JS Patterns

UA Resource Development Unit

2020

AGENDA

1 **Introduction to design patterns**

2 **Creational Patterns**

3 **Structural Patterns**

4 Behavioral patterns

5 DRY, KISS and other

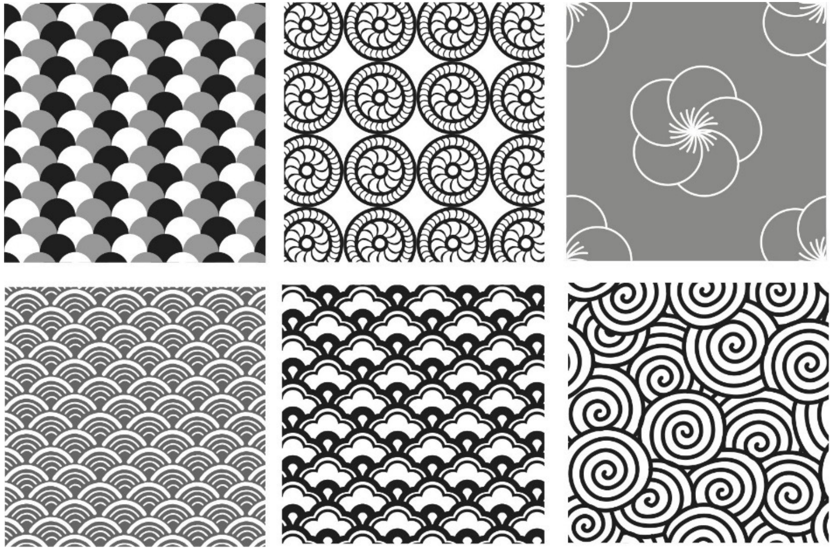
INTRODUCTION TO DESIGN PATTERNS

DESIGN PATTERNS

Each pattern it’s a **solution** for the problem in software programming. Pattern solves this problem millions and billions time in effective way.

What is pattern?

- A pattern is a reusable solution to a commonly occurring problem within a given context in software design.
- Patterns are as templates for how we solve problems—ones that can be used in quite a few different situations
- Patterns are proven solutions (*not* exact solutions)



History

- Patterns originated as an architectural concept by Christopher Alexander (1977/78).
- In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming.
- Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the so-called "Gang of Four" (Gamma, Helm, Johnson and Vlissides .), which is frequently abbreviated as "GoF".

Types of design patterns

- Creational

  Deal with initializing and configuring classes and objects

- Structural

  Deal with decoupling interface and implementation of classes and objects

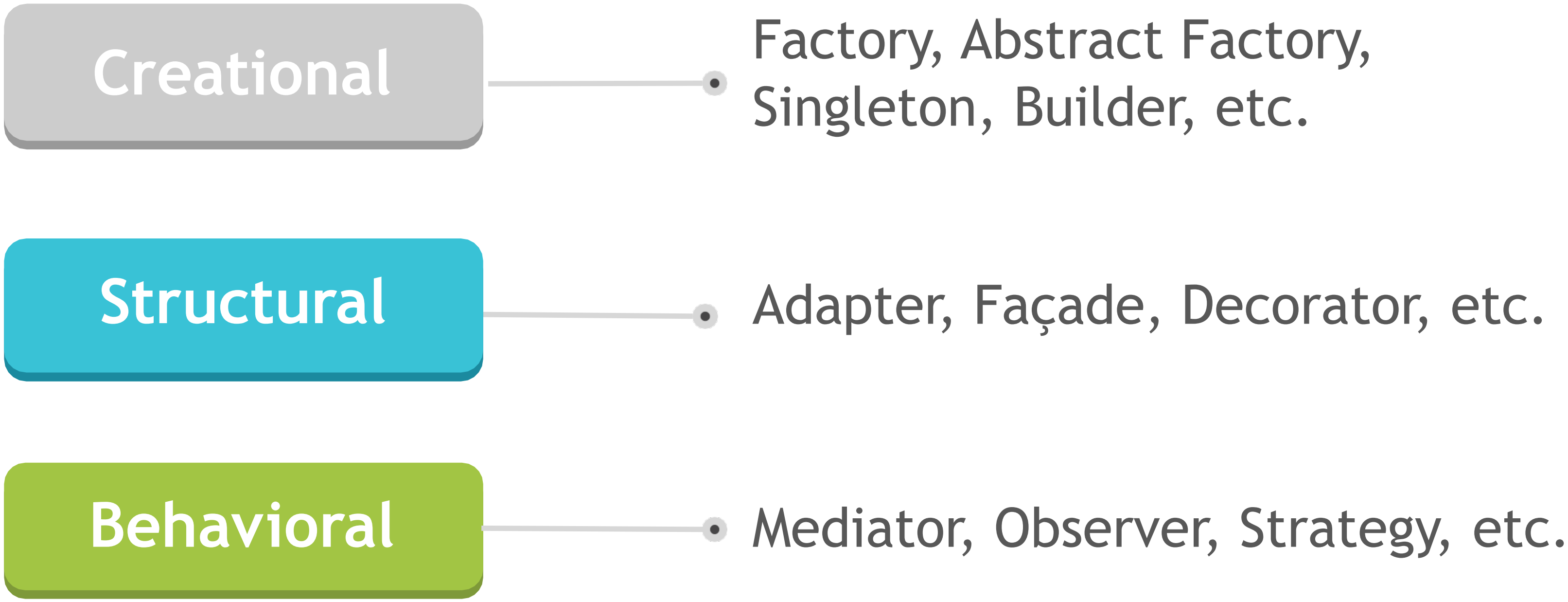
  Composition of classes or objects

- Behavioral

  Deal with dynamic interactions among societies of classes and objects

  Distribute responsibility

Types of Design patterns (cont.)



Creational — Factory, Abstract Factory, Singleton, Builder, etc.

Structural — Adapter, Façade, Decorator, etc.

Behavioral — Mediator, Observer, Strategy, etc.

Module

WHEREFORE

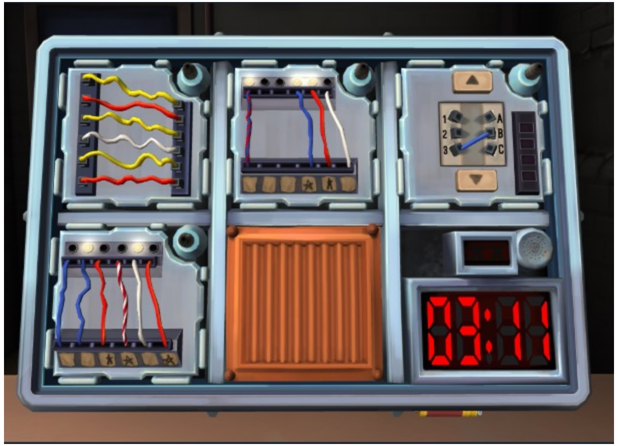
- In JavaScript, the Module pattern is used to further emulate the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope.

HOWTO

- Module is a combination of 2 simple patterns:

  self-executing function

  private variables and methods



Module-object

```
let module = (function () {
        let counter = 0;
        function increaseCounter() {
                counter++;
        }
        function resetCounter() {
                console.log(counter);
                counter = 0;
        }
        return {
                increase: increaseCounter,
                reset: resetCounter
        };
})();
// Usage:
// Increment our counter
module.increase();
// Check the counter value and reset
// Outputs: 1
module.reset();
```

Module-constructor

```
let Neuron = (function () {
    const cell = function () {};
    function say(message) {
        console.log(message);
    }
    cell.prototype = {
        migrate: function() {
            say('travelling’);
        },
        learn: function(subj) {
            say(`studying ${subj}`);}
    };
    return cell;
})();
// Usage:
// Creates new neuron instance
let brainCell = new Neuron();
// Outputs: travelling
brainCell.migrate();
// Outputs: studying math
brainCell.learn('math');
```

Module. Pros/Cons

- Advantages

  support of private data

  splitting the project into logical blocks

  reuse

- Disadvantages

  inability to test private data

  dependencies need to be handled manually

  modules structure organization (ES6 modules)

CREATIONAL PATTERNS

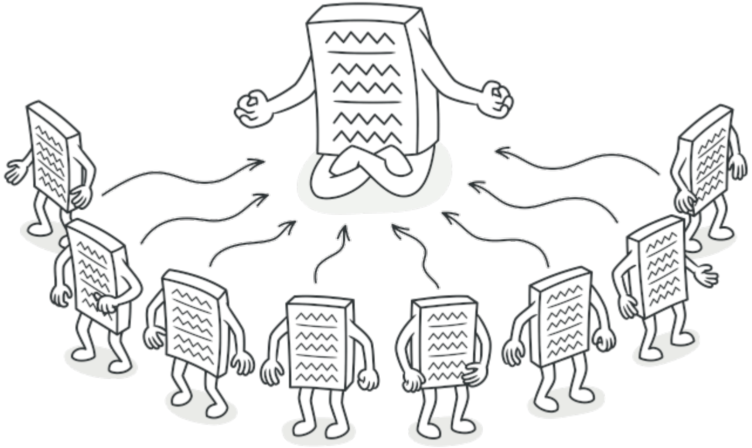
Singleton

WHEREFORE

- Provide single instance of class. When you try to create another instance, the program should receive an item that has already been created.

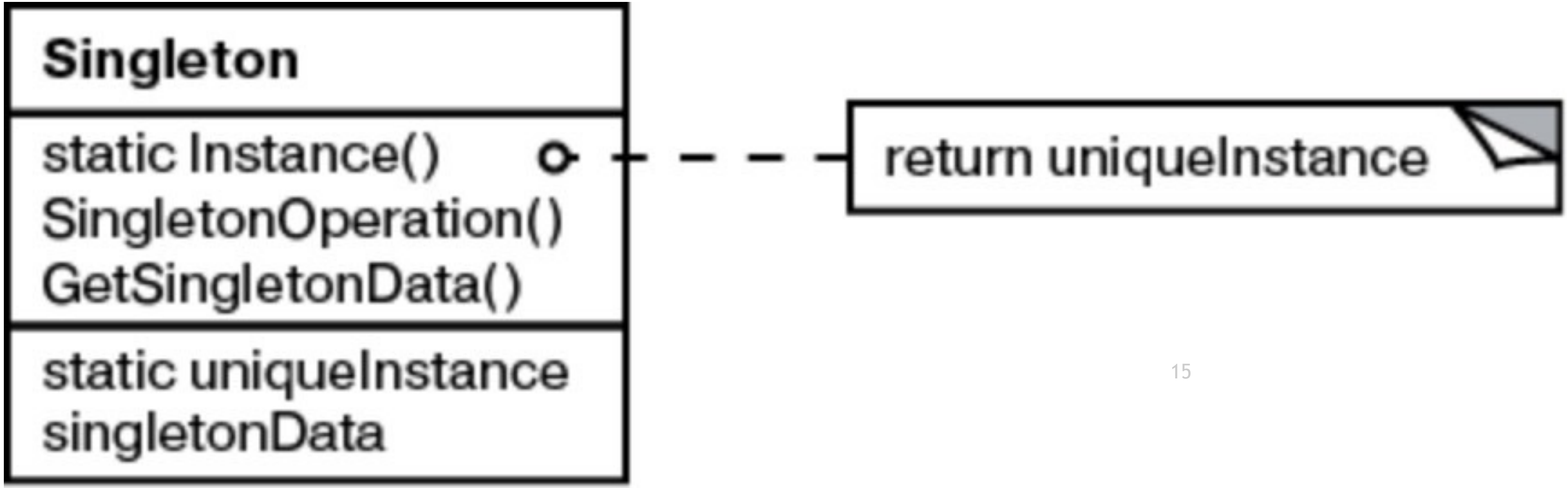
HOWTO

- Using module pattern



SINGLETON

Singleton 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 need to instantiate the object of the class.



Singleton

```
let facebookPage = (function () {
        let user;
        function createInstance() {
                let object = new Object(”I am a facebook user");
                return object;
        }
        return {
                goToMyPage: function () {
                                // creates instance if it doesn't exist
                                if (!user) {
                                                user = createInstance();
                                }
                                return user;
                          }
        };
})();
// Usage:
// Creates new instance
let user1 = facebookPage. goToMyPage();
let user2 = facebookPage. goToMyPage();
// Outputs: true
console.log(user1 === user2);
```

JAVASCRIPT

```
let instance = null;

export default class Singleton {
    constructor() {
        if (instance) { return instance; }
        this.foo = 'hello';
        this.bar = 'world';

        instance = this;

        return this;
    }
    setFoo(foo) {
        this.foo = foo;
    }
    setBar(bar) {
        this.bar = bar;
    }
}
let singl1 = new Singleton();
let singl2 = new Singleton();
singl1 === singl2 // true
```

USE SINGLETON WHEN

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.
- Share state across the application

Singleton. Pros/Cons

- Advantages

  controlled access to a single instance

- Disadvantages

  global access

  problems with testing are possible

  often there is a desire to expand singleton

Factory

WHEREFORE

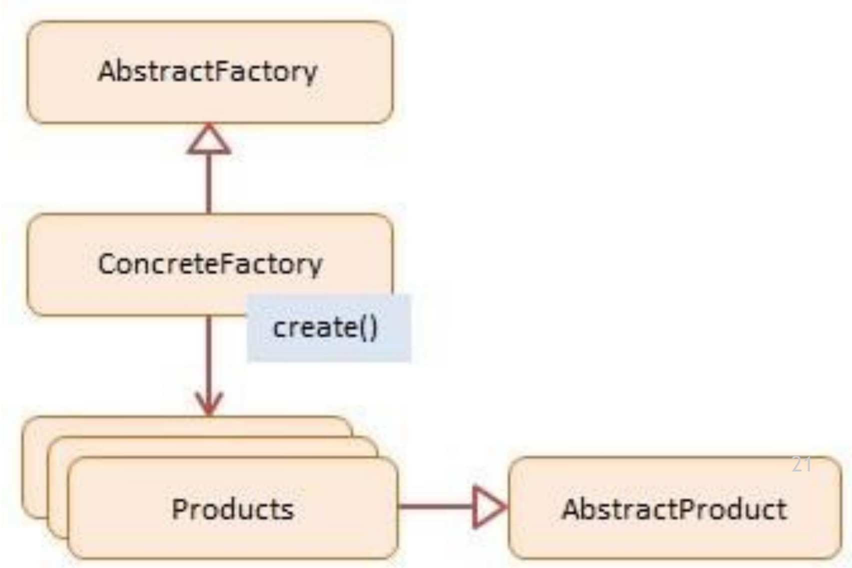
- Factory can provide a generic interface for creating objects, where we can specify the type of factory object we wish to be created.

HOWTO

- Using the factory method, which will select the appropriate constructor.

FACTORY

In **Factory pattern**, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



JAVASCRIPT

```
let vehicles = new
Map();
export default class Factory {
    registerVehicle(type, Vehicle) {
        if (!vehicles.has(type)) {
            vehicles.set(type, Vehicle);
        } else {
            throw new Error(`Vehicle with type ${type} already exists!`);
        }
    }

    createVehicle(type, customizations) {
        if (!vehicles.has(type)) {
            throw new Error(`Vehicle with type ${type} is not supported!`);
        }

        let Vehicle = vehicles.get(type);

        return (typeof Vehicle === 'function' ? new Vehicle(customizations) : null);
    }
}
```

USE FACTORY WHEN

- You want to encapsulate a group of individual factories with a common goal
- When we need to easily generate different instances of objects depending on the environment we are in
- When our object or component setup involves a high level of complexity
- You need to separate the details of implementation of a set of objects from their general usage

Build-in Factory

```
var o = new Object(),
n = new Object(1),
s = new Object(‘1’),
b = new Object(true);
// test
o.constructor === Object; // true
n.constructor === Number; // true
s.constructor === String; // true
b.constructor === Boolean; // true
```

Factory. Example part 1

```
// parent constructor
function CarMaker() {}
// a method of the parent
CarMaker.prototype.drive = function () {
        return `Go, I have ${this.doors} doors`;
};
// the static factory method
CarMaker.factory = function (type) {
        let constr = type;
        let newcar;
        if (typeof CarMaker[constr].prototype.drive !== “function”) {
                CarMaker[constr].prototype = new CarMaker();
        }
        newcar = new CarMaker[constr]();
        return newcar;
};
```

Factory. Example part 2

```
// define specific car makers
CarMaker.Compact = function () {
                this.doors = 4;
};
CarMaker.Convertible = function () {
                this.doors = 2;
};
// Usage:
// objects creating
let corolla = CarMaker.factory(‘Compact’);
let suzuki = CarMaker.factory(‘Convertible’),
corolla.drive(); // “Go, I have 4 doors”
suzuki.drive(); // “Go, I have 2 doors”
```

Factory. Pros/Cons

- Advantages

  allows you to make the code more flexible

  provides a single interface for creating objects

- Disadvantages

  difficult to add support for new types of objects

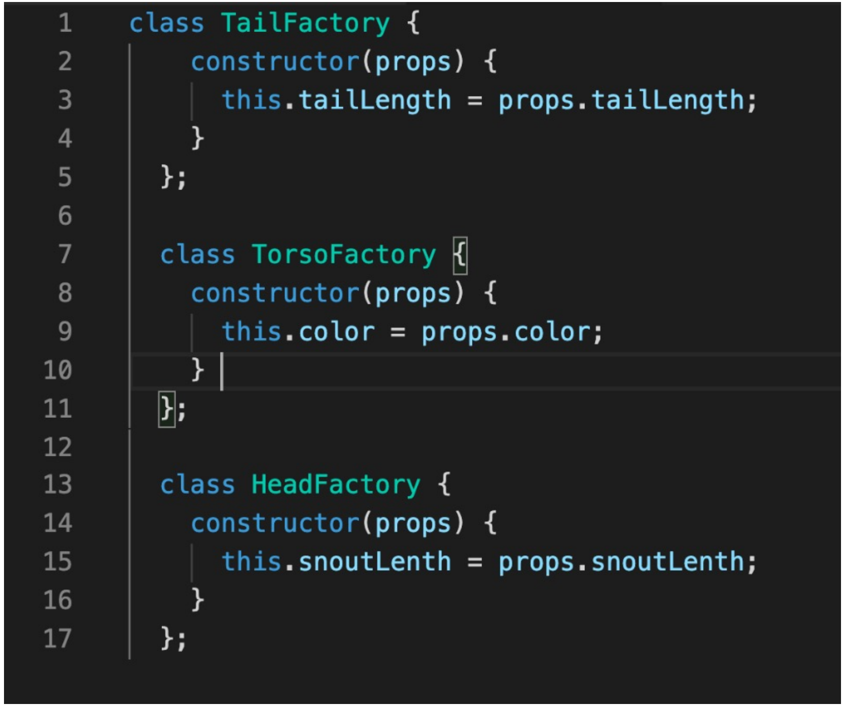
Patterns: Fabric, Abstract Fabric

Organizing code is going to save us from a lot of pain. Using the features of Object Oriented programming, we can employ certain design patterns to achieve better readability, reduce redundancy and create abstractions, if needed. One such pattern is the factory pattern.

The factory pattern is a type of Object Oriented pattern which follows the DRY methodology. As the name suggests, object instances are created by using a factory to make the required object for us.

Patterns: Fabric, Abstract Fabric

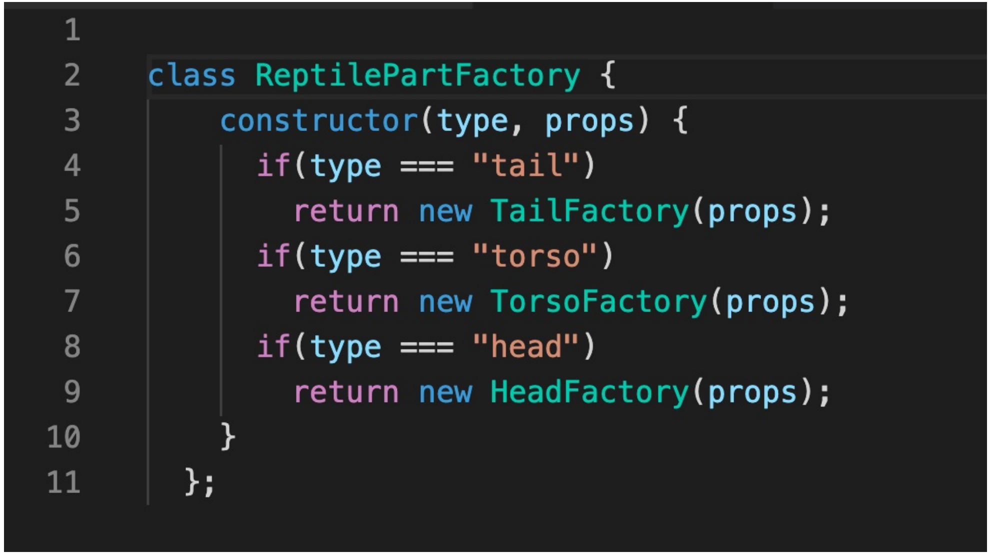
Let’s have a look at a very simple example of using the factory pattern to assemble an alligator object. To do that we first need to make factories that create the alligator parts for us:



```
class TailFactory {
  constructor(props) {
   this.tailLength = props.tailLength;
  }
 };

 class TorsoFactory {
  constructor(props) {
   this.color = props.color;
  }
 };

 class HeadFactory {
  constructor(props) {
   this.snoutLenth = props.snoutLenth;
  }
 };
```

Patterns: Fabric, Abstract Fabric

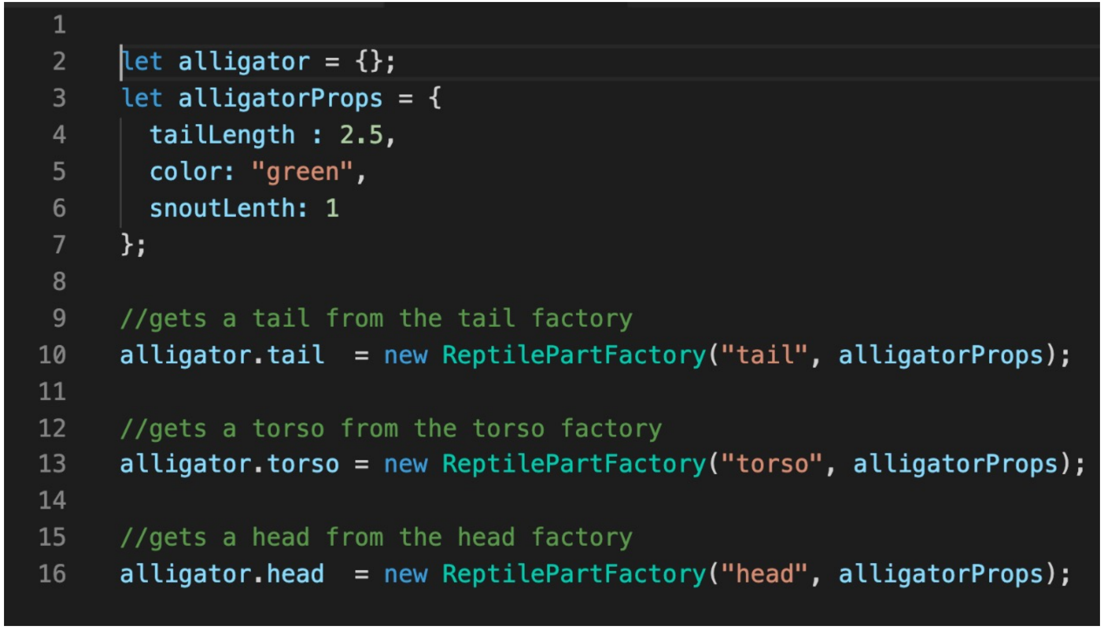
Now, we create a class that acts as an intermediary between the actual factories classes and the user. Let’s call this the ReptilePartFactory:



```

class ReptilePartFactory {
    constructor(type, props) {
      if(type === "tail")
        return new TailFactory(props);
      if(type === "torso")
        return new TorsoFactory(props);
      if(type === "head")
        return new HeadFactory(props);
    }
  };
```

Patterns: Fabric, Abstract Fabric

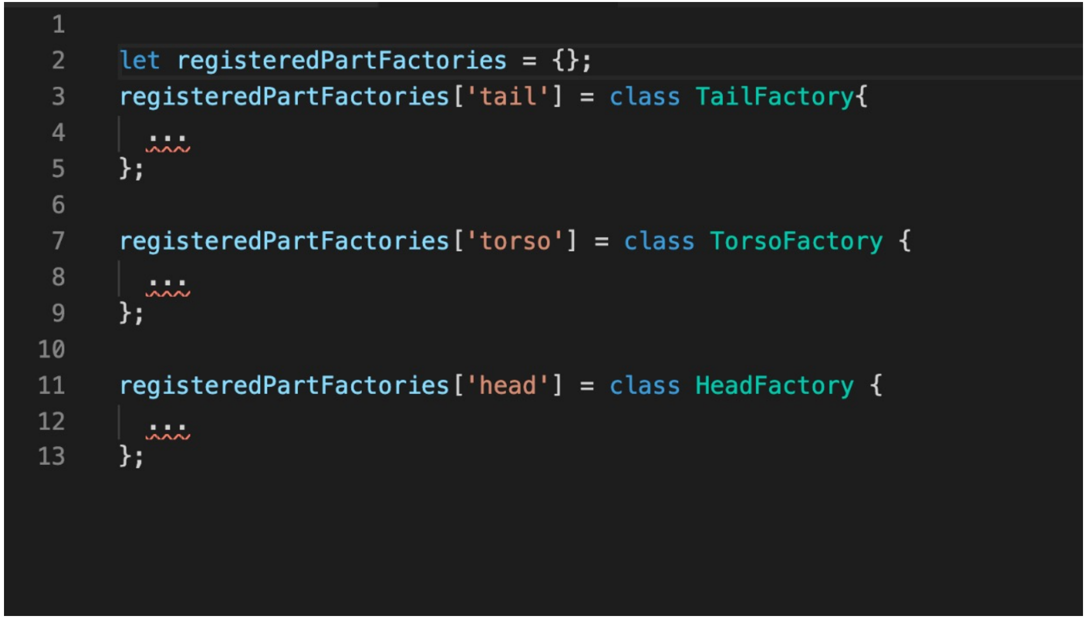
Let’s go ahead and assemble the actual alligator now and use the ReptilePartFactory to get the required parts for us:



```

let alligator = {};
let alligatorProps = {
  tailLength : 2.5,
  color: "green",
  snoutLenth: 1
};

//gets a tail from the tail factory
alligator.tail  = new ReptilePartFactory("tail", alligatorProps);

//gets a torso from the torso factory
alligator.torso = new ReptilePartFactory("torso", alligatorProps);

//gets a head from the head factory
alligator.head  = new ReptilePartFactory("head", alligatorProps);
```

Patterns: Fabric, Abstract Fabric

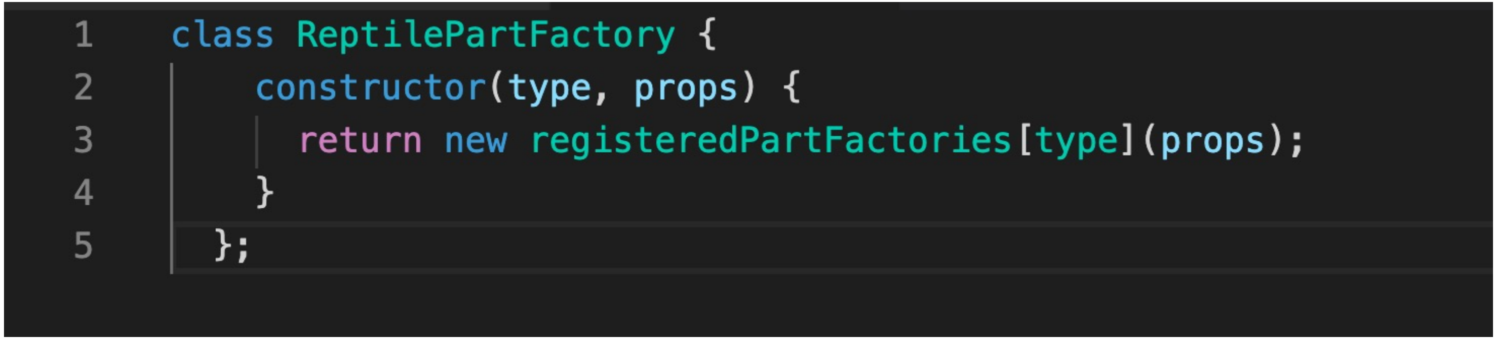
How about we store the factory classes in an object and call the required part factory by using the part we want as the key? First we’d have to register the factories, it’d be as simple as:



```

let registeredPartFactories = {};
registeredPartFactories['tail'] = class TailFactory{
  ...
};

registeredPartFactories['torso'] = class TorsoFactory {
  ...
};

registeredPartFactories['head'] = class HeadFactory {
  ...
};
```

Patterns: Fabric, Abstract Fabric

And now, the abstract layer can call the factories like this:



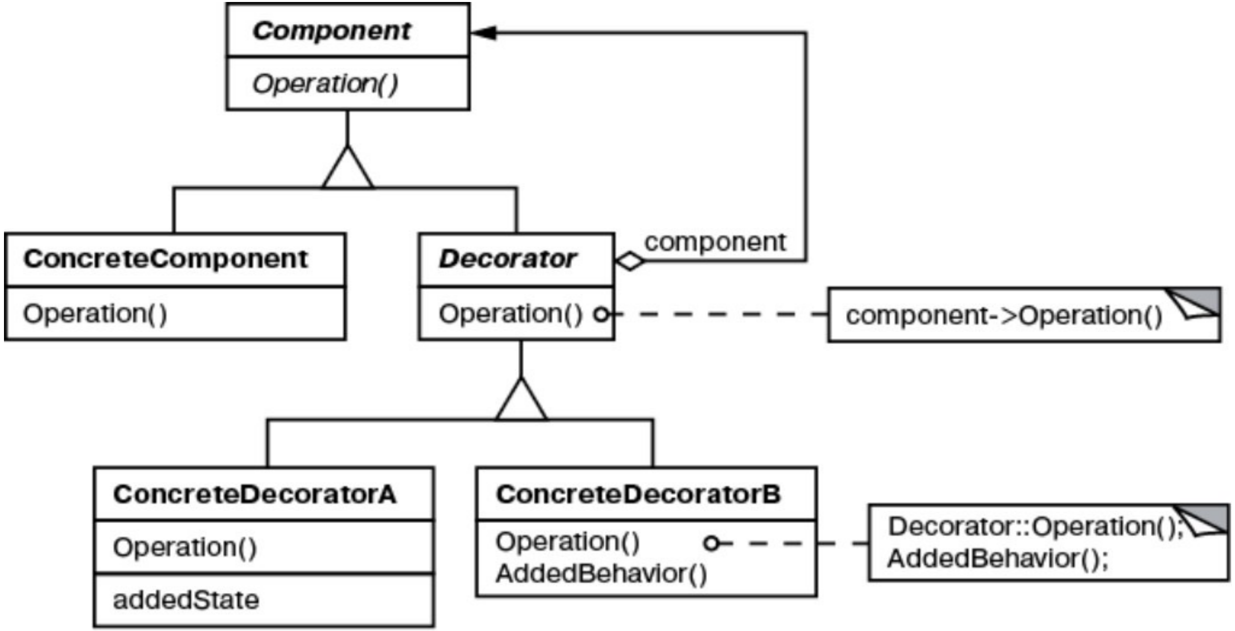
```
class ReptilePartFactory {
    constructor(type, props) {
      return new registeredPartFactories[type](props);
    }
  };
```

This approach is much cleaner and allows to expand our factories without affecting code in the ReptilePartFactory.

STRUCTURAL PATTERNS

DECORATOR

Decorator pattern allows a user to add new functionality to an existing object without altering its structure.



```
// The constructor to
decorate
export default class MacBook {
  constructor() {
    this.price = 997;
    this.screen = 11.6;
  }

  cost() {
    return this.price;
  }

  screenSize() {
    return this.screen;
  }
}
```

```
import MacBook from './macbook';

let decorator = (() => {
  const memory = function(macbook) {
    const v = macbook.cost();
    macbook.cost = () => v + 75;
  };
  const engraving = function(macbook) {
    const v = macbook.cost();
    macbook.cost = () => v + 200;
  };
  const insurance = function(macbook) {
    const v = macbook.cost();
    macbook.cost = () => v + 250;
  };
  return {
    decorate(macbook) {
      memory(macbook);                // Decorator 1
      engraving(macbook); // Decorator 2
      insurance(macbook); // Decorator 3
    }
  };
})();

let macbook = new MacBook();
macbook.cost(); // 997

decorator.decorate(macbook);
macbook.cost(); // 1522
```

USE DECORATOR WHEN

- You need sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

- Rather than sub-classing, we add (decorate) properties or methods to a base object so it's a little more streamlined.

- We need to modify existing systems where we wish to add additional features to objects without the need to heavily modify the underlying code using them

Facade

WHEREFORE

- This pattern provides a convenient higher-level interface to a larger body of code, hiding its true underlying complexity.

HOWTO

- Creating set of facade methods and joining them in one place

Facade Example

```
let pageFacade = {
    updateMenu: function() {
        loadData();
        resizeColumn();
        updateCounter();
        setLog();
    },
    doSmthElse: function() {
        // a lot of methods
    }
};
// Usage:
// Instead of all functions invocation
pageFacade.updateMenu();
```

Facade. Pros/Cons

Advantages

- easy access to a complex system
- resistance to changes

Disadvantages

- It’s not always obvious what's going on in a certain method
- methods can be duplicated

Conclusions

Benefits of Design Patterns

Patterns can be easily reused.

Patterns can make the system more transparent.

Patterns help improve developer communication

Patterns help ease the transition to Object Oriented technology

Drawbacks of Design Patterns

Patterns do not lead to direct code reuse

Teams may suffer from pattern overload

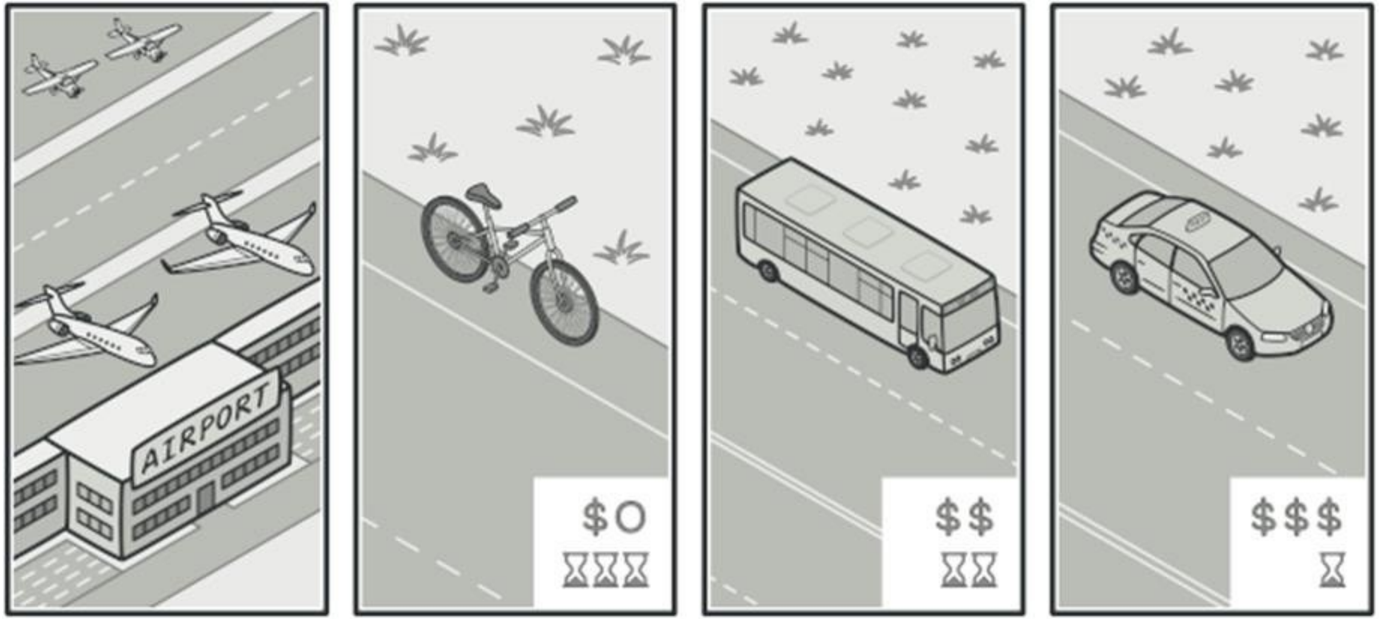
Patterns are validated by experience and discussion rather than by automated testing

Integrating patterns is a human-intensive activity

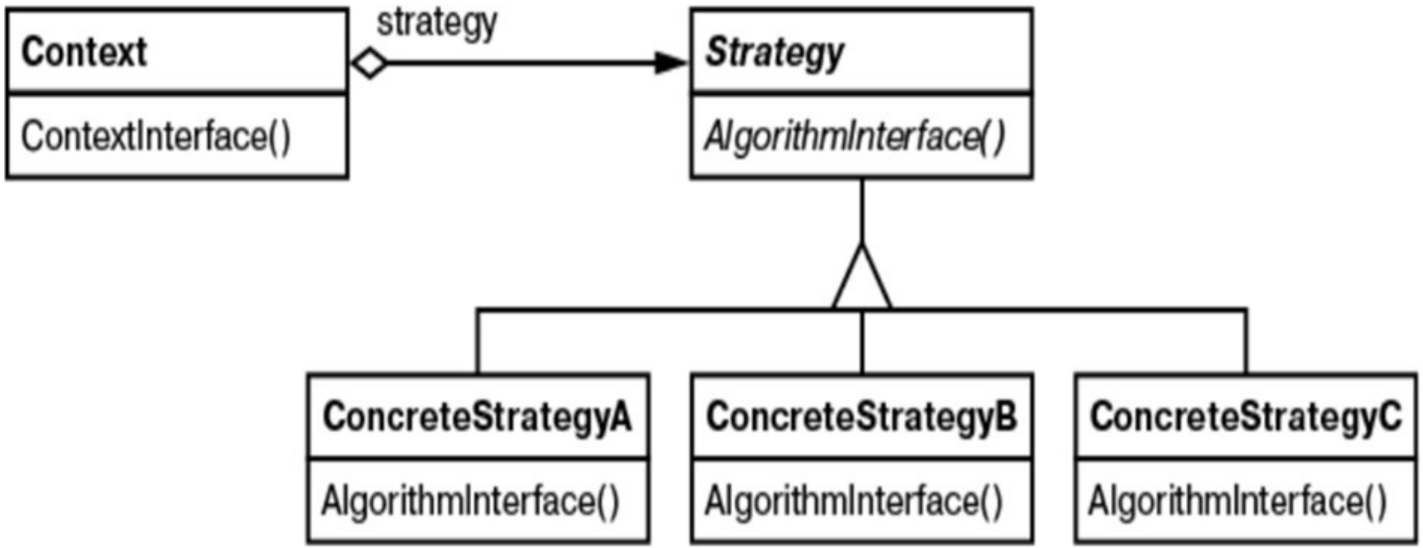
BEHAVIOURAL PATTERNS

STRARTEGY

In **Strategy pattern**, we create objects which represent various strategies and a context object whose behaviour varies as per its strategy object. The strategy object changes the executing algorithm of the context object.



STRARTEGY


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Strategy Example

```
class Strategy {
    execute() {
        throw new Error('Strategy#execute needs to be overridden.');
    }
}

class GreetingStrategy extends Strategy {…}
class PoliteGreetingStrategy extends Strategy {…}
class FriendlyGreetingStrategy extends Strategy {…}

const makeGreet = strategy => strategy.execute;

const simpleGreet = makeGreet(new GreetingStrategy());
const friendlyGreet = makeGreet(new FriendlyGreetingStrategy());
const politeGreet = makeGreet(new PoliteGreetingStrategy());

simpleGreet(); //=> 'Hello, Goodbye.'
politeGreet(); //=> 'Welcome sir, Goodbye.'
friendlyGreet(); //=> 'Hey, Goodbye.'
```

STRATEGY USAGE

- When you need to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

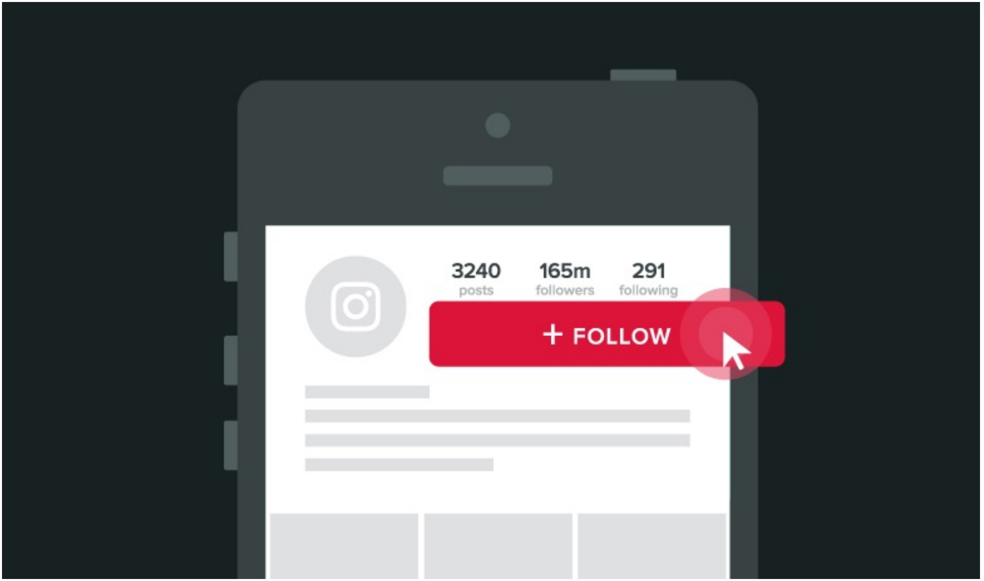
Observer

WHEREFORE

- This is where the objects in a system may subscribe to other objects and be notified by them when an event of interest occurs

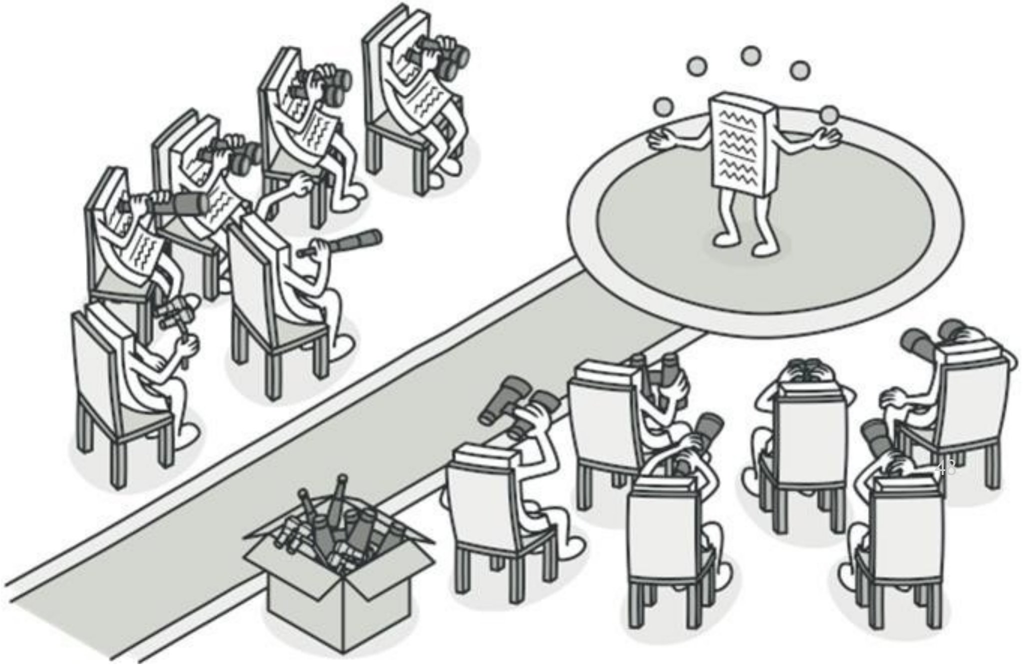
HOWTO

- Using javascript events

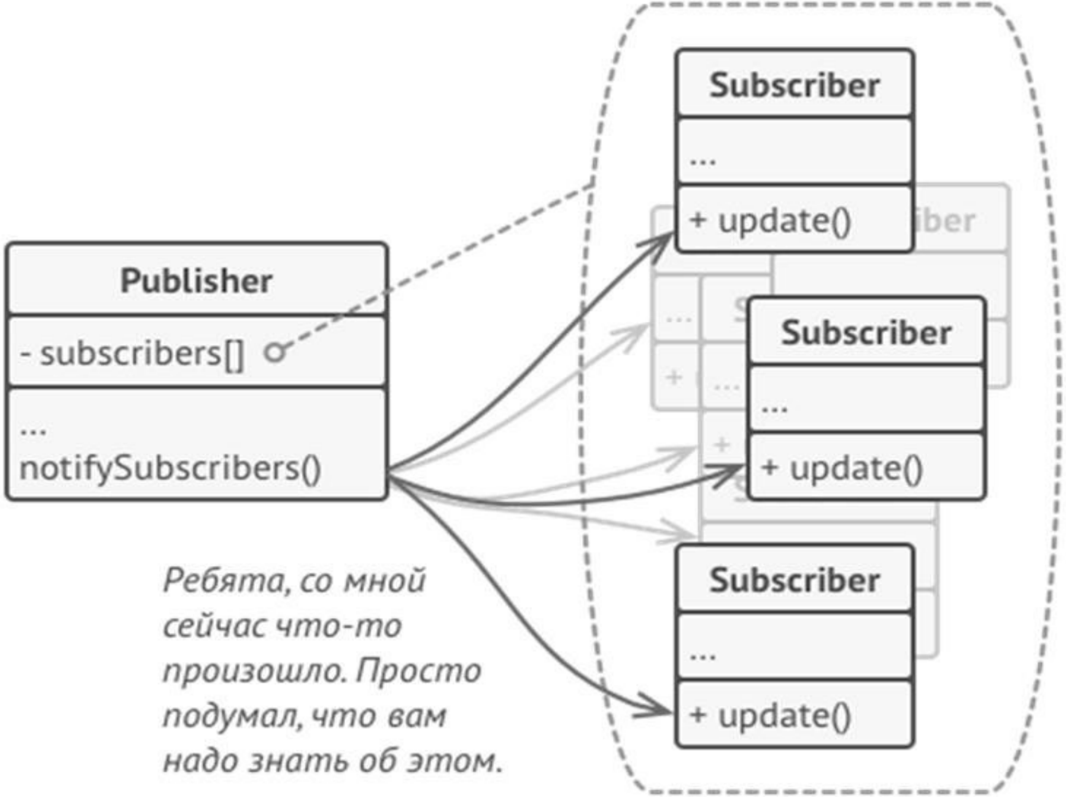


OBSERVER

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically.



OBSERVER



Ребята, со мной сейчас что-то произошло. Просто подумал, что вам надо знать об этом.

JAVASCRIPT

```
export default class Click {
  constructor() {
    this.handlers = [];
  }

  subscribe(observer) {
    this.handlers.push(observer);
  }

  unsubscribe(observer) {
    const index = this.handlers.indexOf(observer);
    if (index !== -1) {
      this.handlers.splice(index, 1);
    }
  }

  notify(opts) {
    this.handlers.forEach(observer => {
      if (typeof observer.update === 'function') {
        observer.update.call(observer, opts);
      }
    });
  }
}
```

```
import Click from './click';

const observer1 = {
  update(args) {
    // args from notify method are
    // accessed here
    console.log('pressed!');
  }
};

const observer2 = {
  update(args) {
   console.log('pressed too!');
  }
};

const click = new Click();

click.subscribe(observer1);
click.subscribe(observer2);

click.notify({ some: 'parameter' });
// => pressed!, pressed too!

click.unsubscribe(observer1);
click.notify(); // => pressed too!
```

OBSERVER USAGE

- When building web apps you end up writing many event handlers
- Event handlers are functions that will be notified when a certain event fires
- These notifications optionally receive an event argument with details about the event (for example the x and y position of the mouse at a click event)

Observer Example part 1

```
class EventObserver {
        constructor () {
                this.observers = []
        }
        subscribe (fn) {
                 this.observers.push(fn)
        }
         unsubscribe (fn) {
                 this.observers = this.observers.filter(subscriber => subscriber !== fn)
        }
         broadcast (data) {
                this.observers.forEach(subscriber => subscriber(data))
        }
 }
const observer = new EventObserver()
observer.subscribe(data => { console.log('subscribe for module 1 fired', data) })
observer.subscribe(data => { console.log('subscribe for module 2 fired', data) })
 observer.broadcast({someData: 'hello'})
```

Observer Example part 2

```
const blogObserver = new EventObserver()
 const textField = document.querySelector('.textField’)
 const countField = document.querySelector('.countField’)

blogObserver.subscribe(text => {
                        console.log('broadcast catched’)
 })

textField.addEventListener('keyup', () => {
                        blogObserver.broadcast(textField.value)
 })

blogObserver.subscribe(text => {
                        countField.innerHTML = text
 })
```

Observer. Pros/Cons

- Advantages

  weakens the relationship between objects

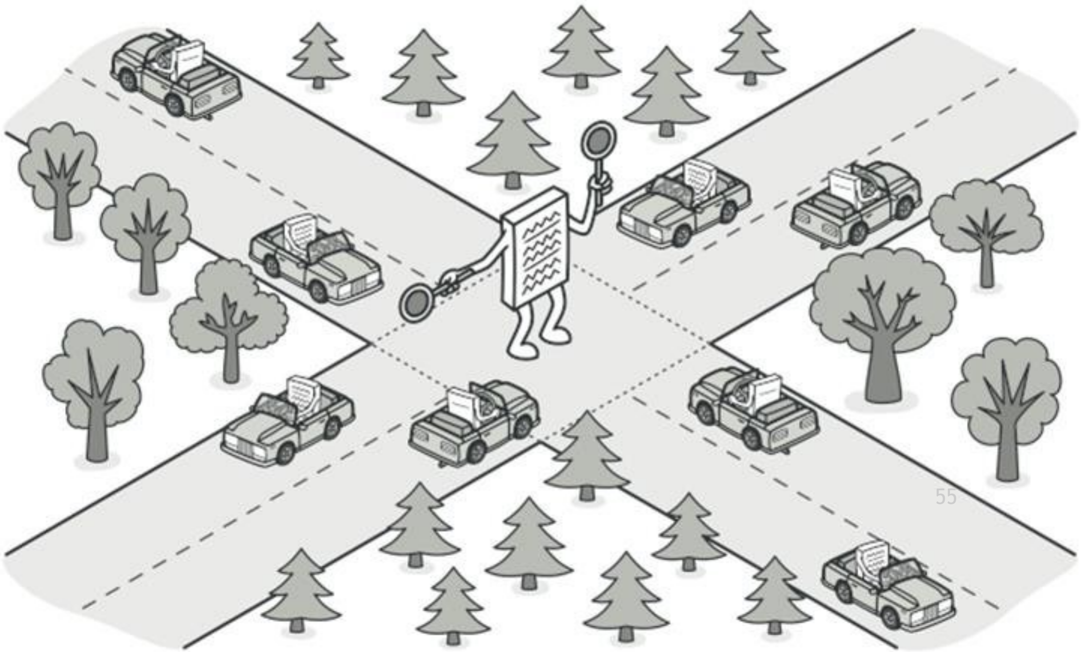
  helps to simplify objects

- Disadvantages

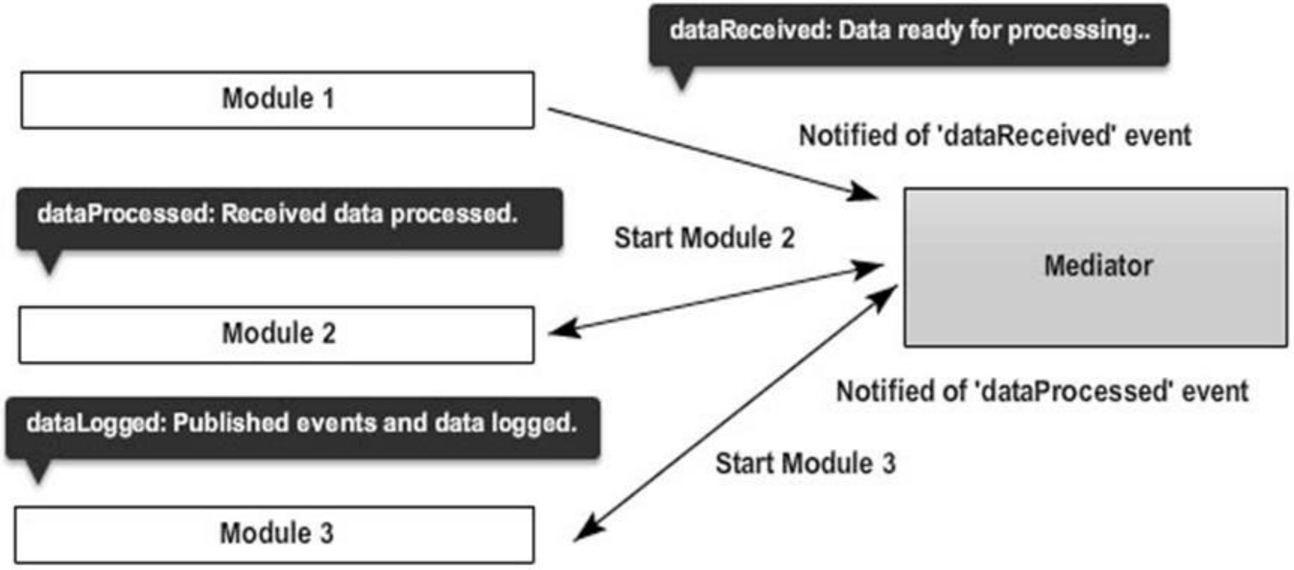
  system is getting less transparent

MEDIATOR

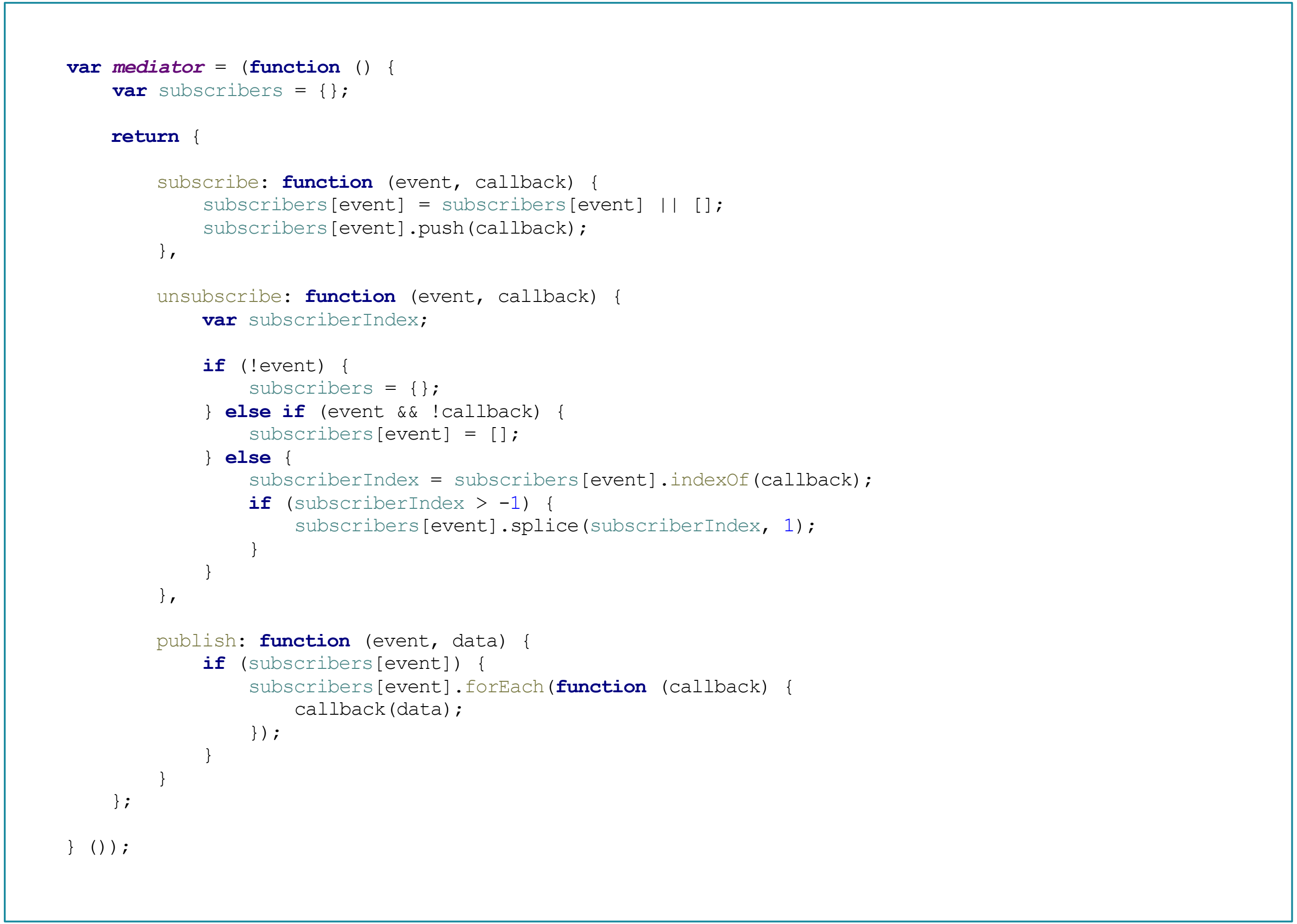
Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects.
The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



MEDIATOR


dataReceived: Data ready for processing..
Module 1
Notified of 'dataReceived' event
dataProcessed: Received data processed.
Start Module 2
Mediator
Module 2
Notified of 'dataProcessed' event
dataLogged: Published events and data logged.
Start Module 3
Module 3

JAVASCRIPT



```
var mediator = (function () {
    var subscribers = {};

    return {

        subscribe: function (event, callback) {
            subscribers[event] = subscribers[event] || [];
            subscribers[event].push(callback);
        },

        unsubscribe: function (event, callback) {
            var subscriberIndex;

            if (!event) {
                subscribers = {};
            } else if (event && !callback) {
                subscribers[event] = [];
            } else {
                subscriberIndex = subscribers[event].indexOf(callback);
                if (subscriberIndex > -1) {
                    subscribers[event].splice(subscriberIndex, 1);
                }
            }
        },

        publish: function (event, data) {
            if (subscribers[event]) {
                subscribers[event].forEach(function (callback) {
                    callback(data);
                });
            }
        }
    };

} ());
```

Applicability

- Use the Mediator pattern when it’s hard to change some of the classes because they are tightly coupled to a bunch of other classes.

- Use the pattern when you can’t reuse a component in a different program because it’s too dependent on other components.

- Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.

- **DRY**
  Don't repeat yourself

- **KISS**
  Keep it simple, stupid

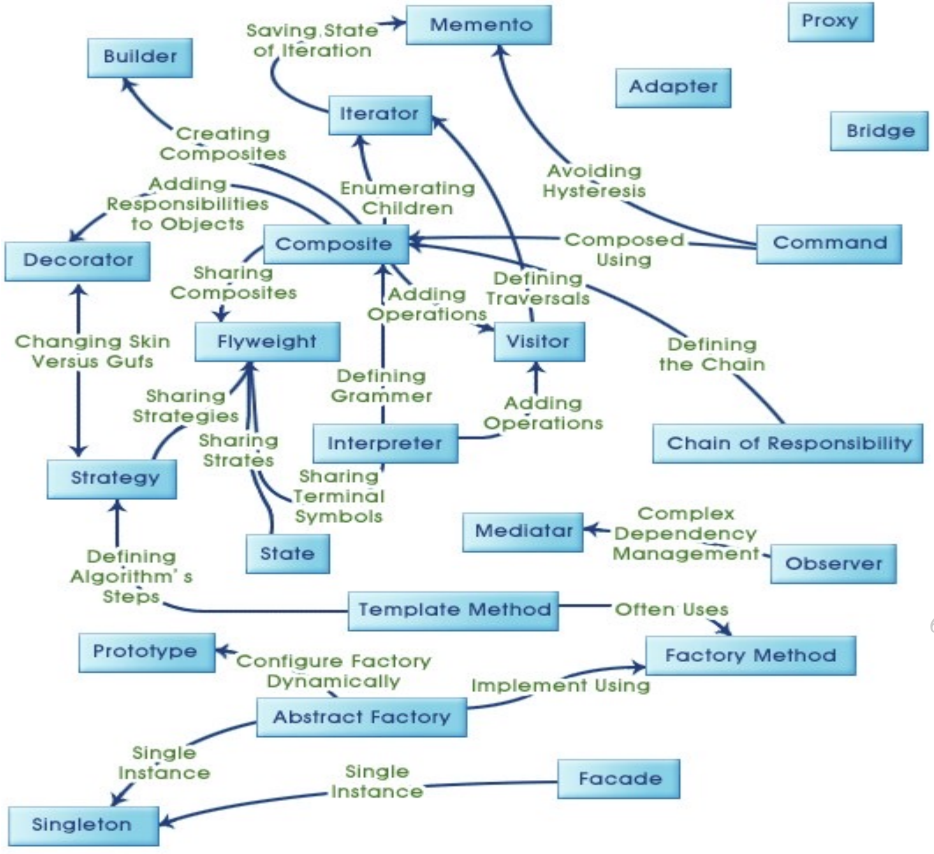
- And many other…

CONCLUSION

What we’ve learn today:

- Design patterns: types and what kind of problems they resolve
- SOLID principles
- Other principles to keep your code clean

DESIGN PATTERNS RELATIONSHIPS

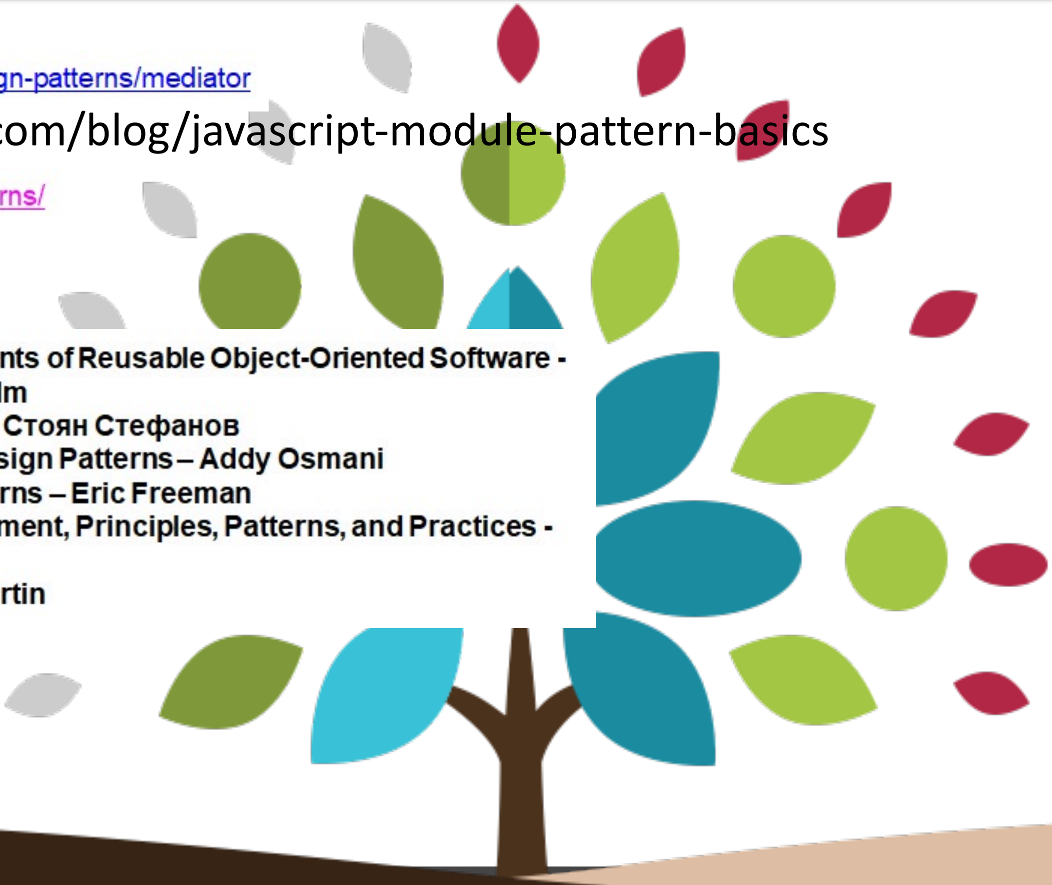


RESOURCES

https://refactoring.guru/design-patterns/mediator

https://coryrylan.com/blog/javascript-module-pattern-basics

http://cpp-reference.ru/patterns/



1. Design Patterns: Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm
2. Javascript. Шаблоны – Стоян Стефанов
3. Learning JavaScript Design Patterns – Addy Osmani
4. Head First Design Patterns – Eric Freeman
5. Agile Software Development, Principles, Patterns, and Practices - Robert Martin
6. Clean Code - Robert Martin

FE Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

serhii_shcherbak@epam.com

With the note [FE Online UA Training Course Feedback]

Thank you.

Q&A

<epam>

DRIVEN CANDID CREATIVE ORIGINAL INTELLIGENT EXPERT

UA Frontend Online LAB