


****You:****

Give me an example to understand about tight coupling and loose coupling in springboot? the example should include GameRunner, MarioGame, and SuperContraGame Please write in simple English language.

****ChatGPT:****

Sure! Let's understand tight coupling and loose coupling using the example of a game application in Spring Boot. In this example, we have three classes: GameRunner, MarioGame, and SuperContraGame.

1. Tight Coupling: In a tightly coupled design, classes are highly dependent on each other. Any changes in one class may require modifications in other classes. Let's see an example:


```
java  Copy code
public class GameRunner {
    private MarioGame game;

    public GameRunner() {
        game = new MarioGame();
    }

    public void playGame() {
        game.start();
    }
}
```

In this example, the GameRunner class directly creates an instance of the MarioGame class using the "new" keyword. This creates a tight coupling between GameRunner and MarioGame. If we want to change the game to SuperContraGame, we would need to modify the GameRunner class, which violates the principle of loose coupling.

2. Loose Coupling: In a loosely coupled design, classes have minimal dependencies on each other, promoting flexibility and easier maintenance. Let's modify the previous example to demonstrate loose coupling:

```
java  Copy code
public interface Game {
    void start();
}

public class GameRunner {
    private Game game;

    public GameRunner(Game game) {
        this.game = game;
    }

    public void playGame() {
        game.start();
    }
}

public class MarioGame implements Game {
```

```
    public void start() {  
        // Code to start Mario game  
    }  
}  
  
public class SuperContraGame implements Game {  
    public void start() {  
        // Code to start Super Contra game  
    }  
}
```

In this example, we introduce an interface called Game, which declares the common behavior of starting a game. Both MarioGame and SuperContraGame implement this interface. The GameRunner class now accepts any class that implements the Game interface, allowing us to easily switch between different games without modifying the GameRunner class. This demonstrates loose coupling, as the GameRunner class is no longer directly tied to a specific game implementation.

By using loose coupling, we can easily extend our application with new games by creating new classes that implement the Game interface. It promotes modularity, reusability, and easier maintenance of the codebase.
