TRƯỜNG ĐẠI HỌC SƯ PHẠM KĨ THUẬT THÀNH PHỐ HỒ CHÍ MINH

**KHOA CÔNG NGHỆ THÔNG TIN**



**MÔN HỌC: MẪU THIẾT KẾ PHẦN MỀM**

GVHD: NGUYỄN MINH ĐẠO

Nhóm thực hiện: Nhóm 16

Họ tên sinh viên: Nguyễn Viết Quang 20110549

Nguyễn Vinh Khang 20110131

Nguyễn Minh Đức 20110461

Nguyễn Văn An 20110434

TP. Hồ Chí Minh, tháng 05 năm 2023.

TRƯỜNG ĐẠI HỌC SƯ PHẠM KĨ THUẬT THÀNH PHỐ HỒ CHÍ MINH

**KHOA CÔNG NGHỆ THÔNG TIN**



**MÔN HỌC: MẪU THIẾT KẾ PHẦN MỀM**

GVHD: NGUYỄN MINH ĐẠO

Nhóm thực hiện: Nhóm 16

Họ tên sinh viên: Nguyễn Viết Quang 20110549

Nguyễn Vinh Khang 20110131

Nguyễn Minh Đức 20110461

Nguyễn Văn An 20110434

TP. Hồ Chí Minh, tháng 05 năm 2023.

# NHẬN XÉT CỦA GIÁO VIÊN

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

* Điểm số:
* Điểm chữ:

Thành phố Hồ Chí Minh, ngày … tháng … năm 2023

Giảng Viên

(Ký và ghi rõ họ tên)

# LỜI CẢM ƠN

Lời đầu tiên, em xin cảm ơn chân thành đến thầy NGUYỄN MINH ĐẠO dã dạy chúng em môn học môn học Design Pattern. Các bài giảng của thầy rất thú vị và hữu ích, giúp tôi có thể áp dụng các mẫu thiết kế để giải quyết các vấn đề trong dự án của mình. Thầy cũng đã truyền đạt cho tôi tinh thần làm việc chuyên nghiệp và sáng tạo, giúp tôi hoàn thiện kỹ năng của mình. Tôi xin chân thành cảm ơn lại thầy về sự giúp đỡ và hỗ trợ trong suốt khoá học. Em rất mong nhận được ý kiến đóng góp của thầy để nhóm chúng em học thêm được nhiều kinh nghiệm và sẽ hoàn thành tốt hơn bài báo cáo lần tới.

# MỤC LỤC

[NHẬN XÉT CỦA GIÁO VIÊN 1](#_Toc136716308)

[LỜI CẢM ƠN 2](#_Toc136716309)

[MỤC LỤC 3](#_Toc136716310)

[Chương 7: creational design patterns 7](#_Toc136716311)

[7.1 Factory 7](#_Toc136716312)

[**7.1.1.** **Định nghĩa:** 7](#_Toc136716313)

[**7.1.2. Code:** 7](#_Toc136716314)

[**7.1.3.** **Giải thích:** 9](#_Toc136716315)

[7.2 Builder 11](#_Toc136716316)

[**7.2.1.** **Định nghĩa:** 11](#_Toc136716317)

[**7.2.2. Code:** 12](#_Toc136716318)

[**7.2.3.** **Giải thích:** 14](#_Toc136716319)

[7.3 Revealing Constructor 17](#_Toc136716320)

[**7.3.1.** **Định nghĩa:** 17](#_Toc136716321)

[**7.3.2. Code:** 17](#_Toc136716322)

[**7.3.3.** **Giải thích:** 18](#_Toc136716323)

[7.4 Singleton 19](#_Toc136716324)

[**7.4.1 Định nghĩa** 19](#_Toc136716325)

[**7.4.2 Code** 19](#_Toc136716326)

[**7.4.3 Giải thích** 21](#_Toc136716327)

[Chương 8: structural design patterns 24](#_Toc136716328)

[8.1. Proxy 24](#_Toc136716329)

[**8.1.1.** **Định nghĩa** 24](#_Toc136716330)

[**8.1.2.** **Xây dựng proxy class** 26](#_Toc136716331)

[**8.1.3.** **Kỹ thuật mở rộng đối tượng (Object augmentation)** 29](#_Toc136716332)

[**8.1.4.** **Sử dụng class Proxy trong javascript** 31](#_Toc136716333)

[**8.1.5.** **So sánh giữa composition và object augumentation** 32](#_Toc136716334)

[**8.1.6.** **Xây dựng Proxy các đối tượng có sẵn trong javascript** 33](#_Toc136716335)

[**8.1.7.** **Thay đổi Observer với Proxy** 34](#_Toc136716336)

[8.2. Decorator 36](#_Toc136716337)

[**8.2.1.** **Định nghĩa** 36](#_Toc136716338)

[**8.2.2.** **Composition** 37](#_Toc136716339)

[**8.2.3.** **Object augmentation** 39](#_Toc136716340)

[**8.2.4.** **Decorating with the Proxy object** 40](#_Toc136716341)

[8.3. Adapter 42](#_Toc136716342)

[**8.3.1 Định nghĩa** 42](#_Toc136716343)

[**8.3.2 Code** 42](#_Toc136716344)

[**8.3.3 Giải thích** 44](#_Toc136716345)

[CHƯƠNG 9: 46](#_Toc136716346)

[9.1 Strategy 46](#_Toc136716347)

[**Code:** 46](#_Toc136716348)

[**Giải thích** 48](#_Toc136716349)

[9.2 State 49](#_Toc136716350)

[**Code:** 49](#_Toc136716351)

[**Giải thích** 53](#_Toc136716352)

[9.3 Template: 54](#_Toc136716353)

[**Code:** 54](#_Toc136716354)

[**Giải thích** 56](#_Toc136716355)

[9.4: Iterator alphabet iterator: 57](#_Toc136716356)

[**Code:** 57](#_Toc136716357)

[**Giải thích** 58](#_Toc136716358)

[9.5 Iterator iterable matrix 59](#_Toc136716359)

[**Code:** 60](#_Toc136716360)

[**Giải thích** 61](#_Toc136716361)

[9.6 Iterator builtins 62](#_Toc136716362)

[**Code:** 62](#_Toc136716363)

[**Giải thích** 64](#_Toc136716364)

[9.7 Simple Generator 65](#_Toc136716365)

[**Code:** 65](#_Toc136716366)

[**Giải thích** 66](#_Toc136716367)

[9.8 Kiểm soát generator iterator 67](#_Toc136716368)

[**Code:** 67](#_Toc136716369)

[**Giải thích** 68](#_Toc136716370)

[9.9 Sử dụng generator thay cho iterators 69](#_Toc136716371)

[**Code:** 69](#_Toc136716372)

[**Giải thích** 70](#_Toc136716373)

[9.10 Ứng dụng async iterators pattern 72](#_Toc136716374)

[**Code:** 72](#_Toc136716375)

[**Giải thích** 73](#_Toc136716376)

[9.11 Ứng dụng generator vào async iterators pattern 74](#_Toc136716377)

[**Code:** 74](#_Toc136716378)

[**Giải thích** 75](#_Toc136716379)

[9.12 Async iterators và Node.js stream 76](#_Toc136716380)

[**Code:** 76](#_Toc136716381)

[**Giải thích** 76](#_Toc136716382)

[9.13 MIDDLEWARE PATTERN 77](#_Toc136716383)

[**Code:** 78](#_Toc136716384)

[**Giải thích** 80](#_Toc136716385)

[9.14 COMMAND PATTERN 81](#_Toc136716386)

[**Code:** 82](#_Toc136716387)

[**Giải thích** 88](#_Toc136716388)

# Chương 7: creational design patterns

## 7.1 Factory

### **7.1.1. Định nghĩa:**

Factory Pattern là một mẫu thiết kế phần mềm (software design pattern) thuộc nhóm "creational patterns" trong lập trình hướng đối tượng (OOP). Nó là một mẫu thiết kế cho phép tạo ra các đối tượng một cách linh hoạt và trừu tượng hóa quá trình tạo đối tượng.

Factory Pattern giúp cho việc tạo ra các đối tượng trở nên linh hoạt hơn bằng cách sử dụng một factory class (lớp factory) để tạo ra các đối tượng thay vì tạo trực tiếp từ constructor function hoặc từ khóa "new". Factory class có thể được sử dụng để tạo ra các đối tượng khác nhau tùy theo các tham số đầu vào và trả về các đối tượng đã được tạo ra.

Factory Pattern giúp cho mã nguồn trở nên dễ bảo trì và mở rộng hơn, do các đối tượng được tạo ra thông qua factory class có thể được xây dựng bằng nhiều cách khác nhau tùy theo các tham số đầu vào.Factory Pattern có thể được triển khai dưới nhiều hình thức khác nhau, tùy thuộc vào yêu cầu và ứng dụng cụ thể.

### **7.1.2. Code:**

**Intern.js**

A screen shot of a computer program

Description automatically generated with low confidence

**Fresher.js**

A screenshot of a computer program

Description automatically generated with medium confidence

**Junior.js**

A screenshot of a computer program

Description automatically generated with low confidence

**Senior.js**

A screen shot of a computer program

Description automatically generated with low confidence

**Employee Factory**

A screenshot of a computer program

Description automatically generated with low confidence

**Index.js**

A picture containing text, electronics, screenshot, display

Description automatically generated

### **7.1.3. Giải thích:**

**Intern.js** định nghĩa một class có tên là "Intern" để tạo ra các đối tượng "Intern", nó nhận vào một tham số "name" và gán giá trị của tham số đó cho thuộc tính "name" của đối tượng mới được tạo ra, nó cũng gán giá trị chuỗi "Intern" cho thuộc tính "type" của đối tượng mới được tạo ra.

**Fresher.js** định nghĩa một class có tên là "Fresher" để tạo ra các đối tượng "Fresher", nó nhận vào một tham số "name" và gán giá trị của tham số đó cho thuộc tính "name" của đối tượng mới được tạo ra, nó cũng gán giá trị chuỗi "Fresher" cho thuộc tính "type" của đối tượng mới được tạo ra.

**Junior.js** định nghĩa một class có tên là "Junior" để tạo ra các đối tượng "Junior", nó nhận vào một tham số "name" và gán giá trị của tham số đó cho thuộc tính "name" của đối tượng mới được tạo ra, nó cũng gán giá trị chuỗi "Junior" cho thuộc tính "type" của đối tượng mới được tạo ra.

**Senior.js** định nghĩa một class có tên là "Senior" để tạo ra các đối tượng "Senior", nó nhận vào một tham số "name" và gán giá trị của tham số đó cho thuộc tính "name" của đối tượng mới được tạo ra, nó cũng gán giá trị chuỗi "Senior" cho thuộc tính "type" của đối tượng mới được tạo ra.

**EmployeeFactory.js** định nghĩa một class có tên là "EmployeeFactory" với một phương thức khởi tạo "create" để tạo ra các đối tượng nhân viên (employee) dựa trên loại (type) của nhân viên được chỉ định.Nó sử dụng các import để lấy các constructor function đã được định nghĩa trong các module "Intern.js", "Fresher.js", "Junior.js", và "Senior.js". Sau đó, phương thức "create" sử dụng câu lệnh switch-case để xác định loại (type) của nhân viên được chỉ định và trả về một đối tượng mới tương ứng với loại nhân viên đó.

Cụ thể, nếu loại nhân viên là 1, phương thức "create" sẽ tạo ra một đối tượng "Intern" mới với tên được chỉ định. Nếu loại nhân viên là 2, phương thức sẽ tạo ra một đối tượng "Fresher" mới, và tương tự cho loại nhân viên là 3 và 4, với các đối tượng "Junior" và "Senior" được tạo ra lần lượt.

Nếu loại nhân viên không hợp lệ, phương thức "create" sẽ trả về một đối tượng "Intern" mới với tên được chỉ định.

EmployeeFactory này cung cấp một cách tiện lợi để tạo ra các đối tượng nhân viên (employee) với loại (type) được chỉ định một cách linh hoạt, giúp cho việc quản lý và mở rộng các loại nhân viên trở nên dễ dàng hơn.

Để tạo một đối tượng Fresher, ta sử dụng



Thay vì



Việc sử dụng một factory function để tạo đối tượng có thể có nhiều ưu điểm hơn:

* Quản lý các đối tượng: Sử dụng một factory function giúp cho việc quản lý các đối tượng trở nên dễ dàng hơn. Bởi vì việc khởi tạo các đối tượng được tổ chức tập trung tại một điểm duy nhất, điều này giúp cho việc theo dõi các đối tượng và quản lý chúng trở nên dễ dàng hơn.
* Tính tái sử dụng: Sử dụng một factory function cho phép tái sử dụng mã khởi tạo các đối tượng, giúp giảm thiểu việc lặp lại mã khởi tạo các đối tượng trong các phương thức khácnhau.
* Linh hoạt: Sử dụng một factory function cho phép tạo ra các đối tượng với các thuộc tính khác nhau dựa trên các tham số đầu vào. Điều này giúp cho việc tạo ra các đối tượng với các thuộc tính khác nhau trở nên linh hoạt hơn và dễ dàng hơn.
* Đóng gói: Sử dụng một factory function giúp đóng gói logic khởi tạo đối tượng vào một chức năng đơn giản, giúp cho mã trở nên dễ đọc và dễ hiểu hơn.
* Dễ kiểm tra: Sử dụng một factory function giúp cho việc kiểm tra các đối tượng trở nên dễ dàng hơn. Ví dụ, nếu có lỗi khi khởi tạo đối tượng trong factory function, ta có thể dễ dàng bắt lỗi và xử lý chúng mà không cần phải xử lý các lỗi đó trong nhiều nơi khác nhau trong mã nguồn.

**Index.js** sử dụng class "EmployeeFactory" đã được định nghĩa trong file "EmployeeFactory.js" để tạo ra 4 đối tượng "employee" với các thuộc tính khác nhau, và sau đó in ra thông tin của các đối tượng này bằng cách sử dụng phương thức console.log.

* Kết quả

A screenshot of a computer code

Description automatically generated with low confidence

## 7.2 Builder

### **7.2.1. Định nghĩa:**

Builder Pattern là một mẫu thiết kế phần mềm (software design pattern) thuộc nhóm "creational patterns" trong lập trình hướng đối tượng (OOP). Nó cho phép xây dựng các đối tượng phức tạp bằng cách chia nhỏ quá trình xây dựng thành các bước đơn giản hơn và cho phép các đối tượng được xây dựng theo các bước này một cách linh hoạt.

Trong Builder Pattern, một lớp "builder" được sử dụng để tạo ra các đối tượng phức tạp bằng cách gọi các phương thức xây dựng từng bước một. Các phương thức này cho phép người dùng thiết lập các thuộc tính và các thành phần của đối tượng một cách độc lập. Sau khi tất cả các bước xây dựng được thực hiện, đối tượng được trả về cho người dùng.

Builder Pattern được sử dụng khi cần tạo ra các đối tượng phức tạp, có nhiều thuộc tính và phụ thuộc vào một số yêu cầu hoặc trạng thái khác nhau. Ví dụ, khi tạo ra một đối tượng có nhiều thông số cấu hình khác nhau như một chiếc máy tính, một chiếc ô tô hoặc một trang web. Builder Pattern giúp cho quá trình tạo ra các đối tượng này trở nên dễ dàng hơn và linh hoạt hơn, đồng thời giúp cho mã nguồn trở nên dễ bảo trì và mở rộng hơn.

### **7.2.2. Code:**

**Person.js**

A screen shot of a computer code

Description automatically generated with low confidence

**PersonBuilder.js**

A screenshot of a computer program

Description automatically generated with medium confidence

**index.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

### **7.2.3. Giải thích:**

**Person.js** định nghĩa một class có tên là "Person" để tạo ra các đối tượng "Person" với các thuộc tính "name", "gender", "weight", và "height" lần lượt là tên, giới tính, cân nặng và chiều cao của một người.

**PersonBuilder.js** định nghĩa một class có tên là "PersonBuilder" để tạo ra các đối tượng "Person" một cách linh hoạt hơn và dễ dàng hơn bằng cách sử dụng Builder Pattern.

Class "PersonBuilder" có các thuộc tính "name", "gender", "weight" và "height" được khởi tạo bởi constructor function. Ngoài ra, class này còn có các phương thức "setWeight", "setHeight" và "build", cho phép người dùng thiết lập giá trị các thuộc tính "weight" và "height" một cách độc lập và xây dựng đối tượng "Person" với các thuộc tính đã thiết lập bằng phương thức "build".

Việc sử dụng class "PersonBuilder" giúp cho việc xây dựng đối tượng "Person" trở nên linh hoạt hơn và dễ dàng hơn, do người dùng có thể thiết lập giá trị các thuộc tính một cách độc lập và linh hoạt. Đồng thời, class "PersonBuilder" cũng giúp cho mã nguồn trở nên dễ bảo trì và mở rộng hơn, do các phương thức xây dựng được chia nhỏ thành các bước đơn giản hơn.

Để tạo một đối tượng Person ta sử dụng

A screenshot of a computer

Description automatically generated with low confidence

Thay vì



Ở đây khi sử dụng PersonBuilder ta biết rõ là 70, 170 là cân nặng và chiều cao của đối tượng, trong khi Person 70, 170 rất mơ hồ, trong tương lai có thể gây nhầm lẫn

Sử dụng class "PersonBuilder" để tạo ra các đối tượng "Person" có một số ưu điểm so với sử dụng trực tiếp constructor function "Person" bằng từ khóa "new".

* Xây dựng đối tượng linh hoạt và dễ dàng: Với class "PersonBuilder", người dùng có thể thiết lập giá trị các thuộc tính một cách độc lập và linh hoạt thông qua các phương thức như "setWeight", "setHeight", và "build". Điều này giúp cho việc xây dựng đối tượng trở nên dễ dàng, linh hoạt và tiện lợi hơn.
* Mã nguồn dễ bảo trì và mở rộng: Với class "PersonBuilder", các phương thức xây dựng được chia nhỏ thành các bước đơn giản hơn, giúp cho mã nguồn trở nên dễ bảo trì và mở rộng hơn.
* Giảm thiểu lỗi khi thiếu thông tin: Trong constructorfunction "Person", nếu thiếu thông tin trong quá trình khởi tạo đối tượng, ví dụ như không truyền giá trị cho các thuộc tính "weight" và "height", thì đối tượng sẽ được tạo ra với các thuộc tính này có giá trị là "undefined". Trong khi đó, sử dụng class "PersonBuilder" và các phương thức như "setWeight" và "setHeight" giúp đảm bảo rằng tất cả các thuộc tính được thiết lập đầy đủ trước khi đối tượng được xây dựng bằng phương thức "build", giúp giảm thiểu lỗi trong quá trình sử dụng.
* Tuy nhiên, sử dụng class "PersonBuilder" cũng có một số hạn chế. Class này đòi hỏi người dùng phải học cách sử dụng Builder Pattern và các phương thức như "setWeight", "setHeight" và "build", trong khi đó sử dụng constructor function "Person" bằng từ khóa "new" là cách đơn giản và trực tiếp hơn để tạo ra đối tượng "Person". Tùy vào tình huống cụ thể, người dùng có thể chọn sử dụng constructor function hoặc class "PersonBuilder" cho việc tạo ra các đối tượng "Person" phù hợp với nhu cầu và yêu cầu của dự án.

**index.js** tạo ra các đối tượng "Person" một cách linh hoạt và dễ dàng hơn bằng cách sử dụng class "PersonBuilder" và các phương thức như "setWeight", "setHeight" và "build". Kết quả là ta có một mảng "persons" chứa các đối tượng "Person" với các thuộc tính tương ứng được thiết lập bởi các phương thức này.

* Kết quả

**A screenshot of a computer program

Description automatically generated with medium confidence**

## 7.3 Revealing Constructor

### **7.3.1. Định nghĩa:**

Revealing Constructor pattern là một mẫu thiết kế (design pattern) trong lập trình hướng đối tượng trong JavaScript. Mẫu thiết kế này kết hợp giữa hai mẫu khác là Constructor pattern và Revealing module pattern.

Constructor pattern là một mẫu thiết kế cho phép tạo ra các đối tượng mới từ một hàm khởi tạo (constructor function) và từ khóa "new". Nó cho phép tạo ra các đối tượng có các thuộc tính và phương thức riêng biệt.

Revealing module pattern là một mẫu thiết kế cho phép tạo ra các module trong JavaScript, cho phép các hàm và biến tạo ra trong module được ẩn đi và không truy cập được từ bên ngoài module. Thay vào đó, chỉ một số thành phần được tiết lộ (reveal) ra bên ngoài module thông qua một đối tượng trả về từ module.

Khi kết hợp với nhau, Revealing Constructor pattern cung cấp cho chúng ta một cách để tạo ra các đối tượng có các thuộctính và phương thức riêng biệt, đồng thời cũng cho phép ẩn các thành phần bên trong hàm khởi tạo (constructor function) và chỉ tiết lộ các thành phần cần thiết thông qua một đối tượng trả về.

Cụ thể, Revealing Constructor pattern cho phép chúng ta định nghĩa một hàm khởi tạo (constructor function) và sử dụng các biến và hàm bên trong hàm khởi tạo để thiết lập các thuộc tính và phương thức cho các đối tượng được tạo ra. Tuy nhiên, thay vì trả về đối tượng mới, hàm khởi tạo sẽ trả về một đối tượng có các phương thức và thuộc tính được tiết lộ (reveal) ra bên ngoài.

### **7.3.2. Code:**

**Person.js**

A picture containing text, screenshot, font

Description automatically generated

**Index.js**

A screen shot of a computer program

Description automatically generated with low confidence

### **7.3.3. Giải thích:**

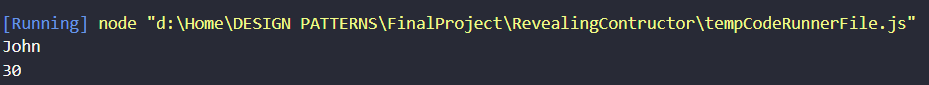
**Person.js** định nghĩa một class có tên là "Person" có hai tham số là "name" và "age", và được sử dụng để tạo ra các đối tượng "Person" có tên và tuổi tương ứng.

Trong hàm khởi tạo "Person", có hai biến "\_name" và "\_age" được sử dụng để lưu trữ thông tin về tên và tuổi của các đối tượng "Person". Hai hàm "getName" và "getAge" được sử dụng để truy xuất các giá trị của "\_name" và "\_age". Điều này đảm bảo rằng các thông tin về tên và tuổi của đối tượng "Person" được bảo vệ và không truy cập được từ bên ngoài.

Cuối cùng, hàm khởi tạo "Person" trả về một đối tượng với hai phương thức là "getName" và "getAge", cho phép bên ngoàicó thể truy xuất các giá trị tên và tuổi của đối tượng "Person" thông qua các phương thức này. Điều này giúp đảm bảo tính đóng gói (encapsulation) của dữ liệu bên trong đối tượng "Person", và tránh việc truy cập trực tiếp các thuộc tính của đối tượng từ bên ngoài.

**Index.js** sử dụng hàm khởi tạo "Person" để tạo ra một đối tượng "person" với thông tin tên là "John" và tuổi là 30. Đối tượng "person" được tạo ra bằng cách sử dụng từ khóa "new" và hàm khởi tạo "Person", và được gán vào biến "person". Tiếp theo, đoạn code sử dụng phương thức "getName" của đối tượng "person" để lấy thông tin về tên của đối tượng, và sử dụng phương thức "getAge" để lấy thông tin về tuổi của đối tượng. Hai phương thức này được trả vềbởi hàm khởi tạo "Person" thông qua một đối tượng trả về, và cho phép bên ngoài truy xuất các thuộc tính của đối tượng "person" một cách an toàn.

* **Kết quả**

****

## 7.4 Singleton

### **7.4.1 Định nghĩa**

Singleton là một mẫu thiết kế phần mềm (software design pattern) thuộc nhóm "creational patterns" trong lập trình hướng đối tượng (OOP). Nó được sử dụng để đảm bảo rằng một lớp chỉ có một đối tượng (instance) duy nhất trong toàn bộ ứng dụng. Điều này đảm bảo rằng việc tạo ra các đối tượng mới của lớp đó sẽ luôn trả về cùng một đối tượng, và đối tượng này sẽ được chia sẻ và sử dụng chung trong toàn bộ ứng dụng.

Để triển khai Singleton pattern, ta cần định nghĩa một lớp với một phương thức tĩnh (static method) để trả về đối tượng duy nhất của lớp đó. Phương thức tĩnh này sẽ kiểm tra xem đối tượng đã được tạo hay chưa, và nếu chưa thì nó sẽ tạo một đối tượng mới và lưu trữ nó để sử dụng cho các lần gọi sau. Nếu đối tượng đã được tạo, phương thức tĩnh sẽ trả về đối tượng đó.

Singleton pattern thường được sử dụng trong các trường hợp mà ta muốn đảm bảo rằng một lớp chỉ có một đối tượng duy nhất, chẳng hạn như các cấu hình ứng dụng, các đối tượng quản lý kết nối đến cơ sở dữ liệu, các đối tượng đại diện cho các thiết bị phần cứng, và các đối tượng khác mà việc tạo ra nhiều đối tượng sẽ làmmất tính nhất quán hoặc làm tăng chi phí bộ nhớ đáng kể.

### **7.4.2 Code**

**Database.js**

**A screenshot of a computer program

Description automatically generated with medium confidence**

**Index.js**

**A screen shot of a computer

Description automatically generated with medium confidence**

### **7.4.3 Giải thích**

**Database.js** triển khai Singleton pattern để tạo ra một đối tượng duy nhất của lớp "Database" trong ứng dụng. Điều này đảm bảo rằng chỉ có một đối tượng "Database" được tạo ra và sử dụng trong toàn bộ ứng dụng.

Trong đoạn mã, ta định nghĩa một hàm ẩn danh (anonymous function) và gán nó vào biến "Database". Hàm ẩn danh này chứa một biến "instance" để lưu trữ đối tượng "Database" duy nhất, và một hàm "createInstance" để tạo ra đối tượng "Database" đó.

Trong hàm "createInstance", ta định nghĩa một biến "connection" để lưu trữ kết nối đến cơ sở dữ liệu, và ba phương thức là "connect", "disconnect" và "getState" để thực hiện các hoạt động liên quan đến kết nối cơ sở dữ liệu. Hàm "connect" được sử dụng để kết nối đến cơ sở dữ liệu, hàm "disconnect" được sử dụng để ngắt kết nối đến cơ sở dữ liệu, và hàm "getState" được sử dụng để trả về trạng thái hiện tại của kết nối đến cơ sở dữ liệu.

Trong hàm "createInstance", ta trả về một đối tượng chứa ba phương thức "connect", "disconnect" và "getState", đảm bảo rằng các phương thức này được bảo vệ và không truy cập được từ bên ngoài.

Trong hàm khởi tạo Singleton, ta sử dụng phương thức "getInstance" để trả về đối tượng "Database" duy nhất. Phương thức này kiểm tra xem biến "instance" đã được khởi tạo chưa. Nếu chưa, ta gọi hàm "createInstance" để tạo ra đối tượng "Database" mới và lưu trữ nó trong biến "instance". Nếu biến "instance" đã được khởi tạo, phương thức "getInstance" sẽ trả về đối tượng "Database" đãtạo trong biến "instance".

# Chương 8: structural design patterns

## Proxy

### **Định nghĩa**

Mẫu proxy là một mẫu thiết kế phần mềm thuộc nhóm cấu trúc (structural design pattern) trong lĩnh vực phát triển phần mềm. Nó được sử dụng để tạo ra một đại diện (proxy) cho một đối tượng khác nhằm kiểm soát quyền truy cập đến đối tượng gốc (subject).

Mục đích chính của mẫu proxy là che giấu hoặc kiểm soát quyền truy cập đến một đối tượng bên trong thông qua việc tạo ra một đối tượng proxy. Đối tượng proxy hoạt động như một lớp trung gian giữa người sử dụng và đối tượng gốc, cho phép kiểm soát các hoạt động trước, trong và sau khi truy cập đối tượng gốc.

Một mẫu proxy có thể hữu ích trong nhiều trường hợp khác nhau, ví dụ:

* Xác thực dữ liệu: Proxy kiểm tra tính hợp lệ của dữ liệu trước khi chuyển tiếp cho đối tượng gốc.
* Bảo mật: Proxy xác minh xem khách hàng có quyền thực hiện hoạt động hay không, và chỉ chuyển tiếp yêu cầu cho đối tượng gốc nếu kết quả của kiểm tra là đúng.
* Lưu trữ cache: Proxy duy trì một bộ nhớ cache nội bộ để các hoạt động của proxy được thực hiện trên đối tượng gốc chỉ khi dữ liệu chưa có trong cache.
* Khởi tạo chậm: Nếu việc tạo ra đối tượng gốc tốn kém, proxy có thể trì hoãn việc khởi tạo cho đến khi thực sự cần thiết.
* Ghi log: Proxy ngăn chặn các lời gọi phương thức và các tham số tương ứng, ghi lại chúng khi chúng xảy ra.
* Đối tượng từ xa: Proxy có thể nhận một đối tượng từ xa và làm cho nó trở nên như là địa phương.

**Code**

Resource.js

export class Resource {

    constructor(resourceName, user) {

        this.resourceName = resourceName;

        this.user = user;

        this.load(this.resourceName);

    }

    load(resourceName) {

        console.log(`Loading ${resourceName}`)

    }

    display() {

        console.log("Displaying " + this.resourceName);

    }

    download() {

        console.log("Downloading " + this.resourceName);

    }

}

User.js

export class User {

    constructor(username, password) {

        this.username = username;

        this.password = password;

    }

}

Index.js

import { Resource } from './Resource.js'

import { User } from './User.js'

const user = new User('admin', '123')

const resource = new Resource("Video", user)

resource.display()

resource.download()

**Giải thích**

Class User có một constructor nhận vào tên người dùng và mật khẩu và lưu chúng trong các thuộc tính của class.

Class Resource có một constructor nhận vào tên tài nguyên và một đối tượng User. Trong constructor, tài nguyên được tải bằng cách gọi phương thức load(), và tên tài nguyên và đối tượng User được lưu trong các thuộc tính của class. Class cũng có hai phương thức khác là display() và download(), mỗi phương thức đơn giản chỉ là in ra màn hình thông báo tương ứng.

Trong file index.js, nó import hai class Resource và User từ các file khác và tạo ra một đối tượng User với tên người dùng và mật khẩu đã cung cấp. Sau đó, nó tạo ra một đối tượng Resource với tên tài nguyên và đối tượng User đã tạo và gọi các phương thức display() và download() trên đối tượng Resource.

Vấn đề là việc xác thực người dùng chỉ xảy ra khi đối tượng User được tạo ra và truyền vào đối tượng Resource trong file index.js. Điều này có nghĩa là mọi tác vụ liên quan đến xác thực người dùng phải hoàn thành trước khi tài nguyên được tải hoặc các phương thức display() và download() được gọi trên đối tượng Resource. Tuy nhiên, trong một ứng dụng thực tế, xác thực người dùng có thể cần được thực hiện ở mỗi bước hoạt động, bao gồm cả việc tải tài nguyên và hiển thị nội dung.

### **Xây dựng proxy class**

**Code**

ResourceProxy.js:

import { Resource } from "../Resource.js";

export class ResourceProxy {

    constructor(resource) {

        this.resource = resource;

    }

    display() {

        this.resource.display();

    }

    download() {

        if (!this.isAuthenticated()) {

            console.log("You do not have permission to download");

            return;

        }

        this.resource.download()

    }

    isAuthenticated() {

        if (this.resource.user.username !== "admin" || this.resource.user.password !== "123") {

            return false;

        }

        return true;

    }

}

index.js:

import { ResourceProxy } from "./ResourceProxy.js";

import { Resource } from "../Resource.js";

import { User } from "../User.js";

const user01 = new User("nva", "1234");

const user02 = new User("admin", "123");

const resource01 = new Resource("Video 10GB", user01);

const resourceProxy01 = new ResourceProxy(resource01, user01);

resourceProxy01.display()

resourceProxy01.download()

console.log("\n")

const resource02 = new Resource("Video 10GB", user02);

const resourceProxy02 = new ResourceProxy(resource02, user01);

resourceProxy02.display()

resourceProxy02.download()

console.log("\n")

resource01.download()

resource02.download()

Kết quả:

Loading Video 10GB

Displaying Video 10GB

You do not have permission to download

Loading Video 10GB

Displaying Video 10GB

Downloading Video 10GB

Downloading Video 10GB

Downloading Video 10GB

**Giải thích**

Để giải quyết vấn đề trên, chúng ta tạo 1 lớp proxy tên ProxyResource, proxy này kiểm soát việc truy cập và thực thi các phương thức của đối tượng gốc, nó được triển khai bằng cách viết lại các phương thức của class Resource nhưng trước khi gọi download() trên đối tượng resource, phương thức isAuthenticated() được gọi để kiểm tra xác thực người dùng. Nếu người dùng không được xác thực thành công, một thông báo sẽ được hiển thị và phương thức download() không được thực thi. Nếu người dùng xác thực thành công, phương thức download() sẽ được gọi trên đối tượng resource.

Phương thức isAuthenticated(): Kiểm tra xem đối tượng resource có thuộc tính user (đối tượng người dùng) và xác thực người dùng bằng cách so sánh tên người dùng và mật khẩu với các giá trị được chỉ định ("admin" và "123"). Nếu xác thực thành công, phương thức trả về true, ngược lại, trả về false.

Trong demo, chúng ta tạo ra 2 tài nguyên resource01 và resource02 tương ứng với 2 user01 và user02. Sau đó, khởi tạo một đối tượng của lớp ProxyResource, proxyResource01 và proxyResource01 tương ứng cho 2 tài nguyên trên.

Khi đối tượng của lớp ProxyResource gọi download(), chúng đã thực thi isAuthenticated() để xác thực người dùng, do đó chỉ có user01 không tải được video.

Còn nếu chúng ta gọi download() từ các đối tượng của Resource, thì nó sẽ cho phép tất cả user đều có quyền tải video

Phương pháp Object Composition trong mã sử dụng Proxy Pattern có các ưu điểm và nhược điểm sau:

* Ưu điểm:
  + Tách biệt trách nhiệm: Proxy Pattern cho phép tách biệt trách nhiệm giữa Proxy và đối tượng gốc. Proxy chịu trách nhiệm xử lý các logic bổ sung như xác thực, kiểm soát truy cập, và đối tượng gốc chỉ chịu trách nhiệm cho logic cốt lõi.
  + Mở rộng và bảo vệ: Với Proxy Pattern, bạn có thể mở rộng chức năng của đối tượng gốc bằng cách thêm các logic bổ sung vào Proxy mà không cần thay đổi đối tượng gốc. Điều này giúp bảo vệ đối tượng gốc khỏi các thay đổi không mong muốn và đảm bảo tính linh hoạt của mã.
  + Kiểm soát truy cập: Proxy Pattern cho phép bạn kiểm soát truy cập đến các phương thức và thuộc tính của đối tượng gốc. Bạn có thể áp dụng các quy tắc xác thực, kiểm tra quyền và thực hiện các hành động phụ khác trước khi cho phép truy cập vào đối tượng gốc.
  + Kế thừa giao diện: Proxy Pattern cho phép Proxy và đối tượng gốc triển khai cùng một giao diện. Điều này đảm bảo rằng Proxy có thể được sử dụng thay thế đối tượng gốc mà không làm thay đổi mã sử dụng.
* Nhược điểm:
  + Khả năng gây rối: Một cấu trúc mã phức tạp với nhiều lớp Proxy có thể gây khó khăn trong việc hiểu và theo dõi luồng điều khiển của mã.
  + Hiệu suất: Việc sử dụng Proxy có thể ảnh hưởng đến hiệu suất vì các phương thức và thuộc tính phải được gọi thông qua một lớp Proxy trung gian. Điều này có thể gây ra một lượng nhỏ overhead thêm cho mã.
  + Quản lý bộ nhớ: Sử dụng Proxy Pattern có thể yêu cầu quản lý bộ nhớ bổ sung do sự có mặt của các đối tượng Proxy.
  + Khó khăn trong gỡ lỗi: Khi sử dụng Proxy Pattern, việc gỡ lỗi có thể trở nên phức tạp hơn do sự thêm vào các lớp Proxy trung gian và các logic bổ sung.

Trong tổng quát, việc sử dụng phương pháp Object Composition và Proxy Pattern mang lại nhiều lợi ích như tách biệt trách nhiệm, mở rộng và bảo vệ, kiểm soát truy cập và kế thừa giao diện. Tuy nhiên, cần lưu ý rằng nó có thể gây rối, ảnh hưởng đến hiệu suất, yêu cầu quản lý bộ nhớ bổ sung và làm phức tạp việc gỡ lỗi. Việc sử dụng phương pháp này nên được cân nhắc dựa trên quy mô và yêu cầu cụ thể của dự án, để đảm bảo tính linh hoạt, bảo mật và quản lý mã hiệu quả.

### **Kỹ thuật mở rộng đối tượng (Object augmentation)**

**Code**

index.js

import { Resource } from "../Resource.js"

import { User } from "../User.js"

function SecurityResource(resource) {

    const downloadResourceOrig = resource.download

    resource.download = () => {

        if (resource.user.username !== "admin" || resource.user.password !== "123") {

            console.log("You do not have permission to download")

            return;

        }

        // if valid delegates to the subject

        return downloadResourceOrig.apply(resource)

    }

    return resource

}

const user01 = new User("nva", "1234");

const resource01 = new Resource("Video 10GB", user01);

resource01.download()

const securityResource01 = new SecurityResource(resource01)

securityResource01.download()

console.log("\n")

const user02 = new User("admin", "123");

const resource02 = new Resource("Video 10GB", user02);

resource02.download()

const securityResource02 = new SecurityResource(resource02)

securityResource02.download()

console.log("\n")

resource01.download()

resource02.download()

**Giải thích**

"Object augmentation" hay còn được gọi là "monkey patching" là một cách đơn giản và phổ biến nhất để tạo proxy cho một số phương thức của một đối tượng. Nó sửa đổi đối tượng gốc trực tiếp bằng cách thay thế một phương thức bằng phương thức mới.

Bằng cách này, chúng ta có thể kiểm soát được các phương thức cụ thể mà chúng ta muốn tạo proxy mà không cần phải tạo một đối tượng mới.

Tuy nhiên, phương pháp này có thể dẫn đến việc làm rối loạn hoặc làm hỏng các tính năng khác của đối tượng, vì vậy nó nên được sử dụng cẩn thận.

Function SecurityResource: Đây là một hàm nhận đối tượng resource làm tham số và thực hiện việc mở rộng đối tượng đó để triển khai xác thực người dùng.

* Tạo một biến downloadResourceOrig để lưu trữ phiên bản ban đầu của phương thức download trong đối tượng resource.
* Ghi đè phương thức download của đối tượng resource bằng một hàm mới. Hàm này kiểm tra xem người dùng có tên người dùng và mật khẩu là "admin" và "123" hay không. Nếu không, hiển thị một thông báo và không tiến hành tải tài nguyên. Nếu xác thực thành công, hàm gọi lại phương thức download ban đầu bằng cách sử dụng apply và truyền đối tượng resource làm ngữ cảnh (this).
* Trả về đối tượng resource sau khi đã được mở rộng.

Ở phần demo: chúng ta thấy tương tự như ở Object composition, nó sẽ thực hiện xác thực, nếu đúng thì sẽ cho phép tải tài nguyên. Nhưng khi gọi download() ở các đối tượng lớp Resource thì nó đã bị "mutate" (thay đổi) so với ban đầu là nó đã thêm việc xác thực user

Phương pháp Object Augmentation trong trường hợp này được sử dụng để mở rộng (augment) đối tượng Resource để triển khai xác thực người dùng trước khi cho phép truy cập vào phương thức download(). Phương pháp này có ưu điểm và nhược điểm sau:

* Ưu điểm:
  + Đơn giản và dễ hiểu: Object augmentation là một phương pháp đơn giản và dễ hiểu để mở rộng chức năng của đối tượng.
  + Tiện lợi và linh hoạt: Bạn có thể dễ dàng mở rộng và thay đổi chức năng của đối tượng trong quá trình chạy.
  + Không yêu cầu tạo lớp mới: Không cần phải tạo ra một lớp mới (như Proxy) để triển khai chức năng bổ sung. Điều này giúp tránh việc tạo ra quá nhiều lớp và giữ mã nguồn gọn gàng hơn.
* Nhược điểm:
  + Ghi đè trực tiếp lên đối tượng gốc: Phương pháp này ghi đè trực tiếp lên đối tượng gốc. Điều này có thể gây xung đột và ảnh hưởng đến các phần khác của mã sử dụng đối tượng đó.
  + Khó quản lý và theo dõi: Vì việc mở rộng đối tượng xảy ra tại nhiều điểm khác nhau trong mã, việc quản lý và theo dõi các thay đổi có thể trở nên khó khăn.
  + Không có kiểm soát truy cập: Phương pháp này không cung cấp cơ chế kiểm soát truy cập tường minh. Việc xác thực và kiểm tra truy cập được thực hiện trong mã mở rộng, và không có quy tắc cụ thể để kiểm soát việc truy cập vào phương thức hay thuộc tính.
  + Khả năng xâm phạm bảo mật: Một kẻ tấn công có thể ghi đè và thay đổi mã mở rộng để bypass xác thực và có quyền truy cập không hợp lệ.

Với những ưu nhược điểm trên, việc sử dụng phương pháp Object augmentation nên được cân nhắc dựa trên quy mô và yêu cầu cụ thể của dự án để đảm bảo tính bảo mật và quản lý mã nguồn hiệu quả

### **Sử dụng class Proxy trong javascript**

**Code**

index.js:

import { Resource } from "../Resource.js"

import { User } from "../User.js";

const safeResourceHandler = {

    get: (target, property) => {

        if (property === 'download') {

            return function () {

                if (target.user.username !== "nva" || target.user.password !== "123") {

                    console.log("You do not have permission to download");

                    return;

                }

                return target.download();

            }

        }

        return target[property];

    }

};

const user01 = new User("nva", "123");

const user02 = new User("admin", "123");

const resource = new Resource("Music", user01);

const safeResource = new Proxy(

    resource,

    safeResourceHandler

)

safeResource.download()

**Giải thích**

Trong bản ES2015 của javascript, có giới thiệu đến cách để tạo 1 đối tượng proxy mạnh mẽ với cú pháp:

const proxy = new Proxy(target, handler)

Trong đó:

* Đối tượng target đại diện cho đối tượng mà proxy được áp dụng lên (tức là subject trong định nghĩa cơ bản), trong khi handler là một đối tượng đặc biệt xác định hành vi của proxy.
* Đối tượng handler chứa một loạt các phương thức tùy chọn có tên được xác định trước gọi là các phương thức trap (ví dụ như apply, get, set và has) sẽ được tự động gọi khi các hoạt động tương ứng được thực hiện trên thể hiện proxy.

### **So sánh giữa composition và object augumentation**

Composition: Composition được xem là một cách đơn giản và an toàn để tạo ra một proxy vì nó không thay đổi đối tượng gốc và không thay đổi hành vi ban đầu của nó. Tuy nhiên, một nhược điểm duy nhất của phương pháp này là chúng ta phải thủ công ủy quyền tất cả các phương thức, ngay cả khi chúng ta chỉ muốn proxy một phương thức duy nhất. Ngoài ra, chúng ta cũng có thể phải ủy quyền truy cập vào các thuộc tính của đối tượng gốc.

Object Augmentation: ngược lại, thay đổi đối tượng gốc, điều này có thể không luôn là lựa chọn lý tưởng, nhưng nó không gặp các bất tiện liên quan đến việc ủy quyền. Vì lý do này, Object Augmentation thường là kỹ thuật được ưa thích trong những tình huống mà việc sửa đổi đối tượng gốc là một lựa chọn.

Tuy nhiên, có ít nhất một tình huống mà Composition gần như là bắt buộc; đó là khi chúng ta muốn kiểm soát việc khởi tạo của đối tượng gốc.

Cuối cùng, đối tượng Proxy là phương pháp tiếp cận được sử dụng nếu bạn cần chặn các lời gọi hàm hoặc có các loại truy cập khác nhau vào các thuộc tính đối tượng, bao gồm các thuộc tính động. Đối tượng Proxy cung cấp một cấp độ tiếp cận cao cấp không có sẵn trong các kỹ thuật khác.

### **Xây dựng Proxy các đối tượng có sẵn trong javascript**

**Code**

date.js

export function createDateTime(datetime) {

    return new Proxy(datetime, { // (1)

        get(target, propKey, receiver) { // (2)

            if (propKey === 'getMonth') { // (3)

                return function () { // (4)

                    return datetime.getMonth() + 1

                }

            }

            return target[propKey] // (5)

        }

    })

}

index.js

import { createDateTime } from "./date.js"

const now = new Date();

const dateProxy = createDateTime(now)

console.log(dateProxy.getMonth())

**Giải thích**

Một đối dượng kiểu Date trong javascript khi getMonth() có giá trị từ 0 – 11 tương ứng với tháng 1 đến tháng 12. Chúng ta thực hiện tạo 1 proxy để chỉnh đầu ra theo đúng thực tế tháng 1 – tháng 12.

Trong date.js:

* Hàm createDateTime nhận đối tượng datetime (là một đối tượng Date) làm đối số và trả về một proxy cho đối tượng này.
* Trong proxy được tạo ra, get là một phương thức được sử dụng để xử lý việc truy cập các thuộc tính của đối tượng proxy.
* Trong phương thức get, chúng ta kiểm tra nếu propKey (tên thuộc tính được truy cập) là 'getMonth'.
* Nếu điều kiện trên đúng, chúng ta trả về một hàm mới trong đó chúng ta trả về kết quả của datetime.getMonth() + 1.
* Nếu không, chúng ta truy cập thuộc tính tương ứng trên đối tượng datetime bằng cách sử dụng target[propKey] và trả về giá trị tương ứng.

Trong index.js:

* Chúng ta nhập hàm createDateTime từ module date.js.
* Chúng ta khởi tạo một đối tượng now từ Date(), đại diện cho thời điểm hiện tại.
* Chúng ta sử dụng hàm createDateTime để tạo một proxy cho đối tượng now, và lưu trữ kết quả trong biến dateProxy.
* Chúng ta gọi phương thức getMonth() trên dateProxy.
* Kết quả sẽ được in ra màn hình console, là giá trị trả về của datetime.getMonth() + 1.

Với code trên chúng ta có thể kiểm soát và thêm logic bổ sung khi truy cập vào thuộc tính getMonth() của đối tượng Date thông qua việc sử dụng mẫu Proxy.

### **Thay đổi Observer với Proxy**

**Code**

create-observable.js

export function createObservable(target, observer) {

    const observable = new Proxy(target, {

        set(obj, prop, value) {

            if (value !== obj[prop]) {

                const prev = obj[prop]

                obj[prop] = value

                observer({ prop, prev, curr: value })

            }

            return true

        }

    })

    return observable

}

index.js

import { createObservable } from './create-observable.js'

function area(rectangle) {

    return rectangle.height \* rectangle.width;

}

const rectangle = {

    height: 32,

    width: 10

}

let result = area(rectangle)

console.log(`Starting result: ${result}`)

const obsRectangle = createObservable(

    rectangle,

    ({ prop, prev, curr }) => {

        result = area(rectangle)

        console.log(`Area: ${result} (${prop} changed: ${prev} -> ${curr})`)

    }

)

obsRectangle.height = 10

obsRectangle.width = 5

console.log(`Final total: ${result}`)

**Giải thích**

Trong đoạn mã trên, chúng ta tạo một hàm createObservable được sử dụng để tạo một đối tượng proxy được gọi là observable. Đối tượng observable này sẽ theo dõi sự thay đổi của các thuộc tính trong đối tượng target và thông báo cho một hàm gọi là observer mỗi khi có sự thay đổi xảy ra.

Hàm area được định nghĩa để tính diện tích của một hình chữ nhật. Nó nhận vào một đối tượng rectangle có thuộc tính height và width và trả về diện tích của hình chữ nhật đó.

Đối tượng rectangle được khởi tạo với các thuộc tính height và width.

Hàm createObservable được sử dụng để tạo một đối tượng proxy obsRectangle từ đối tượng rectangle. Proxy này sẽ theo dõi sự thay đổi của các thuộc tính trong rectangle và gọi hàm observer để thông báo về các thay đổi này. Trong trường hợp này, observer là một hàm được định nghĩa ngay tại đoạn mã và nhận một đối số gồm các thông tin về thuộc tính đã thay đổi.

Sau đó, chúng ta thực hiện một số thay đổi trên obsRectangle.height và xem kết quả. Mỗi lần obsRectangle.height obsRectangle.width hoặc thay đổi, hàm observer sẽ được gọi và tính lại diện tích của hình chữ nhật thông qua hàm area. Kết quả diện tích mới sẽ được hiển thị trên console cùng với thông tin về thuộc tính đã thay đổi.

Cuối cùng, chúng ta in ra kết quả tổng cộng cuối cùng của biến result sau khi các thay đổi đã được áp dụng.

Tóm lại, đoạn mã trên sử dụng proxy để theo dõi sự thay đổi của thuộc tính trong đối tượng rectangle và thực hiện một hành động tương ứng mỗi khi có sự thay đổi xảy ra.

## Decorator

### **Định nghĩa**

Mẫu Decorator là một mẫu thiết kế phần mềm thuộc nhóm các mẫu thiết kế cấu trúc. Nó cho phép bạn mở rộng hoặc thay đổi chức năng của một đối tượng mà không cần thay đổi cấu trúc ban đầu của nó.

Decorator cho phép bọc một đối tượng gốc trong một lớp decorator mới, giúp mở rộng tính năng của đối tượng ban đầu thông qua việc thêm các chức năng mới vào đối tượng đó. Lớp decorator mới thực hiện cùng giao diện hoặc kế thừa từ lớp gốc, nhưng cung cấp một bản mở rộng của các phương thức hoặc thêm các phương thức mới.

Mẫu Decorator cho phép thêm, gỡ bỏ hoặc thay đổi chức năng của một đối tượng trong thời gian chạy mà không ảnh hưởng đến các đối tượng khác có cùng giao diện. Điều này giúp bạn linh hoạt trong việc thay đổi hành vi của một đối tượng mà không cần sửa đổi mã nguồn ban đầu.

Tổng quan, mẫu Decorator giúp bạn mở rộng tính năng của một đối tượng mà không làm thay đổi cấu trúc ban đầu, tăng tính linh hoạt và tái sử dụng mã nguồn.

### **Composition**

**Code**

CoffeeBeverage.js:

export class CoffeeBeverage {

    constructor() { }

    getCost() {

        return 10;

    }

    getDescription() {

        return "Coffee";

    }

}

CoffeeDecorator.js:

export class CoffeeDecorator {

    constructor(coffee) {

        this.coffee = coffee;

    }

    getCost() {

        return this.coffee.getCost();

    }

    getDescription() {

        return this.coffee.getDescription();

    }

}

MilkDecorator.js:

import { CoffeeDecorator } from "./CoffeeDecorator.js";

export class MilkDecorator extends CoffeeDecorator {

    constructor(coffee) {

        super(coffee);

    }

    getCost() {

        return this.coffee.getCost() + 2;

    }

    getDescription(description) {

        return this.coffee.getDescription() + " + Milk"

    }

}

index.js:

import { CoffeeBeverage } from './CoffeeBeverage.js';

import { MilkDecorator } from './MilkDecorator.js';

const coffee = new CoffeeBeverage();

const coffeeMilk = new MilkDecorator(coffee);

console.log(coffeeMilk.getCost());

console.log(coffeeMilk.getDescription());

**Giải thích**

Trong CoffeeBeverage.js:

* CoffeeBeverage là một lớp đại diện cho một đối tượng cơ bản, trong trường hợp này là đối tượng "Coffee". Nó có hai phương thức: getCost() để trả về giá của Coffee và getDescription() để trả về mô tả của Coffee.

Trong CoffeeDecorator.js:

* CoffeeDecorator là một lớp decorator trừu tượng. Nó có một constructor nhận đối tượng Coffee gốc làm đối số.
* Lớp CoffeeDecorator triển khai hai phương thức getCost() và getDescription() và chuyển tiếp các cuộc gọi tương ứng cho đối tượng Coffee gốc.

Trong MilkDecorator.js:

* MilkDecorator là một lớp decorator cụ thể kế thừa từ CoffeeDecorator. Nó mở rộng các chức năng của CoffeeDecorator.
* Lớp MilkDecorator triển khai lại phương thức getCost() và getDescription() để thêm tính năng sữa (Milk) vào đối tượng Coffee. Giá của Coffee với sữa được tính bằng cách thêm 2 đơn vị vào giá gốc, và mô tả của Coffee cũng được bổ sung với chuỗi "+ Milk".

Trong index.js:

* Chúng ta nhập lớp CoffeeBeverage từ module CoffeeBeverage.js và lớp MilkDecorator từ module MilkDecorator.js.
* Chúng ta khởi tạo một đối tượng coffee từ lớp CoffeeBeverage.
* Chúng ta khởi tạo một đối tượng coffeeMilk từ lớp MilkDecorator và truyền đối tượng coffee làm đối số.
* Chúng ta gọi các phương thức getCost() và getDescription() trên coffeeMilk để lấy giá và mô tả của Coffee với sữa.
* Kết quả được in ra màn hình console.

Với mẫu Decorator, chúng ta có thể dễ dàng mở rộng chức năng của đối tượng gốc (trong trường hợp này là Coffee) bằng cách sử dụng các decorator khác nhau (trong trường hợp này là MilkDecorator). Decorator cho phép chúng ta bổ sung tính năng mới mà không cần sửa đổi đối tượng gốc, đồng thời duy trì tính linh hoạt và tái sử dụng mã nguồn.

### **Object augmentation**

**Code**

index.js

import { CoffeeBeverage } from "../../CoffeeBeverage.js";

function milkCoffee(coffee) {

    const cost = coffee.getCost()

    const description = coffee.getDescription()

    coffee.getCost = function () {

        return cost + 2;

    }

    coffee.getDescription = () => {

        return description + " + Milk"

    }

    return coffee;

}

const coffee = new CoffeeBeverage();

const milkCf = milkCoffee(coffee)

console.log(milkCf.getDescription())

console.log(milkCf.getCost())

**Giải thích**

Trong đoạn mã, chúng ta có một đối tượng CoffeeBeverage đại diện cho một loại cà phê. Chúng ta muốn thêm chức năng là thêm sữa vào cà phê này mà không phải thay đổi lớp CoffeeBeverage gốc.

Để làm điều đó, chúng ta tạo một hàm milkCoffee nhận đối tượng coffee làm tham số. Trong hàm này, chúng ta lưu trữ giá trị hiện tại của cost và description bằng cách gọi các phương thức getCost() và getDescription() của đối tượng coffee.

Sau đó, chúng ta ghi đè phương thức getCost() bằng cách tạo một hàm mới. Trong hàm này, chúng ta sử dụng giá trị cost đã lưu trữ trước đó và thêm 2 vào nó, đại diện cho giá trị sữa được thêm vào.

Tương tự, chúng ta cũng ghi đè phương thức getDescription() bằng cách tạo một hàm mới. Trong hàm này, chúng ta sử dụng giá trị description đã lưu trữ trước đó và thêm chuỗi " + Milk" vào cuối, đại diện cho việc thêm sữa vào mô tả của cà phê.

Cuối cùng, chúng ta trả về đối tượng coffee đã được mở rộng và sử dụng nó để in ra mô tả và giá của cà phê đã thêm sữa.

Decorate Pattern cho phép chúng ta mở rộng chức năng của đối tượng mà không làm thay đổi đối tượng gốc, và cũng giúp chúng ta linh hoạt trong việc thêm nhiều tính năng mở rộng khác nhau cho các đối tượng.

### **Decorating with the Proxy object**

**Code**

MilkDecoratorProxy.js:

import { CoffeeBeverage } from "../../CoffeeBeverage.js"

const milkDecoratorHandler = {

    get(target, property) {

        if (property === "getCost") {

            return function getCost() {

                return target.getCost() + 2

            }

        }

        else if (property === "getDescription") {

            return function getDescription() {

                return target.getDescription() + " + Milk"

            }

        }

        return target[property]

    }

}

const coffee = new CoffeeBeverage()

const coffeeProxy = new Proxy(coffee, milkDecoratorHandler)

console.log(coffeeProxy.getCost())

console.log(coffeeProxy.getDescription())

**Giải thích**

Chúng ta tạo một đối tượng milkDecoratorHandler là một handler của Proxy. Handler này chứa một hàm get được sử dụng để xử lý việc gọi các phương thức trên đối tượng gốc.

Trong hàm get của milkDecoratorHandler, chúng ta kiểm tra xem thuộc tính được yêu cầu là "getCost" hay "getDescription". Nếu là "getCost", chúng ta trả về một hàm mới được định nghĩa, mà trong đó chúng ta gọi phương thức getCost() trên đối tượng gốc và cộng thêm 2 đơn vị.

Tương tự, nếu thuộc tính là "getDescription", chúng ta trả về một hàm mới mà trong đó chúng ta gọi phương thức getDescription() trên đối tượng gốc và bổ sung chuỗi "+ Milk" vào mô tả.

Cuối cùng, nếu không phải là "getCost" hay "getDescription", chúng ta trả về thuộc tính tương ứng từ đối tượng gốc bằng cách sử dụng target[property].

Tiếp theo, chúng ta khởi tạo một đối tượng coffee từ lớp CoffeeBeverage.

Sau đó, chúng ta tạo một đối tượng coffeeProxy bằng cách sử dụng Proxy và truyền đối tượng coffee và milkDecoratorHandler làm đối số.

Cuối cùng, chúng ta gọi các phương thức getCost() và getDescription() trên coffeeProxy để lấy giá và mô tả của Coffee với sữa.

Kết quả được in ra màn hình console.

Qua ví dụ này, chúng ta thấy Proxy được sử dụng để tạo ra một decorator cho đối tượng coffee, cho phép chúng ta mở rộng chức năng của đối tượng mà không cần thay đổi đối tượng gốc.

## Adapter

### **8.3.1 Định nghĩa**

Mẫu Adapter là một mẫu thiết kế phần mềm cho phép các đối tượng với các giao diện không tương thích hoặc không tương thích được làm việc cùng nhau thông qua một adapter.

Mục đích chính của mẫu Adapter là chuyển đổi giao diện của một lớp thành một giao diện khác mà các client mong muốn hoặc cần sử dụng.

Một số thành phần quan trọng trong mẫu Adapter bao gồm:

* Target: Định nghĩa giao diện mà client mong muốn sử dụng.
* Adapter: Chuyển đổi giao diện của đối tượng không tương thích thành giao diện mà client mong muốn sử dụng.
* Adaptee: Đối tượng hoặc thành phần không tương thích mà Adapter chuyển đổi để client có thể sử dụng.

Một số ứng dụng của mẫu Adapter bao gồm:

* Khi muốn sử dụng một lớp đã tồn tại có giao diện không tương thích trong một ngữ cảnh khác.
* Khi tái sử dụng các lớp không liên quan nhưng có chức năng tương tự.
* Khi tạo một giao diện thống nhất cho các lớp có giao diện không tương thích để client có thể sử dụng chúng một cách đồng nhất.

Mẫu Adapter giúp tăng tính linh hoạt, sự tái sử dụng và tương thích của các thành phần phần mềm. Nó cho phép các thành phần không tương thích làm việc cùng nhau một cách dễ dàng và mượt mà.

### **8.3.2 Code**

// Target interface

class CelsiusTemperature {

    constructor(temp) {

        this.temp = temp;

    }

    getTemperature() {

        return this.temp;

    }

}

class FahrenheitTemperature {

    constructor(temp) {

        this.temp = temp;

    }

    getTemperature() {

        return this.temp;

    }

}

// Adapter

class CelsiusToFahrenheitAdapter extends FahrenheitTemperature {

    constructor(celsiusTemp) {

        super();

        this.celsiusTemp = celsiusTemp;

    }

    getTemperature() {

        const celsius = this.celsiusTemp.getTemperature();

        const fahrenheit = (celsius \* 9) / 5 + 32;

        return fahrenheit;

    }

}

// Client

const celsiusTemp = new CelsiusTemperature(25);

const fahrenheitTemp = new FahrenheitTemperature(25);

const adapter = new CelsiusToFahrenheitAdapter(celsiusTemp);

console.log(`Celsius: ${celsiusTemp.getTemperature()}°C`);

console.log(`Fahrenheit: ${fahrenheitTemp.getTemperature()}°F`);

console.log(`Adapter: ${celsiusTemp.getTemperature()}°C -> ${adapter.getTemperature()}°F`);

### **8.3.3 Giải thích**

Trong đoạn code trên, chúng ta có hai lớp CelsiusTemperature và FahrenheitTemperature, mỗi lớp đại diện cho một đối tượng nhiệt độ dạng Celsius và Fahrenheit tương ứng. Cả hai lớp này đều có một phương thức getTemperature() để lấy giá trị nhiệt độ.

Sau đó, chúng ta có lớp CelsiusToFahrenheitAdapter kế thừa từ FahrenheitTemperature. Adapter này nhận một đối tượng CelsiusTemperature làm đầu vào và triển khai phương thức getTemperature() để chuyển đổi nhiệt độ từ Celsius sang Fahrenheit.

Trong phần client, chúng ta tạo các đối tượng celsiusTemp và fahrenheitTemp tương ứng với nhiệt độ Celsius và Fahrenheit ban đầu. Sau đó, chúng ta tạo một đối tượng CelsiusToFahrenheitAdapter với celsiusTemp làm đầu vào.

Cuối cùng, chúng ta hiển thị nhiệt độ ban đầu của celsiusTemp và fahrenheitTemp bằng cách gọi phương thức getTemperature(). Đồng thời, chúng ta cũng hiển thị kết quả của adapter bằng cách gọi phương thức getTemperature() của adapter. Kết quả sẽ được in ra trên console.

Với ví dụ này, chúng ta sử dụng mẫu Adapter để chuyển đổi nhiệt độ từ Celsius sang Fahrenheit bằng cách tạo một adapter có thể hoạt động với giao diện của lớp FahrenheitTemperature. Điều này cho phép chúng ta sử dụng adapter để thực hiện chuyển đổi mà không cần thay đổi giao diện của FahrenheitTemperature.

Ưu điểm của mẫu Adapter:

* Tính linh hoạt: Adapter pattern cho phép các đối tượng không tương thích làm việc cùng nhau bằng cách chuyển đổi giao diện và hành vi của chúng.
* Tái sử dụng và mở rộng: Mẫu Adapter cho phép tái sử dụng mã nguồn hiện có bằng cách tạo một adapter mới cho các giao diện không tương thích. Điều này giúp mở rộng chức năng của các đối tượng mà không cần thay đổi mã nguồn gốc.
* Tách biệt và đơn giản hóa: Adapter pattern giúp tách biệt sự phụ thuộc giữa các lớp không tương thích và giúp đơn giản hóa kiến trúc và mã nguồn.

Nhược điểm của mẫu Adapter:

* Tăng phức tạp: Việc thêm một lớp Adapter có thể tăng phức tạp cho mã nguồn, đặc biệt khi cần chuyển đổi giữa nhiều loại Adapter.
* Hiệu suất: Việc sử dụng Adapter có thể ảnh hưởng đến hiệu suất của ứng dụng do việc thêm một lớp trung gian trong quá trình gọi phương thức.

Tuy nhiên, với các tình huống mà chúng ta cần tích hợp các thành phần không tương thích và tận dụng lại mã nguồn hiện có, mẫu Adapter vẫn là một lựa chọn mạnh mẽ và linh hoạt.

# CHƯƠNG 9:

## 9.1 Strategy

Mẫu **Strategy** là một mẫu thiết kế phần mềm thuộc nhóm cấu trúc (behavioral design patterns) trong lĩnh vực phát triển phần mềm. Mẫu này cho phép một đối tượng, được gọi là bối cảnh, hỗ trợ các biến thể theo logic của nó bằng cách trích xuất các phần biến số thành các đối tượng riêng biệt, có thể hoán đổi cho nhau được gọi là các chiến lược.

Mẫu Strategy đặc biệt hữu ích trong tất cả các tình huống hỗ trợ các biến thể trong hành vi của một thành phần yêu cầu logic điều kiện phức tạp (nhiều câu lệnh if...else hoặc switch ) hoặc trộn các thành phần khác nhau của cùng một họ

### **Code:**

*config.js:*

import { *promises* as fs } from 'fs'

import objectPath from 'object-path'

export class Config {

constructor (*formatStrategy*) {

*this*.data = {}

*this*.formatStrategy = *formatStrategy*

  }

  get (*configPath*) {

    return objectPath.get(*this*.data, *configPath*)

  }

  set (*configPath*, *value*) {

    return objectPath.set(*this*.data, *configPath*, *value*)

  }

  async load (*filePath*) {

    console.log(`Dang load file tu ${*filePath*}`)

*this*.data = *this*.formatStrategy.deserialize(

      await *fs*.readFile(*filePath*, 'utf-8')

    )

    // eslint-disable-next-line quotes

    console.log(`Load File ${*filePath*} Thanh Cong\n`)

  }

  async save (*filePath*) {

    console.log(`Dang tai file ve tu ${*filePath*}`)

    await *fs*.writeFile(*filePath*,

*this*.formatStrategy.serialize(*this*.data))

    // eslint-disable-next-line quotes

    console.log(`Load File ${*filePath*} Thanh Cong\n`)

  }

}

Doạn code cho thấy

* Chúng ta tạo một biến thể hiện có tên là **constructor** để chứa dữ liệu cấu hình. Sau đó lưu trữ formatStrategy đại diện cho thành phần sử dụng để phân tích cú pháp và tuần tự hóa dữ liệu.
* Phương thức: set() và get(), để truy cập các thuộc tính cấu hình bằng ký hiệu đường dẫn chấm (ví dụ: property.subProperty) bằng cách tận dụng thư viện ‘object-path’
* Phương thức load() và save() là nơi chúng ta ủy thác, tương ứng, việc giải tuần tự hóa và tuần tự hóa dữ liệu cho chiến lược của chúng ta. Tại đây, tha có thể thay đổi logic của lớp Config dựa trên formatStrategy được truyền vào từ hàm tạo

*Strategy.js*

import ini from 'ini'

export const iniStrategy = {

  deserialize: *data* => ini.parse(*data*),

  serialize: *data* => ini.stringify(*data*)

}

export const jsonStrategy = {

  deserialize: *data* => JSON.parse(*data*),

  serialize: *data* => JSON.stringify(*data*, null, '  ')

}

* Phân tích cú pháp và định dạng tệp dữ liệu INI và JSON

*Index.js*

import { Config } from './config.js'

import { jsonStrategy, iniStrategy } from './strategies.js'

async function main () {

  const iniConfig = **new** Config(iniStrategy)

  await iniConfig.load('samples/conf.ini')

  iniConfig.set('book.nodejs', 'design patterns')

  await iniConfig.save('samples/conf\_mod.ini')

  const jsonConfig = **new** Config(jsonStrategy)

  await jsonConfig.load('samples/conf.json')

  jsonConfig.set('book.nodejs', 'design patterns')

  await jsonConfig.save('samples/conf\_mod.json')

}

main()

* Tải dũ liệu và lưu cấu hình bằng hai định dạng ini và json

**Kết quả:**

A screenshot of a computer program

Description automatically generated with medium confidence

### **Giải thích**

Mẫu thiết kế "Strategy Pattern" cho phép chọn một trong các thuật toán khác nhau tại thời điểm chạy thay vì lựa chọn một thuật toán cố định tại thời điểm biên dịch. Trong đoạn code trên Strategy.js sẽ định nghĩa để biểu diễn các thuật toán xử lý định dạng cấu hình khác nhau. Các lớp cụ thể JSONOutputStrategy, YAMLOutputStrategy, và INIOutputStrategy triển khai interface này để cung cấp các thuật toán xử lý định dạng JSON, YAML và INI tương ứng.

Ưu điểm của mẫu thiết kế:

* + Cho phép tách biệt các thuật toán xử lý khác nhau và giúp dễ dàng thay đổi thuật toán ở thời điểm chạy.
  + Giúp giảm sự phụ thuộc giữa các phần của chương trình và làm cho mã dễ dàng bảo trì và mở rộng.
  + Có thể tái sử dụng các thuật toán xử lý khác nhau cho các tình huống khác nhau mà không cần viết lại mã.

Nhược điểm của mẫu thiết kế:

* + Cần phải triển khai một interface hoặc lớp trừu tượng để biểu diễn các thuật toán xử lý khác nhau, điều này có thể làm tăng độ phức tạp của mã.
  + Cần phải quản lý các lớp triển khai thuật toán khác nhau, nhất là nếu số lượng các thuật toán tăng lên, điều này có thể làm tăng kích thước và độ phức tạp của mã.

## 9.2 State

Mẫu **State** là một mẫu thiết kế phần mềm thuộc nhóm cấu trúc (behavioral design patterns) trong lĩnh vực phát triển phần mềm. Mẫu này là một chuyên môn hóa của mẫu Strategy Pattern trong đó chiến lược thay đổi tùy thuộc vào từng trạng thái của bối cảnh.

Ta có thể nấy mẫu Strategy Pattern sẽ không thay đổi trong phần còn lại của vòng đời của đối tượng bối cảnh. Trong khi mấu State Pattern có thể thay đổi trong suốt thời gian tồn tại của ngữ cảnh, do đó nó có thể cho phép hành vi của nó thích ứng tùy thuộc vào từng trạng thái bên trong của ngữ cảnh. Các đối tượng cótheer chuyển đổi qua từng trạng thái tại từng thời điểm khác nhau, vì vậy mà ta có thể áp dụng những hành vi khác nhau dựa trên từng trạng thái của nó.

### **Code:**

*failsafeSocket.js:*

import { OfflineState } from './offlineState.js'

import { OnlineState } from './onlineState.js'

export class FailsafeSocket {

  constructor (*options*) {

*this*.options = *options*

*this*.queue = []

*this*.currentState = null

*this*.socket = null

*this*.states = {

      offline: **new** OfflineState(*this*),

      online: **new** OnlineState(*this*)

    }

*this*.changeState('offline')

  }

  changeState (*state*) {

    console.log(`Activating state: ${*state*}`)

*this*.currentState = *this*.states[*state*]

*this*.currentState.activate()

  }

  send (*data*) {

*this*.currentState.send(*data*)

  }

}

Lớp này đại diện cho một đối tượng socket, có khả năng tự động chuyển đổi giữa hai trạng thái kết nối: trạng thái online và trạng thái offline.

Các thuộc tính và phương thức:

* + Options: chứa các tùy chọn cho đối tượng socket.
  + Queue": chứa các dữ liệu gửi đi khi đối tượng socket đang ở trạng thái offline.
  + CurrentState": trạng thái hiện tại của đối tượng socket
  + Socket: Đây chính là đối tượng socket
  + States: chứa hai đối tượng trạng thái: trạng thái offline và trạng thái online.
  + ChangeState: được sử dụng để thay đổi trạng thái của đối tượng socket. Phương thức này nhận đối số là tên trạng thái mới và kích hoạt trạng thái mới đó.
  + Send: Cho phép gửi dữ liệu đi

*offlineState.js*

import jsonOverTcp from 'json-over-tcp-2'

export class OfflineState {

  constructor (*failsafeSocket*) {

*this*.failsafeSocket = *failsafeSocket*

  }

  send (*data*) {

*this*.failsafeSocket.queue.push(*data*)

  }

  activate () {

    const retry = () => {

      setTimeout(() => *this*.activate(), 1000)

    }

    console.log('Đang truy cập...')

*this*.failsafeSocket.socket = jsonOverTcp.connect(

*this*.failsafeSocket.options,

      () => {

        console.log('Truy cập thành công')

*this*.failsafeSocket.socket.removeListener('error', retry)

*this*.failsafeSocket.changeState('online')

      }

    )

*this*.failsafeSocket.socket.once('error', retry)

  }

}

Mô hình hóa của đối tượng socket khi ngoại tuyến. Cứ sau 1 giấy, đối tượng sẽ cố gắng truy cập lại nếu bị ngắt kết nối và Thông báo truy cập thành công sau khi kết nối thành công, tại đây, trạng thái của state cũng được chuyển qua trực truyến.

*onlineState.js*

export class OnlineState {

  constructor (*failsafeSocket*) {

*this*.failsafeSocket = *failsafeSocket*

*this*.hasDisconnected = false

  }

  send (*data*) {

*this*.failsafeSocket.queue.push(*data*)

*this*.\_safeWrite(*data*)

  }

  \_safeWrite (*data*) {

*this*.failsafeSocket.socket.write(*data*, (*err*) => {

      if (!*this*.hasDisconnected && !*err*) {

*this*.failsafeSocket.queue.shift()

      }

    })

  }

  activate () {

*this*.hasDisconnected = false

    for (const data of *this*.failsafeSocket.queue) {

*this*.\_safeWrite(data)

    }

*this*.failsafeSocket.socket.once('error', () => {

*this*.hasDisconnected = true

*this*.failsafeSocket.changeState('offline')

    })

  }

}

Mô hình hóa hành vi của state khi máy chủ họat động. Tại đây, đối tượng socket gửi tất cả các dữ liệu trong mảng "queue" đến máy chủ bằng cách gọi phương thức "\_safeWrite". Nếu xảy ra lỗi kết nối, đối tượng socket sẽ lập tức chuyển sang trạng thái offline bằng cách gọi phương thức "changeState" của đối tượng socket gốc. Nếu không có lỗi xảy ra, dữ liệu được gửi thành công sẽ được xóa khỏi mảng "queue" của đối tượng socket.

*Shop.js*

import jsonOverTcp from 'json-over-tcp-2'

const shop = jsonOverTcp.createServer({ port: 5000 })

shop.on('connection', *socket* => {

*socket*.on('data', *data* => {

    console.log('Thông tin về bộ nhớ', *data*)

  })

})

shop.listen(5000, () => console.log('Shop started'))

Đoạn mã trên tạo một máy chủ TCP đơn giản bằng cách sử dụng thư viện "json-over-tcp-2". Máy chủ này lắng nghe các kết nối trên cổng 5000 và in ra thông tin về bộ nhớ của các thông điệp được gửi đến từ khách hàng.

Khi máy chủ được khởi động, nó bắt đầu lắng nghe kết nối trên cổng 5000 bằng cách gọi phương thức "listen". Khi có kết nối được thiết lập đến máy chủ, sự kiện "connection" sẽ được kích hoạt và hàm xử lý sự kiện được đăng ký sẽ được gọi mỗi khi đối tượng socket nhận dữ liệu từ khách hàng.

*Oder.js*

import { FailsafeSocket } from './failsafeSocket.js'

const failsafeSocket = **new** FailsafeSocket({ port: 5000 })

setInterval(() => {

  // send current memory usage

  failsafeSocket.send(process.memoryUsage())

}, 1000)

Đoạn mã trên tạo một đối tượng socket mới của lớp "FailsafeSocket" và thiết lập để gửi thông tin về bộ nhớ sử dụng hiện tại đến máy chủ mỗi giây một lần. Tại đây, sẽ thực hiện tính toán mức sử dụng bộ nhớ cho mỗi đơn hàng và đóng gói lại dưới dạng JSON sau đó gửi đến máy chủ thông qua đối tượng socket. Tại đây máy chủ sẽ in ra mức sử dụng bộ nhớ cho từng hóa đơn.

### **Giải thích**

* **Oder.js** chứa mã nguồn của Khách hàng, kết nối đến shop thông qua địa chỉ và cổng được cấu hình trước đó. Client sẽ gửi thông tin về bộ nhớ sử dụng hiện tại của máy mỗi giây một lần đến server thông qua socket. Các thông tin này được đóng gói lại dưới dạng JSON trước khi gửi đi.
* **Shop.js** chứa mã nguồn của server, chấp nhận các kết nối từ client và lắng nghe các thông tin gửi đến từ client. Nó cũng định nghĩa một số trạng thái khác nhau để xử lý các thông tin này, bao gồm trạng thái kết nối, trạng thái chờ đợi dữ liệu và trạng thái xử lý dữ liệu.
* **State.js** chứa mã nguồn của lớp trạng thái cơ bản, nó định nghĩa một số phương thức để xử lý các sự kiện từ client và server. Các trạng thái khác nhau được triển khai dựa trên lớp này.
* **Connected.js, waiting.js** và **processing.js** chứa mã nguồn của các lớp trạng thái cụ thể, thừa kế từ lớp trạng thái cơ bản. Các lớp này triển khai các phương thức để xử lý các sự kiện tương ứng với từng trạng thái.

Đây là một phương pháp khá hiệu quả trong việc quản lý trạng thái của ứng dụng, bởi vì nó giúp tách biệt việc xử lý các sự kiện tương ứng với từng trạng thái, giúp cho mã nguồn trở nên dễ đọc, dễ bảo trì và dễ mở rộng hơn.

**Ưu điểm:**

1. Giúp tách biệt việc xử lý các sự kiện tương ứng với từng trạng thái, giúp mã nguồn trở nên dễ đọc, dễ bảo trì và dễ mở rộng hơn.
2. Giúp quản lý trạng thái của ứng dụng trở nên dễ dàng hơn, giảm thiểu sự phức tạp của mã nguồn và tăng tính module hóa.
3. Giúp xử lý các sự kiện và tương tác giữa các thành phần của ứng dụng một cách hiệu quả và dễ dàng.
4. Được sử dụng rộng rãi trong các ứng dụng thời gian thực hoặc có khả năng mở rộng lớn.

**Nhược điểm:**

1. Không phù hợp cho các ứng dụng đơn giản, yêu cầu sự tương tác giữa nhiều trạng thái.
2. Có thể dẫn đến việc tạo ra quá nhiều lớp trạng thái, làm cho mã nguồn trở nên phức tạp hơn.
3. Khó khăn trong việc xử lý các sự kiện tương ứng với nhiều trạng thái khác nhau.

## 9.3 Template:

Template pattern xác định một lớp trừu tượng thực hiện khung (đại diện cho các phần chung) của một thành phần, trong đó một số bước của nó không được xác định. Bằng cách triển khai phương thức mẫu thuẫn, các lớp con sau đó có thể lấp đầy các khoảng trống trong thành phần. Mục đích của mẫu này là để có thể định nghĩa một họ các lớp là tất cả các biến thể của một họ các thành phần.

### **Code:**

*configTemplate.js*

import { *promises* as fsPromises } from 'fs'

import objectPath from 'object-path'

export class ConfigTemplate {

  async load (*file*) {

    console.log(`\n Đang load file từ: ${*file}* `)

*this*.data = *this*.\_deserialize(

      await *fsPromises*.readFile(*file*, 'utf-8'))

  }

  async save (*file*) {

    console.log(`Đang lưu file vào ${*file*}\n`)

    await *fsPromises*.writeFile(*file*, *this*.\_serialize(*this*.data))

  }

  get (*path*) {

    return objectPath.get(*this*.data, *path*)

  }

  set (*path*, *value*) {

    return objectPath.set(*this*.data, *path*, *value*)

  }

  \_serialize () {

    throw **new** *Error*('\_serialize() phải được triển khai')

  }

  \_deserialize () {

    throw **new** *Error*('\_deserialize() phải được triển khai')

  }

}

*iniConfig.js*

import { ConfigTemplate } from './configTemplate.js'

import ini from 'ini'

export class IniConfig extends ConfigTemplate {

  \_deserialize (*data*) {

    return ini.parse(*data*)

  }

  \_serialize (*data*) {

    return ini.stringify(*data*)

  }

}

*jsonConfig.js*

import { ConfigTemplate } from './configTemplate.js'

export class JsonConfig extends ConfigTemplate {

  \_deserialize (*data*) {

    return JSON.parse(*data*)

  }

  \_serialize (*data*) {

    return JSON.stringify(*data*, null, '  ')

  }

}

*Index.js*

import { JsonConfig } from './jsonConfig.js'

import { IniConfig } from './iniConfig.js'

async function main () {

  const jsonConfig = **new** JsonConfig()

  await jsonConfig.load('samples/conf.json')

  jsonConfig.set('nodejs', 'design patterns')

  await jsonConfig.save('samples/conf\_mod.json')

  const iniConfig = **new** IniConfig()

  await iniConfig.load('samples/conf.ini')

  iniConfig.set('nodejs', 'design patterns')

  await iniConfig.save('samples/conf\_mod.ini')

}

main()

**Kết Quả**

A screenshot of a computer program

Description automatically generated with medium confidence

### **Giải thích**

Trong đoạn code này, chúng ta có một lớp ConfigTemplate định nghĩa một thuật toán để đọc cấu hình từ một tập tin và trả về một đối tượng JavaScript. Tuy nhiên, cách đọc các tập tin cấu hình khác nhau sẽ khác nhau, vì vậy chúng ta sử dụng mẫu Thiết kế Mẫu để cho phép các lớp con triển khai các bước cụ thể của thuật toán.

Lớp ConfigTemplate định nghĩa hai phương thức: readConfig() và parseData(). Phương thức readConfig() là bước đầu tiên của thuật toán, nó đọc cấu hình từ một tập tin và trả về một chuỗi dữ liệu. Phương thức parseData() là bước tiếp theo, nó nhận vào chuỗi dữ liệu và trả về một đối tượng JavaScript.

Lớp JSONConfig và INIConfig là hai lớp con của lớp ConfigTemplate, chúng triển khai các bước cụ thể để đọc các tập tin cấu hình JSON và INI. Cả hai lớp đều định nghĩa phương thức readConfig() để đọc tập tin cấu hình và chuyển đổi nó thành chuỗi dữ liệu. Tuy nhiên, cách chuyển đổi này khác nhau đối với hai lớp con này, vì vậy chúng ta cần triển khai phương thức parseData() tương ứng để chuyển đổi chuỗi dữ liệu này thành một đối tượng JavaScript.

Lớp JSONConfig sử dụng phương thức JSON.parse() để chuyển đổi chuỗi dữ liệu JSON thành một đối tượng JavaScript. Trong khi đó, lớp INIConfig sử dụng thư viện ini để đọc và chuyển đổi chuỗi dữ liệu INI thành một đối tượng JavaScript.

Kết quả của các lớp con này là một đối tượng JavaScript chứa các giá trị cấu hình được đọc từ tập tin cấu hình. Các giá trị này có thể được truy cập thông qua các thuộc tính của đối tượng này.

Lợi ích của mẫu Thiết kế Mẫu trong đoạn code này là cho phép chúng ta tái sử dụng các bước chung của thuật toán đọc cấu hình, trong khi vẫn cho phép các lớp con triển khai các bước cụ thể của thuật toán. Điều này giúp giảm thiểu mã lặp lại và làm cho code trở nên dễ đọc hơn. Ngoài ra, mẫu Thiết kế Mẫu cũng giúp chúng ta dễ dàng thêm các định dạng tập tin cấu hình mới trong tương lai mà không ảnh hưởng đến cấu trúc tổng thể của thuật toán.

## 9.4: Iterator alphabet iterator:

Trong JavaScript, mẫu Iterator được triển khai thông qua các giao thức thay vì thông qua các cấu trúc chính thức, chẳng hạn như kế thừa. Về cơ bản, điều này có nghĩa là sự tương tác giữa người triển khai và người tiêu dùng mẫu Iterator sẽ giao tiếp bằng cách sử dụng các giao diện và đối tượng có hình dạng được thống nhất trước.

### **Code:**

const OneCharCode = 48

const NineCharCode = 57

function createAlphabetIterator () {

  let currCode = OneCharCode

  return {

    next () {

      const currChar = *String*.fromCodePoint(currCode)

      if (currCode > NineCharCode) {

        return { done: true }

      }

      currCode++

      return { value: currChar, done: false }

    }

  }

}

const iterator = createAlphabetIterator()

let iterationResult = iterator.next()

while (!iterationResult.done) {

  console.log(iterationResult.value)

  iterationResult = iterator.next()

}

**Kết Quả:**

A screenshot of a computer

Description automatically generated

### **Giải thích**

Logic của phép lặp chỉ đơn giản là tại mỗi lần gọi phương thức next() , chúng ta chỉ cần tăng một số đại diện cho mã ký tự của chữ cái, chuyển đổi nó thành một ký tự và sau đó trả về bằng cách sử dụng hình dạng đối tượng được xác định bởi giao thức iterator.

Khía cạnh quan trọng cần lưu ý là một trình vòng lặp thường là một đối tượng có trạng thái vì chúng ta phải theo dõi theo một cách nào đó vị trí hiện tại của phép lặp. Điều này thường tốt hơn về khả năng gỡ lỗi vì chúng ta có thể đọc trạng thái của phép lặp từ chính trình lặp đó bất kỳ lúc nào, nhưng mặt khác, nó không ngăn mã bên ngoài sửa đổi biến đối tượng và do đó can thiệp vào trạng thái của biến đối tượng. phép lặp. Tùy thuộc vào bạn để quyết định ưu và nhược điểm của từng tùy chọn.

Đoạn code trên là một ví dụ về mẫu Iterator trong Node.js, được sử dụng để duyệt các phần tử của một tập hợp mà không cần biết chi tiết về cấu trúc bên trong của tập hợp đó. Chúng ta có một lớp đối tượng AlphabetIterator được định nghĩa bên trong tệp alphabet.js. Lớp này có hai thuộc tính: letters là một mảng các chữ số từ 0 đến 9, và position là vị trí hiện tại của con trỏ trong mảng. Lớp này cũng có hai phương thức quan trọng là next() và hasNext(), được sử dụng để duyệt qua các phần tử của tập hợp.

Phương thức next() trả về phần tử tiếp theo trong tập hợp, và tăng giá trị của position. Phương thức hasNext() kiểm tra xem có phần tử nào còn lại trong tập hợp hay không, bằng cách kiểm tra giá trị của position có nhỏ hơn chiều dài của mảng letters hay không.

Khi sử dụng lớp AlphabetIterator, chúng ta có thể duyệt qua các chữ số từ 0 đến 9 bằng cách sử dụng vòng lặp while và phương thức hasNext() để kiểm tra nếu còn chữ cái nào chưa được duyệt. Trong mỗi lần lặp, chúng ta gọi phương thức next() để lấy chữ cái tiếp theo và in ra màn hình.

**Ưu điểm:**

1. **Tính tái sử dụng:** Mẫu Iterator cho phép tái sử dụng code để duyệt qua các phần tử trong nhiều loại cấu trúc dữ liệu khác nhau, mà không cần phải viết lại code duyệt cho mỗi loại cấu trúc.
2. **Tách biệt logic duyệt và logic xử lý:** Mẫu Iterator tách biệt logic duyệt qua các phần tử của tập hợp và logic xử lý trên từng phần tử. Điều này giúp mã nguồn trở nên rõ ràng hơn, dễ đọc và dễ bảo trì hơn.
3. **Khả năng duyệt qua các phần tử theo nhiều cách khác nhau:** Mẫu Iterator cho phép chúng ta duyệt qua các phần tử theo nhiều cách khác nhau, ví dụ như từ trái sang phải, từ phải sang trái, theo thứ tự ngẫu nhiên, và nhiều cách khác nữa.
4. **Giảm tải bộ nhớ:** Khi sử dụng mẫu Iterator, chúng ta có thể duyệt qua các phần tử của tập hợp một cách nhanh gọn, chỉ lấy từng phần tử cần thiết khi cần sử dụng. Điều này giúp giảm tải bộ nhớ cho ứng dụng, đặc biệt là khi xử lý các tập dữ liệu lớn.

**Nhược điểm**:

1. **Không phù hợp cho các cấu trúc dữ liệu phức tạp:** Mẫu Iterator có thể không phù hợp cho các cấu trúc dữ liệu phức tạp, ví dụ như các cây nhị phân hoặc các tập hợp đa cấp.
2. **Tăng chi phí xử lý:** Việc sử dụng mẫu Iterator có thể tăng chi phí xử lý của ứng dụng, đặc biệt là khi tập hợp có kích thước lớn và phương thức duyệt phải được gọi nhiều lần.

## 9.5 Iterator iterable matrix

Giao thức có thể lặp lại xác định một phương tiện được tiêu chuẩn hóa để một đối tượng trả về một trình vòng lặp. Các đối tượng như vậy được gọi là iterables. Thông thường, một iterable là một thùng chứa các phần tử, chẳng hạn như cấu trúc dữ liệu, nhưng nó cũng có thể là một đối tượng hầu như đại diện cho một tập hợp các phần tử, chẳng hạn như đối tượng Thư mục , cho phép chúng ta lặp lại các tệp trong một thư mục.

Trong JavaScript, chúng ta có thể định nghĩa một iterable bằng cách đảm bảo rằng nó triển khai phương thức “iterator”, hay nói cách khác, một phương thức có thể truy cập thông qua ký hiệu tích hợp sẵn “Symbol.iterator”. Phương thức “iterator” như vậy sẽ trả về một đối tượng iterator, đối tượng này có thể được sử dụng để lặp qua các phần tử được đại diện bởi iterable.

### **Code:**

*Matrix.js:*

export class Matrix {

  constructor (*inMatrix*) {

*this*.data = *inMatrix*

  }

  get (*row*, *column*) {

    if (*row* >= *this*.data.length || *column* >= *this*.data[*row*].length) {

      throw **new** *RangeError*('Out of bounds')

    }

    return *this*.data[*row*][*column*]

  }

  set (*row*, *column*, *value*) {

    if (*row* >= *this*.data.length || *column* >= *this*.data[*row*].length) {

      throw **new** *RangeError*('Out of bounds')

    }

*this*.data[*row*][*column*] = *value*

  }

  [Symbol.iterator] () {

    let nextRow = 0

    let nextCol = 0

    return {

      next: () => {

        if (nextRow === *this*.data.length) {

          return { done: true }

        }

        const currVal = *this*.data[nextRow][nextCol]

        if (nextCol === *this*.data[nextRow].length - 1) {

          nextRow++

          nextCol = 0

        } else {

          nextCol++

        }

        return { value: currVal }

      }

    }

  }

}

*Index.js:*

import { Matrix } from './matrix.js'

const matrix2x2 = **new** Matrix([

  ['Nguyễn Viết Quang', '25/03/2001', '04 CLC', 'Đesign Pattern'],

  ['Nguyễn Vinh Khang', '14/02/2002', '04 CLC', 'Đesign Pattern'],

  ['Nguyễn Van An', '16/07/2001', '04 CLC', 'Đesign Pattern'],

  ['Nguyễn Viết Đức', '06/10/2002', '04 CLC', 'Đesign Pattern']

])

const iterator = matrix2x2[Symbol.iterator]()

let iterationResult = iterator.next()

while (!iterationResult.done) {

  console.log(iterationResult.value)

  iterationResult = iterator.next()

}

**Kết quả:**

A screenshot of a computer program

Description automatically generated with medium confidence

### **Giải thích**

Phương thức iterator sẽ trả về một trình vòng lặp, như được chỉ định bởi giao thức có thể lặp lại và một trình lặp như vậy tuân thủ giao thức trình lặp. Logic của iterator rất đơn giản: chúng ta chỉ duyệt qua các ô của ma trận từ trên cùng bên trái xuống dưới cùng bên phải, bằng cách quét từng cột của mỗi hàng; chúng tôi đang làm điều đó bằng cách tận dụng hai chỉ mục, nextRow và nextCol.

Chúng ta có một lớp đối tượng Matrix được định nghĩa bên trong tệp matrix.js. Lớp này có hai thuộc tính: data là một mảng hai chiều chứa các số nguyên, và rows là số hàng của ma trận. Lớp này cài đặt giao diện Iterable bằng cách định nghĩa phương thức Symbol.iterator() trả về một đối tượng Iterator.

Trong lớp đối tượng Matrix, chúng ta có một lớp đối tượng MatrixIterator được định nghĩa bên trong. Lớp này có hai thuộc tính: x và y là vị trí hiện tại của con trỏ trong ma trận. Lớp này cũng có hai phương thức quan trọng là next() và hasNext(), được sử dụng để duyệt qua các phần tử của ma trận.

Phương thức next() trả về phần tử tiếp theo trong ma trận, và tăng giá trị của x và y. Phương thức hasNext() kiểm tra xem có phần tử nào còn lại trong ma trận hay không, bằng cách kiểm tra giá trị của x và y có lớn hơn hoặc bằng số hàng và số cột của ma trận hay không.

Khi sử dụng lớp Matrix và lặp lại các giá trị của nó, chúng ta có thể duyệt qua các phần tử của ma trận bằng cách sử dụng vòng lặp for...of. Trong mỗi lần lặp, chúng ta có thể truy cập phần tử hiện tại và in ra màn hình.

## 9.6 Iterator builtins

Tại đây, chúng ta có một mảng data chứa các chuỗi. Sử dụng phương thức entries() của mảng, chúng ta có thể tạo ra một đối tượng Iterator, biểu diễn các cặp giá trị key-value của mảng. Sau đó, chúng ta sử dụng vòng lặp for...of để duyệt qua các phần tử của Iterator và in ra màn hình.

### **Code:**

Matrix.js:

export class Matrix {

  constructor (*inMatrix*) {

*this*.data = *inMatrix*

  }

  get (*row*, *column*) {

    if (*row* >= *this*.data.length || *column* >= *this*.data[*row*].length) {

      throw **new** *Error*('Out of bounds')

    }

    return *this*.data[*row*][*column*]

  }

  set (*row*, *column*, *value*) {

    if (*row* >= *this*.data.length || *column* >= *this*.data[*row*].length) {

      throw **new** *Error*('Out of bounds')

    }

*this*.data[*row*][*column*] = *value*

  }

  [Symbol.iterator] () {

    let nextRow = 0

    let nextCol = 0

    return {

      next: () => {

        if (nextRow === *this*.data.length) {

          return { done: true }

        }

        const currVal = *this*.data[nextRow][nextCol]

        if (nextCol === *this*.data[nextRow].length - 1) {

          nextRow++

          nextCol = 0

        } else {

          nextCol++

        }

        return { value: currVal }

      }

    }

  }

}

*Index.js*

import { Matrix } from './matrix.js'

const matrix = **new** Matrix([

  ['Nguyễn Viết Quang', '25/03/2001', '04 CLC', 'Đesign Pattern'],

  ['Nguyễn Vinh Khang', '14/02/2002', '04 CLC', 'Đesign Pattern'],

  ['Nguyễn Van An', '16/07/2001', '04 CLC', 'Đesign Pattern'],

  ['Nguyễn Viết Đức', '06/10/2002', '04 CLC', 'Đesign Pattern']

])

console.log('for...of:')

for (const element of matrix) {

  console.log(element)

}

console.log('spread operator:')

const flattenedMatrix = [...matrix]

console.log(flattenedMatrix)

console.log('destructuring assignment:')

const [N1, N2, N3, N4, N5, N6] = matrix

console.log(N1, N2, N3, N4, N5, N6)

**Kết Quả:**

A screenshot of a computer program

Description automatically generated with medium confidence

### **Giải thích**

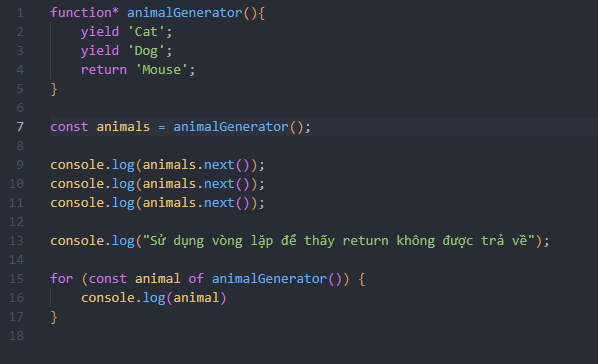
Phương thức entries() của mảng, chúng ta có thể tạo ra một đối tượng Iterator, biểu diễn các cặp giá trị key-value của mảng. Sau đó, chúng ta sử dụng vòng lặp for...of để duyệt qua các phần tử của Iterator và in ra màn hình.

Phương thức entries() trả về một đối tượng Iterator, biểu diễn các cặp giá trị key-value của mảng. Mỗi cặp giá trị này được biểu diễn dưới dạng một mảng con gồm hai phần tử, phần tử đầu tiên là chỉ số (index) của phần tử trong mảng, phần tử thứ hai là giá trị của phần tử.

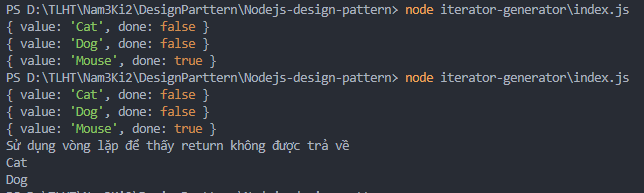
Trong vòng lặp for...of, chúng ta sử dụng cú pháp let [key, value] of data.entries() để khai báo biến key và value, rồi gán giá trị cho chúng từ các phần tử của Iterator. Sau đó, chúng ta có thể sử dụng biến key và value để xử lý các phần tử của mảng.

## 9.7 Simple Generator

### **Code:**



**Kết quả:**

****

### **Giải thích**

Với việc sử dụng phương pháp generator khi định nghĩa một Iterator pattern thì không cần phải triển khai các phương thức ‘next()’ và ‘hasNext()’ để kiểm soát quá trình lặp của iterator. Bằng cách sử dụng phương thức generator, việc tạo ra một iterator trở nên dễ dàng hơn và dễ đọc hơn bởi vì chỉ cần dùng từ khóa ‘yield’ để trả về các value và không cần phải triển khai các phương thức kiểm soát quá trình lặp. Từ khoá yield dừng hàm của chúng ta và cho biết trạng thái (state) của một generator tại một thời điểm xác định.  
 Bài toán trên chúng ta sử dụngfunction\* animalGenerator(): Đây là cú pháp để định nghĩa một generator function bằng từ khóa function\*. Trong generator function này, chúng ta sử dụng từ khóa yield để tạo ra các value, và sau đó sử dụng từ khóa return để kết thúc quá trình lặp với return value.

Ở lần gọi đầu tiên animals.net() được gọi, generator thực thi đến yield thứ nhất sẽ dừng lại và trả về giá trị ‘Cat’ cho người dùng và done ở trạng thái false nghĩa là generator tiếp tục thực thi.

Tiếp tục ở lần gọi thứ hai animals.net() được gọi đến yield thứ hai sẽ dừng lại và trả về giá trị ‘Dog’ cho người dùng và done ở trạng thái false.

Tuy nhiên ở lần gọi thứ ba animals.net() được gọi đến lệnh return và trả về giá trị ‘Mouse’ và done ở trạng thái true lúc này quá trình generator kết thúc. Và từ khóa ‘Mouse’ không được in ra khi chúng ta sử dụng vòng lặp for loop bởi vì sử dụng ‘return’ thay vì ‘yield’ nên ‘Mouse’ không là một phần tử của generator.

Ưu nhược điểm của việc sử dụng generator cho Iterator pattern:

**Ưu điểm:**

* Đơn giản và dễ đọc: Sử dụng generator cho Iterator pattern giúp code trở nên ngắn gọn, dễ hiểu và dễ quản lý. Cú pháp của generator rất rõ ràng và cho phép bạn mô tả việc tạo ra các giá trị tuần tự một cách trực quan.
* Hữu ích khi muốn xử lý một collection theo từng bước. Gia tăng hiệu quả bằng cách theo dõi trạng thái của collection, thay vì tất cả item trong collection. Các item trong collection được đánh giá từng cái tại một thời điểm, và phần đánh giá các item còn lại được hoãn lại sau đó.

**Nhược điểm:**

* Người sử dụng không thể gửi giá trị trực tiếp vào Iterator thông qua phươngthức next() như Iterator pattern truyền thống.

## 9.8 Kiểm soát generator iterator

### **Code:**

A screen shot of a computer program

Description automatically generated with low confidence

**Kết quả:  
A screen shot of a computer

Description automatically generated with medium confidence**

### **Giải thích**

Ở phần kiểm soát generator iterator này sẽ cho thấy cách để truyền một đối số vào phương thức next() khi mà trong phương thức generator của iterator không chấp nhận đối số truyền trực tiếp vào. Đối số này được truyền về làm giá trị trả về của yield theo như code nó sẽ truyền về const number = yield null. Đồng thời ở bài toán này cũng cho biết cách sử dụng và hoạt động của các phương thức throw() và return() của generator.

Code trên định nghĩa một generator function có tên là twoWayGenerator(). Mỗi lần gọi twoWayGenerator(), nó sẽ trả về một generator object.

Trong generator function này, chúng ta có một khối try-catch để bắt và xử lý ngoại lệ. Quá trình lặp của generator sẽ bắt đầu khi gọi phương thức next(). Khi đó, generator sẽ thực hiện các câu lệnh cho đến khi gặp lệnh yield. Generator đầu tiên được tạo bằng cách gọi twoWayGenerator().

Với twoWay1, sau khi gọi next(), generator sẽ thực hiện đến câu lệnh const number = yield null. Tại đây, generator tạm dừng và chờ đợi giá trị được truyền vào bằng cách gọi next(value). Ở đây value được truyền vào next() gián tiếp thông qua yield được tạo trong twoWayGenerator().

Với twoWay3, sử dụng phương thức throw() hoạt động tương tự như next(), nhưng nó cũng ném một ngoại lệ trong generator như thể ngoại lệ được ném tại điểm yield cuối cùng, và trả về đối tượng iterator.

Với twoWay4, sử dụng phương thức return() dừng thực thi generator và không có lệnh yield nào được thực thi và giá trị trả về là: {done: true, value: ‘ReturnValue’}. Có thể sử dụng return() để kết thúc sớm generator của iterator.

## 9.9 Sử dụng generator thay cho iterators

### **Code:**

matrix.js **A screen shot of a computer code

Description automatically generated with low confidence**index.js  
A screen shot of a computer program

Description automatically generated with low confidence

**Kết quả:**

**A screen shot of a computer

Description automatically generated with medium confidence**

### **Giải thích**

Ở bài toán này sử dụng generator để triển khai phương thức @@iterator của các đối tượng. Ở đây chúng ta sẽ biến đổi ma trận của iterator pattern không sử dụng generator sau thành file matrix.js ở trên:

Trong file matrix.js sử dụng hàm generator \* [Symbol.iterator]() để triển khai thay cho phương thức [Symbol.iterator]().

Trong hàm generator \* [Symbol.iterator](), chúng ta sử dụng từ khóa yield để trả về từng giá trị của ma trận một cách tuần tự. Mỗi lần gọi yield, giá trị của phần tử hiện tại sẽ được trả về và quá trình lặp sẽ tạm dừng. Khi chúng ta gọi phương thức next() trên đối tượng iterator, quá trình lặp sẽ tiếp tục từ vị trí đã tạm dừng trước đó.

Điều này giúp cho việc duyệt qua các phần tử của ma trận trở nên tự động và thuận tiện hơn. Chúng ta không cần phải quản lý các biến đếm và trạng thái bên ngoài để duyệt qua ma trận. Logic của trình lặp được dễ dàng hiểu và quản lý trong hàm generator.

Việc sử dụng hàm generator giúp mã nguồn trở nên đơn giản, dễ đọc và dễ bảo trì hơn. Nó cung cấp một cách tiếp cận linh hoạt để triển khai trình lặp cho các đối tượng có khả năng lặp lại.

## 9.10 Ứng dụng async iterators pattern

### **Code:**

***checkBullets.js***  
A screen shot of a computer screen

Description automatically generated with low confidence  
***index.js***  
A picture containing text, screenshot, software, multimedia software

Description automatically generated  
**Kết quả:**

A screen shot of a computer

Description automatically generated with medium confidence

### **Giải thích**

Pattern async iterator là một mô hình cho phép lặp qua một tập hợp dữ liệu hoặc một chuỗi các sự kiện theo cách không đồng bộ. Đây là một mở rộng của mô hình iterator.

Code trên định nghĩa một lớp CheckBullets có chức năng kiểm tra trạng thái của viên đạn trong một mảng súng. Lớp này triển khai một async iterator bằng cách sử dụng Symbol.asyncIterator.

• Trong hàm [Symbol.asyncIterator](), một iterator được tạo từ mảng súng và được lưu trữ trong biến bulletIterator. Phương thức next() của async iterator được triển khai để kiểm tra trạng thái của viên đạn và trả về giá trị tương ứng.

* Nếu avalue.done là true, nghĩa là đã lặp qua tất cả các viên đạn, phương thức next() trả về { done: true } để kết thúc lặp.
* Nếu viên đạn bullet lớn hơn 0, nghĩa là súng còn đạn, phương thức next() trả về { done: false, value: 'Súng còn đạn' }.
* Nếu viên đạn bullet bằng 0, nghĩa là súng hết đạn, phương thức next() trả về { done: false, value: 'Vui lòng nạp đạn' }.
* Nếu viên đạn bullet nhỏ hơn 0, nghĩa là súng bị hỏng, một ngoại lệ sẽ được ném ra và phương thức next() trả về { done: false, value: 'Súng hỏng: <message>' }, với <message> là thông báo lỗi.

Trong hàm main(), một đối tượng checkbullets của lớp CheckBullets được tạo với một mảng các giá trị đại diện cho trạng thái viên đạn. Bằng cách sử dụng vòng lặp for await...of, các giá trị trả về từ async iterator được gán cho biến bullet và được in ra bằng console.log().

## 9.11 Ứng dụng generator vào async iterators pattern

### **Code:**

***checkBullets.js***

A screen shot of a computer program

Description automatically generated with low confidence

**Kết quả:**

A screen shot of a computer

Description automatically generated with medium confidence

***index.js***  
A picture containing text, screenshot, software, multimedia software

Description automatically generated

### **Giải thích**

Phương thức generator có thể ứng dụng vào async iterators. Async generators cho phép sử dụng lệnh await trong cơ thể của chúng và giá trị trả về của phương thức next() là một promise giải quyết thành một đối tượng có các thuộc tính done và value. Điều này có nghĩa là async generator objects cũng là async iterators hợp lệ. Chúng cũng là async iterables hợp lệ, do đó chúng có thể được sử dụng trong vòng lặp for await...of. Điều này giúp cho code ngắn gọn và dễ đọc hơn.

## 9.12 Async iterators và Node.js stream

### **Code:**

A screen shot of a computer program

Description automatically generated with low confidence

### **Giải thích**

Bài toán trên trình bày cách sử dụng Async Iterators để xử lý dữ liệu từ luồng đầu vào tiêu chuẩn (stdin) trong Node.js

Trong đoạn code trên, chúng ta đang sử dụng thư viện 'split2' để tách dữ liệu từ luồng đầu vào tiêu chuẩn (stdin) thành các dòng riêng biệt. Đây là một ví dụ về cách sử dụng Async Iterators và Node.js Streams để xử lý dữ liệu từ luồng đầu vào.

Import thư viện 'split2', một transform stream được sử dụng để tách dữ liệu thành các dòng.

* Tạo một luồng (stream) bằng cách gắn 'split()' vào luồng đầu vào tiêu chuẩn (stdin) bằng cách sử dụng phương thức pipe(). Điều này cho phép dữ liệu từ stdin được truyền vào 'split()' để được tách thành các dòng riêng biệt với const stream = process.stdin.pipe(split()).
* Sử dụng vòng lặp for await...of để lặp qua từng dòng trong luồng. Dòng dữ liệu được lấy từ luồng thông qua biến line. Mỗi lần lặp, chúng ta in ra dòng dữ liệu đó bằng cách sử dụng console.log().

Ưu điểm:

* Hỗ trợ xử lý dữ liệu không đồng bộ và cho phép việc lấy dữ liệu theo yêu cầu.
* Cung cấp một cách trực quan và gọn gàng để lặp qua dữ liệu từ nguồn không đồng bộ.
* Cung cấp API tiện lợi để kết nối, sắp xếp và xử lý các luồng dữ liệu.

Nhược điểm:

* Khả năng quản lý lỗi: Khi sử dụng async iterators và Node.js streams, việc quản lý lỗi có thể trở nên phức tạp hơn so với việc sử dụng các mô hình lặp truyền thống
* Hiệu suất: Việc đọc dữ liệu theo từng phần nhỏ có thể gây ra độ trễ và tốn nhiều tài nguyên hơn so với việc đọc toàn bộ dữ liệu một lần.

## 9.13 MIDDLEWARE PATTERN

Middleware pattern là một mẫu thiết kế phần mềm được sử dụng để xử lý và chuyển tiếp yêu cầu giữa các thành phần trong một hệ thống phần mềm. Nó cung cấp một cách linh hoạt và mở rộng để áp dụng các chức năng xử lý trung gian vào quy trình xử lý yêu cầu.

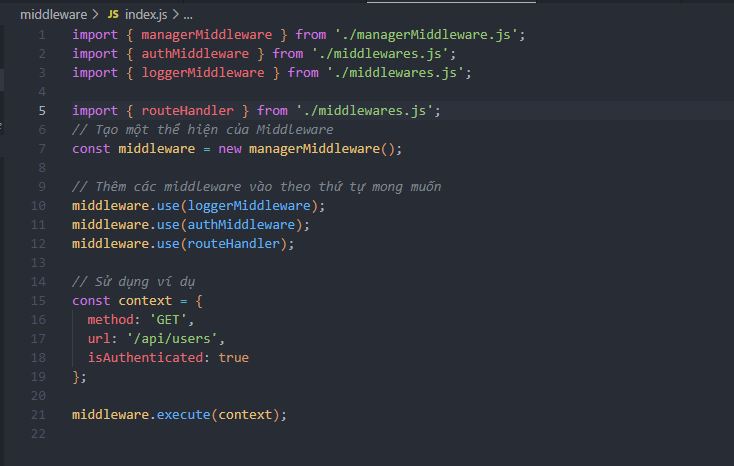
Middleware pattern cho phép bạn chèn các hàm xử lý (middleware) vào quá trình xử lý yêu cầu. Mỗi hàm middleware có thể thực hiện một số công việc xử lý nhất định hoặc thay đổi dữ liệu trước khi chuyển tiếp yêu cầu đến các middleware tiếp theo trong chuỗi. Nó cũng có thể thực hiện các hành động sau khi nhận được phản hồi từ các middleware trước đó.

Một điểm mạnh của Middleware pattern là khả năng xử lý yêu cầu một cách tuần tự và độc lập. Mỗi middleware chỉ quan tâm đến nhiệm vụ cụ thể mà nó thực hiện, không cần biết hoặc phụ thuộc vào các middleware khác trong chuỗi. Điều này cho phép bạn dễ dàng thêm, xóa hoặc thay đổi vị trí của các middleware mà không ảnh hưởng đến toàn bộ hệ thống.

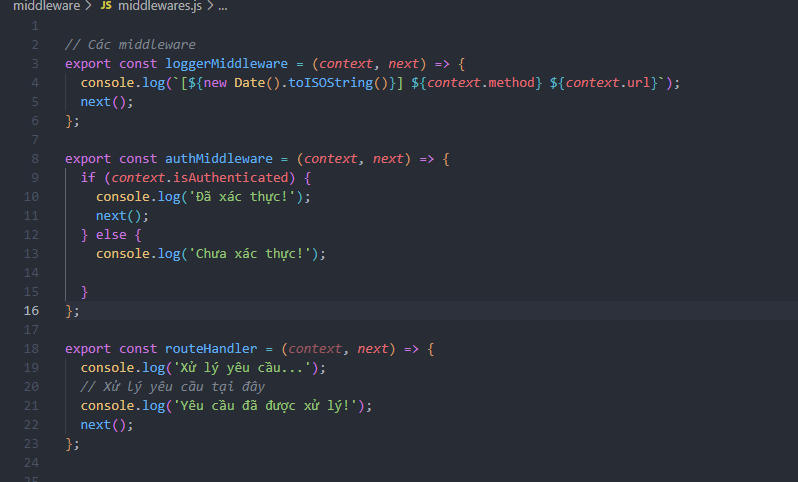
Middleware pattern được sử dụng rộng rãi trong các framework và thư viện phát triển phần mềm để xử lý các tác vụ như xác thực, ghi nhật ký, kiểm tra lỗi, nén dữ liệu và nhiều tác vụ xử lý khác. Nó cũng giúp tách riêng logic xử lý ứng dụng và tạo ra các thành phần độc lập có thể tái sử dụng trong nhiều vị trí khác nhau.

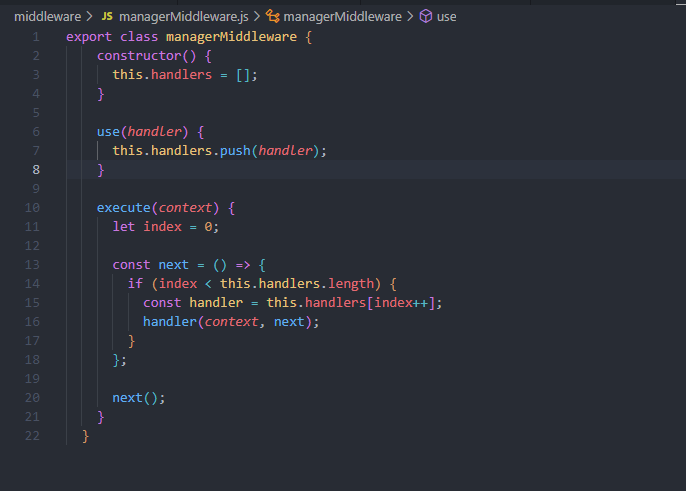
### **Code:**

**index.js:**

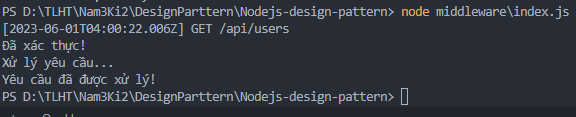
****

**Middlewares.js**

**  
managerMiddleware.js**

****

**Kết quả:**

****

### **Giải thích**

Code trên triển khai một ví dụ về việc sử dụng Middleware Pattern trong Node.js để xử lý các yêu cầu HTTP. Bài toán xây dựng một ứng dụng phản hồi ghi các log đơn giản với ba middleware: loggerMiddleware, authMiddleware, và routeHandler.

* loggerMiddleware: Middleware này được sử dụng để ghi log thông tin về mỗi yêu cầu đến server. Nó in ra thời gian, phương thức HTTP và URL của yêu cầu.
* authMiddleware: Middleware này kiểm tra xem người dùng đã xác thực hay chưa. Nếu người dùng đã xác thực, nó in ra thông báo "Đã xác thực!" và tiếp tục thực thi middleware tiếp theo. Nếu người dùng chưa xác thực, nó in ra thông báo "Chưa xác thực!" và không thực thi middleware tiếp theo.
* routeHandler: Middleware này đại diện cho xử lý yêu cầu. Trong ví dụ này, nó chỉ đơn giản in ra thông báo "Xử lý yêu cầu..." và "Yêu cầu đã được xử lý!".

Trong ví dụ trên, managerMiddleware là một lớp quản lý middleware. Nó cho phép thêm các middleware vào danh sách và thực thi chúng theo thứ tự.

Trong index.js, chúng ta tạo một thể hiện của managerMiddleware, thêm các middleware vào danh sách và thực thi chuỗi middleware khi có yêu cầu đến server. Mỗi middleware sẽ được thực thi theo thứ tự, và chúng ta có thể sử dụng các middleware khác nhau để xử lý các yêu cầu và thực hiện các tác vụ như ghi log hay xác thực trước khi xử lý yêu cầu chính.

**Ưu điểm:**

* Linh hoạt: Middleware pattern cho phép xây dựng các chuỗi xử lý linh hoạt và thay đổi quá trình xử lý bằng cách kết hợp các middleware theo ý muốn.
* Tái sử dụng và mở rộng: Các middleware có thể tái sử dụng và mở rộng trong nhiều vị trí khác nhau trong ứng dụng.
* Tách biệt trách nhiệm: Mỗi middleware chịu trách nhiệm cho một nhiệm vụ cụ thể, làm cho mã nguồn dễ đọc và dễ bảo trì.
* Kiểm soát luồng xử lý: Middleware pattern cho phép kiểm soát luồng xử lý trong quá trình xử lý yêu cầu.

**Nhược điểm:**

* Hiệu suất: Sử dụng quá nhiều middleware có thể ảnh hưởng đến hiệu suất ứng dụng.
* Xử lý lỗi: Xử lý lỗi và xác định nguồn gốc của lỗi có thể khó khăn trong một chuỗi middleware.
* Phụ thuộc: Sử dụng middleware pattern có thể tạo ra sự phụ thuộc mạnh mẽ giữa các middleware và ứng dụng, khiến việc thay đổi hoặc loại bỏ một middleware có thể yêu cầu sự điều chỉnh lớn trong mã nguồn.

## 9.14 COMMAND PATTERN

Mẫu Command là một mẫu thiết kế hành vi (behavioral design pattern) trong Node.js, nơi một đối tượng được sử dụng để đóng gói một hành động hoặc yêu cầu và các thông tin liên quan để thực thi hành động đó sau này. Thay vì gọi trực tiếp một phương thức hoặc hàm, mẫu Command cho phép chúng ta tạo ra một đối tượng đại diện cho ý định thực hiện hành động đó.

Trong mẫu Command, có một số thành phần chính:

* Command là đối tượng đóng gói thông tin cần thiết để gọi một phương thức hoặc hàm.
* Client là thành phần tạo ra command và cung cấp nó cho invoker.
* Invoker là thành phần chịu trách nhiệm thực thi command trên target.
* Target (hoặc receiver) là đối tượng của cuộc gọi. Nó có thể là một hàm đơn độc hoặc một phương thức của một đối tượng.

A diagram of a command

Description automatically generated with low confidence

**Ứng dụng của command pattern:**

* Một command có thể được lên lịch để thực hiện vào một thời điểm sau.
* Một command có thể được tuần tự hóa và gửi qua mạng. Đặc tính đơn giản này cho phép chúng ta phân phối công việc trên các máy từ xa, truyền command từ trình duyệt đến máy chủ, tạo hệ thống gọi thủ tục từ xa (RPC) và vân vân.
* Commands giúp dễ dàng lưu lại lịch sử của tất cả các hoạt động được thực hiện trên một hệ thống.
* Commands là một phần quan trọng của một số thuật toán đồng bộ hóa dữ liệu và giải quyết xung đột.
* Một command được lên lịch để thực hiện có thể được hủy bỏ nếu chưa được thực hiện. Nó cũng có thể bị hoàn ngược (khôi phục), đưa trạng thái của ứng dụng về điểm trước khi command được thực thi.
* Một số command có thể được nhóm lại với nhau. Điều này có thể được sử dụng để tạo giao dịch nguyên tử hoặc triển khai cơ chế mà tất cả các hoạt động trong nhóm được thực hiện cùng một lúc.
* Có thể thực hiện các loại biến đổi khác nhau trên một tập hợp các command, chẳng hạn như loại bỏ bản sao, ghép nối và phân chia hoặc áp dụng các thuật toán phức tạp hơn như biến đổi hoạt động (OT), là cơ sở cho hầu hết các phần mềm hợp tác thời gian thực hiện ngày nay, chẳng hạn như chỉnh sửa văn bản hợp tác.

### **Code:**

**index.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

**gun.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

**command.js**

**A screen shot of a computer program

Description automatically generated with medium confidence**

**createPostStatusCmd.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

**invoker.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

**server.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

**statusUpdateService.js**

**A screen shot of a computer program

Description automatically generated with low confidence**

**Kết quả:**

**A screen shot of a computer

Description automatically generated with medium confidence**

### **Giải thích**

File gun.js:

* Mô tả: Đây là file định nghĩa lớp Gun, đại diện cho một cái súng.
* Chức năng: Lớp Gun có các phương thức như shoot() để bắn súng và reload() để nạp lại đạn. Nó cũng có một phương thức hide() để ẩn súng.

File createPostStatusCmd.js:

* Mô tả: Đây là file định nghĩa hàm createPostStatusCmd để tạo một command để đăng trạng thái lên mạng xã hội.
* Chức năng: Hàm này tạo ra một đối tượng command chứa thông tin cần thiết để đăng trạng thái. Nó có các phương thức run() để thực thi command, undo() để hoàn tác command, và serialize() để trả về thông tin serialized của command.

File command.js:

* Mô tả: Đây là file định nghĩa các lớp command, bao gồm ShootCommand và ReloadCommand.
* Chức năng: Lớp ShootCommand đại diện cho một command bắn súng, trong khi ReloadCommand đại diện cho một command nạp đạn. Cả hai lớp command này có phương thức run() để thực hiện hành động tương ứng và serialize() để trả về thông tin serialized của command.

File statusUpdateService.js:

* Mô tả: Đây là file định nghĩa đối tượng statusUpdateService, đại diện cho dịch vụ cập nhật trạng thái.
* Chức năng: statusUpdateService có hai phương thức: postUpdate(status) để đăng trạng thái và destroyUpdate(id) để xóa trạng thái đã đăng.

File invoker.js:

* Mô tả: Đây là file định nghĩa lớp Invoker, đại diện cho đối tượng chịu trách nhiệm thực thi các command.
* Chức năng: Lớp Invoker có các phương thức như run(command) để thực thi một command, undo() để hoàn tác command cuối cùng, delay(command, delay) để thực thi một command sau một khoảng thời gian định trước và runRemotely(command) để thực thi một command từ xa.

File server.js:

* Mô tả: Đây là file định nghĩa một server HTTP đơn giản.
* Chức năng: Server lắng nghe yêu cầu và kiểm tra URL. Nếu URL không phải '/cmd', server sẽ trả về mã lỗi 400. Khi nhận được yêu cầu POST chứa một command serialized, server sẽ ghi log thông tin command và trả về mã 200.

File index.js:

* Mô tả: Đây là file chính của ứng dụng, nơi các command được tạo và thực thi.
* Chức năng: Trong file này, một số instance của lớp Gun và các command tương ứng (ShootCommand và ReloadCommand) được tạo ra. Các command được thực thi bằng cách sử dụng lớp Invoker. Cũng có một ví dụ về việc gọi command sau một khoảng thời gian và gửi command qua mạng.

**Ưu điểm:**

* Linh hoạt và tái sử dụng: Command Pattern cho phép đóng gói hành động vào các đối tượng command, giúp tăng tính linh hoạt và khả năng tái sử dụng trong hệ thống.
* Mở rộng dễ dàng: Thêm mới các command không ảnh hưởng đến các phần khác trong hệ thống, giúp dễ dàng mở rộng và thêm chức năng mới.
* Hoàn tác và hủy bỏ: Lưu trạng thái trước khi thực hiện hành động, cho phép hoàn tác và hủy bỏ các hành động đã thực thi.
* Quản lý lịch sử và ghi nhật ký: Theo dõi và ghi lại lịch sử các hành động đã thực thi, giúp quản lý và gỡ lỗi dễ dàng.
* Tương thích và truyền tải: Các command có thể truyền tải qua mạng hoặc lưu trữ dưới dạng serialized, giúp tích hợp và truyền tải dễ dàng trong các hệ thống khác nhau.

**Nhược điểm:**

* Tăng độ phức tạp: Sử dụng Command Pattern có thể làm tăng độ phức tạp của mã nguồn, đặc biệt khi có nhiều loại command và xử lý các trường hợp phức tạp.
* Tăng độ trễ: Việc tạo và truyền tải command có thể làm tăng độ trễ trong quá trình thực thi hành động.
* Độ phức tạp về quản lý: Yêu cầu quản lý và theo dõi các đối tượng command, lịch sử và trạng thái của chúng, đòi hỏi sự quản lý kỹ lưỡng và tăng độ phức tạp.