




Dependency Injection & Inversion of Control



Dependency

Relación entre clases

```
class UserService {  
    private val userRepository: IUserRepository  
  
    fun save(user: User) : User {  
        return this.userRepository.save(user)  
    }  
  
    fun findById(id: Long) : User? {  
        return this.userRepository.findById(id)  
    }  
}
```



Dependencia



SOLID

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

<https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898>




Dependency Inversion Principle (DIP)

“Depend on abstractions, not on concretions.”

```
class UserService {  
    private val userRepository: IUserRepository  
  
    fun save(user: User) : User {  
        return this.userRepository.save(user)  
    }  
  
    fun findById(id: Long) : User? {  
        return this.userRepository.findById(id)  
    }  
}
```

La dependencia es una Interfaz



Dependency Injection (DI)

Delegar la construcción de las dependencias


```
class UserService {  
    private val userRepository: IUserRepository  
  
    constructor(userRepository: IUserRepository) {  
        this.userRepository = userRepository  
    }  
  
    fun save(user: User) : User {  
        return this.userRepository.save(user)  
    }  
  
    fun findById(id: Long) : User? {  
        return this.userRepository.findById(id)  
    }  
}
```



La inicialización de la dependencia se delega al constructor



Beneficios

- Desacoplamiento de código
 - Mayor facilidad para testing
 - Permite realizar test doubles
 - Menor impacto a la hora de modificar código de dependencias
 - Código mas reutilizable
 - Código más legible
- 




Inversion of Control (IoC)

Delegar el control de un proceso determinado a una herramienta externa (librería, framework, etc)

Container

Encargado de inyectar las dependencias de las distintas clases



Funcionamiento de un container

```
object Container {  
    private val dependencies: List<String> = listOf()  
  
    fun resolve(dependency: String) {  
        if (!dependencies.contains(dependency)) {  
            throw Exception("Cannot resolve $dependency")  
        }  
  
        when (dependency) {  
            "userService" -> UserService(UserRepository())  
            "userController" -> UserController(resolve("userService"))  
            else -> throw Exception("Cannot resolve $dependency")  
        }  
    }  
  
    fun register(dependency: String,) {  
        this.dependencies.add(dependency)  
    }  
}
```

Disclaimer:

El código de un container real no es tan sencillo

Uso del container

```
class Application{  
    fun main() {  
        Container.register("userController", UserController.class)  
        run()  
    }  
}
```

```
class Api {  
    private val userController: IUserController  
    constructor() {  
        this.userController = Container.resolve("userController")  
    }  
  
    @Get("/")  
    fun findById(id: Long) {  
        return this.userController.findById(id)  
    }  
}
```

Así te mira Uncle Bob cuando instancias las dependencias dentro de la clase



*Thank
you!*