

Agilní Návrh a SOLID

Literatura

- ▶ **Agile Principles, Patterns, and Practices in C#**
MARTIN, Robert C.; MARTIN, Micah.
Kapitoly 7 – 12

Co je to Návrh?

- ▶ Návrh, nebo také design, je abstraktní koncept softwarového projektu. Je to celkový tvar a struktura programu, stejně tak jako detailní tvar a struktura jednotlivých modulů, tříd a metod.
- ▶ Je celá řada médií, kterými lze návrh reprezentovat, jako jsou např. UML Diagramy. Konečná a nejpřesnější reprezentace je ale zdrojový kód.
- ▶ Proto je kvalitní návrh základní předpoklad kvalitního kódu a je nutné mu při vývoji věnovat velkou pozornost.

Návrh v Životním Cyklu Softwaru

Získávání požadavků -> Návrh -> Implementace ->
Testování -> Údržba

- ▶ V dobře známém vodopádovém modelu je doba návrhu položena mezi získávání požadavků na software a jeho implementací.
- ▶ V praxi ale vodopádový model většinou nefunguje a v modelu mohou vznikat i skoky zpět a cykly.
- ▶ Některé modely řízení vývoje přímo počítají s pravidelným iterováním fází životního cyklu (např. SCRUM).
- ▶ Díky tomu může být návrh během životního cyklu SW často rozšiřován nebo modifikován kvůli novým nebo změněným požadavkům od zákazníka.

Znaky Špatného Návrhu I

- ▶ Přestože první verze návrhu může být „čistá“ (plně vystihuje strukturu a filozofii zákaznickových požadavků). Každá jeho další změna do něj může zanést jeden nebo více znaků špatného návrhu:
- ▶ **Rigidita** – tendence SW udělat každou, i jednoduchou změnu obtížnou. Návrh je rigidní, pokud i malá změna způsobí řadu navazujících změn v závislých modulech.
- ▶ **Křehkost** – když je provedena změna programu na jednom místě, rozbije na mnoha jiných místech, často takový, které s místem změny nemají logickou souvislost.

Znaky Špatného Návrhu II

- ▶ **Imobilita** – je případ kdy program obsahuje užitečnou funkcionalitu, kterou lze použít v jiných programech, ale úsilí na její separaci a riziko s tím spojené je příliš velké.
- ▶ **Viskozita** – existuje více způsobů, jak realizovat nějakou změnu v SW. Některé udržují čistotu návrhu, jiné (hacks) čistý návrh obcházejí. Návrh je viskózní, pokud je mnohem jednodušší návrh obcházet, než jej udržovat čistý.
- ▶ **Zbytečná Složitost** – návrh obsahuje elementy které nejsou používány. Designer je přidal proto, aby pomohli při případném rozšíření SW, ale pravděpodobně nebudou použity a jen zvyšují složitost a nesrozumitelnost návrhu.

Znaky Špatného Návrhu III

- ▶ **Zbytečné Opakování** – V SW se vyskytuje několik stejných nebo trochu rozdílných fragmentů kódu. Je složité je v případě potřeby změnit, navíc, protože jsou trochu odlišné i změna může být odlišná.
- ▶ **Nesrozumitelnost** – vlastnost kódu ztěžující jeho čitelnost. Kód, který jeho autor právě napsal a který mu připadá čitelný, se pro něj může stát za nějaký čas nepochopitelným, protože jej správně nerefaktoroval.

Agilní Návrh a SOLID

- ▶ Při agilním návrhu jsou všechny znaky špatného návrhu likvidovány hned v jejich zárodku pravidelnou refaktORIZACÍ.
- ▶ Jako pomůcka při hledání a opravě chyb v návrhu slouží pět principů, které se podle jejich počátečních písmen nazývají **SOLID**. Jsou to:
 - **Single-Responsibility Principle (SRP)**
 - **Open/Close Principle (OCP)**
 - **Liskov Substitution Principle (LSP)**
 - **Interface Segregation Principle (ISP)**
 - **Dependency-Inversion Principle (DIP)**

Single-Responsibility Principle

Třída by měla mít pouze jeden důvod ke změně.

- ▶ Uvažme třídu `Rectangle`, ve které jsou implementovány základní geometrické výpočty a metoda `Draw`, která vykreslí obrazec na obrazovku.
- ▶ Třidu používají dvě aplikace: jedna používá pouze geometrické výpočty, druhá jak výpočty tak vykreslování.
- ▶ To je typický příklad porušení SRP, kdy jedna třída zastává dvě různé logické funkce a může být změněna ze dvou různých důvodů.
- ▶ Protože jsou obě funkcionality umístěny v jedné třídě, jsou navzájem provázané a změna jedné může způsobit změnu druhé. Tím se zanáší do návrhu *křehkost* (změna vykreslování může ovlivnit geometrii a tím první aplikaci, která ve skutečnosti vykreslování nepoužívá).
- ▶ Podle SRP je dobré separovat funkcionalitu do dvou tříd.

Single-Responsibility Principle

- ▶ Uvažme příklad modemu. Na první pohled nedělitelnou funkcionalitu složenou ze čtyř operací (`send`, `recv`, `dial` a `hangup`), lze rozdělit na dvě: navazování spojení a přenos dat.
- ▶ Je nezbytné provést separaci funkcionality pro zajištění čistoty návrhu? Pokud se funkcionalita v našich aplikacích nepoužívá odděleně, nic tím nezískáme a zanášíme do návrhu *zbytečnou složitost*.
- ▶ Odpověď zní: ano, ale pouze pokud to má důvod. Stejně jako ostatní principy je dobré SRP aplikovat pouze tehdy, řeší-li to nějaký problém.

Open/Close Principle

Softwarové entity (třídy, moduly, metody atd.) by měli být otevřeny pro rozšíření, ale uzavřeny pro změnu

- ▶ OCP radí jak psát software, aby jeho změny nevyvolávali série dalších změn v již existujícím kódu – návrh nebyl rigidní.
- ▶ Moduly které odpovídají OCP principu mají následující dvě vlastnosti:
 - Jsou otevřené pro rozšíření, v případě že je třeba přidat novou funkcionalitu do naší aplikace.
 - Jsou uzavřené pro modifikaci. To znamená, že rozšíření funkcionality se neprojeví na zdrojovém ani strojovém kódu modulu: DLL i EXE soubor zůstane nedotčen.
- ▶ Na první pohled nesmyslného požadavku lze dosáhnout pomocí vhodné abstrakce.

Open/Close Principle

- ▶ Příkladem může být jednoduchý klient, komunikující např. přes TCP se serverem. Porušuje OCP princip, protože klient je pevně svázaný s konkrétní komunikační technologií a pokud nastane požadavek na změnu komunikačního rozhraní, musí se změnit jeho kód.
- ▶ Je třeba vytvořit abstrakci komunikace. Klient nebude mít přímo zabudovanou komunikaci přes TCP, místo toho bude používat objekt implementující rozhraní `IChannel`, který dostane jako parametr.
- ▶ Nyní, budeme-li chtít použít jiný způsob komunikace, stačí ji realizovat v objektu implementujícím `IChannel`, a předat jej klientovy. Jeho kód, stejně tak jako kód celé knihovny, v které je implementován nemusí být měněn.
- ▶ Klient je nyní otevřený pro rozšíření, ale uzavřený pro změnu.

Open/Close Principle

- ▶ Naše vylepšení umožnilo rozšiřovat klienta určitým směrem, ale zbytek jeho funkcionality zůstává stále jeho pevnou součástí. Klient např. může získané informace dále zpracovávat, což už naše abstrakce neřeší. Pro zachování OCP i v tomto směru je třeba přidat abstrakci novou.
- ▶ Dá se napsat třída tak aby byla imunní proti všem změnám? Má cenu se o to pokoušet?
- ▶ Odpověď zní ne. Kód, který je příliš abstraktní v sobě nese známky *zbytečné složitosti*, která přidává režii na udržování kódu, a nemusí být využita.
- ▶ Pro které rozšíření kód připravit a pro které ne je otázkou zkušeností a je dobré se řídit pravidlem: napsat kód co nejjednodušeji, ale jakmile je ho třeba rozšířit, refaktorovat jej tak, aby byl otevřený/uzavřený pro všechny změny podobného typu.

Liskov Substitution Principle

Podtyp musí být schopný zastoupit své nadtypy.

- ▶ Základní způsob jak realizovat abstrakci je využít dědičnost. Otázka *co lze dědit od čeho* je často řešena pomocí tzv. relace *IS-A*: je potomek specializovaný případ předka?
- ▶ Jak ukáže následující příklad, tato relace sama osobě nezaručuje čistotu návrhu a způsobuje *křehkost* kódu. Proto je třeba při dědičnosti aplikovat LSP, který říká, že lze použít dědičnost pouze v případě, kdy potomek může být použit kdekoli na místě předka, aniž by to způsobilo problémy.
- ▶ Je zde kladen větší důraz na stejné chování podtypu, než na to, jestli se logicky jedná o specializovaný případ nadtypu.

Liskov Substitution Principle

- ▶ Mějme třídy `Rectangle` a `Square`. Obě implementují vlastnosti *výška* a *délka*, a metodu vracející obsah obrazce. Z hlediska IS-A lze bez problémů podědit `Square` od `Rectangle`, protože čtverec je speciální případ obdélníku. Je to ale správné?
- ▶ Uvažme dvě metody, které berou na vstup typ `Rectangle`:
- ▶ Metoda `f` nastaví délku a výšku obrazce. Už zde je cítit jistý problém v tom, že objekt `Square` nějakým způsobem musí ošetřit svůj invariant „*výška = délka*“. To lze vyřešit přepsáním *setteru* obou vlastností tak, že nastaví jak délku tak výšku na stejnou hodnotu.
- ▶ Pak volání metody projde u obou typů a v případě čtverce bude po skončení metody výška i délka rovna jedné.

```
public void f(Rectangle r)
{
    r.Width = 2;
    r.Height = 1;
}
```

Liskov Substitution Principle

```
public void g(Rectangle r)
{
    r.Height = 4;
    r.Width = 5;
    if (r.GetArea() != 20)
        throw new Exception();
}
```

- ▶ Uvažme metodu `g`, která kontroluje korektnost výpočtu obsahu. Autor zcela legitimně předpokládá, že změna délky nijak neovlivní výšku obrazce (a naopak), a že výsledný obsah bude 20. V případě čtverce však metoda vyhodí výjimku.
- ▶ `Square` nemůže zastoupit nadtyp `Rectangle`, a tím je porušen LSP. Následek je křehkost kódu, a výskyt neočekávaného chování na jiném místě, než je jeho opravdová příčina.

Interface Segregation Principle

Klienti by neměli být nuceni k závislosti na metodách, které nepoužívají.

- ▶ Vždy nelze navrhnout software tak, aby obsahoval pouze malé třídy, s malým množstvím metod, které jsou navzájem *soudržné* (tzn. navzájem spolu souvisí a zajišťují stejnou funkcionalitu)
- ▶ Někdy je třeba implementovat velkou třídu se spoustou *nesoudržných* metod, které lze rozdělit do několika skupin podle toho, jak jsou používány různými druhy dalších objektů (klientů).
- ▶ Tím se do kódu vnáší rigidita, protože některé změny (např. v hlavičce metody) si mohou vynutit změny v částech kódu, kterých se změna logicky netýká a následný rebuild a redeploy.
- ▶ Špatně navržené rozhraní navíc může vynutit implementaci metod, které nejsou pro danou funkcionalitu potřeba, což vede k potenciálnímu porušení LSP.

Interface Segregation Principle

- ▶ Uvažme třídu implementující uživatelskému rozhraní, např. pro Informační Systém.
- ▶ S rozhraním komunikuje celá řada komponent systému, které čtou a zobrazují různé typy dat (sestavy zaměstnanců, statistiky apod.)
- ▶ Každá komponenta volá určitou množinu metod, které by měli být odděleny do zvláštního rozhraní, které velká třída implementuje.
- ▶ Tento přístup má dvě výhody:
 - V případě potřeby přidat novou komponentu stačí navrhnout nové rozhraní s třídou a doimplementovat nové metody. Ostatní komponenty tato změna nezasáhne, protože s třídou komunikují přes svá vlastní rozhraní, která nebyla změněna.
 - V případě potřeby implementovat pro některé komponenty nové uživatelské rozhraní, stačí implementovat pouze jejich rozhraní, a rozhraní ostatních komponent ignorovat.

Dependency–Inversion Principle

- A. Vysokourovňové moduly by neměly být závislé na nízkoúrovňových. Oboje by měli být závislé na abstrakci.
- B. Abstrakce by neměla záviset na detailech, detaily by měly záviset na abstrakci.

- ▶ Klasický nešvar vícevrstevných aplikací je že moduly vyšší úrovně jsou závislé na těch nižších. Je to způsobeno špatným užitím abstrakce a faktem, že závislost je tranzitivní a tím pádem se přenáší přes více úrovní.
- ▶ Snadno lze nahlédnout, že tento stav přináší do návrhu *rigiditu* a *křehkost*. Vysoké vrstvy zajišťující business logiku by neměly být modifikovány v případě změny na nižší úrovni (např. úprava komunikačního protokolu).
- ▶ DIP pojednává o zrušení této závislosti přidáním vhodné abstrakce, závislosti se invertují a nižší moduly se stávají závislé na vyšších.

Dependency–Inversion Principle

- ▶ Používá–li modul (třída) jiný modul a odkazuje na něj přímo, tento vztah postrádá abstrakci. K ní je třeba rozhraní nebo abstraktní třída, kterou modul používá místo konkrétního typu, a kterou druhý modul implementuje.
- ▶ Základní otázkou při aplikaci DIP je vlastnictví rozhraní (kdo má právo rozhraní měnit). Přirozený postoj může být takový, že modul poskytující službu, implementuje rozhraní a tudíž je rozhraní jeho. Změní–li svoji funkcionalitu, upraví podle toho i své rozhraní.
- ▶ Tento přístup však přináší závislost vyššího modulu na rozhraní, a tedy i závislost na nižším modulu. Podle DIP je třeba tento problém vyřešit *inverzí vlastnictví rozhraní*.
- ▶ Pokud modul vyšší úrovně vlastní rozhraní, kterým popisuje, jaké služby potřebuje ke své korektní činnosti od modulů nižších vrstev, není závislí na jejich implementaci.

Dependency–Inversion Principle

- ▶ Otázka vlastnictví rozhraní nemusí být vždy správně zodpovězena tak, že kdo používá službu jiného, vlastní rozhraní, přes které je služba poskytována.
- ▶ Příklad může být vztah server/klient, kdy existuje mnoho klientů, kteří různě využívají služeb serveru. Nelze implementovat pro každého klienta speciální server, a proto je poskytnut jeden, který vlastní rozhraní, které vystavuje ve zvláštní knihovně.
- ▶ Ještě divnější případ je vztah tříd `Button` a `Lamp` propojené rozhraním `ButtonServer` s metodami `TurnOn` a `TurnOff`. Tlačítko s je závislé na rozhraní, ale není závislé na lampě (může klidně zapínat elektromotor). Stejně tak lampa je závislá na rozhraní, ale ne na tlačítku (může jej používat i vypínač).
- ▶ Vzniká tak situace, kdy rozhraní nepatří nikomu, a může být libovolně implementováno a používáno různými moduly.