Softwarové inženýrství

8. přednáška

Radek Janoštík

Univerzita Palackého v Olomouci

13. 11. 2025

Outline

- Kvalitní kódu – úvod
- Čitelnost kódu
- Techniky zvyšující kvalitu/udržitelnost
- Doporučená četba

Kvalitní kód– úvod

- Doteď jsme se bavili (většinou) o vnějším chování SW
- Pojďme nyní na nejnižší úroveň – do programovacího jazyka
- Kdy můžeme o kódu prohlásit, že je **kvalitní?**
  - Dělá to, co má dělat? (uživatelské a systémové požadavky?)
  - Prošel všemi testy?
  - Systém splňuje emergentní vlastnosti?
  - Naprogramovali jsme jej rychle? (levně)
- Asi tušíme, že by většinu z toho měl splňovat
  - Ale nejsou to postačující podmínky
- Kvalitu kódu určuje mnohem více aspektů

Kvalitní kód – motivace – LCM

- Ukázka kódu algoritmu LCM

Vnější vlastnosti SW

- Vnější ukazatele kvality – pohled uživatele:
- *Korektnost* – absence chyb (ve specifikaci, návrhu a implementaci) – „dělá, co má“
- *Použitelnost* – jak snadno se systém uživatelům používá, rychlost zaučení
- *Spolehlivost* – Jak často systém „nedělá, co má“. Průměrná doba mezi selháním
- *Integrita* – jak systém zabraňuje neautorizovanému přístupu k uloženým datům
  - Neporuší svá data (paralelizace, FS, čas)
- *Robustnost* – jak systém plní své funkce při výskytu chybných vstupů
  - Jak se systém chová při ztížených podmínkách
  - Zátěž HW, přetížení sítě, výpadek HW, ...

Vnitřní vlastnosti programu

- Vnitřní ukazatele kvality – pohled programátora:
- *Udržovatelnost* – jak snadno můžeme přidávat nové funkce, měnit funkcionalitu
  - Zvýšit výkon, opravovat chyby
- *Flexibilita* – jak snadno jde upravit systém pro jiné použití než původní
- *Přenositelnost* – snadnost použití systému v jiném prostředí než původní
- *Znovupoužitelnost* – kolik komponent půjde použít v jiných systémech
- *Čitelnost* – jak dobře se čte zdrojový kód, získá orientace, pochopí
- *Testovatelnost* – kolik námahy musíme vyvinout při tvorbě testů
  - Lze testovat všechny komponenty?
  - Lze (automaticky) testovat integrace?
  - Jak moc musíme testovat ručně?
- *Srozumitelnost* – pochopení systému – architektura, vzory, komponenty, úseky kódu

Vlastnosti programu

- Jakou mají mezi sebou vnější vlastnosti vazbu?
- Zaručují jedny druhé? Nebo jsou nezávislé?
- Může být efektivní, korektní, robustní software a přitom špatně udržovatelný a těžce testovatelný?
- 100% přímá souvislost není, ale mohou výrazně ovlivnit
- Např.: Špatná udržitelnost a srozumitelnost $\Rightarrow$ hůře se opravují nedostatky $\Rightarrow$ **menší pravděpodobnost** korektního a spolehlivého programu
- Málo flexibilní SW se hůře přizpůsobuje (bouřlivě) měnícím se požadavkům uživatelů $\Rightarrow$ může být hůře použitelný
- Ale i naopak: Splnění vnějších vlastností může působit komplikace programátorům:
  - SW, který se snadno používá „dá programátorům zabrat“

Čitelnost kódu

- Abstraktní tvrzení: *„Kód by nemělo být těžké pochopit“*
- Ale jak na to? Existuje nějaké `zaručené pravidlo dobré čitelnosti`?
- Čitelnost se těžce exaktně měří
- Poznání až po delší době čtení, námaze, únavě
- Co je podle vás čitelnější:

```
1 return exponent >=0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

  - Nebo

```
1 if (exponent >= 0) {
2   return mantissa * (1 << exponent);
3 } else {
4   return mantissa / (1 << -exponent);
5 }
```

Čitelnost kódu

- Někdy je rozhodnutí těžce subjektivní, až nemožné
- Existuje několik zásadních pravidel/doporučení
  - Jejich dodržováním *obecně* zvýšíme čitelnost kódu
- Nejdůležitější je ovšem **konzistence** napříč projektem
  - Stanovit si pevné konvence pojmenovávání
  - Odsazování (složené závorky, tabulátory)
  - Velká a malá písmena v názvech proměnných/slotů
- Většina (rozumných) jazyků má *Coding convetions*
  - Jednotná pravidla zvyšující čitelnost napříč programátory/projekty
- Projdeme (obecné) faktory ovlivňující čitelnost/srozumitelnost
  - Stanovíme nějaká doporučení
  - Dodržování není složité (sebekázeň), nese ale ovoce

Volba jazyka

- Asi nejzásadnější faktor pro čitelnost kódu
- Extrémní jazyky – binární kód, Whitespace, Brainfuck, Ook!, OstraJava diskutovat nebudeme
- Naštěstí volba (většinou) nebývá na programátorech
- Různé jazyky umožňují různý stupeň abstrakce a vyjadřování
- „Výmluvnost jazyka“

C	1
C++	2.5
Java	2.5
C#	2.5
Perl	6
Python	6
VisualBasic	4.5

Tabulka: Kolik řádků kódu jazyka C nahradí jeden řádek v daném jazyce

Formátování kódu

- Napsat program na jeden řádek není dobrý nápad
- Velmi často určeno konvencemi jazyka
- Velké projekty ale mají své konvence
  - Jádro Linuxu – `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`
  - Chromium – `https://chromium.googlesource.com/chromium/src/+/HEAD/styleguide/styleguide.md`
- Věčná diskuze tabulátor vs. x mezer (anketa)
- Jistě se shodneme, že určité odsazení je vhodné
  - Opět důraz na **konzistenci**
- Dnes velmi ovlivňuje vývojové prostředí, bereme jako automatické

Formátování kódu – příklady

```
1 template<class T>void DoSomething(T a[],int k,int l)
2 {for(int i=k;;){T x=a[(k+l)/2];int j=l;do{while(a[i]<x)++i;
3 while(x<a[j])--j;if(i>j)break;
4 T w=a[i];a[i]=a[j];a[j]=w;++i;--j;}while(i<=j);
5 if(k<j)DoSomething(a,k,j);if(i>=l)return;k=i;}
```

Formátování kódu – příklady

```
1  template<class T>
2  void Quicksort(T a[], int k, int l) {
3  for (int i=k;;) {
4    T x = a[(k + l)/2];
5    int j = l;
6    do {
7      while (a[i]<x) ++i;
8      while (x<a[j]) --j;
9      if (i>j) break;
10     T w = a[i];
11     a[i] = a[j];
12     a[j] = w;
13     ++i; --j;
14   } while (i<=j);
15   if (k<j) Quicksort(a,k,j);
16   if (i>=l) return;
17   k = i;
18 }
19 }
```

- Pomohlo čitelnosti?
- Pomohlo srozumitelnosti?

Vertikální uspořádání

- ⇒ Jak strukturovat kód „shora dolů“
- Oddělování deklarací a funkcí prázdným řádkem
  - Při procházení shora dolů se oči zastavují postupně na řádcích, které jsou po prázdném řádku
- Související části kódu bychom měli psát dostatečně hustě (zbytečně neodsazovat)
- Volá-li jedna metoda druhou, měly by být „blízko sebe“
  - Volající nad volanou (uspořádání nemusí existovat)
  - Při pročítání najdeme ihned (`ctrl+klik` neskáče „úplně mimo“)
- Rozdělení funkcionality do více souborů
- Metody, co se nevlezou na obrazovku (viz dále)

Vertikální uspořádání – příklady

- Kam vám „skáčou“ oči?

```
1   struct Uzel { int c; Uzel *nasl; };
2
3   Uzel *prvni=0;
4
5   void pridat(int c) {
6     Uzel *u = new Uzel;
7     u->c = c;
8     u->nasl = prvni;
9     prvni = u;
10  }
11
12  Uzel *hledat(Uzel *prvni, int x) {
13    for (Uzel *u=prvni; u; u=u->nasl)
14      if (u->c == x) return u;
15    return 0;
16  }
```

Vertikální uspořádání – příklady

- To samé bez odsazení

```
1   struct Uzel { int c; Uzel *nasl; };
2   Uzel *prvni=0;
3   void pridat(int c) {
4     Uzel *u = new Uzel;
5     u->c = c;
6     u->nasl = prvni;
7     prvni = u;
8   }
9   Uzel *hledat(Uzel *prvni, int x) {
10    for (Uzel *u=prvni; u; u=u->nasl)
11      if (u->c == x) return u;
12    return 0;
13  }
```

Horizontální uspořádání

- Jak moc dlouhé řádky budeme tolerovat
- Většina příkazů krátkých
- Ale poměrně populární „řetězení“ metod (*Method chaining, fluent interfaces*)
  - Například `LINQ`
- V projektech bývá stanoven limit na počet znaků na řádku
  - Dříve často $60 - 80$ (fyzické omezení terminálů)
  - S širokoúhlými monitory zvýšeno (≈ 120), vertikální čára v IDE
- Odsazení operátorů mezerami

```
1       Uzel *u = najit(prvni, 12);
```

  - Silný vztah $\Rightarrow$ bez mezery (pointer, závorky u volání)
  - Slabý vztah $\Rightarrow$ s mezerou (oddělení přiřazení)

Horizontální uspořádání – ukázka

- Pouhé zarovnání argumentů metod může zvýšit čitelnost

```
ulozitAdresu("Thamova 15",      "Praha 8",    18600);
ulozitAdresu("Koliste 67a",     "Brno",       60200);
ulozitAdresu("Mlynske Nivy 58", "Bratislava", 82105);
```

Pojmenování konstruktů

- Jméno proměnných, metod, tříd by mělo být nositelem informace
  - Krátké, výstižné, po přečtení ihned jasný smysl
  - Volba velmi složitá, ale přemýšlet se vyplatí
- Je potřeba zvolit specifické jméno, které plně věc pojmenuje (nic obecného)
  - `public string GetPage(string url){...}`
  - `public string FetchPage(string url){...}`
  - `public string DownloadPage(string url){...}`
- Co podle vás vrací metoda `public int GetSize()` u třídy `BinaryTree`?
- Nebojte se použít květnaté názvy (pokud plně vystihují)
  - př. v jazyce PHP funkce `explode()`, která má řetězec jako parametr
  - Co asi dělá?
- Zavedená jména pro „iterátory“ – `i, j, k, it, iter`
  - Všichni jsou na ně zvyklí, dávají si pozor, že „určují tok programu“
  - Neměli bychom je používat jinak

Pojmenování konstruktů

- Pracujeme-li s jednotkami (čas, délka, místo v paměti) uveďme jednotku do názvu proměnné:

```
1       public void Delay(int delay);
```

  - vs.

```
1       public void Delay(int delaySec);
```

```
1       long odpracovanoClovekohodin = project.SumOfManHour();
2       long prevezenoTunokilometru =
            Railway.GetInstance().CargoReport().TonsKilometers;
```

- Přidání (důležité) vlastnosti do jména proměnné

```
1       string unescapedComment = getComment();
2       string pageUtf8 = HttpClient.FetchPage();
```

- Pozor na délku jména, používání (neustálených) zkratek
- Magické konstanty

Pojmenování – nejasnosti

- Vždy bychom měli mít na paměti: `Nemůže si ten název někdo vyložit úplně jinak`?
- `results = Database.Students.Filter("year <= 2")`
  - Co dělá? Vybere s ročníkem $<=$ 2 nebo naopak vyřadí?
  - Lepší jména? `Select, Exclude, Where, ...`
- Rozsahy. Pamatujete si, zda jsou rozsahy „včetně“ nebo „menší rovno“?
  - `var subArray = students[1..5];`
  - Vhodným pojmenováním lze předejít zmatkům.
  - `start, stop` – je jasné?
  - `first, last` pro „včetně“
  - `begin, end` pro první včetně, poslední nezahrnut
- Prefix `get` – lidé očekávají „hloupý getter“
  - Snažit se dodržet
  - Nepoužívat u metod se složitým výpočtem

Rozdělení složitých výrazů

- Lidé dokáží „udržet v hlavě“ pouze omezené množství věcí.
  - $\Rightarrow$ Příliš složité výrazy se nám budou špatně číst
  - Hůře je pochopíme
- Rozdělit si problém na podproblémy
  - Často odhalíme obecnější konstrukty, abstrakce, znovupoužitelnost
  - S vhodným pojmenováním částí bude lépe čitelné
- Využití De Morganových zákonů v podmínkách:

```
1       if (!(fileExists && !isProtected)) {
2         Error("You can't read this file.");
3       }
```

  - Jak zjednodušíme?

```
1       if (!fileExists || isProtected) {}
```

- Pravidlo jedné obrazovky / limit řádků?
- „Return if“

Komentáře

- Komentáře jsou dvousečná zbraň. Mohou být velmi přínosné, ale také škodit
- Dobře napsaný kód by neměl potřebovat mnoho komentářů
  - Často kompenzace špatného kódu
  - Pozor na aktuálnost (změny kódu, ponechání komentáře)
- Když píšeme dlouhý komentář, měli bychom zamyslet: `Neměli bychom upravit kód?`
- Velmi vhodné pro vysvětlení optimalizačních hacků
- Dokumentování na příkladu
- Napsání důvodu, proč jsme zvolili dané řešení
- Dočasné komentáře: (spolupráce s IDE)
  - `//@TODO -- dodelat kontrolu data z klienta`
  - `//@FIXME -- toto potrebuje upravit`
  - `//Algoritmus je pomaly, for testing only`

Komentáře – vhodné

```
1 // Author: Radek Janostik - radek.janostik@upol.cz (2020)
2 // MIT License
3
4 // Parsing format: hh:mm:ss dd.mm.rrrr
5 DateTime ParseDate(string s) {}
6
7 // Pri teto hodnote vetsinou neni zretelne zhorseni
8 // kvality obrazu. Pokud ano, je treba hodnotu zvysit.
9 kvalitaObrazu = 0.75;
10
11 // Pouzita heuristicka metoda nenajde nektera reseni. Metoda,
12 // ktera najde vsechna reseni, by byla casove velmi narocna.
13 ...
```

Komentáře – nevhodné

- Zbytečné komentáře:

```
1 class A {
2   A() { } // konstruktor
3 }
```

- Komentování, který blok uzavíráme

```
1 try {
2   while (...) {
3     ...
4   } // while
5 } // try
6 catch (...) {
7   ...
8 } // catch
```

Komentáře – nevhodné

- Vysvětlující nevhodný název metod:

```
1 // Funkce zrusi obsah tabulky,
2 // samotna tabulka ale zustane zachovana
3 zrusitTabulku()
```

  - ⇒ Změnit název, komentář smazat
- Zakomentování starého, nepotřebného kódu

```
1 // souradnice.x *= 2;
2 // souradnice.y *= 2;
```

  - Snižuje přehlednost
  - „Budoucí generace“ si budou myslet, že je důležitý a neodstraní jej

Proměnné

- Rozumnou prací s proměnnými můžeme zvýšit jak čitelnost, tak udržitelnost programu
- Rozsah platnosti = část programu, ve které je proměnná deklarována a můžeme ji použít
- Oblast života = část programu od místa přiřazení hodnoty po poslední použití přiřazené hodnoty
- Každá deklarovaná proměnná by měla být použita
- Délka názvu by neměla být příliš krátká ani dlouhá
- Výzkumy: $10 - 16$ znaků
- Přímo úměrná rozsahu platnosti
- Oblast života bychom se měli snažit udržovat co nejkratší
  - Nemusíme proměnnou dlouho „držet v hlavě“ – při ladění

Proměnné – oblast života

```
1  void SouhrnDat(...) {
2  ...
3  nactiStaraData(staraData, &pocetStarychDat);
4  nactiNovaData(novaData, &pocetNovychDat);
5  soucetStarychData = Soucet(staraData, pocetStarychData);
6  soucetNovychDat = Soucet(novaData, pocetNovychDat);
7  tiskniStaraData(staraData, soucetStarychDat, pocetStarychDat);
8  tiskniNovaData(novaData, soucetNovychDat, pocetNovychDat);
9  ulozStaraData(soucetStarychDat, pocetStarychDat);
10 ulozNovaData(soucetNovychDat, pocetNovychDat);
11 ...
12 }
```

Proměnné – oblast života – lépe

```
1  void SouhrnDat(...) {
2  ...
3  nactiStaraData(staraData, &pocetStarychDat);
4  celkemStarychData = Soucet(staraData, pocetStarychData);
5  tiskniStaraData(staraData, soucetStarychDat, pocetStarychDat);
6  ulozStaraData(soucetStarychDat, pocetStarychDat);
7  ...
8  nactiNovaData(novaData, &pocetNovychDat);
9  soucetNovychDat = Soucet(novaData, pocetNovychDat);
10 tiskniNovaData(novaData, soucetNovychDat, pocetNovychDat);
11 ulozNovaData(soucetNovychDat, pocetNovychDat);
12 ...
13 }
```

Proměnné

- Každá proměnná by měla sloužit pro jeden účel

```
1 // Vypocet korenu kvadraticke rovnice.
2 // (Predpokladame, ze diskriminant b*b-4*a*c neni zaporny.)
3 double prac = sqrt(b*b - 4*a*c);
4 koren1 = (-b + prac) / (2*a);
5 koren2 = (-b - prac) / (2*a);
6 ...
7 // rozdil mezi koreny
8 prac = abs(koren1 - koren2);
```

- vs.

```
1 // Vypocet korenu kvadraticke rovnice.
2 double sqrtDiskriminant = sqrt(b*b - 4*a*c);
3 koren1 = (-b + sqrtDiskriminant) / (2*a);
4 koren2 = (-b - sqrtDiskriminant) / (2*a);
5 ...
6 double rozdil = abs(koren1 - koren2);
```

Techniky zvyšující kvalitu – Úvod

- V průběhu let vznikly různé metody, doporučení, jak správně psát kód
- Jejich dodržování by mělo částečně zaručit kvalitu kódu
- (snad) by mohlo stačit zopakovat principy z Paradigmat programování
- Dodržením těch nejzákladnějších principů máme polovinu úspěchu
- Velmi důležitá sebekázeň. Zastavit se, popřemýšlet
  - Přirozená lenost je někdy dobrá vlastnost
  - Neměla by ale bránit „udělat věci pořádně“
  - Ale: `Protože jsem líný, udělám to pořádně a pak budu mít pokoj!`
- Časté „zábavné“ pojmenování principů, snadnější zapamatování

Defenzivní programování

- = Ochrana funkce před špatnými daty
- Preventivní opatření – změna kódu, špatné vstupy od uživatele. Základní principy:

1. Kontrola všech hodnot z externích zdrojů (UI, soubor, síť)
   - Ověřovat rozsah hodnot
   - Jsou tam data, která tam mají být
2. Kontrola vstupních hodnot všech parametrů funkcí
   - „všichni lžou“ – preventivní nedůvěra vlastnímu kódu
3. Rozhodnutí, co má program dělat při špatných datech

- Pomáhá zvyšovat kvalitu systému, bohužel snižuje čitelnost
- Nenahrazuje testování

Defenzivní programování – aserce

- Aserce (assertion) je kód (makro, funkce), kterým se program kontroluje za běhu.
  - Pravdivá == program běží jak má
  - Nepravdivá == neočekávaná chyba

```
1 assert(delitel != 0);
2 assert(pocet <= MAXPOCET);
```

- Ověřování:
  - Hodnoty rozsahu parametrů
  - Kontrola otevření streamu/souboru. Módu otevření.
  - Kontrola nezměnění proměnných po volání funkce
  - Nenulový ukazatel, index v rozsahu pole
  - Pole bylo inicializováno, má daný počet hodnot

Defenzivní programování – zásady aserce

- Aserce většinou součástí pouze vývojového kódu
  - Systém maker „IF DEBUG“
  - Některé důležité nechat i na produkci (co uživatel bude tolerovat)
- Kontrolovat podmínky, které by vždy měly nastat.
- Kontrolovat podmínky, které by nikdy neměly nastat.
  - Nepoužívat pro ošetření chyb, kterou mohou nastat
- V asercích by se neměl volat kód (kompilátory mohou odstraňovat

```
1 assert(foo());
```

  - Ale raději:

```
1 bool overeni=foo();
2 assert(overeni);
```

Ortogonální systémy – Míra spřažení komponent

- Moduly softwaru jsou na sobě nezávislé
- Ideál: Změnou jednoho modulu nebude vyžadovat změnu jiné
  - Většinou nedosažitelný ideál
  - Ale má smysl k němu směřovat
- Zpřažení komponent – jak moc jsou na sobě jednotlivé komponenty závislé?
  - Můžeme jednu „beztrestně“ vyměnit za druhou?
  - Těžce exaktně měřitelné
- Úzce zpřažené – hůře se kód mění, musíme upravovat více komponent
  - Větší pravděpodobnost nechtěných chyb
  - Špatně se paralelizuje vývoj
- Volně zpřažené – nezávislé zavádění změň, snadnější paralelizace práce
  - Snadnější testování (náhrada falešnými instancemi, . . .)

Ortogonální systémy – typy zpřažení

- *Zpřažení obsahu* – často uváděno jako „patologické zpřažení“
  - Moduly přímo odkazují interní informace z jiných modulů
  - Čtou/mění jejich interní stav
  - Přímo v rozporu s modularitou
- *Globální zpřažení* – sdílení globálních dat mezi moduly
  - Často se nevyhneme (konfigurace, konstanty)
  - Proměnné mohou být nebezpečné
- *Zpřažení řízení* – střední typ. Jeden řídí druhý předáváním řídících proměnných
  - Nastavení nějakého *flagu* jinému modulu, ten se podle něj zařídí
- *Zpřažení daty* – moduly spolu komunikují pomocí metod
  - Ale ovlivňují si parametry/návratovými hodnotami
- *Zpřažení metodami* – moduly volají metody jiných, bez argumentů

Minimalizování komplexnosti

- Komplexnost software s sebou přináší problémy:
  - Opožděné dodání, předražení
  - SW se nechová, jak by měl. Dostane se do nezamýšleného stavu.
  - Snadněji se přehlédnou bezpečnostní problémy
  - Hůře se měří některé vlastnosti, testuje
- *Podstatné problémy* – jádro problému, který se snažíme řešit
  - Nevyhneme se mu, tato komplexita je opodstatnitelná
- *Vedlejší problémy* – odvozeny ze samotné tvorby SW
  - Naše zvolené konstrukty, postupy, design
  - Musí řešit SW inženýři, měli bychom redukovat a snižovat komplexnost
- Oba typy komplexnosti bychom se měli snažit minimalizovat
- Máme k dispozici několik technik, metod, které nám tím pomůžou

KISS – Keep It Simple, Stupid

- Idea: Systémy obecně fungují dobře, pokud si udrží jednoduchost
- Několik variant: `Keep It Short, Simple;` `Keep It Simple, Straightforward;` `Keep It Simple, Silly`
- Původ (překvapivě) z leteckého průmyslu: „Navrhněte tryskáč tak, aby jej byl člověk schopen na koleni opravit kladivem a hasákem“
- Software/modul/komponenta by měl dělat jednu věc a pořádně
- Kočkopes není vhodnou cestou
- Často uplatňován v „GNU světě“ – řetězeních jednoduchých příkazů
  - `cat file.txt | sort | uniq > list.txt`
- Zase to s jednoduchostí nepřehnat

DRY – Don’t Repeat Yourself

- Snaha nemít redundantní kód = stejné/podobné části kódu
- Redundantní kód = špatně se udržuje (zapomeneme upravit na všech místech)
- Při porušení – `WET (Write Everything Twice, Waste Everyone’s Time, We Enjoy Typing)`
- Často vzniká kopírováním kódu – `Copy & Paste`
  - Často bychom měli být velmi opatrní – opravdu musíme kopírovat?
  - Jsme líní abstrahovat?
  - Důkladně jsme pročetli/změnili všechny části vloženého kódu?
- Magické řetězce, konstanty – často komunikace s API
  - Napsat si metody zajišťující funkcionality
  - Konstanty na jednom místě
  - Využití výčtových datových typů

Skrývání informací (Information hiding)

- Zavedený princip v OOP
- Moduly by neměly prozrazovat svou interní reprezentaci
- Čím méně toho u interní reprezentaci víme, tím méně zpřažené moduly budeme mít
- Oddělení návrhu a implementace
- Uvědomovat si v které *roli* jako programátor jsme
  - Implementujeme něco, co bude někdo používat?
  - Používáme nějaký modul
  - Striktně odlišovat, i když jsme na obou frontách
- Pomůže návrhový vzor *Fasáda*

YAGNI – You Aren't Gonna Need It

- Princip z `extrémního programování` – snaha odstraňovat zbytečnou logiku a funkcionalitu
- Měli bychom implementovat funkce, které opravdu potřebujeme
  - Nenaddělávat zbytečné věci ála „Bude se mi to v budoucnu hodit“
  - Snaha vyhnout se *over-engineeringu*
- Nezatěžujte se budoucností, řešme současné problémy
- Musíme ale počítat z budoucím refaktorováním
- Nemělo by se uplatňovat u architektury, kódu, který ulehčí údržbu

Oddělení odpovědnosti – Separation of Concerns

- = návrhový princip, který snižuje komplexnost rozdělením systému na menší části
- Dané subsystémy mají pevně stanovou oblast, za kterou jsou zodpovědné
- Z pevného oddělení ihned víme, které části máme upravit
- Může být aplikováno na více úrovních aplikace
  - Architektura – Rozdělení logiky, persistence, prezentace, infrastruktury (MVC, MVVM, Client-server, . . .)
  - Třídy – Rozdělení odpovědnosti v modulech do ucelených třídy. Každá má svou roli/funkci
  - Metody – Jasně specifikované, které metody dělají kterou část dané zodpovědnosti
- Př. HTML, CSS, Javascript – každý dělá „to svoje“, navzájem se doplňují

Principy SOLID

- Sada principů, které se zaměřují na srozumitelný, dobře udržovatelný, znovupoužitelný kód.
- Pět základních principů, které by SW architekti měli znát:

1. **S**ingle Responsibility Principle (SRP) – každá třída by měla dělat jednu věc, a to pořádně
   - Více zodpovědnosti, použita na více místech, větší zpřaženost
   - Při potřebě změny, změníme pouze jednu třídu
2. **O**pen/Closed Principle (OCP) – komponenty otevřeny k rozšíření, uzavřené k modifikaci
   - Snadnost přidávání nové funkcionality bez modifikace staré
   - OOP – dědičnost
   - Programování „proti“ rozhraní

Principy SOLID

3. **L**iskov Substitution Principle (LSP) – princip v OOP
   - Podtřídy by neměly porušovat pravidlo `Is a`
   - Metody Nadtřídy by měly korektně fungovat se všemi podtřídami
   - `Rectangle x = new Square(4);` by mělo fungovat dále korektně
4. **I**nterface Segregation Principle (ISP)
   - Rozhraní definují „co se může dělat“, klienti používají bez implementační znalosti
   - Rozhraní bychom měli rozdělovat na co nejmenší, ale ucelené
   - Aby klienti záviseli jen na podmnožině, kterou opravdu využijí
   - Př. hierarchie rozhraní u kolekcí v Java/C#
5. **D**ependency Inversion Principle (DIP) – jak navrhovat volně zpřažené komponenty
   - „Vysokoúrovňové“ moduly by neměly záviset (přímo) na těch nízkoúrovňových
   - Měly by záviset pouze na abstrakci/rozhraních
   - (obrázek na tabuli)
   - Dependency Injection

Doporučená literatura

- Steve McConnel. Dokonalý kód. Computer Press, 2005. ISBN: 80-251-0849-X
- Dustin Boswell, Trevor Foucher. The Art of Readable Code, Simple and Practical Techniques for Writing Better Code. O'Reilly, 2011. ISBN: 978-0-596-80229-5.
- Joseph Ingeno. Software Architect's Handbook. Packt, 2018. ISBN: 978-1-78862-406-0
- T. Uno, M. Kiyomi, and H. Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In FIMI, volume 126, 2004.
- `https://hovnokod.cz/`