

# BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG**

**-------------------------------**

## ISO 9001:2015

**ĐỒ ÁN TỐT NGHIỆP**

## NGÀNH: CÔNG NGHỆ THÔNG TIN

**Sinh viên : Đoàn Văn Thọ**

**Giảng viên hướng dẫn: TS. Nguyễn Trịnh Đông**

**HẢI PHÒNG - 2019**

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG**

**-----------------------------------**

**ÁP DỤNG DESIGN PATTERNS TRONG PHÁT TRIỂN PHẦN MỀM**

**ĐỒ ÁN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY NGÀNH: CÔNG NGHỆ THÔNG TIN**

**Sinh viên : Đoàn Văn Thọ**

**Giảng viên hướng dẫn : TS. Nguyễn Trịnh Đông**

**HẢI PHÒNG - 2019**

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG**

***--------------------------------------***

**NHIỆM VỤ ĐỀ TÀI TỐT NGHIỆP**

## Sinh viên: Đoàn Văn Thọ Mã SV: 1512111005

**Lớp: CT1901C Ngành: Công nghệ thông tin**

**Tên đề tài: Áp dụng Design Pattens trong phát triển phần mềm**

# LỜI CẢM ƠN

## Lời đầu tiên em xin chân thành cảm ơn các thầy, cô trong khoa Công nghệ thông tin, trường Đại học Dân lập Hải Phòng đã tạo điều kiện thuận lợi cho em trong quá trình học tập tại trường cũng như trong thời gian thực hiện đồ án tốt nghiệp. Đặc biệt, em muốn gửi lời cảm ơn tới Tiến sỹ Nguyễn Trịnh Đông – giảng viên trực tiếp hướng dẫn, chỉ bảo giúp em khắc phục những khó khăn, thiếu sót để có thể hoàn thành các phần trong đồ án tốt nghiệp từ lý thuyết cho tới thực hành sử dụng công cụ.

## Mặc dù đã cố gắng với tất cả nỗ lực của bản thân để hoàn thiện đồ án, nhưng do thời gian có hạn, năng lực và kinh nghiệm còn hạn chế nên đồ án không thể tránh khỏi những thiếu sót. Kính mong nhận được sự đóng góp ý kiến từ phía thầy cô, bạn bè để em có thể nâng cao kiến thức của bản thân, hoàn thiện đồ án được tốt hơn.

## Em xin chân thành cảm ơn!

Hải Phòng, ngày ….. tháng ..… năm 2019.

Sinh viên thực hiện

Đoàn Văn Thọ

# MỤC LỤC

[LỜI CẢM ƠN](#_bookmark0)

[MỤC LỤC](#_bookmark1)

[MỞ ĐẦU](#_bookmark2)

[DANH MỤC HÌNH VẼ VÀ BẢNG BIỂU](#_bookmark3)

[DANH MỤC TỪ VIẾT TẮT](#_bookmark4)

[CHƯƠNG 1: GIỚI THIỆU VỀ DESIGN PATTERNS](#_bookmark5)

* 1. Vấn đề trong thiết kế phần mềm hướng đối tượng
  2. Lịch sử hình thành của Design Patterns
  3. [Design Patterns là gì ?](#_bookmark6)
  4. Đặc điểm chung
  5. [Tại sao phải sử dụng Design Patterns ?](#_bookmark12)
  6. Khi nào nên sử dụng Design Patetns ?
  7. [Ưu và nhược điểm khi sử dụng Design Patterns](#_bookmark29)
     1. [Ưu điểm](#_bookmark30)
     2. [Nhược điểm](#_bookmark32)
  8. Phân loại Design Patterns
     1. Nhóm Creational
     2. Nhóm Structural
     3. Nhóm Behavioral
  9. [Kết luận](#_bookmark36)

[CHƯƠNG 2: CÁC KỸ THUẬT CỦA DESIGN PATTERNS](#_bookmark37)

* 1. Nhóm Creational
  2. [Nhóm](#_bookmark42) Structural
  3. [Nhóm](#_bookmark48) Behavioral
  4. [Kết luận](#_bookmark52)

[CHƯƠNG 3:](#_bookmark53) XÂY DỰNG ỨNG DỤNG

* 1. Abc
  2. Xyz
  3. [Kết luận](#_bookmark92)

[KẾT LUẬN](#_bookmark93)

[DANH MỤC TÀI LIỆU THAM KHẢO](#_bookmark94)

# MỞ ĐẦU

## Trong xã hội hiện đại, công nghệ thông tin được coi là ngành quyền lực bậc nhất với hàng loạt ứng dụng trong mọi lĩnh vực của đời sống - từ sản xuất, kinh doanh đến giáo dục, y tế, văn hóa... Đặc biệt, ở thời kỳ Cách mạng 4.0 - mà tại Việt Nam cơ bản là ứng dụng như công nghệ tự động hóa, trao đổi dữ liệu … Trong công nghệ sản xuất, công nghệ thông tin càng khẳng định được tầm quan trọng của mình - vừa là nền tảng, vừa là động lực để bắt kịp đà phát triển của thế giới. Các hệ thống công nghệ thông minh chính là điều kiện để tối ưu hóa năng suất lao động, tiết giảm nhân lực lao động thủ công, tiết kiệm chi phí sản xuất.

## Vậy để hệ thống được dễ triển khai, kiểm soát chặt chẽ, thời gian hoàn thành nhanh chóng cũng như độ chính xác gần như tuyệt đối thì Design Patterns là một giải pháp được ưu tiên hàng đầu.

## Với mong muốn được tìm hiểu sâu về việc phát triển phần mềm nên em đã chọn đề tài “Áp dụng Design Patterns trong phát triển phần mềm.” Trong quá trình làm đồ án, do còn hạn chế về thời gian và kinh nghiệm thực tế, em mong nhận được những góp ý chân thành từ thầy cô và các bạn.

## Đề tài giới thiệu về lý thuyết Design Patterns, phân tích đánh giá các kỹ thuật và xây dựng ứng dụng thực nghiệm.

## Đồ án được tổ chức làm 5 phần như sau:

## Mở đầu: Trình bày rõ lý do chọn đề tài, mục tiêu nghiên cứu đồ án và bố cục của đồ án.

## Chương 1: Giới thiệu về Design Patterns. Chương này trình bày các khái niệm cơ bản, đặc điểm, phân loại, ưu và nhược điểm của Design Patterns.

## Chương 2: Design Patterns trong ngôn ngữ lập trình. Chương này trình bày chi tiết về các kỹ thuật được sử dụng trong các ngôn ngữ lập trình .

## Chương 3: Xây dựng ứng dụng

## Kết luận: Phần này đưa ra những kết quả đồ án đạt được, những thiếu sót chưa thực hiện được và hướng phát triển đề tài trong tương lai.

# DANH MỤC HÌNH VẼ VÀ BẢNG BIỂU

# DANH MỤC TỪ VIẾT TẮT

|  |  |  |  |
| --- | --- | --- | --- |
| STT | KÝ HIỆU | CỤM TỪ ĐẦY ĐỦ | Ý NGHĨA |
| 1 | SDLC | System Development Life Cycle | Vòng đời phát triển phần mềm |
| 2 | UML | Unified Modeling Language | Ngôn ngữ mô hình thống nhất |
| 3 | PloP | Pattern Language of Programming Design |  |
| 4 | DP | Design Patterns | Mẫu thiết kế |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 |  |  |  |
| 8 |  |  |  |
| 9 |  |  |  |
| 10 |  |  |  |

# CHƯƠNG 1: GIỚI THIỆU VỀ DESIGN PATTERNS

## Chương đầu tiên của đồ án đi sâu vào việc tìm hiểu các khái niệm cơ bản, đặc điểm, phân loại, ưu và nhược điểm của Design Patterns.

## Vấn đề trong thiết kế phần mềm hướng đối tượng.

## Người ta nói rằng, việc thiết kế một phần mềm hướng đối tượng là một công việc khó và việc thiết kế một phần mềm hướng đối tượng phục vụ cho mục đích dùng lại càng khó hơn. Vì thế, phải tìm ra những đối tượng phù hợp, đại diện cho một lớp các đối tượng. Sau đó thiết kế giao diện, tạo cây kế thừa cho chúng và thiết lập các mối quan hệ. Thiết kế phải đảm bảo là giải quyết được các vấn đề hiện tại, có thể tiến hành mở rộng trong tương lai mà tránh phải thiết kế lại phần mềm.

## Và một tiêu chí quan trọng là phải nhỏ gọn. Việc thiết kế một phần mềm hướng đối tượng phục vụ cho mục đích dùng lại là một công việc khó, phức tạp vì vậy không thể mong chờ thiết kế của mình sẽ là đúng và đảm bảo các tiêu chí trên ngay được. Thực tế là nó cần phải được thử nghiệm sau vài lần và sau đó sẽ được sửa chữa lại.

## 

## Đứng trước một vấn đề, một người phân tích thiết kế tốt có thể đưa ra nhiều phương án giải quyết, phải duyệt qua tất cả các phương ấn và rồi chọn ra cho mình một phương án tốt nhất. Phương án tốt nhất này sẽ được dùng đi dùng lại nhiều lần và dùng mỗi khi gặp vấn đề tương tự. Mà trong phân tích thiết kế phần mềm hướng đối tượng ta luôn gặp lại những vấn đề tương tự nhau.

## Lịch sử hình thành của Design Patterns.

## Ý tưởng dùng mẫu xuất phát từ ngành kiến trúc do Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King và Angel (1977) lần đầu tiên đưa ra ý tưởng dùng các mẫu chuẩn trong thiết kế xây dựng và truyền thông. Họ đã xác định và lập sưu liệu các mẫu có liên quan để có thể dùng để giải quyết các vấn đề thường xảy ra trong thiết kế các cao ốc. Mỗi mẫu này là một cách thiết kế, chúng đã được phát triển hàng trăm năm như là các giải pháp cho các vấn đề mà người ta làm trong lĩnh vực xây dựng thường gặp. Các giải pháp tốt nhất có được ngày hôm nay là qua một quá trình sàng lọc tự nhiên. Mặc dù ngành công nghệ phần mềm không có lịch sử phát triển lâu dài như nghành kiến trúc, xây dựng nhưng Công nghệ phần mềm là một nghành công nghiệp tiên tiến, tiếp thu tất cả những gì tốt đẹp nhất từ các ngành khác. Mẫu được xem là giải pháp tốt để giải quyết vấn đề xây dựng hệ thống phần mềm.

## Suốt những năm đầu 1990, thiết kế mẫu được thảo luận ở các hội thảo, sau đó người ta nỗ lực để đưa ra danh sách các mẫu và lập sưu liệu về chúng. Những người tham gia bị dồn vào việc cần thiết phải cung cấp một số kiểu cấu trúc ở một mức quan niệm cao hơn đối tượng và lớp để cấu trúc này có thể được dùng để tổ chức các lớp. Đây là kết quả của sự nhận thức đựơc rằng việc dùng các kỹ thuật hướng đối tượng độc lập sẽ không mang lại những cải tiến đáng kể đối với chất lượng cũng như hiệu quả của công việc phát triển phần mềm. Mẫu được xem là cách tổ chức việc phát triển hướng đối tượng, cách đóng gói các kinh nghiệm của những người đi trước và rất hiệu quả trong thực hành.

## Năm 1994 tại hội nghị PLoP(Pattern Language of Programming Design) đã được tổ chức. Cũng trong năm này quyển sách Design patterns: Elements of Reusable Object Oriented Software (Gamma, Johnson, Helm và Vhissdes, 1995) đã được xuất bản đúng vào thời điểm diễn ra hội nghị OOPSLA’94. Đây là một tài liệu còn phôi thai trong việc làm nỗi bật ảnh hưởng của mẫu đối với việc phát triển phần mềm, sự đóng góp của nó là xây dựng các mẫu thành các danh mục với định dạng chuẩn được dùng làm tài liệu cho mỗi mẫu và nổi tiếng với tên Gang of Four và các mẫu nó thường được gọi là các mẫu Gang of Four. Còn rất nhiều các cuốn sách khác xuất hiện trong 2 năm sau và các định dạng chuẩn khác được đưa ra.

## Năm 2000 Evitts có tổng kết về cách các mẫu xâm nhập vào thế giới phần mềm. Ông công nhận Kent Beck và Ward Cunningham là những người phát triển những mẫu đầu tiên với SmallTalk trong công việc của họ được báo cáo tại hội nghị OOPSLA’87. Có 5 mẫu mà Kent Beck và Ward Cunningham đã tìm ra trong việc kết hợp các người dùng của một hệ thống mà họ đang thiết kế. Năm mẫu này đều được áp dụng để thiết kế giao diện người dùng trong môi trường Windows.

## Design Patterns là gì ?

## Theo Christopher Alexander nói: “Mỗi một mẫu mô tả một vấn đề xảy ra lặp đi lặp lại trong môi trường và mô tả cái cốt lõi của giải pháp để cho vấn đề đó. Bằng cách nào đó bạn đã dùng nó cả triệu lần mà không làm giống nhau 2 lần”.

## Theo cuốn Software Engineering thì một mẫu thiết kế phần mềm là một giải pháp chung, có thể tái sử dụng cho một vấn đề thường xảy ra trong một bối cảnh nhất định trong thiết kế phần mềm . Nó không phải là một thiết kế đã hoàn thành có thể được chuyển đổi trực tiếp thành mã nguồn hoặc mã máy . Nó là một mô tả hoặc khuôn mẫu cho cách giải quyết vấn đề có thể được sử dụng trong nhiều tình huống khác nhau. Các mẫu thiết kế được chính thức hóa các thực tiễn tốt nhất mà lập trình viên có thể sử dụng để giải quyết các vấn đề phổ biến khi thiết kế một ứng dụng hoặc hệ thống.

## Kết luận vậy Design Pattern là gì ?

## Là tập các giải pháp cho vấn đề phổ biến trong thiết kế các hệ thống máy tính. Đây là tập các giải pháp đã được công nhận là tài liệu có giá trị. Những người phát triển có thể áp dụng giải pháp này để giải quyết các vấn đề tương tự.

## Giống như với các yêu cầu của thiết kế và phân tích hướng đối tượng thì việc sử dụng các mẫu cũng cần phải đạt được khả năng tái sử dụng các giải pháp chuẩn đối với vấn đề thường xuyên xảy ra.

## 

## Đặc điểm chung

## Mẫu được hiểu theo nghĩa tái sử dụng ý tưởng hơn là mã lệnh. Mẫu cho phép các nhà thiết kế có thể cùng ngồi lại với nhau và cùng giải quyết một vấn đề nào đó mà không phải mất nhiều thời gian tranh cãi. Trong rất nhiều trường hợp, dự án phần mềm thất bại là do các nhà phát triển không có được sự hiểu biết chung trong các vấn đề về kiến trúc phần mềm. Ngoài ra, mẫu cũng cung cấp những thuật ngữ và khái niệm chung trong thiết kế. Nói một cách đơn giản, khi đề cập đến một mẫu nào đấy, bất kỳ ai biết mẫu đó đều có thể nhanh chóng hình dung ra “bức tranh” của giải pháp. Và cuối cùng, nếu áp dụng mẫu hiệu quả thì việc bảo trì phần mềm cũng được tiến hành thuận lợi hơn, nắm bắt kiến trúc hệ thống nhanh hơn.

## Mẫu hỗ trợ tái sử dụng kiến trúc và mô hình thiết kế phần mềm theo quy mô lớn. Cần phân biệt Design Patterns với Framework. Framework hỗ trợ tái sử dụng mô hình thiết kế và mã nguồn ở mức chi tiết hơn. Trong khi đó, Design Patterns được vận dụng ở mức tổng quát hơn, giúp các nhà phát triển hình dung và ghi nhận các cấu trúc tĩnh và động cũng như quan hệ tương tác giữa các giải pháp trong quá trình thiết kế ứng dụng đối với một chuyên khu riêng biệt.

## Mẫu đa tương thích. Mẫu không phụ thuộc vào ngôn ngữ lập trình, công nghệ hoặc các nền tảng lớn như J2EE của Sun hay Microsoft .NET Framework.

## 

## Tại sao phải dùng Design Patterns ?

## Design Pattern giúp bạn tái sử dụng mã lệnh và dễ dàng mở rộng.

## Nó là tập hơn những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề. Vậy khi bạn gặp bất kỳ khó khăn gì, Design Patterns là kim chỉ nam giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh.

## Design Pattern cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.

## Dùng lại các Design Pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp, bảo trì về sau.

## Giúp cho các lập trình viên có thể hiểu code của người khác 1 cách nhanh chóng. Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà không mất quá nhiều thời gian.

## Tiềm năng ứng dụng của mẫu là rất lớn. Các thiết kế dựa trên mẫu được sử dụng khá nhiều ở các phần mềm mã nguồn mở, trong nền tảng J2EE hoặc .NET... Trong các dạng ứng dụng này, có thể dễ dàng nhận ra một số tên lớp chứa các tiền tố hoặc hậu tố như Factory, Proxy, Adapter...

## Khi nào nên sử dụng Design Patterns ?

## Khi bạn muốn giữ cho chương trình của mình thực sự đơn giản. Việc sử dụng các Design Pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ ra các cách giải quyết cho những vấn đề đã có lời giải.

## Chúng ta phải sử dụng các mẫu thiết kế trong giai đoạn phân tích và yêu cầu của vòng đời phát triển phần mềm - SDLC (System Development Life Cycle).

## Các mẫu thiết kế giảm bớt giai đoạn phân tích và yêu cầu của SDLC bằng cách cung cấp thông tin dựa trên kinh nghiệm thực hành trước đó.

## Ưu và nhược điểm của Design Patterns

## Ưu điểm

## Mẫu có thể tái sử dụng trong nhiều dự án.

## Mẫu cung cấp các giải pháp giúp xác định kiến ​​trúc hệ thống.

## Mẫu nắm bắt những kinh nghiệm kỹ thuật phần mềm.

## Mẫu cung cấp sự minh bạch cho việc thiết kế một ứng dụng.

## Mẫu là những giải pháp đã được chứng minh và chứng thực vì chúng được xây dựng dựa trên kiến ​​thức và kinh nghiệm của các nhà chuyên gia phát triển phần mềm.

## Các mẫu thiết kế không đảm bảo một giải pháp tuyệt đối cho một vấn đề. Chúng cung cấp sự rõ ràng cho kiến ​​trúc hệ thống và khả năng xây dựng một hệ thống tốt hơn.

## Nhược điểm

## There are no silver bullets in programming, and design patterns are not an exception

## Design patterns add a layer of abstraction to code which increases initial design and development effort

## Over-use of design patterns or forcing them to fit makes code harder to understand

## 

## Phân loại Design Patterns

## Năm 1994, bốn tác giả Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides đã cho xuất bản một cuốn sách với tiêu đề Design Patterns – Elements of Reusable Object-Oriented Software, đây là khởi nguồn của khái niệm Design Patterns trong lập trình phần mềm.

## Bốn tác giả trên được biết đến rộng rãi dưới tên Gang of Four. Theo quan điểm của bốn người, Design Patterns chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng.

## 

## Hệ thống các mẫu Design pattern hiện có 23 mẫu được định nghĩa trong cuốn “Design patterns Elements of Reusable Object Oriented Software” và được chia thành 3 nhóm:

## Creational Pattern (nhóm khởi tạo – 5 mẫu) gồm: Factory Method, Abstract Factory, Builder, Prototype, Singleton. Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.

## Structural Pattern (nhóm cấu trúc – 7 mẫu) gồm: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Những Design pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

## Behavioral Pattern (nhóm tương tác/ hành vi – 11 mẫu) gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau.

## 

Figure : nội dung

## Nhóm Creational (nhóm khởi tạo)

# 

## Singleton:

## Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.

## Tần suất sử dụng: cao trung bình.

## Abstract Factory:

## Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng.

## Tần suất sử dụng: cao.

## Factory Method:

## Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con.

## Tần suất sử dụng: cao.

## Builder:

## Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau.

## Tần suất sử dụng: trung bình thấp.

## Prototype:

## Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này.

## Tần suất sử dụng: trung bình.

## Nhóm Structural (nhóm cấu trúc)

## 

## Adapter:

## Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp.

## Tần suất sử dụng: cao trung bình.

## Bridge:

## Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt, mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau.

## Tần suất sử dụng: trung bình.

## Composite:

## Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây. Tất cả các đối tượng trong cấu trúc được thao tác theo một cách thuần nhất như nhau.

## Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau -> khả năng tổng quát hoá trong code của client -> dễ phát triển, nâng cấp, bảo trì.

## Tần suất sử dụng: cao trung bình.

## Decorator:

## Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).

## Tần suất sử dụng:trung bình.

## Facade:

## Cung cấp một interface thuần nhất cho một tập hợp các interface trong một “hệ thống con” (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn.

## Tần suất sử dụng: cao.

## Flyweight:

## Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng “cở nhỏ” (chẳng hạn paragraph, dòng, cột, ký tự…).

## Tần suất sử dụng: thấp.

## Proxy:

## Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy.

## Tần suất sử dụng: cao trung bình.

## Nhóm Behavioral (nhóm hành vi/ tương tác)

# 

## Chain of Responsibility:

## Khắc phục việc ghép cặp giữa bộ gởi và bộ nhận thông điệp. Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi nầy đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cứng giữa phần tử gởi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có có cơ hội xử lý request. Liên kết các đối tượng nhận request thành 1 dây chuyền rồi gửi request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.

## Tần suất sử dụng: trung bình thấp.

## Command:

## Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gởi đi như các đối tượng.Đóng gói request vào trong một Object, nhờ đó có thể nthông số hoá chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo…

## Tần suất sử dụng: cao trung bình.

## Interpreter:

## Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.

## Tần suất sử dụng: thấp.

## Iterator:

## Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, …) mà không phụ thuộc vào biểu diễn bên trong của các phần tử.

## Tần suất sử dụng: cao.

## Mediator:

## Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau.

## Tần suất sử dụng: trung bình thấp.

## Memento:

## Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu.

## Tần suất sử dụng: thấp.

## Observer:

## Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo.

## Tần suất sử dụng: cao.

## State:

## Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi, ta có cảm giác như class của đối tượng bị thay đổi.

## Tần suất sử dụng: trung bình.

## Strategy:

## Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán.Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng.

## Tần suất sử dụng: cao trung bình.

## Template method:

## Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa.

## Tần suất sử dụng: trung bình.

## Visitor:

## Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó.

## Tần suất sử dụng: thấp.

## Kết luận

## Design Patterns thể hiện tính kinh nghiệm của công việc lập trình, xây dựng và thiết kế phần mềm. Người hiểu và vận dụng được Design Patterns tích hợp trong hệ thống sẽ tiết kiệm được rất nhiều thời gian, công sức, dễ phát triển, mở rộng, bảo trì. Tuy nhiên không nên quá lạm dụng nó.

## Khi muốn tiếp cận đến một Design Patterns mới thì hãy tập trung chú ý vào ba phần này:

## Mẫu được sử dụng khi nào, vấn đề mà Design Patterns đó giải quyết là gì?

## Sơ đồ UML mô tả Design Pattern.

## Code minh họa, ứng dụng thực tiễn của mẫu là gì?

# CHƯƠNG 2: CÁC KỸ THUẬT CỦA DESIGN PATTERNS

# Nhóm Creational

# 1.1 Singleton Design Pattern

## Đôi khi, trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng cần tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi.

## Làm thế nào để hiện thực được một đối tượng như thế khi xây dựng mã nguồn? Chúng ta có thể nghĩ tới việc sử dụng một biến toàn cục (global variable). Tuy nhiên, việc sử dụng biến toàn cục nó phá vỡ quy tắc của OOP (encapsulation).

## Để giải bài toán trên, người ta hướng đến một giải pháp là sử dụng Singleton pattern.

# 1.1.1 Singleton Pattern là gì ?

## Singleton đảm bảo chỉ duy nhất một thể hiện (instance) được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

# 

## Sử dụng Singleton khi chúng ta muốn:

## Đảm bảo rằng chỉ có một instance của lớp.

## Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.

## Có thể quản lý số lượng thể hiện của một lớp trong giớn hạn chỉ định.

# 1.1.2 Thực thi Singleton Pattern như thế nào ?

## Có rất nhiều cách để implement Singleton Pattern. Nhưng dù cho việc implement bằng cách nào đi nữa cũng dựa vào nguyên tắc dưới đây cơ bản dưới đây:

## private constructor để hạn chế truy cập từ class bên ngoài.

## Đặt private static final variable đảm bảo biến chỉ được khởi tạo trong class.

## Có một method public static để return instance được khởi tạo ở trên.

# Những cách nào để thực thi Singleton Pattern

## 

# 1.1.4 Sử dụng Singleton Pattern khi nào ?

## Dưới đây là một số trường hợp sử dụng của Singleton Pattern thường gặp:

## Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …

## Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,…

## Đã được sử dụng trong một số class của core java như: java.lang.Runtime, java.awt.Desktop.

# Tổng kết

## Có rất nhiều cách implement cho singleton, thường sử dụng Billpughsingleton vì có hiệu suất cao, sử dụng Lazyinitializedsingleton cho những ứng dụng chỉ làm việc với ứng dụng single-thread và sử dụng Doublechecklockingsingleton khi làm việc với ứng dụng multi-thread. Tùy theo trường hợp cụ thể, hãy chọn cách implement phù hợp nhất.

# 1.2 Abstract Factory

# 1.2.1 Abstract Factory Pattern là gì ?

## Là phương pháp tạo ra một Super-Factory dùng để tạo ra các Factory khác. Hay còn được gọi là Factory của các Factory. Abstract Factory Pattern là một Pattern cấp cao hơn so với Factory Method Pattern.

## Trong Abstract Factory Pattern, một interface có nhiệm vụ tạo ra một Factory của các object có liên quan tới nhau mà không cần phải chỉ ra trực tiếp các class của object. Mỗi Factory được tạo ra có thể tạo ra các object bằng phương pháp giống như Factory pattern.

## Hãy tưởng tượng, Abstract factory như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

# 1.2.2 Cài đặt Abstract Factory Pattern như thế nào ?

## Một Abstract Factory Pattern bao gồm các thành phần cơ bản sau:

## AbstractFactory: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.

## ConcreteFactory: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.

## AbstractProduct: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.

## Product: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại AbstractProduct.

## Client: là đối tượng sử dụng AbstractFactory và các AbstractProduct.

# 1.2.3 Lợi ích của Abstract Factory Pattern là gì ?

## Các lợi ích của Factory Pattern cũng tương tự như Factory Method Pattern như: cung cấp hướng tiếp cận với Interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, …

## Giúp tránh được việc sử dụng điều kiện logic bên trong Factory Pattern. Khi một Factory Method lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình Abstract Factory để dễ quản lý hơn (cách phân chia có thể là gom nhóm các sub-class cùng loại vào một Factory).

## Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các sub-class khác.

## Dễ dàng xây dựng một hệ thống đóng gói (encapsulate): sử dụng được với nhiều nhóm đối tượng (factory) và tạo nhiều product khác nhau.

# 1.3 Factory Method

# 1.3.1 Factory Method Pattern là gì ?

## Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn.

## Factory Pattern đúng nghĩa là một nhà máy, và nhà máy này sẽ “sản xuất” các đối tượng theo yêu cầu của chúng ta.

## Trong Factory Pattern, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.

## Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

# 1.3.2 Cài đặt Factory Pattern như thế nào ?

## Một Factory Pattern bao gồm các thành phần cơ bản sau:

## Super Class: môt supper class trong Factory Pattern có thể là một interface, abstract class hay một class thông thường.

## Sub Classes: các sub class sẽ implement các phương thức của supper class theo nghiệp vụ riêng của nó.

## Factory Class: một class chịu tránh nhiệm khởi tạo các đối tượng sub class dựa theo tham số đầu vào. Lưu ý: lớp này là [Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/)hoặc cung cấp một public static method cho việc truy xuất và khởi tạo đối tượng. Factory class sử dụng if-else hoặc switch-case để xác định class con đầu ra.

# 1.3.3 Sử dụng Factory Pattern khi nào ?

## Factory Pattern được sử dụng khi:

## Chúng ta có một super class với nhiều class con và dựa trên đầu vào, chúng ta cần trả về một class con. Mô hình này giúp chúng ta đưa trách nhiệm của việc khởi tạo một lớp từ phía người dùng (client) sang lớp Factory.

## Chúng ta không biết sau này sẽ cần đến những lớp con nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.

# 1.3.4 Lợi ích của Factory Pattern là gì ?

## Lợi ích của Factory Pattern:

## Factory Pattern giúp giảm sự phụ thuộc giữa các module (loose coupling): cung cấp 1 hướng tiếp cận với Interface thay thì các implement. Giúp chuơng trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.

## Mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra sub class và implement thêm vào factory method.

## Khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đấy. Người dùng không biết logic thực sực được khởi tạo bên dưới phương thức factory.

## Dễ dạng quản lý life cycle của các Object được tạo bởi Factory Pattern.

## Thống nhất về naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

## Bạn có thể thấy Factory Pattern được áp dụng trong:

## JDK: java.util.Calendar, ResourceBundle, NumberFormat, …

## BeanFactory trong Spring Framework.

## SessionFactory trong Hibernate Framework.

## …

# 1.4 Builder

# 1.4.1 Builder Pattern là gì ?

## Là mẫu thiết kế đối tượng được tạo ra để xây dựng một đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng đôc lập với các đối tượng khác.

## Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính.

## Có ba vấn đề chính với  Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính:

## Quá nhiều tham số phải truyền vào từ phía client tới Factory Class.

## Một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.

## Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.

# 1.4.2 Cài đặt Builder Pattern như thế nào?

# 

## Một builder gồm các thành phần cơ bản sau:

## Product : đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.

## Builder : là abstract class hoặc interface khai báo phương thức tạo đối tượng.

## ConcreteBuilder : kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.

## Director/ Client: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

# 1.4.3 Lợi ích của Builder Pattern là gì ?

## Hỗ trợ, loại bớt việc phải viết nhiều constructor.

## Code dễ đọc, dễ bảo trì hơn khi số lượng thuộc tính (propery) bắt buộc để tạo một object từ 4 hoặc 5 propery.

## Giảm bớt số lượng constructor, không cần truyền giá trị null cho các tham số không sử dụng.

## Ít bị lỗi do việc gán sai tham số khi mà có nhiều tham số trong constructor: bởi vì người dùng đã biết được chính xác giá trị gì khi gọi phương thức tương ứng.

## Đối tượng được xây dựng an toàn hơn: bởi vì nó đã được tạo hoàn chỉnh trước khi sử dụng.

## Cung cấp cho bạn kiểm soát tốt hơn quá trình xây dựng: chúng ta có thể thêm xử lý kiểm tra ràng buộc trước khi đối tượng được trả về người dùng.

## Có thể tạo đối tượng immutable.

## Một số ví dụ sử dụng Builder Pattern trong JDK:

## java.lang.[StringBuilder](https://gpcoder.com/2149-huong-dan-su-dung-java-string-stringbuffer-va-stringbuilder).append().

## java.lang.[StringBuffer](https://gpcoder.com/2149-huong-dan-su-dung-java-string-stringbuffer-va-stringbuilder).append().

# 1.4.4 Nhược điểm của Builder Patten là gì ?

## Builder Pattern có nhược điểm là duplicate code khá nhiều: do cần phải copy tất cả các thuộc tính từ class Product sang class Builder.

## Tăng độ phức tạp của code (tổng thể) do số lượng class tăng lên.

## 1.4.5 Sử dụng Builder Pattern khi nào ?

## Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc (requried), một số không bắt buộc (optional).

## Khi có quá nhiều hàm constructor, bạn nên nghĩ đến Builder.

## Muốn tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng.

## Muốn kiểm soát quá trình xây dựng.

## Khi người dùng (client) mong đợi nhiều cách khác nhau cho đối tượng được xây dựng.

## 1.5 Prototype

## 1.5.1 Prototype Pattern là gì ?

## Nó có nhiệm vụ khởi tạo một đối tượng bằng cách clone một đối tượng đã tồn tại thay vì khởi tạo với từ khoá new. Đối tượng mới là một bản sao có thể giống 100% với đối tượng gốc, chúng ta có thể thay đổi dữ liệu của nó mà không ảnh hưởng đến đối tượng gốc.

## Prototype Pattern được dùng khi việc tạo một object tốn nhiều chi phí và thời gian trong khi bạn đã có một object tương tự tồn tại.

## Trong Java cung cấp mẫu prototype pattern này bằng việc implement interface [Cloneable](https://gpcoder.com/2361-object-cloning-trong-java/) và sử dụng method clone() để tạo object có đầy đủ thuộc tính của đối tượng ban đầu

## 1.5.2 Cài đặt Prototype Pattern như thế nào ?

## 

## Một Prototype Pattern gồm các thành phần cơ bản sau:

## Prototype : khai báo một class, interface hoặc abtract class cho việc clone chính nó.

## ConcretePrototype class : các lớp này thực thi interface (hoặc kế thừa từ lớp abstract) được cung cấp bởi Prototype để copy (nhân bản) chính bản thân nó. Các lớp này chính là thể hiện cụ thể phương thức clone(). Lớp này có thể không cần thiết nếu: Prototype là một class và nó đã implement việc clone chính nó.

## Client class : tạo mới object bằng cách gọi Prototype thực hiện clone chính nó.

# 1.5.3 Lợi ích của Prototype Pattern là gì ?

## Cải thiện hiệu suất: giảm chi phí để tạo ra một đối tượng mới theo chuẩn, điều này sẽ làm tăng hiệu suất so với việc sử dụng từ khóa new để tạo đối tượng mới.

## Giảm độ phức tạp cho việc khởi tạo đối tượng: do mỗi lớp chỉ implement cách clone của chính nó.

## Giảm việc phân lớp, tránh việc tạo nhiều lớp con cho việc khởi tạo đối tượng như của [Abstract Factory Pattern](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/).

## Khởi tạo object mới bằng cách thay đổi một vài thuộc tính của object (các object có ít điểm khác biệt nhau): Một hệ thống linh động sẽ để cho chúng ta tự định nghĩa một hành động nào đó thông qua sự kết hợp với một object (nghĩa là một phương thức của một class) hơn là định nghĩa một class mới.

## Khởi tạo object mới bằng cách thay đổi cấu trúc: Rất nhiều ứng dụng xây dựng hệ thống từ nhiều phần và các phần con. Các phần con lại khởi tạo từ nhiều phần con khác (chia nhỏ bài toán). Prototype pattern cũng hỗ trợ điều này. Nghĩa là các phần đó có thể được khởi tạo từ việc copy một nguyên mẫu từ một “cấu trúc” khác. Miễn là các phần kết hợp đều thể hiện clone() và được sử dụng với cấu trúc khác nhau làm nguyên mẫu. Xem thêm về [Object cloning trong java](https://gpcoder.com/2361-object-cloning-trong-java/) bạn sẽ thấy rõ điều này.

## 1.5.4 Sử dụng Prototype khi nào ?

## Có một object và cần phải tạo 1 object mới khác dựa trên object bạn đầu mà không thể sử dụng toán tử new hay các hàm constructor để khởi tạo. Lý do đơn giản là ở đây chúng ta ko hề được biết thông tin nội tại của object đó hoặc object đó đã có thể bị che dấu đi nhiều thông tin khác mà chỉ cho ta một thông tin rất giới hạn không đủ để hiểu được. Do vậy ta ko thể dùng toán tử new để khởi tạo nó được. Giải pháp: để cho chính object mẫu tự xác định thông tin và dữ liệu sao chép.

## Khởi tạo đối tượng lúc run-time: chúng ta có thể xác định đối tượng cụ thể sẽ được khởi tạo lúc runtime nếu class được implement / extend từ một Prototype.

## Cấu hình một ứng dụng với dynamic class.

## Muốn truyền đối tượng vào một hàm nào đó để xử lý, thay vì truyền đối tượng gốc có thể ảnh hưởng dữ liệu thì ta có thể truyền đối tượng sao chép.

## Chi phí của việc tạo mới đối tượng (bằng cách sử dụng toán tử new) là lớn.

## Ẩn độ phức tạp của việc khởi tạo đối tượng từ phía Client.

# Nhóm Structural

# 2.1 Adapter

# 2.1.1 Adapter Pattern là gì ?

## Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

## Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.

## 2.1.2 Cài đặt Adapter Pattern như thế nào ?

## Một Adapter Pattern bao gồm các thành phần cơ bản sau:

## Adaptee: định nghĩa interface không tương thích, cần được tích hợp vào.

## Adapter: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.

## Target: một interface chứa các chức năng được sử dụng bởi Client (domain specific).

## Client: lớp sử dụng các đối tượng có interface Target.

## Có hai cách để thực hiện Adapter Pattern dựa theo cách cài đặt (implement) của chúng:

## Object Adapter – Composition (Tổng hợp): trong mô hình này, một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có interface không tương thích.

## 

## Class Adapter – Inheritance (Kế thừa) : trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của interface người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có interface không tương thích.

## 

## So sánh Class Adapter với Object Adapter:

## Sự khác biệt chính là Class Adapter sử dụng Inheritance (kế thừa) để kết nối Adapter và Adaptee trong khi Object Adapter sử dụng Composition (tổng hợp) để kết nối Adapter và Adaptee.

## Trong cách tiếp cận Class Adapter, nếu một Adaptee là một class và không phải là một interface thì Adapter sẽ là một lớp con của Adaptee. Do đó, nó sẽ không phục vụ tất cả các lớp con khác theo cùng một cách vì Adapter là một lớp phụ cụ thể của Adaptee.

## Tại sao Object Adapter lại tốt hơn?

## Nó sử dụng Composition để giữ một thể hiện của Adaptee, cho phép một Adapter hoạt động với nhiều Adaptee nếu cần thiết.

# 2.1.3 Lợi ích của việc dùng Adapter Pattern là gì ?

## Việc sử dụng Adapter Pattern đem lại các lợi ích sau:

## Cho phép nhiều đối tượng có interface giao tiếp khác nhau có thể tương tác và giao tiếp với nhau.

## Tăng khả năng sử dụng lại thư viện với interface không thay đổi do không có mã nguồn.

## Bên cạnh những lợi ích trên, nó cũng nó một số khuyết điểm nhỏ sau:

## Tất cả các yêu cầu được chuyển tiếp, do đó làm tăng thêm một ít chi phí.

## Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter (chain) trước khi đến được yêu cầu thực sự.

# 2.1.4 Sử dụng Adapter Pattern khi nào ?

## Có thể dùng Adapter Pattern trong những trường hợp sau:

## Adapter Pattern giúp nhiều lớp có thể làm việc với nhau dễ dàng mà bình thường không thể. Một trường hợp thường gặp phải và có thể áp dụng Adapter Pattern là khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.

## Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.

## Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.

## Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.

## Khi cần đảm bảo nguyên tắc [Open/ Close](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Open-Closed_principle_OCP) trong một ứng dụng.

## Một vài class sử dụng Adapter Pattern:

## java.util.Arrays#asList()

## java.io.InputStreamReader(InputStream) (returns a Reader)

## java.io.OutputStreamWriter(OutputStream) (returns a Writer)

## javax.xml.bind.annotation.adapters.XmlAdapter#marshal() và #unmarshal()

# 2.2 Bridge

# 2.2.1 Bridge Pattern là gì ?

## 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 mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu.

## Điều đó có nghĩa là, ban đầu chúng ta thiết kế một class với rất nhiều xử lý, bây giờ chúng ta không muốn để những xử lý đó trong class đó nữa. Vì thế, chúng ta sẽ tạo ra một class khác và move các xử lý đó qua class mới. Khi đó, trong lớp cũ sẽ giữ một đối tượng thuộc về lớp mới, và đối tượng này sẽ chịu trách nhiệm xử lý thay cho lớp ban đầu.

## Bridge Pattern khá giống với mẫu [Adapter Pattern](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/) ở chỗ là sẽ nhờ vào một lớp khác để thực hiện một số xử lý nào đó. Tuy nhiên, ý nghĩa và mục đích sử dụng của hai mẫu thiết kế này hoàn toàn khác nhau:

## Adapter Pattern hay còn gọi là Wrapper pattern được dùng để biến đổi một class/ interface sang một dạng khác có thể sử dụng được. Adapter Pattern giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể.

## Bridge Pattern được sử dụng được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.

## Adapter Pattern làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại). Bridge Pattern nên được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.

# 2.2.2 Cài đặt Bridge Pattern như thế nào ?

# 

## Một Bridge Pattern bao gồm các thành phần sau:

## Client: đại diện cho khách hàng sử dụng các chức năng thông qua Abstraction.

## Abstraction : định ra một abstract interface quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor).

## Refined Abstraction (AbstractionImpl) : hiện thực (implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implementer.

## Implementor : định ra các interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.

## ConcreteImplementor : hiện thực Implementor interface.

# 2.2.3 Lợi ích của Bridge Pattern là gì ?

## 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.

## 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: trường hợp 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 gàn 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.

# 2.2.4 Sử dụng Bridge Pattern khi nào?

## Khi bạn muốn tách ràng buộc giữa Abstraction và Implementation, để có thể dễ dàng mở rộng độc lập nhau.

## Cả Abstraction và Implementation của chúng nên được mở rộng bằng subsclass.

## Sử dụng ở những nơi mà những thay đổi được thực hiện trong implement không ảnh hưởng đến phía client.

# 2.3 Composite

# 2.3.1 Composite Pattern là gì ?

## Là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.

## Composite Pattern được sử dụng khi chúng ta cần xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object. Composite pattern sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp. Pattern này tạo một lớp chứa nhóm đối tượng của riêng nó. Lớp này cung cấp các cách để sửa đổi nhóm của cùng 1 object. Pattern này cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.

# 2.3.2 Cài đặt Composite Pattern như thế nào?

## Một Composite Pattern bao gồm các thành phần cơ bản sau:

## Base Component: là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.

## Leaf: là lớp hiện thực (implements) các phương thức của Component. Nó là các object không có con.

## Composite: lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.

## Client: sử dụng Base Component để làm việc với các đối tượng trong Composite.

## 

# 2.3.3 Lợi ích của Composite Pattern là gì?

## Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm các đối tượng với nhau.

## 2.3.4 Sử dụng Composite Pattern như thế nào ?

## Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).

## Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

## 2.4 Decorator

## 2.4.1 Decorator Pattern là gì ?

## Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).

## Decorator pattern sử dụng [composition](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/#Cai_dat_Adapter_Pattern_nhu_the_nao) thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là Wrapper hay Smart Proxy.

## 2.4.2 Cài đặt Decorator Pattern như thế nào?

## Decorator Pattern hoạt động dựa trên một đối tượng đặc biệt, được gọi là decorator (hay wrapper). Nó có cùng một interface như một đối tượng mà nó cần bao bọc (wrap), vì vậy phía client sẽ không nhận thấy khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.

## Tất cả các wrapper có một trường để lưu trữ một giá trị của một đối tượng gốc. Hầu hết các wrapper khởi tạo trường đó với một đối tượng được truyền vào constructor của chúng.

## Vậy làm thế nào để có thể thay đổi hành vi của đối tượng? Như đã đề cập, wrapper có cùng interface với các đối tượng đích. Khi bạn gọi một phương thức decorator, nó thực hiện cùng một phương thức trong một đối tượng được wrap và sau đó thêm một cái gì đó (tính năng mới) vào kết quả, công việc này tùy thuộc vào logic nghiệp vụ.

## 

## Các thành phần trong mẫu thiết kế Decorator:

## Component: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.

## ConcreteComponent : là lớp hiện thực (implements) các phương thức của Component.

## Decorator : là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component  interface.

## ConcreteDecorator : là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.

## Client : đối tượng sử dụng Component.

## 2.4.3 Lợi ích của Decorator Pattern là gì ?

## Tăng cường khả năng mở rộng của đối tượng, bởi vì những thay đổi được thực hiện bằng cách implement trên các lớp mới.

## Client sẽ không nhận thấy sự khác biệt khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.

## Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.

## Cho phép thêm hoặc xóa tính năng của một đối tượng lúc thực thi (run-time).

## 2.4.4 Sử dụng Decorator Pattern khi nào ?

## Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này.

## Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator.

## Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.

## 2.4.5 So sánh Decorator và Adapter

## Giống nhau:

## Cả hai đều là [structural pattern](https://gpcoder.com/tag/structuaral-pattern/) như định nghĩa của GOF.

## Cả hai sử dụng cách [composition](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/#Cai_dat_Adapter_Pattern_nhu_the_nao) để cài đặt.

## Khác nhau:

## Decorator cho phép thêm một tính năng mới vào một object nhưng không được phép sử dụng thừa kế. Nó cho phép thay đổi lúc thực thi (run-time). Adapter được sử dụng khi bạn có một interface, và bạn muốn ánh xạ interface đó đến một đối tượng khác có vai trò chức năng tương tự, nhưng là một interface khác.

## Decorator có xu hướng hoạt động trên một đối tượng. Adapter có xu hướng hoạt động trên nhiều đối tượng (có thể wrap nhiều interface).

## 2.5 Facade

## 2.5.1 Facede Pattern là gì ?

## Cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.

## Facade Pattern cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.

## Facade Pattern tương tự với [Adapter Pattern](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/). Hai Pattern này làm việc theo cùng một cách, nhưng mục đích sử dụng của chúng khác nhau. Adapter Pattern chuyển đổi mã nguồn để làm việc được với mã nguồn khác. Nhưng Facade Pattern cho phép bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

## 2.5.2 Cài đặt Facade Pattern như thế nào?

## 

## Các thành phần cơ bản của một Facade Pattern:

## Facade: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của client, sẽ chuyển yêu cầu của client đến các đối tượng của hệ thống con tương ứng.

## Subsystems: cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.

## Client: đối tượng sử dụng Facade để tương tác với các subsystem.

## Các đối tượng Facade thường là [Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/)bởi vì chỉ cần duy nhất một đối tượng Facade.

## 2.5.3 Lợi ích của Facade Pattern là gì?

## Giúp cho hệ thống của bạn trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ chung.

## Giảm sự phụ thuộc của các mã code bên ngoài với hiện thực bên trong của thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động trong phát triển các hệ thống.

## Đóng gói tập nhiều hàm API được thiết kế không tốt bằng một hàm API đơn có thiết kế tốt hơn.

## Sử dụng Facade Pattern khi nào?

## Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.

## Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Façade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác, do đó tăng khả năng độc lập và khả chuyển của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.

## Khi bạn muốn phân lớp các hệ thống con. Dùng Façade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con, do đó giúp giảm sự phụ thuộc của các hệ thống con vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó.

## Khi bạn muốn bao bọc, che giấu tính phức tạp trong các hệ thống con đối với phía Client.

# CHƯƠNG 3: XÂY DỰNG CHƯƠNG TRÌNH DỰA THEO MẪU FACTORY METHOD PATTERN

# KẾT LUẬN

Design Patterns là một vấn đề hết sức quan trọng đối với các tổ chức phát triển phần mềm hiện nay. Trong quá trình thực hiện đồ án của mình do thời gian nghiên cứu và kinh nghiệm bản thân còn hạn chế nên một số phần của đồ án nghiên cứu chưa được sâu.

Sau 03 tháng thực hiện nghiên cứu đề tài, dưới sự hướng dẫn tận tình của Tiến sỹ Nguyễn Trịnh Đông, đồ án của em đã đạt được những kết quả sau:

* Kết quả đạt được
* Hạn chế

Trong thời gian qua, em đã cố gắng hết sức để tìm hiểu thực hiện đề tài. Tuy nhiên với kinh nghiệm và thời gian hạn chế nên không thể tránh khỏi những thiếu sót trong đồ án. Cụ thể:

# DANH MỤC TÀI LIỆU THAM KHẢO