OOAD-assignment-1

Assignment in Object-oriented analysis and design at Nackademin

GitHub: https://github.com/RickardPedersen/OOAD-assignment-1

Setup

Install dependencies

```
npm install
```

Start app with live server

```
npm start
```

Uppgift

Välj 3 valfria designmönster från boken "Learning JavaScript Design Patterns"

1. Facade Pattern
2. Factory Pattern
3. Revealing Module Pattern

Beskrivning av mina designmönster

Facade Pattern

Facade Pattern är ett strukturellt designmönster vilket innebär att det behandlar relationer mellan objekt.

Grundprincipen bakom Facade Pattern är att dölja komplex kod eller stora kodblock bakom ett användarvänligt gränssnitt.

jQuery är ett exempel på Facade Pattern. Med jQuery kan utvecklare på ett enkelt sätt hantera bland annat DOM-manipulation och AJAX-requests, som innan ES6 var väldigt komplext med JavaScript.

Exempel

Så här kan det se ut om man inte använder Facade Pattern:

```
async function withoutFacade() {
    async function getUsers() {
```

```
        const response = await fetch('https://jsonplaceholder.typicode.com/users',
{
            method: 'GET',
            headers: { 'Content-Type': 'application/json' }
        })
        return response.json()
    }

    async function getUserPosts(userId) {
        const response = await fetch(`https://jsonplaceholder.typicode.com/posts?
userId=${userId}`, {
            method: 'GET',
            headers: { 'Content-Type': 'application/json' }
        })
        return response.json()
    }

    const users = await getUsers()
    console.log(users)
    for (const user of users) {
        const posts = await getUserPosts(user.id)
        console.log(posts)
    }
}

withoutFacade()
```

Vi kan istället dölja den komplexa koden bakom en fasad som skulle kunna se ut så här. I detta exemplet är fasaden inbygd i en Revealing Module. Du kan läsa mer om Revealing Module Pattern längre ner.

```
const rixios = (function () {
    const privateDefaultHeaders = { 'Content-Type': 'application/json' }

    async function publicGet(url = '', config = {}) {
        if (this.baseUrl) {
            url = this.baseUrl + url
        }
        const { headers, params } = config
        return await privateRequest(url, {
            method: 'GET',
            headers: headers || privateDefaultHeaders,
            params: { ...params },
        })
    }

    async function privateRequest(url, options) {
        const queryString = Object.entries(options.params)
            .map((param) => {
                return `${param[0]}=${param[1]}`
            })
            .join('&')
```

```
        const fullUrl = `${url}?${queryString}`
        delete options.params

        console.log(`${options.method} ${fullUrl}`)
        const res = await fetch(fullUrl, options)
        return res.json()
    }

    function publicCreate(baseUrl) {
        return {
            baseUrl,
            get: publicGet,
        }
    }

    return {
        get: publicGet,
        create: publicCreate,
    }
})()
```

Nu har utvecklarna tillgång till ett användarvänligt gränssnitt! Så här skulle det kunna se ut:

```
async function withFacade() {
    const jsonplaceholder = rixios.create('https://jsonplaceholder.typicode.com')

    async function getUsers() {
        return await jsonplaceholder.get('/users')
    }

    async function getUserPosts(userId) {
        return await jsonplaceholder.get('/posts', { params: { userId } })
    }

    const users = await getUsers()
    console.log(users)
    for (const user of users) {
        const posts = await getUserPosts(user.id)
        console.log(posts)
    }
}
withFacade()
```

Factory Pattern

Factory Pattern är, till skillnad från Facade Pattern, ett Skapande designmönster, vilket innebär att det behandlar mekanismer för objektskapande. Istället för att skapa object med nyckelordet `new` så låter vi vår Factory skapa objektet åt oss.

Exempel

Vi börjar med att definera våra classer:

```
class Vehicle {
    constructor({ name, id }) {
        this.name = name
        this.id = id
    }
}

class Rocket extends Vehicle {
    constructor({ name, id, description, active }) {
        super({ name, id })
        this.description = description
        this.active = active
    }
}

class Dragon extends Vehicle {
    constructor({ name, id, description, active }) {
        super({ name, id })
        this.description = description
        this.active = active
    }
}

class Ship extends Vehicle {
    constructor({ name, id, type, roles }) {
        super({ name, id })
        this.type = type
        this.roles = roles
    }
}
```

Så här skapar man objekt enligt Constructor Pattern:

```
const rocket = new Rocket()
const dragon = new Dragon()
const ship = new Ship()
```

Men vi kan istället bygga en fabrik som skapar object åt oss:

```
class VehicleFactory {
    constructor() {
        this.vehicleClass = Vehicle;
    }
```

```
    createVehicle(options = {}) {
        switch (options.vehicleType) {
            case 'rocket':
                this.vehicleClass = Rocket
                break
            case 'dragon':
                this.vehicleClass = Dragon
                break
            case 'ship':
                this.vehicleClass = Ship
                break
            default:
                this.vehicleClass = Vehicle
                break
        }

        return new this.vehicleClass(options)
    }
}
```

Nu kan vi på ett enkelt sätt skapa många olika typer av object genom att använda vår fabrik.

```
const factory = new VehicleFactory()
const rocket = factory.createVehicle({ vehicleType: 'rocket' })
const dragon = factory.createVehicle({ vehicleType: 'dragon' })
const ship = factory.createVehicle({ vehicleType: 'ship' })
```

Revealing Module Pattern

Revealing Module Pattern är, precis som Facade Pattern, ett strukturellt designmönster, som behandlar relationer mellan objekt. Till skillnad från Facade Pattern, så används Revealing Module Pattern för att efterlikna klasser med public och private inkapsling, en funtionalitet som ES6 klasser har begränsat stöd för. På så sätt kan vi skydda variabler och metoder från det globala scopet. Revealing Module Pattern är en "förbättrad" version av Module Pattern som är väldigt liknande.

Exempel

I Module Pattern så defineras alla privata variabler och metoder i det privata scopet och alla publika i det publika return-objektet.

```
const modulePattern = (function () {
    // Encapsulation
    let counter = 0 // private variable

    return {
        // public methods

        incrementCounter: function () {
            return counter++
```

```
        },

        resetCounter: function () {
            console.log("counter value prior to reset: " + counter)
            counter = 0
        }
    }
})()
```

Med Revealing Module Pattern så definerar vi istället alla varibler och metoder i det privata scopet och sen avslöjar (revealar) vi de variabler och metoder som ska vara publika.

```
const revealingModulePattern = (function () {
    let privateCounter = 0

    function publicIncrementCounter () {
        return privateCounter++
    }

    function publicResetCounter () {
        console.log("counter value prior to reset: " + privateCounter)
        privateCounter = 0
    }

    // Reveal public pointers to
    // private functions and properties
    return {
        incrementCounter: publicIncrementCounter,
        resetCounter: publicResetCounter
    }
})()
```

Dokumentation

Min kund är SpaceX som vill att jag bygger en webbsida åt dom.

Den strategiska nivån

Intervju med Elon Musk:

Vi vill visa upp våra raketer, kapslar (dragons) och båtar. Vi vill få så många besökare som möjligt (globalt). Webbsidan ska ha ett rymdtema. Webbsidan ska väcka intresse för rymden och SpaceX hos besökaren. Webbsidan ska vara cool.

Produktmål

- Visa racketer, dragons och båtar.
- Locka så många besökare som möjligt.
- Coolt och inspirerande rymdtema.

Intervju med potentiella användare:

Jag vill få en beskrivning av dom olika rymdfordonen. Jag vill se vad båtarna används till. Jag vill kunna använda webbsidan på min dator och telefon. Jag vill se många bilder.

Användarbehov

- Se information om de olika rymdfordonen.
- Se vad båtarna används till.
- Se bilder på rymden och fordonen.
- Kunna använda webbsidan på dator och mobila enheter.

Omfattningsnivån

Kravspecifikation

Innehåll:

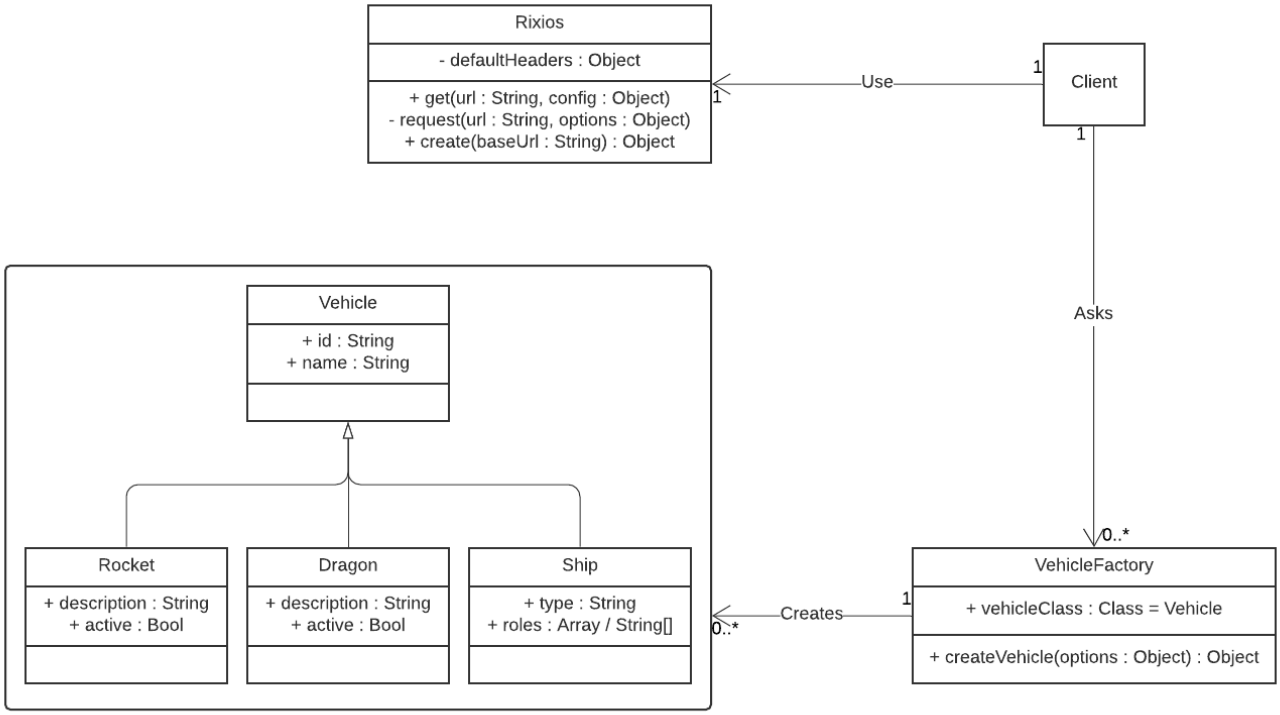
- Alla fordon visas med namn, bild.
- Raketer och Dragons visas med beskrivning.
- Båtar visas med information om användingsområden.
- Rymdbilder.

Funktionalitet:

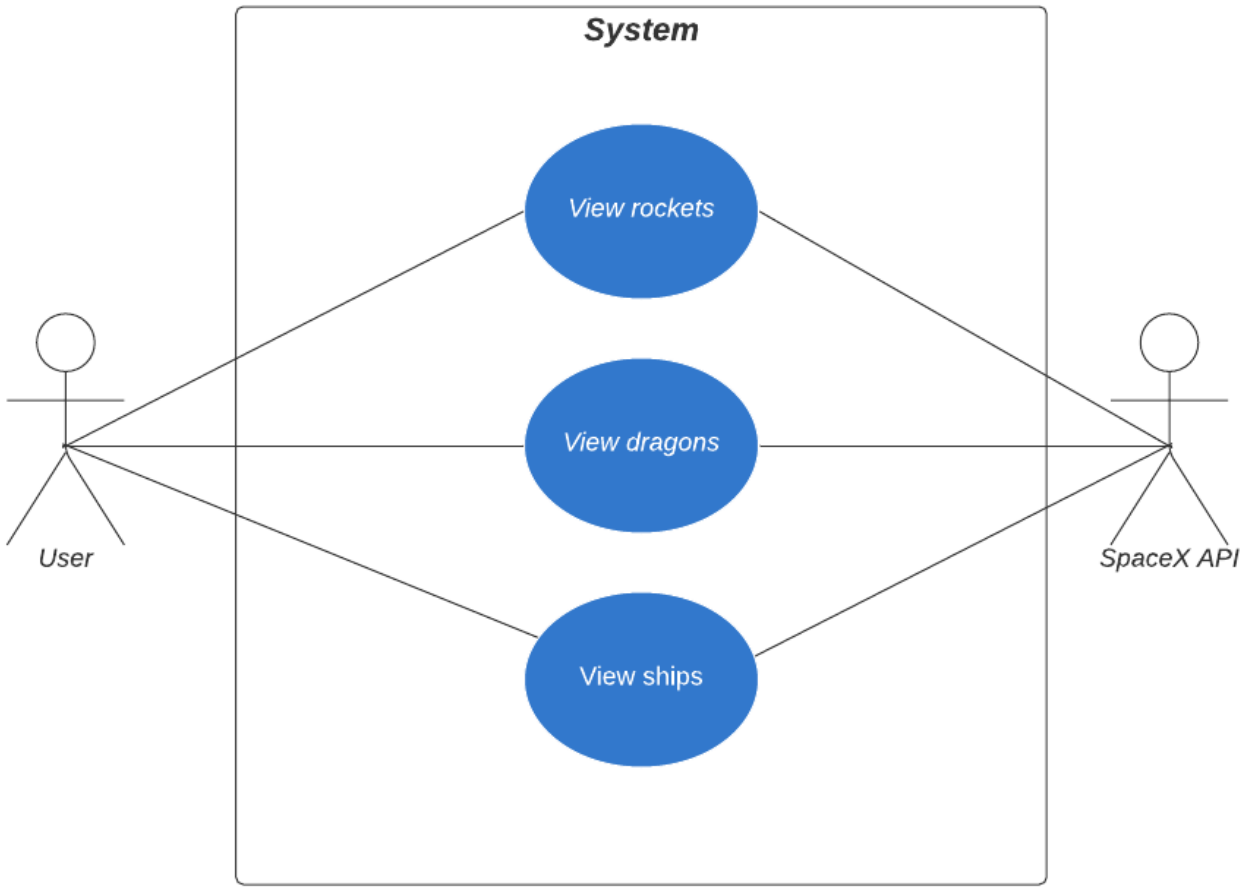
- Hämta data från SpaceX API

UML-diagram

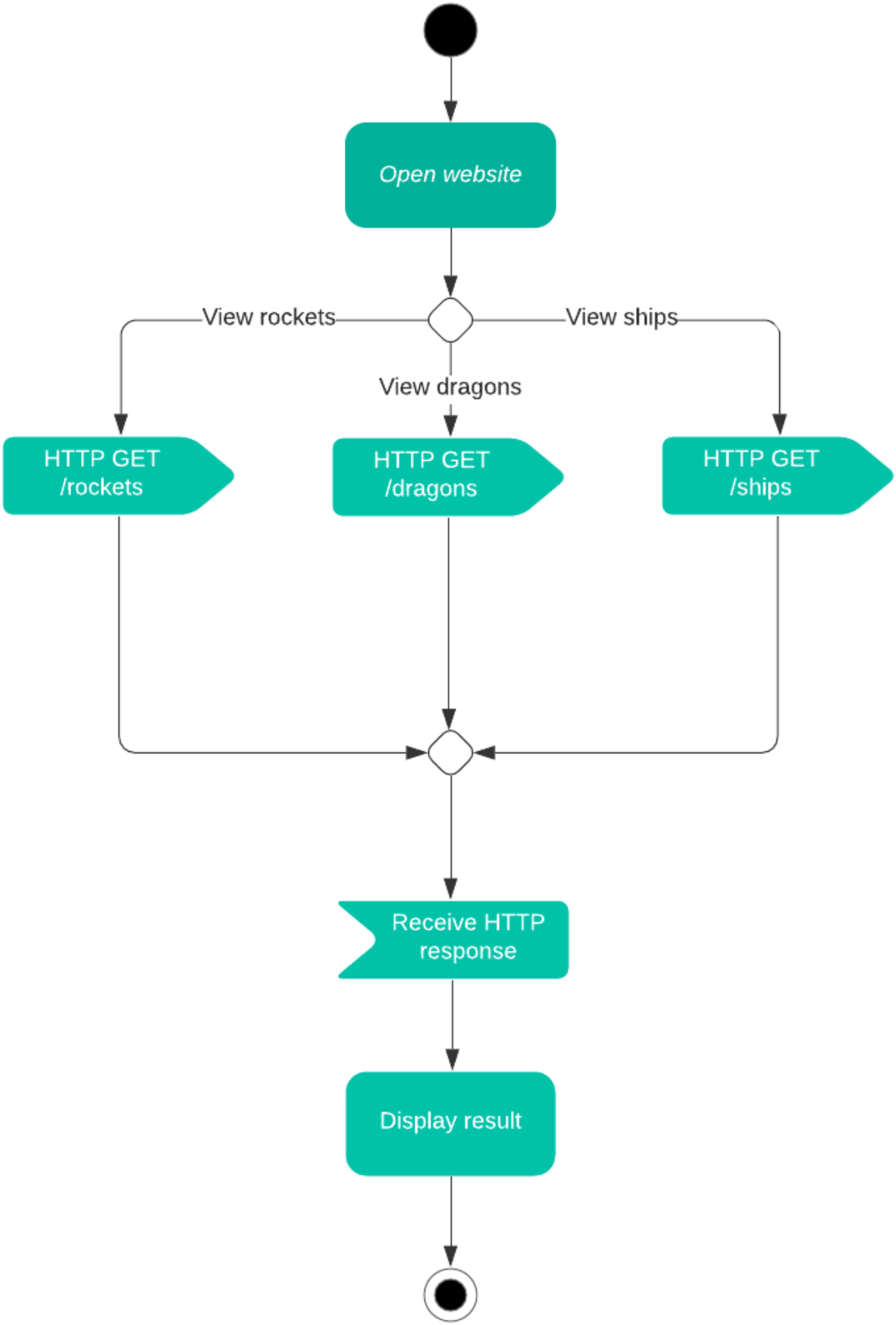
Klassdiagram



Användningsfallsdiagram



Aktivitetsdiagram


Open website
View rockets
View dragons
View ships
HTTP GET /rockets
HTTP GET /dragons
HTTP GET /ships
Receive HTTP response
Display result