Decoupling Components with the Dependency Inversion Principle (DIP)

Dan Geabunea

PASSIONATE SOFTWARE DEVELOPER | BLOGGER

@romaniancoder www.romaniancoder.com

Overview

What is the Dependency Inversion Principle?

Writing code that respects this principle

Dependency Injection (DI)

Inversion of Control (IoC)

Demo: Decoupling components and improving testability with DIP

Dependency Inversion Principle

1. High level modules should not depend on low level modules; both should depend on abstractions.

2. Abstractions should not depend on details. Details should depend upon abstraction.

High Level Modules

Modules written to solve real problems and use cases

They are more abstract and map to the business domain

What the software should do

Low Level Modules

Contain implementation details that are required to execute the business policies

They are considered the “plumbing” or “internals” of an application

How the software should do various tasks

Examples of Low Level Modules

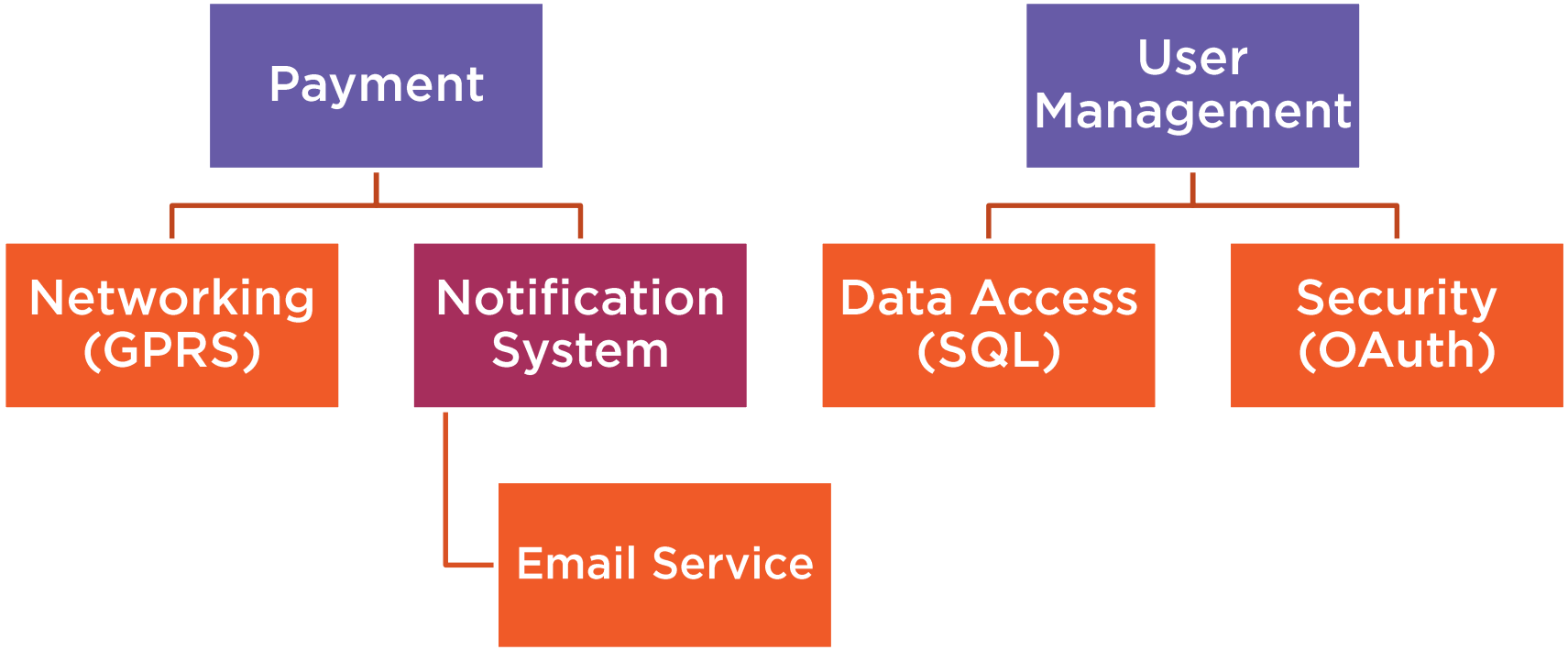
Logging

Data access

Network communication

IO

High Level Modules Work Together with Low Level Modules



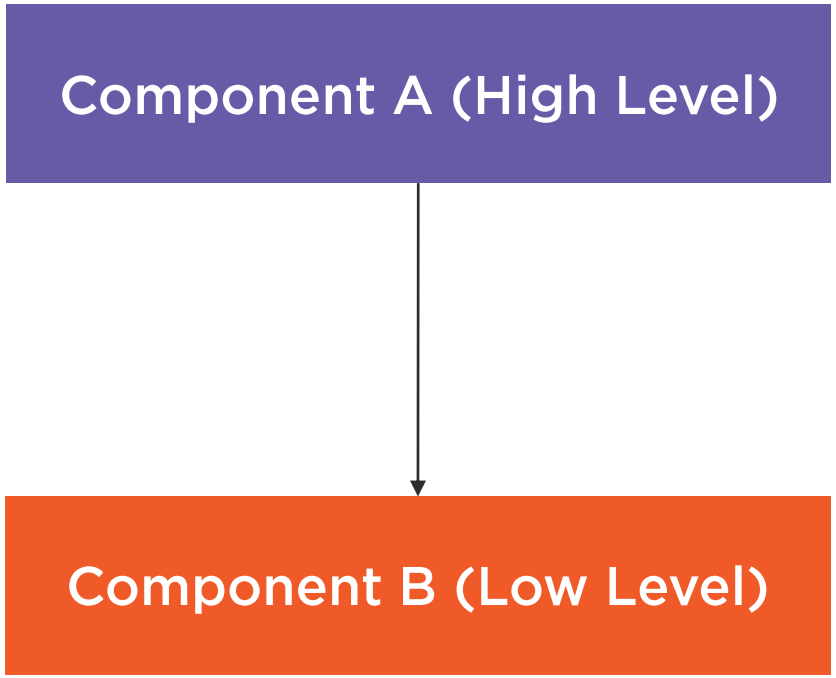
Abstraction

Something that is not concrete.

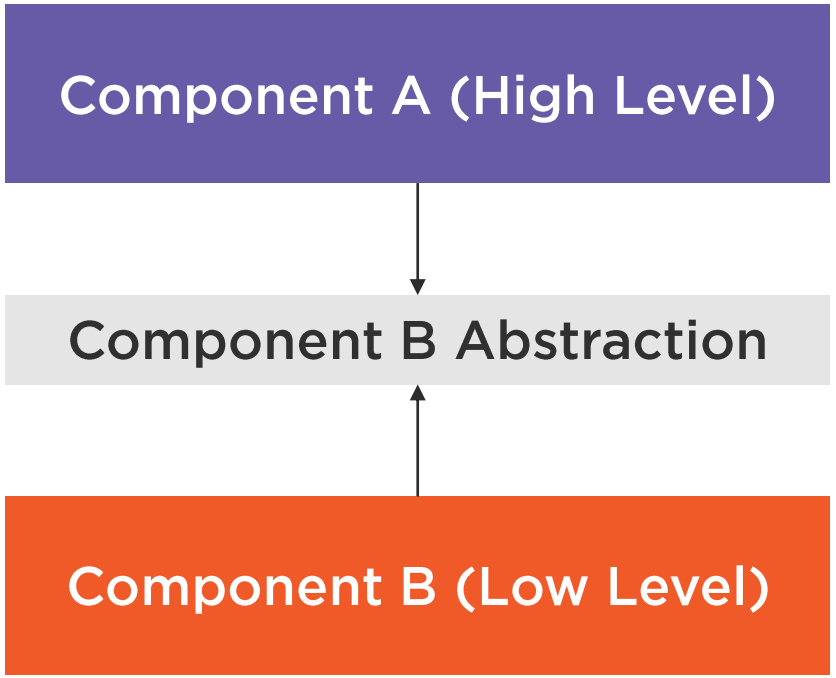
Something that you can not “new” up. In Java applications, we tend to model abstractions using interfaces and abstract classes.

Putting It All Together

Depend on details



Depend on abstractions



Writing Code That Respects the DIP

Low Level Class

```
class SqlProductRepo{
    public Product getById(String productId){
        // Grab product from SQL database
    }
}
```

High Level Class

```
class PaymentProcessor{
    public void pay(String productId){
        SqlProductRepo repo = new SqlProductRepo();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}
```

Product Repository Abstraction

```
interface ProductRepo {
    Product getById(String productId);
}
```

Low Level Class Depends on Abstraction

```
class SqlProductRepo implements ProductRepo{
    @Override
    public Product getById(String productId){
        // Concrete details for fetching a product
    }
}
```

High Level Class Depends on Abstraction

```
class PaymentProcessor{
    public void pay(String productId){
        ProductRepo repo = ProductRepoFactory.create();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}
```

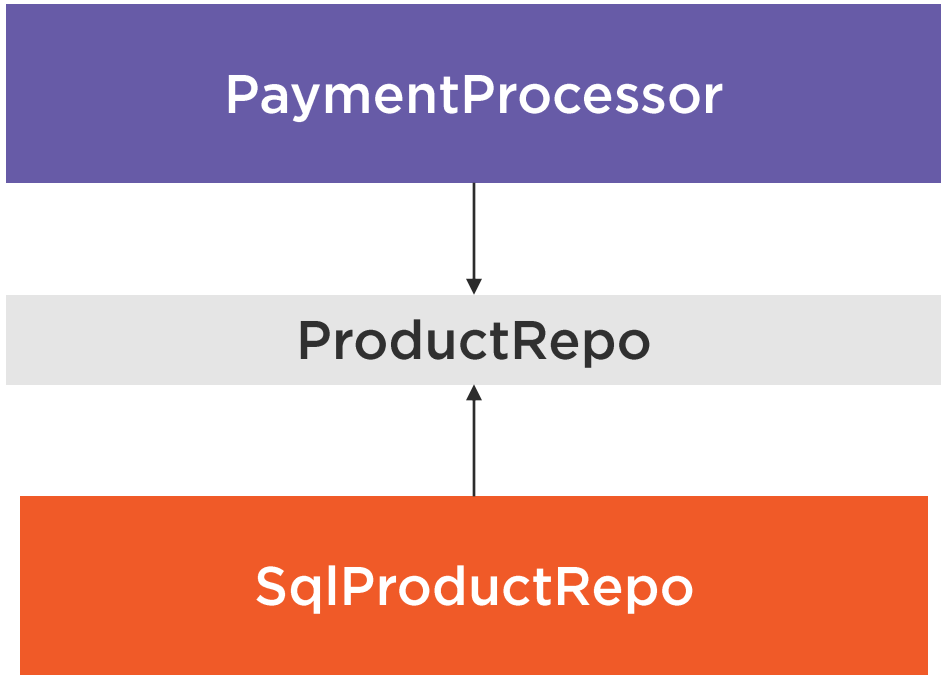
Product Repository Factory

```
class ProductRepoFactory{
    public static ProductRepo create(){
        return new SqlProductRepo();
    }
}
```

Product Repository Factory

```
class ProductRepoFactory{
    public static ProductRepo create(String type){
        if(type.equals(“mongo”)){
            return new MongoProductRepo();
        }
        return new SqlProductRepo();
    }
}
```

After Applying the DIP



Dependency Injection (DI)

Still Some Coupling

```
class PaymentProcessor{
    public void pay(String productId){
        ProductRepo repo = ProductRepoFactory.create();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}
```

Dependency Injection

A technique that allows the creation of dependent objects outside of a class and provides those objects to a class.

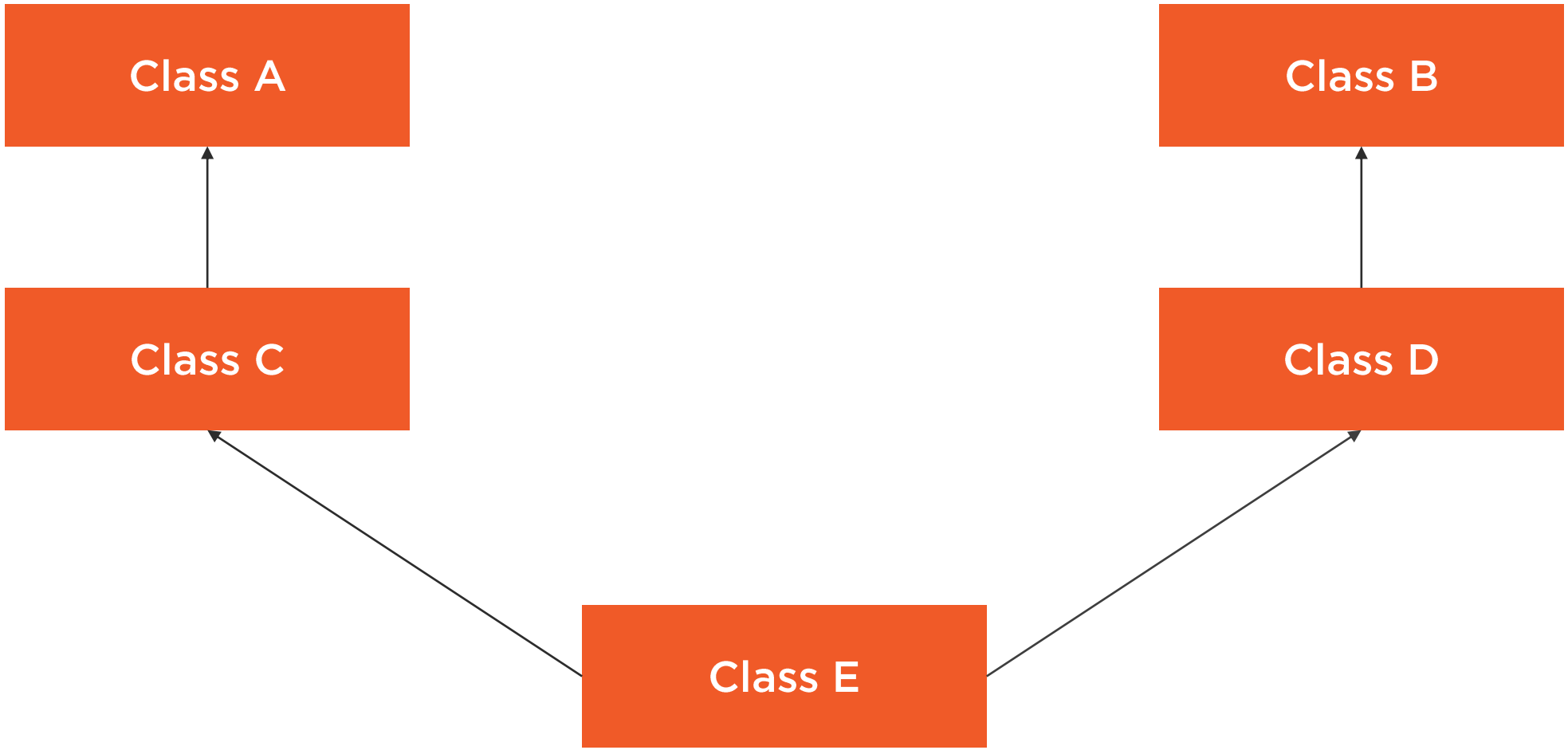
Declaring Dependencies in Constructor

```
class PaymentProcessor{
    public PaymentProcessor(ProductRepo repo){
        this.repo = repo;
    }
    public void pay(String productId){
        Product product = this.repo.getById(productId);
        this.processPayment(product);
    }
}
```

Passing Dependencies

```
ProductRepo repo = ProductRepoFactory.create();
PaymentProcessor paymentProc = new PaymentProcessor(repo);
paymentProc.pay("123");
```

More Complex Dependencies



Handling Dependencies Becomes Harder

```
A a = new A();
B b = new B();
C c = new C(a);
D d = new D(b);
E e = new E(c,d);

e.doSomething();
```

Inversion of Control (IoC)

Inversion of Control

Inversion of Control is a design principle in which the control of object creation, configuration, and lifecycle is passed to a container or framework.

You don’t “new” up objects

- They are created by someone else (IoC container)

Control of object creation is inverted

- It’s not the programmer, but someone else who controls the objects

It makes sense to use it for some objects in an application (services, data access, controllers)

- For others (entities, value objects) it doesn’t

IoC Container Benefits

Makes it easy to switch between different implementations at runtime

Increased program modularity

Manages the lifecycle of objects and their configuration

Spring Bean

Objects used by your application and that are managed by the Spring IoC container. They are created with the configuration that you supply to the container.

Spring Bean Definition Example

```
@Configuration
public class DependencyConfig {
    @Bean public A a(){return new A();}

    @Bean public B b(){return new B();}

    @Bean public C c(A a, B b){return new C(a(),b());}
}
```

Spring Bean Definition Example

```
public class C {
    private A a;
    private B b;
    public C(A a, B b){
        this.a = a;
        this.b = b;
    }
}
```

DIP, DI and IoC

Dependency Inversion Principle

Dependency Injection

Inversion of Control & Containers

Demo

Applying the DIP

- Decoupling components
- Improving testability

Summary

Classes should depend on abstractions, not implementation details

DIP, DI and IoC work hand in hand to eliminate coupling and make applications less brittle

Testability can greatly be improved by using the DIP and the DI technique

Take advantage of the powerful capabilities of the Spring IoC container

“New is glue.”

Steve Smith (Ardalis), Pluralsight Author

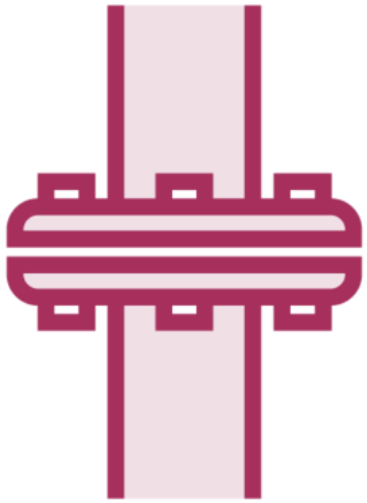
The DIP, DI and IoC are the most effective ways to eliminate code coupling and keep systems easy to maintain and evolve

Course Summary

Course Key Takeaways

Technical Debt

Silent killer of software projects



Coupling

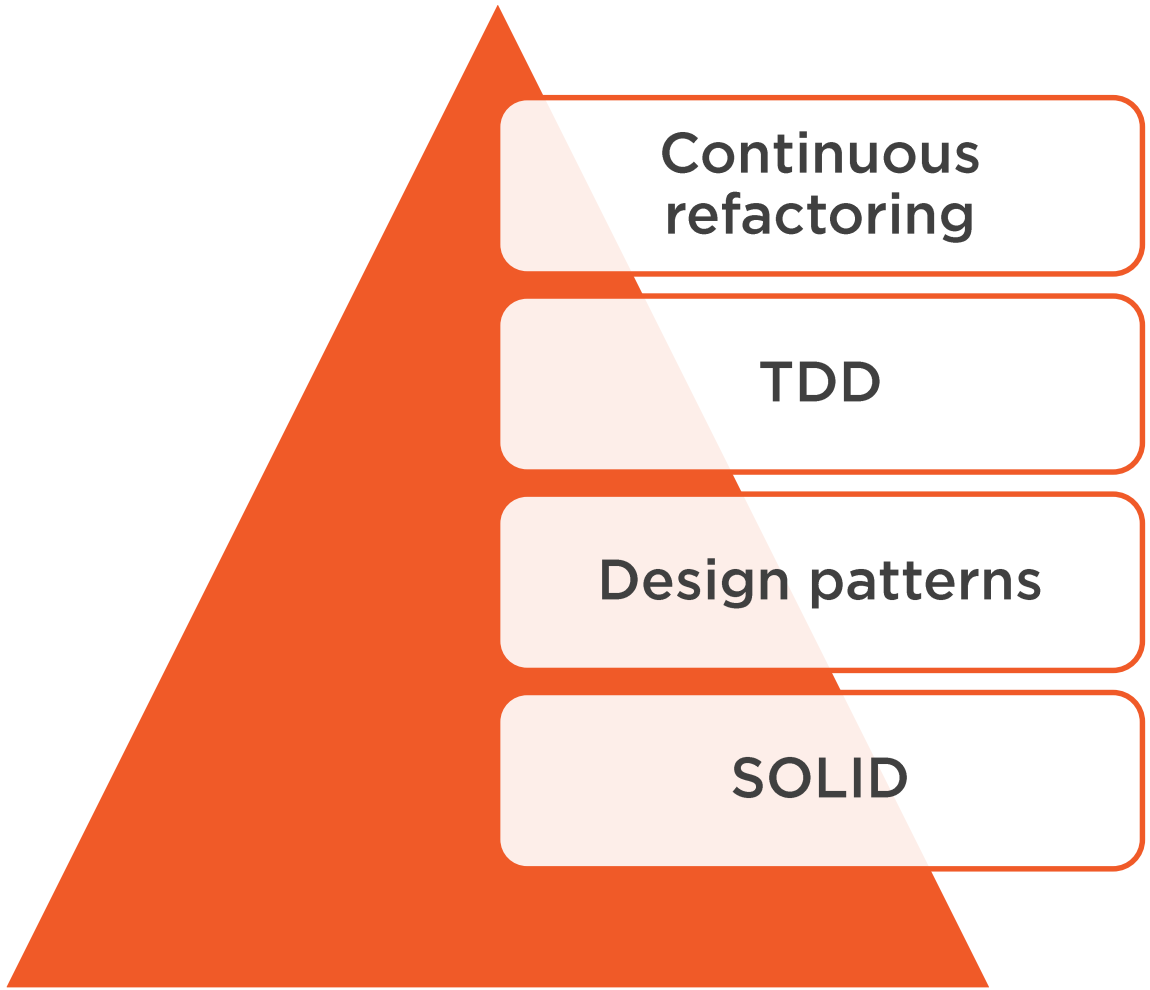
Main reason of technical debt

SOLID

Tackle coupling and promote designs that can evolve and grow

The SOLID principles of
OO design are the foundation
of clean system design.

Pyramid of Clean Code



Where to Go from Here?

Design Patterns in Java

Bryan Hansen

Test Driven Development Practices in Java

Mike Nolan

Code Samples

https://github.com/dangeabunea/pluralsight-solid-principles-java