studio HIVE

SOLID

Fullstack Bootcamp

2021 |hive-studio.net

Contents

01
Intro

02
Single Responsibility

03
Open/closed principle

04
Liskov Substitution

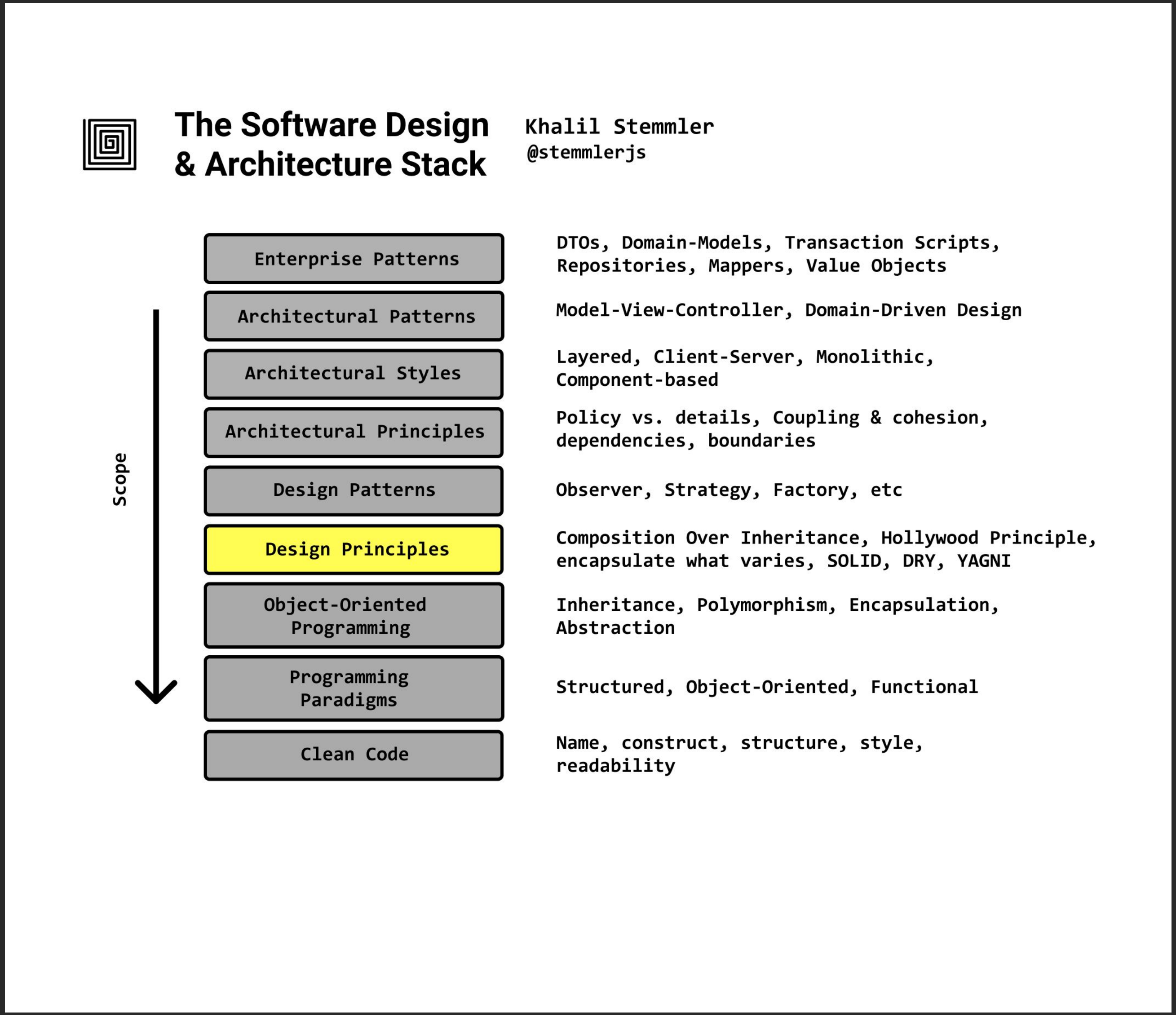
05
Interface Segregation

06
Dependency inversion

studio HIVE

Intro

Intro



The Software Design & Architecture Stack

Khalil Stemmler
@stemmlerjs

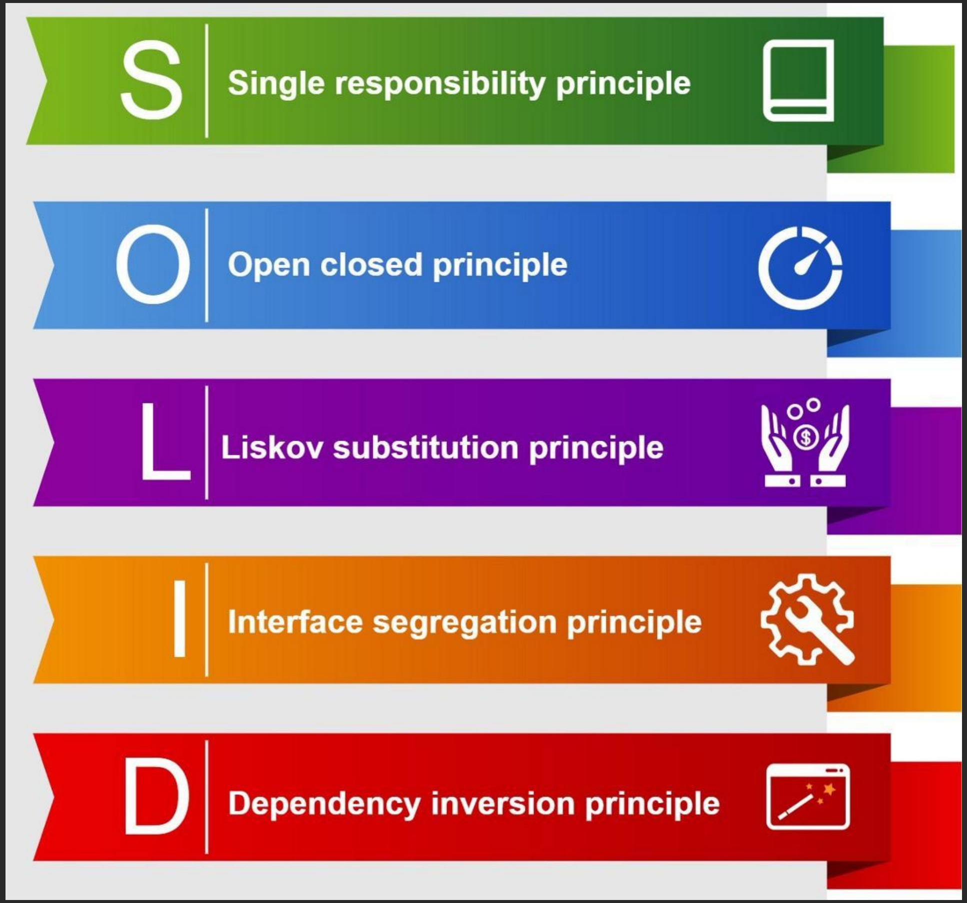
Enterprise Patterns	DTOs, Domain-Models, Transaction Scripts, Repositories, Mappers, Value Objects
Architectural Patterns	Model-View-Controller, Domain-Driven Design
Architectural Styles	Layered, Client-Server, Monolithic, Component-based
Architectural Principles	Policy vs. details, Coupling & cohesion, dependencies, boundaries
Design Patterns	Observer, Strategy, Factory, etc
Design Principles	Composition Over Inheritance, Hollywood Principle, encapsulate what varies, SOLID, DRY, YAGNI
Object-Oriented Programming	Inheritance, Polymorphism, Encapsulation, Abstraction
Programming Paradigms	Structured, Object-Oriented, Functional
Clean Code	Name, construct, structure, style, readability

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 principle

O Open closed principle

L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

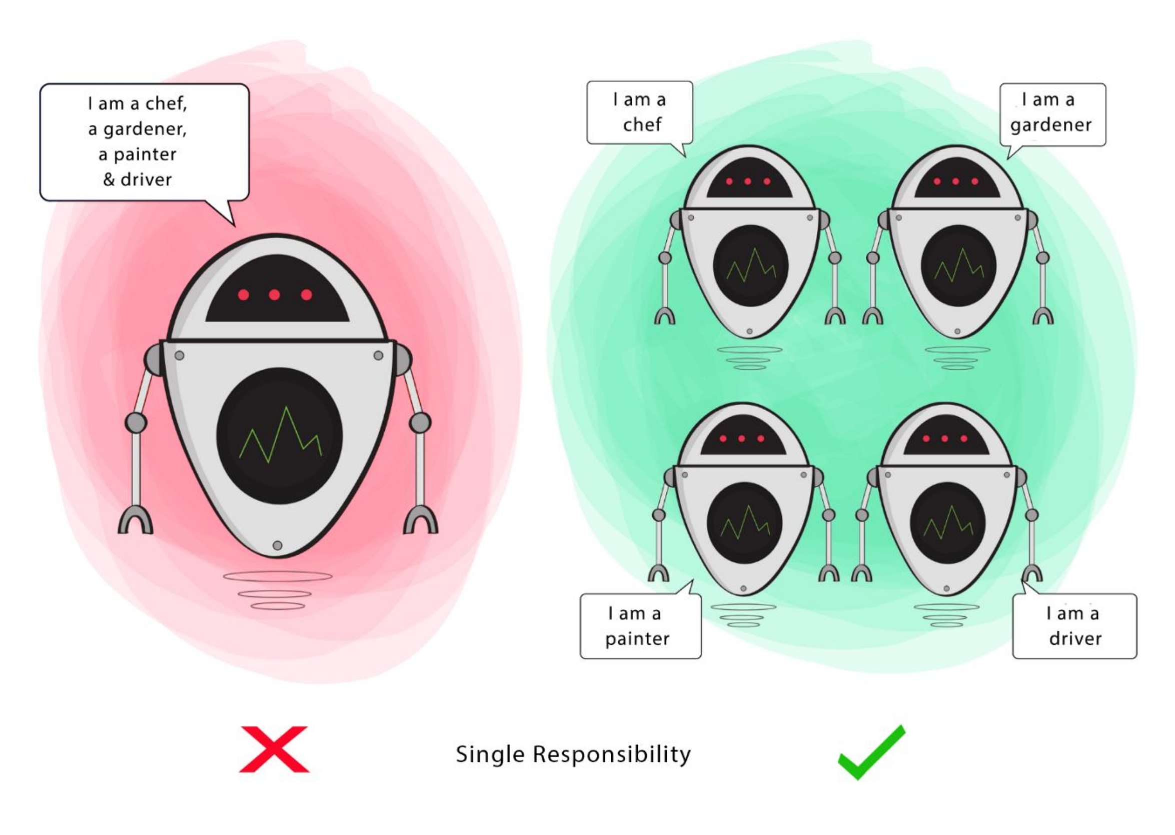
S – Single Responsibility

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.

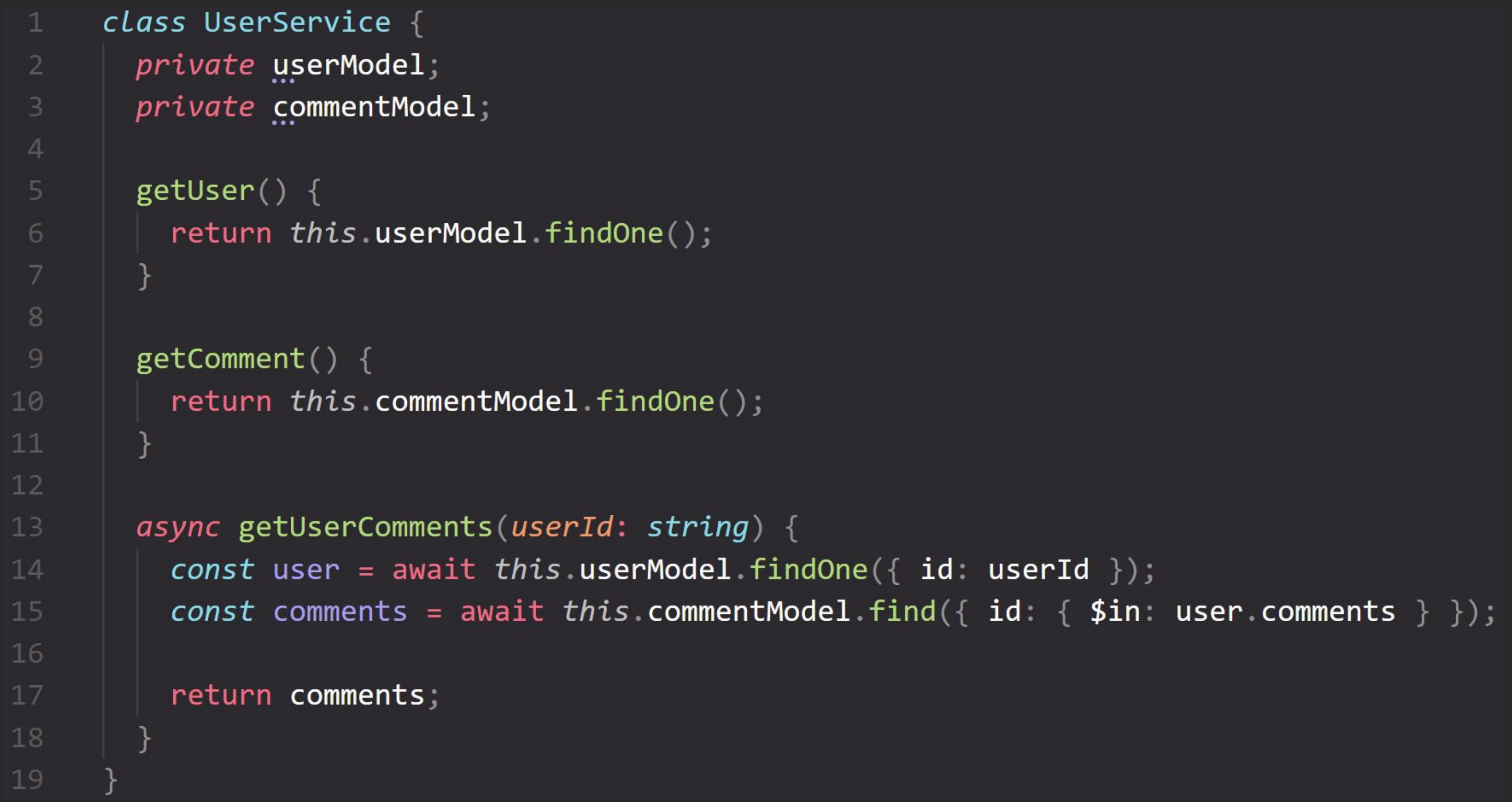
A class should have a single responsibility.

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.



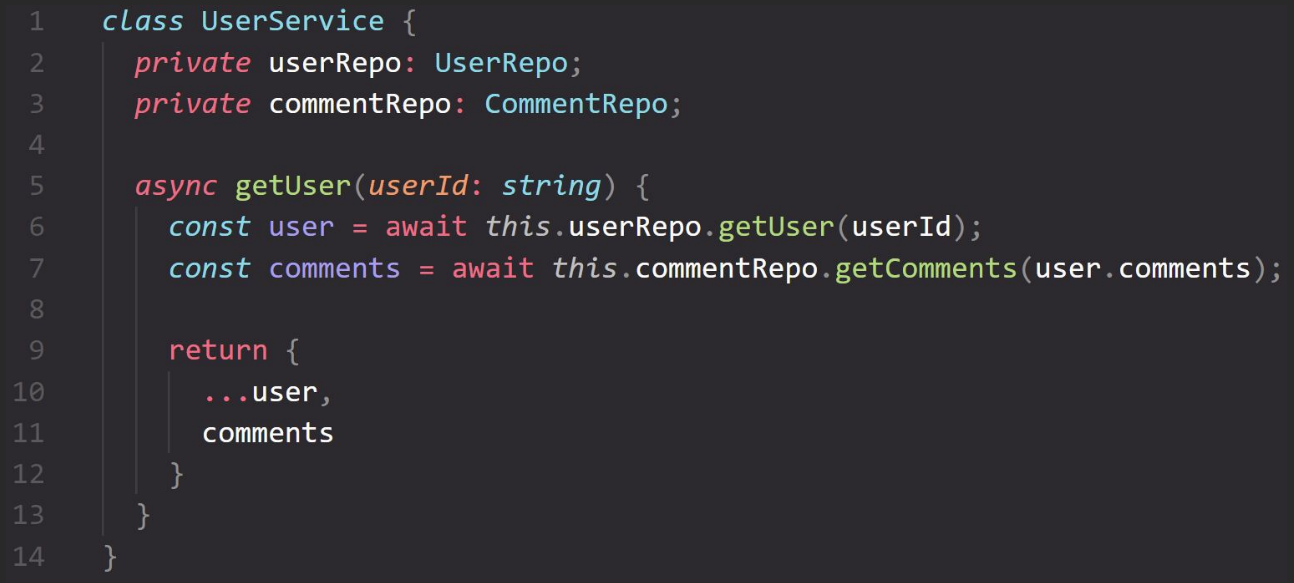
studio HIVE

S – Single Responsibility

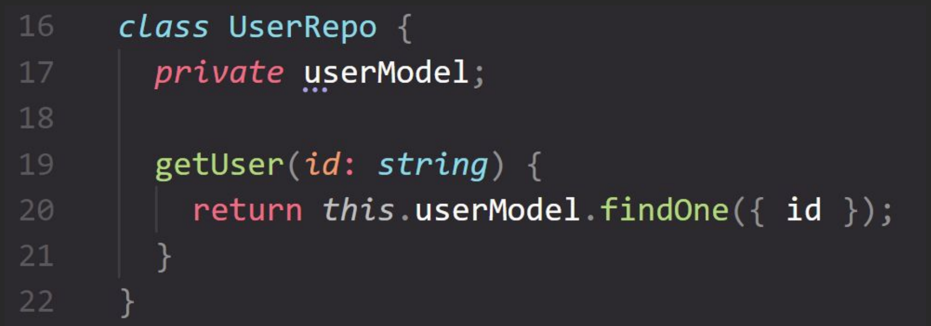


```
class UserService {
  private userModel;
  private commentModel;

  getUser() {
    return this.userModel.findOne();
  }

  getComment() {
    return this.commentModel.findOne();
  }

  async getUserComments(userId: string) {
    const user = await this.userModel.findOne({ id: userId });
    const comments = await this.commentModel.find({ id: { $in: user.comments } });

    return comments;
  }
}
```

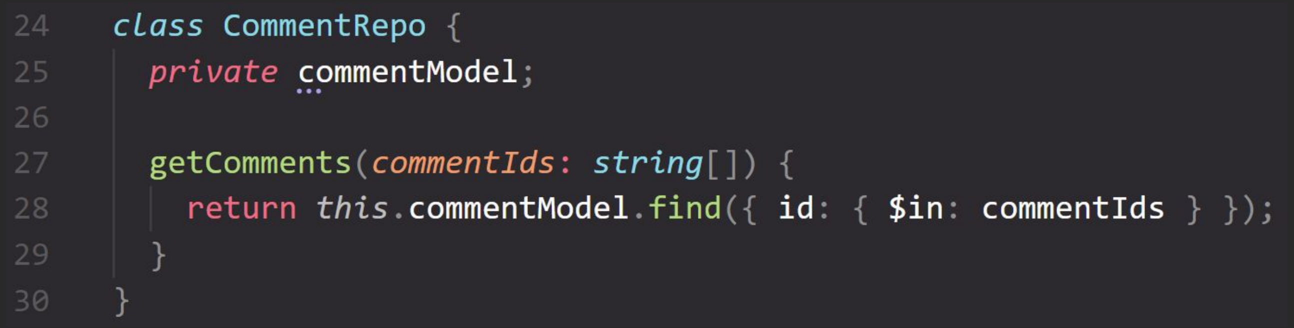
S – Single Responsibility



```
class UserService {
  private userRepo: UserRepo;
  private commentRepo: CommentRepo;

  async getUser(userId: string) {
    const user = await this.userRepo.getUser(userId);
    const comments = await this.commentRepo.getComments(user.comments);

    return {
      ...user,
      comments
    }
  }
}
```



```
class UserRepo {
  private userModel;

  getUser(id: string) {
    return this.userModel.findOne({ id });
  }
}
```



```
class CommentRepo {
  private commentModel;

  getComments(commentIds: string[]) {
    return this.commentModel.find({ id: { $in: commentIds } });
  }
}
```

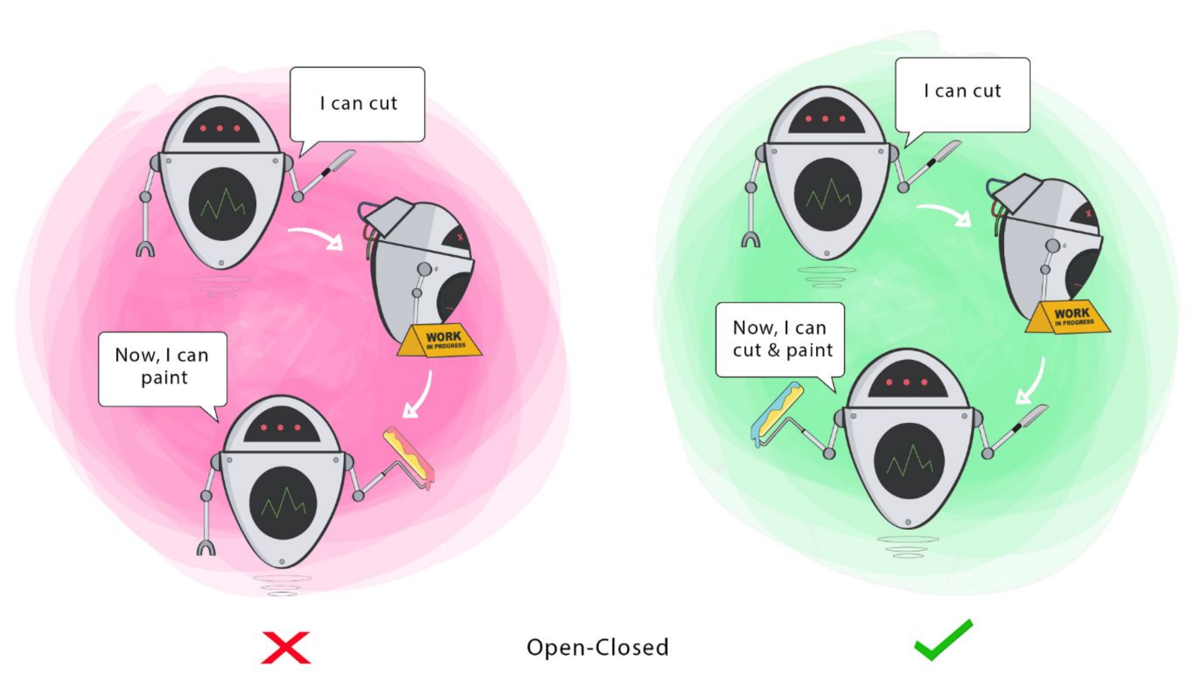
O - Open/closed principle

O - Open/closed principle

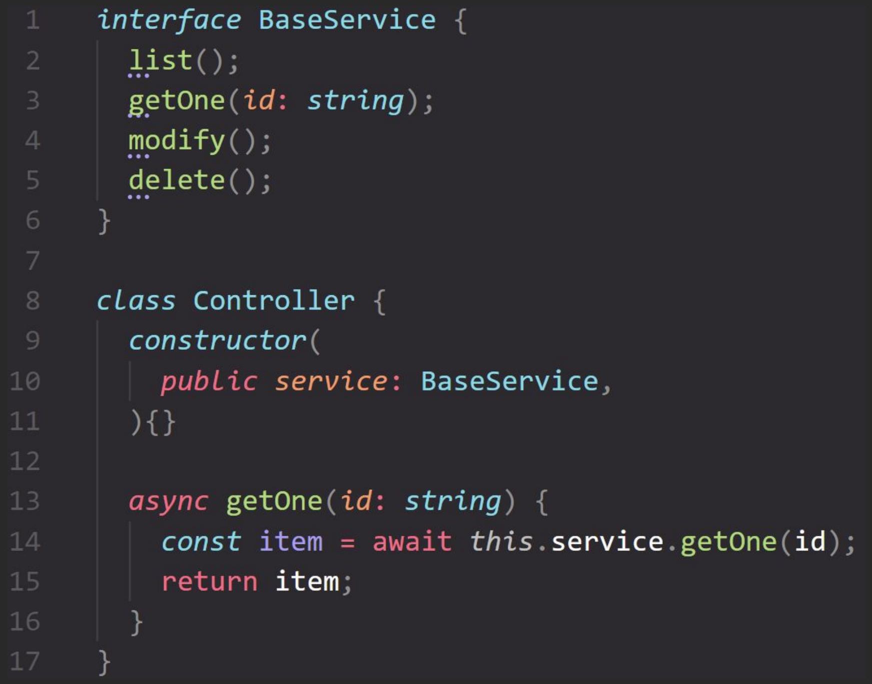
In programming, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extensions, but closed for modification.

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

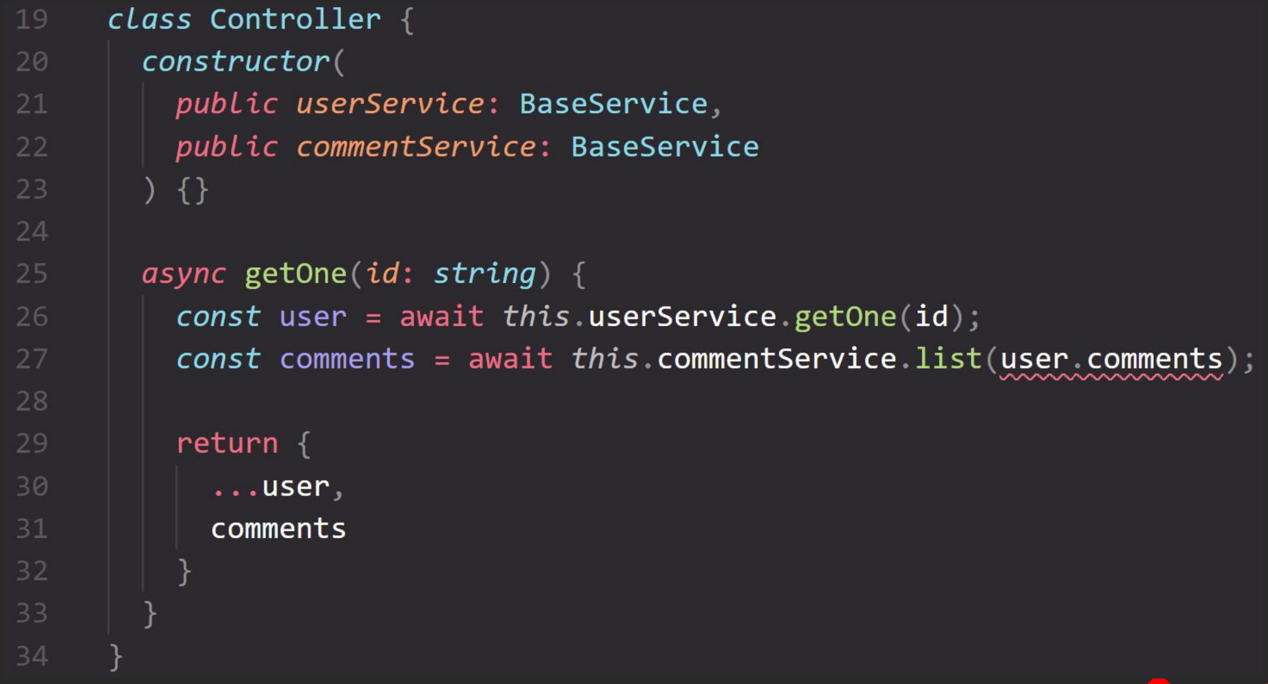
This principle aims to extend a Class’s behavior without changing the existing behavior of that Class. This is to avoid causing bugs wherever the Class is being used.



O - Open/closed principle



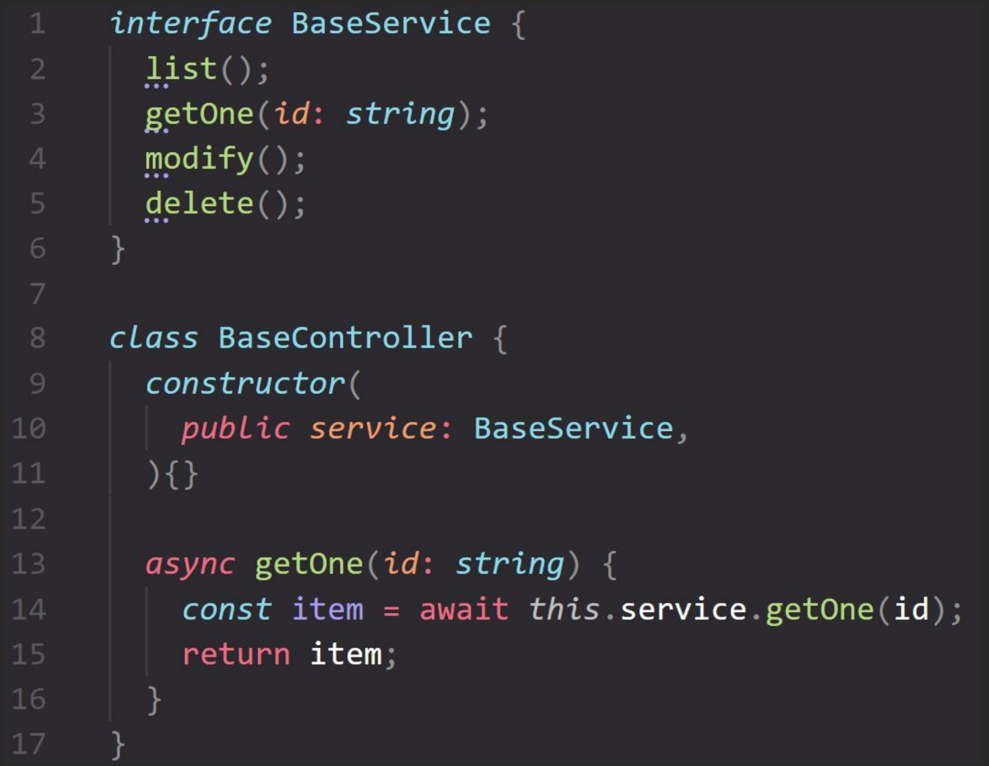
```typescript
interface BaseService {
  list();
  getOne(id: string);
  modify();
  delete();
}

class Controller {
  constructor(
    public service: BaseService,
  ){}

  async getOne(id: string) {
    const item = await this.service.getOne(id);
    return item;
  }
}
```



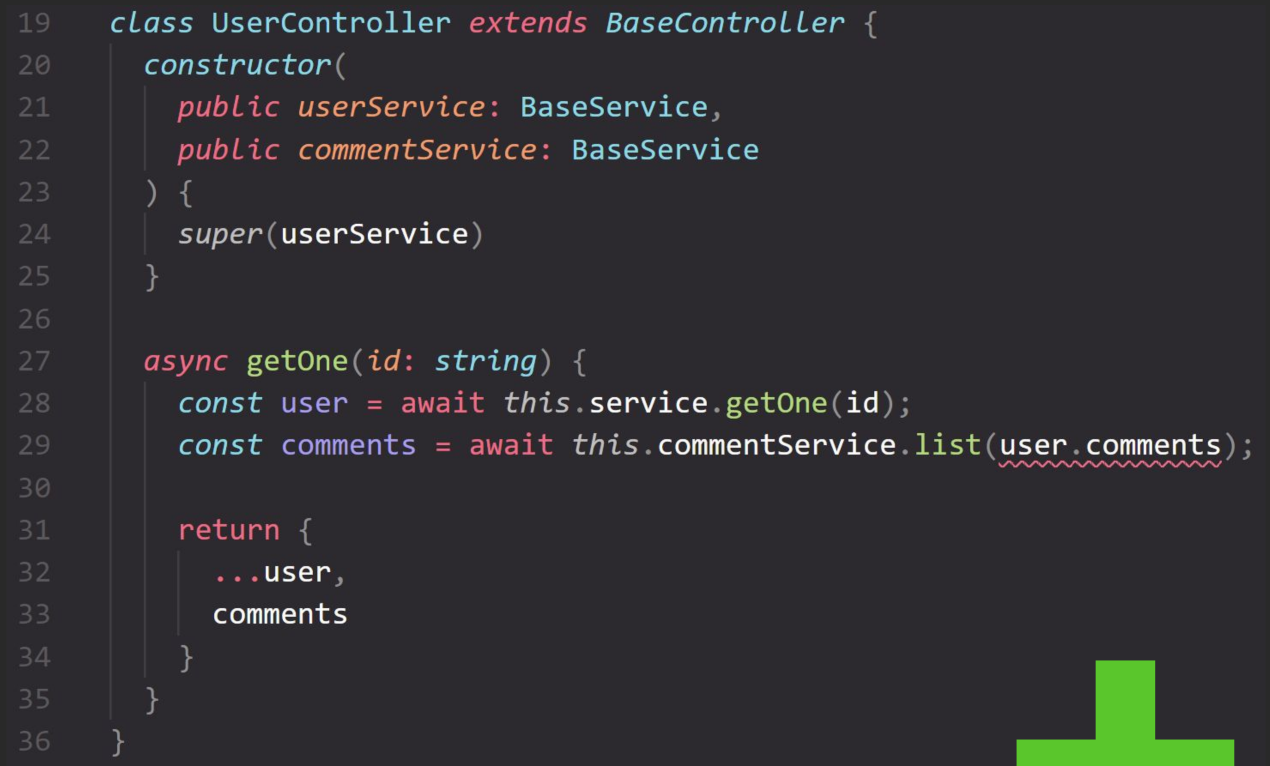
```typescript
class Controller {
  constructor(
    public userService: BaseService,
    public commentService: BaseService
  ) {}

  async getOne(id: string) {
    const user = await this.userService.getOne(id);
    const comments = await this.commentService.list(user.comments);

    return {
      ...user,
      comments
    }
  }
}
```

❌

O - Open/closed principle



```
interface BaseService {
  list();
  getOne(id: string);
  modify();
  delete();
}

class BaseController {
  constructor(
    public service: BaseService,
  ){}

  async getOne(id: string) {
    const item = await this.service.getOne(id);
    return item;
  }
}
```



```
class UserController extends BaseController {
  constructor(
    public userService: BaseService,
    public commentService: BaseService
  ) {
    super(userService)
  }

  async getOne(id: string) {
    const user = await this.service.getOne(id);
    const comments = await this.commentService.list(user.comments);

    return {
      ...user,
      comments
    }
  }
}
```

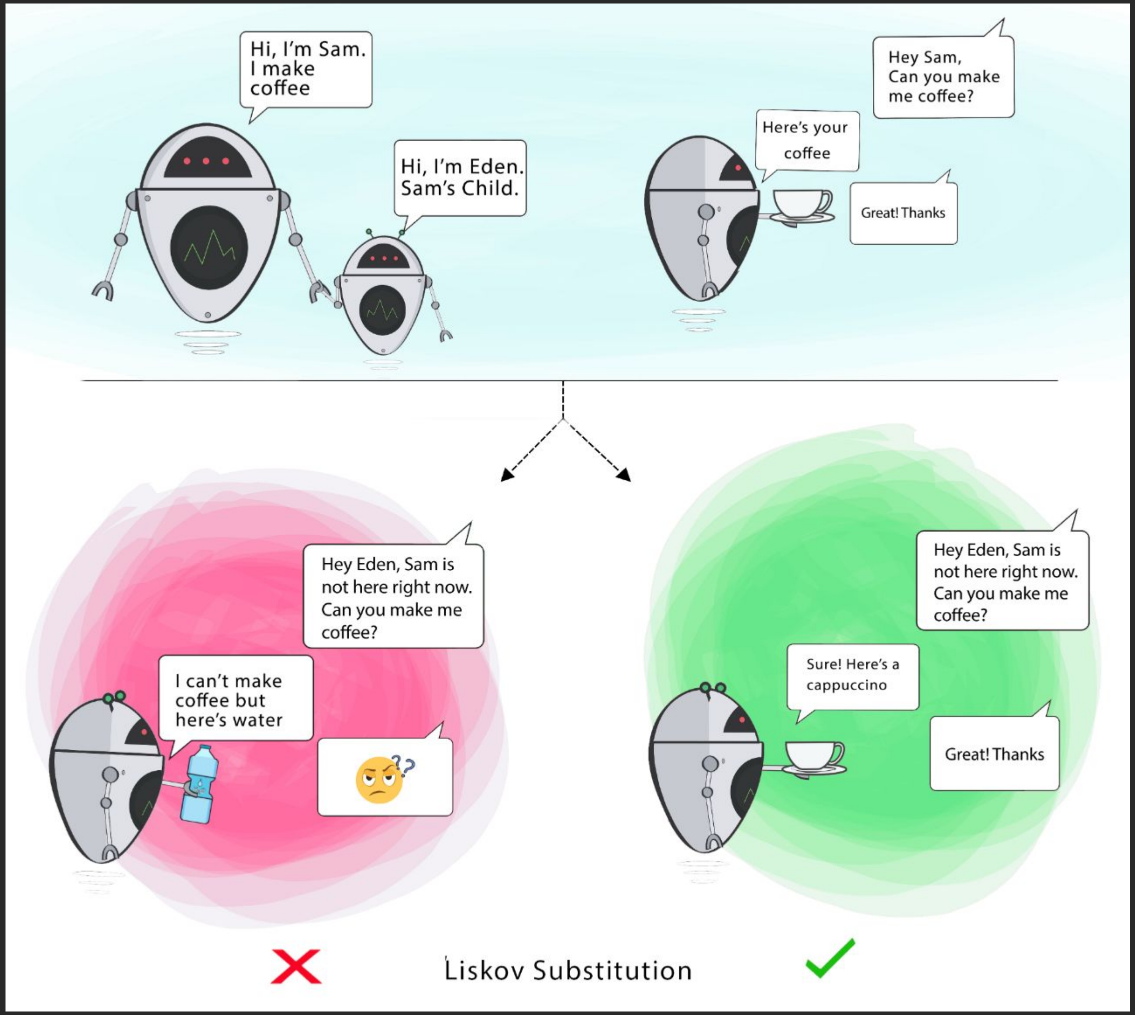
L - Liskov Substitution

L - Liskov Substitution

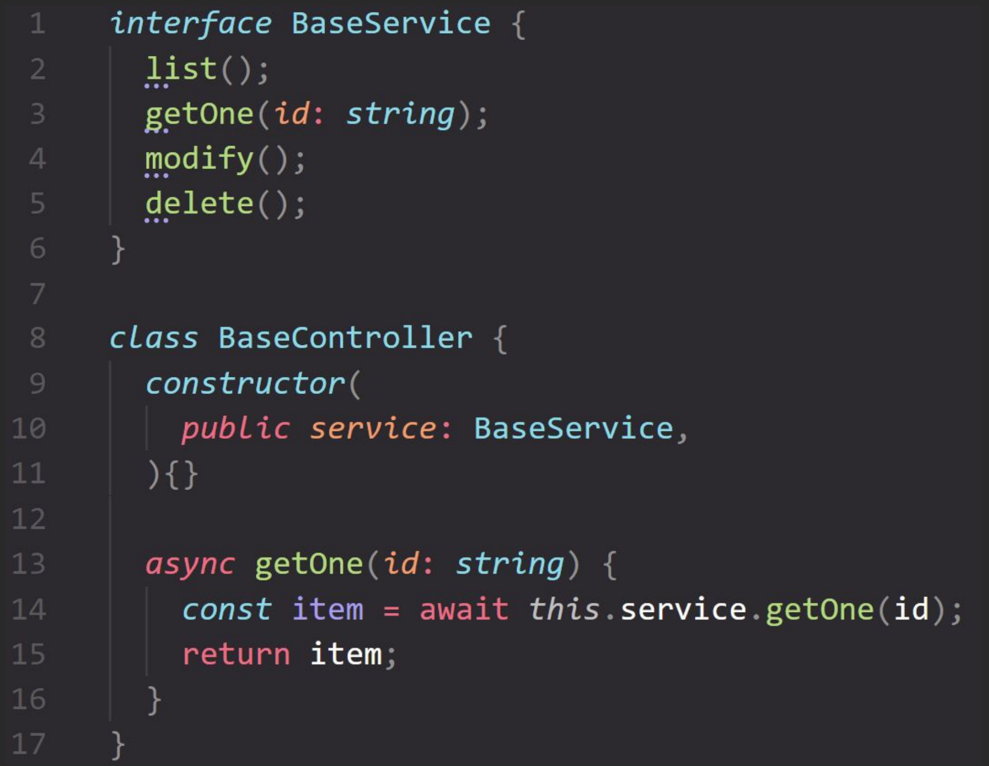
In programming, the Liskov substitution principle states that if S is a subtype of T, then objects of type T may be replaced (or substituted) with objects of type S.

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

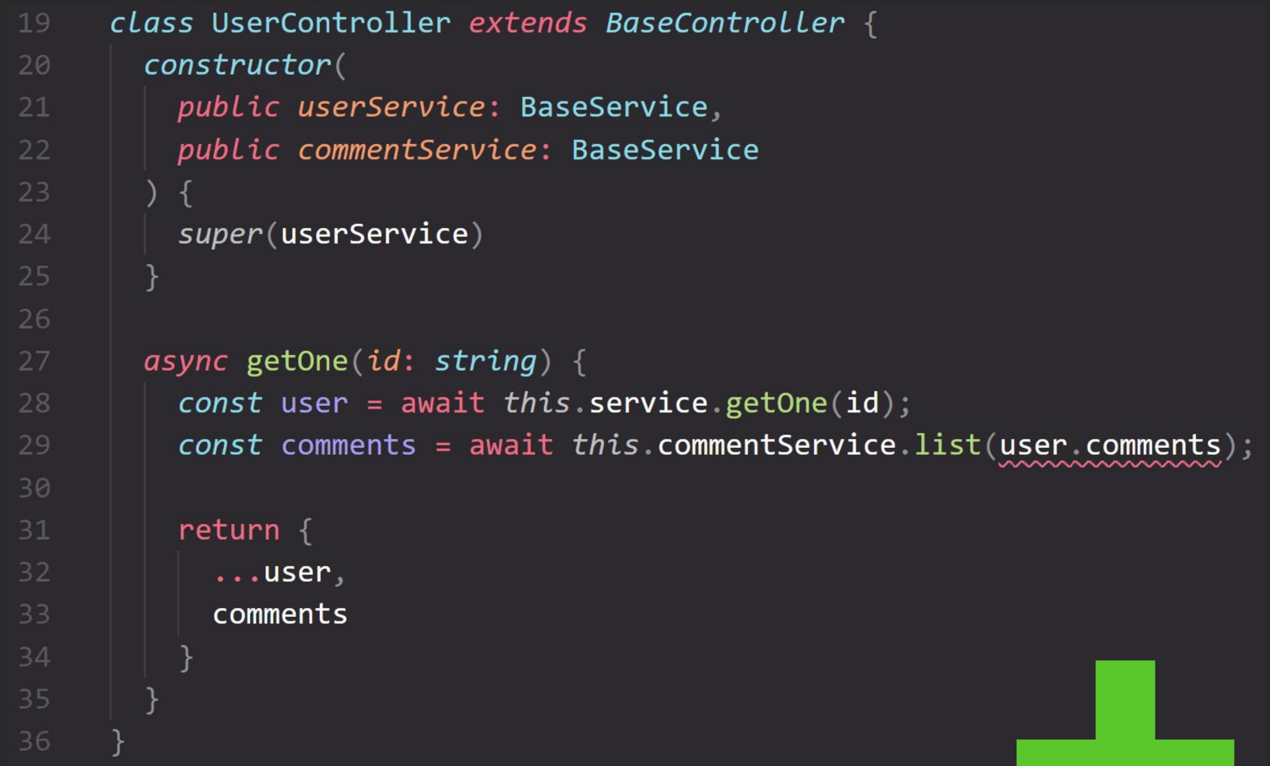
The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
✗
Liskov Substitution
✓

L - Liskov Substitution



```
interface BaseService {
  list();
  getOne(id: string);
  modify();
  delete();
}

class BaseController {
  constructor(
    public service: BaseService,
  ){}

  async getOne(id: string) {
    const item = await this.service.getOne(id);
    return item;
  }
}
```



```
class UserController extends BaseController {
  constructor(
    public userService: BaseService,
    public commentService: BaseService
  ) {
    super(userService)
  }

  async getOne(id: string) {
    const user = await this.service.getOne(id);
    const comments = await this.commentService.list(user.comments);

    return {
      ...user,
      comments
    }
  }
}
```

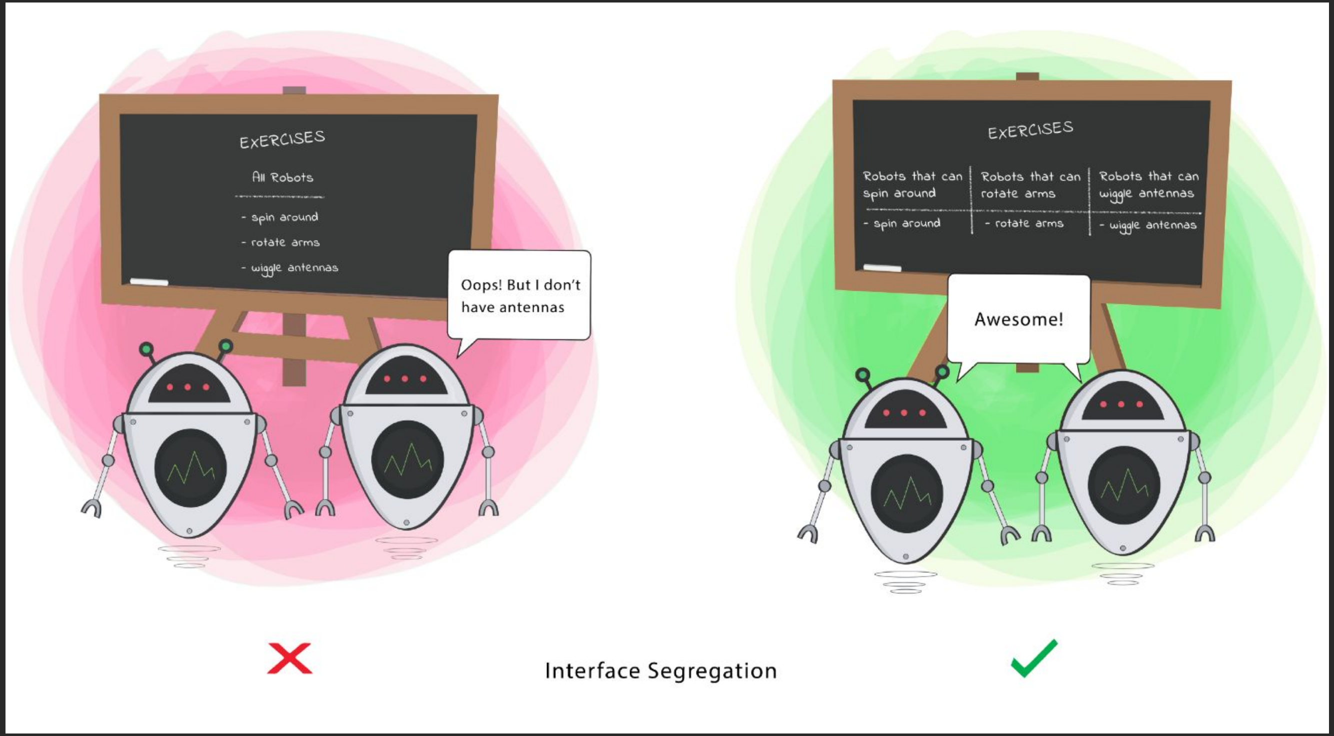
I - Interface Segregation

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.

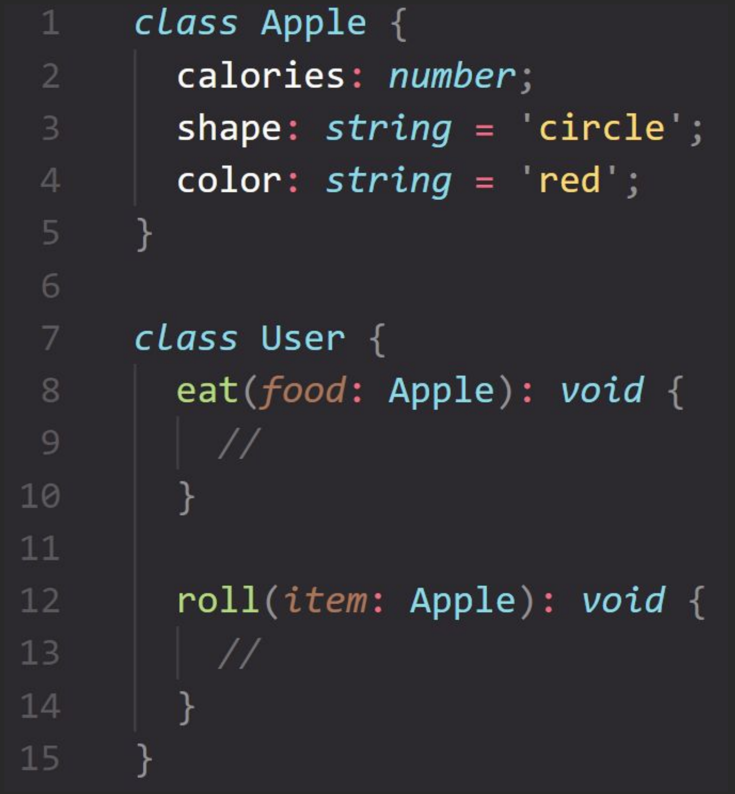
Clients should not be forced to depend on methods that they do not use.

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



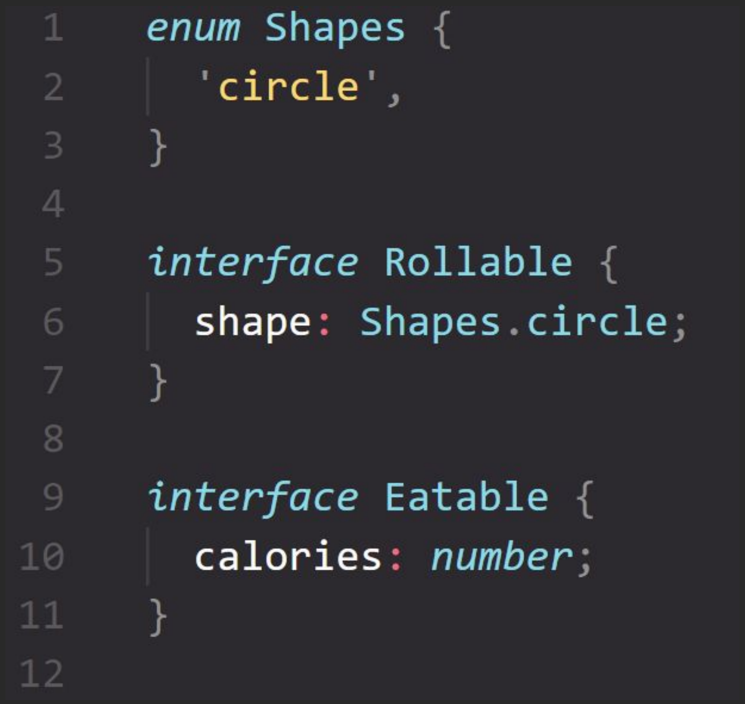
studio HIVE

I - Interface Segregation

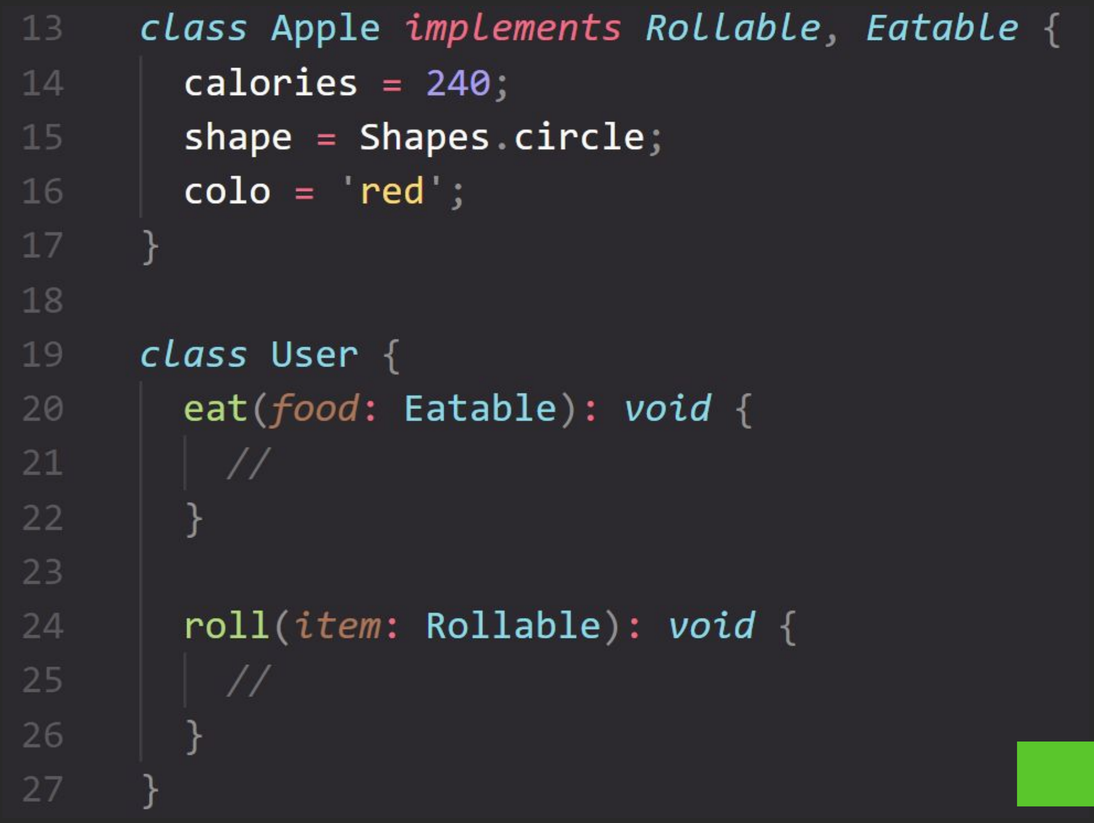


```
class Apple {
  calories: number;
  shape: string = 'circle';
  color: string = 'red';
}

class User {
  eat(food: Apple): void {
    //
  }

  roll(item: Apple): void {
    //
  }
}
```

I - Interface Segregation



```
enum Shapes {
  'circle',
}

interface Rollable {
  shape: Shapes.circle;
}

interface Eatable {
  calories: number;
}

```



```
class Apple implements Rollable, Eatable {
  calories = 240;
  shape = Shapes.circle;
  colo = 'red';
}

class User {
  eat(food: Eatable): void {
    //
  }

  roll(item: Rollable): void {
    //
  }
}
```

D - Dependency inversion

D - Dependency inversion

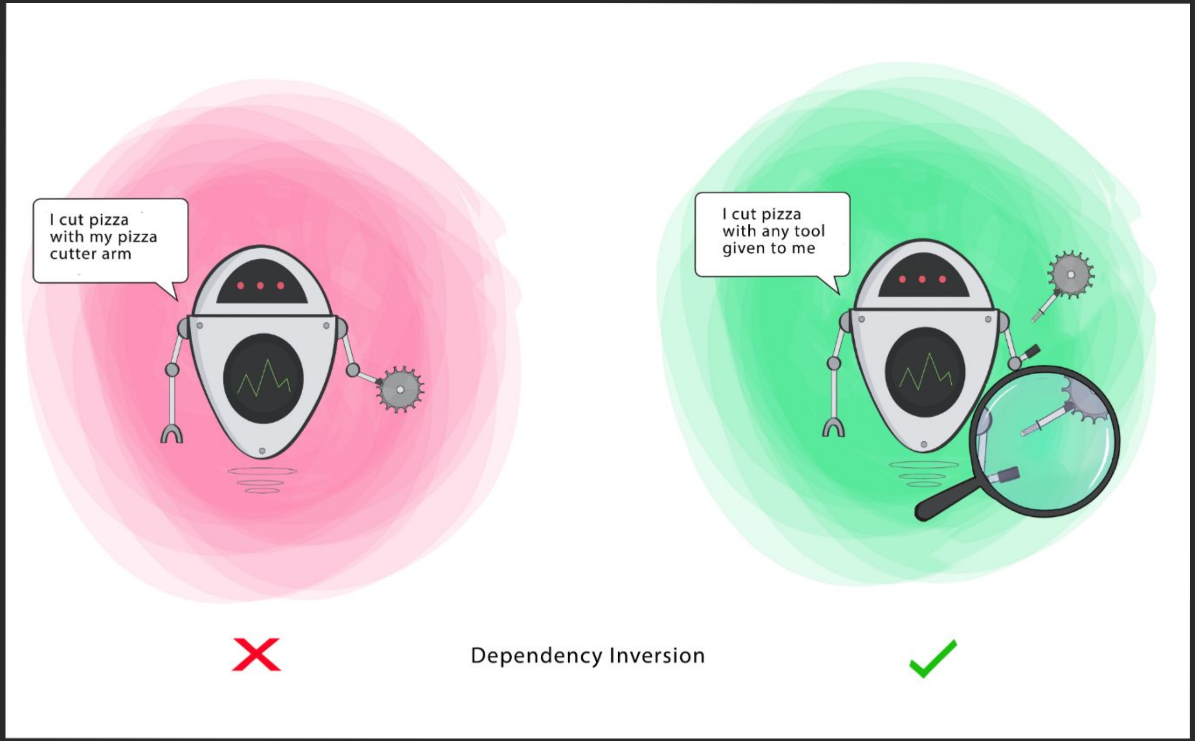
In programming, the dependency inversion principle is a way to decouple software modules.

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.


I cut pizza with my pizza cutter arm
I cut pizza with any tool given to me
✗
Dependency Inversion
✓

D - Dependency inversion

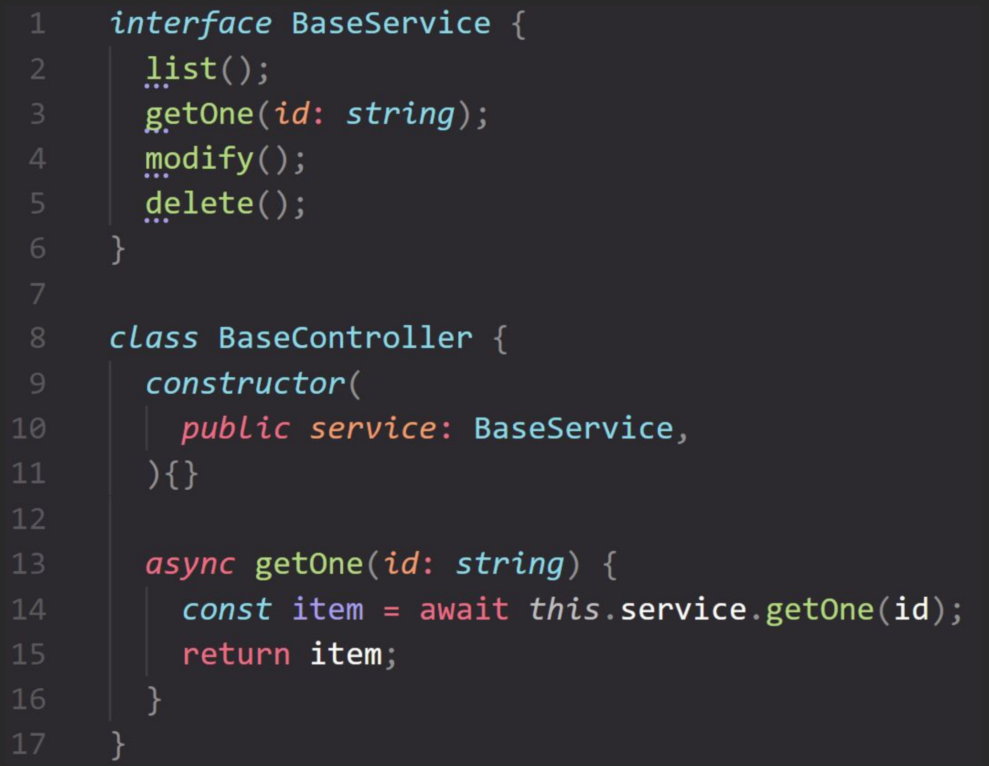
High-level Module(or Class) - Class that executes an action with a tool.

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

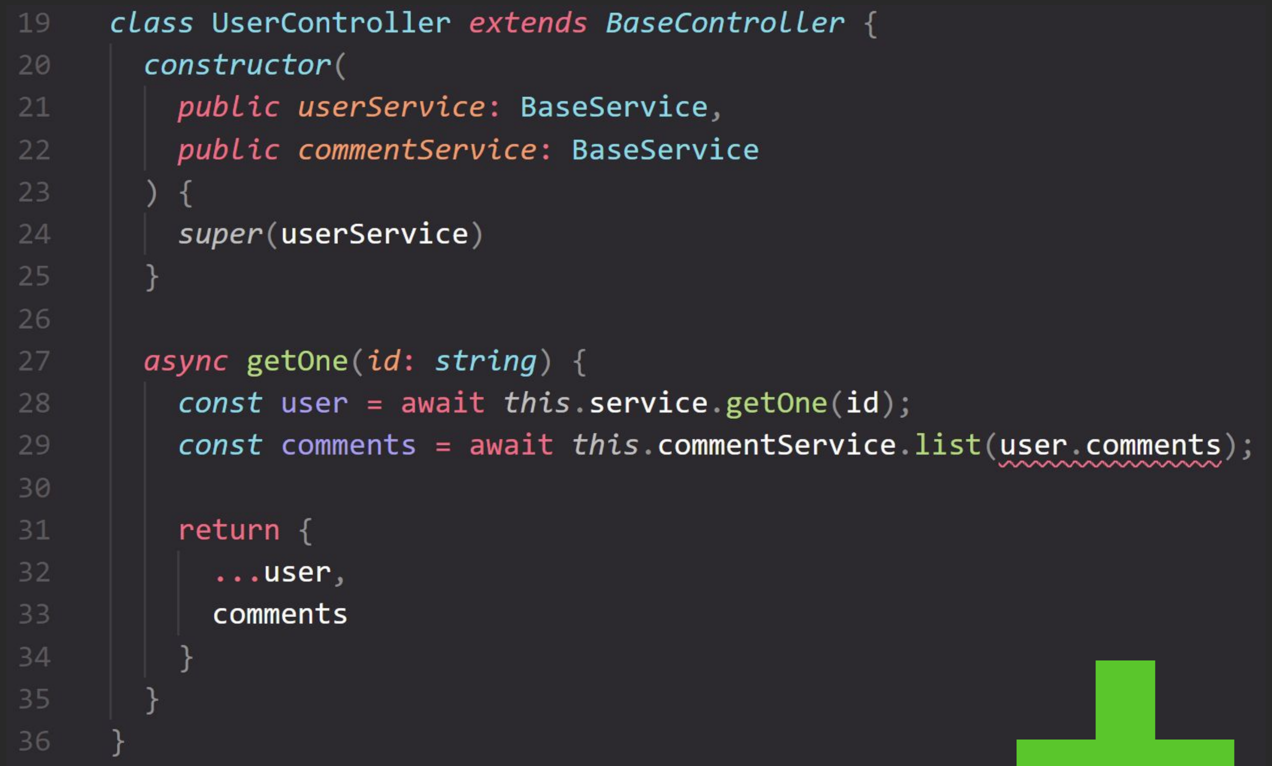
Abstraction - Represents an interface that connects the two Classes.

Details - How the tool works

D - Dependency inversion



```
interface BaseService {
  list();
  getOne(id: string);
  modify();
  delete();
}

class BaseController {
  constructor(
    public service: BaseService,
  ){}

  async getOne(id: string) {
    const item = await this.service.getOne(id);
    return item;
  }
}
```



```
class UserController extends BaseController {
  constructor(
    public userService: BaseService,
    public commentService: BaseService
  ) {
    super(userService)
  }

  async getOne(id: string) {
    const user = await this.service.getOne(id);
    const comments = await this.commentService.list(user.comments);

    return {
      ...user,
      comments
    }
  }
}
```

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