Otázka 17 - Y36SI3

Zadání

Kvalita zdrojového kódu. Duplikace kódu a duplikace dat, příčiny a jak jim předcházet. Princip DRY. Demeterův zákon pro třídy. Generátory kódu, pasivní generátory kódu, aktivní generátory kódu. Refaktoring. Zpětné inženýrství. (Y36SI3)

Slovníček pojmů

- **DRY** - *Don't Repeat Yourself* (nebo také DIE - Duplication Is Evil), princip softwarového vývoje od Andy Hunt a Dave Thomas z knihy Pragmatic Programmer, který se zaměřuje na duplikaci čekoholiv (kódu, data, dokumentace). „Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.“ - Každý znalost musí mít jedinou, jednoznačnou a autoritativní reprezentaci v systému.
- **Demeterův zákon** - princip softwarového vývoje zejména toho objektového, jedná se o konkrétní případ volné vazby (loose coupling). Daný objekt by měl o struktuře ostatních objektů předpokládat co nejméně.
- **Refaktoring** - Proces úpravy zdrojového kódu beze změny jeho vnější funkčnosti tak, abychom vylepšili nefunkční atributy softwaru (např. čitelnost kódu, výkon, udržovatelnost, architektura).
- **Zpětné inženýrství** (reverse engineering) - proces, kdy se snažíme zjistit, jakou má něco strukturu, jak to funguje a pracuje. Například máme program, ke kterému není dokumentace ani zdrojové kódy, ale je pro nás důležitý a potřebujeme ho nějak upravit nebo zjistit, jak funguje, tak ho například můžeme prohnat disassemblerem a z kódu zjistit, jak pracuje.

Email od Stoklasy
Ještě doplnění k otázce 17, rozdíl mezi Aktivními a pasivními generátory kódu: není tak podstatné jestli se generování děje za běhu nebo ne. Jde o to jestli je generování kódu jednorázové na začátku vývoje (pasivní generátory), nebo jestli generování kódu probíhá při každé kompilaci programu (aktivní generátory). Generátory kódu které generují kód za běhu jsou tedy podmnožinou aktivních generátorů kódu ale nemusejí nutně generovat kód za běhu, mohou ho generovat i při každém překladu.

Kvalita zdrojového kódu

Kvalita zdrojového kódu je jen velmi těžko definovatelný pojem a dá se velmi špatně měřit, respektive je třeba si definovat kritéria dle kterých budeme kvalitu posuzovat. Obecně lze ale aplikovat známa pravidla jako jsou **ortogonalita**, **DRY**, **Demeterův zákon** a další, které nám zajistí alespoň určitou úroveň kvality.

Ortogonalita

- **Moduly** projektu musí být na sobě maximálně **nezávislé**. Kolik modulů se musí změnit, pokud změníme požadavek? Ideálně pouze jeden!
- **Volná vazba** (loose coupling) - třídy jsou provázány přes svoje rozhraní, ne implementace.
- **Oddělení politiky od mechanizmu**.
- Využití **abstrakce**.
- Detaily v metadatech.
- Více možností - volba v konfiguračním souboru.

Duplikace

- Stejná data na dvou místech.
- Stejný kód na dvou místech.

Příčiny

- **Copy - Paste**.
- **IDE** a jeho **průvodci** (wizard).
- **Neznalost** programátora - neví, že něco v systému již je.
- **Optimalizace** - předčasná optimalizace něčeho (např. denormalizace databáze), aniž bychom měli ověřeno, že to bude výkonový problém.

Jak se vyhnout duplikaci

Duplikaci dat se lze vyhnout **normalizovanou** databází (normální formy).

Duplikace kódu:

- **Revize** (code review) - formální, neformální.
- Stanovením **kanonické implementace** (= normativní/výchozí implementace) (technický vedoucí SW týmu) a ostatní se musí přepsat.
- **Komunikace** mezi programátory.
- **Generátory kódu** - Ruby on Rails, PHP frameworky a další.
- **Refaktoring** - přišlo z komunity jazyka Smalltalk, např. přidání parametru metody, spojení podobných metod do metody rodiče, odvození podobných tříd od společného rodiče, průběžný unit testing.
- **MDA** (Model Driven Architecture)

DRY

Don't Repeat Yourself - princip zmíněný v knize Pragmatic Programmer od Andy Hunta a Dava Thomase.

Princip, který se snaží zabránit duplikaci a je velmi jednoduchý - neměli bychom dvakrát psát něco, co už jsme jednou napsali. Jeho úspěšnou aplikací získáme to, že **změnou** jedné věci **nenarušíme funkci** jakékoliv jiné. Využívá různých *generátorů* kódu, *modelů*, *transformací* atd.

Při psaní jakéhokoliv systému bychom se měli snažit mít veškerá data a kód pouze **v jedné instanci**, včetně dokumentace, schéma databáze, atd. Dovedeno do extrému to znamená, že například vytvoříme schéma databáze (SQL skript, to bude **kanonická reprezentace** této informace) a kód pro přístup do této databáze nebudeme psát ručně (to by byla duplikace), ale necháme ho vygenerovat automaticky ze schématu.

To nám zajistí, že **pokud se změní** schéma databáze, automaticky jsme schopni **přegenerovat kód** a vše **bude fungovat dále**, zatímco při manuálním psaní bychom museli kód přepsat a určitě by v něm byly *chyby*. Samozřejmě v praxi to takto extrémně dělá málokdo a musí se najít nějaký rozumný kompromis.

Demeterův zákon

Demeterův zákon nebo také princip nejmenších znalostí (Principle of Least Knowledge). Jedna z metod jak dosáhnout ortogonality a volné vazby v objektově orientovaném programovacím jazyce.

Metoda objektu smí volat pouze:

- Jiné metody objektu.
- Metody objektů které vlastní.
- Metody objektů které vytvořila.
- Metody objektů které jsou parametrem metody.
- Případně metody globálních objektů.

Jednodušeji řečeno, objekt smí volat objekty pouze přímo (ve většině OOP jazyků to znamená použít *pouze jednu tečku*), např. **objekt.metoda()** a neměl by používat konstrukce jako objekt.getDalsiObjekt().metoda().

Výhodou Demeterova zákona je lépe spravovatelný a upravitelný kód. Nevýhodou je zdlouhavější volání některých metod, respektive nutnost tvořit jednoúčelové metody, které pouze volají nějakou jinou metodu.

Generátory kódu

Program, který generuje zdrojový kód na základě nějakých dat (model, SQL skript, šablona).

Obrovskou **výhodou** generátorů je fakt, že **z jednoho zdroje** (model, SQL skript, šablona) můžeme vygenerovat zdrojový kód v „nekonečném“ **množství jazyků** (*Java, C#, C, PHP, Haskell, …*) a zároveň možnost **přegenerovat** kód v případě změny. Toto je mnohem **rychlejší** a **bezpečnější** (z hlediska tvorby chyb) než manuální tvorba takového kódu.

Naopak určitou **nevýhodou** je fakt, že tvorba složitějšího generátoru **může být časově náročná** a v dnešní uspěchané době plné deadlinů se většina programátorů raději věnuje tvorbě produkčního kódu namísto podpůrných utilit jako jsou generátory.

Dělí se na:

- **Aktivní** - součást build skriptu nebo generování za běhu programu.
- **Pasivní** - průvodci v IDE, šablony v IDE.

Napsat jednoúčelový generátor není těžké. Např.:

- Práce s textem.
- Procházení textového souboru/XML stromu, rozhodování na základě atributů.
- Popis modelu.
- Šablony pro různé typy výstupu.
- Tip: jednoduché generování kódu v Excelu =CONCATENATE(„insert into Osoby(Jmeno,Plat) values (‘“;A1;“’“,“; B1;“)“)

Aktivní

Generování kódu za běhu (on-the-fly)

- Příkaz eval v PHP.
- Reflexe - Java, C# a další jazyky.
- Ken Thompson, tvůrce Unixu, generoval strojový kód z regulárních výrazů a ten hned spouštěl.

Aktivní generování kódu za běhu je velmi **pokročilá technika**, která skrývá neuvěřitelné možnosti, ale zároveň je velmi složitá. Příkaz **eval** z PHP (obecně není doporučováno ho používat) nám umožňuje **interpretovat text**, takže kdyby nám uživatel zadal do textového pole kus PHP kódu, my ho můžeme spustit. To se nám sice může často hodit, ale zároveň to skýtá obrovská **bezpečnostní rizika** (mohl by spustit shell a smazat nám stránky apod.).

Podobně bychom mohli za běhu generovat kód v Javě, překládat ho a za použití **reflexe** ho spouštět.

Pasivní

- IDE - NetBeans, Eclipse, …
- Hibernate/NHibernate

- Ruby On Rails
- PHP frameworky - CakePHP
- Axis2 - generování Java kódu z WSDL a naopak
- NBusiness - (C#)
- ADO.NET Entity Framework

Jedněmi z nejčastěji používaných pasivních generátorů jsou IDE, která obsahují nespočet různých **průvodců**, které automatizují tvorbu kódu/konfiguračních souborů, které bychom jinak museli napsat sami. Obzvláště u složitých technologií jako je J2EE (hlavně ve starších verzích) to je hodně nápomocné.

Výhodou je **rychlost** a možnost rychle **zkoušet** nové a nové věci. **Nevýhodou** naopak **odstínění** od dané technologie (v případě problémů přesně nevíme, co se děje, protože většinu kódu jsme nenapsali sami!, technologii mohou využívat i úplní amatéři, kteří tomu vůbec nerozumí).

Mezi další populární generátory patří různé **frameworky**. Například *Ruby on Rails* obsahuje konzolovou aplikaci, která za programátora vygeneruje téměř všechen kód (modely z databáze, kontrolery, lokalizační soubory, …). Ten pak stačí **poupravit** pro naše potřeby. Podobně funguje i CakePHP a další.

Refaktoring

V případě, že náš kód trpí příznaky **Code Smell** (duplikace kódu, dlouhé metody, špatné zapouzdření, těsné vazby, celkově příliš složité), vyplatí se využít refaktoring pro jeho zlepšení, tedy přepracovat zdrojový kód, případně i architekturu tak, aby **funkce zůstala nezměněna**, ale **udržovatelnost** a **čitelnost** kódu (kvalita) byla výrazně vylepšena. Refaktoring byl nejvíce proslaven knihou od Martina Fowlera.

Pro refaktoring se vyplatí využít služeb IDE (Eclipse, IntelliJ IDEA, NetBeans).

Metody refaktoringu

- **Zapouzdření** atributu - přístup pouze přes getter/setter.
- **Generalizace** - vytvoříme předka více podobných tříd a ty od něj dědí.
- Nahrazení podmínek **polymorfismem**.
- **Extrakce metody** - větší metodu rozdělíme na více menších, je to pak více srozumitelné a menší kousky lze i použít jinde.
- **Extrakce třídy** - vytvoření nové třídy, která obsahuje nějaké funkce z jiné třídy.
- **Přesun** metody/atributu do jiné třídy (více vhodné).
- **Přejmenování** metody/atributu, aby jméno lépe vystihovalo účel metody/atributu.
- **Pull Up** - přesunutí metody/atributu do předka.
- **Push Down** - přesunutí metody/atributu do potomka.

Více naleznete v odkazech.

Zpětné inženýrství

Zpětné inženýrství (reverse engineering) je proces, kdy se snažíme zpětně zjistit, jakou má něco *strukturu* nebo **jak** to *funguje*. Běžně máme při vývoji dostupné zdrojové kódy nebo alespoň slušnou dokumentaci, v takovém případě zpětné inženýrství mnoho nevyužijeme. Ale jsou i situace, kdy nemáme k dispozici nic (častý případ starých - legacy - systémů nebo kód staženého od neznámého autora na internetu) a přesto potřebujeme do programu nějak proniknout a zjistit, jak funguje.

Příkladem může být i portování Windows ovladačů na operační systém Linux, kdy výrobce zařízení *neposkytne* ani ovladač pro Linux, ani *žádnou dokumentaci* k zařízení. V takovém případě musí vývojář využít **disassembler** a pochopit, jak se se zařízením *komunikuje* atd.

Někdy máme k dispozici **zdrojové kódy**, ale pokud je to program *větší a složitější*, je těžké se v něm zorientovat. V takovém případě můžeme využít zpětného inženýrství k získání UML **diagramu** (případně jiných diagramů, např. tok řízení, tok dat) automatickou **analýzou** zdrojového kódu. Z diagramu jsme schopni pochopit základní **strukturu a organizaci** programu a odtud se pak můžeme ponořit přímo do zdrojáků.

Typy zpětného inženýrství:

- Ze **zdrojového kódu** do nějakého **diagramu** (UML, tok dat, tok řízení).
- **Analýzou komunikace** - např. po síti, komunikace se zařízeními.
- **Disassembler** - z binárky se vytvoří assembler kód, kterému lze alespoň trochu porozumět.
- **Dekompilace** - pokus o vytvoření původního kódu v některém z vyšších jazyků z binárky nebo bytekódu.
- **Cracking** - snaha o prolomení ochrany softwaru (licence).