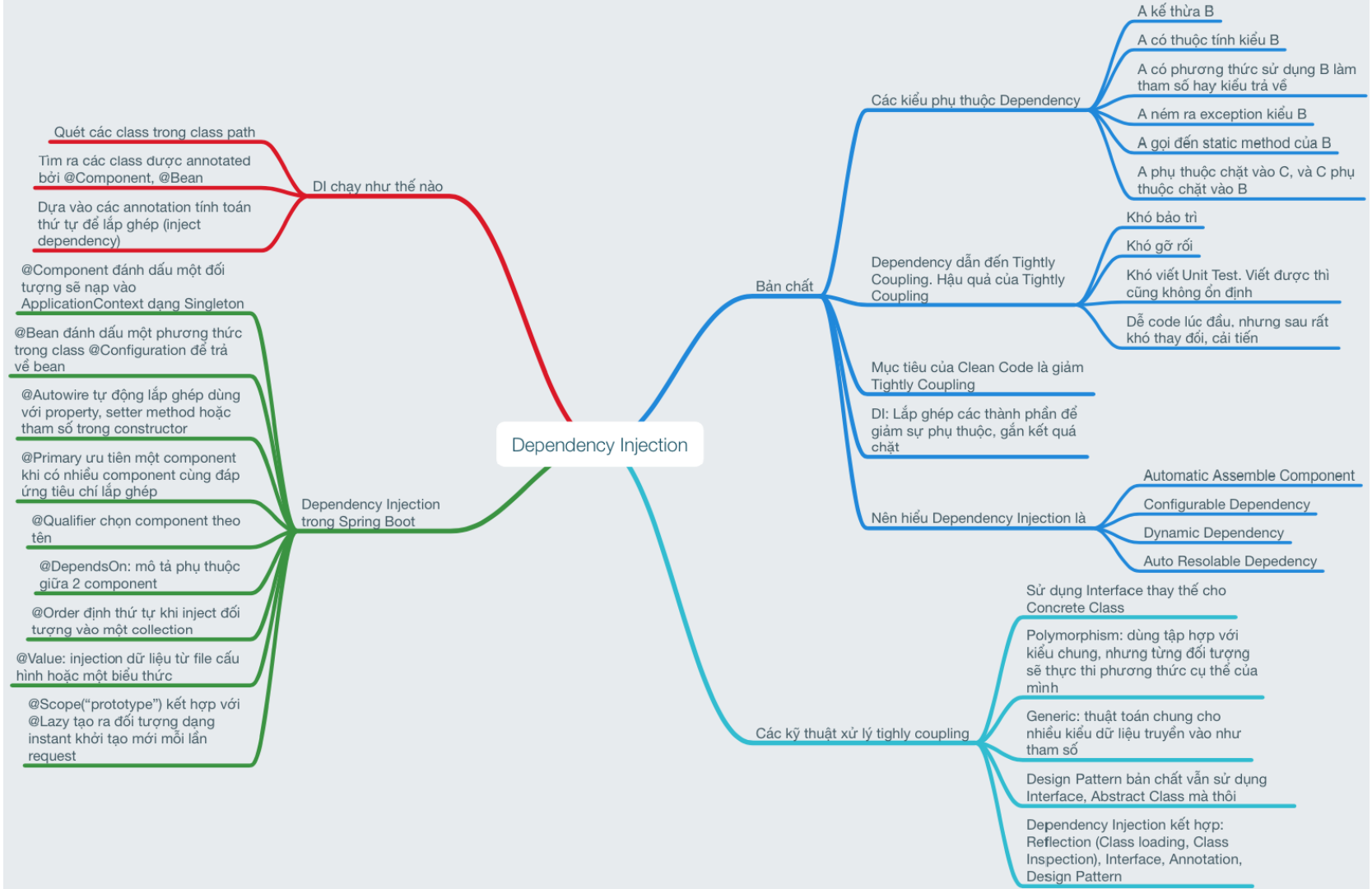
Dependency Injection Inversion of Control

cuong@techmaster.vn


Dependency Injection
Bản chất
Các kiểu phụ thuộc Dependency
A kế thừa B
A có thuộc tính kiểu B
A có phương thức sử dụng B làm tham số hay kiểu trả về
A ném ra exception kiểu B
A gọi đến static method của B
A phụ thuộc chặt vào C, và C phụ thuộc chặt vào B
Dependency dẫn đến Tightly Coupling. Hậu quả của Tightly Coupling
Khó bảo trì
Khó gỡ rối
Khó viết Unit Test. Viết được thì cũng không ổn định
Dễ code lúc đầu, nhưng sau rất khó thay đổi, cải tiến
Mục tiêu của Clean Code là giảm Tightly Coupling
DI: Lắp ghép các thành phần để giảm sự phụ thuộc, gắn kết quả chặt
Nên hiểu Dependency Injection là
Automatic Assemble Component
Configurable Dependency
Dynamic Dependency
Auto Resolable Depedency
Các kỹ thuật xử lý tighly coupling
Sử dụng Interface thay thế cho Concrete Class
Polymorphism: dùng tập hợp với kiểu chung, nhưng từng đối tượng sẽ thực thi phương thức cụ thể của mình
Generic: thuật toán chung cho nhiều kiểu dữ liệu truyền vào như tham số
Design Pattern bản chất vẫn sử dụng Interface, Abstract Class mà thôi
Dependency Injection kết hợp: Reflection (Class loading, Class Inspection), Interface, Annotation, Design Pattern
DI chạy như thế nào
Quét các class trong class path
Tìm ra các class được annotated bởi @Component, @Bean
Dựa vào các annotation tính toán thứ tự để lắp ghép (inject dependency)
Dependency Injection trong Spring Boot
@Component đánh dấu một đối tượng sẽ nạp vào ApplicationContext dạng Singleton
@Bean đánh dấu một phương thức trong class @Configuration để trả về bean
@Autowire tự động lắp ghép dùng với property, setter method hoặc tham số trong constructor
@Primary ưu tiên một component khi có nhiều component cùng đáp ứng tiêu chí lắp ghép
@Qualifier chọn component theo tên
@DependsOn: mô tả phụ thuộc giữa 2 component
@Order định thứ tự khi inject đối tượng vào một collection
@Value: injection dữ liệu từ file cấu hình hoặc một biểu thức
@Scope("prototype") kết hợp với @Lazy tạo ra đối tượng dạng instant khởi tạo mới mỗi lần request

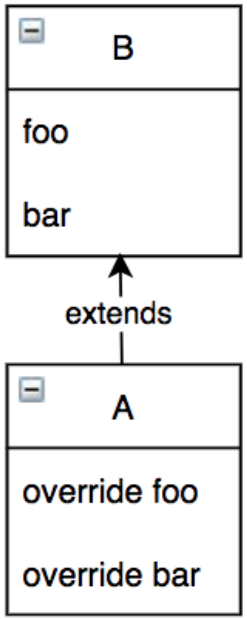
Dependency Injection ~ Tiêm sự phụ thuộc

- Dependency Injection có thể hiểu theo cách dễ hơn:
    - Configurable Dependency
    - Dynamic Dependency
    - Auto configurable Dependency
    - Resolvable Dependency
- Để các bạn không bị rối trí bởi từ ngữ khó hiểu, các bạn hãy hiểu Dependency Injection là **lắp ghép các thành phần lại**

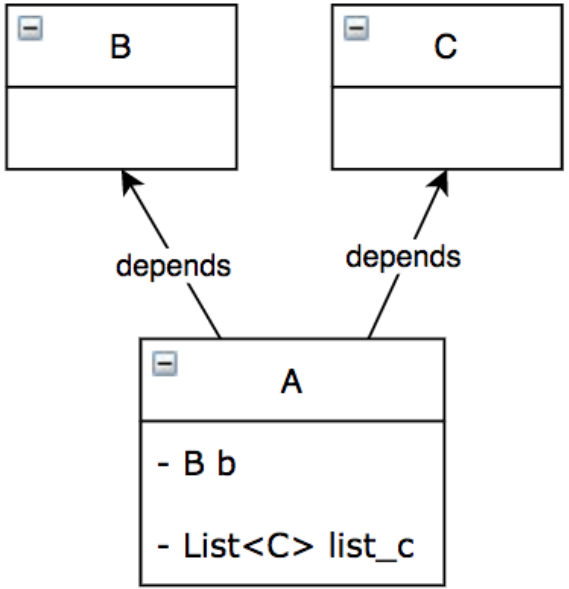
Class A “phụ thuộc” vào Class B khi

- Class A kế thừa Class B
- Class A chứa thuộc tính có kiểu Class B
- Phương thức của Class A có tham số truyền vào hoặc trả về là Class B
- Phương thức của Class A gọi đến static method của class B

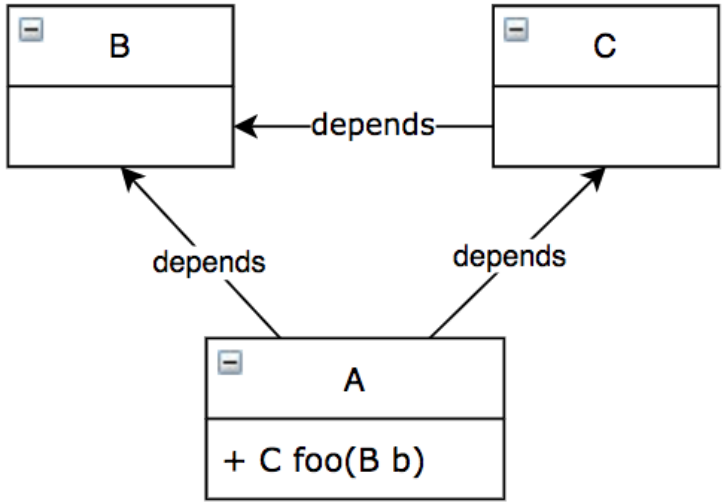
```
class B {
  public void foo() {

  }
  public void bar() {

  }
}
class A extends B {

  @Override
  public void bar() {
    super.bar();
  }

  @Override
  public void foo() {
    super.foo();
  }
}
```



```
class B {
}

class C {
}

class A {
  private B b;
  private List<C> list_c;
}
```

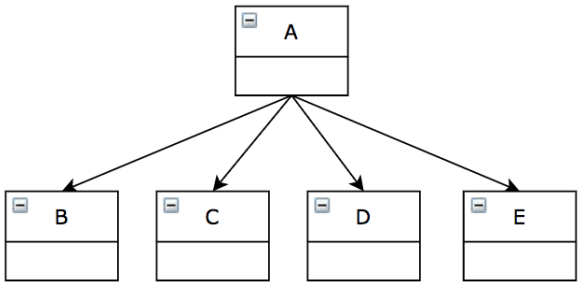


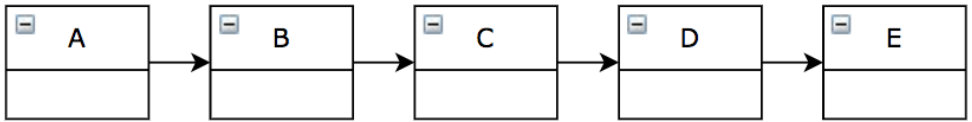
```
class B {
}
class C {
  private B b;
  public C(B b) {
    this.b = b;
  }
}
class A {
  public C foo(B b) {
    return new C(b);
  }
}
```



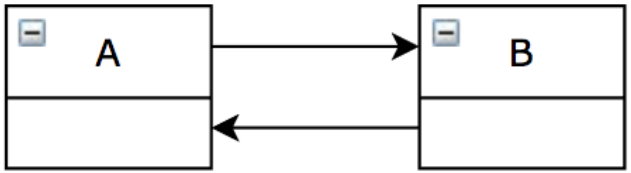
Tighly coupling – gắn quá chặt

- Class A phụ thuộc quá nhiều class B, C, E, F, X, Y, Z
- Class A phụ thuộc class B. Class B phụ thuộc Class C. C phụ thuộc D….
- Class A phụ thuộc class B. Ngược lại class B phụ thuộc class A. Circular reference.





```java
class B {
  public A a;
}

class A {
  public B b;
}

class App {
  public static void main(String[] args)
{
    A a = new A();
    B b = new B();
    a.b = b;
    b.a = a;
  }
}
```



Circular Reference

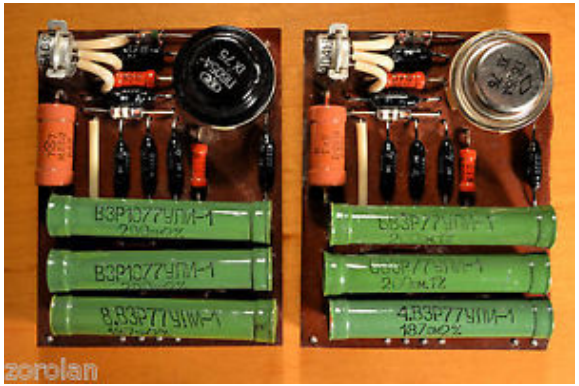
Khó khăn khi lập trình, kiểm thử, bảo trì

- Dependency gây ra Tightly Coupling
- Đặc điểm của Tightly Coupling là:
    - Dễ lập trình, khó kiểm thử, khó bảo trì, khó sửa lỗi

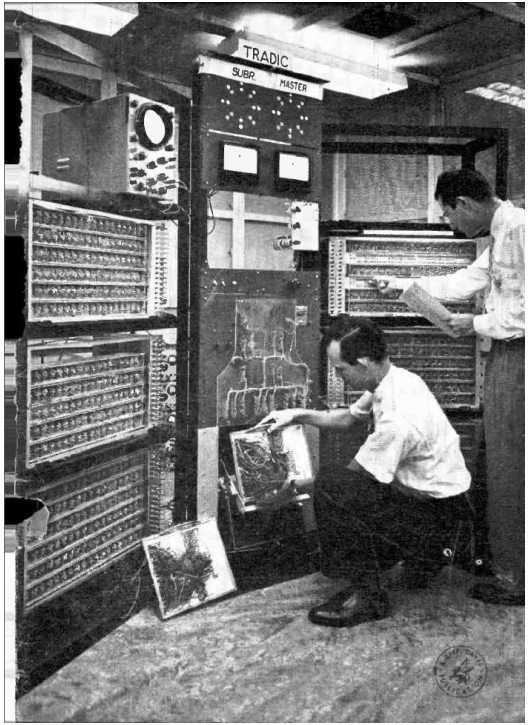
Các kỹ thuật giải quyết Tightly Coupling

- **Polymorphism** (đa hình) tập hợp các đối tượng chung kiểu gốc (base type) khi thực thi thì chạy phương thức cụ thể của đối tượng đó.
- **Interface** (giao diện) thay thế cho concrete class (lớp cụ thể)
- **Generic** (tổng quát) một method áp dụng cho nhiều kiểu dữ liệu khác nhau
- **Design Pattern**: factory, builder…
- **Dependency Injection** kết hợp Interface + Reflection + Design Pattern + Annotation

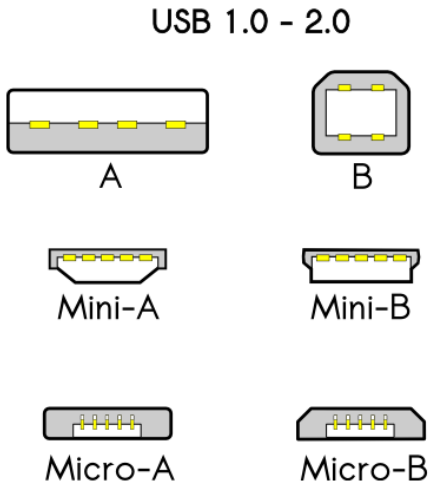
Ví dụ Dependency Injection qua chuẩn giao tiếp của máy tính

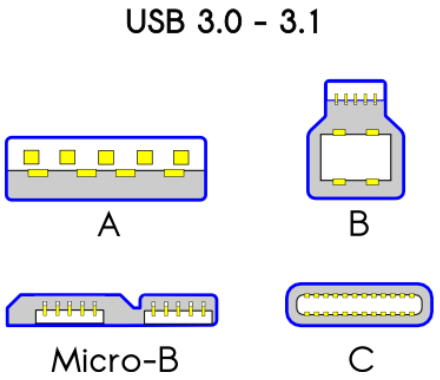


Code tất cả logic vào các phương thức trong 1 class duy nhất



Đóng gói chức năng thành các module có thể thay thế nhưng không lắp lẫn





Khuyến khích **biến thể đa dạng** miễn là tuân thủ **interface**

Giao tiếp giữa các thành phần máy tính

- Mọi thành phần máy tính giao tiếp với nhau qua interface.
- Các thành phần có thể tháo ra thay thế.
- Có thể lắp lẫn miễn tuân thủ interface: SATA, USB, DDR, CPU Socket
    - RAM có thể tăng từ 2G -> 4G ->6G ->8G ->12G -> 16G -> 32G
    - Card đồ hoạ có thể nâng cấp GPU

DI

Thời điểm Dependency Inject

- Lúc thiết kế ~ Design time
- Lúc biên dịch lắp ráp các thành phần ~ Compile time
- Lúc triển khai ~ Deploy time (deploy môi trường test khác deploy môi trường product)
- Lúc chạy ~ Run time

Dependency Injection trong Java dựa trên kỹ thuật nào?

- Reflection java.lang.reflect
- Class loader:
  - Quét tất cả các file *.class
  - Inspect class có chứa các annotation để từ đó thực hiện các logic
- Configuration:
  - Đọc cấu hình từ annotation, XML, hoặc thực thi code để lắp ráp các component

Trình tự thực hiện DI bên trong Spring Boot

1. Quét tất cả các class trong file *.class trong class
2. Chọn ra những class được đánh dấu bởi annotation @Component hoặc biến thể của @Component hoặc các kiểu trả về từ hàm đánh dấu bởi @Bean
3. Quét tiếp các annotation @Autowired, @DependensOn, @Primary, @Scope, @Lazy…để xây dựng thứ tự tạo đối tượng và lắp ghép (inject) chúng theo thứ tự hợp lý
4. Kết thúc quá trình DI với các singleton component
5. Quá trình tạo, lắp ghép đối tượng instant (non singleton) sẽ thực thi khi cần.

```
public static List<Class> componentScan() {
    List<Class> matchingClasses = new ArrayList<Class>();
    List<Class> classes = getAllKnownClasses();
    for (Class clazz : classes) {
      if (clazz.isAnnotationPresent(Component.class)) {
        matchingClasses.add(clazz);
      }
    }
    return matchingClasses;
}
```

DI trong Spring Boot

3 phương pháp DI chính

1. Property
2. Constructor
3. Setter

@Autowired, @Inject, @Resource

Là những annotation có chung một mục đích chính là lắp ghép tìm đúng đối tượng phù hợp vào đúng chỗ cần lắp. (Không thể lắp CPU AMD vào khe cắm Intel CPU)

Các thuộc tính chỉ khai báo kiểu interface chứ không phải class cụ thể.

@Autowire của Spring Boot

@Inject là annotation có từ Java EE

@Component khác gì @Bean?

- @Bean được tạo ra bằng phương thức trong class đánh dấu bởi @Configuration
- @Bean tuỳ biến logic khởi tạo tốt hơn @Component. Ngược lại khai báo @Component nhanh, ngắn gọn hơn.

```java
@Configuration
public class CarConfig {
  @Autowired
  private ApplicationContext context;

  @Value("${engineType}")
  private String engineType;

  @Bean
  public Car car() {
    Engine engine;
    switch (engineType) {
      case "gas":
        engine = (Engine) context.getBean("gasEngine");
        break;
      case "electric":
        engine = (Engine) context.getBean("electricEngine");
        break;
      case "hybrid":
        engine = (Engine) context.getBean("hybridEngine");
        break;
      default:
        engine = (Engine) context.getBean("gasEngine");
    }
    return new Car(engine);
  }
```

@Primary vs @Qualifier

Khi có nhiều hơn 1 Component cùng kiểu, Spring Boot sẽ không biết chọn Component nào để lắp ghép. Có 2 annotation giúp Spring Boot chọn component phù hợp

- @Primary đánh dấu tại Component cần ưu tiên
- @Qualifier chọn Component theo tên tại điểm lắp ghép. @Qualifier sẽ linh hoạt hơn khi ở mỗi điểm khác nhau, bạn cần chọn component khác nhau để lắp ghép.

@DependsOn xác định phụ thuộc Component A với Component B

```java
@Component
@DependsOn({"powersupply"})
public class Computer {
  @Autowired private PowerSupply psu;

  public Computer(@Qualifier("fujitsu") HardDisk hdd) {
    this.hdd = hdd;
  }
```

Khi có `@DependsOn({"powersupply"})` thì PowerSupply sẽ được khởi tạo trước sau đó mới đến Computer
Ngược lại nếu bỏ `@DependsOn` thì Computer khởi tạo trước rồi mới đến PowerSupply

Collection injection

```
@Autowired private List<USB> usbDevices2;  //Collection inject: tự động quét tất cả các đối tượng có kiểu USB nạp vào List
```

```
@Component("mouse")
@Order(2)
public class Mouse implements USB1{
  public Mouse() {
    System.out.println("Mouse");
  }
}
```

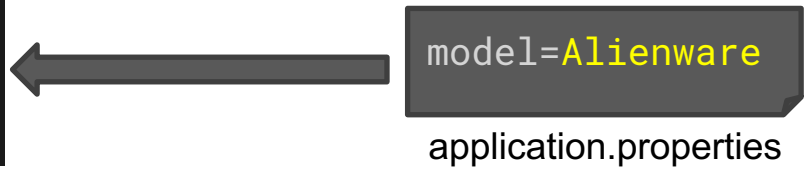
```
@Component("keyboard")
@Order(5)
public class Keyboard implements USB1 {
  public Keyboard() {
    System.out.println("Keyboard");
  }
}
```

```
@Component
@Order(3)
public class WebCam implements USB2 {
  public WebCam() {
    System.out.println("Web cam");
  }
}
```

@Order xác định thứ tự nạp vào

@Value đọc dữ liệu từ file cấu hình hoặc biểu thức giá trị

```java
@Component
@Configuration
public class Computer {
  @Value("${model}")
  private String model;
```



Tham khảo thêm
https://www.baeldung.com/spring-value-annotation

Bài tập thực hành

- Hãy tạo 2 profile trong application.properties: “dev” và “prod”
- Tạo một interface Booster và 2 class thể hiện Booster là DebugOverClock và FastOverClock
- Khi active profile là “dev” hãy inject DebugOverClock
- Khi active profile là “prod” hãy inject FastOverClock