AHEAD
THROUGH
KNOWLEDGE

PROFINIT

Design and Construction

Jiří Toušek, Vlastimil Jinoch

3. a 10. ledna 2018

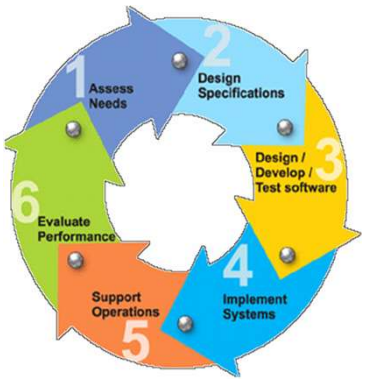
Obsah

1. Design a Návrhové vzory
2. Návrh v malém
3. Zásady elementárního návrhu
4. Konstrukce
5. Jak psát kvalitní kód

PROFINIT

Design

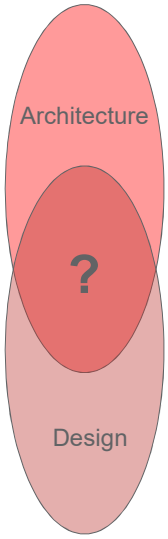
Softwarový proces



PROFINIT

4

Architektura vs. Design



Software architecture

- Realizace **nefunkčních** požadavků
- Strategický design
  - Programovací paradigmata, architektonické styly, principy, standardy, …

Software design

- Realizace **funkčních** požadavků
- Taktický design
  - Design patterns, programovací idiomy, refaktoring, …

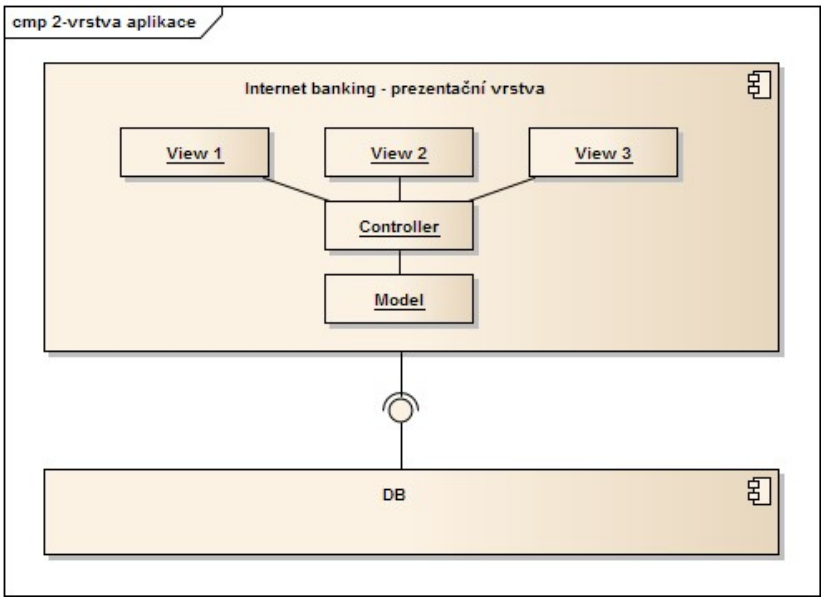
*„Architecture is about the **important** stuff. Whatever that is …“*
Martin Fowler, Who needs an Architect ?

PROFINIT

5

Od drobného SW k robusní aplikaci

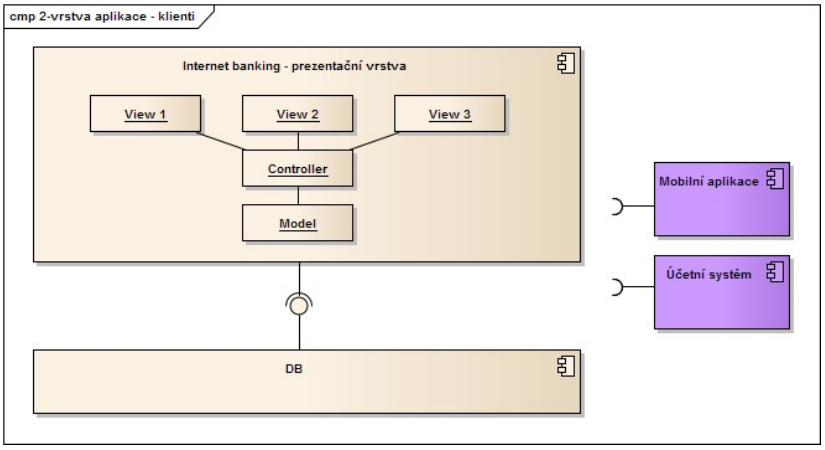
Jednoduchá aplikace



PROFINIT

7

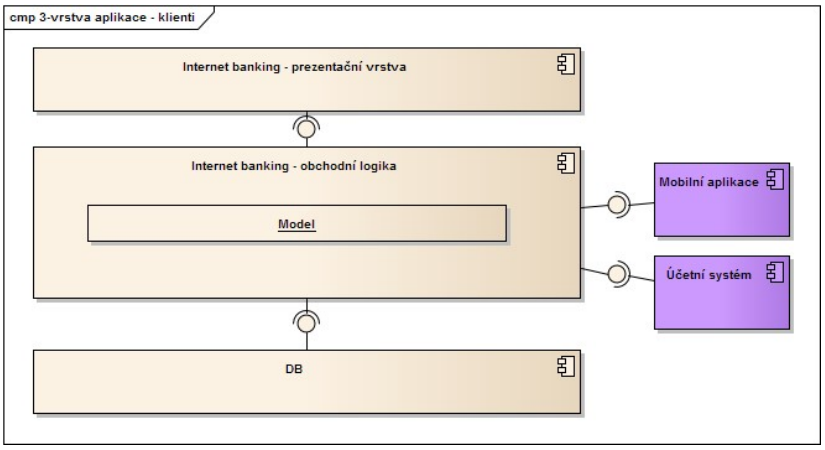
Další klientské aplikace



PROFINIT

8

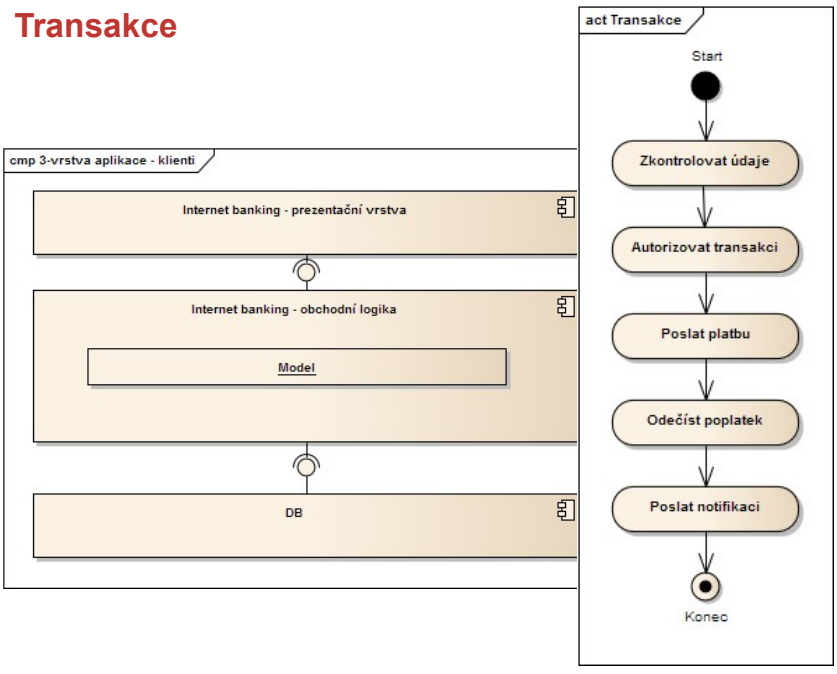
Třívrstvá aplikace



PROFINIT

9

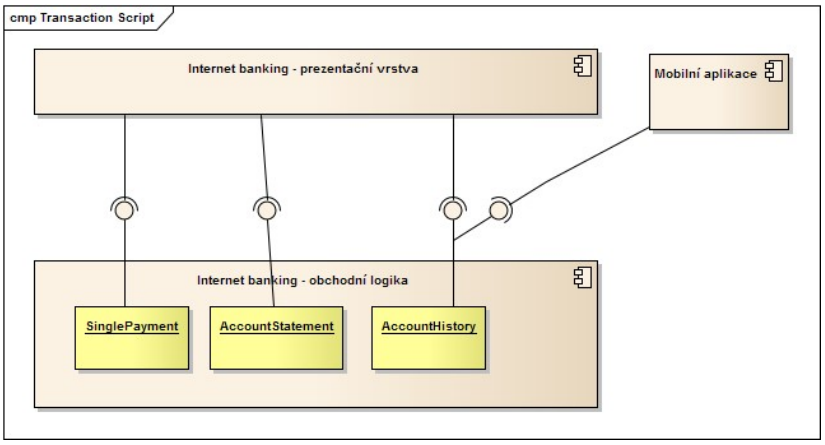
Transakce



PROFINIT

10

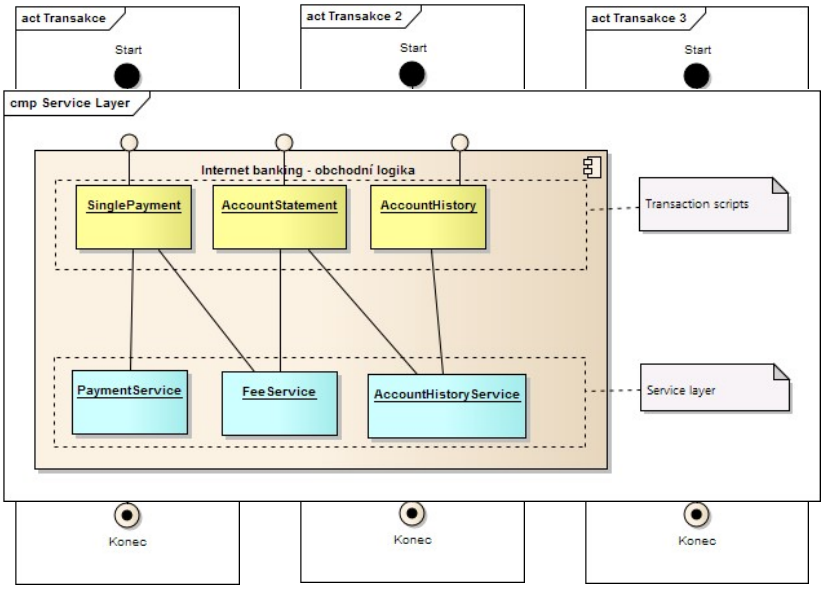
Transaction Script



PROFINIT

11

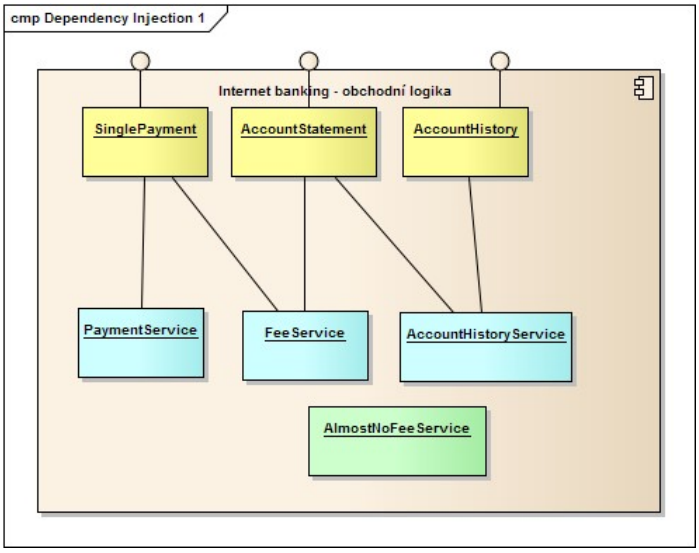
Service Layer



PROFINIT

12

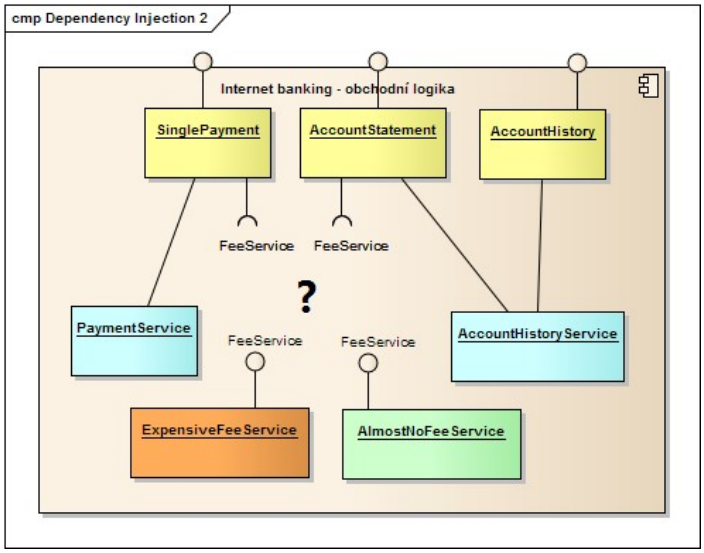
Dependency Injection 1



PROFINIT

13

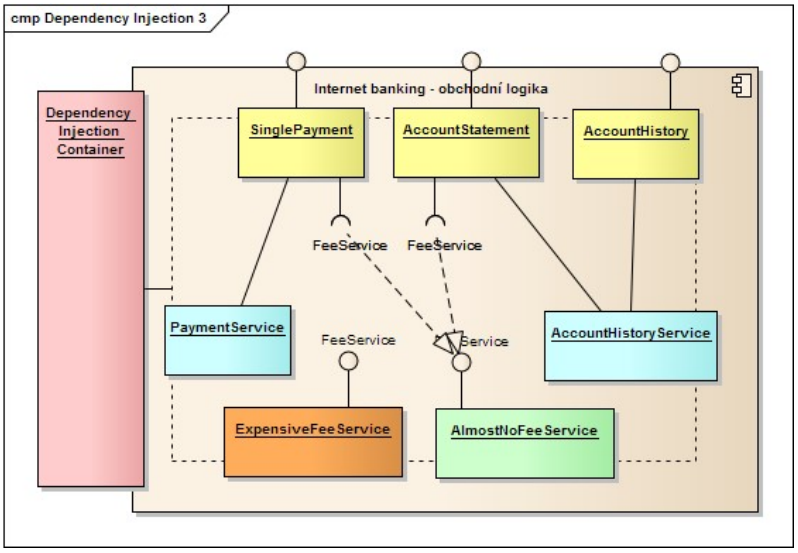
Dependency Injection 2



PROFINIT

14

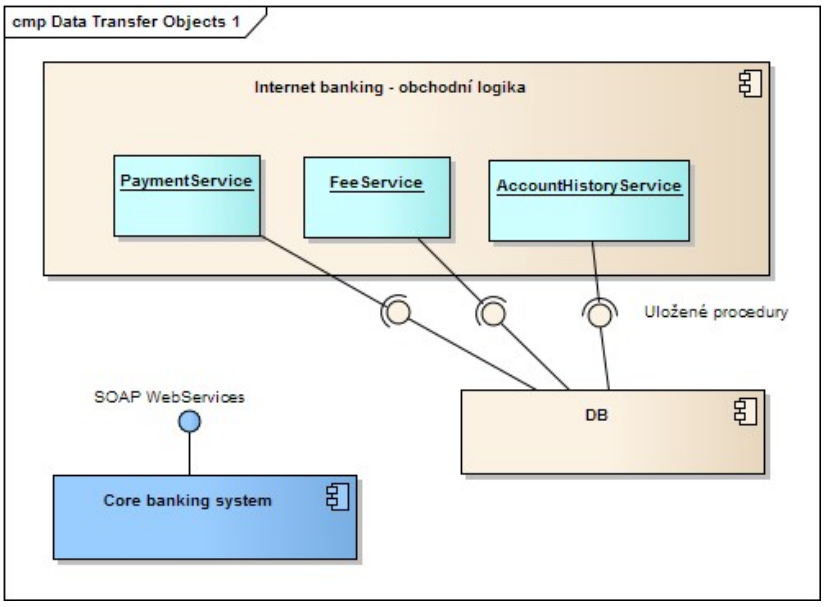
Dependency Injection 3



PROFINIT

15

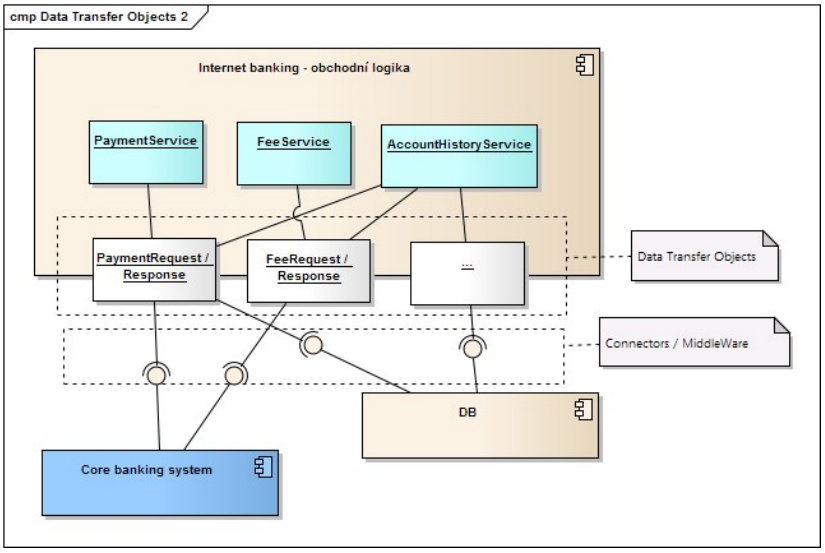
Data Transfer Objects 1



PROFINIT

16

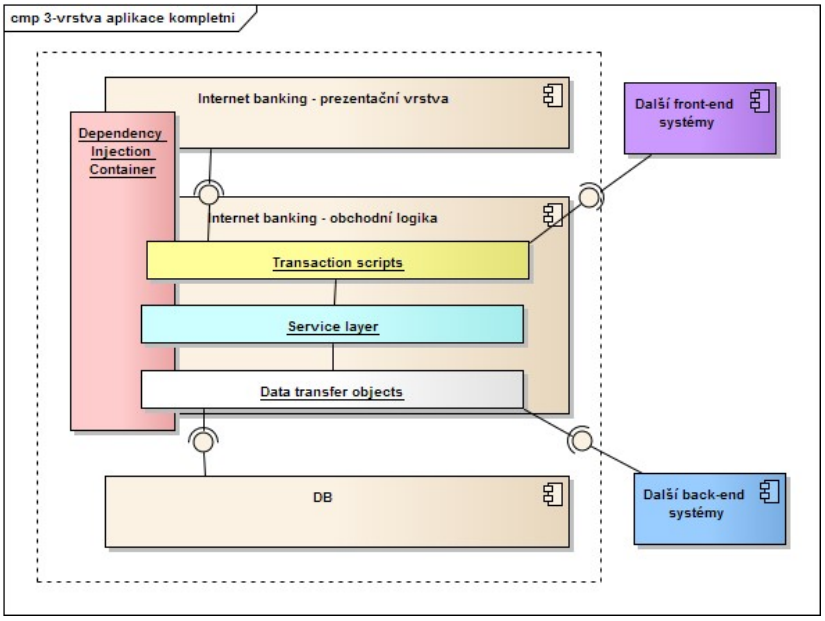
Data Transfer Objects 2



PROFINIT

17

Třívrstvá aplikace – implementační pohled

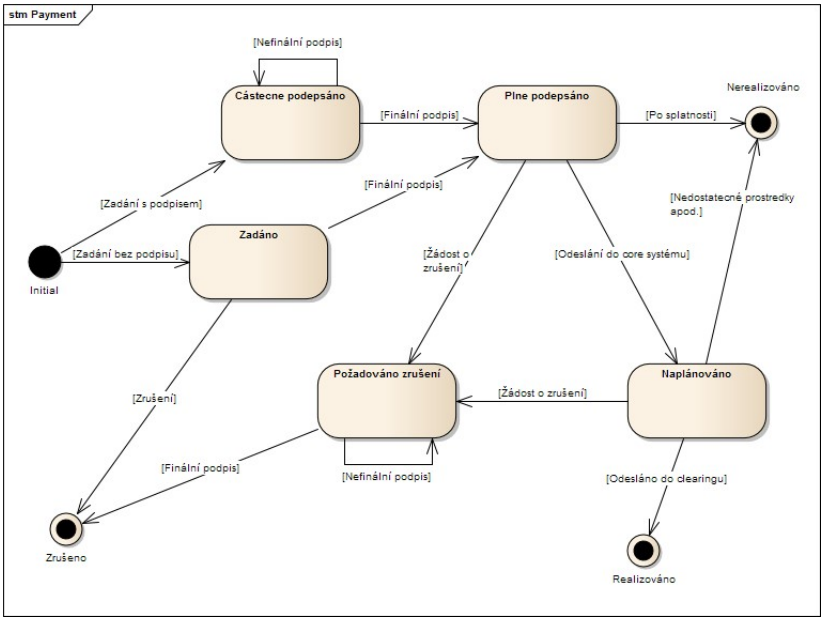


PROFINIT

18

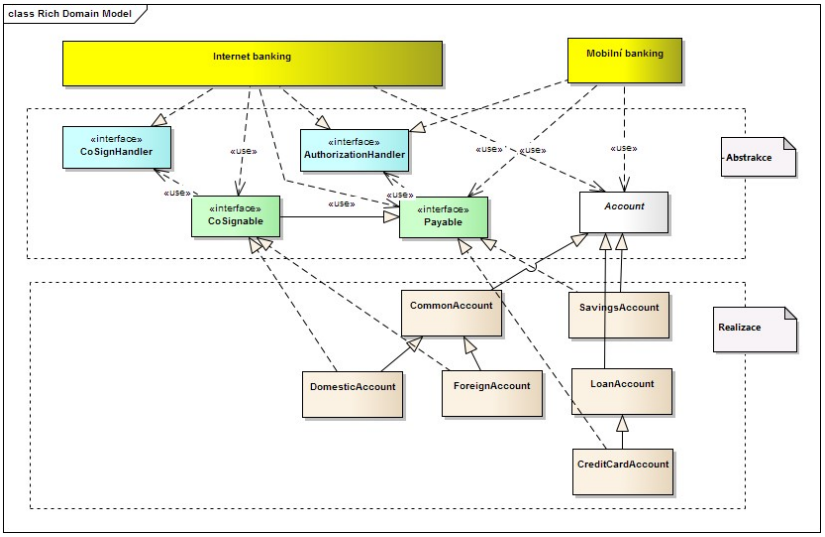
Návrhové vzory

Procedurální programování



PROFINIT

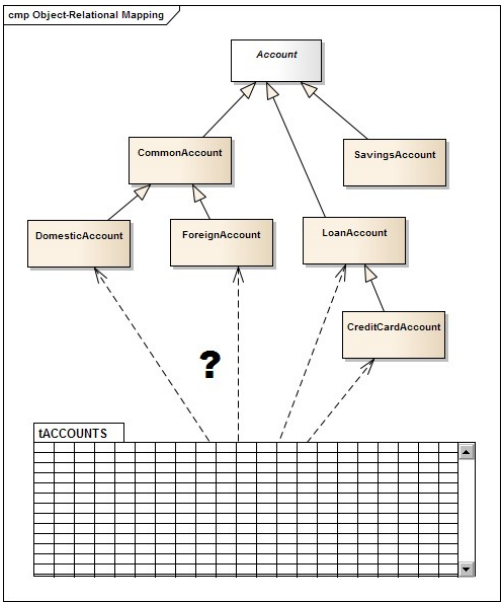
Rich Domain Model



PROFINIT

21

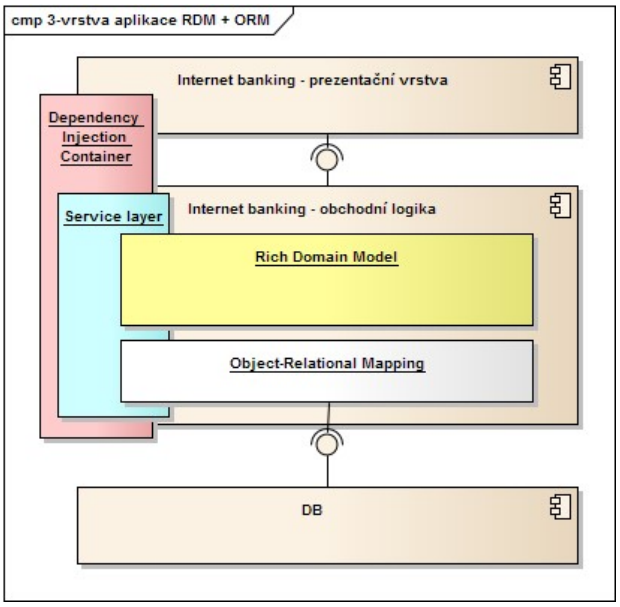
Object – Relational Mapping (ORM)



PROFINIT

22

„Dnešní“ aplikace (kolem r. 2009)

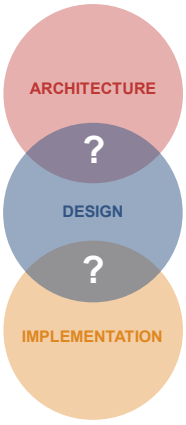


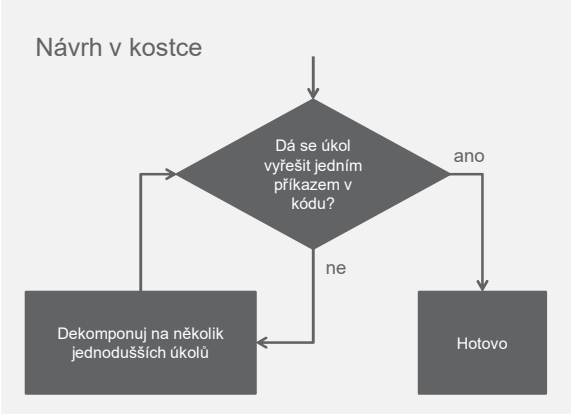
PROFINIT

23

Návrh v malém

Architektura vs. návrh vs. implementace





PROFINIT

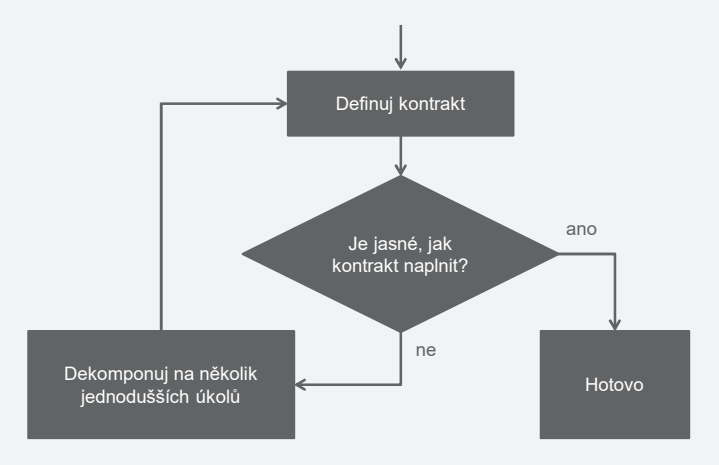
25

Kontrakt

PROFINIT

26

Návrh v kostce – lépe



PROFINIT

27

Technický dluh

- Jakékoliv rozhodnutí, které ztíží budoucí údržbu a rozvoj
- Úroky v podobě:
  - Dražšího rozvoje
  - Složitějších (a tím i dražších a delších) oprav chyb
  - Zbytečně zanesených chyb
  - Delšího zaučování nových vývojářů
  - …
- Co s ním?
  - Splatit i s úroky
    - Tj. stálo nás to původní implementaci, úpravu odstraňující dluh a úroky
  - Splácet doživotně úroky
    - Tj. stálo nás to původní implementaci a bude nás to stát další a další úroky
  - Zahodit a začít znovu
    - Tj. stálo nás to původní implementaci a úroky a na konci nemáme nic

PROFINIT

28

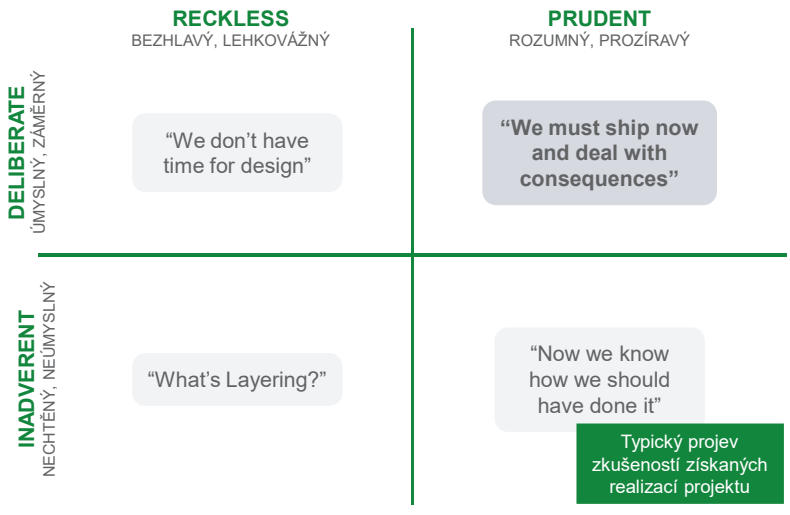
Technický dluh – příklad

- Přebrali jsme systém s 20 importy od třetích stran
- Každý import byl napsán znovu s vydatným využitím copy-paste
  - Vývoj importu ~ 20MD
  - Ruční testování ~ 5MD
  - Automatické testy nejsou
  - První rok typicky ~ 30MD na opravu chyb, i dále pak hodně chybové
- Splacení dluhu:
  - Nová architektura, řešení paralelního běhu se starým systémem: 120MD
  - Infrastruktura pro automatické testy: 40MD
  - Přepsání importů do nové architektury: 100MD
- V součtu:
  - Splacení dluhu stálo cca. 250MD (u aplikace zhruba 20x menší než IB ČS)
  - Přesto se vyplatilo už po 2-3 nových importech

PROFINIT

29

Technical debt



Více informací na http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

PROFINIT

30

Základy dobrého (OOP) návrhu

- Decomposition
- Abstraction

- High cohesion
- Loose coupling
- Encapsulation

PROFINIT

31

Cohesion (soudržnost)

Service1
+ overHeslo () : boolean + tiskniFakturu () : void

Service2
+ vytvorUzivatele () : void + tiskniObjednavku () : void

VS.

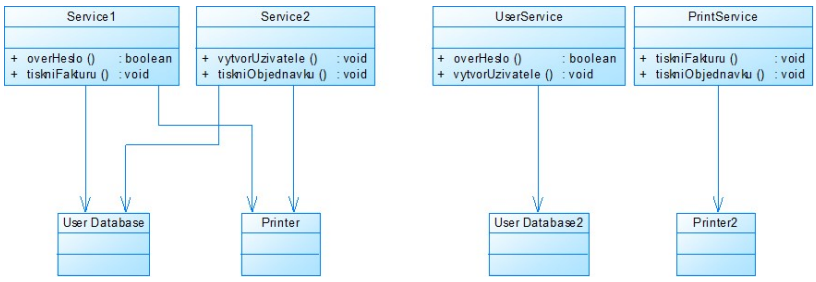
UserService
+ overHeslo () : boolean + vytvorUzivatele () : void

PrintService
+ tiskniFakturu () : void + tiskniObjednavku () : void

PROFINIT

32

Coupling (provázanost)



PROFINIT

33

Loose coupling

- Cíl: Modul (třída, …) má co nejméně záviset na svém okolí
- Výhody:
  - Odolnost proti změnám okolí
  - Snazší pochopení
  - Znovupoužitelnost
  - Testovatelnost
- Android `Intents`, Spring IoC
- Pozor na skryté vazby
  - Časové
  - Souslednost událostí

```
/*
 * ...
 * Pozor! Tato metoda musí být volána až po metodě …
 */
```

PROFINIT

34

Encapsulation

- Cíl:
  - Znemožnit okolí záviset na mých implementačních detailech
  - Ochránit vnitřní stav před vnějšími zásahy
- Výhody:
  - Omezení dopadu změn na okolí („Encapsulate what changes“)
  - Snazší testování a verifikace
  - Snazší debugging
- Interfaces
  - Ultimátní podoba zapouzdření – implementační detaily tu vůbec nejsou
  - Jasně definují kontrakt a oddělují ho od implementačních detailů
- Microservices

PROFINIT

35

Třívrstvá architektura

- Rozhraní vůči uživatelům
- Jádro – vlastní aplikační logika
- Komunikace s okolím

PROFINIT

36

Zásady elementárního návrhu

DRY – Don’t Repeat Yourself

- Každá informace (algoritmus, myšlenka, nastavení, …) má být v aplikaci jen jedenkrát
  - „Single Source of Truth“
- Pokud to tak není:
  - Kód je větší, složitější a nepřehlednější
  - Každá oprava nebo úprava se musí udělat na více místech
    - Dražší
    - Určitě na to někdy zapomenete
  - Snadno ztratíte „jediný zdroj pravdy“ / jedinou verzi pravdy

PROFINIT

38

DRY – příklad

- Při analýze technického dluhu u zákazníka jsme se dozvěděli:
  - „20 verzí importů pro 20 klientů => 20x více kódu, o který je potřeba se starat a rozumět mu (od analýzy, přes vývoj, testování, po provoz a aplikační podporu)“
  - „násobně dražší testování, mnoho chyb“
  - „nikdo reálně neví co to dělá a co to dělat má“

PROFINIT

39

SRP – Single Responsibility Principle

- Každý logický celek (modul, třída, metoda, …) má dělat vždy jen jednu věc
  - Co když budu potřebovat přepoužít jen jeden z několika kroků?
- Symptom: inline komentáře ve stylu: `// krok 2: teď uděláme …`

```
// Spocitej poradi, ve kterem je nutne parsovat DDL skripty
…

// Odstran existujici DDL skripty ve vystupnim adresari
…

// Pro kazdy databazovy objekt ve spocitanem poradi ziskej jeho
// DDL a zparsuj ho
for (DdlName ddl : extractionOrder) {
    …
    // Zparsuj DDL
    …
    // Uloz DDL
```

PROFINIT

40

Broken Windows

PROFINIT

41

Broken Windows

„Stejně už je to zprasený“

PROFINIT

42

Fail Fast

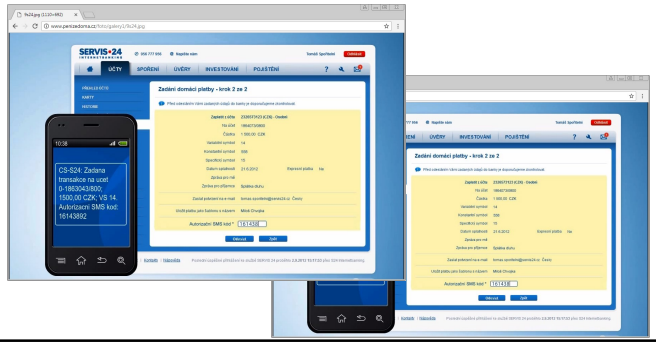
- Je-li v programu chyba, měl by selhat co nejdříve
- Aktivně kontrolujte konzistenci a ošetřujte možné chyby
- „Dead programs tell no lies“
- Čím dřív chybu odchytíte, tím víc informací můžete poskytnout o tom, co se stalo

PROFINIT

43

Další dobré rady

- Premature optimalization is root of all evil
- Jak implementovat vlastní cache – rada první: nedělejte to
- Jak implementovat vlastní transakce – viz předchozí bod
- Jak implementovat vlastní lazy fetching, zámky, … – však víte
- Bezpečnost nelze do kódu přidat dodatečně
- Thread-safety nelze do kódu přidat dodatečně



PROFINIT

44

YAGNI a KISS

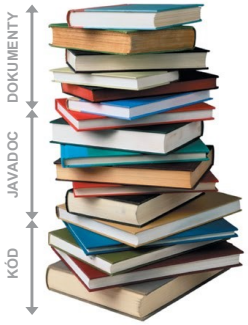
- YAGNI – You Ain't Gonna Need It
- KISS – Keep It Simple Stupid

PROFINIT

45

Self-documenting code

- Kód, který se svou formou (strukturou, jmennými konvencemi apod.) snaží omezit nutnost číst dokumentaci
- Neznamená úplnou absenci dokumentace
  - čitelný kód nenahradí koncepční dokumentaci (architektura, design moments, …)
- Problémy při chybějící dokumentaci
  - Význam větších funkčních celků
  - Pre/post conditions, invariants
  - Inheritance – kontrakt vůči potomkům



PROFINIT

46

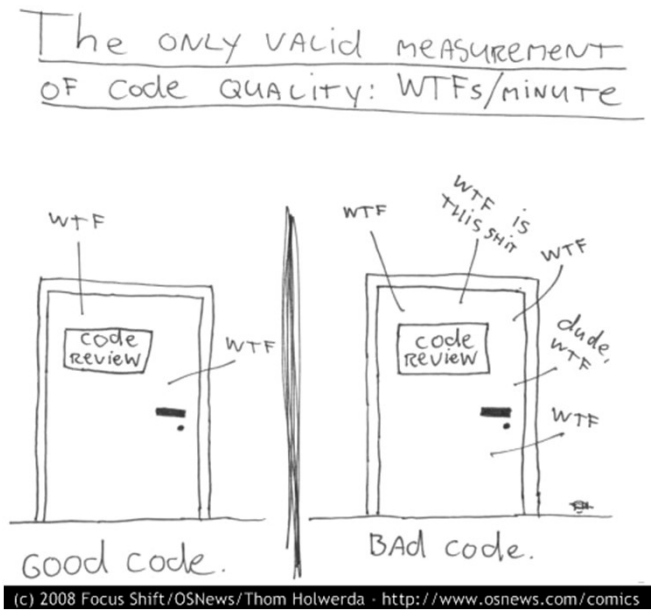
Co je to kvalitní kód?

Kvalitní kód je…

- Bezchybný
- Rychlý / Efektivní
- Krátký?
- Rafinovaný?
- **Srozumitelný**
- **Udržovatelný**

PROFINIT

48



PROFINIT

49

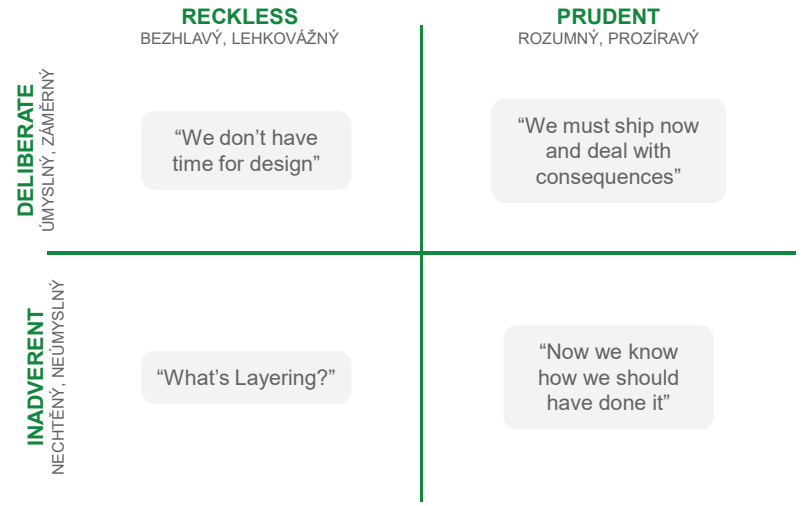
Proč psát srozumitelný kód

- Kód píšete jednou, číst ho (vy nebo někdo jiný) budete pravděpodobně vícekrát
- Pokud váš kód někdo nepochopí, snadno v jeho použití nebo úpravě udělá chybu
- Nesrozumitelný kód nikdo nebude chtít používat

PROFINIT

50

Kdy nepsat kvalitní kód?



PROFINIT

51

Jak psát srozumitelný kód

Co bude uživatel vaší metody číst

1. Jméno metody
2. Deklarované typy parametrů a návratové hodnoty
3. Jména deklarovaných parametrů
4. Javadoc
5. Samotný kód metody

› Čím dříve se dozví vše, co potřeboval, tím dříve může přestat číst (a ušetřit čas)

PROFINIT

53

Příklad: StringUtils

```
join()
```

PROFINIT

54

Příklad: StringUtils

```
public static String join(Object[], String, int, int)
```

PROFINIT

Příklad: StringUtils

```
public static String join(Object[] array, String separator, int startIndex, int endIndex)
```

PROFINIT

Příklad: StringUtils

```
/**
Joins the elements of the provided array into a single String containing the provided list of elements.

No delimiter is added before or after the list. A null separator is the same as an empty String (""). Null
objects or empty strings within the array are represented by empty strings.

 StringUtils.join(null, *)               = null
 StringUtils.join([], *)                 = ""
 StringUtils.join([null], *)             = ""
 StringUtils.join(["a", "b", "c"], "--") = "a--b--c"
 StringUtils.join(["a", "b", "c"], null) = "abc"
 StringUtils.join(["a", "b", "c"], "")   = "abc"
 StringUtils.join([null, "", "a"], ',')  = ",,a"

Parameters:
    array the array of values to join together, may be null
    separator the separator character to use, null treated as ""
    startIndex the first index to start joining from. It is an error to pass in a start index past the end
        of the array
    endIndex the index to stop joining from (exclusive). It is an error to pass in an end index past the end
        of the array
Returns:
    the joined String, null if null array input
*/
public static String join(Object[] array, String separator, int startIndex, int endIndex)
```

PROFINIT

57

Příklad: StringUtils

```
/**
Joins the elements of the provided array into a single String containing the provided list of elements.

No delimiter is added before or after the list. A null separator is the same as an empty String (""). Null
objects or empty strings within the array are represented by empty strings.

 StringUtils.join(null, *)               = null
 StringUtils.join([], *)                 = ""
 StringUtils.join([null], *)             = ""
 StringUtils.join(["a", "b", "c"], "--") = "a--b--c"
 StringUtils.join(["a", "b", "c"], null) = "abc"
 StringUtils.join(["a", "b", "c"], "")   = "abc"
 StringUtils.join([null, "", "a"], ',')  = ",,a"

Parameters:
    array the array of values to join together, may be null
    separator the separator character to use, null treated as ""
    startIndex the first index to start joining from. It is an error to pass in a start index past the end
        of the array
    endIndex the index to stop joining from (exclusive). It is an error to pass in an end index past the end
        of the array
Returns:
    the joined String, null if null array input
*/
public static String join(Object[] array, String separator, int startIndex, int endIndex)
```

PROFINIT

58

Příklad: StringUtils

```
public static String join(Object[] array, char separator, int startIndex, int endIndex) {
    if (array == null) {
        return null;
    }
    int bufSize = (endIndex - startIndex);
    if (bufSize <= 0) {
        return EMPTY;
    }

    bufSize *= ((array[startIndex] == null ? 16 : array[startIndex].toString().length()) + 1);
    StrBuilder buf = new StrBuilder(bufSize);

    for (int i = startIndex; i < endIndex; i++) {
        if (i > startIndex) {
            buf.append(separator);
        }
        if (array[i] != null) {
            buf.append(array[i]);
        }
    }
    return buf.toString();
}
```

PROFINIT

59

Název metody

- To první a často jediné, co z vaší metody bude většina lidí číst – věnujte mu pozornost
- Měl by být stručný a přitom dostatečně popisovat, co metoda dělá
- Pokud se vám nedaří vhodný název vymyslet, může to být špatným návrhem metody
  - Metoda dělá více různých věcí (měla by dělat vždy jen jeden svůj úkol)
  - Nemáte vlastně jasno, co má metoda dělat
  - …
- Pokud jste při vymýšlení jména přišli na špatný návrh metody, ušetřili jste si dost času, který byste jinak strávili později přepisováním kódu
- Kvíz: co dělá tato metoda?
  `dontStoreDataToCache()`

PROFINIT

60

Název metody

```
isSaveBasicDataAndTemplateAttributeValuesDeactivateDocumentReturnPossible()
```

PROFINIT

61

Jména a typy parametrů

› Jméno by mělo popisovat, co daný objekt reprezentuje – nejen sám o sobě, ale i v kontextu použití parametru:

```
void merge(Graph graph1, Graph graph2)
```
vs.
```
void merge(Graph sourceGraph, Graph targetGraph)
```

› U typovaných jazyků využívejte datové typy
- Toto by měla být naprostá samozřejmost
- Speciálně v Javě se ale často setkávám zejména s ignorováním enums (místo enumu se často používají booleany nebo soustavy konstant)
- Zdá se, že velmi populární je místo „strongly typed“ používat „stringly typed“ přístup:
  - Text: `"abc"`
  - Číslo: `"42"`
  - Objekt obsahující jednu textovou a dvě číselné položky: `"abc_42_666"`
  - Přístup k první číselné položce objektu: `myObjectString.split("_")[1]`

PROFINIT

62

Parametry

- Metoda s příliš velkým počtem parametrů je nepřehledná

```
protected PdfPTable getPrijemcePlneniTable(float documentWidth,
   String nazevLeasingovky, String cisloLeasingoveSmlouvy, String
   smluvniServis, String prijemceCisloUctu, String prijemceJmeno,
   String prijemceUliceCP, String prijemceObecPSC, boolean
   pojisteniJindeExistuje, String pojisteniJindeNazev, Boolean
   dphAnoNeNull, String zpusobUhradyKey, String datum, boolean
   isDPH, String sTextPodPodpisem)
```

- Parametry typu boolean nejsou příliš vhodné
  - Nutno v názvu parametru jasně deklarovat, co znamená true a co false
    - Jméno parametru ale nemusí být vždy dostupné (např. v Javě, mám-li knihovnu bez javadoc a zkompilovanou bez jmen parametrů, vidím jen typ bez jména)
  - Při volání metody není na první pohled význam boolean parametru obvykle zřejmý
    - `newTable(tableName, parentDatabase, true)`
    - `newTable(tableName, parentDatabase, TableType.GLOBAL_TEMPORARY)`

PROFINIT

63

Javadoc

- Dohodněte se na jazyku a dodržujte ho
  - V angličtině může být problém se vyjádřit (zejména odborné pojmy)
  - Komentáře v češtině vyžadují neustále přepínat klávesnici a jazyk „v hlavě“
  - Komentáře v „cestine“ nevypadají dobře
- Dokumentujte okrajové stavy, nestandardní případy použití
- Dokumentujte chování vůči null (všude tam, kde to není očividné)
  - U atributů objektu / třídy – zda může někdy být null a co to znamená
  - U parametrů metod – zda může být null a co se v tom případě stane
  - U návratových hodnot – zda a v jaké situaci může metoda vrátit null
- U kolekcí a polí vnímejte rozdíl mezi prázdnou kolekcí a null
  - Většinou dává lepší smysl prázdná kolekce („seznam nalezených prvků je prázdný“) než null („seznam nalezených prvků neexistuje“)
  - Nenechte se vést leností nebo dojmem větší efektivity / úspory paměti při použití null (Java: `Collections.emptyList()`)

PROFINIT

64

Kód

- Dodržujte konvence daného jazyka (jména, formátování, …)
- Orientace na čtenáře – kód bude čten víckrát, píšete jej jednou
  - Lenost zde není platným argumentem
- Jména proměnných (a všeho ostatního) volte dostatečně srozumitelná
  - Příklad z jedné revize:
    v cele třídě mnoho nevhodných názvů proměnných – „bi“, „is“, „bos“, „asr“, „s“, „m“, „baos“, „dos“ a můj favorit, pole s názvem „array“
  - Kvíz: co ošetřuje následující podmínka?
    ```
    if(lv >= 0 && df >= 0 && this.hasRows() &&
        df < this.getRows().length) { … }
    ```
  - Všeho s mírou – obecně čím větší rozsah platnosti, tím důležitější je jasné jméno
  - Srozumitelnost != délka – např. význam proměnné „i“ je obecně známý
- Nepoužívejte jednu proměnnou ke dvěma účelům
- Ternární operátor je efektní, ale leckdy poněkud nečitelný
  - Kvíz: co vrátí následující výraz?
    ```
    (0 < 1) ? 2 : 3 + 4
    ```

PROFINIT

65

Užitečné čtení

- The Pragmatic Programmer. From Journeyman to Master (A. Hunt, D. Thomas, W. Cunningham)
- Clean Code (R. Martin)
- Pokud píšete v Javě: Effective Java (J. Bloch)

PROFINIT

66