Seminar report

Behavioral design patterns in JavaScript

Mark Budiak

Department of Bioinformatics and Computational Biology, Faculty for Informatics Boltzmannstr. 3 85748 Garching

ABSTRACT

This report is oriented on behavioral patterns in JavaScript. In the beginning of the report is presented the introduction to the topic and description of the Behavioral design patterns and their classification. After presenting the Behavioral pattern, the report focuses on Chain of Responsibility and Command Behavioral design patterns. In addition to the theory, this report presents two refactored solutions for JavaScript projects where each of the patterns is applied.

1 INTRODUCTION

Design patterns are very well known among software developers but in connection with JavaScript just very few of them would even think about patterns in this language. This is partially understandable since this language is not object oriented and some people do not even recognize it as a programming language. Despite these facts, there are design patterns for JavaScript and this report explains some of them, how they work in JavaScript and their application in the JavaScript projects.

The research was mainly based on the book "Design Patterns – Elements of Reusable Object-Oriented Software" by authors commonly referred to as the "Gang of Four". [1] This book provides readers of a basic understanding of behavioral design patterns. The next stage of the research was focused on the patterns in JavaScript. The main source of information in this stage was the web page "dofactory.com". [2] The research found that there are no specific behavioral patterns for JavaScript language. There are just the usual behavioral design patterns presented in the book by the "Gang of Four" authors which are adjusted to the specifications of JavaScript language.

2 BEHAVIORAL DESIGN PATTERNS

Behavioral design patterns are the largest group among all of the three categories of design patterns (structural, behavioral and creational). It comprises eleven patterns: Chain of Responsibility, Command, Iterator, Observer, Strategy, Interpreter, Mediator, Memento, State, Template Method and Visitor. According to the "Gang of Four" book, behavioral design patterns can be also divided into class and object oriented patterns, where class oriented patterns use inheritance to distribute behavior and object oriented patterns use object composition. [1]

The main function of Behavioral design patterns is that they deal with the algorithms and the assignment of responsibilities between objects. [1] They make programs more flexible e.g.: by specifying which algorithms should be used at runtime, the ability to be easily extendable and by specifying which objects are responsible for particular task. Some behavioral patterns like the Mediator or the Observer facilitate communication between objects. They set standards how objects can communicate with each other. Other patterns such as Chain of Responsibility or State simplify complex control flow by reducing complex if-else and switch-case statements and encapsulating them into new objects.

The behavioral patterns are usually used in combination with the inheritance. Object oriented languages provide inheritance but JavaScript does not provide typical inheritance but instead it provides prototypal inheritance. However, it cannot be used in the same way as in object oriented languages. The prototypal inheritance is done via the variable prototype. In this variable is stored object which is considered as "super object" in parallel to super class. [3] In JavaScript, the inheritance is mainly used for sharing the same method and it does not make much sense to define their prototypes since the language does not support overriding of methods.

3 CHAIN OF RESPONSIBILITY PATTERN

Chain of Responsibility is mainly concerned with simplifying complex control flow and with assigning the responsibilities to objects. It decouples the sender of a request from its receivers by giving the chance also to other objects to handle the request. [1] The client does not know which handler can handle a specific request – the client just sends it through the chain. It is based on the same principle as a linked list but in this case objects in the chain do not store values but they are trying to handle a particular request.

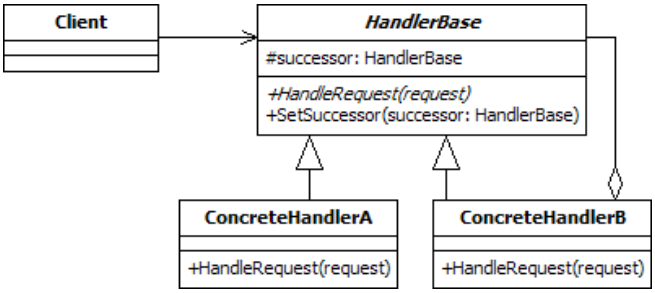

Figure 1 UML diagram of the Chain of Responsibility pattern [4]

The workflow of the Chain of Responsibility pattern begins with the Client initialization of the request into the chain. Subsequently, each Handler is trying to handle the request. The handler does not have to handle the request itself but instead it can contribute to the solution. The implementation depends on a developer. Although, allowing contribution of multiple handlers is similar to Blackboard architectural pattern which is used to coordinate different subsystems in solving problems on a blackboard.

This pattern can be implemented in JavaScript in two ways. One way is to use it without inheritance, so there will be just concrete handlers' objects with method handle and variable which stores link to a next handler in a chain. The alternative is to use inheritance and introduce a super object which will store the variable "next" as the link to another handler. The second option is recommended when there are many handlers. This option can also make code more readable and reduce lines of code. Otherwise, the first option is recommended.

3.1 Uses

This pattern can be used in projects with complex if - else if - else or switch - case statements. It can be used in systems where there is not known which object can handle the request. Examples of such systems are help or error handling subsystems. This pattern is also used in the JavaScript prototypal inheritance.

3.2 Drawbacks

The main drawback of this pattern is that the request does not have to be handled at all and the client will not be informed about this state. Since the pattern is just a generalized solution we can adjust it to our concrete implementation and introduce such countermeasures that will avoid this state to occur.

Sometimes can be difficult to follow the logic of the path in the code at runtime especially when the chain is big and the logic of the handling mechanism is complex. [5]

4 COMMAND PATTERN

The Command pattern is mainly used for decoupling invokers - boundary objects (GUI) from the actions - implementation of behavior and thereby enables easy change of boundary objects. Command encapsulates a request as an object and thereby allows to parametrize clients with different requests and supports undoable operations. [1] Basically, the main role is to maintain the binding between a receiver and an action.

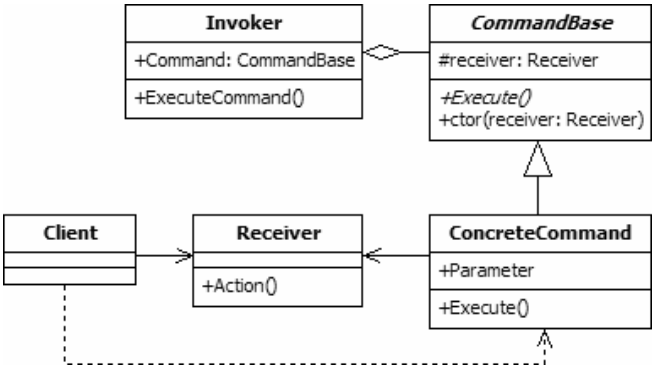

Figure 2 UML diagram of the Command pattern [6]

The workflow of this pattern begins with the Client which sets the environment of the program. When the invoker (usually GUI object or keyboard) is invoked the particular command is executed. Inside of the method "execute" is called method "action" of the receiver object which does the specific action.

This pattern can be also implemented in the same way as the Chain of Responsibility pattern in terms of inheritance. In this case, using the super object does not make sense because it would just store receiver object and there are many better ways how to do it. One of them will be presented in the Snake game project.

4.1 Uses

The Command pattern is mostly used in menus of GUI (graphical user interface) applications. It can be also used when there is a need for Redo/Undo manager since commands are objects and hence can be easily stored. Usage of the Command pattern in a Redo/Undo manager is tightly connected to the Memento pattern which deals with storage of objects.

4.2 Drawbacks

By using the Command pattern, many new classes or objects are introduced because each command has its own class. When there is a lot of commands used the design can look cluttered but this is more or less drawback of all behavioral patterns. [7]

5 REFACTORED PROJECTS

There were refactored three projects as the showcase how patterns can be used in JavaScript. All the original and refactored projects of this research are available in the GitHub repository. [8] First project is the Snake game which demonstrates usage of the Command pattern. The second is Rock-Paper-Scissors game where is applied the Chain of Responsibility pattern. The third is the Date parser where is also applied the Chain of Responsibility pattern. The third project is not described in this report since there is applied the same pattern as in Rock-Paper-Scissors game and this game is better showcase. However, the result of refactoring of the third project is also available in the repository of this research.

5.1 Snake game

The Snake game is a JavaScript web browser game. [9] The goal of this game is to command a snake which is getting bigger by collecting food.

The problem of this game is that invokers - boundary objects (in this case keyboard) are coupled with actions (changing of directions). In the Figure 3 we can see that changing of direction of snake is done in the Invoker. This is not very good approach because when there is need to reuse this commands, the code gets unnecessarily bigger.

```
handleKeyboard: function() {
  if (this.keyboarder.isDown(this.keyboarder.KEYS.LEFT) &&
      this.direction.x !== 1) {
    this.direction.x = -1;
    this.direction.y = 0;
  } else if (this.keyboarder.isDown(this.keyboarder.KEYS.RIGHT) &&
             this.direction.x !== -1) {
    this.direction.x = 1;
    this.direction.y = 0;
  }

  if (this.keyboarder.isDown(this.keyboarder.KEYS.UP) &&
      this.direction.y !== 1) {
    this.direction.y = -1;
    this.direction.x = 0;
  } else if (this.keyboarder.isDown(this.keyboarder.KEYS.DOWN) &&
             this.direction.y !== -1) {
    this.direction.y = 1;
    this.direction.x = 0;
  }
},
```

Figure 3 the snake game before refactoring

Possible solution is to use the command pattern and decouple invokers from actions. There are four directions where the snake can move (left, right, up, down). We can introduce a command for each of these actions. In order to reuse these commands the receiver object (context of the game) is passed as an argument to execute the command which is responsible for invoking the action. In the Figure 4 is the example of one of the commands. In this case, the action is the direction which changes the direction of the snake.

```
var upCommand = {
  execute: function(receiver){
      receiver.direction.y = -1;
      receiver.direction.x = 0;
  }
};
```

Figure 4 Example of the UpCommand

After applying the command pattern, the actions are not called directly from the invoker (function that handles keyboard events) but instead the particular commands are called. The Figure 5 shows the refactored solution.

```
handleKeyboard: function() {
  if (this.keyboarder.isDown(this.keyboarder.KEYS.LEFT) &&
      this.direction.x !== 1) {
    leftCommand.execute(this);
  } else if (this.keyboarder.isDown(this.keyboarder.KEYS.RIGHT) &&
             this.direction.x !== -1) {
    rightCommand.execute(this);
  }

  if (this.keyboarder.isDown(this.keyboarder.KEYS.UP) &&
      this.direction.y !== 1) {
    upCommand.execute(this);
  } else if (this.keyboarder.isDown(this.keyboarder.KEYS.DOWN) &&
             this.direction.y !== -1) {
    downCommand.execute(this);
  }
},
```

Figure 5 the snake game after refactoring

Advantage of this approach is mainly its reusability. It means that in case of adding option for multiplayer, there is no need to add the same lines of code twice but instead of this it is better to use the same commands with different receiver object (another snake) as an argument. Beside this advantage, it also increases code readability. In order to introduce multiplayer option, other changes are required in the code. The main one is to decouple snake object from game context. This change was not done because it would not be refactoring but will lead to the change of the project.

5.2 Rock-Paper-Scissors game

Rock-Paper-Scissors game is the JavaScript web browser lightweight game. [10] It uses alert dialogs for communicating with the user. Game supports only User vs. Computer mode. Firstly the user enters input from three choices (rock, paper, and scissors). Then the code computes the Computer choice. Finally both choices are inputs for the comparing function which decides who won.

The problem of this code is in the comparing function which is implemented by the complex structure of if - else statements. It is difficult to follow the logic for the first time. The Figure 5 shows the complex structure of if-else statements.

```
var compare = function(choice1,choice2) {
if(choice1===choice2) {
return "The result is a tie!";}

else if (choice1 === "rock"){

if (choice2 === "scissors")
{
return "rock wins";}
else
{ return "paper wins";}
}
else if (choice1 === "paper"){
if ( choice2 === "rock"){
return "paper wins"}
else (choice2 === "scissors");{
return "scissors wins"}
}
else if (choice1 === "scissors");{
if (choice2 === "rock")
{return "rocks wins"}
else(choice2 === "paper");{
return "scissors wins"}
}
};
```

Figure 6 Comparing function of the Rock-Paper-Scissors game

The possible solution is to use the Chain of Responsibility pattern. There are just five scenarios which can happen inside comparing function. (Draw, Rock wins, Paper wins, Scissors win and undefined) Each of the scenarios can be encapsulated into handler. As described earlier there are two main possibilities how to realize the Chain of Responsibility pattern. In this case is used the combination of both of them. Handler does not use inheritance to inherit super object but it also does not have variable "next" where the link to next handler is stored. Instead of this, the handler takes an advantage of the prototype variable used in JavaScript inheritance and stores there the link to the next Handler. In the Figure 7 is the example of the Handler implementation.

```
var rockWins = {
    handle: function(choice1, choice2){
        if((choice1 === "rock" || choice2 === "rock")
            && (choice1 === "scissors" || choice2 === "scissors"))
            return "rock wins";
        else
            return rockWins.__proto__.handle(choice1, choice2);
    }
};
```

Figure 7 Example of the rockWins Handler

The drawback of the Chain of Responsibility pattern that the request does not have to be handled at all is eliminated by adding the fifth handler which is in the end of the chain and is called when none of the handlers can handle the request and returns undefined value.

This approach eliminates the complex structure of if-else statements and the code is reduced just to one line of code which invokes the first handler in the chain. Figure 8 shows the result after refactoring the comparing function.

```
var compare = function(choice1,choice2) {
    return tie.handle(choice1, choice2);
};
```

Figure 8 comparing function after refactoring

The main benefit of this approach is that it increases readability and it is easily extendable. One of the drawbacks of this approach is that it uses inheritance variable "prototype" to store the next handler which is not very good solution in terms of semantic but in this case it is appropriate. This approach requires initialization of the chain (connect handlers to each other) which can be also perceived as the drawback.

SUMMARY

This report presents behavioral patterns in JavaScript and shows how they can be applied in JavaScript projects. This proves that object oriented patterns can be also applied with small modifications in non-object oriented languages such as JavaScript. Introduction of Node.JS extended the usage of this language from frontend to backend. Despite this extension, many JavaScript projects are not usually complex and therefore applying the Behavioral design patterns in many cases can lead to over-engineering.

The most interesting part is how JavaScript handles inheritance in connection to the Behavioral design patterns. The Behavioral design patterns usually do not use an inheritance in JavaScript because it does not have advantages as the object oriented languages have e.g.: JavaScript does not support overriding of methods. Decision how the Behavioral design patterns are used always depends on the concrete situation (size of the project). Therefore sometime it is good to use inheritance (big projects with a lot of objects – reduction of unnecessary lines of code) and sometimes it is not effective to use inheritance (small projects with a few objects).

REFERENCES

[1] Erich Gamma et al., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, Massachusetts, USA, 2007
[2] http://www.dofactory.com/javascript/design-patterns - May 6th, 2015
[3] http://javascript.info/tutorial/inheritance - May 6th, 2015
[4] http://www.blackwasp.co.uk/ChainOfResponsibility.aspx - May 6th, 2015
[5] http://java.dzone.com/articles/design-patterns-uncovered-chain-of-responsibility - May 6th, 2015
[6] http://www.blackwasp.co.uk/Command.aspx - May 6th, 2015
[7] http://java.dzone.com/articles/design-patterns-command - May 6th, 2015
[8] https://github.com/markb21/behavioral-patterns - May 6th, 2015
[9] https://github.com/maryrosecook/retro-games - May 6th, 2015
[10] https://github.com/macikokoro/games - May 6th, 2015