



# SOLID

Fullstack Bootcamp

# Contents

**01**

Intro

**02**

Single Responsibility

**03**

Open/closed principle

**04**

Liskov Substitution

**05**

Interface Segregation

**06**

Dependency inversion



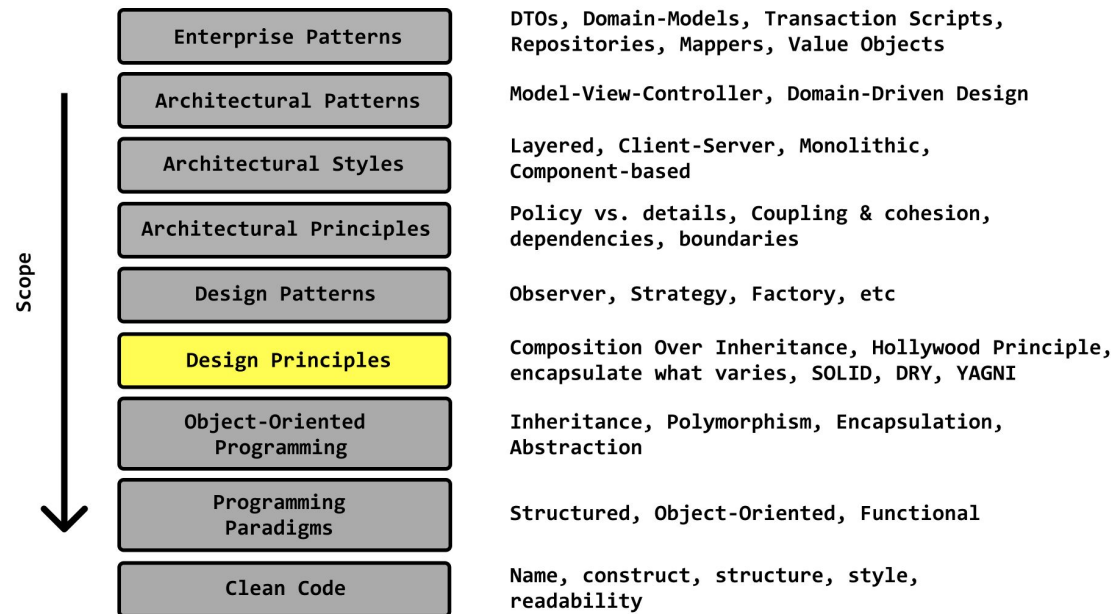


# Intro



## The Software Design & Architecture Stack

Khalil Stemmler  
@stemmlerjs



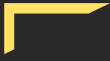
# Intro

SOLID is an acronym for 5 important design principles when doing OOP (Object Oriented Programming).

These five software development principles are guidelines to follow when building software so that it is easier to scale and maintain. They were made popular by a software engineer, Robert C. Martin.

The intention of these principles is to make software designs more understandable, easier to maintain and easier to extend.





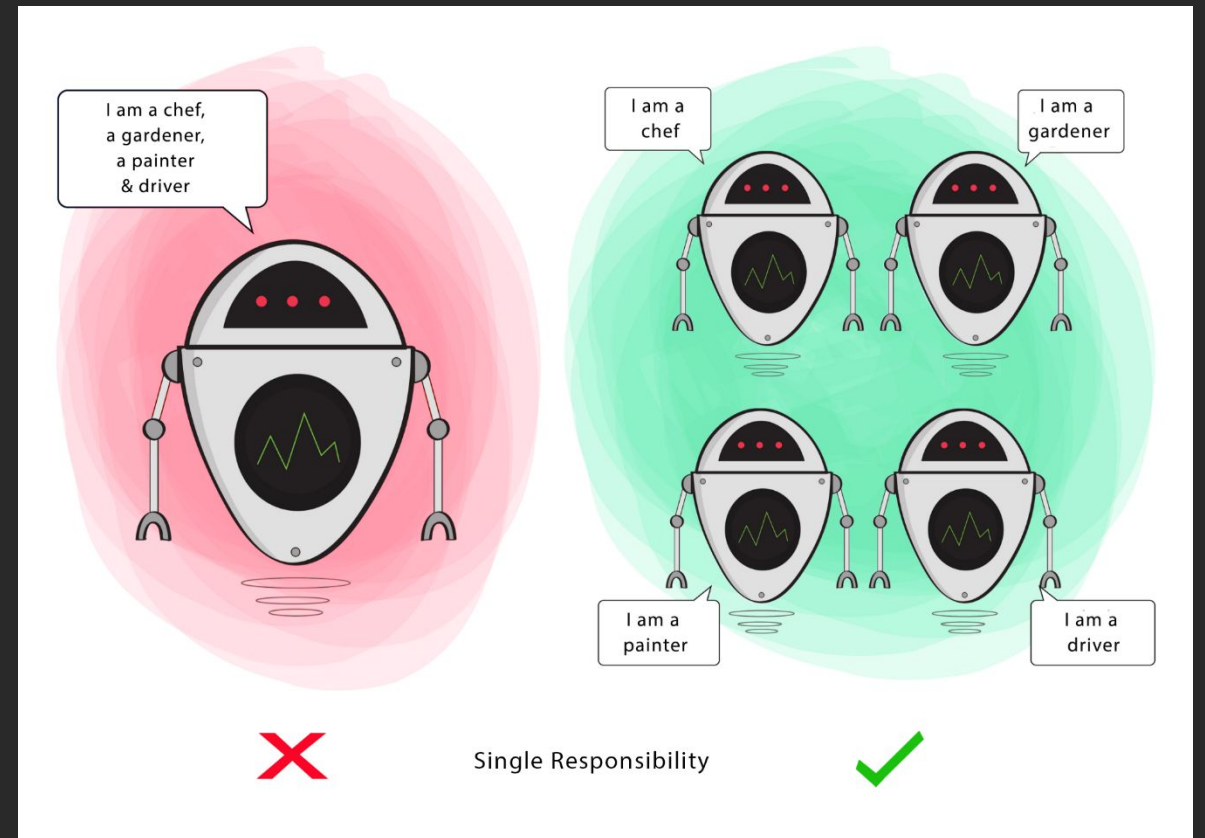
# S – Single Responsibility

# S – Single Responsibility

In programming, the Single Responsibility Principle states that every module or class should have responsibility over a single part of the functionality provided by the software.

**A class should have a single responsibility.**

If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities, could affect the other ones without you knowing.



# S - Single Responsibility

```
1 class UserService {
2   private userModel;
3   private commentModel;
4
5   getUser() {
6     return this.userModel.findOne();
7   }
8
9   getComment() {
10    return this.commentModel.findOne();
11  }
12
13  async getUserComments(userId: string) {
14    const user = await this.userModel.findOne({ id: userId });
15    const comments = await this.commentModel.find({ id: { $in: user.comments } });
16
17    return comments;
18  }
19 }
```





# S - Single Responsibility

```
1 class UserService {
2   private userRepo: UserRepo;
3   private commentRepo: CommentRepo;
4
5   async getUser(userId: string) {
6     const user = await this.userRepo.getUser(userId);
7     const comments = await this.commentRepo.getComments(user.comments);
8
9     return {
10      ...user,
11      comments
12    }
13  }
14 }
```

```
24 class CommentRepo {
25   private commentModel;
26
27   getComments(commentIds: string[]) {
28     return this.commentModel.find({ id: { $in: commentIds } });
29   }
30 }
```

```
16 class UserRepo {
17   private userModel;
18
19   getUser(id: string) {
20     return this.userModel.findOne({ id });
21   }
22 }
```





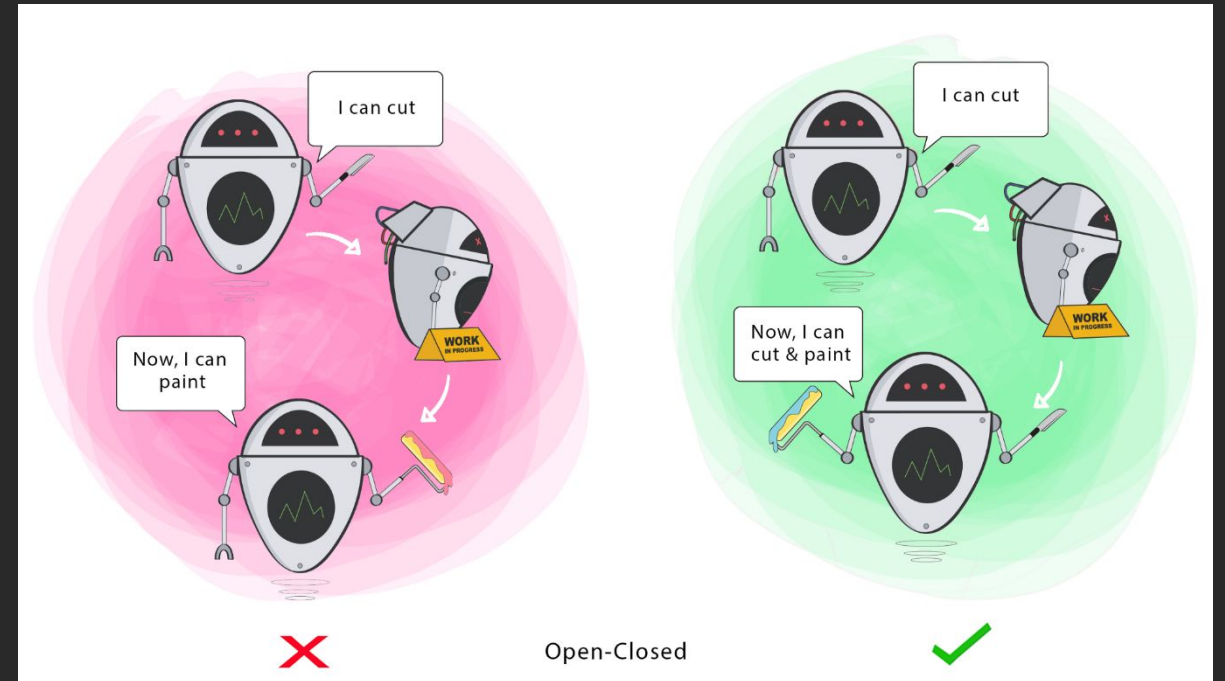
# O - Open/closed principle

# O - Open/closed principle

In programming, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extensions, but closed for modification.

**Classes should be open for extension, but closed for modification.**

This principle aims to extend a Class's behavior without changing the existing behavior of that Class. This is to avoid causing bugs wherever the Class is being used.



# O - Open/closed principle

```
1 interface BaseService {
2     list();
3     getOne(id: string);
4     modify();
5     delete();
6 }
7
8 class Controller {
9     constructor(
10        public service: BaseService,
11    ){}
12
13    async getOne(id: string) {
14        const item = await this.service.getOne(id);
15        return item;
16    }
17 }
```

```
19 class Controller {
20     constructor(
21         public userService: BaseService,
22         public commentService: BaseService
23     ) {}
24
25     async getOne(id: string) {
26         const user = await this.userService.getOne(id);
27         const comments = await this.commentService.list(user.comments);
28
29         return {
30             ...user,
31             comments
32         }
33     }
34 }
```



# O - Open/closed principle

```
1 interface BaseService {
2     list();
3     getOne(id: string);
4     modify();
5     delete();
6 }
7
8 class BaseController {
9     constructor(
10         public service: BaseService,
11     ){}
12
13     async getOne(id: string) {
14         const item = await this.service.getOne(id);
15         return item;
16     }
17 }
```

```
19 class UserController extends BaseController {
20     constructor(
21         public userService: BaseService,
22         public commentService: BaseService
23     ) {
24         super(userService)
25     }
26
27     async getOne(id: string) {
28         const user = await this.service.getOne(id);
29         const comments = await this.commentService.list(user.comments);
30
31         return {
32             ...user,
33             comments
34         }
35     }
36 }
```





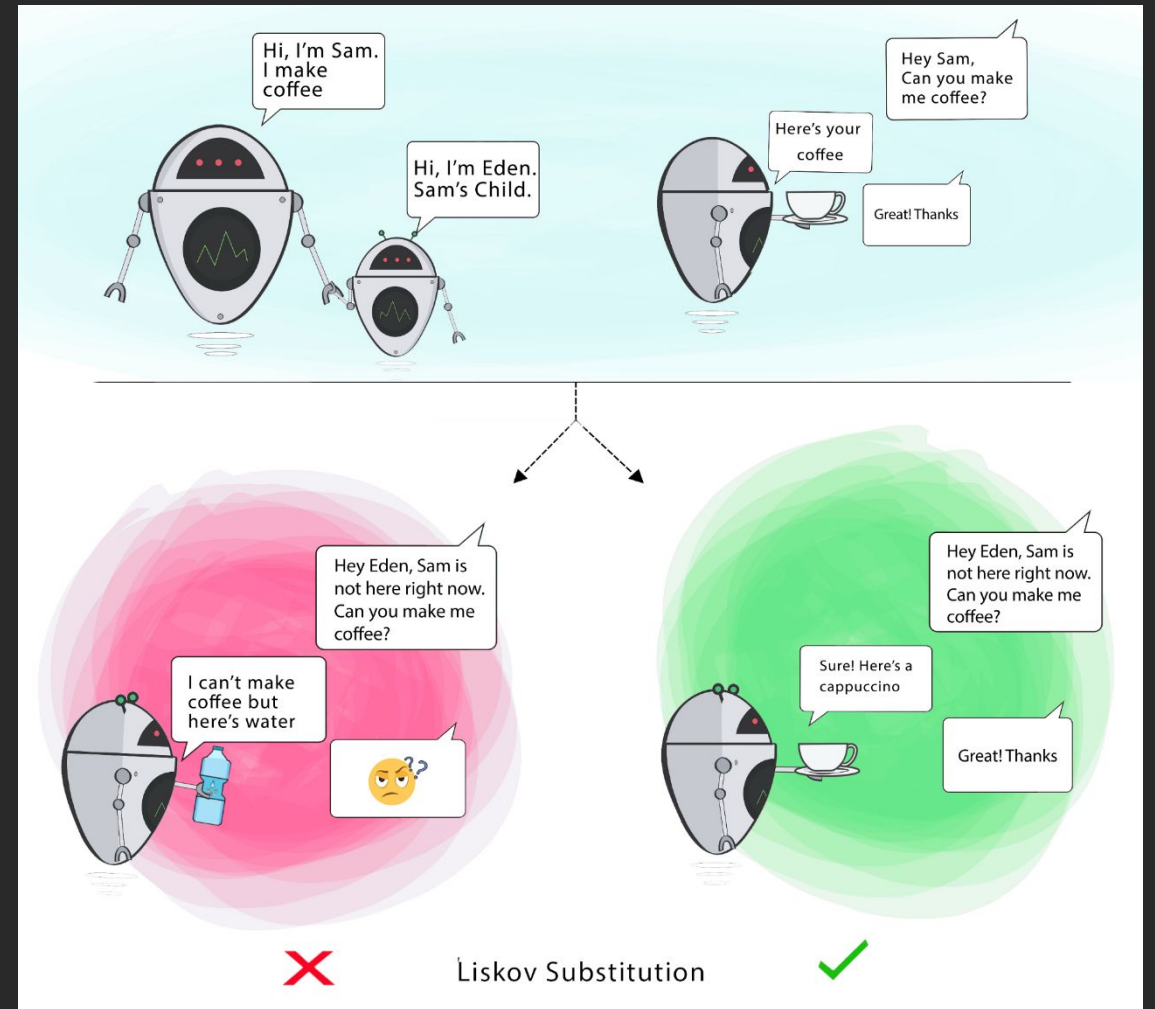
# L - Liskov Substitution

# L - Liskov Substitution

In programming, the Liskov substitution principle states that if S is a subtype of T, then objects of type T may be replaced (or substituted) with objects of type S.

More generally it states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.



# L - Liskov Substitution

```
1 interface BaseService {
2     list();
3     getOne(id: string);
4     modify();
5     delete();
6 }
7
8 class BaseController {
9     constructor(
10        public service: BaseService,
11    ){}
12
13    async getOne(id: string) {
14        const item = await this.service.getOne(id);
15        return item;
16    }
17 }
```

```
19 class UserController extends BaseController {
20     constructor(
21         public userService: BaseService,
22         public commentService: BaseService
23     ) {
24         super(userService)
25     }
26
27     async getOne(id: string) {
28         const user = await this.service.getOne(id);
29         const comments = await this.commentService.list(user.comments);
30
31         return {
32             ...user,
33             comments
34         }
35     }
36 }
```







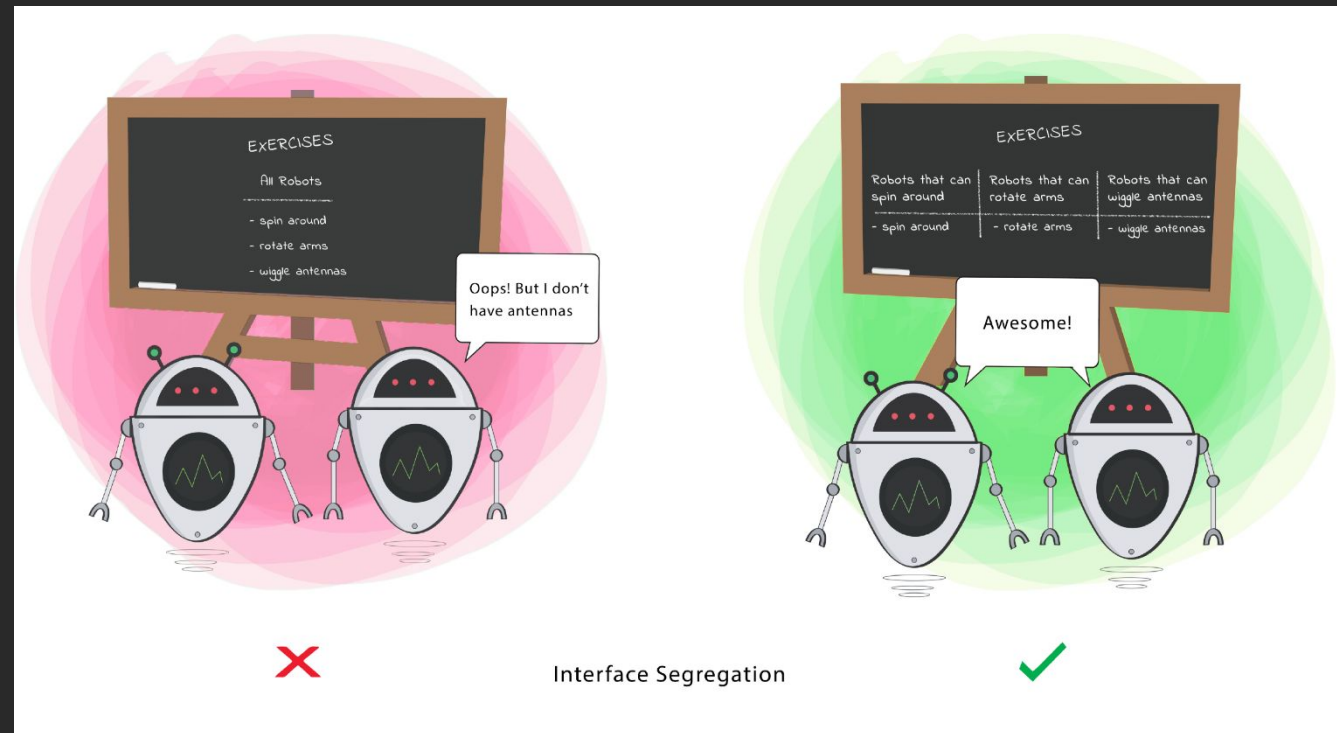
# I - Interface Segregation

# I - Interface Segregation

In programming, the interface segregation principle states that no client should be forced to depend on methods it does not use. Put more simply: Do not add additional functionality to an existing interface by adding new methods. Instead, create a new interface and let your class implement multiple interfaces if needed.

**Clients should not be forced to depend on methods that they do not use.**

This principle aims at splitting a set of actions into smaller sets so that a Class executes **ONLY** the set of actions it requires.



# I - Interface Segregation

```
1 class Apple {
2     calories: number;
3     shape: string = 'circle';
4     color: string = 'red';
5 }
6
7 class User {
8     eat(food: Apple): void {
9         //
10    }
11
12    roll(item: Apple): void {
13        //
14    }
15 }
```



# I - Interface Segregation

```
1  enum Shapes {
2  |   'circle',
3  | }
4
5  interface Rollable {
6  |   shape: Shapes.circle;
7  | }
8
9  interface Eatable {
10 |   calories: number;
11 | }
12
```

```
13 class Apple implements Rollable, Eatable {
14 |   calories = 240;
15 |   shape = Shapes.circle;
16 |   colo = 'red';
17 | }
18
19 class User {
20 |   eat(food: Eatable): void {
21 |     //
22 |   }
23
24 |   roll(item: Rollable): void {
25 |     //
26 |   }
27 | }
```





# D - Dependency inversion

# D - Dependency inversion

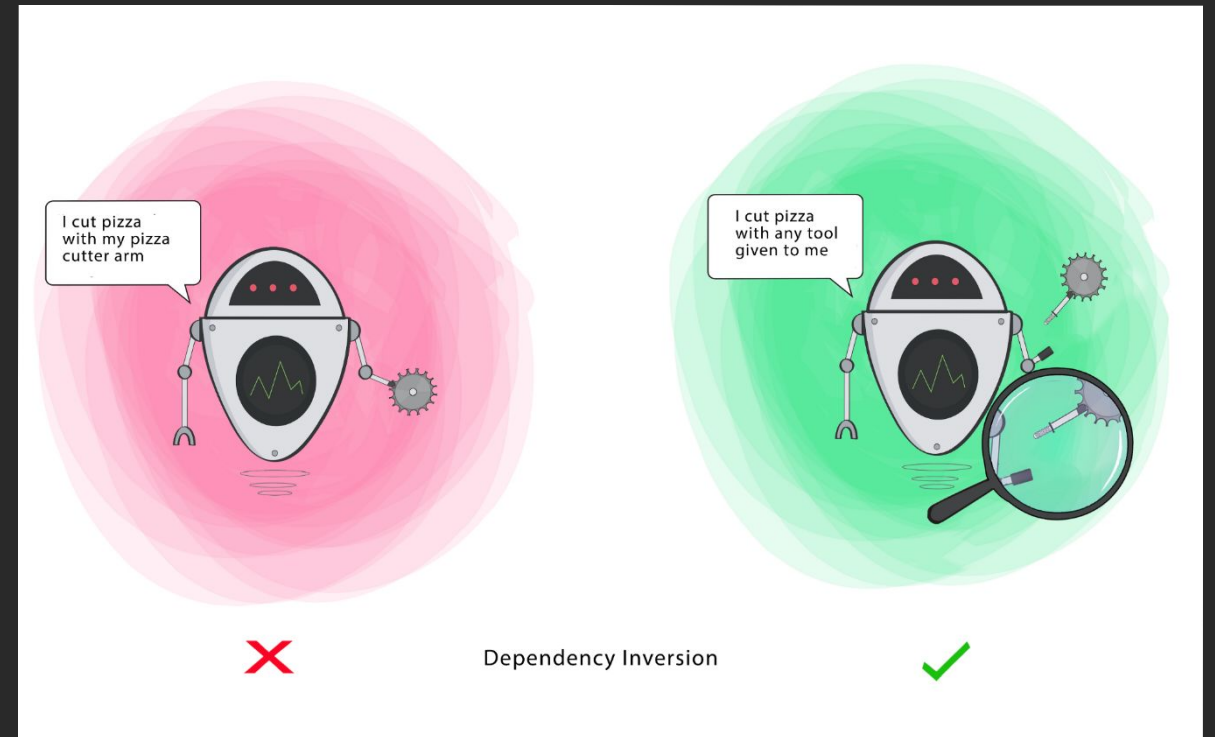
In programming, the dependency inversion principle is a way to decouple software modules.

This principle states that:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

To comply with this principle, we need to use a design pattern known as a dependency inversion pattern, most often solved by using dependency injection.

Typically, dependency injection is used simply by 'injecting' any dependencies of a class through the class' constructor as an input parameter.



# D - Dependency inversion

**High-level Module(or Class)** - Class that executes an action with a tool.

**Low-level Module (or Class)** - The tool that is needed to execute the action

**Abstraction** - Represents an interface that connects the two Classes.

**Details** - How the tool works

# D - Dependency inversion

```
1 interface BaseService {
2     list();
3     getOne(id: string);
4     modify();
5     delete();
6 }
7
8 class BaseController {
9     constructor(
10        public service: BaseService,
11    ){}
12
13    async getOne(id: string) {
14        const item = await this.service.getOne(id);
15        return item;
16    }
17 }
```

```
19 class UserController extends BaseController {
20     constructor(
21         public userService: BaseService,
22         public commentService: BaseService
23     ) {
24         super(userService)
25     }
26
27     async getOne(id: string) {
28         const user = await this.service.getOne(id);
29         const comments = await this.commentService.list(user.comments);
30
31         return {
32             ...user,
33             comments
34         }
35     }
36 }
```





# Links

<https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4>

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