ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo Đồ án Design Pattern

Pattern: Bridge (structural) – Observer (patterns)

Môn học: Lập trình hướng đối tượng (OOP)

Sinh viên thực hiện:
Trịnh Anh Tài – 22127373
Phan Trung Nguyên – 22127300

Giáo viên hướng dẫn:
TS. Bùi Tiến Lên

Ngày 12 tháng 12 năm 2023

Mục lục

1 Design Pattern, nó là gì? Những khái niệm cơ bản **2**

2 Bridge Pattern – Dạng Thức Bắc Cầu **3**
2.1 Định nghĩa . . . 3
2.2 Tình huống thực tế và hướng giải quyết . . . 3
2.3 Kiến trúc . . . 5
2.3.1 Mã giả . . . 6
2.4 Khi nào nên dùng Pattern này? . . . 7
2.5 Ưu & nhược điểm . . . 7
2.6 Tự thiết kế và cài đặt . . . 8
2.6.1 Mã giả . . . 10

3 Observer Pattern – Mô Hình Mối Quan Hệ Một Và Nhiều **12**
3.1 Định nghĩa . . . 12
3.2 Tình huống thực tế và hướng giải quyết . . . 12
3.3 Kiến trúc . . . 13
3.3.1 Mã giả . . . 14
3.4 Khi nào nên dùng Pattern này? . . . 14
3.5 Ưu & nhược điểm . . . 15
3.6 Tự thiết kế và cài đặt . . . 15
3.6.1 Mã giả . . . 17

Tóm tắt nội dung

Báo cáo này phác thảo và tập trung phân tích chuyên sâu vào việc triển khai 2 *Mẫu Thiết Kế (Design Pattern)* thường thấy trong các dự án Lập trình hướng đối tượng là: **Bridge – Dạng Thức Bắc Cầu** và **Observer – Mô Hình Mối Quan Hệ Một Và Nhiều**. [1]

Ngôn ngữ triển khai: `C++`

Phân công		
Công việc	**Pattern**	**Người thực hiện**
Viết báo cáo	Bridge Pattern	Trịnh Anh Tài
	Observer Pattern	Phan Trung Nguyên
Viết code	Bridge Pattern	Trịnh Anh Tài
	Observer Pattern	Phan Trung Nguyên
Làm slide	Bridge Pattern	Nguyễn Quốc Anh
	Observer Pattern	Huỳnh Quang Huy
Quay video	Bridge Pattern	Nguyễn Quốc Anh
	Observer Pattern	Huỳnh Quang Huy

1 Design Pattern, nó là gì? Những khái niệm cơ bản

- **Mẫu Thiết Kế** (**Design Pattern**) là một giải pháp phổ thông cho các vấn đề thường xảy ra trong thiết kế phần mềm, được quy ước một cách có hệ thống bởi các coder "lão làng" đi trước, mục đích là để cho code của một tình huống nào đó dễ quản lí hơn, căn bản nó giống như một sự chia sẻ kinh nghiệm.
- Về cơ bản, **Mẫu Thiết Kế** như là một bản thiết kế được tạo sẵn mà bạn có thể tùy chỉnh để giải quyết vấn đề thường hay xuất hiện trong các quá trình viết code của mình.
- **Mẫu Thiết Kế** không phải là một đoạn mã cụ thể, mà là một khái niệm chung để giải quyết một vấn đề cụ thể, quyết định cuối cùng là ở bạn sau khi bạn tham khảo xong các chi tiết của **Mẫu** và thực hiện giải pháp phù hợp với nhu cầu dự án thực tế của riêng bạn.
- **Mẫu Thiết Kế** không phải là **Thuật Toán**, mặc dù chúng đều có điểm chung là "cung cấp giải pháp cho vấn đề cụ thể". Một **Thuật Toán** là một tập hợp các hành động rõ ràng có thể đạt được một số mục tiêu, còn một **Mẫu Thiết Kế** là một mô hình mô tả cấp cao hơn, trừu tượng hơn về một giải pháp. Code của cùng một **Mẫu** được áp dụng cho hai chương trình khác nhau có thể khác nhau.

Tóm lại, **Mẫu Thiết Kế** giống như một bản kế hoạch chi tiết: bạn có thể thấy kết quả và các tính năng của nó là gì, nhưng thứ tự chính xác của việc thực hiện là tùy thuộc vào bạn.

Phần bên dưới sẽ đi vào phân tích chuyên sâu hai **Mẫu Thiết Kế** là **Bridge – Dạng Thức Bắc Cầu** và **Observer – Mô Hình Mối Quan Hệ Một Và Nhiều**, lúc đó bạn sẽ hiểu rõ hơn như thế nào là một **Mẫu Thiết Kế** và cách mà ta áp dụng tư duy **Mẫu Thiết Kế** vào viết code.

[1] Các tên tiếng Việt được tham khảo tại đây: Dạng Thức Bắc Cầu và Mô Hình Mối Quan Hệ Một Và Nhiều.

2 Bridge Pattern – Dạng Thức Bắc Cầu

2.1 Định nghĩa

- Bridge Pattern là một trong những Pattern thuộc nhóm Structural Pattern.
- Ý tưởng của nó là tách tính trừu tượng (**Abstraction**) ra khỏi tính hiện thực (**Implementation**) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế các thuộc tính mà không làm ảnh hưởng đến những thuộc tính khác trong lớp ban đầu.

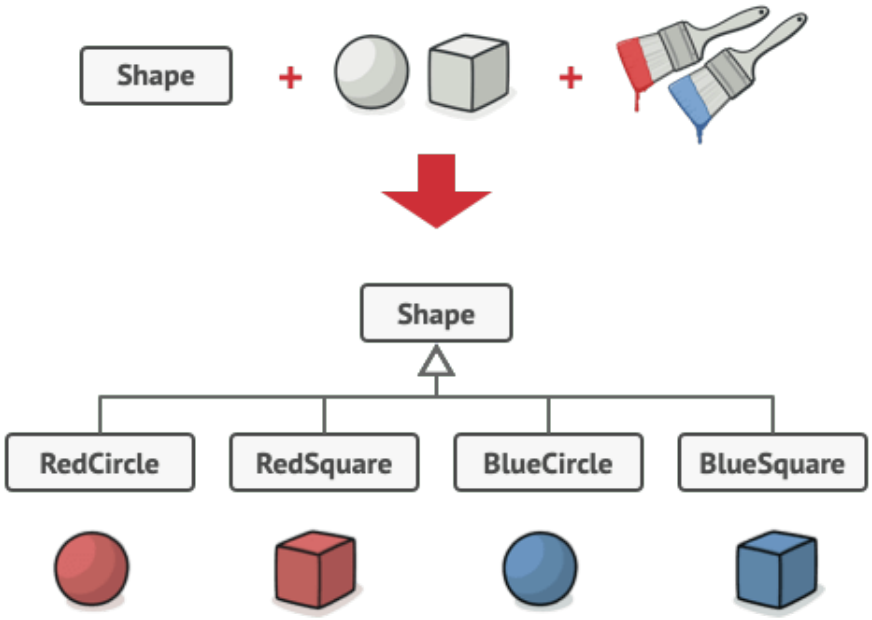
2.2 Tình huống thực tế và hướng giải quyết

Ta xét tình huống sau:

> **Tình huống:** *Giả sử bạn có class **Shape** và 2 subclass là **Hình Tròn** và **Hình vuông**. Do nhu cầu phát sinh, bạn muốn kết hợp thêm màu sắc vào là **Đỏ** và **Xanh**. Tuy nhiên thì bạn đã có hai subclass rồi, nên muốn thêm màu sắc thì bạn lại phải tạo mới 4 subclass là **Hình vuông Xanh**, **Hình vuông Đỏ**, **Hình Tròn Xanh** và **Hình Tròn Đỏ**. Nếu ta thêm một màu hoặc một hình nữa thì lại phải tạo thêm các lớp kế thừa nữa (ví dụ thêm subclass **Hình chữ nhật** thì ta lại phải tạo mới 2 subclass là **Hình chữ nhật Xanh** và **Hình chữ nhật Đỏ**).*

Vấn đề phát sinh: các lớp kế thừa sẽ tăng theo ***cấp số nhân*** theo quy luật trên khi ta càng thêm các đối tượng hoặc thuộc tính mới. Hệ quả hiển nhiên là khi hệ thống càng lớn, việc thêm vào các thuộc tính mới sẽ khiến kích thước tăng trưởng của các class sẽ càng trở nên tồi tệ hơn.

> *Hãy tưởng tượng ta cần 16 hình và 3 màu, tức tổng cộng phải triển khai 48 subclass! Như thế vẫn chưa đủ kinh khủng, giả sử ta cần bổ sung thêm 4 màu, như vậy phải tạo thêm 64 subclass cho 16 hình kia! Một việc làm vô nghĩa! Chắc chắn phải có một phương pháp quản lí các thuộc tính tốt hơn!*

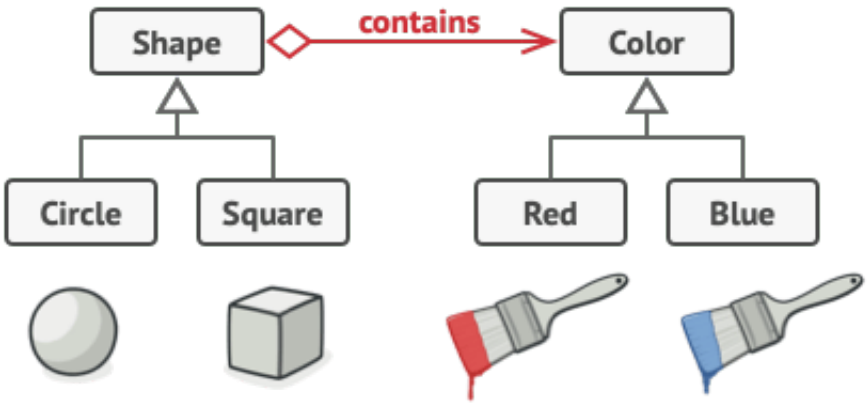


Hình 1: Vấn đề tăng trưởng lượng class theo cấp số nhân (Nguồn: refactoring.guru)

Nguyên nhân của tất cả những điều trên là do lối tư duy của chúng ta: Ta đang cố gắng mở rộng các lớp kế thừa theo hai chiều thuộc tính độc lập: theo ***Hình dáng (Shape)*** và theo ***Màu sắc (Colors)***. Đây là một tư duy rất phổ biến trong lập trình kế thừa lớp.

> **Giải pháp:** *Thay đổi quan hệ kế thừa thành quan hệ composition. Nghĩa là ta trích xuất một (hoặc nhiều) thuộc tính của vấn đề trên thành (những) class riêng biệt, xong cho class gốc tham chiếu đến (các) đối tượng thuộc (các) class mới ta vừa tạo đó – tức là ta đang tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của class gốc.*

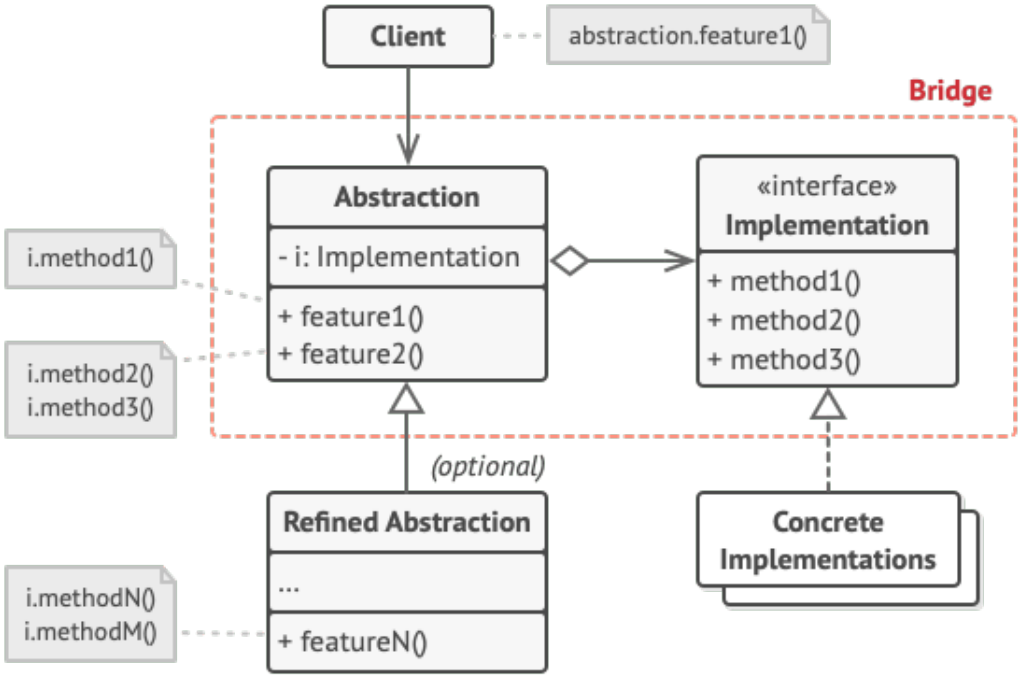
Áp dụng giải pháp cho lớp ***Shape***, ta thêm hẳn một thuộc tính mới là class ***Color***, ***Color*** thì có thể thêm các màu kế thừa như ***Xanh***, ***Đỏ***, ***Tím***, ***Vàng***,... tùy ý. Khi đó muốn ***Hình Chữ Nhật Đỏ*** ta chỉ cần ***Hình Chữ Nhật*** có thuộc tính ***Color*** là ***Đỏ*** thôi, tương tự với các hình khác mà không cần phải kế thừa nhiều.



Hình 2: Giải pháp (nguồn: refactoring.guru)

2.3 Kiến trúc

Ta xem xét hướng thiết kế kiến trúc chương trình tổng quát như sau:



Hình 3: Kiến trúc chương trình cơ bản (nguồn: refactoring.guru)

Trong đó:

1. **Client**: Khách hàng chỉ quan tâm đến việc làm việc với lớp trừu tượng (**Abstraction**). Việc của họ là liên kết đối tượng trừu tượng (**Abstraction**) với một trong các đối tượng hiện thực cụ thể (**Implementation**).

2. **Abstraction** (**Shape**): định nghĩa giao diện của lớp trừu tượng, nó cung cấp sự kiểm soát luân lí, các phương thức chung mang tính trừu tượng ở tầng cao, và tham chiếu vào các đối

tượng hiện thực cụ thể (**Implementation**) để chúng xử lí tiếp các công việc trên ở tầng thấp.

3. **Refined Abstraction** (**Circle**, **Square**): kế thừa **Abstraction**, nó cung cấp thêm các phương thức kiểm soát luân lí cho các dạng mở rộng cụ thể nâng cao.

4. **Implementation** (**Color**): định nghĩa giao diện cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của **Abstraction**.

5. **ConcreteImplementations** (**Red**, **Blue**): kế thừa **Implementation** và định nghĩa chi tiết các hàm thực thi.

2.3.1 Mã giả

```
interface Implementation {
    method1()
    method2()
    method3()
}

class Abstraction {
    i: Implementation

    feature1() {
        i.method1()
    }

    feature2() {
        i.method2()
        i.method3()
    }
}

class RefinedAbstraction extends Abstraction {

    featureN() {
        // Additional implementation for featureN
        // This can call methods from the Implementation interface as needed
    }
}

class ConcreteImplementationA implements Implementation {

   method1() {
       // Concrete implementation of method 1 for platform A
   }

   method2() {
       // Concrete implementation of method 2 for platform A
   }

   method3() {
       // Concrete implementation of method 3 for platform A
   }
}
```

```
42
43 class Client {
44
45     clientCode(abstraction: Abstraction) {
46         abstraction.feature1()
47         abstraction.feature2()
48
49         if (abstraction instanceof RefinedAbstraction)
50             abstraction.featureN()
51
52     }
53 }
```

2.4 Khi nào nên dùng Pattern này?

- Khi bạn muốn phân chia và sắp xếp một monolithic class (lớp nguyên khối) [2] có nhiều các chức năng, thuộc tính không liên quan với nhau. Từ đó bạn muốn tách ràng buộc giữa **Abstraction** và **Implementation** để có thể dễ dàng mở rộng độc lập nhau.

  > Bởi lẽ nếu lớp càng lớn thì càng khó tìm ra cách thức hoạt động và càng mất nhiều thời gian để thực hiện thay đổi. Những thay đổi được thực hiện đối với một trong các biến thể của thuộc tính có thể kéo theo các thay đổi trên toàn bộ lớp, từ đó có thể dẫn đến sai sót hoặc một số tác dụng phụ nghiêm trọng.

  > Bridge Pattern cho phép bạn chia lớp nguyên khối thành nhiều hệ thống phân cấp lớp. Sau này bạn có thể thay đổi các lớp trong mỗi hệ thống phân cấp một cách độc lập mà không ảnh hưởng gì đến các lớp khác. Cách tiếp cận này đơn giản hóa việc bảo trì code và giảm thiểu rủi ro làm "nhiễu" code hiện có.

  Lưu ý rằng cả **Abstraction** và **Implementation** nên được mở rộng bằng subsclass.

- Sử dụng ở những nơi mà những thay đổi được thực hiện trong **Implement** mà không ảnh hưởng đến phía **Client**.

  > Bridge Pattern cho phép bạn thay thế đối tượng hiện thực bên trong lớp trừu tượng, khiến cho việc này dễ dàng như việc gán một giá trị mới cho một trường.

2.5 Ưu & nhược điểm

Ưu điểm:

- Giảm sự phục thuộc giữa **abstraction** và **implementation** (loose coupling): tính kế thừa trong OOP thường gắn chặt **abstraction** và **implementation** lúc build chương trình. Bridge Pattern có thể được dùng để cắt đứt sự phụ thuộc này và cho phép chúng ta chọn **implementation** phù hợp lúc runtime.

[2] Một lớp chứa nhiều thuộc tính thành phần không được kết nối với nhau về mặt logic hoặc bị ghép nối một cách vô nghĩa, từ đó ta nên chia nó thành các đơn vị con logic để dễ kiểm soát hơn.

- Giảm số lượng những lớp con không cần thiết: một số trường hợp sử dụng tính inheritance sẽ tăng số lượng subclass rất nhiều.

> **Ví dụ:** *Ta cần triển khai chương trình view hình ảnh trên các hệ điều hành khác nhau, ta có 6 loại hình (JPG, PNG, GIF, BMP, JPEG, TIFF) và 3 hệ điều hành (Window, MacOS, Ubuntu). Sử dụng inheritance trong trường hợp này sẽ làm ta thiết kế 18 lớp: JpgWindow, PngWindow, GifWindow, ... Trong khi áp dụng Bridge sẽ giảm số lượng lớp xuống 9 lớp: 6 lớp ứng với từng implement của Image và 3 lớp ứng với từng hệ điều hành, mỗi hệ điều hành sẽ gồm một tham chiếu đến đối tượng Image cụ thể.*

- Code sẽ gọn gàng hơn và kích thước ứng dụng sẽ nhỏ hơn: do giảm được số class không cần thiết.
- Dễ bảo trì hơn: các **Abstraction** và **Implementation** của nó sẽ dễ dàng thay đổi lúc runtime cũng như khi cần thay đổi thêm bớt trong tương lai.
- Dễ dàng mở rộng về sau: thông thường các ứng dụng lớn thường yêu cầu chúng ta thêm module cho ứng dụng có sẵn nhưng không được sửa đổi framework/ứng dụng có sẵn vì các framework/ứng dụng đó có thể được công ty nâng cấp lên version mới. Bridge Pattern sẽ giúp chúng ta trong trường hợp này.
- Cho phép ẩn các chi tiết **Implement** từ **Client**: do **Abstraction** và **Implementation** hoàn toàn độc lập nên chúng ta có thể thay đổi một thành phần mà không ảnh hưởng đến phía **Client**. Ví dụ, các lớp của chương trình view ảnh sẽ độc lập với thuật toán vẽ ảnh trong các **Implementation**. Như vậy ta có thể update chương trình xem ảnh khi có một thuật toán vẽ ảnh mới mà không cần phải sửa đổi nhiều.

Nhược điểm:

- Có thể làm tăng độ phức tạp khi áp dụng cho một lớp có tính gắn kết cao

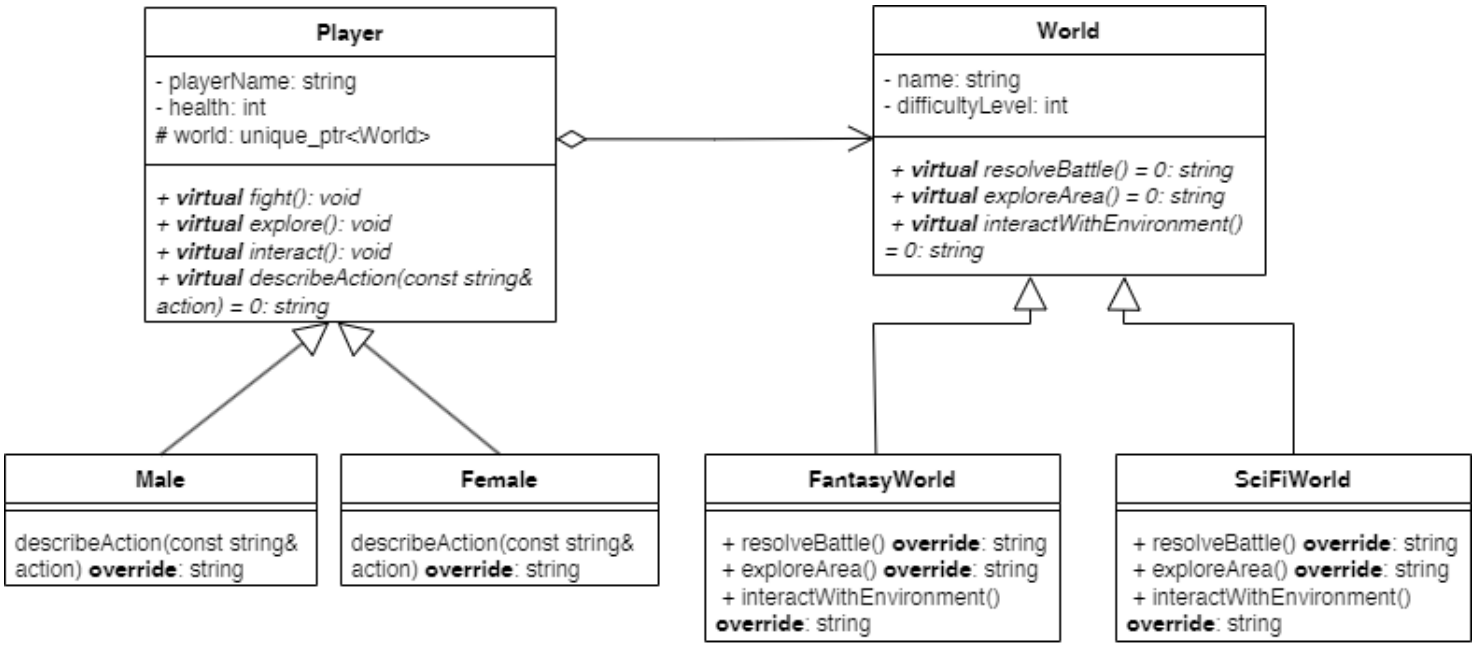
2.6 Tự thiết kế và cài đặt

Áp dụng các tư duy trên, ta thử đến với tình huống của riêng chúng ta và cài đặt nó:

> **Tình huống:** *Bạn là nhà phát triển game độc lập, ở dự án sắp tới, bạn sẽ triển khai một trò chơi với ý tưởng là nơi người chơi có thể chọn chơi ở các **Thế Giới** khác nhau, chẳng hạn như **Thế Giới Fantasy** hoặc **Thế Giới Khoa Học Viễn Tưởng**. Mỗi **Thế Giới** có thể những đặc điểm riêng về cách xử lý những thứ như trận chiến và tương tác với môi trường khác nhau. Tuy nhiên, bất kể thế giới nào, các **Hành Động** của người chơi – chẳng hạn như **Chiến Đấu**, **Khám Phá** và **Tương Tác** – vẫn giống nhau về concept.*

Hình 4: Poster game

Áp dụng giải pháp về Bridge Pattern, ta sẽ tạo ra một sự trừu tượng cho ***Hành Động*** của người chơi (***Abstraction***) và liên kết nó với các cách triển khai cụ thể chúng trong từng loại ***Thế Giới*** (***Implementation***) – tức mặc dù người chơi có thể thực hiện cùng một ***Hành Động*** (chẳng hạn như ***Chiến Đấu***), nhưng kết quả và tác động của những ***Hành Động*** này có thể khác nhau tùy theo ***Thế Giới*** mà họ đang ở mà không làm thay đổi cách người chơi triển khai ***Hành Động***.



Hình 5: Kiến trúc chương trình (không bao gồm các hàm **constructor()**, **destructor()**, **getter()** hay **setter()**)

2.6.1 Mã giả

```
class Player {
    playerName: string
    health: int
    world: unique_ptr<World>

    virtual fight(): void
    virtual explore(): void
    virtual interact(): void
    virtual describeAction(const string& action): string
}

class Male extends Player {
    describeAction(const string& action): string
}

class Female extends Player {
    describeAction(const string& action): string
}

class World {
    name: string
    difficultyLevel: int

    virtual resolveBattle(): string
    virtual exploreArea(): string
    virtual interactWithEnvironment(): string
}

```

```
class FantasyWorld extends World {
    resolveBattle(): string
    exploreArea(): string
    interactWithEnvironment(): string
}

class SciFiWorld extends World {
    resolveBattle(): string
    exploreArea(): string
    interactWithEnvironment(): string
}
```

Source code đầy đủ được cài đặt tại đây (Github)

3 Observer Pattern – Mô Hình Mối Quan Hệ Một Và Nhiều

3.1 Định nghĩa

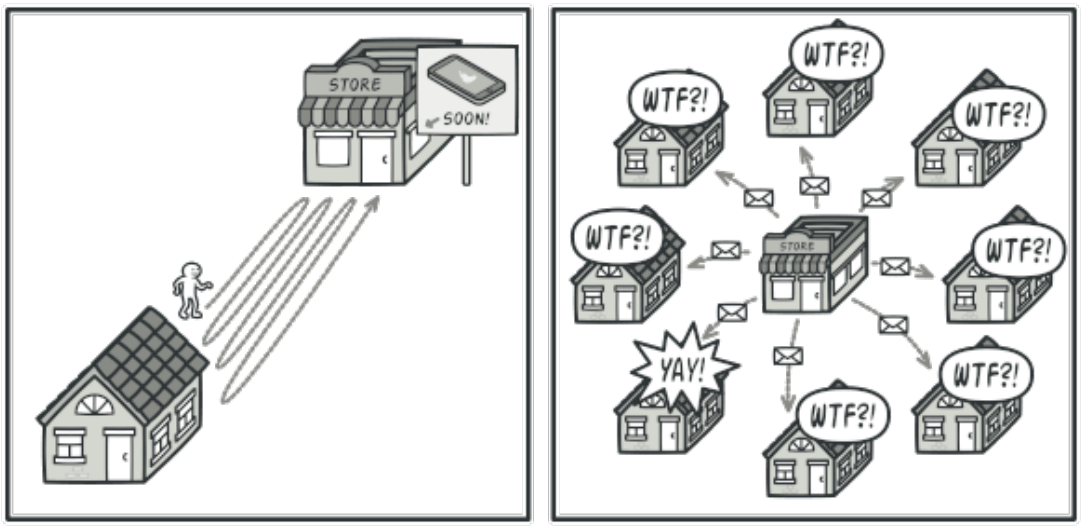
- Observer Pattern là một mẫu thiết kế trong lập trình hướng đối tượng, thuộc nhóm hành vi (Behavioral Pattern).
- Observer Pattern định nghĩa mối phụ thuộc một-nhiều giữa các đối tượng để khi một đối tượng (hay còn gọi là Subject) thay đổi trạng thái, tất cả các thành phần phụ thuộc của nó (Observer) sẽ được thông báo và cập nhật một cách tự động.

3.2 Tình huống thực tế và hướng giải quyết

Ta xét tình huống sau:

> **Tình huống:** *Tưởng tượng rằng bạn có hai loại đối tượng:* ***Khách hàng*** *và* ***Cửa hàng****.* ***Khách hàng*** *rất quan tâm đến một thương hiệu sản phẩm cụ thể (giả sử là một mẫu iPhone mới) chuẩn bị được bày bán tại* ***Cửa hàng****.* ***Khách hàng*** *có thể ghé thăm* ***Cửa hàng*** *mỗi ngày để kiểm tra đã có sản phẩm chưa. Nhưng trong khi sản phẩm vẫn chưa được tung ra, hầu hết những chuyến đi tới* ***Cửa hàng*** *này sẽ là vô nghĩa. Do đó,* ***Cửa hàng*** *nảy ra ý tưởng là sẽ thêm cơ chế gửi email mỗi khi có sản phẩn mới cho tất cả các* ***Khách hàng*** *với suy nghĩa là họ sẽ đỡ phải mất công đến* ***Cửa hàng*** *vô số lần.*

Vấn đề phát sinh: Điều này sẽ làm khó chịu những ***Khách hàng*** khác không quan tâm đến sản phẩm mới. Có vẻ như có một sự xung đột xảy ra ở đây. Hoặc là ***Khách hàng*** lãng phí thời gian kiểm tra xem sản phẩm đã có hàng hay chưa, hoặc là ***Cửa hàng*** lãng phí nguồn lực khi thông báo cho ***Khách hàng*** không muốn nhận thông báo.

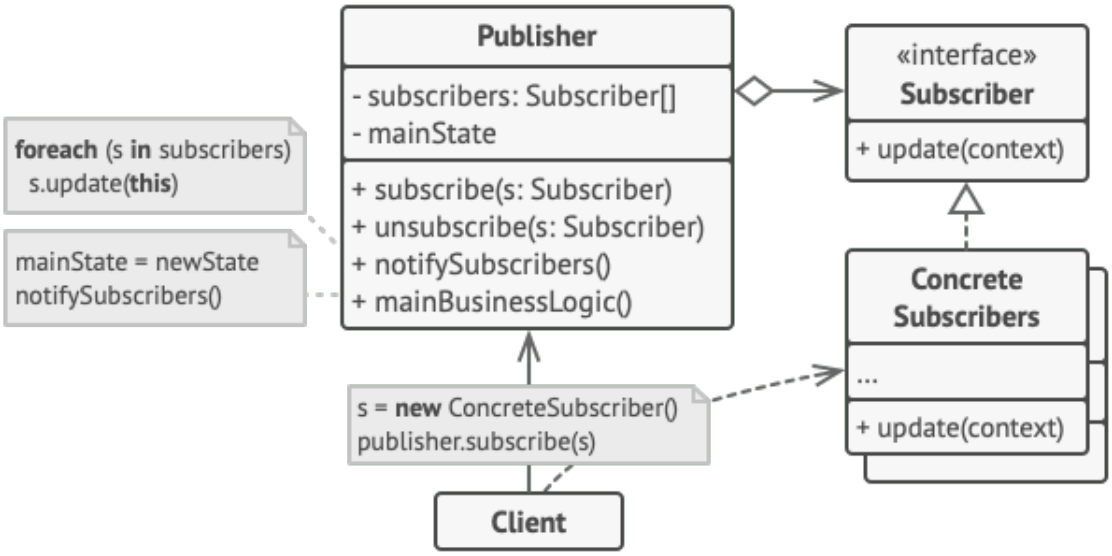


Hình 6: Hoặc là ***Khách hàng*** thăm ***Cửa hàng*** thường xuyên, hoặc là ***Cửa hàng*** gửi cả tấn spam email cho tất cả ***Khách hàng*** – trong đó có người không cần. (nguồn: refactoring.guru)

> **Giải pháp:** *Thêm cơ chế subscribe vào class **Cửa hàng** để các đối tượng riêng lẻ có thể đăng ký hoặc hủy đăng ký khỏi luồng sự kiện đến từ **Cửa hàng** đó. Mặt khác, các ứng dụng thực có thể có hàng chục class **Khách hàng** khác nhau quan tâm đến việc theo dõi các sự kiện của cùng một class **Cửa hàng**. Trong trường hợp này, chúng ta không nên gán trực tiếp **Cửa hàng** vào tất cả các class đó. Hãy xây dựng một giao diện chung và **Cửa hàng** chỉ giao tiếp với các **Khách hàng** thông qua giao diện đó.*

3.3 Kiến trúc

Ta xem xét hướng thiết kế kiến trúc chương trình tổng quát như sau:



Hình 7: Kiến trúc chương trình cơ bản (nguồn: refactoring.guru)

Trong đó:

1. **Publisher**: là lớp cần lắng nghe. Khi một sự kiện mới xảy ra, **Publisher** sẽ xem qua danh sách subscriber và gọi phương thức thông báo được khai báo trong **Subscriber interface** trên từng subscriber object. **Publisher** cũng cho phép người mới gia nhập nhận thông báo hoặc người hiện tại rời đi không nhận thông báo nữa.

2. **Subscriber interface**: là giao diện để **Publisher** thông báo mỗi khi có sự kiện. Trong hầu hết các trường hợp, nó chỉ bao gồm một phương thức duy nhất là **update**.

3. **Concrete Subscriber**: cài đặt cụ thể hành động để phản hồi khi nhận thông báo của **Publisher** đưa ra.

4. **Client**: tạo các đối tượng publisher và subscriber riêng biệt và sau đó đăng ký subscriber và lắng nghe các cập nhật của **Publisher**.

3.3.1 Mã giả

```
interface Subscriber {
    update(context: Publisher)
}

class Publisher {
    subscribers: Subscriber[]

    subscribe(subscriber: Subscriber) {
        // Add the subscriber to the subscribers list
    }

    unsubscribe(subscriber: Subscriber) {
        // Remove the subscriber from the subscribers list
    }

    notifySubscribers() {
        // For each subscriber in the subscribers list, call the update method
    }

    mainBusinessLogic() {
        // Execute some business logic
        // At some point, call notifySubscribers to issue an event
    }
}

class ConcreteSubscriber implements Subscriber {
    update(context: Publisher) {
        // Perform some action in response to the update
        // Use the context to fetch any required data
    }
}

class Client {
    clientCode() {
        // Create publisher and subscriber objects
        // Register subscribers for publisher updates
    }
}
```

3.4 Khi nào nên dùng Pattern này?

- Các thay đổi đối với trạng thái của một đối tượng có thể yêu cầu việc thay đổi các đối tượng khác và danh sách đối tượng trong thực tế không được biết trước hoặc có thể thay đổi động.
- Một số đối tượng trong ứng dụng của bạn phải quan sát những đối tượng khác, nhưng chỉ trong thời gian giới hạn hoặc trong các trường hợp cụ thể.

> **Ví dụ:** *Trên các ứng dụng mạng xã hội điển hình như là **Youtube** đều cho phép đăng ký hay hủy đăng ký các kênh nào đó tuy theo sở thích của người dùng. Do đó, mỗi kênh đều đóng vai trò là **Publisher**, cần quản lý danh sách những người đăng*

> *ký kênh của mình và ra thông báo khi có nội dung mới được tải lên kênh cho những người quan tâm (**Subscriber**).*

3.5 Ưu & nhược điểm

Ưu điểm:

- Đảm bảo nguyên tắc Open/Closed Principle (OCP): Cho phép thay đổi ***Publisher*** và ***Observer*** một cách độc lập. Nó cho phép thêm hoặc bớt một số lượng ***Observer*** mà không sửa đổi ***Publisher*** và ngược lại (nếu có giao diện ***Publisher***).
- Thiết lập mối quan hệ giữa các đối tượng trong thời gian chạy chương trình.
- Code sẽ gọn gàng, dễ bảo trì hơn.

Nhược điểm:

- Không kiểm soát được thứ tự các subscriber nhận thông báo.

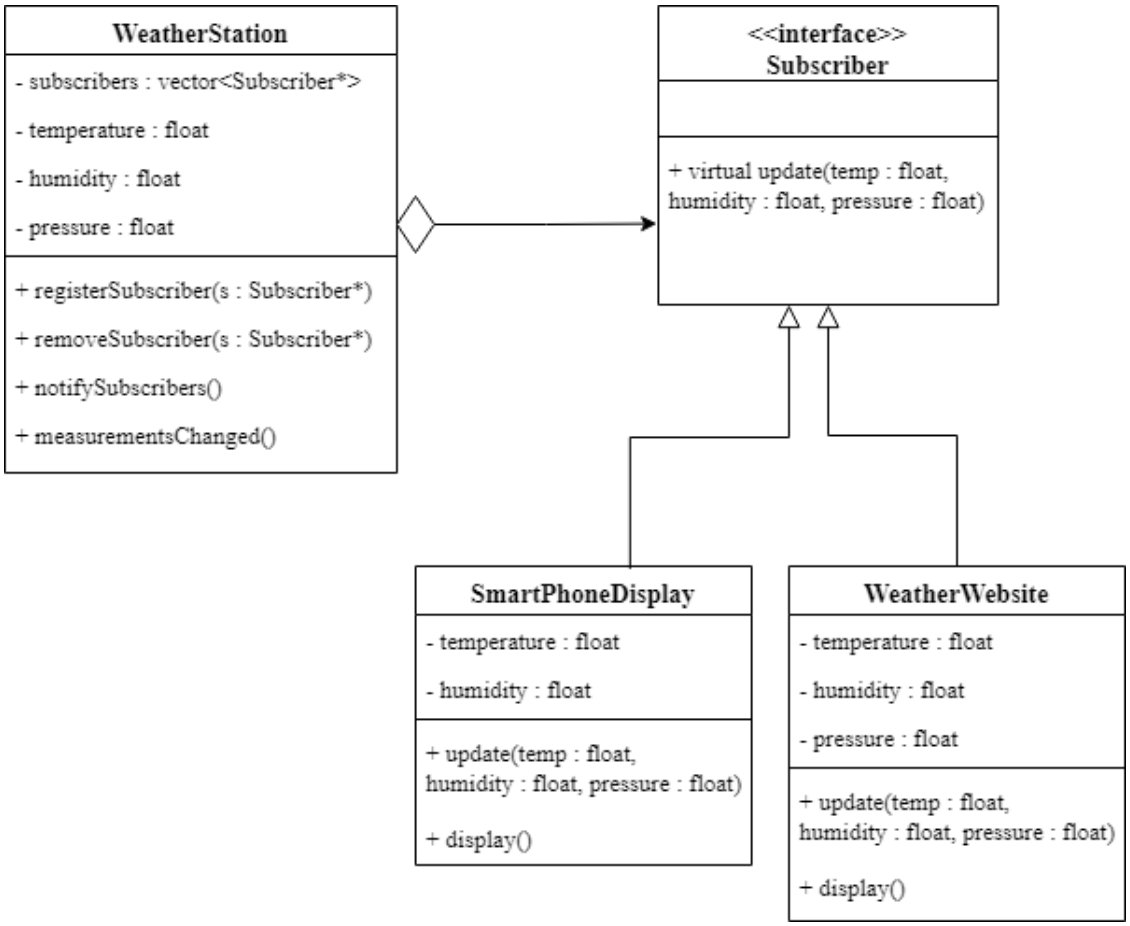
3.6 Tự thiết kế và cài đặt

Áp dụng các tư duy trên, ta thử đến với các tình huống của riêng chúng ta và cài đặt nó:

> **Tình huống:** *Bạn là developer làm việc tại **Trung tâm Dự báo khí tượng thuỷ văn quốc gia** và dự án lần này là xây dựng một hệ thống theo dõi và báo cáo thời tiết theo thời gian thực cho nhân dân thông qua các thiết bị điện tử. Trong đó phần lõi của hệ thống này – **WeatherStation** sẽ cập nhật dữ liệu thời tiết như nhiệt độ, độ ẩm, áp suất không khí, v.v... và sẽ có nhiều ứng dụng khác nhau muốn nhận thông tin cập nhật từ **WeatherStation**, chẳng hạn như **SmartPhoneDisplay**, **WeatherWebsite**, v.v... Mỗi khi **WeatherStation** có dữ liệu mới, tất cả các ứng dụng này đều sẽ được cập nhật cùng một lúc.*

Hình 8: Hình minh họa hệ sinh thái của hệ thống trên

Áp dụng giải pháp về Observer Pattern, ta sẽ tạo ra lớp ***Subscriber*** đóng vai trò là giao diện (interface) giao tiếp giữa ***WeatherStation*** (nó chính là lớp ***Publisher*** mà nãy giờ ta nói vừa nãy) và các ứng dụng – tức ***WeatherStation*** sẽ thông báo cho chúng mỗi khi có sự cập nhật các chỉ số thời tiết thông qua trung gian interface. Lớp ***WeatherStation*** này sẽ quản lý danh sách các ***Subscriber*** và các chỉ số thời tiết. Các lớp đại diện cho các ứng dụng như ***SmartPhoneDisplay***, ***WeatherWebsite***, v.v.. là các ***Concrete Subscriber*** kế thừa từ lớp ***Subscriber***.



Hình 9: Kiến trúc chương trình (không bao gồm các hàm **constructor()**, **destructor()**, **getter()** hay **setter()**)

3.6.1 Mã giả

```
interface Subscriber {
    virtual update(temp: float, humidity: float, pressure: float)
}

class WeatherStation {
    subscribers: vector<Subscriber*>
    temperature: float
    humidity: float
    pressure: float

    registerSubscriber(s: Subscriber*) {
        // Add the subscriber to the subscribers list
    }

    removeSubscriber(s: Subscriber*) {
        // Remove the subscriber from the subscribers list
    }

    notifySubscribers() {
        // For each subscriber in the subscribers list, call the update method
```

```
    }

    measurementsChanged() {
        // Execute some business logic
        // At some point, call notifySubscribers to issue an event
    }
}

class SmartPhoneDisplay implements Subscriber {
    temperature: float
    humidity: float

    update(temp: float, humidity: float, pressure: float) {
        // Perform some action in response to the update
        // Use the context to fetch any required data
    }

    display() {
        // Display the updated data
    }
}

class WeatherWebsite implements Subscriber {
    temperature: float
    humidity: float
    pressure: float

    update(temp: float, humidity: float, pressure: float) {
        // Perform some action in response to the update
        // Use the context to fetch any required data
    }

    display() {
        // Display the updated data
    }
}
```

Source code đầy đủ được cài đặt tại đây (Github).

Tài liệu tham khảo

1. https://refactoring.guru/design-patterns/bridge
2. https://vi.wikipedia.org/wiki/Dòểáng_thòệẩc_bòểẩccòểấu
3. https://viblo.asia/p/bridge-design-pattern-tro-thu-dac-luc-cua-developers-gDVK2oG2ZLj
4. https://refactoring.guru/design-patterns/observer
5. https://codelearn.io/sharing/observer-pattern-va-ung-dung