**Hướng dẫn Java Design Pattern – Strategy**

*Đăng vào 03/01/2019 Được đăng bởi*[**GP Coder**](https://gpcoder.com/author/gpadmin/)***12851****Lượt xem*

Có một vài trường hợp, các lớp chỉ khác nhau về hành vi của chúng. Trong trường hợp như vậy, ý tưởng tốt là chúng ta sẽ tách biệt các thuật toán trong các lớp riêng biệt để có khả năng lựa chọn các thuật toán khác nhau trong thời gian chạy (run-time). Ý tưởng này được gọi là Strategy Pattern, một pattern giúp chúng ta giải quyết vấn đề về sự thay đổi, tương tự như [State Design Pattern](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/).

**Nội dung**[[Ẩn](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/)]

* [1 Strategy Pattern là gì?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#Strategy_Pattern_la_gi)
* [2 Cài đặt Strategy Pattern như thế nào?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#Cai_dat_Strategy_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Strategy Pattern với ứng dụng sắp xếp](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#Vi_du_Strategy_Pattern_voi_ung_dung_sap_xep)
  + [2.2 Ví dụ Strategy Pattern với ứng dụng thanh toán online](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#Vi_du_Strategy_Pattern_voi_ung_dung_thanh_toan_online)
* [3 Lợi ích của Strategy Pattern là gì?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#Loi_ich_cua_Strategy_Pattern_la_gi)
* [4 Sử dụng Strategy Pattern khi nào?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#Su_dung_Strategy_Pattern_khi_nao)
* [5 So sánh Strategy Pattern vs State Pattern](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/#So_sanh_Strategy_Pattern_vs_State_Pattern)

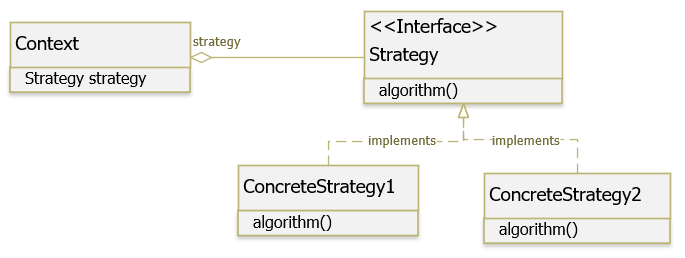
**Strategy Pattern là gì?**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Strategy Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại, và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

Ý nghĩa thực sự của Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

**Cài đặt Strategy Pattern như thế nào?**

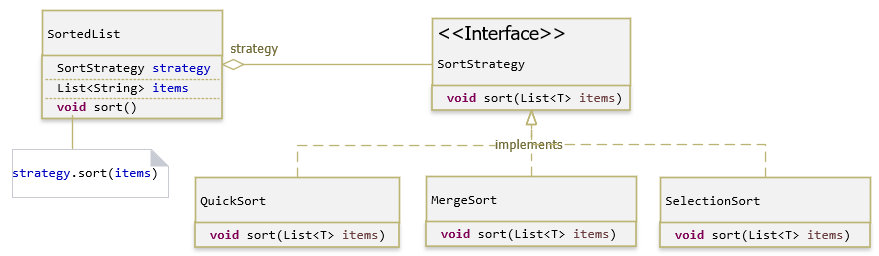
[](https://gpcoder.com/wp-content/uploads/2019/01/design-patterns-strategy-diagram.png)

Các thành phần tham gia Strategy Pattern:

* **Strategy** : định nghĩa các hành vi có thể có của một Strategy.
* **ConcreteStrategy** : cài đặt các hành vi cụ thể của Strategy.
* **Context** : chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.

**Ví dụ Strategy Pattern với ứng dụng sắp xếp**

Chương trình của chúng ta cung cấp nhiều giải thuật sắp xếp khác nhau: quick sort, merge sort, selection sort, heap sort, tim sort, …. Tùy theo loại dữ liệu, số lượng phần tử, … mà người dùng có thể chọn một giải thuật sắp xếp phù hợp.

[](https://gpcoder.com/wp-content/uploads/2019/01/design-patterns-strategy-example.png)

SortStrategy.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | package com.gpcoder.patterns.behavioral.strategy.sort;    import java.util.List;    public interface SortStrategy {        <T> void sort(List<T> items);  } |

QuickSort.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package com.gpcoder.patterns.behavioral.strategy.sort;    import java.util.List;    public class QuickSort implements SortStrategy {        @Override      public <T> void sort(List<T> items) {          System.out.println("Quick sort");      }  } |

MergeSort.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package com.gpcoder.patterns.behavioral.strategy.sort;    import java.util.List;    public class MergeSort implements SortStrategy {        @Override      public <T> void sort(List<T> items) {          System.out.println("Merge sort");      }  } |

SelectionSort.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package com.gpcoder.patterns.behavioral.strategy.sort;    import java.util.List;    public class SelectionSort implements SortStrategy {        @Override      public <T> void sort(List<T> items) {          System.out.println("Selection sort");      }  } |

SortedList.java

|  |  |
| --- | --- |
| 1 |  |

SortStrategy.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package com.gpcoder.patterns.behavioral.strategy.sort;    import java.util.ArrayList;  import java.util.List;    public class SortedList {        private SortStrategy strategy;      private List<String> items = new ArrayList<>();        public void setSortStrategy(SortStrategy strategy) {          this.strategy = strategy;      }        public void add(String name) {          items.add(name);      }        public void sort() {          strategy.sort(items);      }  } |

StrategyPatternExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.gpcoder.patterns.behavioral.strategy.sort;    public class StrategyPatternExample {        public static void main(String[] args) {            SortedList sortedList = new SortedList();          sortedList.add("Java Core");          sortedList.add("Java Design Pattern");          sortedList.add("Java Library");          sortedList.add("Java Framework");            sortedList.setSortStrategy(new QuickSort());          sortedList.sort();            sortedList.setSortStrategy(new MergeSort());          sortedList.sort();      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2 | Quick sort  Merge sort |

**Ví dụ Strategy Pattern với ứng dụng thanh toán online**

Code cũng tương tự như ví dụ trên, các vui lòng xem trên [Github](https://github.com/gpcodervn/Design-Pattern-Tutorial/tree/master/DesignPatternTutorial/src/com/gpcoder/patterns/behavioral/strategy).

**Lợi ích của Strategy Pattern là gì?**

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Single_responsibility_principle_SRP) : một lớp định nghĩa nhiều hành vi và chúng xuất hiện dưới dạng với nhiều câu lệnh có điều kiện. Thay vì nhiều điều kiện, chúng ta sẽ chuyển các nhánh có điều kiện liên quan vào lớp Strategy riêng lẻ của nó.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Open-Closed_principle_OCP) : chúng ta dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng.
* Cung cấp một sự thay thế cho kế thừa.

**Sử dụng Strategy Pattern khi nào?**

* Khi muốn có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
* Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
* Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
* Cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.

**So sánh Strategy Pattern vs State Pattern**

Sự phụ thuộc (dependency):

* [State Pattern](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/) đi kèm với **một chút phụ thuộc** trong các lớp con, chẳng hạn một State biết về các trạng thái khác xuất hiện trước/ sau nó. Ví dụ: t[ín hiệu đèn giao thông](https://github.com/gpcodervn/Design-Pattern-Tutorial/tree/master/DesignPatternTutorial/src/com/gpcoder/patterns/behavioral/state/trafficlight) như đèn đỏ -> đèn xanh -> đèn vàng, ứng dụng workflow nơi mà chúng có thể biết được bước trước và sau nó làm gì.
* Strategy Pattern **không có sự phụ thuộc** như State. Bất kỳ loại trạng thái nào cũng có thể được khởi tạo một cách độc lập, chúng không biết được sự tồn tại của các Strategy khác. Ví dụ: người dùng có thể lựa chọn giải thuật sắp xếp, phương thức thanh toán, nén file rar/zip, xuất báo cáo ra file excel/ csv, …

Mục đích sử dụng (intent) :

* Strategy Pattern quyết định cách thực hiện một số hành động, sử dụng Strategy khi chúng ta cần  trả lời **how**.
* Trong khi State Pattern quyết định khi nào để thực hiện chúng, sử dụng Strategy khi chúng ta cần  trả lời **when/ what**(state or type).

Thời điểm khởi tạo (binding time):

* Strategy Pattern là một mô hình khởi tạo một lần, không có sự luân chuyển trạng thái.
* Trong khi State  Pattern thì năng động hơn (có thể nhiều lần), có sự luân chuyển từ trạng thái này sang trạng thái khác. State Pattern có thể coi là trường hợp mở rộng của Strategy Pattern.

# Hướng dẫn Java Design Pattern – Factory Method

*Đăng vào 12/09/2018 Được đăng bởi*[**GP Coder**](https://gpcoder.com/author/gpadmin/)***27880****Lượt xem*

Trong bài trước chúng ta đã tìm hiểu về [Singleton Design Pattern](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/). Trong bài này chúng ta tiếp tục tìm hiểu một Pattern khác trong nhóm **Creational Design Pattern** là **Factory Method Design Pattern**.

**Nội dung**[[Ẩn](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/)]

* [1 Factory Method Pattern là gì?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/#Factory_Method_Pattern_la_gi)
* [2 Cài đặt Factory Pattern như thế nào?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/#Cai_dat_Factory_Pattern_nhu_the_nao)
* [3 Sử dụng Factory Pattern khi nào?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/#Su_dung_Factory_Pattern_khi_nao)
* [4 Lợi ích của Factory Pattern là gì?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/#Loi_ich_cua_Factory_Pattern_la_gi)

## Factory Method Pattern là gì?

[**Factory Method**](https://en.wikipedia.org/wiki/Factory_method_pattern) is a creational design pattern that Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Factory Method Design Pattern**hay gọi ngắn gọn là **Factory Pattern** là một trong những Pattern thuộc nhóm **Creational Design Pattern**. 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.

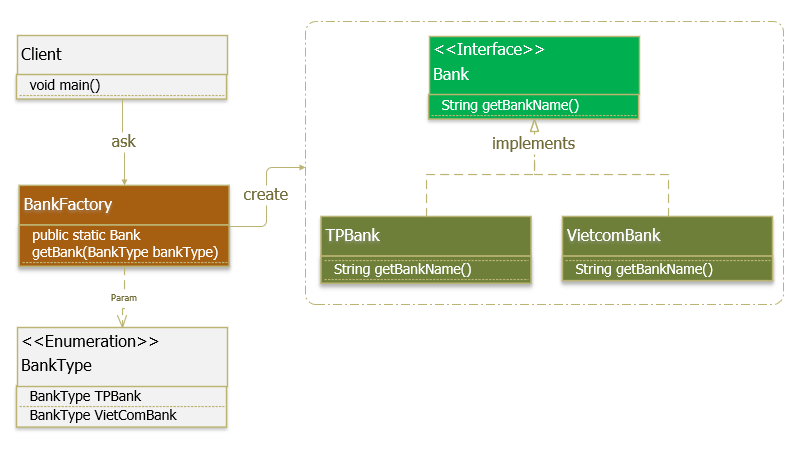
## Cài đặt Factory Pattern như thế nào?

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.

Ví dụ: Tất cả hệ thống ngân hàng có cung cấp API để truy cập đến hệ thống của họ. Team được giao nhiệm vụ thiết kế một API để client có thể sử dụng dịch vụ của một ngân hàng bất kỳ. Hiện tại, phía client chỉ cần sử dụng dịch vụ của 2 ngân hàng là VietcomBank và TPBank. Tuy nhiên để dễ mở rộng sau này, và phía client mong muốn không cần phải thay đổi code của họ khi cần sử dụng thêm dịch vụ của ngân hàng khác. Với yêu cầu như vậy, chúng ta có thể sử dụng một Pattern phù hợp là Factory Method Pattern.

Hệ thống được minh họa như sau:

[](https://gpcoder.com/wp-content/uploads/2018/09/design-patterns-factory-method-diagram.png)

Chương trình được cài đặt theo Factory Pattern như sau:

**Supper Class:**

|  |  |
| --- | --- |
| 1  2  3 | public interface Bank {      String getBankName();  } |

**Sub Classes:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | package com.gpcoder.patterns.creational.factorymethod;    public class TPBank implements Bank {        @Override      public String getBankName() {          return "TPBank";      }    } |
| 1  2  3  4  5  6  7  8  9  10 | package com.gpcoder.patterns.creational.factorymethod;    public class VietcomBank implements Bank {        @Override      public String getBankName() {          return "VietcomBank";      }    } |

**Factory class:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class BankFactory {        private BankFactory() {      }        public static final Bank getBank(BankType bankType) {          switch (bankType) {            case TPBANK:              return new TPBank();            case VIETCOMBANK:              return new VietcomBank();            default:              throw new IllegalArgumentException("This bank type is unsupported");          }      }    } |

**Bank type:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | public enum BankType {        VIETCOMBANK, TPBANK;    } |

**Client:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Client {        public static void main(String[] args) {          Bank bank = BankFactory.getBank(BankType.TPBANK);          System.out.println(bank.getBankName()); // TPBank      }  } |

Như bạn thấy, phía client chỉ cần gọi duy nhất một phương thức **BankFactory.getBank()** là có thể sử dụng được dịch vụ của một ngân hàng bất kỳ.

Khi hệ thống muốn cung cấp thêm dịch vụ của một ngân hàng khác, chẳng hạn VietinBank, thì cần tạo thêm một class mới implement từ interface Bank, và thêm vào logic khởi tạo Bank trong Factory là xong. Nó không làm ảnh hưởng đến code ở phía Client.

## 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.

## 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.
* …

Như vậy tôi đã giới thiệu xong về Factory Method Pattern, hy vọng bài viết sẽ giúp ích cho các bạn trong quá trình xây dựng và phát triển ứng dụng. Mọi góp ý và thắc mắc hãy comment bên dưới, tôi sẽ cố gắng trả lời cho các bạn. Hẹn gặp lại ở các bài viết tiếp theo.

**Hướng dẫn Java Design Pattern – State**

*Đăng vào 31/12/2018 Được đăng bởi*[**GP Coder**](https://gpcoder.com/author/gpadmin/)***7948****Lượt xem*

Trong các ứng dụng, một số đối tượng có thông tin về trạng thái. Hành vi của đối tượng phụ thuộc vào trạng thái của nó tại thời điểm thực thi (run-time) và các phương thức xử lý nghiệp vụ có thể thay đổi trạng thái buộc đối tượng có hành vi xử lý khác đi. Trong trường hợp như vậy, chúng ta có thể sử dụng **State Pattern**.

**Nội dung**[[Ẩn](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/)]

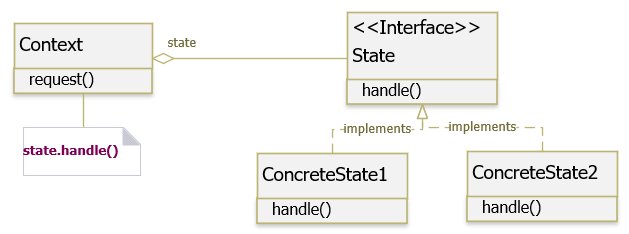
* [1 State Pattern là gì?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/#State_Pattern_la_gi)
* [2 Cài đặt State Pattern như thế nào?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/#Cai_dat_State_Pattern_nhu_the_nao)
  + [2.1 Ví dụ State Pattern](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/#Vi_du_State_Pattern)
* [3 Lợi ích của State Pattern là gì?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/#Loi_ich_cua_State_Pattern_la_gi)
* [4 Sử dụng State Pattern khi nào?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/#Su_dung_State_Pattern_khi_nao)

**State Pattern là gì?**

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

State Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

**Cài đặt State Pattern như thế nào?**

[](https://gpcoder.com/wp-content/uploads/2019/01/design-patterns-state-diagram.png)

Các thành phần tham gia State Pattern:

* **Context** : được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
* **State** : là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng ConcreteState. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
* **ConcreteState** : cài đặt các phương thức của State. Mỗi ConcreteState có thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context.

Một vài điểm cần ghi nhớ khi áp dụng pattern này:

* Một đối tượng nên thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi.
* Mỗi State nên được xác định độc lập.
* Thêm các trạng thái mới sẽ không làm ảnh hưởng đến các trạng thái hoặc chức năng khác.

**Ví dụ State Pattern**

Giả sử chúng ta cần xây dựng một ứng dụng quản lý Document. Một Document có thể bao gồm các trạng thái: tạo mới (New), trình phê duyệt (Submitted), phê duyệt (Approved) và từ chối (Rejected).

Với yêu cầu trên, chương trình của chúng ta như sau:

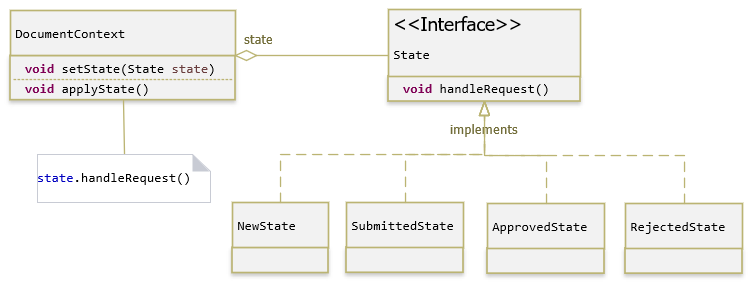
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | package com.gpcoder.patterns.behavioral.state.document.bad;    enum DocumentState {      NEW, SUBMITTED, APPROVED, REJECTED  }    class DocumentService {      private DocumentState state;        public void setState(DocumentState state) {          this.state = state;      }        public void handleRequest() {          switch (state) {          case NEW:              System.out.println("Create a new document");              break;          case SUBMITTED:              System.out.println("Submitted");              break;          case APPROVED:              System.out.println("Approved");              break;          case REJECTED:              System.out.println("Rejected");              break;            default:              break;          }      }  }    public class DocumentApp {        public static void main(String[] args) {          DocumentService service = new DocumentService();            service.setState(DocumentState.NEW);          service.handleRequest();            service.setState(DocumentState.SUBMITTED);          service.handleRequest();            service.setState(DocumentState.APPROVED);          service.handleRequest();      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3 | Create a new document  Submitted  Approved |

Như bạn thấy chương trình trên chạy ok, không vấn đề gì. Nhưng bây giờ chúng ta muốn thêm một trạng thái mới như lưu nháp (Save Draft). Đơn giản chúng ta chỉ cần thêm vào enum một giá trị mới và thêm điều kiện xử lý trong switch-case. Tuy nhiên, nếu làm như vậy thì chúng ta đã vi phạm nguyên tắc Open/ Close. Mỗi khi có thêm một trạng thái mới chúng ta phải sửa nhiều nơi, code trong phương thức handleRequest() ngày càng nhiều và cần phải test lại toàn bộ app.

Bây giờ chúng ta hãy áp dụng State Pattern cho chương trình trên:

[](https://gpcoder.com/wp-content/uploads/2019/01/design-patterns-state-example.png)

* Đầu tiên chúng ta sẽ tạo 1 base inteface để nhận yêu cầu xử lý. Lớp này gọi là **State**.
* Tiếp theo, ứng với mỗi giá trị trong enum, chúng ta sẽ tạo một class mới và implement các phương thức của State.
* Cuối cùng, chúng ta tạo một class **Context**. Class này chứa thông tin State hiện tại và nhận yêu cầu xử lý trực tiếp từ Client.

State.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package com.gpcoder.patterns.behavioral.state.document.improve;    public interface State {        void handleRequest();  } |

NewState.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com.gpcoder.patterns.behavioral.state.document.improve;    public class NewState implements State {        @Override      public void handleRequest() {          System.out.println("Create a new document");      }  } |

SubmittedState.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com.gpcoder.patterns.behavioral.state.document.improve;    public class SubmittedState implements State {        @Override      public void handleRequest() {          System.out.println("Submitted");      }  } |

ApprovedState.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com.gpcoder.patterns.behavioral.state.document.improve;    public class ApprovedState implements State {        @Override      public void handleRequest() {          System.out.println("Approved");      }  } |

RejectedState.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com.gpcoder.patterns.behavioral.state.document.improve;    public class RejectedState implements State {        @Override      public void handleRequest() {          System.out.println("Rejected");      }  } |

DocumentContext.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | package com.gpcoder.patterns.behavioral.state.document.improve;    public class DocumentContext {        private State state;        public void setState(State state) {          this.state = state;      }        public void applyState() {          this.state.handleRequest();      }  } |

StatePatternExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.patterns.behavioral.state.document.improve;    public class StatePatternExample {        public static void main(String[] args) {          DocumentContext context = new DocumentContext();            context.setState(new NewState());          context.applyState();            context.setState(new SubmittedState());          context.applyState();            context.setState(new ApprovedState());          context.applyState();      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3 | Create a new document  Submitted  Approved |

Như bạn thấy, kết quả cũng không đổi. Tuy nhiên, chúng ta rất dễ dàng mở rộng. Nếu muốn thêm một trạng thái mới như lưu tạm (Save Draft), đơn giản tạo một class mới implements từ State mà không làm ảnh hưởng đến các State khác và Context.

**Lợi ích của State Pattern là gì?**

Lợi ích:

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Single_responsibility_principle_SRP) : tách biệt mỗi State tương ứng với 1 class riêng biệt.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Open-Closed_principle_OCP) : chúng ta có thể thêm một State mới mà không ảnh hưởng đến State khác hay Context hiện có.
* Giữ hành vi cụ thể tương ứng với trạng thái.
* Giúp chuyển trạng thái một cách rõ ràng.

**Sử dụng State Pattern khi nào?**

* Khi hành vi của đối tượng phụ thuộc vào trạng thái của nó và nó phải có khả năng thay đổi hành vi của nó lúc run-time theo trạng thái mới.
* Khi nhiều điều kiện phức tạp buộc đối tượng phụ thuộc vào trạng thái của nó.

**Builder Design Pattern**

Trong công nghệ phần mềm, một mẫu thiết kế *design pattern* là một giải pháp tổng thể cho các vấn đề chung trong thiết kế phần mềm, giúp thiết kế của chúng ta linh hoạt, dễ dàng thay đổi và bảo trì hơn.

Ở trong bài ngày hôm nay, mình xin giới thiệu với các bạn một loại *design pattern*, đó là **Builder Pattern**.

**Builder Pattern là gì?**

Tất nhiên rồi **Builder Pattern** là một trong những design pattern, nhưng cụ thể **Builder pattern** là một trong những *Creational pattern* - những mẫu thiếu kế cho việc khởi tạo đối tượng của lớp.

Hơn nữa, **Builder pattern** được tạo ra để xây dựng một đôi 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 giải quyết vấn đề gì?**

Tất cả chúng ta đều biết, ở ngôn ngữ Java, trong mỗi lớp đều có những hàm *constructor* và nếu chúng không được khai báo thì trình biên dịch sẽ tự động xây dựng một hàm *constructor* mặc định cho lớp ấy.

Khi chúng ta xây dựng, chúng ta cũng thể tùy ý lựa chọn những tham số truyền vào cho *constructor* miễn là nó hữu ích cho chúng ta.

Như vậy, vấn đề đặt ra ở đây khi một đối tượng của chúng ta có thể được khởi tạo với rất nhiều những tham số truyền vào và có thể một vài trong số đó không nhất thiết phải truyền vào trong lúc khởi tạo(có thể có hoặc không).

public Student(String id, String firstName, String lastName, String dayOfBirth, String currentClass, String address,

String phone) {

this.id = id;

this.firstName = firstName;

this.lastName = lastName;

this.dayOfBirth = dayOfBirth;

this.currentClass = currentClass;

this.address = address;

this.phone = phone;

}

Không sao cả, chúng ta có thể dễ dàng giải quyết bằng cách truyền những giá trị *null* vào tham số trong hàm *constructor* hoặc là nạp chồng nhiều hàm *constructor* khác với những tham số tùy ý. Khi đó lớp của chúng ta sẽ trông như này:

public Student(String id, String firstName, String lastName) {

this.id = id;

this.firstName = firstName;

this.lastName = lastName;

}

public Student(String id, String currentClass, String address, String phone) {

this.id = id;

this.currentClass = currentClass;

this.address = address;

this.phone = phone;

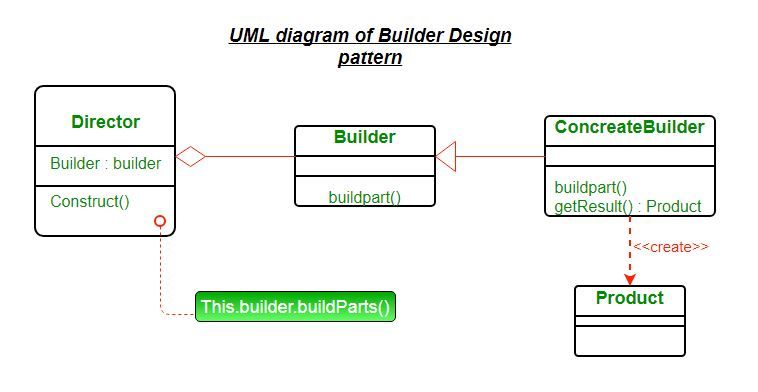
}

Cách này ban đầu nghe có vẻ ổn nhưng sẽ có những nhược điểm như sau:

* Phải tạo rất nhiều hàm constructor trong những trường hợp khác nhau
* Khó khăn trong việc xác định thứ tự của những tham số truyền vào

Vấn đề này lại sinh ra vấn đề khác, vậy liệu **Builder Pattern** sẽ giải quyết vấn đề này như thế nào? Chúng ta cùng xem cấu tạo của **Builder Pattern** nhé!

**Cấu tạo Builder Pattern như nào?**

Đây là biểu đồ lớp khi chúng ta thiết kế Builder Pattern

Chúng ta có thể thấy, Builder Pattern sẽ gồm có 4 thành phần chính:

* *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*: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

**Cài đặt Builder Pattern**

Bây giờ chúng ta cùng nhau cài đặt *Builder Pattern* nhé

Như mình đã nói ở trên, chúng ta sẽ có 4 thành phần chính: *Product, Builder, ConcreteBuilder, Director*.

Mình sẽ thiết kế thành phần *Product* trước, để hình dung ra được đầu ra của chúng ta là gì, trong ví dụ của mình, đại diện cho thành phần *Product* là lớp *Student*

Rất đơn giản thôi, lớp này chúng ta sẽ khai báo những thuộc tính của *Student*, ở đây mình chỉ viết thêm những phương thức *getter* vì không cần có nhu cầu gán giá trị qua những hàm *setter*

// Product

public class Student {

private String id;

private String firstName;

private String lastName;

private String dayOfBirth;

private String currentClass;

private String phone;

public Student(String id, String firstName, String lastName, String dayOfBirth, String currentClass, String phone) {

this.id = id;

this.firstName = firstName;

this.lastName = lastName;

this.dayOfBirth = dayOfBirth;

this.currentClass = currentClass;

this.phone = phone;

}

}

Tiếp theo chúng ta cùng thiết kế thành phần *Builder*, định nghĩa những phương thức của *Builder*

// Builder

public interface StudentBuilder {

StudentBuilder setId(String id);

StudentBuilder setFirstName(String firstName);

StudentBuilder setLastName(String lastName);

StudentBuilder setDayOfBirth(String dayOfBirth);

StudentBuilder setCurrentClass(String currentClass);

StudentBuilder setPhone(String phone);

Student build();

}

Sau khi định nghĩa được những phương thức của *Builder* rồi, ta cùng viết thành phần *ConcreteBuilder* kế thừa từ *Builder*

// ConcreteBuilder

public class StudentConcreteBuilder implements StudentBuilder {

private String id;

private String firstName;

private String lastName;

private String dayOfBirth;

private String currentClass;

private String phone;

@Override

public StudentBuilder setId(String id) {

this.id = id;

return this;

}

@Override

public StudentBuilder setFirstName(String firstName) {

this.firstName = firstName;

return this;

}

@Override

public StudentBuilder setLastName(String lastName) {

this.lastName = lastName;

return this;

}

@Override

public StudentBuilder setDayOfBirth(String dayOfBirth) {

this.dayOfBirth = dayOfBirth;

return this;

}

@Override

public StudentBuilder setCurrentClass(String currentClass) {

this.currentClass = currentClass;

return this;

}

@Override

public StudentBuilder setPhone(String phone) {

this.phone = phone;

return this;

}

@Override

public Student build() {

return new Student(id, firstName, lastName, dayOfBirth, currentClass, phone);

}

}

Như vậy là chúng ta đã thiết kế xong 3 thành phần rồi, còn thành phần cuối cùng là *Director* chính là nơi chúng ta gọi ra *Builder* để khởi tạo đối tượng, ví dụ này mình sẽ khởi tạo đối tượng trong hàm \*main() \*

public static void main(String[] args) {

StudentBuilder studentBuilder = new StudentConcreteBuilder()

.setFirstName("Tran")

.setLastName("Quang Huy");

System.out.println(studentBuilder.build());

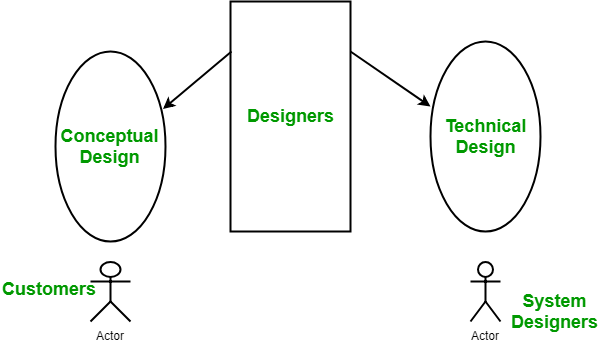
}

Software Engineering | Coupling and Cohesion

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium)
* Last Updated : 08 Jan, 2020

**Introduction:** The purpose of Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS(Software Requirement Specification) document. The output of the design phase is Sofware Design Document (SDD).

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer’s problem.



**Conceptual design of system:**

* Written in simple language i.e. customer understandable language.
* Detail explanation about system characteristics.
* Describes the functionality of the system.
* It is independent of implementation.
* Linked with requirement document.

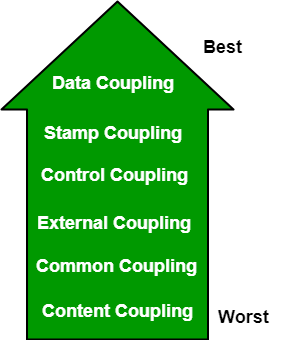
**Technical Design of system:**

* Hardware component and design.
* Functionality and hierarchy of software component.
* Software architecture
* Network architecture
* Data structure and flow of data.
* I/O component of the system.
* Shows interface.

**Modularization:** Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

* Easy to understand the system.
* System maintenance is easy.
* A module can be used many times as their requirements. No need to write it again and again.

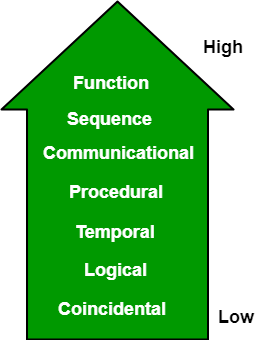
**Coupling:** Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.



**Types of Coupling:**

* **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don’t contain tramp data. Example-customer billing system.
* **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice made by the insightful designer, not a lazy programmer.
* **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
* **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
* **Common Coupling:** The modules have shared data such as global data structures.The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
* **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

**Cohesion:** Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



**Types of Cohesion:**

* **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
* **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
* **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record int the database and send it to the printer.
* **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
* **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at init time.
* **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
* **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Attention reader! Don’t stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the [**CS Theory Course**](https://practice.geeksforgeeks.org/courses/SDE-theory?vC=1) at a student-friendly price and become industry ready.

## SOLID là gì?

**SOLID** là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc thiết kế hướng đối tượng. Giúp cho lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain. Nó được đưa ra bởi [Robert C. Martin](http://www.goodreads.com/author/show/45372.Robert_C_Martin) và Michael Feathers. 5 nguyên tắc đó bao gồm:

* **S**ingle responsibility priciple (SRP)
* **O**pen/Closed principle (OCP)
* **L**iskov substitution principe (LSP)
* **I**nterface segregation principle (ISP)
* **D**ependency inversion principle (DIP)

## Single responsibility priciple

**Nội dung:**

Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi.

Nguyên lý đầu tiên ứng với chữ S trong SOLID, có ý nghĩa là một class chỉ nên giữ một trách nhiệm duy nhất. Một class có quá nhiều chức năng sẽ trở nên cồng kềnh và trở nên khó đọc, khó maintain. Mà đối với ngành IT việc requirement thay đổi, cần thêm sửa chức năng là rất bình thường, nên việc code trong sáng, dễ đọc dễ hiểu là rất cần thiết.

Ví dụ: Hình dung rằng nhân viên của một công ty phần mềm cần phải làm 1 trong 3 việc sau đây: lập trình phần mềm (developer), kiểm tra phần mềm (tester), bán phần mềm (salesman). Mỗi nhân viên sẽ có một chức vụ và dựa vào chức vụ sẽ làm công việc tương ứng. Khi đó bạn có nên thiết kế lớp “Employee” với thuộc tính “position” và 3 phương thức developSoftware(), testSoftware() và saleSoftware() không?

class Employee

{

string position;

function developSoftware(){};

function testSoftware(){};

function saleSoftware(){};

}

Câu trả lời là KHÔNG. Thử hình dung nếu có thêm một chức vụ nữa là quản lí nhân sự, ta sẽ phải sửa lại lớp “Employee”, thêm phương thức mới vào sao? Nếu có thêm 10 chức vụ nữa thì sao? Khi đó các đối tượng được tạo ra sẽ dư thừa rất nhiều phương thức: Developer thì đâu cần dùng hàm testSoftware() và saleSoftware() đúng không nào, lỡ may dùng lầm phương thức cũng sẽ gây hậu quả khôn lường.

**Áp dụng nguyên tắc Single Responsibility**: mỗi lớp 1 trách nhiệm. Ta sẽ tạo 1 lớp trừu tượng là “Employee” có phương thức là working(), từ đây bạn kế thừa ra 3 lớp cụ thể là Developer, Tester và Salesman. Ở mỗi lớp này bạn sẽ implement phương thức working() cụ thể tuy theo nhiệm vụ của từng người. Khi đó chúng ta sẽ bị tình trạng dùng nhầm phương thức nữa.

## Open/Closed principle

**Nội dung:**

Không được sửa đổi một Class có sẵn, nhưng có thể mở rộng bằng kế thừa.

Nguyên lý thứ 2 ứng với chữ O trong SOLID.

Theo nguyên lý này, mỗi khi ta muốn thêm chức năng cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) chứ không nên sửa đổi class cũ. Việc này dẫn đến tình trạng phát sinh nhiều class, nhưng chúng ta sẽ không cần phải test lại các class cũ nữa, mà chỉ tập trung vào test các class mới, nơi chứa các chức năng mới.

Thông thường việc mở rộng thêm chức năng thì phải viết thêm code, vậy để thiết kế ra một module có thể dễ dàng mở rộng nhưng lại hạn chế sửa đổi code ta cần làm gì. Cách giải quyết là tách những phần dễ thay đổi ra khỏi phần khó thay đổi mà vẫn đảm bảo không ảnh hưởng đến phần còn lại.

**Ví dụ:**

* Đặt vấn đề: Ta cần 1 lớp đảm nhận việc kết nối đến CSDL. Thiết kế ban đầu chỉ có SQL Server và MySQL. Thiết kế ban đầu có dạng như sau:

class ConnectionManager

{

public function doConnection(Object $connection)

{

if($connection instanceof SqlServer) {

//connect with SqlServer

} elseif($connection instanceof MySql) {

//connect with MySql

}

}

}

Sau đó yêu cầu đặt ra phải kết nối thêm đến Oracle và một vài hệ CSDL khác.  
Để thêm chức năng ta phải thêm vào code những khối esleif khác, việc này làm code cồng kềnh và khó quản lý hơn.

* Giải pháp:
  + Áp dụng Abstract thiết kế lại các lớp SqlServer, MySql, Oracle…
  + Các lớp này đều có chung nhiệm vụ tạo kết nối đến csdl tương ứng có thể gọi chung là Connection.
  + Cách thức kết nối đến csdl thay đổi tùy thuộc vào từng loại kết nối nhưng có thể gọi chung là doConect.
  + Vậy ta có lớp cơ sở Connection có phương thức doConnect, các lớp cụ thể là SqlServer, MySql, Oracle… kế thừa từ Connection và overwrite lại phương thức doConnect phù hợp với lớp đó.

Thiết kế sau khi làm lại có dạng như sau:

abstract class Connection()

{

public abstract function doConnect();

}

class SqlServer extends Connection

{

public function doConnect()

{

//connect with SqlServer

}

}

class MySql extends Connection

{

public function doConnect()

{

//connect with MySql

}

}

class ConnectionManager

{

public function doConnection(Connection $connection)

{

//something

//.................

//connection

$connection->doConnect();

}

}

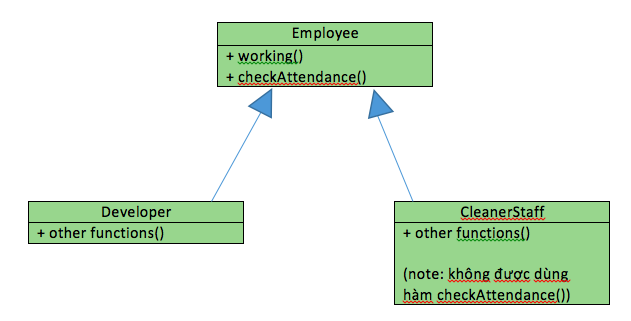
Với thiết kế này khi cần kết nối đến 1 loại csdl mới chỉ cần thêm 1 lớp mới kế thừa Connection mà không cần sửa đổi code của lớp ConnectionManager, điều này thỏa mãn 2 điều kiện của nguyên lý OCP.

## Liskov substitution principle

**Nội dung:**

Các đối tượng (instance) kiểu class con có thể thay thế các đối tượng kiểu class cha mà không gây ra lỗi.

Nguyên tắc thứ 3, ứng với chữ L trong SOLID.

Minh hoạ một trường hợp vi phạm nguyên tắc Liskov substitution. Nếu thiết kế lớp như thế này, thì lớp CleanerStaff sẽ dùng được hàm checkAttendance(), mà điều này là không đúng, nên đây sẽ là một kiểu thiết kế sai nguyên tắc.

Quay trở lại ví dụ lớp Emloyee trong phần 1, ta giả sử có công ty sẽ điểm danh vào mỗi buổi sáng, và chỉ có các nhân viên thuộc biên chế chính thức mới được phép điểm danh. Ta bổ sung phương thức checkAttendance() vào lớp Employee.

Hình dung có một trường hợp sau: công ty thuê một nhân viên lao công để làm vệ sinh văn phòng, mặc dù là một người làm việc cho công ty nhưng do không được cấp số ID nên không được xem là một nhân viên bình thường, mà chỉ là một nhân viên thời vụ, do đó sẽ không được điểm danh.

**Nguyên tắc này nói rằng**: Nếu chúng ta tạo ra một lớp CleanerStaff kế thừa từ lớp Employee, và implement hàm working() cho lớp này, thì mọi thứ đều ổn, tuy nhiên lớp mới này cũng lại có hàm checkAttendance() để điểm danh, mà như thế là sai quy định dẫn đến chương trình bị lỗi. Như vậy, thiết kế lớp CleanerStaff kế thừa từ lớp Employee là không được phép.

Có nhiều cách để giải quyết tình huống này ví dụ như tách hàm checkAttendance() ra một interface riêng và chỉ cho các lớp Developer, Tester và Salesman implements interface này.

## Interface segregation principle

**Nội dung:**

Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể.

Nguyên lý này rất dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn, khoảng 100 methods. Việc implements sẽ rất vất vả vì các class impliment interface này sẽ bắt buộc phải phải thực thi toàn bộ các method của interface. Ngoài ra còn có thể dư thừa vì 1 class không cần dùng hết 100 method. Khi tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc implement và quản lý sẽ dễ hơn.

Ví dụ:

Chúng ta có một interface Animal như sau:

interface Animal {

void eat();

void run();

void fly();

}

Chúng ta có 2 class Dog và Snake implement interface Animal. Nhưng thật vô lý, Dog thì làm sao có thể fly(), cũng như Snake không thể nào run() được? Thay vào đó, chúng ta nên tách thành 3 interface như thế này:

interface Animal {

void eat();

}

interface RunnableAnimal extends Animal {

void run();

}

interface FlyableAnimal extends Animal {

void fly();

}

## Dependency inversion principle

**Nội dung:**

1.Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

2.Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại (Các class giao tiếp với nhau thông qua interface (abstraction), không phải thông qua implementation.)

Có thể hiểu nguyên lí này như sau: những thành phần trong 1 chương trình chỉ nên phụ thuộc vào những cái trừu tượng (abstraction). Những thành phần trừu tượng không nên phụ thuộc vào các thành phần mang tính cụ thể mà nên ngược lại.

Những cái trừu tượng (abstraction) là những cái ít thay đổi và biến động, nó tập hợp những đặc tính chung nhất của những cái cụ thể. Những cái cụ thể dù khác nhau thế nào đi nữa đều tuân theo các quy tắc chung mà cái trừu tượng đã định ra. Việc phụ thuộc vào cái trừu tượng sẽ giúp chương trình linh động và thích ứng tốt với các sự thay đổi diễn ra liên tục.

**Ví dụ:**

Lấy ví dụ về ổ cứng của máy tính, bạn có thể dùng loại ổ cứng thể rắn SSD đời mới để chạy cho nhanh, tuy nhiên cũng có thể dùng ổ đĩa quay HDD thông thường. Nhà sản xuất Mainboard không thể nào biết bạn sẽ dùng ổ SSD hay loại HDD đĩa quay thông thường. Tuy nhiên họ sẽ luôn đảm bảo rằng bạn có thể dùng bất cứ thứ gì bạn muốn, miễn là ổ đĩa cứng đó phải có chuẩn giao tiếp SATA để có thể gắn được vào bo mạch chủ. Ở đây chuẩn giao tiếp SATA chính là interface, còn SSD hay HDD đĩa quay là implementation cụ thể.

Trong khi lập trình cũng vậy, khi áp dụng nguyên lý này, ở những lớp trừu tượng cấp cao, ta thường sử dụng interface nhiều hơn thay vì một kiểu kế thừa cụ thể. Ví dụ, để kết nối tới Database, ta thường thiết kế lớp trừu tượng DataAccess có các phương thức phương thức chung như save(), get(), … Sau đó tùy vào việc sử dụng loại DBMS nào (vd: MySql, MongoDB, …) mà ta kế thừa và implement những phương thức này. Tính chất đa hình của OOP được vận dụng rất nhiều trong nguyên lý này.

## Tổng kết

**SOLID** là 5 nguyên tắc cơ bản trong việc thiết kế phần mềm. Nó giúp chúng ta tổ chức sắp xếp các function, method, class một cách chính xác hơn. Làm sao để kết nối các thành phần, module với nhau.

### Rõ ràng, dễ hiểu

Teamwork là điều không thể tránh trong lập trình. Áp dụng SOLID vào công việc bạn sẽ tạo ra các hàm tốt, dễ hiểu hơn. Giúp cho bạn và đồng nghiệp đọc hiểu code của nhau tốt hơn.

### Dễ thay đổi

SOLID giúp tạo ra các module, class rõ ràng, mạch lạc, mang tính độc lập cao. Do vậy khi có sự yêu cầu thay đổi, mở rộng từ khách hàng, ta cũng không tốn quá nhiều công sức để thực hiện việc thay đổi.

### Tái sử dụng

SOLID khiến các lập trình viên suy nghĩ nhiều hơn về cách viết phần mềm, do vậy code viết ra sẽ mạch lạc, dễ hiểu, dễ sử dụng.

Nguồn tham khảo: