formation.tech

Software Architecture

Software Architecture - SOLID Principles

- SOLID means :
  - **S**ingle responsibility principle
  - **O**pen–closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation principle
  - **D**ependency inversion principle

Software Architecture - SOLID Principles

- Single responsibility principle

  Every module, class or function in a computer program should have responsibility over a single part of that program's functionality, and it should encapsulate that part.

Software Architecture - SOLID Principles

- Open–closed principle

  Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

Software Architecture - SOLID Principles

- Liskov substitution principle

  If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

Software Architecture - SOLID Principles

- Interface segregation principle

  Many client-specific interfaces are better than one general-purpose interface

Software Architecture - SOLID Principles

- Dependency inversion principle

  Depend upon abstractions, not concretions

Software Architecture - Bad Example

```ts
export class CoffeeCup {
  constructor(
    private coffeeType: "arabica" | "robusta",
    private cupCapacity: number
  ) {}
}
```

```ts
import { CoffeeCup } from "./CoffeeCup";

const coffeeCup = new CoffeeCup("arabica", 20);
```

- ✘ Single Responsibility : coffee is not separate from the cup
- ✘ Open–closed principle : to create a cup of tea we would need to modify the code
- ✘ Dependency inversion principle : CoffeeCup does not have any dependency

Software Architecture - Bad Example

```ts
import fs from "fs";

export class FileLogger {
  constructor(private filePath: string) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await fs.promises.appendFile(this.filePath, formatted);
  }
}
```

```ts
import { FileLogger } from "./FileLogger";

const logger = new FileLogger("app.log");
await logger.log('Message');
```

- ✘ Single Responsibility : file writing is not separate from logging
- ✘ Open–closed principle : to log in the terminal we would need to modify the code
- ✘ Dependency inversion principle : FileLogger does not have any dependency

Software Architecture - Bad Example

```ts
export class Coffee {
  constructor(
    private type: "arabica" | "robusta",
  ) {}
}
```

```ts
import { Coffee } from "./Coffee";

export class Cup {
  private coffee: Coffee;
  constructor(
    private capacity: number
  ) {
    this.coffee = new Coffee('arabica');
  }
}
```

```ts
import { Cup } from "./Cup";

const coffeeCup = new Cup(10);
```

- ✔ Single Responsibility : 1 class for coffee, 1 class for cup
- ✘ Open–closed principle : to create a cup of tea we would need to modify the code
- ✘ Dependency inversion principle : Coffee is a hard-coded dependency, it can't be replaced programmatically

Software Architecture - Bad Example

```ts
import fs from "fs";

export class FileWriter {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.promises.appendFile(this.filePath, msg);
  }
}
```

```ts
import { FileWriter } from "./FileWriter";

export class Logger {
  private fileWriter: FileWriter;

  constructor(filePath: string) {
    this.fileWriter = new FileWriter(filePath);
  }

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.fileWriter.write(formatted);
  }
}
```

```ts
import { Logger } from "./Logger";

const logger = new Logger("app.log");
await logger.log('Message');
```

Software Architecture - Bad Example

- ✔ Single Responsibility : 1 class for file writing, 1 class for logging
- ✘ Open–closed principle : to log in the terminal we would need to modify the code
- ✘ Dependency inversion principle : Coffee is a hard-coded dependency, it can't be replaced programmatically

Software Architecture - Bad Example

```ts
export class Coffee {
  constructor(
    private type: "arabica" | "robusta",
  ) {}
}
```

```ts
import { Coffee } from "./Coffee";

export class Cup {
  constructor(
    private capacity: number,
    private coffee: Coffee
  ) {}
}
```

```ts
import { Coffee } from "./Coffee";
import { Cup } from "./Cup";

const coffee = new Coffee('arabica');
const coffeeCup = new Cup(10, coffee);
```

- ✔ Single Responsibility : 1 class for coffee, 1 class for cup
- ✔ Open–closed principle : we could pass a specialization of Coffee to modify the code (but it could break the Liskov substitution principle)
- ✘ Dependency inversion principle : Coffee is a hard-coded dependency, it can't be replaced programmatically

Software Architecture - Bad Example

```ts
import fs from "fs";

export class FileWriter {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.promises.appendFile(this.filePath, msg);
  }
}
```

```ts
import { FileWriter } from "./FileWriter";

export class Logger {
  constructor(private fileWriter: FileWriter) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.fileWriter.write(formatted);
  }
}
```

```ts
import { FileWriter } from "./FileWriter";
import { Logger } from "./Logger";

const writer = new FileWriter("app.log");
const logger = new Logger(writer);
await logger.log('Message');
```

Software Architecture - Bad Example

- ✔ Single Responsibility : 1 class for file writing, 1 class for logging
- ✔ Open–closed principle : we could pass a specialization of FileWriter to modify the code (but it could break the Liskov substitution principle)
- ✘ Dependency inversion principle : Coffee is a hard-coded dependency, it can't be replaced programmatically

Software Architecture - Good Example

```ts
export interface DrinkInterface {}
```

```ts
import { DrinkInterface } from "./DrinkInterface";

export class Coffee implements DrinkInterface {
  constructor(private type: "arabica" | "robusta") {}
}
```

```ts
import { DrinkInterface } from "./DrinkInterface";

export class Cup {
  constructor(private capacity: number, private drink: DrinkInterface)
}
```

```ts
import { Coffee } from "./Coffee";
import { Cup } from "./Cup";

const coffee = new Coffee("arabica");
const coffeeCup = new Cup(10, coffee);
```

- ✔ Single Responsibility : 1 class for coffee, 1 class for cup
- ✔ Open–closed principle : we could pass an implementation of DrinkInterface to modify the code
- ✔ Dependency inversion principle : Cup depends on an abstraction, we could easily create a cup of Tea

Software Architecture - Good Example

```ts
export interface WriterInterface {
  write(msg: string): Promise<void>;
}
```

```ts
import fs from "fs";
import { WriterInterface } from "./WriterInterface";

export class FileWriter implements WriterInterface {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.promises.appendFile(this.filePath, msg);
  }
}
```

```ts
import { WriterInterface } from "./WriterInterface";

export class Logger {
  constructor(private writer: WriterInterface) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.writer.write(formatted);
  }
}
```

```ts
import { FileWriter } from "./FileWriter";
import { Logger } from "./Logger";

const writer = new FileWriter("app.log");
const logger = new Logger(writer);
await logger.log('Message');
```

Software Architecture - Good Example

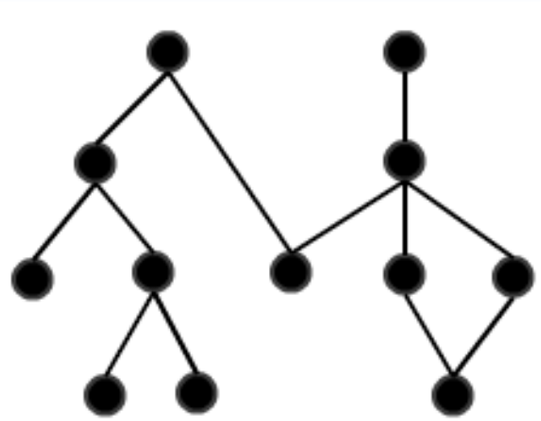
- ✔ Single Responsibility : 1 class for file writing, 1 class for logging
- ✔ Open–closed principle : we could pass an implementation of WriterInterface to modify the code
- ✔ Dependency inversion principle : Logger depends on an abstraction, we could easily create a logger that would log on a different

InversifyJS

InversifyJS - Dependency Injection Container

- The Dependency Inversion Principle conduct the code to instantiate a lot of objects
- When we work with external libraries it can be difficult to know which objects to create
- Some of those dependencies could be shared between services
- The Dependency Injection Container is a library that will manage those dependencies for the application

- We will describe how to create the dependencies and associate them with a key
- When we will need the dependency we will ask them using that key
- The dependency will be created on demand
- The dependency could be cached



InversifyJS - Setup

- InversifyJS is can be used in JavaScript and TypeScript
- It is based on decorators which a still experimental :
  https://github.com/tc39/proposal-decorators
- Installation :
  npm i inversify reflect-metadata
- tsconfig.json :

```
{
  "compilerOptions": {
    /* ... */
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    /* ... */
  }
}
```

InversifyJS - Container

- To create the container :

```
const container = new Container();
```

- Registering a value directly :

```
const writer = new ConsoleWriter();
const logger = new Logger(writer);
container.bind<Logger>('logger').toConstantValue(logger);
```

- Registering a function :

```
container.bind<Logger>('logger').toDynamicValue(() => {
  const writer = new FileWriter();
  const logger = new Logger(writer);
  return logger;
});
```

- Getting a service

```
const logger = container.get<Logger>('logger');
```

InversifyJS - Container

- To be injected, a service has to use the injectable decorator :

```
@injectable()
export class Logger {}
```

- Classes can be used as key

```
container.bind(Logger).to(Logger);
```

```
const logger = container.get(Logger);
```

- toSelf is a shorthand method

```
container.bind(Logger).toSelf();
```

- autoBindInjectable is even shorter, no need to do the declaration :

```
let container = new Container({
  autoBindInjectable: true,
});
```

```
const logger = container.get(Logger);
```

InversifyJS - Container

- During the declaration we can choose if the service will be cache (Singleton)

```
container.bind(Logger).toSelf().inSingletonScope();
```

- It can be defaulted :

```
let container = new Container({
  defaultScope: 'Singleton',
});
```

- 3 possibles values :
  - Transient (default) : each request (.get) will create a new objet
  - Singleton : object will be shared across requests (.get)
  - Request : object will be shared within one request (.get)

InversifyJS - Container

- Binding interfaces

```
container.bind(WriterInterface).toService(FileWriter);
```

```
@injectable()
export class Logger {
  constructor(@inject(WriterInterface) private writer: WriterInterface) {}
}
```

- Inject in properties

```
@injectable()
export class Logger {
  @inject(WriterInterface)
  private writer!: WriterInterface
}
```

- Modules

```
const loggerModule = new ContainerModule((bind: interfaces.Bind) => {
  bind(Logger).toSelf().inSingletonScope;
});
```

InversifyJS - Container

- Tagged services

```
container.bind(WriterInterface).to(FileWriter).whenTargetTagged('env', 'production');
container.bind(WriterInterface).to(ConsoleWriter).whenTargetTagged('env', 'test');
```

```
@injectable()
export class Logger {
  @inject(WriterInterface) @tagged("env", process.env.NODE_ENV)
  private writer!: WriterInterface
}
```