Angular Design Patterns:

Strategy Pattern

The Strategy pattern is a behavioral design pattern that provides a mechanism to select an algorithm at runtime from a family of algorithms, and make them interchangeable. In context of Angular, you can think algorithms as services here but it can also be used for components and classes.

Key benefits

Purpose: To define a family of interchangeable algorithms and allow the client to choose dynamically which one to use at runtime. This provides flexibility.

Encapsulation: Each strategy is encapsulated as a separate class, which helps keep the code clean and organized. The client code does not know the details of how each strategy is implemented, as it only interacts with the common interface.

Composition over inheritance: Strategy pattern is based on composition to achieve behavior reuse instead of relying on inheritance. This leads to more flexible designs.

Useful Scenarios 💎

When you have multiple algorithms or approaches that can be used interchangeably to solve a problem.

Classic Scenarios

Navigation app: A navigation might use different routing strategies for cars, pedestrians, or cyclists.

Sorting algorithms: Different sorting algorithms (quicksort, bubble sort, merge sort) can be implemented as strategies, letting you choose the most suitable one at runtime.

An important advantage of the Strategy patttern is to provide users to choose different strategies at runtime. ✨

Glossary 🌍

Glossary is going to make more sense when we visit our example.

1. Context:

The Context has a reference to one of the concrete strategies and which is used by its other methods. To achieve it, the context class usually has a public method called setStategy(stategy) which is set by client. That reference is type of Strategy Interface. The context does not know which strategy it uses, the key point is that strategy should implement Strategy Interface.

2. Strategy interface:

It is common to all concrete strategies. It defines a blueprint which is implemented by concrete strategies.

3. Concrete Strategies:

Concrete Strategies are implementations of algorithms that is used by client and used by context. They implement Strategy Interface.

4. The Client: initializes a specific strategy object and passes it to the context via setStategy(stategy)

Examples

Example 1: Shipping app

Assume you have e-commerce application. You want to provide shipping information based on client's preference.

In the context of this post, you want to provide interchangeable strategies to choose from.

Problem

Let’s first examine naive/brute force approach: ⬇️

Source Code v1

https://github.com/vugar005/angular-design-patterns/blob/main/src/app/strategy-pattern/shipping/shipping-v1/shipping-v1.component.ts

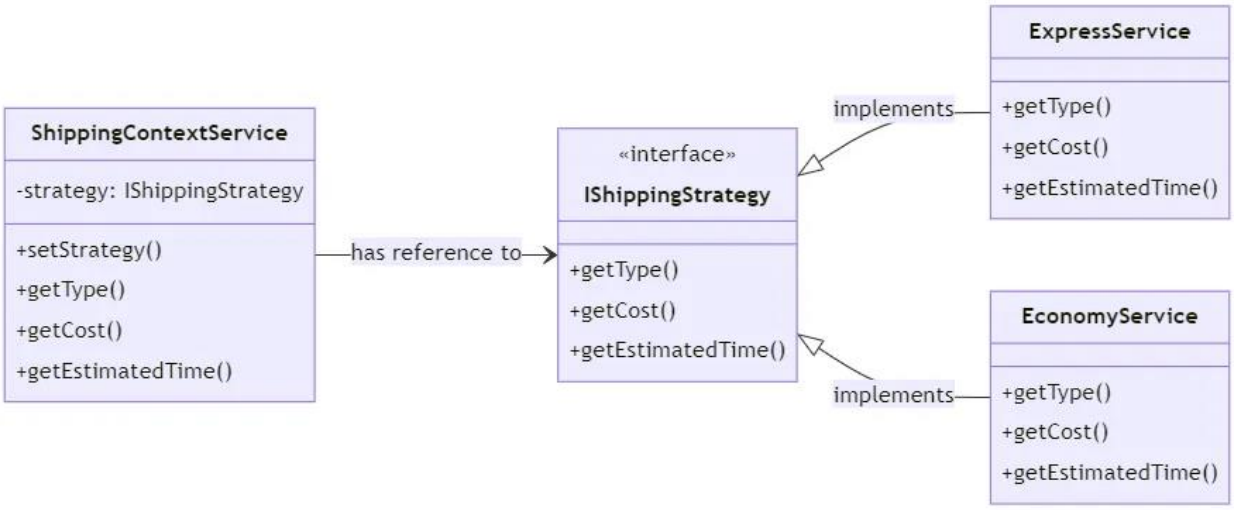
```
export class ShippingV1Component {
  public readonly shippingOptions = ['EXPRESS', 'ECONOMY'];
  public selectedOption!: string;
  public type?: string;
  public cost?: string;
  public estimatedTime?: string;

  constructor(
    private readonly expressShippping: ExpressShippingService,
    private readonly economyShipping: EconomyShippingService
  ) {}

  public onStrategyChange(option: string): void {
    this.selectedOption = option;
    this.getData(option);
  }
```

```
  private getData(option: string): void {
    if (option === 'EXPRESS') {
      this.type = this.expressShippping.getType();
      this.cost = this.expressShippping.getCost();
      this.estimatedTime = this.expressShippping.getEstimatedTime();
    } else if (option === 'ECONOMY') {
      this.type = this.economyShipping.getType();
      this.cost = this.economyShipping.getCost();
      this.estimatedTime = this.economyShipping.getEstimatedTime();
    }
  }
}
```

As you see on getData method we already have undesired condition. This can get more complicated. What if we introduce new shipping services such as Sea Shipping and so on? We are going to have more complexity. 😕

Solution: 🛠

The diagram below shows how our final implementation should look like:



Step 1: Define a strategy interface for concrete strategies

The Shipping Strategy interface is common to all variants of the strategies.

```
export interface IShippingStrategy {
    getType: () => string;
    getCost: () => string;
    getEstimatedTime: () => string;
}
```

Step 2: Create concrete strategies which implements

`IShippingStrategy`

```
@Injectable({
  providedIn: 'root',
})
export class EconomyShippingService implements IShippingStrategy {
  public getType(): string {
    return 'ECONOMY';
  }

  public getCost(): string {
    return '15$';
  }

  public getEstimatedTime(): string {
    return '5-12 days';
  }
}
```

and

```
@Injectable({
  providedIn: 'root'
})
export class ExpressShippingService implements IShippingStrategy {
  public getType(): string {
    return 'EXPRESS';
  }

  public getCost(): string {
    return '100$';
  };

  public getEstimatedTime(): string {
    return '1-2 days';
  };
}
```

Step 3: Create a **context service** which has reference to concrete strategies.

```
@Injectable({
  providedIn: 'root',
})
export class ShippingContextService implements IShippingStrategy {
  private strategy!: IShippingStrategy;

  public hasChosenStrategy(): boolean {
    return !!this.strategy;
  }

  public setStrategy(strategy: IShippingStrategy): void {
    this.strategy = strategy;
  }

  public getType(): string {
    return this.strategy.getType();
  }

  public getCost(): string {
    return this.strategy.getCost();
  }

  public getEstimatedTime(): string {
    return this.strategy.getEstimatedTime();
  }
}
```

The `setStrategy` method is called by client.

*The context is **not limited** to `IShippingStrategy` interface. You can also have specific methods that is shared by all concrete stategies.*

Step 4: Allow client to choose preferred strategy.

Our new ShippingV2Component:

Source Code v2

https://github.com/vugar005/angular-design-patterns/blob/main/src/app/strategy-pattern/shipping/shipping-v2/shipping-v2.component.ts

```
export class ShippingV2Component {
  public readonly shippingOptions = ['EXPRESS', 'ECONOMY'];
  public selectedOption!: string;
  public type?: string;
  public cost?: string;
  public estimatedTime?: string;

  constructor(
    private readonly injector: Injector,
    private readonly shippingContext: ShippingContextService
  ) {}

  public onStrategyChange(option: string): void {
    this.selectedOption = option;

    switch (option) {
      case 'EXPRESS': {
        const strategy = this.injector.get(ExpressShippingService);
        this.shippingContext.setStrategy(strategy);
        break;
      }

      case 'ECONOMY': {
        const strategy = this.injector.get(EconomyShippingService);
        this.shippingContext.setStrategy(strategy);
        break;
      }
    }
```

```
    this.getData();
  }

  private getData(): void {
    if (!this.shippingContext.hasChosenStrategy) {
      return;
    }
    this.type = this.shippingContext.getType();
    this.cost = this.shippingContext.getCost();
    this.estimatedTime = this.shippingContext.getEstimatedTime();
  }
}
```

As you noticed we use condition only once at `onStrategyChange` when the Client chooses shipping option.

That's it 😳

Factory Pattern

Intro

The Factory pattern is **creational design pattern** that provides a mechanism to **outsource object creation** to either subclass or seperate class called factory class.

Source Code

https://github.com/vugar005/angular-design-patterns/tree/main/src/app/factory-pattern

Enhanced markdown version

https://vugar.app/posts/angular-design-patterns-factory

Key Benefits

- **Purpose:** Centralizes object creation, providing a **flexible/dynamic** way to create objects without exposing the details of their concrete classes to the rest of your application. The objects can refer to classes/components/services.
- **Flexibility:** It enables you to **dynamically** decide, **at runtime**, which objects to instantiate.
- **Encapsulation:** Hides object instantiation details from the client code. Clients interact with a factory to obtain objects, **rather than directly instantiating** concrete classes. This promotes loose coupling and makes code more adaptable to changes.

Useful Scenarios 💎

- You **don't know in advance** the exact objects/services your application will need **at runtime**
- When you want to provide a mechanism for users to add their own object types.

An important advantage of the Factory Pattern is its ability to instantiate objects **dynamically at runtime.** ✨

Glossary 🌍

Glossary is going to make more sense when we visit our example.

1. Concrete Factory Class:

The factory class is responsible for **creating objects**. The objects can refer to classes/components/services.

Consists of **factory method**.

2. Concrete Factory Method :

Factory Method is a method **within** a **factory class** actually returns objects based on **conditional statements**.

3 .Concrete Product:

The objects/components/services that are created/returned by **factory method.**

The **type** of Concrete Product should be **same or extended** type of Product returned by **Concrete Factory Method**.

You can also add **Abstract Factor class, Abstract Factory Method** and **Product interface** for each of those but it is generally preferred to use **interfaces** so I did not add them here separately.

Examples:

Example 1: Online/Offline note app:

Assume you have note taking application. Currently, application uses **online** APIs to save and update data.

Now you would like to add **offline** capabilities for user to save notes when internet is unavailable.

When it is online the notes would be saved via API, if offline the notes would be saved on local storage.

As you noticed, we want **dynamically** switch the way the client updates notes.

Problem:

Let's examine naive/brute force approach: ⬇️

```
@Injectable({
  providedIn: 'root',
})
export class NoteService {
  constructor(
    private readonly onlineNoteApi: OnlineNoteApiService,
    private readonly offlineNoteApi: OfflineNoteApiService
  ) {}

  public saveNote(data: INote): void {
    if (this.isOnline) {
      this.onlineNoteApi.save(data);
    } else {
      this.offlineNoteApi.save(data);
    }
  }

  private get isOnline(): boolean {
```

```
    return window.navigator.onLine;
  }
```

As you see on `saveNote` method we already have undesired condition. This can get more complicated. What if we **introduce new API** such as Fast API and so on? We are going to have more complexity. 😳

Solution: 🛠

Step 1: Create Interface for concrete products:

```
export interface INoteApiService {
  save: (note: INote) => void;
}
```

Step 2: Implement this `INoteApiService` on **Concrete Products:**

```
@Injectable({
  providedIn: 'root',
})
export class OnlineNoteApiService implements INoteApiService {
  public save(data: INote): void {
    console.log('Save Online');
  }
}

@Injectable({
  providedIn: 'root',
})
export class OfflineNoteApiService implements INoteApiService {
```

```
  public save(data: INote): void {
    console.log('Save Offline');
  }
}
```

Step 3: Create a **Concrete Factory Class**:

```
@Injectable({
  providedIn: 'root',
})
class NoteServiceFactory {
  constructor(private readonly injector: Injector) {}

  public getNoteService(): INoteApiService {
    if (window.navigator.onLine) {
      return this.injector.get(OnlineNoteApiService);
    } else {
      return this.injector.get(OfflineNoteApiService);
    }
  }
}
```

Step 4: Use `NoteServiceFactory` on `NoteService`:

Our refactored NoteService:

```
@Injectable({
  providedIn: 'root',
})
export class NoteService {
  constructor(private readonly noteServiceFactory:
NoteServiceFactory) {}
  public saveNote(data: INote): void {
    const noteApiService = this.createNoteService();
    noteApiService.save(data);
  }

  private createNoteService(): INoteApiService {
    return this.noteServiceFactory.getNoteService();
```

```
  }
}
```

That's it 😳

Warning

Do you notice the issue here? On every call we call `createNoteService` *which creates and returns* ***new reference necessarily****. This can cause memory leak. To improve it we can create service* ***on demand*** *when there is a change in network. This solution below resembles* ***Strategy*** *pattern.*

Updated solution:

```
@Injectable({
  providedIn: 'root',
})
class NoteServiceFactoryV2 {
  private currentService!: INoteApiService;

  constructor(private readonly injector: Injector) {
    this.setApiService(window.navigator.onLine);
    this.listenToConnection();
  }

  public getNoteService(): INoteApiService {
    return this.currentService;
  }
```

```
  private listenToConnection(): void {
    merge(
      fromEvent(window, 'online').pipe(map(() => true)),
      fromEvent(window, 'offline').pipe(map(() => false)),
      fromEvent(document, 'DOMContentLoaded').pipe(map(() =>
navigator.onLine)) // current status
    )
      .pipe(startWith(window.navigator.onLine))
      .subscribe((isOnline: boolean) => {
        console.log('isOnline', isOnline);
        this.setApiService(isOnline);
      });
  }

  private setApiService(isOnline: boolean): void {
    if (isOnline) {
      this.currentService = this.injector.get(OnlineNoteApiService);
    } else {
      this.currentService =
this.injector.get(OfflineNoteApiService);
    }
  }
}
```

Example 2: Dynamic Notification provider

Problem:

Imagine you're developing a UI library that needs to support multiple versions of notification components, such as Material Design V1 and Material Design V2. You want users to be able to choose between these versions dynamically, possibly to compare them within the same application environment.

Solution: 🛠

Similarly,in this scenario, we can create Factory Class `NotificationServiceFactory` and concrete products `NotificationServiceV1` and `NotificationServiceV2` which implement `INotificationService`.

Since it follows same steps as in Example 1, I skip implementation of it.

Why not use Angular's `useFactory` for services?

You might be curious why we don't just use Angular's built-in factory service provider, specifically the `useFactory` option, for creating our notification services? In that case it would inject service **statically** but in our scenario we are interested in **dynamic** injection of services.

Tip

*If you are interested in **more flexible** injection of services, I would recommended to use **Strategy Pattern** rather than Factory Pattern.*

That's all hope you enjoyed it and found it useful. Thanks for reading. 🍍

References:

https://refactoring.guru/design-patterns/factory-method

https://medium.com/@baranoffei/angulars-gof-patterns-factory-method-0c740fbd3b2e

https://chat.openai.com/

https://gemini.google.com

Java Design Pattern

Builder Design Pattern

https://medium.com/javarevisited/builder-design-pattern-in-java-3b3bfee438d9

Adapter Pattern in Java: A

Structural Design Pattern

https://medium.com/@Neelesh-Janga/adapter-pattern-in-java-a-structural-design-pattern-b3d227dcb6c9

Facade Design Pattern || Java

https://medium.com/@ngneha090/facade-design-pattern-java-79a29904808d **Strategy**

Design Pattern — All You Need

To Know

https://medium.com/@basecs101/strategy-design-pattern-all-you-need-to-know-updated-2024-e8a8303bd49e

Factory Design Pattern And

Abstract Factory Design

Pattern

https://medium.com/@adarsh_tech/factory-design-pattern-and-abstract-factory-design-pattern-c3ead03c5c3c