Applying SOLID Design Principles in Angular: Best Practices and Examples

SOLID is a set of design principles that can be used to develop maintainable and scalable software applications. It stands for Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP). Angular is a popular open-source framework for building web applications, and it can be used to implement SOLID design patterns.

Single Responsibility Principle (SRP)

The SRP states that a class should have only one reason to change. In Angular, this can be implemented by creating small, focused components that do one thing well. For example, a component that displays a list of items should only be responsible for displaying the list, and not for fetching the data or updating the items.

typescript
```
@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
})
export class ItemListComponent {
  @Input() items: Item[];

  constructor() { }
}
```

Open-Closed Principle (OCP)

The OCP states that a class should be open for extension but closed for modification. In Angular, this can be implemented by using interfaces and abstract classes to define behavior, and then implementing those interfaces or extending those classes in concrete classes. For example, if we have a service that fetches data from a backend API, we can define an interface for the service and then create concrete implementations of that interface for each backend API we need to support.

typescript
```
interface DataService {
  getData(): Observable<any>;
}

@Injectable({
  providedIn: 'root'
})
export class ApiService implements DataService {
  constructor(private http: HttpClient) {}

  getData(): Observable<any> {
    return this.http.get('/api/data');
  }
}
```

Liskov Substitution Principle (LSP)

The LSP states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In Angular, this can be implemented by using inheritance to create subclasses that inherit behavior and properties from a superclass. For example, if we have

a base component that displays a list of items, we can create a child component that extends the base component and adds additional behavior or styling.

typescript
```
@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
})
export class ItemListComponent {
  @Input() items: Item[];

  constructor() { }
}

@Component({
  selector: 'app-special-item-list',
  templateUrl: './special-item-list.component.html',
})
export class SpecialItemListComponent extends ItemListComponent {
  constructor() {
    super();
  }

  getSpecialItems(): Item[] {
    // implementation details
  }
}
```

Interface Segregation Principle (ISP)

The ISP states that a class should not be forced to depend on methods it does not use. In Angular, this can be implemented by creating small, focused interfaces that define behavior for a specific use case. For example, if we have a service that manages user authentication, we can create separate interfaces for login, logout, and user information retrieval.

typescript
```
interface LoginService {
  login(username: string, password: string): Observable<boolean>;
}

interface LogoutService {
  logout(): Observable<boolean>;
}

interface UserService {
  getUserInfo(): Observable<User>;
}

@Injectable({
  providedIn: 'root'
})
export class AuthService implements LoginService, LogoutService, UserService {
  // implementation details
}
```

Dependency Inversion Principle (DIP)

The DIP states that high-level modules should not depend on low-level modules, but should depend on abstractions. In Angular, this can be implemented by using dependency injection to provide dependencies to a class or component. For example, if we have a component that needs to make HTTP requests, we can inject an HTTP service into that component instead of creating a new instance of the service inside the component.

typescript

```
@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
})
export class ItemListComponent {
  items: Item[];

  constructor(private http: HttpClient) { }

  ngOnInit() {
    this.http.get('/api/items').subscribe((response) => {
      this.items = response;
    });
  }
}
```

By following these SOLID design principles in Angular, we can create maintainable and scalable applications that are easy to modify and extend over time.

Pros of using Solid Design Principles

Increased Maintainability

Increased maintainability, in the context of SOLID design principles, refers to the ability of developers to modify and update the codebase easily without introducing bugs or breaking existing functionality. By following SOLID principles, developers create code that is more modular, reusable, and easier to understand.

For example, by following the Single Responsibility Principle (SRP), each class or component in the application has a single responsibility. This means that it is easier to modify or update a single component without affecting other parts of the codebase. This makes it easier for developers to maintain the codebase over time, since changes can be made more quickly and with less risk of introducing bugs or breaking existing functionality.

Similarly, by following the Open-Closed Principle (OCP), developers create code that is open for extension but closed for modification. This means that when new requirements or features are added to the application, developers can extend the codebase without modifying the existing code. This makes it easier to maintain the codebase over time, since changes can be made without affecting existing functionality.

Overall, increased maintainability means that the codebase is easier to modify and update over time, which reduces the risk of introducing bugs or breaking existing functionality. This can save developers time and effort, and make it easier to maintain the codebase over the long term.

Improved Scalability

Improved scalability, in the context of SOLID design principles, refers to the ability of the application to handle increases in load, users, and data without a corresponding increase in complexity or difficulty to maintain the application. By following SOLID principles, developers create code that is more modular, loosely coupled, and reusable.

For example, by following the Dependency Inversion Principle (DIP), components in the application depend on abstractions rather than concrete implementations. This makes it easier to replace or update dependencies without affecting other parts of the codebase. This makes it easier for developers to scale the application by adding or replacing components without needing to modify the existing codebase.

Similarly, by following the Interface Segregation Principle (ISP), components only depend on the interfaces that they actually use, rather than on a larger interface that includes unnecessary methods. This reduces the dependencies between components and makes it easier to modify or replace them without affecting other parts of the codebase. This makes it easier for developers to scale the application by modifying or replacing components as needed.

Overall, improved scalability means that the application can handle increases in load, users, and data without a corresponding increase in complexity or difficulty to maintain the application. This can save developers time and effort, and make it easier to scale the application over the long term.

Better Testability

Better testability, in the context of SOLID design principles, refers to the ability of the application to be easily tested using automated testing tools. By following SOLID principles, developers create code that is more modular, loosely coupled, and decoupled from external dependencies.

For example, by following the Single Responsibility Principle (SRP), each component in the application has a single responsibility, which makes it easier to write tests for that component. This is because each test can focus on a single responsibility of the component, making it easier to isolate and identify any issues or bugs. Additionally, by following the Dependency Inversion Principle (DIP), components in the application depend on abstractions rather than concrete implementations, which makes it easier to mock or stub dependencies when testing the component.

Similarly, by following the Interface Segregation Principle (ISP), components only depend on the interfaces that they actually use, which reduces the dependencies between components and makes it easier to mock or stub those dependencies when testing the component. This makes it easier to write tests for individual components without needing to test the entire application.

Overall, better testability means that the application can be more easily tested using automated testing tools. This reduces the amount of manual testing that needs to be done and increases the speed and accuracy of testing. This can save developers time and effort, and make it easier to maintain the application over the long term.

Increased Reusability

Increased reusability, in the context of SOLID design principles, refers to the ability of the code to be reused in different parts of the application or in other applications. By following SOLID principles, developers create code that is more modular, loosely coupled, and decoupled from external dependencies.

For example, by following the Single Responsibility Principle (SRP), each component in the application has a single responsibility, which makes it easier to reuse that component in other parts of the application. This is because the component can be easily extracted and reused without needing to modify or adapt it for the new context. Additionally, by following the Open-Closed Principle (OCP), components in the application are open for extension but closed for modification, which means that they can be extended or customized without needing to modify the existing code.

Similarly, by following the Dependency Inversion Principle (DIP), components in the application depend on abstractions rather than concrete implementations, which makes it easier to replace or update dependencies without affecting other parts of the codebase. This makes it easier to reuse components in other applications, since the dependencies can be easily replaced with new ones that are compatible with the new application.

Overall, increased reusability means that the code can be more easily reused in different parts of the application or in other applications. This reduces the amount of code that needs to be written from

scratch and increases the efficiency and speed of development. This can save developers time and effort, and make it easier to maintain and scale the application over the long term.

Cons of using Solid Design Principles

Increased Complexity

SOLID principles can sometimes lead to more complex code, especially if they are not implemented correctly. This can make it harder for developers to understand the code and make changes to it.

Increased Development Time

Increased development time, in the context of SOLID design principles, refers to the potential increase in the time and effort required to write code that follows SOLID principles. This is because SOLID principles require developers to spend more time designing and planning the architecture of the application, and to write code that is more modular, loosely coupled, and decoupled from external dependencies.

For example, by following the Single Responsibility Principle (SRP), developers may need to spend more time identifying and separating responsibilities for each component of the application. Similarly, by following the Dependency Inversion Principle (DIP), developers may need to spend more time designing and implementing interfaces and abstractions for components.

While this increased development time may seem like a disadvantage, it is important to note that the long-term benefits of SOLID design principles can outweigh the short-term costs. By following SOLID principles, developers create code that is more maintainable, scalable, and testable, which can save time and effort in the long run by reducing the need for extensive manual testing, refactoring, and bug fixing.

Additionally, SOLID principles can help to mitigate the risk of technical debt, which is the accumulation of unaddressed technical issues in the codebase over time. By designing code that is more modular and loosely coupled, developers can reduce the risk of technical debt and make it easier to maintain and evolve the codebase over time.

Overall, while following SOLID design principles may increase development time in the short term, it can result in significant long-term benefits in terms of maintainability, scalability, and testability of the codebase.

Over-Engineering

Over-engineering, in the context of SOLID design principles, refers to the potential risk of creating a solution that is more complex than necessary to solve the problem at hand. This can happen when developers try to apply SOLID principles too rigorously, without considering the specific needs and requirements of the application.

For example, over-engineering can happen when a developer tries to apply the Open-Closed Principle (OCP) too rigidly, creating abstractions and interfaces that are too generic or abstract to be useful in the specific context of the application. Similarly, over-engineering can happen when a developer tries to apply the Dependency Inversion Principle (DIP) too broadly, creating complex dependency injection frameworks or using design patterns that are not well-suited for the specific needs of the application.

Over-engineering can be a problem because it can increase the complexity and maintenance burden of the codebase without providing any significant benefits in terms of scalability, maintainability, or

testability. In some cases, over-engineering can actually make the code more difficult to understand and modify, leading to higher development and maintenance costs over time.

To avoid over-engineering, it is important for developers to apply SOLID design principles in a pragmatic and context-sensitive way, taking into account the specific needs and requirements of the application. This may involve making trade-offs between the idealized design principles and the practical realities of the application, and finding a balance between simplicity, maintainability, and flexibility.

Steep Learning Curve

The "steep learning curve" in the context of SOLID design principles refers to the difficulty that developers may initially face when learning and applying these principles to their code. SOLID principles can require a significant shift in mindset and coding practices, which can be challenging for developers who are used to more traditional approaches to software design.

For example, developers who are not familiar with SOLID principles may find it difficult to identify and separate concerns within their codebase, or to create interfaces and abstractions that can be easily swapped out or extended. Similarly, developers who are not used to writing unit tests or working with dependency injection may find it difficult to adopt these practices as part of their workflow.

The steep learning curve of SOLID principles can be a disadvantage, especially in the short term, as it can increase the time and effort required to write and maintain code. However, it is important to note that this learning curve can be overcome with practice and experience. Moreover, the long-term benefits of SOLID principles, such as improved code quality, maintainability, and testability, can far outweigh the initial investment in learning and applying these principles.

To overcome the steep learning curve of SOLID principles, developers can start by familiarizing themselves with the basic concepts and terminology, and by practicing these principles in small, isolated code examples. They can also seek out training and mentorship from experienced developers who have already mastered SOLID principles, and participate in online communities and forums to ask questions and get feedback on their code. With time and practice, developers can develop the skills and knowledge needed to apply SOLID principles effectively and efficiently in their work.

Summary

In summary, SOLID design principles are a set of best practices for writing maintainable, scalable, and testable software. They aim to reduce complexity in software design by encouraging developers to write code that is modular, loosely coupled, and easy to understand. By following SOLID principles, developers can create code that is easier to maintain, scale, and test, and that can evolve over time to meet changing business needs and requirements.

While SOLID principles may require increased development time in the short term, the long-term benefits of reduced technical debt and improved code quality can outweigh these costs. However, there is a risk of over-engineering when applying SOLID principles too rigidly, so it's important for developers to apply them in a pragmatic and context-sensitive way. Overall, SOLID design principles provide a powerful framework for creating high-quality, maintainable software that can adapt to changing business needs and requirements.

SOLID: The Dependency Inversion Principle in Angular

Get a better understanding of the DI principle and how it is implemented in Angular.

Chidume Nnamdi 🔥💻🎵🎮

.

Follow

Published in

Bits and Pieces

.
8 min read
·
Aug 28, 2019

A. HIGH-LEVEL MODULES SHOULD NOT DEPEND UPON LOW-LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

This principle states that classes and modules should depend on abstractions not on concretions.

The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.

Tip: Use Bit to get the most out of your SOLID Angular project

SOLID code is modular and reusable. With **Bit,** you can easily **share and organize your reusable components.** Let your team see what you've been working on, install and reuse your components across projects, and even collaborate together on individual components. Give it a try.

Share reusable code components as a team · Bit

Easily share reusable components between projects and applications to build faster as a team. Collaborate to develop…

bit.dev

What are Abstractions?

Abstractions are Interfaces. Interfaces define what implementing Classes must-have. If we have an Interface Meal:

```
interface Meal {
    type: string
}
```

This holds information on what type of meal is being served; Breakfast, Lunch or Dinner. Implementing classes like `BreakFastMeal`, `LunchMeal` and `DinnerMeal` must have the `type` property:

```
class BreakFastMeal implements Meal {
    type: string = "Breakfast"
}class LunchMeal implements Meal {
    type: string = "Lunch"
}class DinnerMeal implements Meal {
    type: string = "Dinner"
}
```

So, you see Interface gives the information on what properties and methods the class that implements it must have. An Interface is called an Abstraction because it is focused on the characteristic of a Class rather than the Class as a whole group of characteristics.

What are Concretions?

Concretions are Classes. They are the opposite of Abstractions, they contain the full implementation of their characteristics. Above we stated that the interface `Meal` is an abstraction, then the classes that implemented it, `DinnerMeal`, `BreakfastMeal` and `LunchMeal` are the concretions, because they contain the full implementation of the `Meal` interface. `Meal` has a characteristic `type` and said it should be a string type, then the `BreakfastMeal` came and said the `type` is `"Breakfast"`, `LunchMeal` said the `type` is `"Lunch"`.

The DIP says that if we depend on Concretions, it will make our class or module tightly coupled to the detail. The coupling between components results in a rigid system that is hard to change, and one that fails when changes are introduced.

Example: Copier

Let's use an example to demonstrate the effects of using the DIP. Let's say we have a program that gets input from a disk and copies the content to a flash drive.

The program would read a character from the disk and pass it to the module that will write it to the flash drive.

The source will look like this:

```
function Copy() {
    let bytes = []
    while(ReadFromDisk(bytes))
        WriteToFlashDrv(bytes)
}
```

Yes, that's a work well done, but this system is rigid, not flexible. The system is restricted to only reading from a disk and writing to a flash drive. What happens when the client wants to read from a disk and write to a network? We will see ourself adding an if statement to support the new addition

```
function Copy(to) {
    let bytes = []
    while(ReadFromDisk(bytes))
        if(to == To.Net)
            WriteToNet(bytes)
        else
            WriteToFlashDrv(bytes)
}
```

See we *touched* the code, which shouldn't be so. As time goes on, and more and more devices must participate in the copy program, the `Copy` function will be littered with `if/else` statements and will be dependent upon many lower-level modules. It will eventually become rigid and fragile.

To make the Copy function reusable and less-fragile, we will implement interfaces Writer and Reader so that any place we want to read from will implement the Reader interface and any place we want to write to will implement the Write interface:

```
interface Writer {
    write(bytes)
}interface Reader {
    read(bytes)
}
```

Now, our disk reader would implement the Reader interface:

```
class DiskReader implements Reader {
    read(bytes) {
        //.. implementation here
    }
}
```

then, network writer and flash drive writer would both implement the Writer interface:

```
class Network implements Writer {
    write(bytes) {
        // network implementation here
```

```
    }
}class FlashDrv implements Writer {
    write(bytes) {
        // flash drive implementation
    }
}
```

The Copy function would be like this:

```
function Copy(to) {
    let bytes = []
    while(ReadFromDisk(bytes))
        if(to == To.Net)
            WriteToNet(bytes)
        else
            WriteToFlashDrv(bytes)
}
|
|
vfunction Copy(writer: Writer, reader: Reader) {
    let bytes = []
    while(reader.read(bytes))
        writer.write(bytes)
}
```

See, our Copy has been shortened to a few codes. The Copy function now depends on interfaces, all it knows is that the Reader will have a read method it would call to write bytes and a Reader with a read method where it would get bytes to write, it doesn't concern how to get the data, it is the responsibility of the class implementing the Writer.

This makes the Copy function highly re-usable and less-fragile. We can pass any Reader or Writer to the Copy function, all it cares:

```
// read from disk and write to flash drive
Copy(FlasDrvWriter, DiskReader)// read from flash and write to disk
Copy(DiskWriter, FlasDrvReader)// read from disk and write to network ie.
uploading
Copy(NetworkWriter, DiskReader)// read from network and write to disk ie.
downloading
Copy(DiskWriter, NetworkReader)
```

Example: Nodejs `Console class`

The Nodejs `Console` class is an example of a real-world app that obeys the DIP. The `Console` class produces output, yeah majorly used to output to a terminal, but it can be used to output to other media like:

- file
- network

When we do console.log("Nnamdi")

`Nnamdi` is printed to the screen, we can channel the output to another place like we outlined above.

Looking at the Console class

```
function Console(stdout, stderr) {
    this.stdout = stdout
    this.stderr = stderr ? stderr : stdout
}Console.prototype.log = function (whatToWrite) {
    this.write(whatToWrite, this.stdout)
```

```
}Console.prototype.error = function (whatToWrite) {
    this.write(whatToWrite, this.stderr)
}Console.prototype.write = function (whatToWrite, stream) {
    stream.write(whatToWrite)
}
```

It accepts a `stdout` and `stderr` which are streams, they are generic, the stream can be a terminal or file or anywhere like network stream. `stdout` is where to write out, the `stderr` is where it write any error. The console object we have globally has already been initialized with stream set to be written to terminal:

```
global.console = new Console(process.stdout, process.stderr)
```

The stdout and stderr are interfaces that have the write method, all that Console knows is to call the write method of the stdout and stderr.

The Console depends on abstracts stdout and stderr, it is left for the user to supply the output stream and must have the write method.

To make the Console class write to file we simply create a file stream:

```
const fsStream = fs.createWritestream('./log.log')
```

Our file is `log.log`, we created a writable stream to it using fs's createWriteStream API.

We can create another stream we can log our error report:

```
const errfsStream = fs.createWritestream('./error.log')
```

We can now pass the two streams to the Console class:

```
const log = new Console(fsStream, errfsStream)
```

When we call `log.log("logging an input to ./log.log")`, it won't print it to the screen, rather it will write the message to the `./log.log` file in your directory.

Simple, the Console does not have to have a long chain of if/else statement to support any stream.

Angular

Coming to Angular how do we obey the DIP?

Let’s say we have a billing app that lists peoples license and calculates their fees, our app may look like this:

```
@Component({
    template: `
        <div>
            <h3>License</h3>
            <div *ngFor="let p of people">
                <p>Name: {{p.name}}</p>
                <p>License: {{p.licenseType}}</p>
                <p>Fee: {{calculateFee(p)}}</p>
            </div>
        </div>
    `
})
export class App {
    people = [
```

```
        {
            name: 'Nnamdi',
            licenseType: 'personal'
        },
        {
            name: 'John',
            licenseType: 'buisness'
        },
        // ...
    ]    constructor(private licenseService: LicenseService) {}
calculateLicenseFee(p) {
        return this.licenseService.calculateFee(p)
    }
}
```

We have a Service that calculates the fees based on the license:

```
@Injectable()
export class LicenseService {
    calculateFee(data) {
        if(data.licenseType == "personal")
             //... calculate fee based on "personal" licnese type
        else
         //... calculate fee based on "buisness" licnese type
    }
}
```

This Service class violates the DIP, when another license type is introduced we will see ourself adding another if statement branch to support the new addition:

```
@Injectable()
export class LicenseService {
    calculateFee(data) {
        if(data.licenseType == "personal")
             //... calculate fee based on "personal" licnese type
        else if(data.licenseType == "new license type")
            //... calculate the fee based on "new license type" license type
        else
            //... calculate fee based on "buisness" licnese type
    }
}
```

To make it obey the DIP, we will create a License interface:

```
interface License {
    calcFee():
}
```

Then we can have classes that implement it like:

```
class PersonalLicense implements License {
    calcFee() {
        //... calculate fee based on "personal" licnese type
    }
    // ... other methods and properties
}class BuisnessLicense implements License {
    calcFee() {
        //... calculate fee based on "buisness" licnese type
    }
    // ... other methods and properties
}
```

Then, we will refactor the `LicenseService` class:

```
@Injectable()
export class LicenseService {
    calculateFee(data: License) {
        return data.calcFee()
    }
}
```

It accepts data which is a `License` type, now we can send any license type to the `LicenseService#calculateFee`, it does not care about the type of license, it just knows that the data is a `License` type and calls its `calcFee` method. It is left for the class that implements the `License` interface to provide its license fee calculation in the `calcFee` method.

Angular itself also obeys the DIP, in its source. For example in the Pipe concept.

Pipe

Pipe is used to transform data without affecting the source. In array, we transform data like:

- `map`ping
- `filter`ing
- `sort`ing
- `splic`ing
- `slic`ing
- `substr`ing *wink emoji here*
- etc

All these transform data based on the implementation.

In Angular templates, if we did not have the Pipe interface, we would have classes that transform data `pipe` like the Number, Date, JSON or custom pipe, etc. Angular would have its implementation of Pipe like this:

```
pipe(pipeInstance) {
    if (pipeInstance.type == 'number')
        // transform number
    if(pipeInstance.type == 'date')
        // transform date
}
```

The list would expand if Angular adds new pipes and it would be more problematic to support custom pipes.

So to Angular created a PipeTransform interface that all pipes would implement:

```
interface PipeTransform {
    transform(data: any)
}
```

Now any Pipe would implement the interface and provides its piping function/algorithm in the transform method.

```
@Pipe(...)
class NumberPipe implements PipeTransform {
    transform(num: any) {
        // ...
    }
}@Pipe(...)
class DatePipe implements PipeTransform {
    transform(date: any) {
```

```
        // ...
    }
}@Pipe(...)
class JsonPipe implements PipeTransform {
    transform(jsonData: any) {
        // ...
    }
}
```

Now, Angular would call the transform without bothering about the type of the Pipe

```
function pipe(pipeInstance: PipeTransform, data: any) {
    return pipeInstance.transform(data)
}
```

Conclusion

We saw in this post how DIP makes us write reusable and maintainable code in Angular and in OOP as a whole.

In ***Engineering Notebook columns for The C++ Report*** in *The Dependency Inversion Principle* column, it says:

A piece of software that fulfills its requirements and yet exhibits any or all of the following three traits has a bad design.

1. It is hard to change because every change affects too many other parts of the system. (Rigidity)
2. When you make a change, unexpected parts of the system break. (Fragility)
3. It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)

If you have any question regarding this or anything I should add, correct or remove, feel free to comment, email or DM me

Thanks !!