Project 2-B 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 2B 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: Singleton pattern, Template Pattern, and Strategy Pattern. In the end, the paper will declare the expected maximum grades for the project. The filter used for this project is 'monte_carlo' file. The language to be coded for the project is Java.

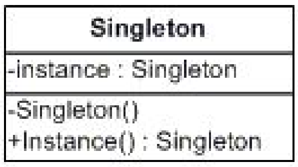
KEYWORDS

design, abstraction, pattern, command, iterator, factory, decorator, composite, state machines, strategy, macros, template, abstract factory, java, monte carlo

TOP TEN ABSTRACTIONS

1. Singleton Pattern

The singleton pattern is a software design pattern that restricts instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without the need to instantiate the object of the class.



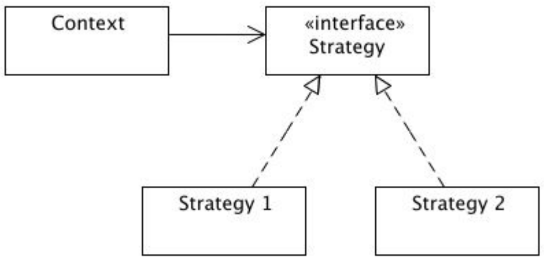
The Singleton class defines an Instance operation that lets clients access its unique instance. Instance is a class operation and it is responsible for creating and maintaining its own unique instance.

Singleton pattern is useful when we only want one instance of an object. For example, if we want a logging class. A Singleton can be used because a logging class usually needs to be used over and over again, and it does not affect the execution of the code.

2. 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.

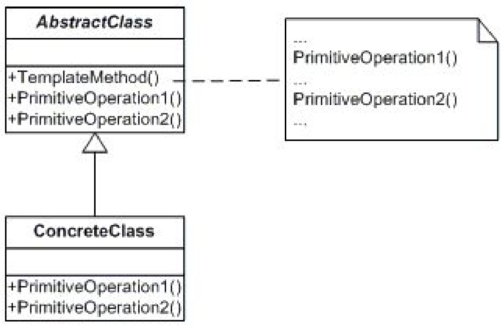
The strategy pattern shows how to implement different behaviors in a separate class hierarchy with an interface at its root. The goal is to decouple the objects which use your encapsulated Context and the Strategy so that they can vary independently.



The Strategy pattern is to be used where we want to choose the algorithm to use at runtime. Good examples of the Strategy pattern would include saving files in different formats, running various sorting algorithms, or file compression as these might use various algorithms depending on their contexts.

3. 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 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 these operations as well as operations defined in AbstractClass or those of other objects. The ConcreteClass implements the operations to carry out subclass-specific steps of the algorithm

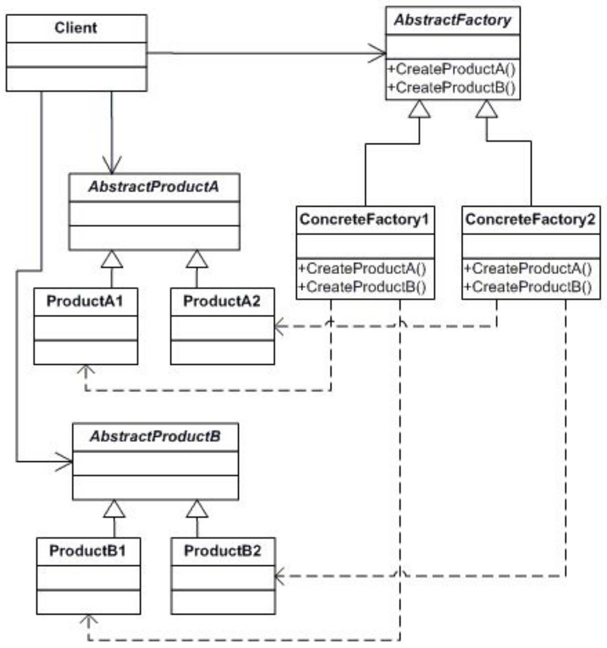
The template method pattern is a useful abstraction for creating a defined way to execute methods, especially for the monte carlo filter. For example, there can be an abstract class with either the primitive or defined operations. We will then have a concrete class that will inherit the abstract methods defined from the abstract class. For the implementation of the abstract methods in the subclass, we can first add the arguments (ex.: number of repeats and the random seed number) to the parser before parsing. Once that is done, we can then parse the added arguments and store the required values to the class fields, such as the number of repeats and the random number generator for the given seed, and then begin generating random samples.

In general, the template method pattern is useful for defining steps to use in a particular order for an algorithm. This provides classes more flexibility in how they would like to implement the steps in the algorithm, which can reduce code duplication and increase code reusability.

4. Abstract Factory Pattern

Abstract Factory design pattern is one of the Creational patterns. Abstract Factory pattern is similar to Factory Pattern and is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.

Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern. So at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type.



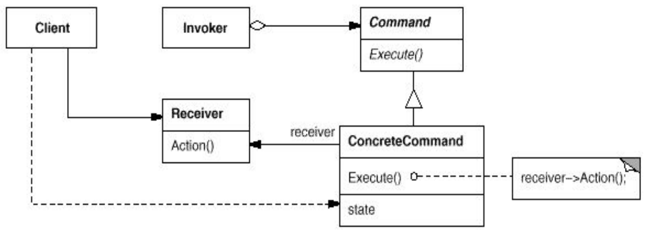
There are 5 classes and objects participating in this pattern. The AbstractFactory declares an interface for operations that create abstract products. The ConcreteFactory implements the operations to create concrete product objects. The AbstractProduct declares an interface for a type of product object. The Product defines a product object to be created by the corresponding concrete factory and it implements the AbstractProduct interface. The Client uses interfaces declared by AbstractFactory and AbstractProduct classes.

For example, the Abstract Factory pattern can be used to create the different variable types that needed to be instantiated (ex.: creating the pomposity instance that will contain min and max values associated with that particular object). By using this design pattern, all of the required variables for the MonteCarlo object can be created, where each variable will have its own factory method as well.

In general, the Abstract Factory method pattern is useful for creating families of related objects without necessarily specifying their concrete classes, unlike the Factory method pattern. To simplify the explanation, the Factory method can be used to create one product whereas the Abstract Factory method can be used to create families of related products.

5. 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.



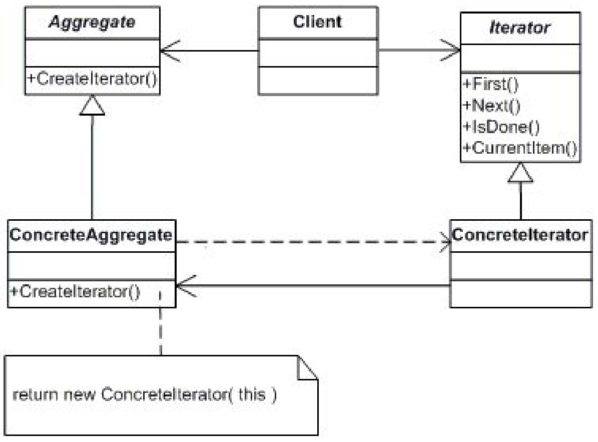
Five participants are involved in command pattern structure: Command, ConcreteCommand, Client, Invoker and Receiver. The Command declares an interface for executing and operation. The ConcreteCommand defines a binding between a Receiver object and an action, and it implements Execute by invoking the 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 execution process for a particular algorithm because there is no need to know about what the command to run any arbitrary program is, what the program needs, or what the program does specifically. By using the command pattern for any program, the content and execution process is encapsulated, which allows the execution of each action to be independent from other programs.

6. 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 nearly every language that implement the Iterator Pattern in some way. Each collection must provide an iterator that lets it or clients iterate through its objects.

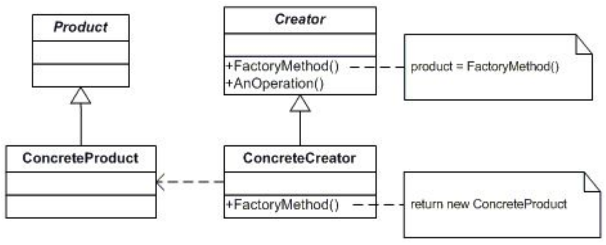
Four participants are in the iterator pattern: Iterator, ConcreteIterator, Aggregate and ConcreteAggregate. The Iterator defines an interface for accessing and traversing elements. The ConcreteIterator implements the Iterator interface and keeps track of the current position in the traversal of the aggregate. 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 any collections. For example, suppose we are building an application that requires us to print a collection of Strings. Eventually, some of the code will require iteration over the String collection to print all the Strings.

7. 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 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 is useful when you need to abstract the creation of an object away from its actual implementation. For example, suppose that we are going to build a “MobileDevice” product type. A mobile device could be made up of any number of components, some of which can and will change later, depending on advances in technology. The logic to create the mobile object has been encapsulated into class itself. If we want to change any of the component in the class, we can make the isolated change in the class without affecting the rest of the program.

8. 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.

For example, if we want to define the square of a number, we could implement a macro to simply achieve our goal, “#define square(x) (x*x)”. After being defined like this, our macro can be used in the code body as to find the square of a number.

However, we cannot implement macros for this project. The language we used for this project is Java. Java does not get preprocessed like C/C++ counterpart, hence there are no macros for Java.

9. State Machines Pattern

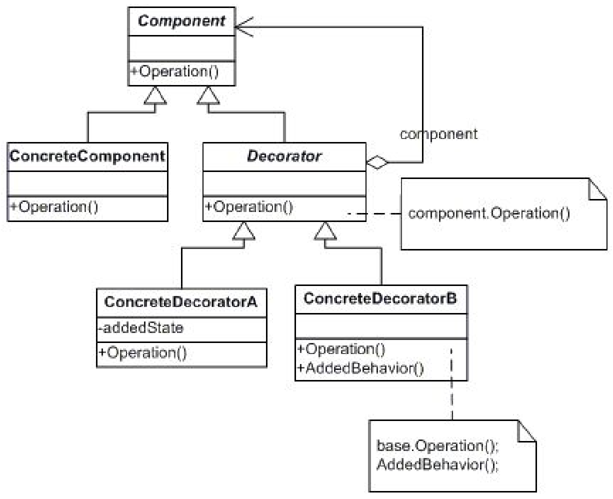
The State Machines Pattern allows an object to alter its behavior when its internal state changes. Automata could be constructed from independent state classes. The classes designed with State Machine pattern are more reusable than ones designed with State pattern. An intent of State Machine is the same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

For the monte carlo filter, 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 adding state where the arguments (ex.: the number of repeats and the random seed number) are added to the parser. Once the arguments have been added, the machine can then move to the parsing state, where the arguments added will be parsed. Then, the machine can move to the initialization state, such as creating a MonteCarlo object and having the necessary fields (ex.: pompository and learning curve) initialized as well.

In general, state machines offers a good, compact way to represent a set of rules and conditions that must be satisfied and to process certain inputs based on the current state of the machine or program. State machines can be used to represent an object that contains a limited number of states or conditions and can progress to the next state based on the defined rules. In other words, state machines can be used to simplify workflow.

10. 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 pizza x toppings problems. The pizza consists of a base where the client adds the toppings, then the system calculates the price and generates the order according the toppings added to the pizza.

EXPERIENCE & RECOMMENDATIONS

The three abstractions that we have chosen to implement for Project 2B are the following: Singleton pattern, Template pattern, and Strategy pattern.

The singleton pattern was a successful implementation, as it ensures that there is only one instance of a random seed generator present during the monte carlo simulation. For our implementation of the singleton pattern for the random seed generator, we made sure that only one instance of the generator exists and that the random seed is generated and stored in a field of type Random. In general, we would recommend using the singleton pattern if there is a need to encapsulate only one instance and have that readily available for use in various applications. In other words, a singleton instance is unique and ensures orderly access to shared resources. Even though our implementation was successful, we feel as if using the singleton pattern added unnecessary complexity to the already simple monte carlo filter. Without making the random number generator a singleton instance based on the given seed, we could simply generate and store a random number based on the given seed, which would reduce overall complexity and make it easier to track and understand what is going on in the program.

For the template pattern, the implementation was also a success. The template pattern was used in the monte carlo filter, where the methods for retrieving the number of repeats and the seed to generate the random number, creating the random number generator based on the given seed, and generating the random samples as part of the monte carlo simulation are defined as abstract methods to be implemented in subclasses. The template method pattern is useful for defining an algorithm as an outline of operations to perform and to leave the implementation details to the child or sub classes. We recommend using the template method pattern if you would like to run an algorithm in a particular order, but for a particular algorithm, you would like to define your own operations. We believe the template method pattern is appropriate since workflows can be defined and customized depending on how the program or application is running. For instance, for this monte carlo filter, we are expected to read in the number of repeats and the seed as command-line arguments. Suppose instead of reading the command-line arguments, those values are instead generated after creating the MonteCarlo instance. If there are different ways of running the monte carlo simulation, then the template method pattern allows certain operations to be customized based on certain needs without creating drastic change to the whole algorithm, which is good.

And for the strategy pattern, the implementation was also a success. For the monte carlo filter, we implemented one abstract strategy called PrintStrategy and two concrete strategies called CSVPrintStrategy and DictPrintStrategy. We believe the strategy pattern is appropriate for the monte carlo filter because there may be different ways a user would like to print the results from the monte carlo simulation. For instance, one user may like the results to be printed in comma separated value format (csv) whereas the other user may like the results to be printed in python dictionary format. So, for the strategy pattern in general, we recommend using the pattern for changing specific algorithms for specific situations. For the example used earlier, users can be given the option to have the results printed in a specific format, and the algorithm to print the formatted results can be changed easily. The only downside to using strategy would be that the user must have knowledge and understanding of the different strategies that exist in the application or program.

Overall, the use of the three abstractions for the monte carlo filter was good. When running the monte carlo filter 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 Java and contains three abstractions. Based on recommendations provided for the last project demo, we made updates on the report and added generalizations for each design pattern for this project (ex.: where each pattern would be useful).

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]: https://stackoverflow.com/questions/228164/on-design-patterns-when-should-i-use-the-singleton
[7]: https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm
[8]: Online lecture sources