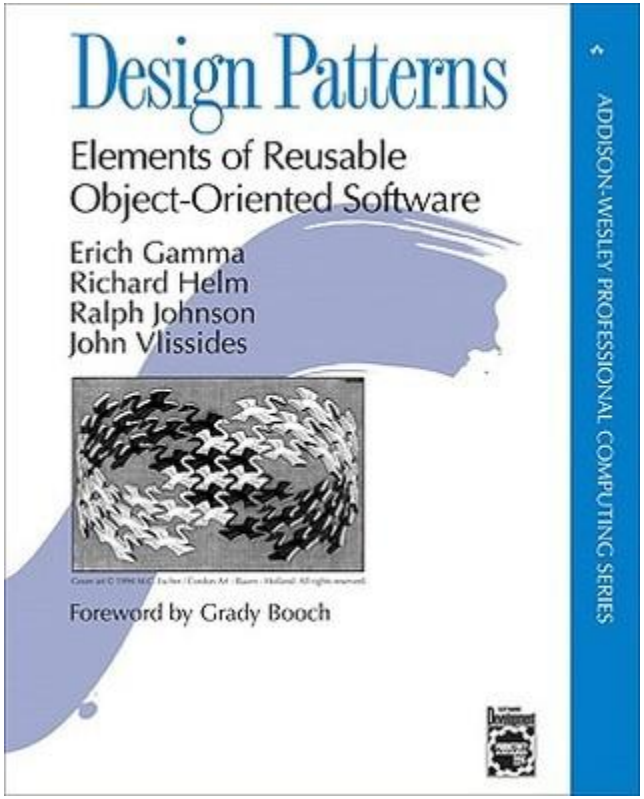
Behavioral Patterns

Mark Budiak , Benjamin Helgert

Technische Universität München

Materials

GoF book



Dofactory

Materials

JS - The Good Parts

Content

Overview

- Behavioral Patterns in nutshell
- Strategy
- Iterator
- Observer
- Chain of Responsibility
- Command

Refactored Projects

- Computer Science in JavaScript
- JavaScript Date
- impress.js editor
- JS snake game
- JS rock, paper, scissor game

Content

Overview

- Behavioral Patterns in nutshell
- Strategy
- Iterator
- Observer
- Chain of Responsibility
- Command

Refactored Projects

- Computer Science in JavaScript
- JavaScript Date
- impress.js editor

Behavioral patterns in nutshell

- Concerned with algorithms and the assignment of responsibilities between objects
- Facilitate the communication between objects
- Simplify complex control flow

Behavioral pattern family

- Chain of Responsibility
- Command
- Iterator
- Observer
- Strategy
- Interpreter
- Mediator
- Memento
- State
- Template method
- Visitor

Behavioral patterns in nutshell

- Concerned with algorithms and the assignment of responsibilities between objects
- Facilitate the communication between objects
- Simplify complex control flow

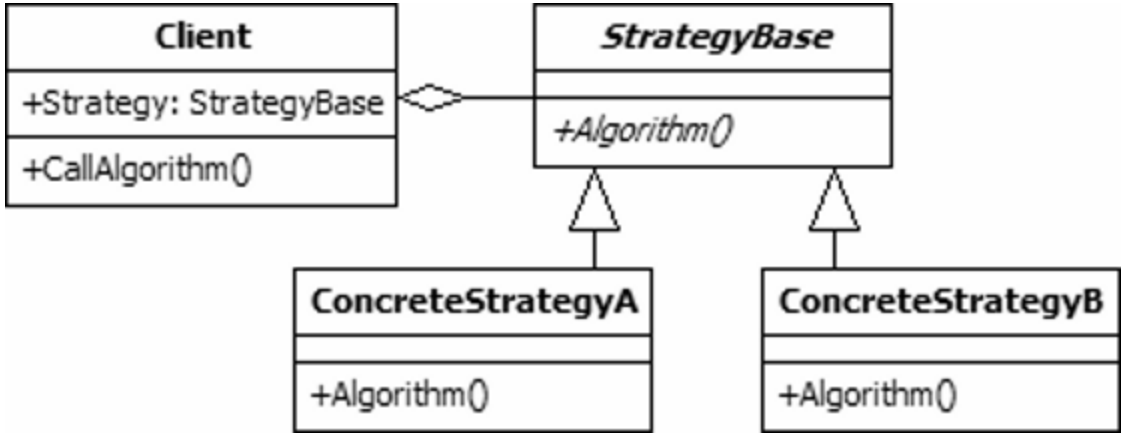
Behavioral pattern family

- Chain of Responsibility
- Command
- Iterator
- Observer
- Strategy
- Interpreter
- Mediator
- Memento
- State
- Template method
- Visitor

Strategy

- Can be applied when there exists many ways (algorithms) of solving one problem
- Hides the concrete implementation of an algorithm and enables easy add or removal of algorithms

Strategy - Structure



Strategy

Uses

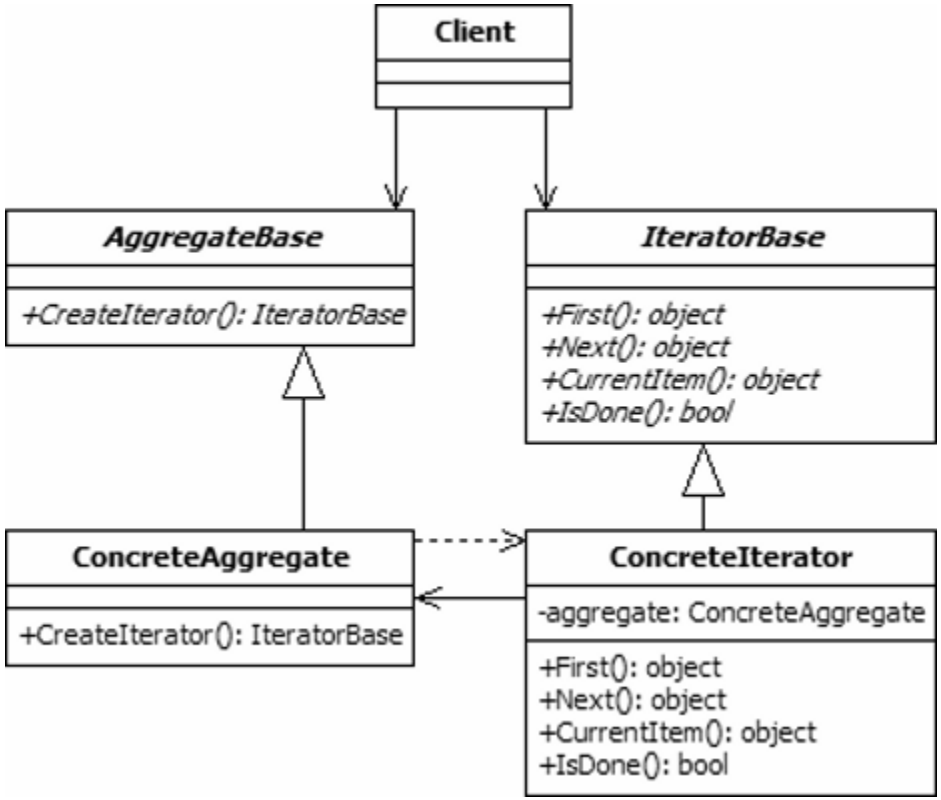
- online shopping systems (way of shipping and payment)
- choosing sort algorithms
- encryption programs

Iterator

- Allows iterating through a collection of objects regardless of the implementation of the collection

- Different types of Iterators can be implemented that support different ways of traversal

Iterator - Structure



Iterator

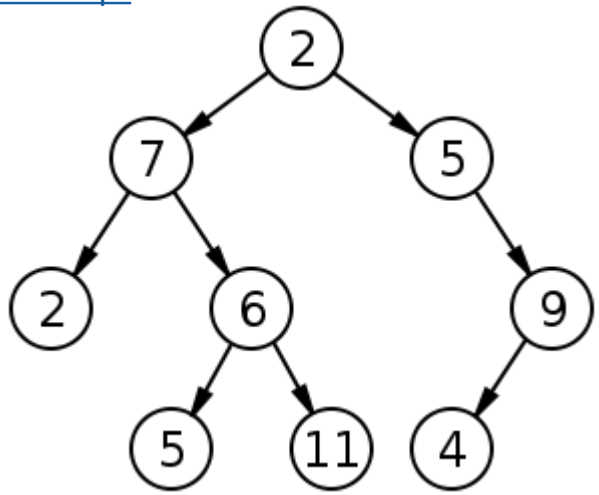
Uses

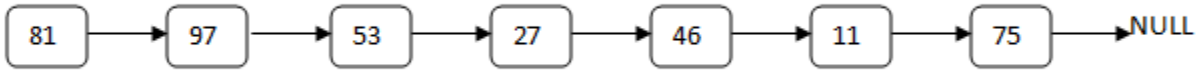
- Common in object-oriented languages, e.g. Java to provide a way to iterate over lists, arrays, sets, …

Computer Science in JavaScript

Implementation of several algorithms and data structures.

https://github.com/nzakas/computer-science-in-javascript





Computer Science in JavaScript

Sorting algorithms

Before

(global scope)

```
function swap(items, firstIndex, secondIndex){
...
}

function partition(items, left, right) {
...
}

function quickSort(items, left, right) {
...
}
```

After

```
var newQuickSort = function (){
    var that = {};
    var swap = function(items, firstIndex, secondIndex){
    ...
    }
    var partition = function(items, left, right) {
    ...
    }
    var sort = function( items, left, right ) {
    ...
    }
    that.sort = function(items){
        return sort(items);
    }
    return that;
};
```

Computer Science in JavaScript

Sorting algorithms

```
var newSorter = function(){
    var that = {},
        spec = {};
    that.setStrategy =  function( strategy ){
            spec.currentStrategy = strategy;
        }
    that.sort = function( array ){
            return spec.currentStrategy.sort( array );
        }
    return that;
};
```

```
var sorter = newSorter();
sorter.setStrategy( createQuickSort() );
var unsorted_array = [3,1,2];
var sorted_array = sorter.sort( unsorted_array );
```

Tree iterators

There was no way to traverse Binary Search Trees, so we implemented an Iterator for that purpose.

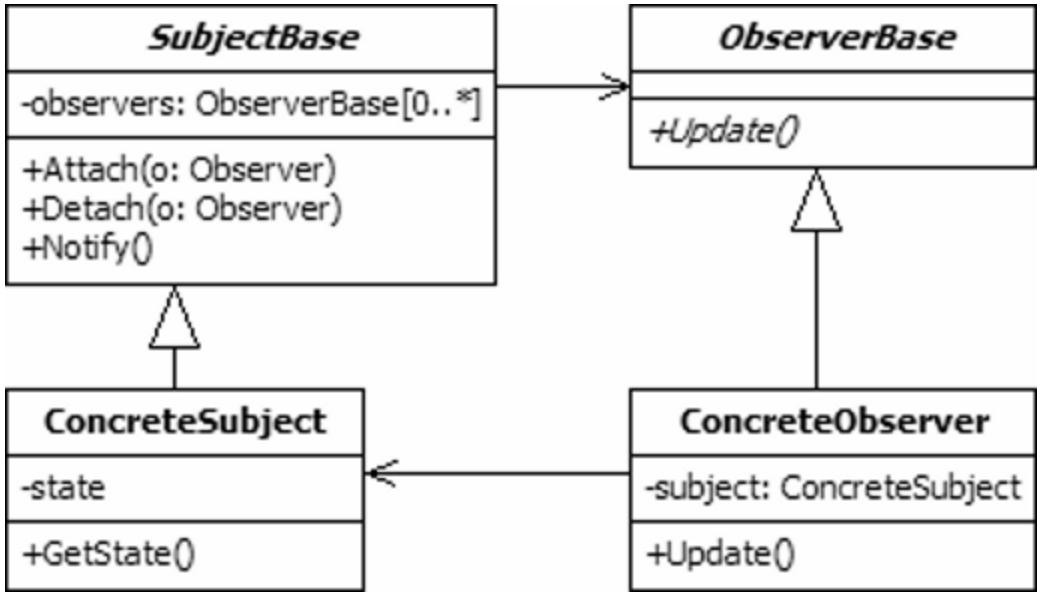
```
function Iterator(){
    this.next = function(){
            throw "This iterator does not implement next()."
        }
};
// You can check whether some object is an Iterator by using
// instanceof Iterator
function newBinarySearchTreeIterator( spec ){
    var that = new Iterator(),
        currentNode = null;

    that.next = function(){
        if(currentNode===null){
            currentNode = spec.minNode(spec._root);
        }
        else{
            try{
                currentNode = spec.successorNode(currentNode);
            }
            catch(e){
                currentNode = null;
            }
        }
        return currentNode;
    }
    return that;
}
```

Observer

- Applies to problems with multiple views or objects which have to be notified about the current state of data

- Supports extensibility by decoupling Subject (Observable) from Observers

Observer - Structure



Observer

Uses

- Maintaining consistency across multiple views
- Event management

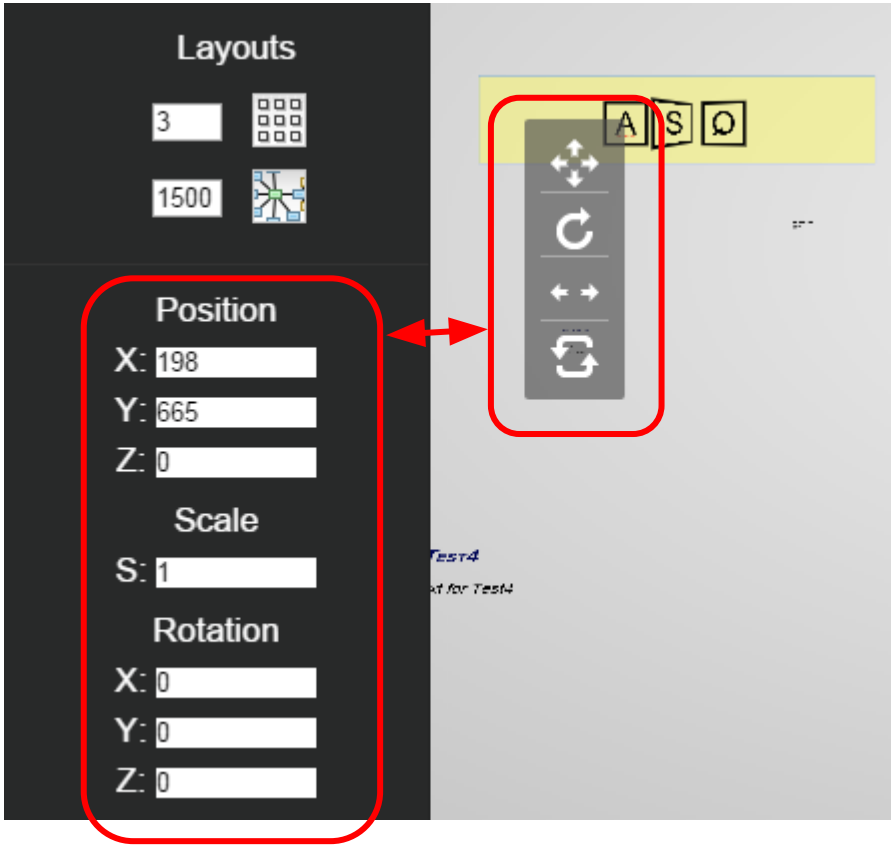
Drawback

- It can be hard for observers to track down changes without a change log

impress.js editor

A graphical editor for creating impress.js presentations.

https://github.com/giokokos/editor



impress.js editor - before

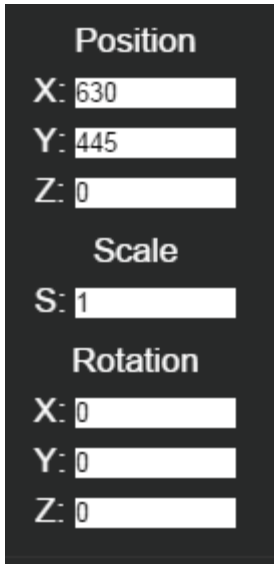
Textboxes

changed the slide position

-> updated the global state

-> updated the controls

-> updated the text



Controls

updated the global state

-> updated the controls

-> updated the text

-> updated the slide position

impress.js editor - before

Code sample

global functions that call each other to update the view

```
// Gets called when using the controls
// to move a slide
function handleMouseMove(e) {
    ...
    redraw();
    ...
}
// positions text boxes and controls
function showControls($where) {
    ...
}
```

```
// positions the selected slide
function redraw() {
    ...
    showControls(...);
    ...
}
// updates the state with the data of $el
function updateSync($el) {
    ...
    selection.setX(state.data.x);
    redraw();
    ...
}
```

impress.js editor - after

ObservableState object

- hides the state object
- has 2 registered observers, one for each view (text boxes and controls)

Textboxes and Controls

- call observableState.setState() to update the hidden state object
- observableState notifies its observers which in turn update the views

impress.js editor - after

ObservableState object

```
var observableState = newObservableState();
var textBoxObserver = newTextBoxObserver();
var controlsObserver = newControlsObserver();
observableState.addObserver(textBoxObserver);
observableState.addObserver(controlsObserver);
```

```
var notify = function (){
    var iterator = observerSet.createIterator(),
        observer = iterator.next();

    while( observer !== undefined ){
        observer.update(that);
        observer = iterator.next();
    }
}
```

impress.js editor - after

HashSet Iterator

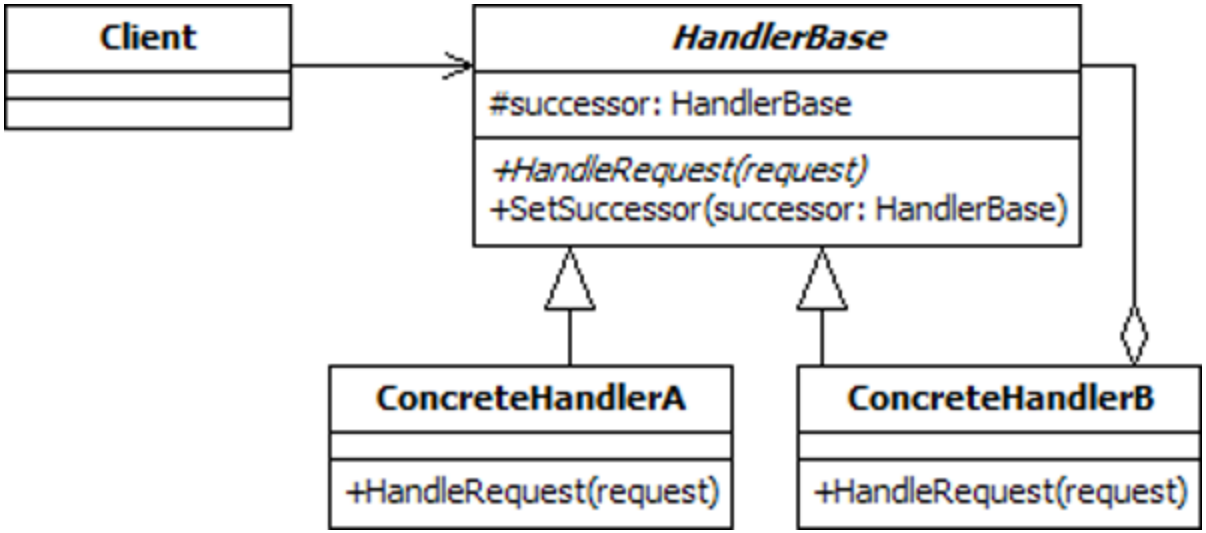
```
var newHashSet = function (){
    var that = {},
        objectSet = {};
    var createIterator = function () {
        var that = {},
            currentIndex = 0,
            keys = Object.keys( objectSet );
        var next = function () {
            return objectSet[ keys[ currentIndex++ ] ];
        }
        that.next = next;
        return that;
    }
    ...
}
```

Chain of Responsibility

- Decouples the sender of a request from its receivers by giving the chance also to other objects to handle the request

- The client does not know which handler can handle specific request. Client just send it through the chain

Chain of Responsibility - Structure



Client – initiates the request to the chain

Handler – handles the request and implements successor link to other handlers

Chain of Responsibility

Uses

- System where we do not know certainly which handler can handle the request
- Help subsystem
- Error handling system
- is used in Javascript prototype chain of objects
- Instead of huge if - else if statements

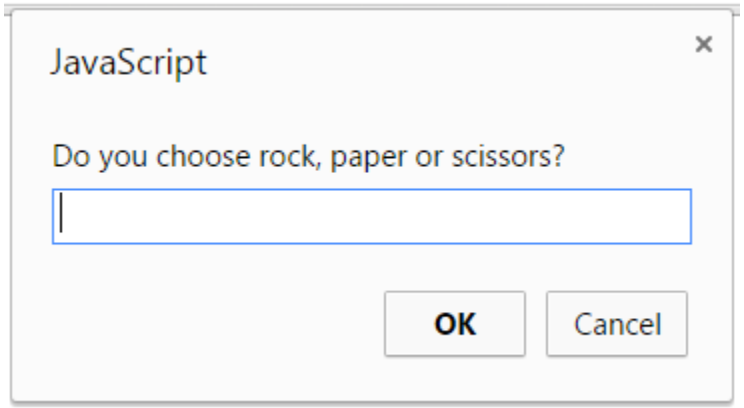
Drawbacks

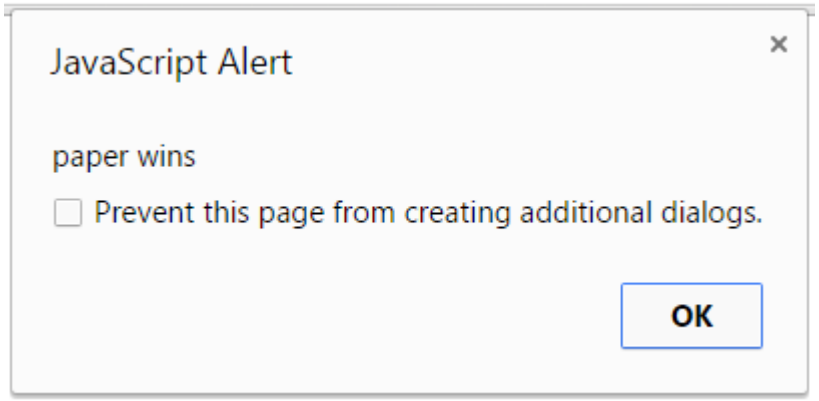
- difficult to follow the logic of a path in code at runtime
- potential threat - the request will not be handled at all

Rock-Paper-Scissors game

https://github.com/macikokoro/games

- JavaScript web browser lightweight game





Rock-Paper-Scissors game before

```
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"}
}
};
```

Rock-Paper-Scissors game refactoring

```
var tie = {
    handle: function(choice1, choice2){
        if(choice1===choice2)
            return "The result is a tie!";
        else
            return tie.__proto__.handle(choice1, choice2);
    }
};

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);
    }
};

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

Rock-Paper-Scissors game after

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

JavaScript Date

https://github.com/MatthewMueller/date

- Node JS console application
- date parser
- takes input in english language and calculates date which corresponds to the input

Input

```
var parse = require('./lib/parser');

var mon = new Date('May 13, 2013 01:30:00');

console.log(parse("7 days ago", mon));
```

Output

```
Terminal
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Mark\Desktop\date-master1>node .
Mon May 13 2013 01:30:00 GMT+0200 (Central Europe Daylight Time)
Mon May 06 2013 01:30:00 GMT+0200 (Central Europe Daylight Time)
```

JavaScript Date - Refactoring

Before:

```
/**
 * Advance a token
 */

parser.prototype.advance = function() {
  var tok = this.eos()
    || this.space()
    || this._next()
    || this.last()
    || this.dayByName()
    || this.monthByName()
    || this.timeAgo()
    || this.ago()
    || this.yesterday()
    || this.tomorrow()
    || this.noon()
    || this.midnight()
    || this.night()
    || this.evening()
    || this.afternoon()
    || this.morning()
    || this.tonight()
    || this.meridiem()
    || this.hourminute()
    || this.athour()
    || this.week()
    || this.month()
    || this.year()
    || this.second()
    || this.minute()
    || this.hour()
    || this.day()
    || this.number()
    || this.string()
    || this.other();

  this.tokens.push(tok);
  return tok;
};
```

After:

```
/**
 * Advance a token
 */

function advance() {

  var tok = space.handle();

  Data.tokens.push(tok);
  //console.log(tok);
  return tok;
};
```

JavaScript Date - Refactoring

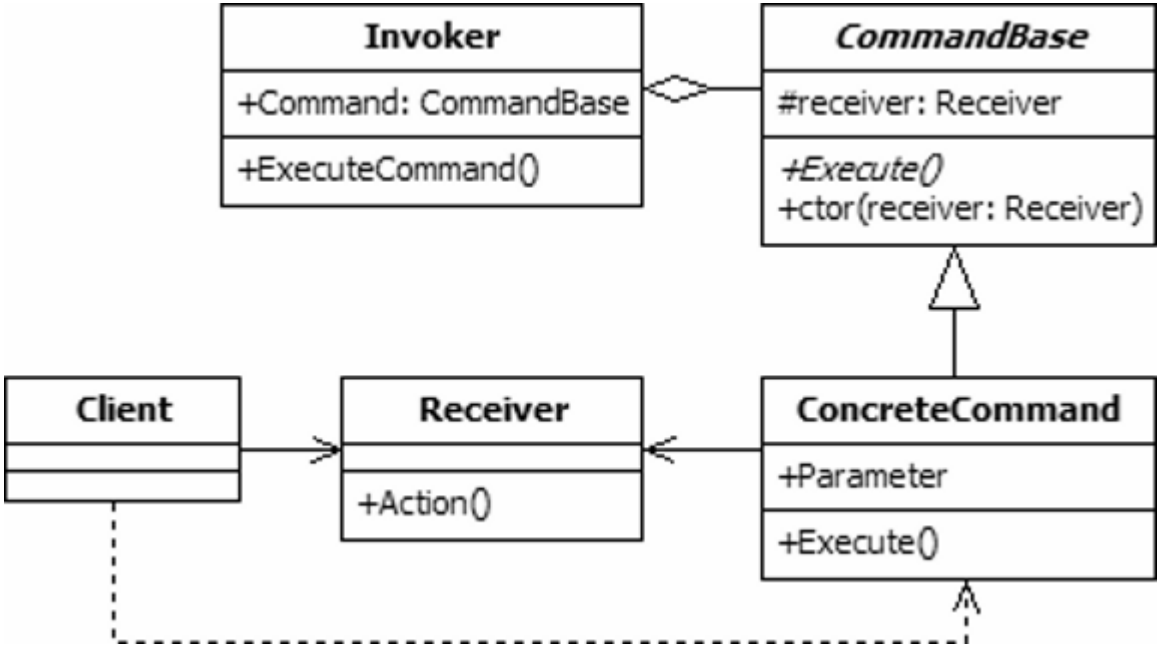
```javascript
/**
 * Space
 */
var space={
  handle: function() {
    var captures;
    if (captures = /^([ \t]+)/.exec(Data.str)) {
      Data.skip(captures);
      return advance();
    }
    return space.__proto__.handle();
  }
};
```

```javascript
var initialize = function(){
  space.prototype = _next;
  _next.__proto__ = last;
  last.__proto__ = dayByName;
  dayByName.__proto__ = monthByName;
  monthByName.__proto__ = timeAgo;
  timeAgo.__proto__ = ago;
  ago.__proto__ = yesterday;
  yesterday.__proto__ = tomorrow;
  tomorrow.__proto__ = noon;
  noon.__proto__ = midnight;
  midnight.__proto__ = night;
  night.__proto__ = evening;
  evening.__proto__ = afternoon;
  afternoon.__proto__ = morning;
  morning.__proto__ = tonight;
  tonight.__proto__ = meridiem;
  meridiem.__proto__ = hourminute;
  hourminute.__proto__ = athour;
  athour.__proto__ = week;
  week.__proto__ = month;
  month.__proto__ = year;
  year.__proto__ = second;
  second.__proto__ = minute;
  minute.__proto__ = hour;
  hour.__proto__ = day;
  day.__proto__ = number;
  number.__proto__ = string;
  string.__proto__ = other;
  other.__proto__ = eos;
};
```

Command

- Decouples boundary objects (GUI) from the implementation of concrete action and separates them from values (entity objects)
- Encapsulates actions into commands so it is easy to work with them (store, pass them as argument)
- Enables easy change of boundary objects (GUI) without any change in functionality

Command - Structure



Client – sets the environment

Invoker – invokes specific command

Command – concrete implementation of command

Receiver – object on which the command is executed

Command

Uses

- Menu (GUI)
- Redo/Undo manager
- Request based system

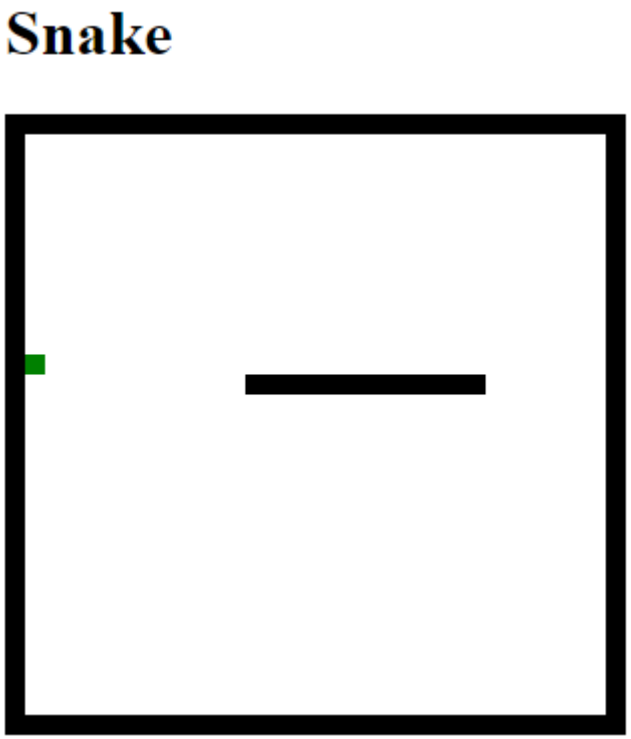
Drawbacks

- design can look cluttered
- maintenance issues with controller

Snake game (retro game)

https://github.com/maryrosecook/retro-games

- JavaScript web browser game
- old snake game



Snake game (retro game) - before

```
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;
  }
},
```

Snake game (retro game) - refactoring

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

var rightCommand = {
  execute: function(obj){
      obj.direction.x = 1;
      obj.direction.y = 0;
  }
};

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

var downCommand = {
  execute: function(obj){
      obj.direction.y = 1;
      obj.direction.x = 0;
  }
};
```

```
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);
  }
},
```

Thank you!
Questions?

Sources

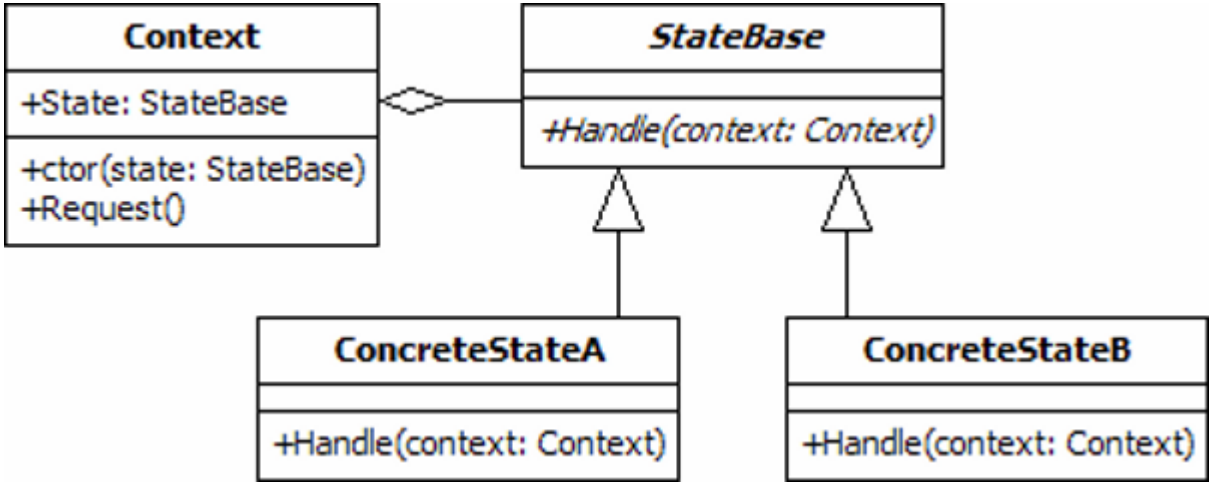
- Erich Gamma et al., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, Massachusetts, USA, 2007.
- Addy Osmani, Learning JavaScript Design Patterns, O‘Reilly, USA, 2012.
- Douglas Crockford, JavaScript: The Good Parts, O'Reilly Media / Yahoo Press, 2008.
- http://www.dofactory.com/ - 09.04.2015

Back up slides

State

- Allows an object to alter its behavior depending on its internal state

Structure



Context – provides clients with an interface and has a Concrete State object

State – interface for encapsulating the behavior associated with a state

Concrete State – implements a behavior for a particular state

State

Uses

- Can be used e.g. for video players that have states like play, pause, buffering, …
- replace huge switch case statements(repeating in methods/functions)

Drawbacks

- increased number of classes