CSCI 4448: Homework 4
Design Patterns

Domenic Murtari
Collaborated with Irakli Zhuzhunashvili and Sean Callahan

3/13/2014

1 Structural Problems

1.1 Structural Problem 3: Decorator

Structural Problem 3 gives the description of a Mario character who starts as a plain, small Mario, and can pick up various power-ups. The decorator pattern is most applicable for this application, since the desire is to have a base character (the plain Mario), and to be able to customize that Mario by giving him power-ups.

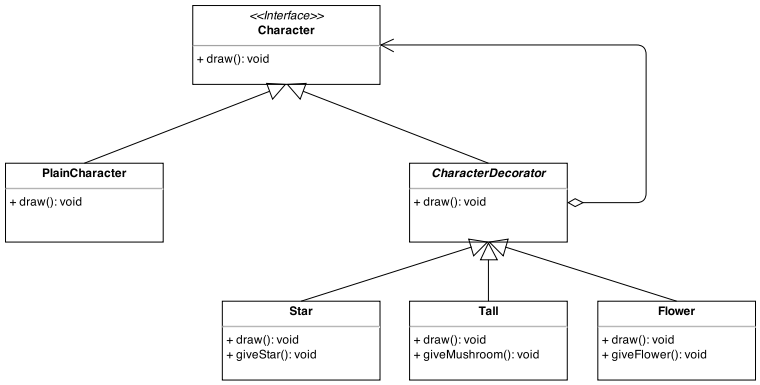

Figure 1: Class diagram of Decorator pattern implementation

- Component interface is implemented by `Character`, which describes the method that will be used to decorate the object (in this case, each decorator will implement `draw()`).
- The concrete component is implemented by `PlainCharacter`, which defines the default character that is able to be decorated.
- The decorator component which provides the ability to decorate Mario with different states of being powered up is implemented by `CharacterDecorator`.
- The concrete decorators which provide the ability to add different power-ups to the plain Mario are provided by `Star, Tall` and `Flower`.

1.2 Structural Problem 5: Proxy

Structural Problem 5 describes a bank that needs to access an SQL database, but would like the commands for the SQL database to be executed after the database has been closed for the day. The Proxy pattern best solves this problem, because it allows for the proxy to stand in for the actual database, and the bank will issue commands to the proxy which will send the commands to the actual database once the bank closes for the day.

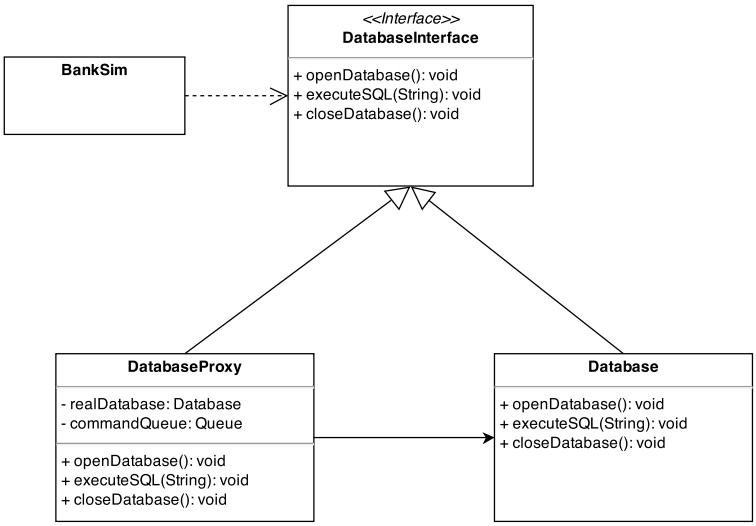

Figure 2: Class diagram of Proxy pattern implementation

- The role of the subject interface is played by `DatabaseInterface`, which both the real database and the proxy implement allowing for a consistent interface between the two.
- The RealSubject is implemented by `Database` which is the actual instance of the database that the bank interacts with through the proxy.
- The Proxy is implemented by `Proxy` which allows the bank to interact with it as if it were the bank, but issues the SQL commands to the real database after the bank closes.
- The Client is provided by `BankSim` which simulates the bank interacting with the database.

2 Creational Problems

2.1 Creational Problem 1: Singleton

Creational Problem 1 presents an auto-grader that could potentially be multi-threaded, which gives rise to the need for a queue that will be compatible with multiple threads. The Singleton Pattern solves this, since only one singleton can exist (even for multiple threads), and will thus ensure that each auto-grader running in its own thread will pull from the same queue

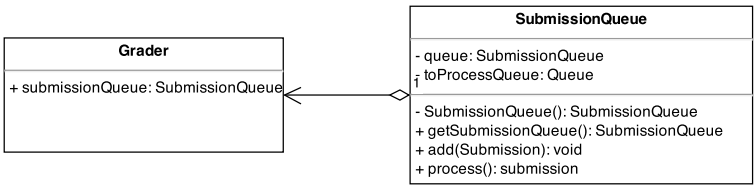

Figure 3: Class diagram of Singleton pattern implementation

- The Singleton is represented by `SubmissionQueue`, which maintain a queue of submissions that need to be graded. The constructor for the SubmissionQueue is private, only being invoked by `getSubmissionQueue` which only creates a new `SubmissionQueue` if one does not exist already.
- The client is represented by `Grader`, which aggregates an instance of `SubmissionQueue` and can only instantiate SubmissionQueues through the `getSubmissionQueue` method.

2.2 Creational Problem 2: Prototype

Creational Problem 2 gives a neural network that learns. The ability to make new neural networks without having to have the new neural network relearn everything that the first neural network learned. The Prototype pattern resolves this issue, since the Prototype pattern gives a way for new instances of a class to be created which contain information from the original object.

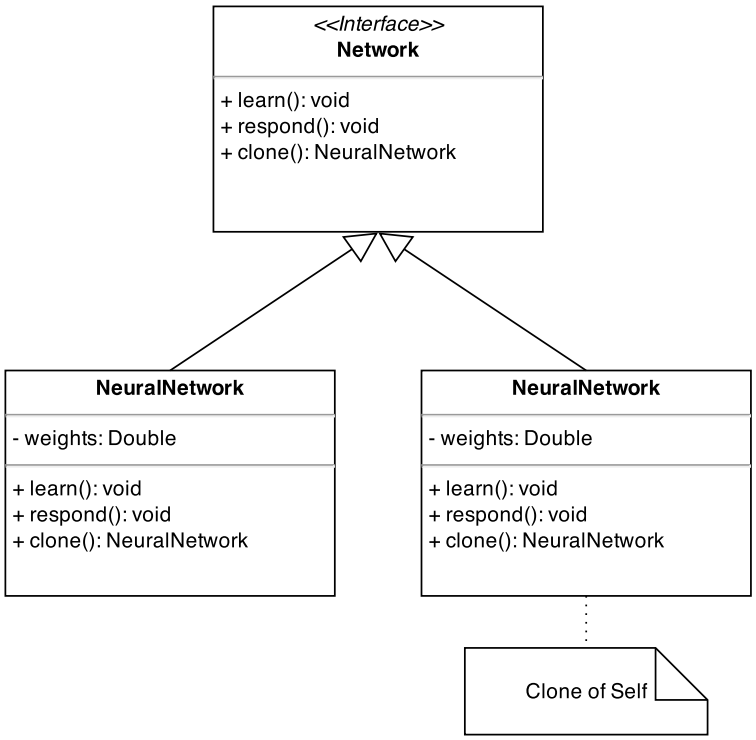

Figure 4: Class diagram of Prototype pattern implementation

- The Prototype interface is represented by `Network`, which ensure that any class implementing it will implement a `clone()` method.
- The Concrete Prototypes are represented by `NeuralNetwork` and `NeuralNetwork1`. The client will call `NeuralNetwork`'s `clone()` method, which creates a new neural network object `NeuralNetwork1` which contains everything `NeuralNetwork` has learned

3 Behavioral Problems

3.1 Behavioral Problem 2: Strategy

Behavioral Problem 2 gives an example of designing an algorithm that will best select an artificial intelligence to play a game based on criteria such as timers and the game complexity. The strategy pattern fits this problem well, since it allows the selection of different strategies at runtime depending on the type of problem. The game player will be able to select the appropriate artificial intelligence based on parameters from which it will instantiate the appropriate game playing algorithm.

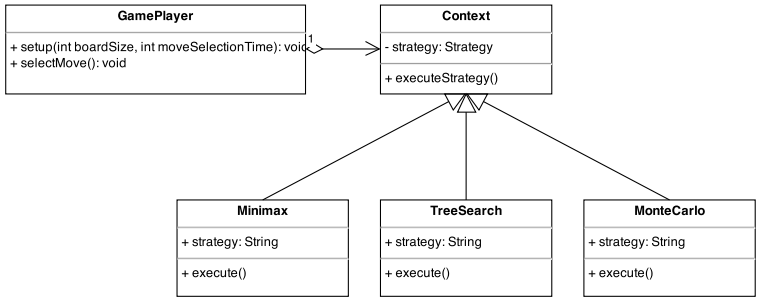

Figure 5: Class diagram of Strategy pattern implementation

- The strategy abstraction is represented by `Context`, which handles the variations in strategy. `Context` forces each strategy to implement the same methods so that the client will be able to expect the same functionality no matter which strategy is used.
- The concrete strategies are represented by `Minimax`, `TreeSearch`, and `MonteCarlo`. These each implement the methods in `Context` and solve the given problem according to their algorithm.
- The client is represented by `GamePlayer`, which selects a strategy appropriate to the given game through the `setup()` method. `GamePlayer` will be able to call the same methods no matter which strategy was selected.

3.2 Behavioral Problem 3: Visitor

Behavioral Problem 3 presents the problem of a simulation that needs to be logged while the simulation is running. The Visitor pattern solves this problem well, since it allows an outside object to collect information about a running object without needing to expose the fields or methods of the object that is being visited.

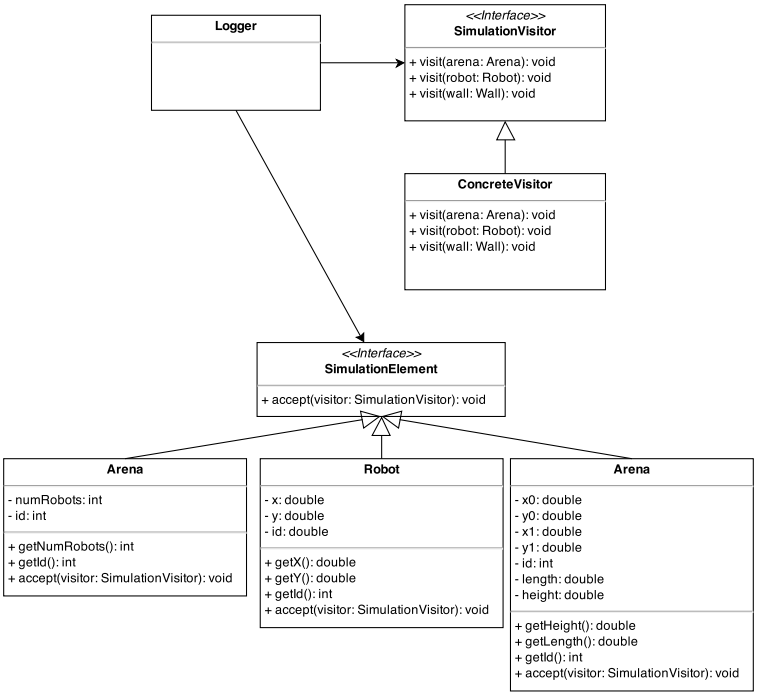

Figure 6: Class diagram of Visitor pattern implementation

- The Visitor interface is implemented by `SimulationVisitor`, which ensures that the visitors will each implement a visit method for each of the classes that are visitable.
- The Concrete Visitor is implemented by `ConcreteVisitor` which implements the `SimulationVisitor` interface, meaning that it must support visiting each of the classes that accept visitors
- The Element is represented by `SimulationElement`, which enforces that each element of the simulation implements a `accept()` method to accept visitors.
- The Concrete Elements are represented by `Arena`, `Robot`, and `Wall`, which implement the `SimulationElement` interface by having an `accept()` method that accepts a visitor object and allows that object to see the information contained within that object
- The Client is represented by `Logger` and accesses the various elements of the simulation using the Visitor model.