Design patterns in Javascript

22.03.2016

CONFIDENTIAL

Agenda

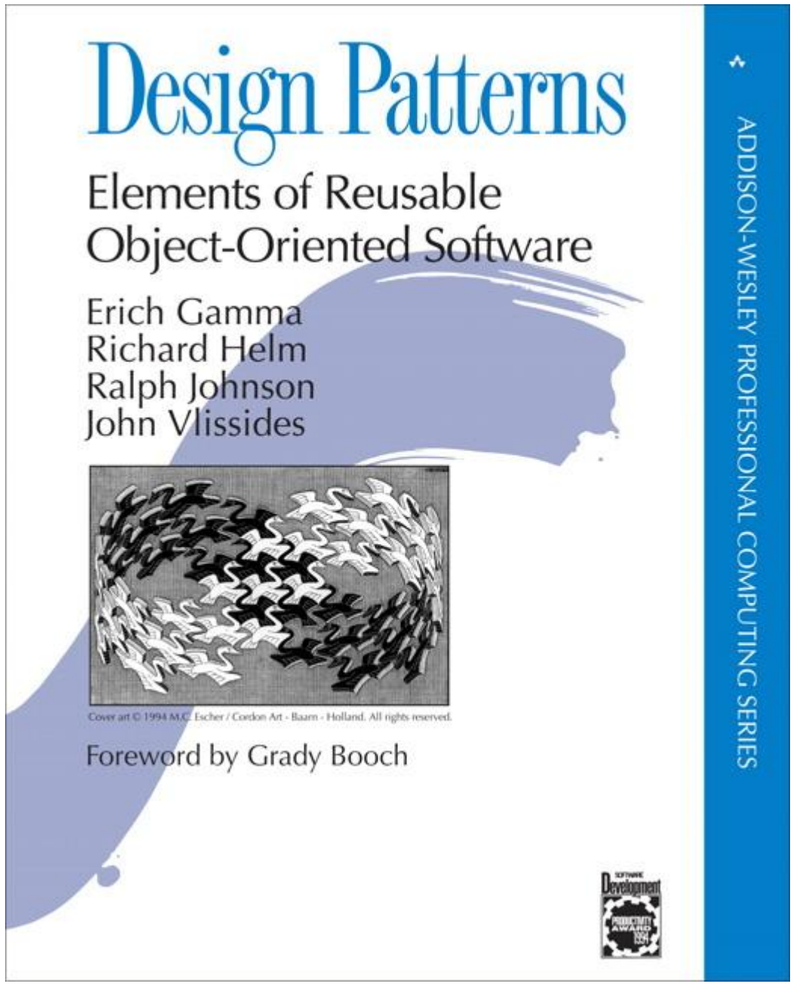
- Why design patterns ?
- Creational patterns: Singleton, Factory, Mixins
- Structural patterns: Adapter, Decorator
- Behavioural patterns: Observer, Command, State

Why design patterns ?

An established, well documented approach to a common problem

- Tested and vetted by a large community
- Wide applicability across many different languages and situations
- Establishes a common vocabulary for devs - more expressive

Design patterns



23 patterns

3 large categories: **Creational**, **Structural**, **Behavioural**

Creational patterns

Creational patterns

Patterns concerned with object creation and initialization

Singleton pattern

Problem statement

Ensure that **one** and **only one** instance of a class exists in the application.

Why ?

To avoid creating multiple expensive objects

To ensure a piece of data & associate behaviour it’s the same when accessed from multiple places

Singleton pattern

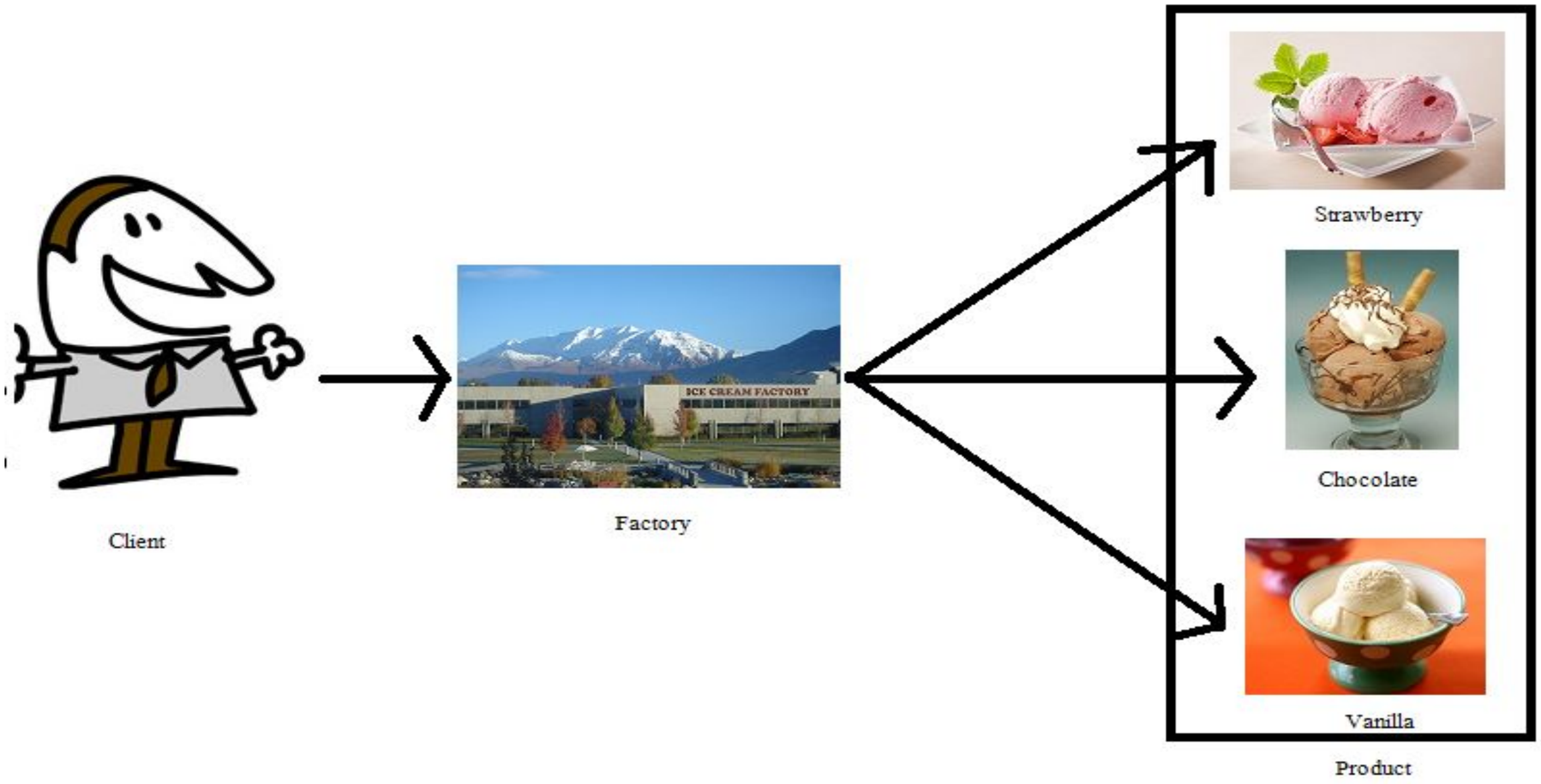
Simplest singleton ever

```
var Logger = {
      dataToLog: [],
      log: function(message, level) {}
}
Logger.log(“I’m a singleton”, “debug”)
```

More elaborate singleton

```
function Logger() {}
Logger._instance = null;
Logger.getInstance = function() {
        if(Logger._instance === null ) {
                Logger._instance = new Logger();
        }
        return Logger._instance;
}
```

Factory pattern



Factory pattern

Problem statement

Create and setup multiple object instances based on dynamic criterias.

Why ?

Avoid repeating a complex setup process for objects in multiple places

Abstract away what object gets created based on dynamic criterias (environment, server received data, application config)

Factory pattern

Assume server sends

products = [{type: “Electronics”, data: {...}}, {type: “Food”, data:{...}}]

VAT is different between Electronics(24%) and Food(9%)

Electronics have a warranty of 2 years. Food do not.

How to model these differences UI side ?

Factory patterns

1. Create 2 object classes: “Electronics” and “Food”
2. Create a ProductFactory that creates one or the other

```
function ProductFactory() {}
ProductFactory.create = function(type, data) {
        if(type === “Electronics”) { return new Electronics(data); }
        if(type === “Food”) { return new Food(data); }
}
```

Mixins

The ability to “mix” into existing objects additional pieces of functionality.

Problem statement

You have multiple, not necessarily related objects, that want to share common code

Why ?

Granular control over what gets shared: don’t share more, don’t share less

No “static” relationship between classes like inheritance

An object => multiple mixins that give functionality to it

Mixins

```
ResizableMixin = {
   resizeHorizontal: function() {},
   resizeVertical: function() {}
}


 MoveableMixin = {
    moveHorizontal: function(dx) {},
    moveVertical: function(dy) {}
 }
```

Mixins - how to use

```
function PanelWidget() {}
function TreeViewWidget() {}
augument(PanelWidget.prototype, ResizableMixin, MoveableMixin)
augument(TreeViewWidget.prototype, ResizableMixin, MoveableMixin)
```

Both widgets are resizable and moveable!

Mixins - how to augument

```
function augument(receivingProto) {
    var giverObjects = Array.prototype.slice.call(arguments, 1);
    for( var i = 0, l = giverObjects.length; i < l; i++ ) {
        var giver = giverObjects[i];
        for( var key in giver ) {
            if(!receivingProto.hasOwnProperty(key)) {
                receivingProto[key] = giver[key];
            }
        }
    }
}
```

Structural patterns

Structural patterns

Concerned with how the objects are composed together.

The one **constant** in software development:

CHANGE!

Decorator pattern

Problem statement

Enhance an object with additional responsibilities at runtime.

Why ?

The set of responsibilities an object has can be configured individually.

“Client code” is unaware if an object has additional responsabilities baked in

Much more flexible than inheritance

Keeps inheritance chain spiriling out of control

Decorator pattern

Create a table component that can be searcheable and filterable.

Decorator pattern - The Table

```
function Table() {}
Table.prototype.drawHeaders = function() { …. }
Table.prototype.drawContent = function() { …. }
```

Decorator pattern

```
function SortableDecorator(table) {
   this._table = table;
}
SortableDecorator.prototype.drawHeaders = function() {
    this._table.drawHeaders();
    addSortingIndicators(this._table, this._onColumnSortClicked);
}
SortableDecorator.prototype._onColumSortClicked = function
(columnName) {
}
```

Decorator pattern

```
function SearcheableDecorator(table) {
     this._table = table;
}
SearcheableDecorator.prototype.drawHeaders = function() {
      this._table.drawHeaders();
      addFilteringDropdowns(this._table, this._onFilter);
}
SearcheableDecorator.prototype._onFilter =  function(columnName) {
}
```

Decorator pattern

Putting it all together

```
var tbl = new SearcheableTable(new SortableTable(new Table
({data:[]}) ) )
```

Adapter pattern

Adapter pattern

Makes 2 different interfaces be compatible with each other.

Adapter pattern

A shopping cart

```
ShoppingCart.prototype.addProduct = function(product) {
// Some logic here
products.push(product);
localStorage.setItem(“products”, products);
}
```

Adapter pattern

“Saving data locally is great, but we need it available on the server! We should store only the recently viewed products in local storage.”

- Project Architect

Adapter pattern

```
ServerPersistenceAdapter.prototype.save = function(item) {
     xhr.send(“POST”,  “/items/”, item);
}
ServerPersistenceAdapter.prototype.delete = function(item) {
    xhr.send(“DELETE”,  “/items/”, item);
}
```

Adapter pattern

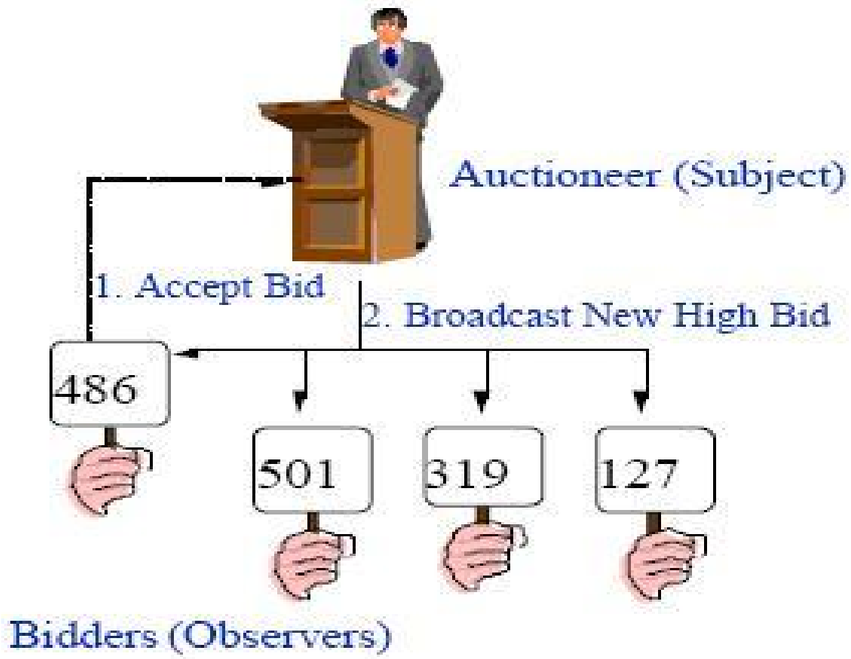
```
LocalPersistenceAdapter.prototype.save = function(item) {
    localStorage.set(“item”, item);
}
LocalPersistenceAdapter.prototype.delete = function(item){
    localStorage.delete(“item”);
}
```

Adapter pattern

```
function ShoppingCart(persistenceLayer) {
  this.persistenceLayer = persistenceLayer;
}
ShoppingCart.prototype.save = function(product) {
  this.persistenceLayer.save(product);
}
var cart = new ShoppingCart( new ServerPersistenceAdapter()
);
```

Behavioural patterns

Observer



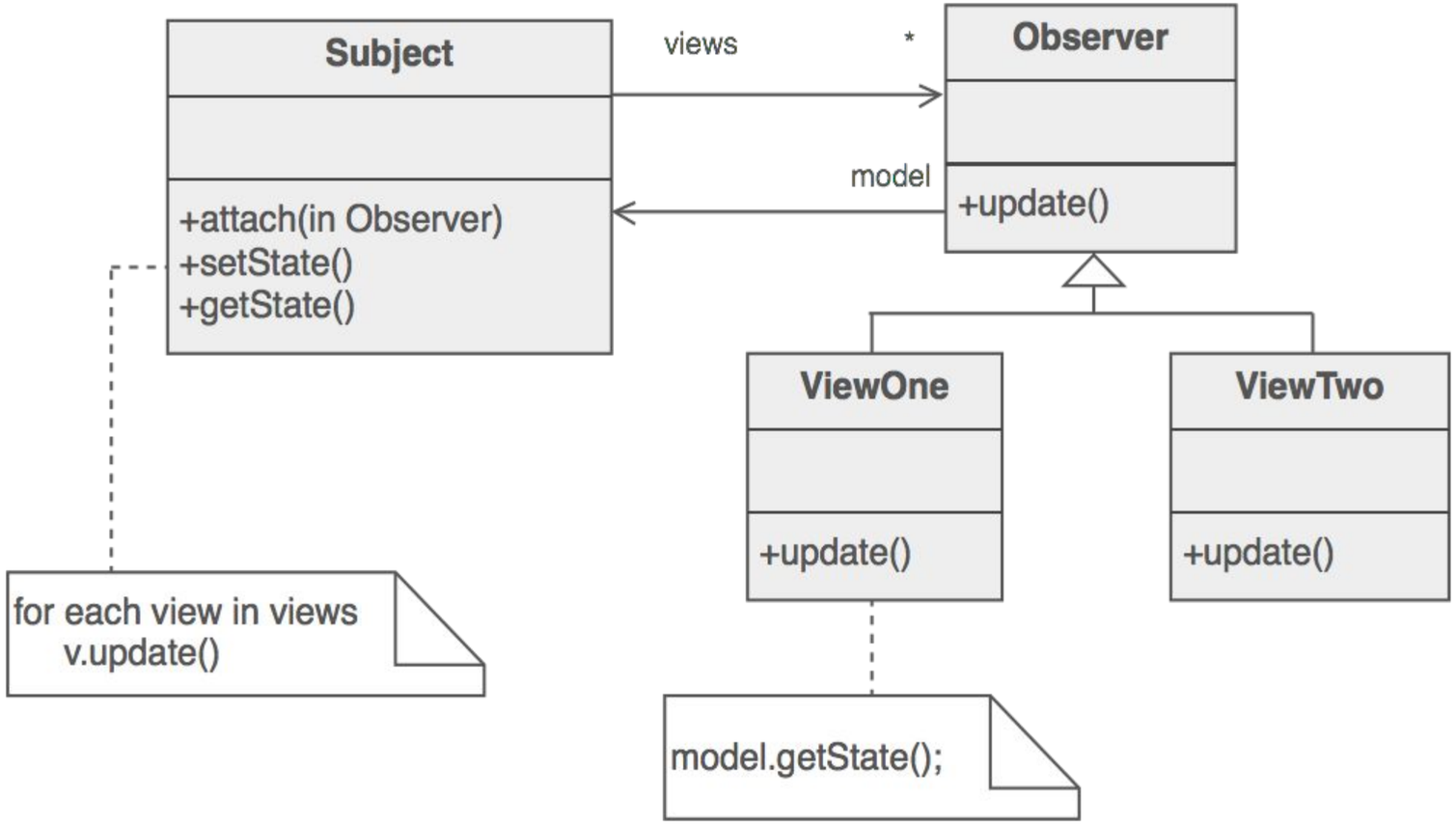
When a bidder at an auction accepts a bid, he or she raises a numbered paddle which identifies the bidder. The bid price then changes and all *Observers* must be notified of the change. The auctioneer then broadcasts the new bid to the bidders.

Behavioral

Observer pattern

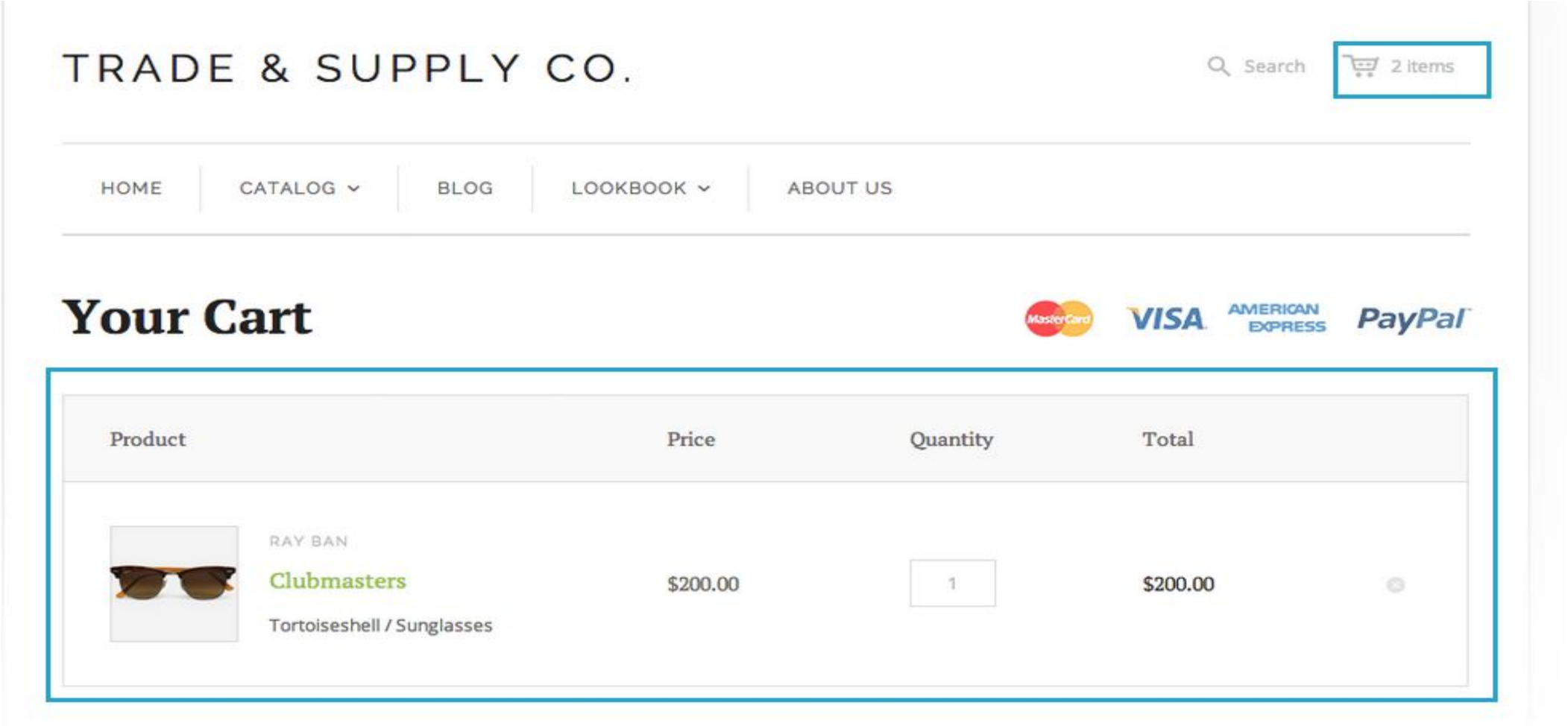
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Observer pattern



Observer pattern

Same data and multiple ways to show it.



Observer pattern

```
function ShoppingCartDataModel() {
    this._items = [
        {"name": "laptop", "price":1000},
        {"name": "ipad", "price":500}
    ];
    this._observers = [];
}
ShoppingCartDataModel.prototype.getItemsCount = function() {
    return this._items.length;
}
ShoppingCartDataModel.prototype.getAllItems = function() {
    return this._items;
}
ShoppingCartDataModel.prototype.register = function(observer) {
    this._observers.push(observer);
}
ShoppingCartDataModel.prototype._notifyObservers = function() {
    for(var i = 0, l = this._observers.length; i < l; i++ ) {
        this._observers[i].update();
    }
}
ShoppingCartDataModel.prototype.addProduct = function(product) {
    this._items.push(product);
    this._notifyObservers();
}
```

Observer pattern

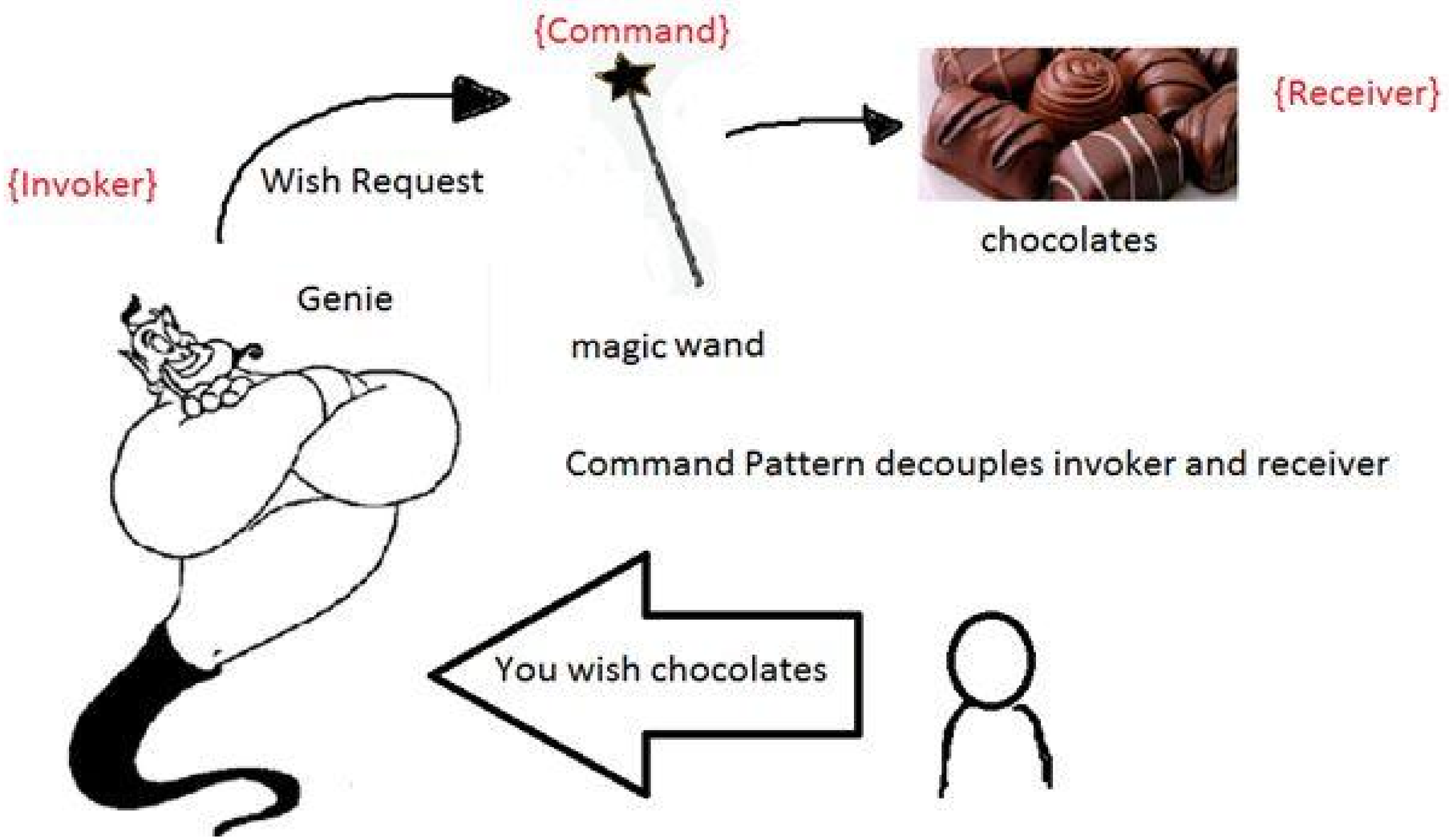
```
function ItemsCountView(shoppingCart) {
    this._shoppingCart = shoppingCart;
}
ItemsCountView.prototype.render = function() {
    document.querySelector(".items-count").innerHTML = this._shoppingCart.getItemsCount();
}
ItemsCountView.prototype.update = function() {
    this.render();
}
function ProductsSummary(shoppingCart) {
    this._shoppingCart = shoppingCart;
}
ProductsSummary.prototype.render = function() {
    var products = this._shoppingCart.getAllItems();
    for(var i = 0, l = products.length; i < l; i++) {
        generateOrderLineHtml(products[i]);
    }
}
ProductsSummary.prototype.update = function() {
    this.render();
}
var shoppingCartModel = new ShoppingCartDataModel();
var countView = new ItemsCountView(shoppingCartModel);
var summary = new ProductsSummary(shoppingCartModel);
shoppingCartModel.addProduct({name: "iphone", price: 800});
```

Command pattern

“Commands are an object-oriented replacement for callbacks.”

- Gang of Four

Command pattern



Command pattern

Encapsulate requests into a class

Expose a common interface for all command classes

```
function RunCommand( player ) {
    this.player = player;
}

RunCommand.prototype.execute = function() {
    this.player.run();
}
```

Command pattern

Create as many commands as needed to encapsulate ***actions*** within the app

```
function FireCommand( player ) {
    this.player = player;
}

FireCommand.prototype.execute = function() {
    this.player.fire();
}
```

Command pattern

Running the commands

```
function handleInput(e, player) {
  if (e.key == 'x') {
      var strength = 10;
      return new FireCommand(player, strength);
  }
  if (e.key === 'j') {
      var speed = 100;
      return new RunCommand(player, speed);
  }
}

function onKeyPressed(e) {
    var command = handleInput(e, currentPlayer);
    command.execute();
}
```

Command pattern

All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

Command pattern

Commands...

can be passed around.

can be invoked at a later time, at the discretion of the caller

decouple the calling object from executing object

can be swapped out easily with another command

easy do to undo/redo

State pattern

A vending machine has different behaviour depending on

cash has been introduced

product is in stock

order has been placed

money are sufficient

…… and many more

How ?

Enter the State Pattern

State pattern - Step 1

Create a *context* class that holds the current state

```
function VendingMachine() {
    this._currentState = new PendingState();
}

VendingMachine.prototype.withdrawMoney = function() {
    this._currentState = this._currentState.withdrawMoney();
}

VendingMachine.prototype.enterMoney = function() {
    this._currentState = this._currentState.enterMoney();
}

VendingMachine.prototype.selectProduct = function() {
    this._currentState = this._currentState.selectProduct();
}

VendingMachine.prototype.getProduct = function() {
    this._currentState = this._currentState.getProduct();
}
```

State pattern - Step 2

Define a state class *for each* conceptual state in which the context can be

All classes have the *same* interface

Outside world interacts only with the context(eg VendingMachine)

The next state can be determined

By each state class

By context class depending on the action performed

State pattern - Step 2

```
function PendingState() {

}

PendingState.prototype.enterMoney = function() {
    return new SelectProductsState();
}

PendingState.prototype.withdrawMoney = function() {
    display("Please enter amount...");
}
```

State pattern - Step 2

```
function SelectProductsState() {
    this._money = 0;
}

SelectProductsState.prototype.enterMoney = function(amount) {
    this._money += amount;
}

SelectProductsState.prototype.withdrawMoney = function() {
    return new PendingState();
}
```

State pattern - benefits

A large, monolithic object, is broken down in small size pieces

The context is able to dramatically change behaviour dynamically

Strict control on how the object behaves

State pattern

State objects are usually singletons

The context object can be passed in to State objects

A base State class can define common behaviour for all states

Module pattern

Not included in Gof 23 patterns

A way to minimize global scope access

A way to expose the public API of a class and hide the rest

Module pattern structure

```
var Logger = (function() {
    var _messages = [];
    function Logger() {
    }
    Logger.prototype.log = function(message) {
        _messages.push(message);
    }
    return Logger;
})();
```

Module pattern variations

```
(function(NS) {
    var _messages = [];
    function Logger() {
    }
    Logger.prototype.log = function(message) {
        _messages.push(message);
    }
    NS.Logger = Logger;
})(window);
```

Q & A