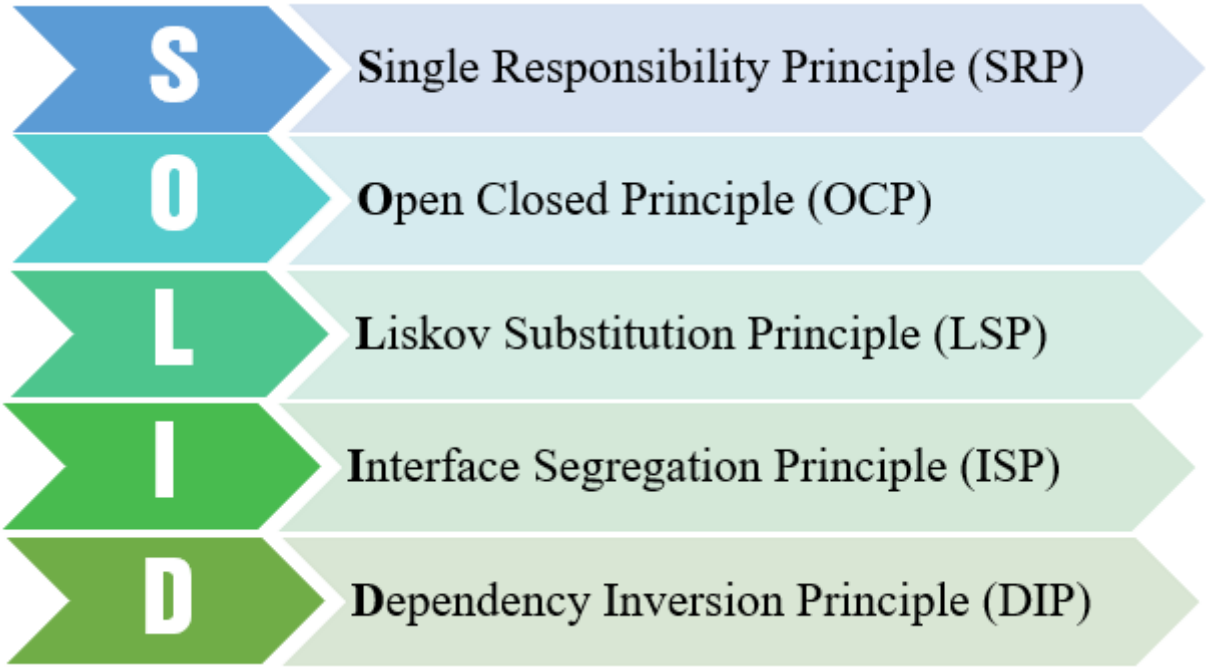
Các nguyên lý thiết kế hướng đối tượng – SOLID

Đăng vào **09/08/2018** *Được đăng bởi* **GP Coder** ***12116** Lượt xem*

Chào các bạn, trong các bài viết trước tôi đã giới thiệu với các bạn 4 tính chất cơ bản của lập trình hướng đối tượng trong Java. Đây là các tính chất rất quan trọng của lập trình hướng đối tượng (OOP) mà hầu hết chúng ta đã biết, nhưng cách thức để phối hợp các tính chất này với nhau để tăng hiệu quả của ứng dụng thì không phải ai cũng nắm được. Một trong những nguyên tắc để giúp chúng ta xây dựng được các ứng dụng OOP hiệu quả hơn đó là SOLID, nó là một bộ 5 nguyên tắc đã được nhắc tới từ lâu bởi các nhà phát triển phần mềm và được tổng hợp, phát biểu thành nguyên tắc bởi Robert C. Martin, cũng chính là tác giả của các cuốn sách nổi tiếng: Clean Code: A Handbook of Agile Software Craftsmanship, The Clean Coder: A Code of Conduct for Professional Programmers, …

Trong phần tiếp theo của bài viết này tôi sẽ giới thiệu với các bạn các nguyên lý thiết kế hướng đối tượng (Object Oriented Design Principle), đó là **S.O.L.I.D**. Đây được xem là 5 nguyên lý hàng đầu trong việc thiết kế chương trình ở mức lớp và đối tượng.



Mục tiêu của những nguyên tắc này là tạo ra những sự thay đổi mã code ít ảnh hưởng các phần còn lại. Điều đó có nghĩa là khi sửa hoặc cải tiến code nên gây ra ảnh hưởng đến các phần còn lại càng ít càng tốt. Chúng ta giảm bớt chi phí bảo trì thông qua một thiết kế được thực hiện để thích ứng với thay đổi.

Nội dung [Ẩn]

1 Single responsibility principle (SRP) – Nguyên lý đơn chức năng
2 Open-Closed principle (OCP) – Nguyên lý đóng mở
3 Liskov substitution principle (LSP) – Nguyên lý thay thế
  3.1 Ví dụ class con thay đổi hành vi class cha
  3.2 Ví dụ class con throw exception khi gọi hàm
  3.3 Những vi phạm về nguyên lý LSP
  3.4 Lưu ý
4 Interface segregation principle (ISP) – Nguyên lý phân tách
5 Dependency Inversion Principle (DIP) – Nguyên lý đảo ngược phụ thuộc
6 Lời kết

Single responsibility principle (SRP) – Nguyên lý đơn chức năng

Nguyên tắc này được phát biểu như sau:

> Một class chỉ nên giữ 1 trách nhiệm duy nhất, chỉ có thể sửa đổi class với 1 lý do duy nhất.
>
> *A class should have one and only one reason to change, meaning that a class should have only one job.*

Để hiểu nguyên lý này, hãy xem ví dụ với 1 class vi phạm nguyên lý như sau:

```
1  package com.gpcoder.solid;
2
3  /**
4   * SRP – Single responsibility principle example
5   *
6   * @author gpcoder
7   */
8  class UserService {
9      // Get data from database
10     public User getUser() {
11         return null;
12     }
13
14     // Check validation
15     public boolean isValid() {
16         return true;
17     }
18
19     // Show Notification
20     public void showNotification() {
21
22     }
23
24     // Logging
25     public void logging() {
26         System.out.println("...");
27     }
28
29     // Parsing
30     public User parseJson(String json) {
```

```
            return null;
        }
    }
```

Như bạn thấy, class này thực hiện rất nhiều trách nhiệm khác nhau: lấy dữ liệu từ DB, validate, thông báo, ghi log, xử lý dữ liệu, … Khi chỉ cần ta thay đổi cách lấy dữ liệu DB, thay đổi cách validate, … ta sẽ phải sửa đổi class này, càng về sau class sẽ càng phình to ra. Rất khó khăn khi maintain, upgrade, fix bug, test, …

Theo đúng nguyên tắc này, ta phải tách class này ra làm nhiều class riêng, mỗi class chỉ làm một nhiệm vụ duy nhất. Tuy số lượng class nhiều hơn những việc sửa chữa sẽ đơn giản hơn, dễ dàng tái sử dụng hơn, class ngắn hơn nên cũng ít bug hơn.

Chẳng hạn, với chương trình trên chúng ta có thể tách thành các class: UserRepository, UserValidator, SystemLogger, JsonConverter, ….

Một số ví dụ về SRP cần xem xét có thể cần được tách riêng bao gồm: Persistence, Validation, Notification, Error Handling, Logging, Class Instantiation, Formatting, Parsing, Mapping, …

Open-Closed principle (OCP) – Nguyên lý đóng mở

Nguyên tắc này được phát biểu như sau:

> Có thể thoải mái mở rộng 1 class, nhưng không được sửa đổi bên trong class đó.
>
> *Objects or entities should be open for extension, but closed for modification.*

Nghe qua thấy nguyên lý có sự mâu thuẫn do thường chúng ta thấy rằng dễ mở rộng là phải dễ thay đổi, đằng nay dễ mở rộng nhưng không cho thay đổi. Thực sự theo nguyên lý này, chúng ta không được thay đổi hiện trạng của các lớp có sẵn, nếu muốn thêm tính năng mới, thì hãy mở rộng class cũ bằng cách kế thừa để xây dựng class mới. Làm như vậy sẽ tránh được các tình huống làm hỏng tính ổn định của chương trình đang có.

Lợi ích của nguyên lý này là chúng ta không phải lo lắng về code sử dụng các class nguồn bởi vì chúng ta đã không sửa đổi chúng, vì vậy hành vi của chúng phải giống nhau. Tuy nhiên, chúng ta nên chú ý vào ý nghĩa của các chức năng, tránh tạo ra quá nhiều class dẫn xuất. Mặc dù những sửa đổi nhỏ trong class thường không ảnh hưởng, chúng ta cũng cần phải test cẩn thận. Và đó là lý do chính tại sao chúng ta cần phải viết test case cho các chức năng, để có thể nhận thấy hành vi không mong muốn xảy ra trong code.

Quay lại với ví dụ ở đoạn code ở trên, nếu các thao tác validation để cùng với logic thì chúng ta có thể gặp vấn đề sau:

- Thêm một validation mới chúng ta phải trực tiếp sửa code bằng if-else condition.
- Sửa code nếu validation bị thay đổi logic.
- Testing khó khăn, chúng ta phải test cả phần thực hiện logic và validation.

Bây giờ, nếu chúng ta chuyển các thao tác validation sang các lớp khác để xử lý. Cách giải quyết này được gọi là **Dependence Injection**. Nếu chúng ta muốn thay đổi cách validate khác cho user, chỉ cần thay đổi class validator truyền vào.

Thực hiện theo cách này chúng ta đã hoàn thành nguyên tắc **Single responsibility principle** (chúng ta đã chuyển trách nhiệm bổ sung sang một class khác). Bây giờ, chúng ta không phải sửa đổi class gốc nếu chúng ta muốn thêm một class khác để validate dữ liệu. Chúng ta chỉ cần tạo một class thích hợp mới và gọi nó là tham số trong trường hợp phù hợp.

```
1  package com.gpcoder.solid;
2
3  /**
4   * OCP - Open/ Closed principle example
5   *
6   * @author gpcoder
7   */
8  class UserServiceV2 {
9      private Validator validator;
```

```
10
11      public UserServiceV2(Validator validator) {
12          this.validator = validator;
13      }
14
15      public void saveUser() {
16          if (this.validator.isValid()) {
17              // Do save
18          } else {
19              // Show error
20          }
21      }
22  }
23
24  interface Validator {
25      boolean isValid();
26  }
27
28  class UserValidator1 implements Validator {
29      @Override
30      public boolean isValid() {
31          return true;
32      }
33  }
34
35  class UserValidator2 implements Validator {
36      @Override
37      public boolean isValid() {
38          return false;
39      }
40  }
41
42  public class OCPExample {
43      public static void main(String[] args) {
44          UserServiceV2 userService1 = new UserServiceV2(new Use
45          UserServiceV2 userService2 = new UserServiceV2(new Use
46      }
47  }
```

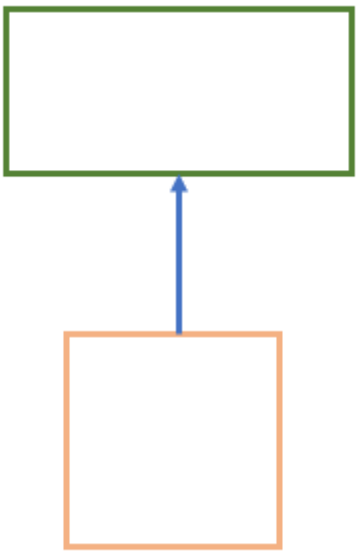
Nguyên lý Open-Closed cũng có thể đạt được bằng nhiều cách khác, bao gồm cả việc sử dụng thừa kế (**inheritance**) hoặc thông qua các mẫu thiết kế tổng hợp (compositional design patterns) như **Strategy pattern**. Chúng ta sẽ tìm hiểu các mẫu thiết kế này trong các bài viết tiếp theo.

Liskov substitution principle (LSP) – Nguyên lý thay thế

Nguyên tắc này được phát biểu như sau:

> Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình.
>
> *Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*

Ví dụ class con thay đổi hành vi class cha



Để minh họa điều này, chúng ta sẽ đi với một ví dụ kinh điển kinh điển về hình vuông và hình chữ nhật mà mọi người thường dùng để giải thích LSP vì nó rất đơn giản và dễ hiểu.

```
1  package com.gpcoder.solid;
2
3  /**
4   * LSP - Liskov substitution principle example
5   *
6   * @author gpcoder
7   */
8  class Rectangle {
9      private int width;
10     private int height;
11
12     public int calculateArea() {
13         return this.width * this.height;
14     }
15
16     public void setWidth(int width) {
17         this.width = width;
18     }
19
20     public void setHeight(int height) {
21         this.height = height;
22     }
23 }
24
25 class Square extends Rectangle {
26
27     @Override
28     public void setWidth(int width) {
29         super.setWidth(width);
30         super.setHeight(width);
31     }
32
33     @Override
34     public void setHeight(int height) {
35         super.setWidth(height);
```

```
36            super.setHeight(height);
37        }
38    }
39
40    public class LSPExample1 {
41        public void example1() {
42            Rectangle rect = new Rectangle();
43            rect.setWidth(5);
44            rect.setHeight(10);
45            System.out.println(rect.calculateArea()); // 50
46
47            Square square = new Square();
48            square.setWidth(5);
49            square.setHeight(10);
50            System.out.println(rect.calculateArea()); // 100
51        }
52    }
```

Nhìn ví dụ trên ta thấy mọi tính toán đều rất hợp lý. Do hình vuông có 2 cạnh bằng nhau, mỗi khi set độ dài 1 cạnh thì ta set luôn độ dài của cạnh còn lại.

Tuy nhiên, Class Square kế thừa từ class Rectangle nhưng class Square có những hình vi khác và nó đã thay đổi hành vi của của class *Rectangle*, dẫn đến vi phạm LSP.

Như ví dụ trên, do Class Square kế thừa từ class Rectangle nên chúng ta có thể sử dụng như sau:

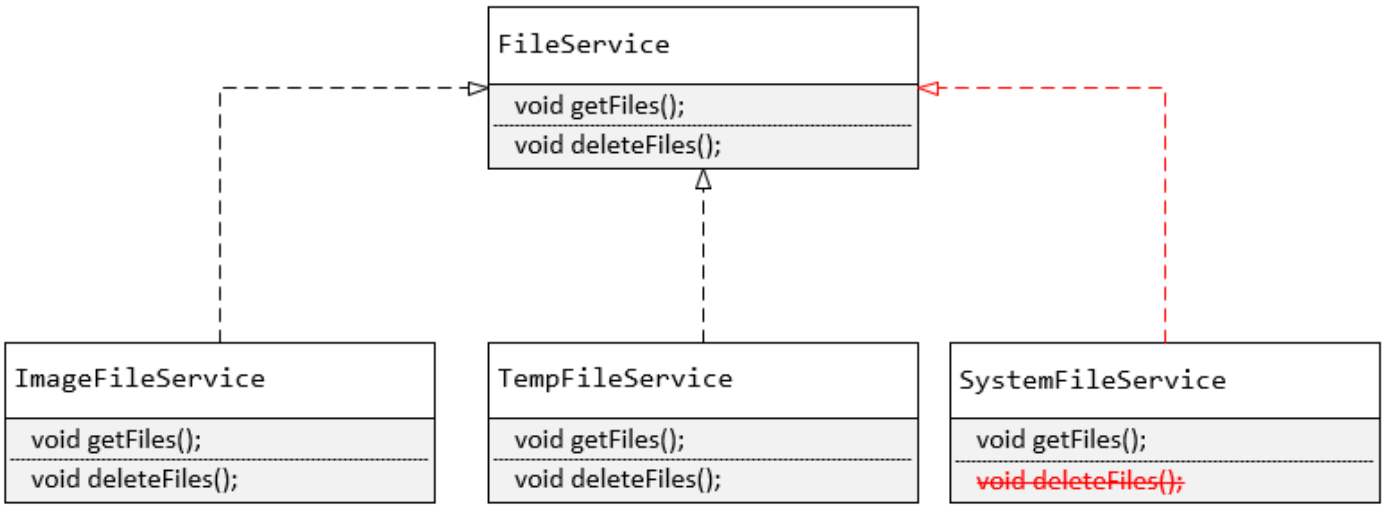
```
1   public class LSPExample1 {
2       public void example2() {
3           Rectangle rect = new Square();
4           rect.setWidth(5);
5           rect.setHeight(10);
6           System.out.println(rect.calculateArea()); // 100
7       }
8   }
```

Rõ ràng kết quả không đúng, diện tích của hình chữ nhật phải là 5*10=50.

Theo nguyên tắc này, chúng ta phải bảo đảm rằng khi một lớp con kế thừa từ một lớp khác, nó sẽ không làm thay đổi hành vi của lớp đó.

Trong trường hợp này, để code không vi phạm nguyên lý LSP, chúng ta phải tạo 1 class cha là class Shape, sau đó cho Square và Rectangle kế thừa class Shape này.

Ví dụ class con throw exception khi gọi hàm



Một trường hợp khác cũng vi phạm LSP là class con throw exception khi gọi hàm.

```
1  package com.gpcoder.solid;
2
3  /**
4   * LSP - Liskov substitution principle example
5   *
6   * @author gpcoder
7   */
8  interface FileService {
9      void getFiles();
10     void deleteFiles();
11 }
12
13 class ImageFileService implements FileService {
14
15     @Override
16     public void getFiles() {
17         // Load image files
18     }
19
20     @Override
21     public void deleteFiles() {
22         // Delete image files
23     }
24 }
25
26 class TempFileService implements FileService {
27
28     @Override
29     public void getFiles() {
30         // Load temp files
31     }
32
33     @Override
34     public void deleteFiles() {
35         // Delete temp files
36     }
37 }
```

Những class implement ở trên không có vấn đề gì, mọi thứ đều chạy tốt. Bây giờ chúng ta thêm một class SystemFileService mới. Với yêu cầu là không được xóa các file hệ thống, nên lớp này sẽ quăng ra lỗi

UnsupportedOperationException. Một phương thức được thiết kế nhưng không được sử dụng, đây cũng không phải là một thiết kế tốt.

Khi thực thi phương thức deleteFiles(), class SystemFileService gây lỗi khi chạy. Nó không thay thế được class cha của nó là FileService, vì thế nó đã vi phạm LSP.

```
class SystemFileService implements FileService {

    @Override
    public void getFiles() {
        // Load temp files
    }

    @Override
    public void deleteFiles() {
        throw new UnsupportedOperationException();
    }
}
```

Những vi phạm về nguyên lý LSP

Một số dấu hiệu điển hình có thể chỉ ra rằng LSP đã bị vi phạm:

- Các lớp dẫn xuất có các phương thức ghi đè phương thức của lớp cha nhưng với chức năng hoàn toàn khác.
- Các lớp dẫn xuất có phương thức ghi đè phương thức của lớp cha là một phương thức rỗng.
- Các phương thức bắt buộc kế thừa từ lớp cha ở lớp dẫn xuất nhưng không được sử dụng.
- Phát sinh ngoại lệ trong phương thức của lớp dẫn xuất.

Lưu ý

Đây là nguyên lý… dễ bị vi phạm nhất, nguyên nhân chủ yếu là do sự thiếu kinh nghiệm khi thiết kế class. Thuông thường, design các class dựa theo đời thật: hình vuông là hình chữ nhật, file nào cũng là file. Tuy nhiên, không thể bê nguyên văn mối quan hệ này vào code. Hãy nhớ 1 điều:

- Trong thực tế, A là B (hình vuông là hình chữ nhật) không có nghĩa là class A nên kế thừa class B. Chỉ cho class A kế thừa class B khi class A thay thế được cho class B.
- File hệ thống cũng là file nhưng không thay thế được cho file, do đó ví dụ này vi phạm LSP.

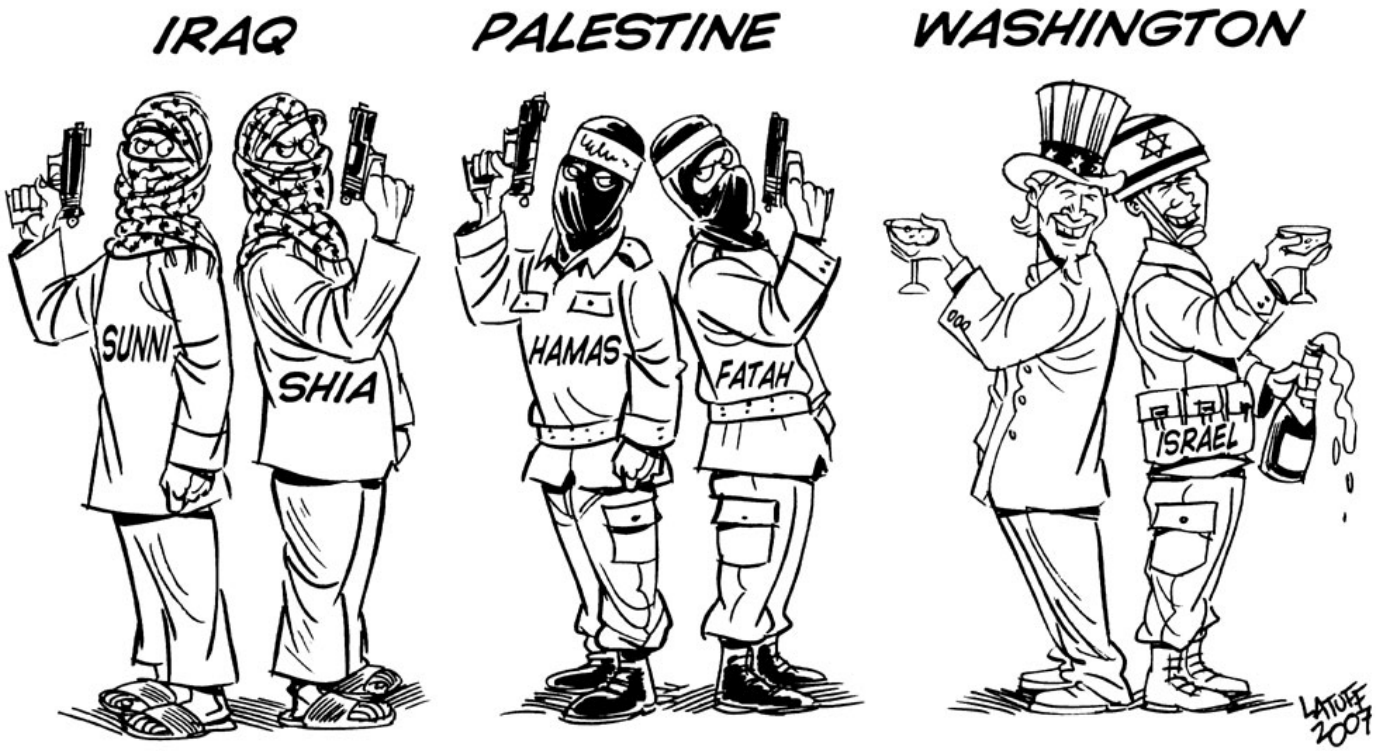
Nguyên lý này ẩn giấu trong hầu hết mọi đoạn code, giúp cho code linh hoạt và ổn định mà ta không hề hay biết. Ví dụ như trong Java, ta có thể chạy hàm foreach với List, ArrayList, LinkedList bởi vì chúng cùng kế thừa interface Iterable. Các class List, ArrayList, … đã được thiết kế đúng LSP, chúng có thể thay thế cho Iterable mà không làm hỏng tính đúng đắn của chương trình.

Interface segregation principle (ISP) – Nguyên lý phân tách

Nguyên tắc này được phát biểu như sau:

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

Many client-specific interfaces are better than one general-purpose interface.



Nguyên lý này khá dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn, khoảng 100 methods. Việc implements sẽ khá cực khổ, 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:

```
1  package com.gpcoder.solid;
2
3  /**
4   * ISP – Interface segregation principle example
5   *
6   * @author gpcoder
7   */
8  interface Repository<T, ID> {
9
10     Iterable<T> findAll();
11
12     T findOne(Long id);
13
14     T save(T entity);
15
16     void update(T entity);
17
18     void delete(T entity);
19
```

```
    Page<T> findAll(Pageable pageable);

    Iterable<T> findAll(Sort sort);
}
```

Như bạn thấy interface **Repository**, class này bao gồm các rất nhiều phương thức: lấy danh sách, lấy theo id, insert, update, delete, lấy danh sách có phân trang, sắp xếp, … Việc implement tất cả các phương thức này hết sức cực khổ, đôi khi không cần thiết do chúng ta không sử dụng hết.

Thay vào đó chúng ta có thể rách ra như sau:

```
interface CrudRepository<T, ID> {

    Iterable<T> findAll();

    T findOne(Long id);

    T save(T entity);

    void update(T entity);

    void delete(T entity);

}

interface PagingAndSortingRepository<T, ID> extends CrudReposi

    Page<T> findAll(Pageable pageable);

    Iterable<T> findAll(Sort sort);
}
```

Đối với một số chức năng đặc biệt chúng ta mới cần implement từ interface **PagingAndSortingRepository**, với chức năng thông thường chỉ cần **CrudRepository** là đủ.

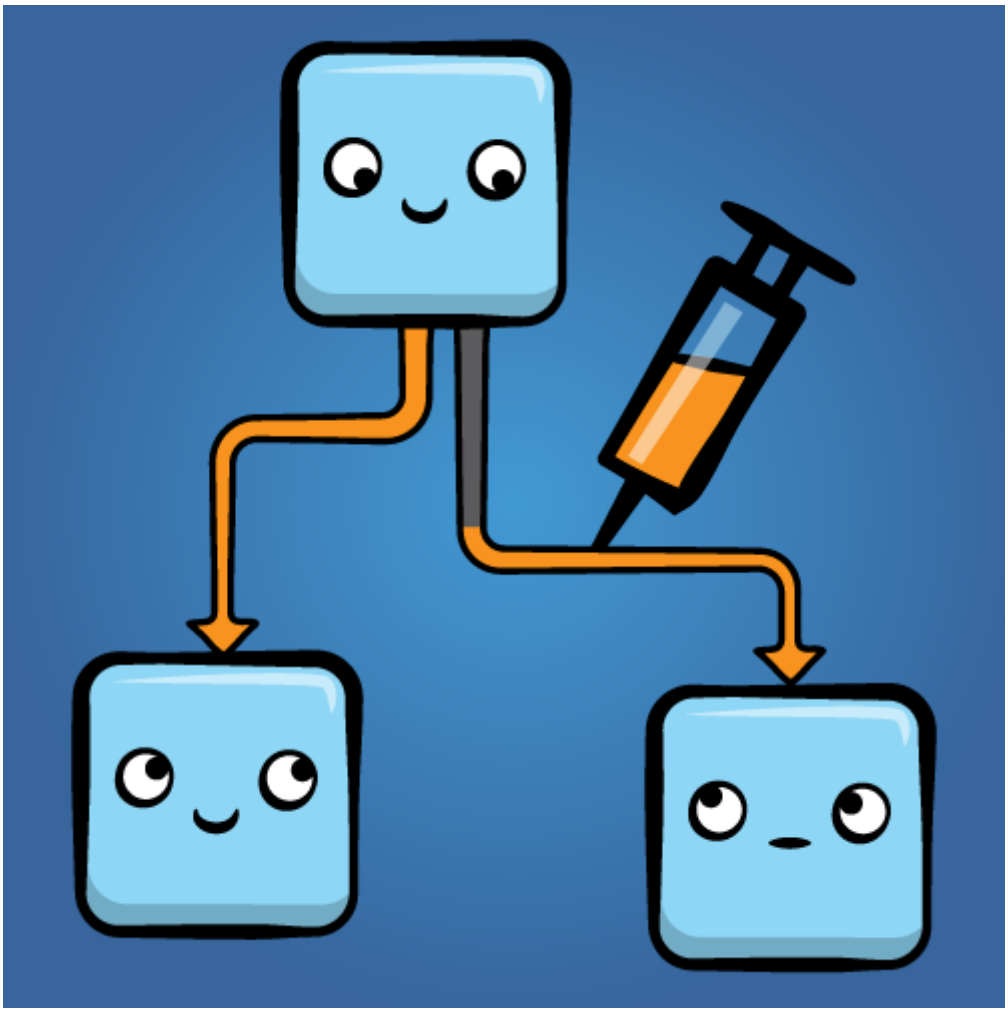
Một vi phạm khác cũng thường gặp trong dự án là class **CommonUtil**. Lớp này bao gồm rất nhiều phương thức: xử lý ngày giờ, số, chuỗi, chuyển đổi định dạng JSON, … tất cả mọi thức liên quan đến xử lý common đều được đặt vào đây. Càng ngày số lượng các phương thức càng lớn, khi đó sẽ phát sinh rất nhiều vấn đề như: trùng code, nhiều phương thức giống nhau không biết sử dụng phương thức nào, khi phát sinh bug rất khó bảo trì. Đối với những trường hợp này chúng ta nên chia nhỏ theo chức năng của nó, ví dụ như: **DateTimeUtil**, **StringUtil**, **NumberUtil**, **JsonUtil**, **ReflectionUtil**, … như vậy sẽ dễ dàng quản lí và sử dụng hơn.

Dependency Inversion Principle (DIP) – Nguyên lý đảo ngược phụ thuộc

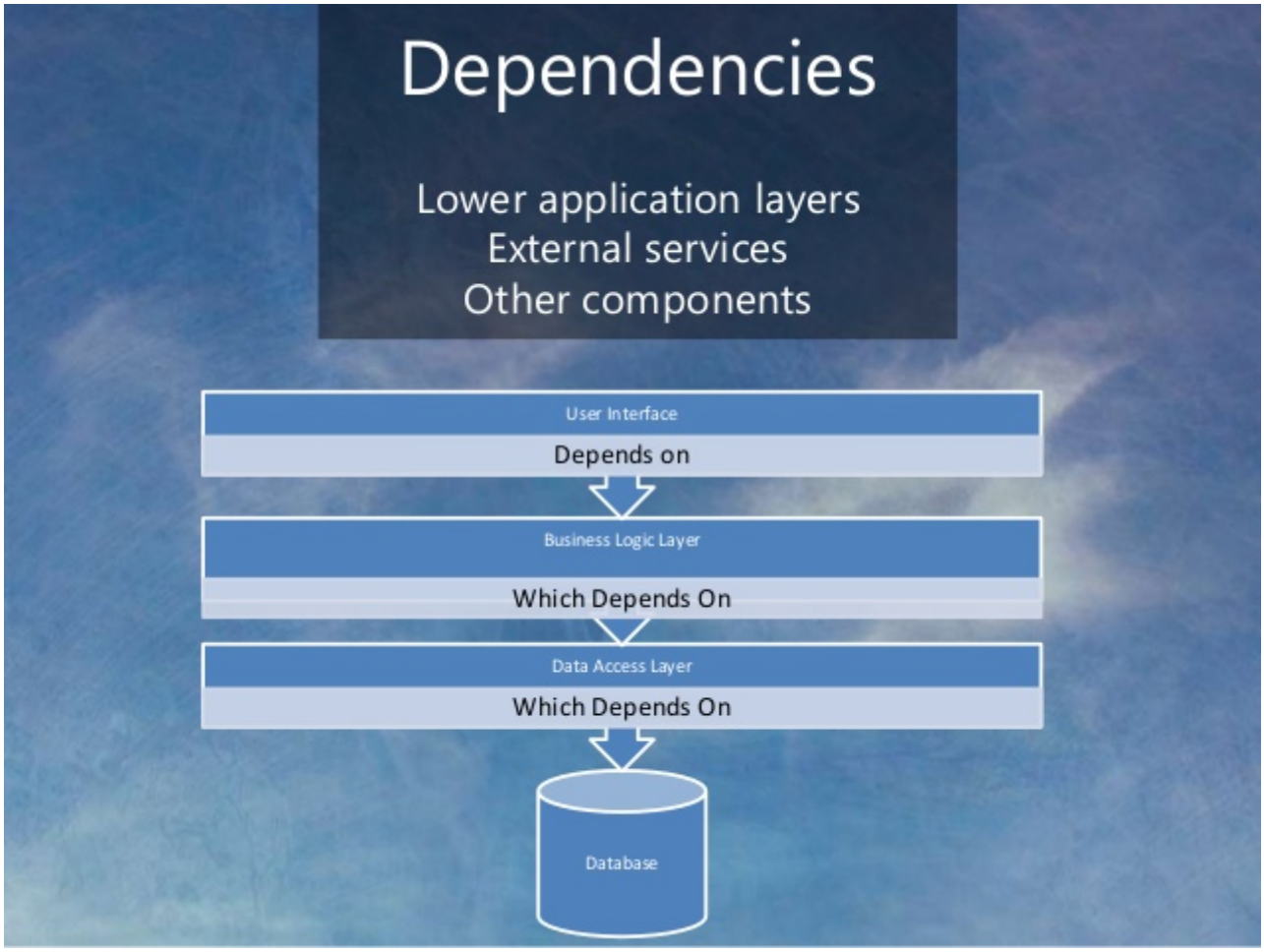
Nguyên tắc này được phát biểu như sau:

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, không phải thông qua implementation.)

1. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
2. *Abstractions should not depend upon details. Details should depend upon abstractions.*



Với cách code thông thường, các module cấp cao sẽ gọi các module cấp thấp. Module cấp cao sẽ phụ thuộc và module cấp thấp, điều đó tạo ra các **dependency**. Khi module cấp thấp thay đổi, module cấp cao phải thay đổi theo. Một thay đổi sẽ kéo theo hàng loạt thay đổi, giảm khả năng bảo trì của code.



Nếu tuân theo **Dependendy Inversion principle**, các module sẽ cùng phụ thuộc vào một interface không đổi. Nghĩa là thay vì để các module cấp cao sử dụng các interface do các module cấp thấp định nghĩa và thực thi, thì nguyên lý này chỉ ra rằng các lớp module cấp cao sẽ định nghĩa ra các interface, sau đó các lớp module cấp sẽ thực thi các interface đó. Khi đó, ta có thể dễ dàng thay thế, sửa đổi module cấp thấp mà không ảnh hưởng gì tới module cấp cao.

Ví dụ:

```
1  package com.gpcoder.solid;
2
3  /**
4   * DIP – Dependency inversion principle example
5   *
6   * @author gpcoder
7   */
8  interface DBConnection {
9      void connect();
10 }
11
12 class OracleConnection implements DBConnection {
13     @Override
14     public void connect() {
15         System.out.println("Oracle connected");
16     }
17 }
18
19 class MySQLConnection implements DBConnection {
20     @Override
21     public void connect() {
```

```
            System.out.println("MySQL connected");
        }
    }

    class PostgreSQLConnection implements DBConnection {
        @Override
        public void connect() {
            System.out.println("PostgreSQL connected");
        }
    }

    class DatabaseConfig {
        private DBConnection dbConnection;

        public DatabaseConfig(DBConnection dbConnection) {
            this.dbConnection = dbConnection;
            this.dbConnection.connect();
        }

        public DBConnection getConnection() {
            return this.dbConnection;
        }
    }

    public class DIPExample {

        public static void main(String[] args) {
            DBConnection conn = new OracleConnection();
            DatabaseConfig config = new DatabaseConfig(conn);
        }
    }
```

Như bạn thấy, các module của chúng ta không phụ thuộc vào nhau.Việc thay đổi code của một module này không ảnh hưởng đến các module còn lại. Nếu muốn hỗ trợ thêm SQLServer chỉ việc tạo thêm class mới, implement từ DbConnection. Nếu muốn đổi kết nối sang Oracle chỉ việc thay đổi trong config, …

Lời kết

Khi phát triển bất kỳ phần mềm nào, có hai khái niệm rất quan trọng cần nắm: sự gắn kết (**cohesion** – khi các phần khác nhau của một hệ thống sẽ làm việc cùng nhau để có kết quả tốt hơn nếu mỗi phần sẽ hoạt động riêng lẻ) và ghép nối (**coupling** – có thể được xem là mức độ phụ thuộc của một lớp, phương thức hoặc bất kỳ thực thể phần mềm nào khác).

Có rất nhiều nguyên tắc trong thiết kế phần mềm và trên đây là 5 nguyên tắc thiết kế hướng đối tượng (SOLID) quan trọng nhất mà bất cứ lập trình viên nào cũng phải nắm vững nếu như muốn cải thiện kĩ năng code của mình. Việc nắm vững và vận dụng được các nguyên lý trong thiết kế sẽ giúp cho mã nguồn chương trình của chúng ta nhìn rõ ràng hơn, tận dụng được các ưu điểm của OOP, các thành phần không bị phụ thuộc quá nhiều vào nhau, để thuận tiện cho việc bảo trì và mở rộng sau này.

Tuy nói nhiều lợi ích như vậy, nhưng các nguyên lý này chỉ mới chỉ ra cho chúng ta biết: thiết kế nào là đúng, thiết kế nào là sai chứ chưa giúp chúng ta giải quyết được vấn đề. Trong khi làm thực tế, gặp phải những vấn đề cụ thể như: làm thế nào để giảm thiểu số lượng các đối tượng phải tạo ra trong chương trình, làm thế nào để tích hợp một module có sẵn vào trong hệ thống của mình, … Tất cả những chuyện như vậy sẽ được giải quyết bằng cách áp dụng các "Mẫu thiết kế" (**Design Pattern**). Nói đơn giản hơn, các mẫu thiết kế sẽ giúp chúng ta giải quyết những bài toán thường gặp trong những ngữ cảnh nhất định. Các mẫu thiết kế cũng tuân theo các nguyên lý thiết kế hướng đối tượng làm cơ sở. Vì vậy việc nắm

vững các nguyên lý này là điều cần thiết nếu các bạn muốn tiến sâu hơn trong việc tạo ra 1 sản phẩm có kiến trúc đẹp, chất lượng.

Tài liệu tham khảo:

- https://toidicodedao.com/2015/03/24/solid-la-gi-ap-dung-cac-nguyen-ly-solid-de-tro-thanh-lap-trinh-vien-code-cung/
- https://toidicodedao.com/2015/11/03/dependency-injection-va-inversion-of-control-phan-1-dinh-nghia/
- https://edwardthienhoang.wordpress.com/2013/11/09/cac-nguyen-ly-thiet-ke-huong-doi-tuong/
- https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/
- https://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inve
- https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp
- https://www.netguru.co/codestories/topic/solid
- http://codebuild.blogspot.com/2010/09/oop-solid-rules-interface-segregation.html