NÁVRHOVÉ VZORY

Přednáška 12

SHRNUTÍ

- Probraná témata:
  - Čistý kód
    - SOLID, KISS, Antipaterny
  - Návrhové vzory
  - Refactoring a návrh struktury softwarového díla

REFACTORING

- Clean code (čistý kód)
- Hlavním cílem refaktoringu je boj s technickým dluhem. Refaktoring přeměňuje nepořádek na čistý kód a jednoduchý design.
- **Čistý kód je pro ostatní programátory srozumitelný**.
  - A nemáme na mysli extrémně sofistikované algoritmy. Špatné pojmenování proměnných, nafouklé třídy a metody, magická čísla — to vše dělá kód chaotickým a těžko pochopitelným.
- **Čistý kód neobsahuje duplicity**.
  - Kdykoli potřebujete změnit duplicitní kus kódu, musíte si pamatovat, že musíte provést stejnou změnu ve všech jeho výskytech. To zvyšuje kognitivní zátěž a zpomaluje vývoj.
- **Čistý kód obsahuje minimální počet tříd a dalších „pohyblivých částí“.**
  - Méně kódu znamená méně věcí, které musíte udržovat v hlavě.Méně kódu znamená méně údržby.Méně kódu znamená méně chyb.Kód je závazek — držte ho krátký a jednoduchý.
- **Čistý kód prochází všemi testy**.
  - Víte, že máte špinavý kód, když projde jen 95 % testů.A víte, že jste v háji, když je pokrytí testy 0 %.
- **Čistý kód je snazší a levnější na údržbu!**
  - Pokud chceme, mohu také přeložit pokračování, nebo připravit příklad kódu ukazující refaktoring do čistší podoby pomocí návrhových vzorů.

TECHNICKÝ DLUH

- Každý programátor se snaží psát vynikající kód hned od začátku. Pravděpodobně neexistuje vývojář, který by úmyslně psal nečistý kód na úkor projektu. Ale kdy se vlastně čistý kód stává nečistým?
- Metafora **„technického dluhu“** ve vztahu k nečistému kódu byla původně navržena **Wardem Cunninghamem**.
- Pokud si někdo vezme půjčku v bance, umožní mu to rychleji realizovat nákupy. Za toto urychlení však zaplatí – nejenže musí splatit jistinu, ale také úroky navíc. Není nutné dodávat, že úroky mohou narůst natolik, že mohou převýšit celkový příjem, a tím učinit splacení nemožným.
- Úplně stejná situace může nastat u kódu. Je možné dočasně zrychlit vývoj tím, že se nenapíšou testy pro nové funkce, ale tato strategie postupně zpomalí další práci, dokud nebude dluh splacen – například právě dopsáním testů.

PŘÍČINY TECHNICKÉHO DLUHU

- **Tlak byznysu**
  - Někdy obchodní okolnosti donutí tým nasadit nové funkce dříve, než jsou zcela dokončené. V takovém případě se v kódu mohou začít objevovat různé provizoria a improvizace, aby se zakryly nedokončené části projektu.
- **Nedostatečné porozumění následkům technického dluhu**
  - Někteří zaměstnavatelé nemusí rozumět tomu, že technický dluh má **„úrok“**, který zpomaluje vývoj, jak dluh narůstá. Kvůli tomu může být složité alokovat čas týmu na refaktoring, protože vedení nevidí jeho okamžitou hodnotu.
- **Neschopnost bojovat proti silné provázanosti komponent**
  - Tato situace nastává, když projekt připomíná monolit místo souboru jednotlivých modulů. V takovém případě má změna jedné části dopad na části ostatní. Týmová práce se stává obtížnou, protože není možné jednoduše izolovat práci jednotlivých vývojářů.

PŘÍČINY TECHNICKÉHO DLUHU II

- **Nedostatek testů**
  - Absence okamžité odezvy podporuje rychlá, ale riskantní řešení. V nejhorších případech mohou být tyto změny nasazeny přímo do produkce bez testování. Následky mohou být katastrofální – třeba zdánlivě neškodný hotfix může odeslat testovací e-mail tisícům zákazníků, nebo ještě hůře, poškodit celou databázi.
- **Nedostatek dokumentace**
  - To zpomaluje začlenění nových členů týmu a může zcela zastavit vývoj, pokud z projektu odejdou klíčoví lidé.
- **Nedostatečná komunikace mezi členy týmu**
  - Pokud není znalostní báze sdílena napříč firmou, lidé pracují se zastaralým porozuměním projektových procesů a informací. Situace se může ještě zhoršit, pokud jsou juniorští vývojáři špatně vedeni mentory.
- **Dlouhodobý paralelní vývoj v několika větvích**
  - To může vést k akumulaci technického dluhu, který se po sloučení změn ještě zvýší. Čím více se změny provádějí izolovaně, tím větší bude výsledný dluh.

PŘÍČINY TECHNICKÉHO DLUHU III

- **Odkládání refaktoringu**
  - Požadavky na projekt se neustále vyvíjejí a v určitém okamžiku může být zřejmé, že části kódu jsou zastaralé a je třeba je přepracovat. Na druhou stranu vývojáři každý den doplňují nový kód, který na těchto zastaralých částech závisí. Čím déle se refaktoring odkládá, tím více kódu bude potřeba v budoucnu přepracovat.
- **Nedodržování společných standardů**
  - To se stává, pokud každý člen týmu píše kód podle sebe – například stejným způsobem, jako psal při předchozím projektu.
- **Inkompetence**
  - To nastává, když vývojář jednoduše neví, jak psát kvalitní a udržovatelný kód.

KDY REFAKTOROVAT?

- Pravidlo tří
  - Když něco děláte poprvé, prostě to udělejte.Když děláte něco podobného podruhé, zakruťte se nevolí, že to musíte zopakovat, ale udělejte to stejně.Když něco děláte potřetí, začněte refaktorovat.
- Při přidávání funkce
  - Refaktoring pomáhá pochopit kód ostatních. Pokud se musíte vypořádat s cizím nepořádným kódem, snažte se jej nejdřív refaktorovat. Čistý kód se mnohem lépe chápe. Zlepšíte ho nejen sami pro sebe, ale i pro ty, kteří jej budou používat po vás.Refaktoring také usnadňuje přidávání nových funkcí. Změny se mnohem snáz provádějí v čistém kódu.

KDY REFAKTOROVAT? II

- Při opravě chyby
  - Chyby v kódu se chovají stejně jako v reálném světě: žijí v nejtemnějších, nejšpinavějších koutech kódu. Uklidíte-li kód, chyby se prakticky odhalí samy.Manažeři oceňují proaktivní refaktorování, protože eliminuje potřebu vytvářet speciální úkoly na refaktoring později. Šťastní šéfové dělají šťastné programátory!
- Během code review
  - Code review může být poslední šancí uklidit kód předtím, než se stane veřejně přístupným.Nejlepší je provádět takové kontroly ve dvojici s autorem kódu. Tak můžete rychle opravit jednoduché problémy a zároveň odhadnout čas na opravu těch složitějších.

JAK REFAKTOROVAT?

- Refaktoring by měl být prováděn jako série malých změn, z nichž každá existující kód nepatrně zlepší, přičemž program zůstává stále funkční.

KONTROLNÍ SEZNAM SPRÁVNĚ PROVEDENÉHO REFAKTORINGU

- **Kód by měl být čistší.**
  - Pokud kód zůstane po refaktoringu stejně nečitelný… pak bohužel právě promarnili hodinu života. Je potřeba zjistit, proč k tomu došlo.
  - To se často stává, když se od ustálených malých kroků odkloní a smíchá se mnoho změn do jedné velké úpravy, ve které se lze snadno ztratit – zvláště pokud je málo času.
  - Může se to stát i u extrémně špatného kódu. Ať se vylepší sebevíc, celek zůstane nepořádek.
  - V takovém případě stojí za to zvážit úplné přepsání části kódu. Ale až poté, co jsou napsané testy a vyhrazena dostatečně velká časová rezerva. Jinak se dostaví výsledky, o kterých byla řeč v prvním odstavci.

KONTROLNÍ SEZNAM SPRÁVNĚ PROVEDENÉHO REFAKTORINGU

- **Během refaktoringu by neměla vznikat nová funkcionalita.**
  - Refaktoring a vývoj nových funkcí by se neměly míchat. Tyto procesy je vhodné oddělit alespoň na úrovni jednotlivých commitů.

KONTROLNÍ SEZNAM SPRÁVNĚ PROVEDENÉHO REFAKTORINGU

- **Všechny existující testy musí po refaktoringu projít.**
  - Nastávají dvě situace, kdy testy po refaktoringu selžou:
    1. **Při refaktoringu nastala chyba.**
       Zřejmé řešení: chybu opravit.
    2. **Testy byly příliš nízkoúrovňové.**
       Například testovaly privátní metody tříd.
  - V takovém případě jsou na vině samotné testy. Je potřeba je zrefaktorovat nebo napsat novou, vyšší vrstvu testů.

  - Skvělým způsobem, jak se takové situaci vyhnout, je psát testy ve stylu BDD (Behavior-driven development).

TO JE VŠE ☺

ZKOUŠKA

- Témata:
  - Odhalení prohřešků v existujícím kódu
  - Refactoring existujícího kódu
  - Návrh struktury SW
- Jak bude zkouška probíhat?