pokrocile-programovani

Petr Zámečník

April 10, 2024

Abstract

Tato práce se zabývá třemi důležitými tématy v oblasti vývoje softwaru: čistým kódem, Test Driven Developmentem (TDD) a návrhovým vzorem Command.

Práce shrnuje klíčové poznatky o těchto třech tématech a zdůrazňuje jejich důležitost pro vývoj kvalitního softwaru.

Klíčová slova: čistý kód, funkce, Test Driven Development (TDD), návrhový vzor Command, objektově orientované programování, vývoj softwaru

1 Definice čistého kódu

Čistý kód je kód, který se vyznačuje čitelností, srozumitelností, udržitelností a efektivitou. Je snadno pochopitelný i pro programátory, kteří s ním dosud nepracovali, a umožňuje snadnou údržbu a rozšiřování. Čistý kód má také pozitivní vliv na testovatelnost a snižuje riziko výskytu chyb. [**CleanCodeBook**]
Mezi důležité vlastnosti čistého kómu můžeme zařadit například tyto:

1.1 Logická struktura

- Kód je rozdělen do funkcí a tříd, které odpovídají za konkrétní úkoly.
- Funkce a třídy jsou uspořádány do logických celků (např. balíčky v Javě).
- Kód je strukturován pomocí podmínek a smyček, které řídí jeho tok.

1.2 Konzistentní formátování

- Kód je formátován dle konzistentních pravidel (např. odsazení, mezery).
- Formátování usnadňuje čtení a pochopení kódu.
- Existují nástroje pro automatické formátování kódu.

1.3 Smysluplná pojmenování

- Proměnné, funkce a další entity jsou pojmenovány tak, aby jasně vyjadřovaly svůj účel.
- Názvy by měly být výstižné a snadno zapamatovatelné.
- Je vhodné se vyhnout zkratkám a jednoznačným názvům

1.4 Komentáře

- Komentáře by měly být stručné a výstižné
- Není nutné psát komentáře všude, zejména tam kde je logika kódu jednoznačná

1.5 Jednoduchost

- Kód je vhodné psát jednoduchý, a kde není potřeba tak jej zbytečně nekomplikovat
- Jednoduchost kódu umožnuje lepší čitelnost a udržitelnost

1.6 Testovatelnost

- Jednoduchý kód je snažší testovat
- Testy je tudíž možno psát pro menší části kódu

Example of clean code

```
class ProductService {  no usages
    private products: Product[] = [
        {id: '1'...},
        {id: '2'...},
        {id: '3'...},
    ];

    getProducts(): Product[] {  no usages
        return this.products;
    }

    getProductById(id: string): Product | undefined {  Show usages
        return this.products.find((product: Product) : boolean  => product.id === id);
    }

    getProductsByCategory(category: string): Product[] {  no usages
        return this.products.filter(product : Product  => product.category === category);
    }

    getProductsByRating(minRating: number): Product[] {  no usages
        return this.products.filter(product : Product  => product.rating >= minRating);
    }

    addProduct(product: Product): void {  no usages
        this.products.push(product);
    }

    updateProduct(id: string, updatedProduct: Partial<Product>): boolean {  no usages
        const index = this.products.findIndex((product: Product) : boolean  => product.id === id);
        if (index !== -1) {
            this.products[index] = { ...this.products[index], ...updatedProduct };
            return true;
        }
        return false;
    }

    deleteProduct(id: string): boolean {  no usages
        const index = this.products.findIndex((product: Product) : boolean  => product.id === id);
        if (index !== -1) {
            this.products.splice(index,  deleteCount: 1);
            return true;
        }
        return false;
    }

    addReview(productId: string, review: Review): boolean {  no usages
        const product = this.getProductById(productId);
        if (product) {
            product.reviews.push(review);
            return true;
        }
        return false;
    }
}
```

2 Funkce

Samotné využívání funkcí představuje základní kámen v konstrukci čistého a efektivního kódu. Díky nim lze rozdělit program do logických bloků, což usnadňuje jeho strukturování a porozumění. Když je kód rozdělen do menších částí, je mnohem snazší sledovat jeho tok a identifikovat případné chyby nebo nedostatky.

To znamená, že i složité úkoly mohou být rozloženy na menší, lépe řešitelné úseky, což v konečném důsledku vede k lepší údržbě a vývoji softwaru. Dalším klíčovým aspektem je znovupoužitelnost kódu díky funkcím. Pokud je správně navržena a implementována funkce, může být použita v různých částech programu nebo dokonce v jiných projektech. Tím se snižuje redundance kódu a zvyšuje se jeho modularita. Pokud je potřeba provést změnu, je často nutné upravit pouze jednu funkci, což minimalizuje riziko chyb a zjednodušuje úpravy. Tato schopnost znovu používat existující kód výrazně zlepšuje produktivitu vývojářů a zkracuje čas potřebný k vytvoření nových funkcí.

Celkově lze tedy říci, že funkce nejsou pouze prostředkem k dosažení cíle, ale jsou základem pro efektivní a kvalitní vývoj softwarových projektů. Jejich správné využití přináší řadu výhod včetně zlepšené čitelnosti, snížení komplexity, lepší údržby a možnosti znovupoužití kódu, což v konečném důsledku vede k vyšší efektivitě a kvalitě výsledného produktu.

2.1 Výhody funkcí

- Zlepšení čitelnosti: Funkce s dobrým názvem mohou sloužit jako komentáře, které vysvětlují, co určitá část kódu dělá.
- Usnadnění údržby: Úpravy v jedné funkci se obvykle nešíří do ostatních částí kódu, což snižuje riziko neočekávaných chyb.
- Zjednodušení testování: Jednotlivé funkce lze testovat izolovaně, což usnadňuje identifikaci chyb.
- Opakované využití kódu: Funkce umožňují znovupoužití kódu, což snižuje redundanci a usnadňuje změny.

2.2 Nejlepší praktiky pro psaní funkcí

- Jedna funkce, jeden účel: Každá funkce by měla mít jedinou zodpovědnost, což usnadňuje její pochopení a údržbu.
- Krátké a jasné: Funkce by měly být co nejkratší, aniž by se obětovala čitelnost. Delší funkce je často možné rozdělit na menší, specializovanější funkce.
- Jasná pojmenování: Název funkce by měl jasně vysvětlovat, co funkce dělá. Vhodné pojmenování usnadňuje pochopení účelu funkce bez nutnosti prohlížet její implementaci.

- Omezení počtu parametrů: Funkce by měly mít co nejméně parametrů. Mnoho parametrů může funkci zkomplikovat a snížit její čitelnost.
- Používání návratových hodnot: Funkce by měly vracet hodnoty, které jasně indikují výsledek jejich provádění, což usnadňuje psaní čitelného a udržitelného kódu.

2.3 Nevýhody a omezení

- Nadužívání funkcí: Přílišné rozdělení kódu do funkcí může vést k nadbytečné abstrakci a zbytečné složitosti, což může ztížit pochopení toku programu.
- Výkonnostní kompromisy: V některých případech může volání funkcí způsobit mírné zpomalení, zejména v jazycích, kde je overhead volání funkcí významný. V kritických částech kódu, kde je výkon klíčový, je třeba zvážit rovnováhu mezi čistotou kódu a výkonem.

2.4 Příklad funkce - Typescript

Následující funkce v jazyce Typescript vrací n-tý prvek Fibonacciho posloupnosti:

```
function fibonacci(n: number): number {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

2.5 Příklad funkce - Python

Následující funkce v jazyce Python vrací n-tý prvek Fibonacciho posloupnosti:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

3 Test Driven Development

Test Driven Development (TDD) je metodika vývoje softwaru, která klade důraz na testování. Testy jsou psány před samotným kódem a slouží jako specifikace požadované funkcionality. Tato metodika podporuje agilní vývoj a pomáhá vytvářet robustní a kvalitní software.

3.1 Historie a principy TDD

TDD poprvé popsal Kent Beck v roce 1999 v knize **"Extreme Programming Explained"**. Metodika vychází z principů extrémního programování a zdůrazňuje následující:

- Testy před kódem: Než napíšete kód, nejprve napište test, který ověří požadovanou funkcionalitu.
- Malé kroky: Implementujte funkcionalitu v malých krocích, tak aby každý krok byl pokryt testy.
- Refaktoring: Po implementaci funkcionality proveďte refaktoring kódu, abyste jej zjednodušili a zlepšili jeho čitelnost.

3.2 Výhody TDD

- Zvýšená kvalita softwaru: TDD pomáhá odhalit chyby v rané fázi vývoje a snižuje tak počet chyb v produkčním kódu.
- Lepší design: TDD podporuje modularizaci kódu a jeho testování v izolaci, což vede k čistšímu a lépe strukturovanému designu.
- Větší jistota: TDD dává vývojářům jistotu, že kód funguje správně a splňuje požadavky.
- Lepší dokumentace: Testy slouží jako dokumentace požadované funkcionality.

3.3 Nevýhody TDD

- Zvýšená náročnost: TDD vyžaduje disciplínu a zkušenost od vývojářů.
- Zvýšený čas: Psaní testů před kódem může zpočátku zdržet vývoj.
- Není vhodné pro všechny typy projektů: TDD nemusí být vhodné pro všechny typy projektů, například pro prototypování nebo rychlé dodávky.

3.4 Implementace TDD

Implementace TDD sestává z následujících kroků:

1. Definice požadavků: Jasně definujte požadavky na funkcionalitu, kterou chcete implementovat.
2. Napsání testu: Napište test, který ověří požadovanou funkcionalitu. Test by měl být specifický, měřitelný, dosažitelný, relevantní a časově ohraničený.
3. Implementace kódu: Implementujte kód, který splňuje požadavky testu. Začněte s nejjednodušší možnou implementací a postupně ji rozšiřujte.
4. Refaktoring: Po implementaci kódu proveďte refaktoring, abyste jej zjednodušili a zlepšili jeho čitelnost.
5. Opakování cyklu: Opakujte kroky 2 až 4 pro všechny požadované funkce.

3.5 Nástroje pro TDD

Existuje mnoho nástrojů, které usnadňují implementaci TDD. Mezi nejoblíbenější patří:

- Testovací frameworky: Testovací frameworky, jako JUnit (Java) nebo NUnit (.NET), usnadňují psaní a spouštění testů.
- Mockingové frameworky: Mockingové frameworky, jako Mockito (Java) nebo Moq (.NET), umožňují simulovat závislosti v testech.
- Nástroje pro refaktoring: Nástroje pro refaktoring, jako IntelliJ IDEA (Java) nebo Visual Studio (.NET), usnadňují úpravu kódu bez narušení jeho funkcionality.

3.6 Příklad - Typescript

Kód testující funkci validace emailu

```
describe('validateEmail', () => {
  it('should return true for a valid email address', () => {
    expect(validateEmail('test@example.com')).toBe(true);
  });

  it('should return false for an invalid email address', () => {
    expect(validateEmail('test@example')).toBe(false);
    expect(validateEmail('testexample.com')).toBe(false);
    expect(validateEmail('@example.com')).toBe(false);
  });
});
```

Funkce sloužící pro validaci emailu, psaná dle testu uvedeného výše

```
export const validateEmail = (email: string): boolean => {
  const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return re.test(email);
};
```

3.7 Shrnutí

TDD je efektivní metodika vývoje softwaru, která pomáhá vytvářet robustní a kvalitní software. TDD vyžaduje disciplínu a zkušenost od vývojářů, ale přináší mnoho výhod, jako je zvýšená kvalita softwaru, lepší design a větší jistota.

4 Návrhový vzor Command

Návrhový vzor Command zapouzdřuje operaci a její parametry do objektu, čímž umožňuje odložit provedení operace na později. To umožňuje flexibilnější a jednodušší implementaci logiky programu, jelikož umožňuje oddělit požadavek na provedení akce od jejího skutečného provedení.

4.1 Charakteristika

Návrhový vzor Command se skládá ze tří základních komponent:

- **Příkaz (Command)**: Definuje rozhraní pro provedení akce. Konkrétní implementace příkazu definuje, jaká akce se má provést.
- **Klient (Client)**: Vytváří instance příkazů a předává je objektu Invoker.
- **Invoker (Vyvolávač)**: Uchovává seznam příkazů a zodpovídá za jejich spuštění.

4.2 Výhody

Návrhový vzor Command přináší mnoho výhod, mezi které patří:

- Zvýšená flexibilita: Umožňuje odložit provedení akce na později, čímž umožňuje dynamické plánování a spouštění akcí.
- Zjednodušení kódu: Odděluje logiku požadavku na provedení akce od logiky samotné akce, čímž umožňuje jednodušší a přehlednější kód.
- Podpora undo/redo funkcionality: Umožňuje snadno implementovat funkce pro vrácení a opakování provedených akcí, jelikož historii provedených akcí lze reprezentovat seznamem příkazů.
- Rozšiřitelnost: Umožňuje snadno přidávat nové akce do systému, jelikož stačí implementovat novou třídu příkazu.

4.3 Nevýhody

Návrhový vzor Command má i svá negativa, mezi která patří:

- Zvýšená režie: Zavedení objektu Command pro každou akci může vést k mírnému zvýšení režie.
- Složitější struktura: V porovnání s přímým voláním funkcí může struktura s objekty Command a Invoker působit složitěji.

4.4 Příklad použití

Návrhový vzor Command se dá použít v široké škále aplikací. Níže je uveden příklad jeho použití v grafickém uživatelském rozhraní (GUI):

- Tlačítka v GUI můžou reprezentovat objekty Command. Kliknutím na tlačítko se instance Command předá objektu Invoker, který ji následně spustí.
- Makra v textových editorech můžou být implementována pomocí vzoru Command. Makro se skládá ze seznamu příkazů, které se po spuštění makra postupně provedou.
- Historie příkazů v grafických programech můžou být implementovány pomocí vzoru Command. Historie uchovává seznam provedených příkazů, které se dají zpětně spustit nebo vrátit zpět.