits that are received:
- Can provide encapsulated reuse of existing behavior.
- Can provide polymorphism with a foreign class.
- Promotes the open-closed principle.

1.7.3 *Disadvantages.*

- Object adapters make it harder to override adaptee behavior.
- A class adapter won't work when we want to adapt a class and all of it's subclasses.

1.8 Command

1.8.1 *Overview.*

The command pattern is a behavioral design pattern that is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.

1.8.2 *Advantages.*

- Decouples the object that invokes the operation from the one that knows how to perform it.
- Commands can be manipulated and extended like any other object.
- Easy to add new commands, since you don't have to change existing classes.

1.8.3 *Disadvantages.*

- Can be hard to maintain with the high number of classes and objects.

1.9 Composite

1.9.1 *Overview.*

The composite pattern is a structural design pattern that is used to combine one or more objects so that they can be manipulated as one object.

1.9.2 *Advantages.*

- Allows primitive objects to be composed into more complex objects, which can be composed, and so on recursively.
- Makes clients simple since they can treat composite structures and individual objects uniformly.
- Makes it easier to add new kinds of components. New parts work automatically.

1.9.3 *Disadvantages.*

- Can make the design overly general and harder to restrict components of a composite.

1.10 Facade

1.10.1 *Overview.*

The facade pattern is a structural design pattern that is used to define a simplified interface to a more complex subsystem.

1.10.2 *Advantages.*

- Shields clients from subsystem components, reducing the number of objects that clients deal with.
- Reduces compilation dependencies between components by promoting weak coupling.

1.10.3 *Disadvantages.*

- Promoting weak coupling can be a disadvantage if your system needs strong coupling.

2 EPILOGUE

2.1 Overview of Rewrite

I chose to rewrite monte_carlo for project 2c. This program reads command line arguments for the number of iterations, or rows to generate, the seed for the random number generator, and a verbose boolean flag. With these arguments the program generates data for a series of attributes with hard coded ranges. This data is outputted in rows to standard out, which is then piped to the next part of the pipeline. I chose to write this in Kotlin after we discussed the Kotlin language in lecture. The abstraction I chose to use were: singleton, builder, and pattern matching.

Execution of my program starts by parsing the command line arguments to determine the number of iteration to run, the seed of the random number generator, and the verbose flag. After this the array of attribute names is created and passed to the constructor of the monte_carlo singleton.

The monte_carlo singleton uses the AttributeBuilder to create the ranges for each attribute when it is constructed. Then the generator method of the singleton is then called in the main method and is passed the read in command line arguments. The generator method runs a for loop for the number of iterations specified and uses a random number generator with the given seed to produce a value for each attribute built by the AttributeBuilder. Each iteration is then printed out to standard output so it can be piped to the next part of the pipeline.

2.2 Structure of Code

2.2.1 *AttributeBuilder.*

The attribute builder is a class defined to take an array of string attributes attributes and create a map of maps where each attribute has a min and max defined range.

2.2.2 *MonteCarlo.*

This is the class that contains the functionality of the code with the generator method. The constructor of this object uses the AttributeBuilder to create the attributes and their ranges. There is only one instance of this class, thus it is the singleton.

2.2.3 *Double.round.*

This is the decorator in my code. I am extending the double class using Kotlin's ability to extend the functionality of any class with ease. This round method rounds a double to specified number of decimal places which is passed in as a parameter.

3 END

Since my program works, and has been finished on time, my base grade (before modifiers) is 10 marks. My choice to replace monte_carlo didn't yield me any extra marks, but my choice of Kotlin earns me a bonus mark. This means I have earned 10 + 1 = 11 marks for this assignment.