Návrhové vzory

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

04.12.2025

Úvod

Co jsou návrhové vzory?

Co si představíte pod pojmem **návrhový vzor**?

Setkali jste se již s nějakým návrhovým vzorem?

Co jsou návrhové vzory?

Návrhové vzory (Design Patterns) jsou osvědčená řešení běžných problémů v návrhu softwaru

- Představují **šablony** pro řešení opakujících se problémů
- Jsou to **abstraktní popisy**, ne konkrétní implementace
- Byly poprvé systematicky popsány v knize Design Patterns: Elements of Reusable Object-Oriented Software (1994) („Gang of Four“)
- Pomáhají vytvářet **udržovatelný** a **škálovatelný** kód

Co jsou návrhové vzory?

Proč jsou návrhové vzory důležité?

- **Znovupoužitelnost** - osvědčená řešení
- **Komunikace** - společný jazyk mezi vývojáři
- **Udržovatelnost** - předvídatelná struktura kódu
- **Škálovatelnost** - snadné rozšiřování funkcionality
- **Testovatelnost** - lepší struktura = snazší testování

Kde se návrhové vzory používají?

Návrhové vzory se používají prakticky všude:

- **Frameworky a knihovny** - React, Angular, Vue, Express.js, NestJS, …
- **Databáze** - ORM jako TypeORM, Prisma, …
- **State management** - Redux, MobX
- **DOM manipulace** - jQuery internals, Alpine.js, …

Kde se návrhové vzory používají?

Příklad z reálného světa:

```
// React uses Observer pattern for state management
const [count, setCount] = useState(0);

// Express.js uses Middleware pattern
app.use(express.json());

// TypeORM uses Repository pattern
const userRepository = dataSource.getRepository(User);
```

Kategorie návrhových vzorů

1. **Creational** (Vytvářející)
   - Řeší vytváření objektů
   - Factory, Builder, Singleton
2. **Structural** (Strukturální)
   - Řeší kompozici tříd a objektů
   - Adapter, Decorator, Composite
3. **Behavioral** (Behaviorální)
   - Řeší komunikaci mezi objekty
   - Observer, Iterator, Strategy

Creational Patterns

Factory Pattern

Factory Pattern

Jaký problém řeší **Factory Pattern**?

Jak byste vytvořili objekty různých typů bez znalosti jejich konkrétních tříd?

Factory Pattern

Představte si **výrobní linku v továrně**

- Zákazník si objedná produkt (např. „červené auto“)
- Továrna rozhodne, kterou linku použít
- Zákazník dostane hotový produkt, aniž by znal detaily výroby

Factory Pattern funguje stejně - **skrývá složitost vytváření objektů**

Factory Pattern

Factory Pattern (Tovární vzor) umožňuje vytvářet objekty bez specifikace jejich přesné třídy

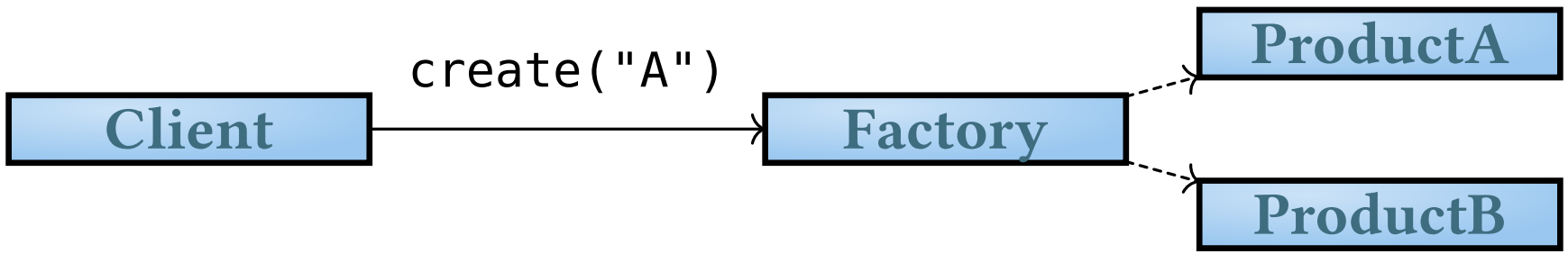
- Místo přímého volání konstruktoru používáme **factory metodu**
- Factory rozhoduje, jakou třídu instancovat
- Umožňuje **flexibilní** vytváření objektů
- **Návratový typ** factory metody **je rozhraní (interface)**

Factory Pattern

Kdy použít Factory Pattern?

- Když nevíme předem, jakou třídu budeme potřebovat
- Když chceme **centralizovat** logiku vytváření objektů
- Když chceme **skrýt** složitost vytváření objektů
- Když potřebujeme podporovat **různé typy** objektů

Struktura Factory Pattern



Příklad Factory

```
interface Animal {
  makeSound(): void;
}

class Dog implements Animal {
  makeSound(): void { console.log("Woof!"); }
}

class Cat implements Animal {
  makeSound(): void { console.log("Meow!"); }
}
```

Příklad Factory

```
class AnimalFactory {
  static createAnimal(type: string): Animal {
    switch (type) {
      case "dog": return new Dog();
      case "cat": return new Cat();
      default:
        throw new Error(`Unknown animal type: ${type}`);
    }
  }
}
```

Příklad Factory

```
const dog = AnimalFactory.createAnimal("dog");
dog.makeSound(); // "Woof!"
```

Pokročilejší příklad Factory

```
type AnimalConstructor = new () => Animal;

class AnimalFactory {
  private static creators:
    Map<string, AnimalConstructor> = new Map();
...
```

Pokročilejší příklad Factory

```
class AnimalFactory {
  ...
  static register(
    type: string,
    creator: AnimalConstructor
  ): void {
    this.creators.set(type, creator);
  }
  ...
```

Pokročilejší příklad Factory

```
class AnimalFactory {
  ...
  static create(type: string): Animal {
    const Creator = this.creators.get(type);
    if (!Creator) {
      throw new Error(`Unknown type: ${type}`);
    }

    return new Creator();
  }
}
```

Pokročilejší příklad Factory

```
// Register types
AnimalFactory.register("dog", Dog);
AnimalFactory.register("cat", Cat);

// Usage
const animal = AnimalFactory.create("dog");
animal.makeSound(); // "Woof!"
```

Pokročilejší příklad Factory

Výhody tohoto přístupu:

- Snadné přidávání nových typů
- Decentralizovaná registrace
- Flexibilní konfigurace

Builder Pattern

Builder Pattern

Jak byste vytvořili objekt s **10 volitelnými parametry**?

Byl by takový konstruktor čitelný a snadno použitelný?

Builder Pattern

Představte si **stavbu domu**

- Nejprve základy, pak zdi, střecha, okna, …
- Můžete si vybrat různé materiály a styly
- Na konci dostanete hotový dům

Builder Pattern staví objekty **krok za krokem**

Builder Pattern

Builder Pattern (Stavitel) umožňuje vytvářet složité objekty krok za krokem

- Odděluje konstrukci objektu od jeho reprezentace
- Umožňuje vytvářet různé varianty objektu
- Zlepšuje čitelnost kódu při vytváření složitých objektů

Builder Pattern

Kdy použít Builder Pattern?

- Když má objekt **mnoho parametrů** konstruktoru
- Když chceme vytvářet objekty **postupně**
- Když potřebujeme různé **varianty** stejného objektu
- Když chceme **validovat** parametry před vytvořením objektu

Builder Pattern

Příklad - Builder pro User:

```
class User {
  constructor(
    public name: string,
    public email: string,
    public age?: number,
    public phone?: string,
    public address?: string,
    public roles: string[] = []
  ) {}
}
```

Builder Pattern

Řešení pomocí Builder Pattern:

```
class UserBuilder {
  private name!: string;
  private email!: string;
  private age?: number;
  private phone?: string;
  private address?: string;
  private roles: string[] = [];
  ...
```

Builder Pattern

```
...
setName(name: string): this {
  this.name = name;
  return this;
}

setEmail(email: string): this {
  this.email = email;
  return this;
}
```

Builder Pattern

```
  setAge(age: number): this {
    this.age = age;
    return this;
  }

  setPhone(phone: string): this {
    this.phone = phone;
    return this;
  }
  ...
```

Builder Pattern

```
...
setAddress(address: string): this {
  this.address = address;
  return this;
}

addRole(role: string): this {
  this.roles.push(role);
  return this;
}
```

Builder Pattern

```
    ...
    build(): User {
      if (!this.name || !this.email) {
        throw new Error("Name and email are required");
      }
      return new User(
        this.name, this.email, this.age,
        this.phone, this.address, this.roles
      );
    }
}
```

Builder Pattern

Použití Builder Pattern:

```
const user = new UserBuilder()
  .setName("John Doe")
  .setEmail("john@example.com")
  .setAge(30)
  .setPhone("123-456-7890")
  .setAddress("123 Main St")
  .build();
```

Builder Pattern

Výhody:

- **Čitelný** kód
- **Flexibilní** vytváření
- **Validace** před vytvořením objektu
- **Fluent interface** (method chaining)

Builder Pattern

Nevýhody:

- **Více kódu** - musíte napsat builder třídu
- **Komplexnost** - pro jednoduché objekty je zbytečný
- **Nekonzistence** - objekt může být v neplatném stavu během stavby
- **Sdílení reference** - stejná builder instance může být omylem použita vícekrát

Builder Pattern

Pozor na sdílení builder instance!

```
const builder = new UserBuilder()
  .setName("John")
  .setEmail("john@example.com")
  .addRole("user");

const user1 = builder.setAge(25).addRole("admin").build();
const user2 = builder.setAge(30).build();

console.log(user1);
console.log(user2);
```

Builder Pattern

Problém: Stejná builder instance může být omylem použita vícekrát a omylem nastavit parametry i pro jiné objekty

Řešení 1: Při každé modifikaci builderu vytvořte nový builder - implementujte **immutable builder**

Řešení 2: Vytvořte si Factory na Builder

```
const user1 = UserBuilderFactory.create(builder) // copy
  .setAge(25).addRole("admin").build();
const user2 = UserBuilderFactory.create(builder) // copy
  .setAge(30).build();
```

Singleton Pattern

Singleton Pattern

Kdy je potřeba mít v aplikaci **pouze jednu instanci** nějaké třídy?

Jak byste zajistili, že se vytvoří právě jedna instance?

- Zakázat přímé vytváření objektů (`new`)?
- Uložit si tu jednu instanci
- Kontrolovat, jestli už instance existuje

Singleton Pattern

Představte si **prezidenta státu**

- V jednom okamžiku může být jen jeden prezident
- Všichni občané přistupují ke stejnému prezidentovi
- Nelze vytvořit druhého prezidenta

Singleton zajišťuje **právě jednu instanci** v celé aplikaci

Singleton Pattern

Singleton Pattern (Jedináček) zajišťuje, že třída má pouze **jednu instanci**

- Poskytuje **globální přístupový bod** k této instanci
- **Kontroluje vytváření** instancí (private constructor)
- Užitečné pro **sdílené zdroje**: databáze, loggery, cache
- Když potřebujeme **globální stav** v celé aplikaci

Singleton Pattern

Kdy NEpoužívat Singleton Pattern?

- Když potřebujete **testovatelný** kód - Singleton ztěžuje **mockování**
- Když chcete **volné propojení** (**loose coupling**) mezi třídami
  - Loose coupling znamená, že třídy nejsou závislé na sobě a mohou být snadno vyměňovány
- Když globální stav může způsobit **nepředvídatelné chování**
- Když plánujete **paralelní zpracování** (více instancí)

Alternativa: Dependency Injection - instance se předává jako závislost

Singleton Pattern

Implementace Singleton v TypeScriptu:

```
class Database {
  private static instance: Database;

  private constructor() {
    // Private constructor prevents direct instantiation
    console.log("Database connection established");
  }
```

Singleton Pattern

```
  static getInstance(): Database {
    if (!Database.instance) {
      Database.instance = new Database();
    }
    return Database.instance;
  }
```

Singleton Pattern

```
  query(sql: string): void {
    if (!Database.instance) {
      throw new Error("Database not initialized");
    }
    console.log(`Executing: ${sql}`);
  }
}
```

Singleton Pattern

Použití Singleton:

```
// Always get the same instance
const db1 = Database.getInstance();
const db2 = Database.getInstance();

console.log(db1 === db2); // true

db1.query("SELECT * FROM users");
db2.query("SELECT * FROM posts");
```

Poznámka: Kvůli **private constructoru** nelze volat `new Database()`

Singleton Pattern

Poznámka k thread-safety

JavaScript/TypeScript běží v **jednom vlákně** (single-threaded)

- Náš jednoduchý Singleton je **dostatečný** pro JS/TS
- Thread-safe implementace je potřeba v jiných jazycích (Java, C#,...)
- V Node.js jsou moduly **cacheovány** a jedná se tedy o přirozený Singleton

```
// In Node.js module itself is a Singleton
export const database = new Database();
```

Shrnutí Creational Patterns

Co jsme se naučili o Creational Patterns?

Factory	Centralizované vytváření objektů různých typů
Builder	Postupné vytváření složitých objektů s fluent API
Singleton	Zajištění pouze jedné instance v celé aplikaci

Kdy použít: Když potřebujete kontrolovat **jak** se objekty vytvářejí

Existují i další Creational Patterns a je vhodné si je ve volném čase prohlédnout

Structural Patterns

Adapter Pattern

Adapter Pattern

Jak propojíte **dva systémy** s různými rozhraními?

Znáte nějaký příklad adaptéru z reálného světa?

- Elektrické zásuvky CZ/UK/US
- Video adaptéry HDMI/VGA/DVI
- Audio adaptéry 3,5mm/HDMI/USB
- USB adaptéry USB-C/USB-A

Adapter Pattern

Představte si **elektrický adaptér** pro zahraniční zástrčku

- Váš notebook má evropskou zásuvku
- V USA jsou jiné zásuvky
- Adaptér převádí jedno rozhraní na druhé

Adapter Pattern **překládá** mezi **nekompatibilními rozhraními**

Adapter Pattern

Adapter Pattern (Adaptér) umožňuje objektům s nekompatibilními rozhraními spolupracovat

- Funguje jako **most** mezi dvěma nekompatibilními rozhraními
- Obaluje objekt a poskytuje nové rozhraní
- Umožňuje **znovupoužití** existujícího kódu

Adapter Pattern

Kdy použít Adapter Pattern?

- Když potřebujeme použít **knihovnu třetí strany** s jiným rozhraním
- Když chceme **integraci** starého kódu s novým
- Když potřebujeme **sjednotit** různé API
- Když chceme **izolovat** změny v externích závislostech

Adapter Pattern

Příklad - Adaptace **starého API** na **nové**:

```
// Old API (legacy code)
class OldPaymentService {
  pay(amount: number, currency: string): void {
    console.log(`Paying ${amount} ${currency}`);
  }
}
```

Adapter Pattern

```
// New interface we want to use
interface PaymentProcessor {
  processPayment(amount: number): void;
}
```

Adapter Pattern

Adapter implementuje nové rozhraní:

```
class PaymentAdapter implements PaymentProcessor {
  private oldService: OldPaymentService;

  constructor() {
    this.oldService = new OldPaymentService();
  }
```

Adapter Pattern

```
  processPayment(amount: number): void {
    // Adapt the call to the old API
    this.oldService.pay(amount, "USD");
  }
}
```

Adapter Pattern

Použití Adapteru:

```
function processOrder(
  processor: PaymentProcessor,
  amount: number
): void {
  processor.processPayment(amount);
}

// We can use the new interface with the adapted old service
const adapter = new PaymentAdapter();
processOrder(adapter, 100);
```

Adapter Pattern

Výhody:

- **Znovupoužití** existujícího kódu
- **Oddělení** závislostí
- **Flexibilita** při změnách API

Decorator Pattern

Decorator Pattern

Decorator Pattern je podrobně vysvětlen v prezentaci o TypeScriptu

Nicméně stejný princip **lze vytvořit i v jiných jazycích**, které přímo nepodporují syntaxi dekorátorů.

Composite Pattern

Composite Pattern

Jak byste reprezentovali **stromovou strukturu** (např. složky a soubory)?

Jak zajistíte jednotné rozhraní pro listy i kontejnery?

Composite Pattern

Představte si **organizační strukturu firmy**

- Firma má oddělení
- Oddělení mají týmy
- Týmy mají zaměstnance
- Můžete spočítat platy na libovolné úrovni

Composite umožňuje pracovat s **částmi i celky** stejným způsobem

Composite Pattern

Composite Pattern (Kompozit) umožňuje skládat objekty do **stromové struktury**

- Reprezentuje části a celky stejným způsobem
- Umožňuje pracovat s jednotlivými objekty i skupinami jednotně
- Užitečné pro **hierarchické struktury**

Composite Pattern

Kdy použít Composite Pattern?

- Když máme **hierarchickou strukturu** objektů
- Když chceme **jednotně** pracovat s jednotlivci i skupinami
- Když potřebujeme **rekurzivní** operace
- Příklady: souborové systémy, UI komponenty, organizační struktury

Příklad - Souborový systém

```
interface FileSystemNode {
  getName(): string;
  getSize(): number;
  display(indent: string): void;
}
```

Příklad - Souborový systém

```
class File implements FileSystemNode {
  constructor(
    private name: string,
    private size: number
  ) {}

  getName(): string { return this.name; }

  getSize(): number { return this.size; }
```

Příklad - Souborový systém

```
  display(indent: string): void {
    console.log(
      `${indent} ${this.name} (${this.size} bytes)`
    );
  }
}
```

Příklad - Souborový systém

Složka (Composite):

```
class Folder implements FileSystemNode {
  private children: FileSystemNode[] = [];

  constructor(private name: string) {}

  getName(): string { return this.name; }

  add(node: FileSystemNode): void {
    this.children.push(node)
  }
```

Příklad - Souborový systém

```
  remove(node: FileSystemNode): void {
    const index = this.children.indexOf(node);
    if (index > -1) { this.children.splice(index, 1); }
  }

  getSize(): number {
    return this.children.reduce(
      (total, child) => total + child.getSize(),
      0
    );
  }
```

Příklad - Souborový systém

```
  display(indent: string): void {
    console.log(`${indent} ${this.name}/`);
    this.children.forEach(child => {
      child.display(indent + "  ");
    });
  }
}
```

Příklad - Souborový systém

Použití Composite Pattern:

```
const root = new Folder("root");
const documents = new Folder("documents");
const photos = new Folder("photos");

documents.add(new File("readme.txt", 1024));
documents.add(new File("notes.md", 2048));
photos.add(new File("vacation.jpg", 5120));

root.add(documents);
root.add(photos);
```

Příklad - Souborový systém

```
root.display("");
// root/
//   documents/
//     readme.txt (1024 bytes)
//     notes.md (2048 bytes)
//   photos/
//     vacation.jpg (5120 bytes)

console.log(`Total size: ${root.getSize()} bytes`); // 8192
```

Stejné rozhraní pro soubory i složky!

Shrnutí Structural Patterns

Co jsme se naučili o Structural Patterns?

Adapter	Převod nekompatibilních rozhraní (jako elektrický adaptér)
Decorator	Dynamické přidávání funkcionalit (vrstvení)
Composite	Stromová struktura s jednotným rozhraním

Kdy použít: Když potřebujete měnit **strukturu** nebo **rozhraní** objektů

Behavioral Patterns

Observer Pattern

Jak byste implementovali **systém notifikací**?

Znáte příklad z reálného světa?

- Předplatné novin nebo notifikace o novinkách na YouTube kanálu
- Email notifikace
- SMS notifikace
- Push notifikace
- ...

Observer Pattern

Představte si **předplatné YouTube kanálu**

- Přihlásíte se k odběru kanálu (subscribe)
- Když YouTuber nahraje video, dostanete notifikaci
- Můžete se kdykoli odhlásit (unsubscribe)

Observer Pattern implementuje **publish-subscribe** mechanismus

Observer Pattern

Observer Pattern (Pozorovatel) definuje závislost jeden-na-mnoho (one-to-many) mezi objekty

- Když se změní stav jednoho objektu, všichni pozorovatelé (**Observers**) jsou **automaticky upozorněni**
- Pozorovaný objekt (**Subject**) může udržovat seznam pozorovatelů (**Observers**)
- Umožňuje **volné propojení** mezi objekty (loose coupling)

Observer Pattern

Kdy použít Observer Pattern?

- Když změna jednoho objektu vyžaduje změnu **jiných objektů** (např. notifikace)
- Když nevíme předem, kolik objektů potřebuje být upozorněno
- Když chceme **oddělit** pozorovaný objekt od pozorovatelů
- Příklady: event systémy, MVC architektura, reactive programming (např. RxJS)

Observer Pattern

Kdy NEpoužívat Observer Pattern?

- Když **zapomenete odhlásit** observery - způsobuje **memory leaks**
- Když na **pořadí notifikací** záleží - není garantované
- Když máte **cyklické závislosti** mezi observery
- Když je **debugging** obtížný - těžko sledovatelný tok dat

Tip: Vždy implementujte `detach()` a používejte ho při destrukci objektů

Observer Pattern

Observer Pattern uvidíte lehce v NestJS a TypeORM a poté v přístím pololetí v Reactu

Iterator Pattern

Iterator Pattern

Jak byste **procházeli kolekci** bez znalosti její vnitřní struktury?

Jak zajistíte jednotný způsob procházení pro různé typy kolekcí?

Iterator Pattern

Představte si **dálkový ovladač TV**

- Tlačítko „další kanál“ přepne na další
- Nemusíte vědět, jak jsou kanály uložené v paměti
- Funguje stejně pro různé značky TV

Iterator poskytuje **jednotný způsob procházení** bez znalosti vnitřní struktury

Iterator Pattern

Iterator Pattern (Iterátor) poskytuje způsob, jak procházet prvky kolekce bez odhalení její vnitřní struktury

- Odděluje algoritmus procházení od struktury kolekce
- Poskytuje **jednotné rozhraní** pro různé typy kolekcí
- Umožňuje **různé způsoby** procházení stejné kolekce

Iterator Pattern

Kdy použít Iterator Pattern?

- Když chceme procházet kolekci **bez znalosti** její struktury
- Když potřebujeme **různé způsoby** procházení
- Když chceme **jednotné rozhraní** pro různé kolekce
- TypeScript/JavaScript má vestavěnou podporu pomocí `Symbol.iterator`, kterou jsme viděli v předchozích přednáškách

Iterator Pattern

Vlastní implementace Iterator Pattern:

```
interface Iterator<T> {
  next(): { value: T | undefined; done: boolean };
  hasNext(): boolean;
}

interface Iterable<T> {
  createIterator(): Iterator<T>;
}
```

Iterator Pattern

Konkrétní kolekce s iterátorem:

```
class NumberCollection implements Iterable<number> {
  constructor(private numbers: number[]) {}

  createIterator(): Iterator<number> {
    return new NumberIterator(this.numbers);
  }
}
```

Iterator Pattern

```
class NumberIterator implements Iterator<number> {
  private index = 0;

  constructor(private numbers: number[]) {}

  hasNext(): boolean {
    return this.index < this.numbers.length;
  }
```

Iterator Pattern

```
  next(): { value: number | undefined; done: boolean } {
    if (this.hasNext()) {
      return {
        value: this.numbers[this.index++],
        done: false
      };
    }
    return { value: undefined, done: true };
  }
}
```

Iterator Pattern

Použití vlastního iterátoru:

```
const collection = new NumberCollection([1, 2, 3, 4, 5]);
const iterator = collection.createIterator();

while (iterator.hasNext()) {
  const result = iterator.next();
  console.log(result.value);
}
// 1 ... 5
```

Iterator Pattern

TypeScript/JavaScript má vestavěnou podporu:

```
class CustomCollection implements Iterable<number> {
  private items: number[] = [1, 2, 3, 4, 5];

  [Symbol.iterator](): Iterator<number> {
    let index = 0;
    const items = this.items;
```

Iterator Pattern

```
    return {
      next(): IteratorResult<number> {
        if (index < items.length) {
          return { value: items[index++], done: false };
        }
        return { value: undefined, done: true };
      }
    };
  }
}
```

Iterator Pattern

Použití s `for...of`:

```
const collection = new CustomCollection();

for (const item of collection) {
  console.log(item);
}
// 1 ... 5

// Or using spread operator
const array = [...collection];
console.log(array); // [1, 2, 3, 4, 5]
```

Strategy Pattern

Strategy Pattern

Jak byste umožnili **změnu algoritmu za běhu** programu?

Jak se vyhnout dlouhým `if-else` větvím při výběru algoritmu?

Strategy Pattern

Představte si **navigaci v autě**

- Můžete zvolit: nejkratší cesta, nejrychlejší, bez dálnic
- Navigace funguje stejně, jen použije jiný algoritmus
- Strategii můžete změnit kdykoli za jízdy

Strategy Pattern umožňuje **zaměnitelné algoritmy**

Strategy Pattern

Strategy Pattern (Strategie) umožňuje definovat rodinu algoritmů, zapouzdřit je a učinit je zaměnitelnými

- Umožňuje **měnit algoritmus** za běhu
- Odděluje algoritmus od kódu, který ho používá
- Umožňuje **snadné přidávání** nových algoritmů

Strategy Pattern

Kdy použít Strategy Pattern?

- Když máme **více způsobů**, jak provést stejnou operaci
- Když chceme **izolovat** implementaci algoritmu
- Když chceme **měnit algoritmus** dynamicky
- Když chceme **vyhnout se** podmínkám v kódu

Strategy Pattern

Příklad - Různé strategie platby:

```
interface PaymentStrategy {
  pay(amount: number): void;
}

class CreditCardPayment implements PaymentStrategy {
  constructor(private cardNumber: string) {}
```

Strategy Pattern

```
  pay(amount: number): void {
    console.log(
      `Paid ${amount} using credit card ${this.cardNumber}`
    );
  }
}
```

Strategy Pattern

Další strategie:

```
class PayPalPayment implements PaymentStrategy {
  constructor(private email: string) {}

  pay(amount: number): void {
    console.log(
      `Paid ${amount} using PayPal ${this.email}`
    );
  }
}
```

Strategy Pattern

```
class BankTransferPayment implements PaymentStrategy {
  constructor(private accountNumber: string) {}

  pay(amount: number): void {
    console.log(
      `Paid ${amount} via bank transfer to
${this.accountNumber}`
    );
  }
}
```

Strategy Pattern

```
class PaymentProcessor {
  private strategy?: PaymentStrategy;

  setStrategy(strategy: PaymentStrategy): void {
    this.strategy = strategy;
  }
```

Strategy Pattern

```
  processPayment(amount: number): void {
    if (!this.strategy) {
      throw new Error("Payment strategy not set");
    }
    this.strategy.pay(amount);
  }
}
```

Strategy Pattern

```
const processor = new PaymentProcessor();

// We can change strategy dynamically
processor.setStrategy(
  new CreditCardPayment("1234-5678-9012-3456")
);
processor.processPayment(100);
// Paid 100 using credit card 1234-5678-9012-3456
```

Strategy Pattern

```
processor.setStrategy(
  new PayPalPayment("user@example.com")
);
processor.processPayment(50);
// Paid 50 using PayPal user@example.com
```

Strategy Pattern

Příklad - Řazení:

```
interface SortStrategy<T> {
  sort(items: T[]): T[];
}

class QuickSort<T> implements SortStrategy<T> {
  sort(items: T[]): T[] {
    // Quick sort implementation
    return [...items].sort();
  }
}
```

Strategy Pattern

```
class MergeSort<T> implements SortStrategy<T> {
  sort(items: T[]): T[] {
    // Merge sort implementation
    return [...items].sort();
  }
}
```

Strategy Pattern

```
class Sorter<T> {
  constructor(private strategy: SortStrategy<T>) {}

  setStrategy(strategy: SortStrategy<T>): void {
    this.strategy = strategy;
  }

  sort(items: T[]): T[] {
    return this.strategy.sort(items);
  }
}
```

Strategy Pattern

```
const sorter = new Sorter(new QuickSort());
const numbers = [3, 1, 4, 1, 5, 9, 2, 6];
console.log(sorter.sort(numbers));
```

Shrnutí Behavioral Patterns

Co jsme se naučili o Behavioral Patterns?

Observer	Automatické notifikace při změně stavu (pub-sub)
Iterator	Jednotný způsob procházení kolekcí
Strategy	Zaměnitelné algoritmy za běhu

Kdy použít: Když potřebujete řídit **komunikaci** nebo **chování** mezi objekty

Shrnutí

Klíčové poznatky

Creational Patterns - vytváření objektů

- **Factory** - centralizované vytváření objektů
- **Builder** - postupné vytváření složitých objektů
- **Singleton** - zajištění jedné instance třídy

Klíčové poznatky

Structural Patterns - kompozice objektů

- **Adapter** - přizpůsobení rozhraní
- **Decorator** - dynamické přidávání funkcionalit
- **Composite** - stromová struktura objektů

Klíčové poznatky

Behavioral Patterns - komunikace mezi objekty

- **Observer** - notifikace o změnách
- **Iterator** - procházení kolekcí
- **Strategy** - zaměnitelné algoritmy

Best Practices

Kdy použít návrhové vzory?

- Používejte je, když **řeší konkrétní problém**
- **Nepoužívejte** je jen proto, že „to tak děláme“
- Zvažte **komplexnost** vs. **přínos**
- V TypeScriptu/JavaScriptu některé vzory jsou **vestavěné** (Iterator, Observer)

Best Practices

Varování:

- **Nepřehánějte** to - ne každý problém potřebuje vzor
- **Začátečníkům** mohou vzory zbytečně komplikovat kód
- **Znalost vzorů** pomáhá, ale není všespásná
- **Praxe** je důležitější než teorie

Závěr

Návrhové vzory jsou **nástroje**, ne cíle

- Pomáhají řešit **běžné problémy**
- Zlepšují **komunikaci** mezi vývojáři
- Vedou k **lepší struktuře** kódu
- Jsou **osvědčená řešení**, ne dogma