Project 2-A Document

Minglun Zhang
Computer Science
NCSU
Raleigh NC US
mzhang17@ncsu.edu

Jimmy Nguyen
Computer Science
NCSU
Raleigh NC US
jnguyen6@ncsu.edu

Daniel Rabstejnek
Computer Science
NCSU
Raleigh NC US
djrabste@ncsu.edu

Connor J. Parke
Computer Science
NCSU
Raleigh NC US
cjparke@ncsu.edu

ABSTRACT

This report presents the top ten possible abstractions for CSC 417, Group J, Project 2A with descriptions and examples. The report then will present an epilogue that describes the group's experience with the top three abstractions we chose and the analysis for advantages and disadvantages for adopting these abstractions. Our top three abstractions to use are the following: Command, Iterator, and Factory Method. In the end, the paper will declare the expected maximum grades for the project. The filter used for this project is 'dom' file. The language to be coded for the project is Python.

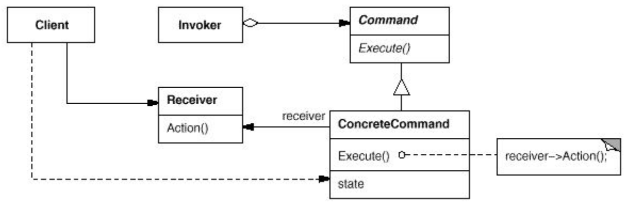
KEYWORDS

design, abstraction, pattern, command, iterator, factory, delegation, decorator, composite, state machines, strategy, macros, template method, python, dom

TOP TEN ABSTRACTIONS

1. Command Pattern

The command pattern is a behavioral design pattern in which an object is used to encapsulate all information (method names and the object that owns the method and values for the method parameters) needed to perform an action or trigger an event at a later time.

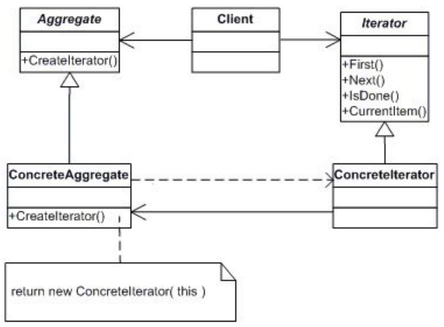


Five participants are involved in command pattern structure: Command, ConcreteCommand, Client, Invoker and Receiver. The Command declares an interface for executing and operation. The ConcreteCommand defines a binding between a Receiver object and an action, and it implements Execute by invoking the corresponding operation(s) on Receiver. The Client creates a ConcreteCommand object and sets its receiver. The Invoker asks the command to carry out the request. The Receiver knows how to perform the operations associated with carrying out a request. And any class may serve as a Receiver.

The command pattern would be useful for encapsulating the full calculation process for finding the dom score. The command pattern would be useful here because when the program is being executed, there is no need to know about what the command to run the dom program is, what the dom program needs, or what the dom program does specifically. By using the command pattern for dom, the content and execution process is encapsulated, which allows the execution of each action in the dom file to be independent from other filter programs.

2. Iterator Pattern

The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data



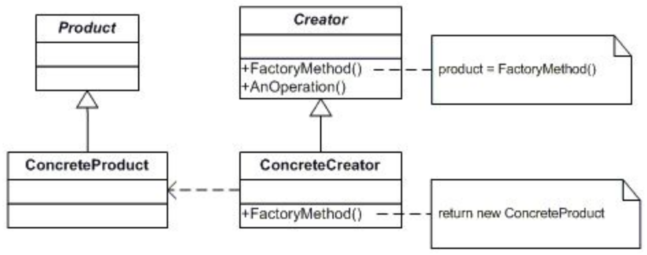
structures available in every language. Each collection must provide an iterator that lets it iterate through its objects.

Four participants are in the iterator pattern, ConcreteIterator, Aggregate and ConcreteAggregate. The Iterator defines an interface for accessing and traversing elements. The ConcreteIterator implements the Iterator interface and keeps track of the current position in the traversal of the aggregate. The Aggregate defines an interface for creating an Iterator object. The ConcreteAggregate implements the Iterator creation interface to return an instance of the proper ConcreteIterator. This structure provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The iterator pattern is a useful abstraction for iterating through each column and row from the given table of data values and then printing the values from each corresponding column and row to standard output. Without an iterator, we wouldn't be able to look through each row in the table and extract the values needed to print the table back to standard output after the dom score calculation process.

3. Factory Method Pattern

The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library in a way such that it doesn't have a tight coupling with the class hierarchy of the library.



There are four classes and objects participating in this pattern: Product, ConcreteProduct, Creator, ConcreteCreator. The Product defines the interface of objects the factory method creates. The ConcreteProduct implements the Product interface. The Creator declares the factory method, which returns an object of type Product. The Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object. The ConcreteCreator overrides the factory method to return an instance of a ConcreteProduct.

The factory method pattern would be useful for distinguishing between the high values (>) and low values (<) whenever we have extracted a certain value from a row. The factory method abstraction can be used to create two new objects: one to indicate that the column contains a high value (>) and another object to indicate that the column contains a low value (<). For the dom file, we can assign one object with a value of 1 to indicate a high value if the difference in two column values is positive (>) or -1 to indicate a low value if the difference in two column values is negative (<).

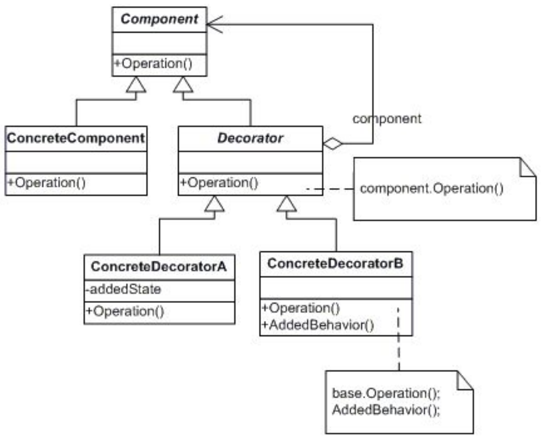
4. Delegation Pattern

The delegation pattern is an object-oriented design pattern that allows object composition to achieve the same code reuse as inheritance. Delegation is like inheritance done manually through object composition. In delegation, an object handles a request by delegating to a second object (the delegate). The delegate is a helper object, but with the original context.

The delegation pattern would be a useful pattern for separating the different sets of functionality in a program, specifically dom.lua. For example, this pattern can be used to separate the process of calculating the dom score using two rows from the given table. One delegated task would be to compute the dom score for one row and a randomly selected row, or the other delegated task would be to compute the dom score for one row and another row that is adjacent to the previous row. The doms function would still calculate the dom score, but the responsibility of calculating the score given two rows would be delegated to an object that can handle the calculation, but for different circumstances.

5. Decorator Pattern

The decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

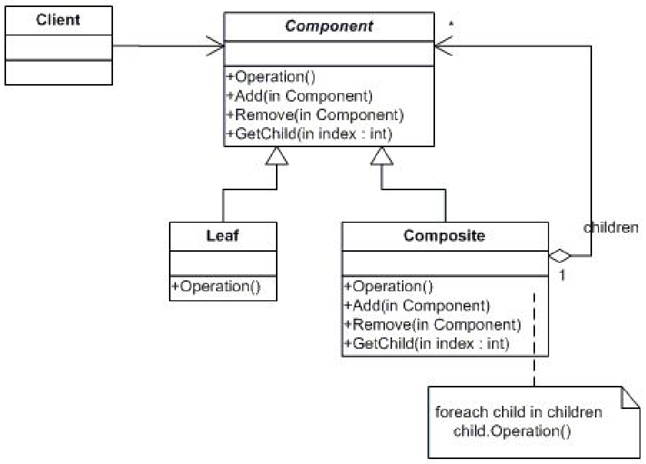


Four classes and objects participating in this pattern: Component, ConcreteComponent, Decorator and ConcreteDecorator. The Component defines the interface for objects that can have responsibilities added to them dynamically. The ConcreteComponent defines an object to which additional responsibilities can be attached. The Decorator maintains a reference to a Component object and defines an interface that conforms to Component's interface. The ConcreteDecorator adds responsibilities to the component. Decorator design pattern provides a flexible alternative to subclassing for extending functionality.

The decorator pattern can be useful for defining the attributes of the table, where the table can be composed of the row and the column, which would act as the concrete decorator. The functionality for the two attributes would be different from each other since they contain different values that would be used for different parts of the dom program. For example, the column can be used to distinguish between what the current header is, and the row can be used to extract the data values to be used for finding the dom score.

6. Composite Pattern

The composite pattern is a partitioning design pattern. The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object. It represents whole-part relationships of objects in a system as Tree structure.



Four classes and objects participating in this pattern: Component, Leaf, Composite and Client. The Component declares the interface for objects in the composition. It implements default behavior for the interface common to all classes. It also declares an interface for accessing and managing its child components. The Leaf represents leaf objects in the composition and it defines behavior for primitive objects in the composition. The Composite defines behavior for components having children. It stores child components and implements child-related operations in the Component interface. The Client manipulates objects in the composition through the Component interface. Overall, the Composite design pattern lets clients treat individual objects and compositions of objects uniformly.

An example of where the composite pattern can be useful would be the table and rows The table can represent a group of rows, but is still considered a single instance of the same type. This can be useful when we start reading rows from each table, but would like to consider groups of rows or even the table as one instance of the same type.

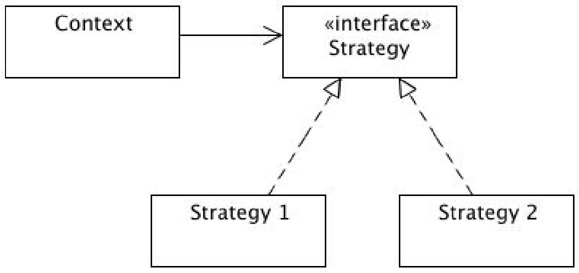
7. State Machines Pattern

The State Machines Pattern allows an object to alter its behavior when its internal state changes. Automata could be constructed from independent state classes. The classes designed with State Machine pattern are more reusable than ones designed with State pattern. An intent of State Machine is the same as an intent of State: to make it possible for an object to alter its behavior when its internal state changes.

For the dom file, the state machine design pattern would be useful for indicating the different states that the program can be in and the different transitions to take. For instance, the initial state would be the initialization state, such as finding the number of samples to use, finding the number of columns for the dom score, and creating a new column called "dom" to store the dom values.. Once the initialization has been completed, the machine can then transition to the reading state, where the rows in the given table. Once the rows have been read, the machine can move to the calculation state, where the dom score for the two given rows can be calculated. The machine will then continuously return to the reading state if there are more rows to read and then the calculation state to find the dom score. If there are no more rows left, then the machine transitions to the end state, where all the rows in the table are printed to standard output. The rows will contain the update dom column, which will show the list of dom scores calculated.

8. Strategy Pattern

The strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives



run-time instructions as to which in a family of algorithms to use.

The strategy pattern shows how to implement different behaviors in a separate class hierarchy with an interface at its root. The goal is to decouple the objects which use your encapsulated Context and the Strategy so that they can vary independently.

The strategy pattern would be useful for abstracting the dom score algorithm where the values from the first given row and the second given row are being used to find the dom score. For instance, on lines 13 and 14 of dom.lua, calculating the average can be abstracted using the strategy pattern, where the algorithm will remain the same throughout, but the results will vary depending on the parameter values given.

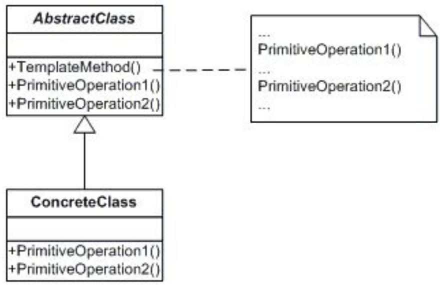
9. Macros

A macro is a program called at runtime to write other programs. It is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output sequence according to a defined procedure.

Macros can be used to define exactly how the dom algorithm should run. For example, the dom function can be represented as a macro itself, where the procedure for calculating the dom score would be defined.

10. Template Method Pattern

The template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. The overall structure and sequence of the algorithm is preserved by the



parent class.

Two classes are participating in this pattern: AbstractClass and ConcreteClass. The AbstractClass declares abstract primitive operations that concrete subclasses define to implement steps of an algorithm. It also implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects. The ConcreteClass implements the primitive operations to carry out subclass-specific steps of the algorithm

The template method pattern is a useful abstraction for creating a defined way to execute methods involved in the dom score algorithm. For example, there can be an abstract class with either the primitive or defined operations. First, find the number of samples and column number for dom. Then, iterate through the rows of the table over the sample size, using each along with a randomly selected other row as parameter values for the dom score calculating function. Then, calculate the dom score for the two given rows and return that dom score. Then, repeat the process of reading the rows and calculating the dom score until there are no more rows to read. Finally, send all the rows out to standard output. Those methods can be abstracted and used in that particular order in another class.

EXPERIENCE & RECOMMENDATIONS

The three abstractions that we have chosen to implement for Project 2A are the following: Command, Iterator, and Factory Method.

The command pattern was a successful implementation, as it encapsulated the execution process of the dom score algorithm. The command pattern was also a simple implementation for the dom file. For instance, the DomCommand class is created to represent the dom command object, which contains various methods for calculating the dom score. The most important method for this particular pattern was the execute() method, which executes the dom algorithm using the row values from the given table. Since the command pattern helps encapsulate the dom algorithm by binding actions related to the algorithm that would be later invoked whenever the command is executed, we would recommend using this pattern for dom and in general for any area in the code where the executor of a specific command does not need to know what the command is, what it needs, or what it does.

The iterator pattern was another successful implementation for the dom file. The iterator pattern was used in the csvIterator function, which is a generator function that creates the table to print to standard output row by row. We intended on implementing the iterator pattern for stepping through each column and each row in the table, since the rows and columns are considered attributes of the table that are iterable. We would recommend using the iterator if there is a need to walk through each element in a collection. In this case, an iterator for walking through each row and column in the table would be appropriate since we would want to access the header name in each column and then access each data value in each row to print to standard output in csv format. Although the iterator is useful for walking through each element, the major downside to the iterator is that you would not be able to backtrack whenever you are processing data from a collection.

The factory method pattern is also another successful implementation for the dom file. The factory method is used to create the command object based on the given string name. Although the implementation of this pattern is quite simple, we think it is still a convenient and important pattern to consider since the pattern allows objects to be created without having to know what object is needed to be created or how the object is created. In this case, there is no need to know what the dom command object is or how it is created. All that matters is that the dom command object can be created. For the current implementation of the CommandFactory class, only the dom command object is created if the name "dom" is given, but no command objects will be created for anything other than "dom". We would recommend using the factory method pattern if you have a class that cannot or should not have to know which objects to create beforehand or when you would like to abstract the creation of the object away from the actual implementation of the object. The downside of using this particular pattern is that there could be some potential coupling. For instance, dom command can be tightly coupled to the command object.

Overall, the use of the three abstractions for the dom file was good. When running the dom file with those abstractions, the program works as expected.

EXPECTED GRADES

We expect to earn the full 10 marks for this project since the documentation provides all the details and examples for the abstractions we thought about using, details on the group's experience and recommendations on what abstractions to use, and a working program that was written in Python and contains three abstractions.

REFERENCES

[1]: https://www.cs.mcgill.ca/~hv/classes/CS400/01.hchen/doc/command/command.html
[2]: https://en.wikipedia.org/wiki/Command_pattern
[3]: https://www.geeksforgeeks.org/iterator-pattern/
[4]: https://www.dofactory.com/net/iterator-design-pattern
[5]:http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/delegation.html
[6]: Online lecture sources