Softwarový projekt a jeho životní cyklus

Základy algoritmizace – 1. cvičení

Zuzana Petříčková
prezentace z velké části převzata od Tomáše Oberhubera
http://geraldine.fjfi.cvut.cz/~oberhuber/data/vyuka/zalg/01-uvod.pdf

15. února 2020

Životní cyklus softwarového projektu

Klasické „vodopádové" schéma:

1. Motivace, definice problému
2. Analýza požadavků
3. Návrh
4. Implementace
5. Integrace a testování
6. Údržba

Motivace pro započetí projektu

Základem úspěchu je jasně a stručně definovaný cíl:

1. chci vydělat peníze
   - typicky zadání od zákazníka
   - mnoho projektů velkých zavedených firem (Microsoft,...)
2. vytipuji si nějaký (zajímavý, praktický) problém a ten chci vyřešit
   - př. Google, GNU,...
3. chci něco řešit jen tak pro zábavu
   - př. Linux, Facebook...

Motivace pro započetí projektu

Jasně a stručně definovaný cíl:

1. Google: Organizovat všechny známé informace
2. Apple: Vyrábět jednoduché počítače pro každého
3. IBM/Mainframe: Vyrábět maximálně spolehlivé systémy
4. GNU: Vytvořit volně šiřitelnou alternativu systému UNIX
5. nějaký zákazník: „Máme problém s evidencí materiálu na skladu."
6. ...

cíl by neměl naznačovat způsob řešení

Rešerše

Máme jasně a stručně definovaný cíl → provedeme rešerši:

- jaká je konkurence na trhu?
- neřešil už podobný problém někdo jiný?
- dokážeme přijít s **výrazně** lepším řešením?

pokud je odpověď na třetí otázku NE, je lepší projekt zrušit (a příp. odkázat zákazníka na konkurenci)

Předběžné požadavky (specifikace)

Sepíšeme formální dokument (tzv. **hlavní dokument**) obsahující:

- **hlavní cíl projektu**
  př. ,,Zorganizovat všechny informace"
- **dílčí cíle projektu**
  př. ,,Implementovat indexovací algoritmus PageRank"
  + rozmyslíme si, co program dělat **nebude**
- **priority projektu** (př. maximální výkon, spolehlivost, flexibilita, bezpečnost...)

pokud jsou dílčí cíle popsány dopodrobna, říká se hlavnímu dokumentu **specifikace**

Alternativně: agilní programování

místo specifikace rychle sestavíme **prototyp** aplikace nebo tzv. **mock-up**

- maketa, která není (plně) funkční, ale ukazuje, jak bude výsledný produkt fungovat
- použijeme nástroje vyšší úrovně (Python, Java, ne C/C++)

Analýza požadavků

Cílem specifikace, prototypů nebo mock-upů je určit **funkcionalitu** produktu s ohledem na stanovené priority.

- méně znamená více
  - není umění sepsat všechny funkce, které nás napadnou
    $\rightarrow$ je třeba vybrat jen ty podstatné
  - design je dokonalý, když už z něj nelze nic odebrat
- základem úspěchu jsou jednoduché funkce a jejich kvalitní implementace
  - pokud nesvedu danou funkcionalitu implementovat téměř perfektně, raději ji nepřidávám

Příklady

- jednoduché, malé a jednoúčelové programy vs. distribuce se spoustou zbytečných a ne úplně funkčních programů

Prezentace projektu před zahájením

Cílem je získat zdroje pro vypracování projektu:

- předložení návrhu **smlouvy** zákazníkovi:
  - hlavní dokument (specifikace)
  - data provedení dílčích cílů
  - odhad ceny za celý projekt
- zveřejnění hlavního dokumentu na webu (open source nebo vědecké projekty)
- vypracování žádosti o grant
- prezentace manažerům firmy

Prezentace musí jasně ukázat priority projektu, aby je pochopil i laik

Návrh architektury

Přemýšlíme nad implementací:

- typ zařízení, operační systém
- programovací jazyk, vývojové prostředí a nástroje
- dostupné knihovny
- hlavní datové struktury a použité algoritmy
- uživatelské rozhraní, zpracování chyb
- výkonnost, paměťové požadavky, správa paměti

U každého bodu:

- zjistíme všechny možnosti/alternativy
- řádně vysvětlíme naši volbu
  - volíme spíše osvědčené produkty než absolutní novinky (riziko neúspěchu)
- vše zdokumentujeme

Návrh architektury

1. Systém rozdělíme na menší moduly (podprogramy) a popíšeme vztahy mezi nimi
   - např. stanovíme hlavní třídy objektů a jejich vzájemné vztahy (dědičné hierarchie)
2. Určíme vnitřní strukturu modulů
   - např. použité datové struktury a algoritmy

Správně navržený modul:

- dokážeme stručně a jasně popsat, co má dělat
- má jen několik vstupních parametrů
- funguje nezávisle na ostatních modulech (tzv. ortogonalita)
  - vyhýbáme se globálním datům

Návrh architektury

U všech projektů, na kterých pracuje více programátorů, je třeba se předem domluvit na ,,štábní kultuře”:

- konvence pro názvy funkcí, proměnných, objektů
- konvence pro komentáře (! jazyk dokumentace) a indentaci (odsazování)
- jak bude program přistupovat k chybám
- jak bude program zacházet s textovými řetězci (lokalizace?)

Návrh architektury

Zpracování chyb

- zejména chyby vstupních a výstupních operací
- v běžných programech se zpracováním chyb zabývá až 90% kódu
- Rozmyslíme si jednotný přístup:
  - aktivní / pasivní přístup
  - konvence pro ohlašování chyb
  - míra tolerance k chybám
  - úroveň, na které budou detekovány a ošetřovány chyby
  - zodpovědnost jednotlivých modulů za správnost jeho vstupních dat
  - robustnost programu (schopnost pokračovat poté, co došlo k chybě)

Implementace

1. Vytvoříme co nejjednodušší a zároveň bezchybnou implementaci, která vyhovuje zadání
   - vyhýbáme se předčasné optimalizaci
2. Pak teprve případně optimalizujeme
   - jen pokud zjistíme, že aktuální implementace není dostatečně výkonná
   - původní implementaci nezahazujeme!, ale použijeme ke srovnání (efektivity, výsledků)

Čistota kódu

Čistý kód

- je dobře čitelný a srozumitelný
- snadno se doplňuje a upravuje

Jak na to?

- Kód odsazujeme a rozdělujeme ho do krátkých celků (např. funkcí)
- Zbytečně nepíšeme (skoro) ten samý kód dvakrát
- Vyhýbáme se ,,chytrým" fintám
- Kde to jde, používáme návrhové vzory

Refaktorizace = pročišťování kódu

Dokumentace

Proč psát dokumentaci?

- aby bylo jasné, co má program dělat
- abychom byli schopni program změnit (cizí, náš za pár let)
- abychom mohli spolupracovat

Dokumentace

- komentáře v kódu
- doprovodný dokument (technická dokumentace)
- uživatelská dokumentace
  - referenční příručka
  - interaktivní nápověda
  - sada vzorových úloh, příklady
  - tutoriál

Komentáře v programu

- Ideální kód je srozumitelný sám o sobě a komentáře nepotřebuje:
  - entitám v programu dáváme samovysvětlující názvy
  - pokud je v programu ,,ošklivé místo" vyžadující komentář, zamyslíme se nad tím, jak ho reimplementovat čistěji
- dokumentace **nesmí** být zastaralá oproti kódu
- méně je více
  - opravovat hromady komentářů při každé změně kódu je otrava

Komentáře v programu

Kam psát komentáře?

- je dobré komentovat jen větší celky:
  - moduly, podprogramy, definice struktur a tříd
  - hlavičky funkcí a metod
  - deklarace glob. proměnných
- někdy bývá zvykem uvést ,,na začátku" každého souboru identifikaci programu
  - nazev programu, autor, roku vzniku programu, účel (,,Piškvorky, Petr Pavel, 1. ročník, obor ASI, letní semestr 2019/20, Zápočtový program pro předmět ZALG")
- příp. vysvětlit použité algoritmy, ,,ošklivá místa"

Komentáře v programu ... příklad

Kam psát komentáře?

- Komentáře by měly obsahovat informaci **co a proč se v daném úseku programu dělá** a nikoli jak se to dělá

```
pocetAut += 1; // koupe dalsiho auta          ... :)
x += 1; // pricteni jednicky k promenne x     ... :(
```

Odstrašující příklad ... zde se bez komentáře neobejdeme:

```
void zpracuj(Pole& a, Pole &b, int n)
{
    ...
}
```

Komentáře v programu ... příklad

Odstrašující příklad ... komentář, který nám moc nepomůže:

```
/* Otoceni zacatku pole. */
void zpracuj(Pole& a, Pole &b, int n)
{
    ...
}
```

Komentáře v programu ... příklad

Odstrašující příklad ... o něco lepší komentář

```
/* Do zacatku pole B az po index (N-1) se zkopiruje
   otoceny zacatek pole A delky N.
   Hodnoty zbytku pole B se nemeni.
*/
void zpracuj(Pole& a, Pole &b, int n)
{
    ...
}
```

zároveň by bylo lepší se zamyslet nad výstižnějším názvem funkce a polí

Komentáře v programu

Dokumentace modulu/funkce

- k čemu slouží
- vstupy / parametry funkce
- za jakých podmínek může být modul spuštěn / volání funkce
- výstup modulu / co funkce vrací

Nepřehledná dokumentace bývá známkou špatně navrženého modulu / funkce

Technická dokumentace

Dokument, který shrnuje návrh architektury projektu. Často obsahuje:

- specifikaci projektu (anotaci, přesné zadání)
- návrh architektury
  - zvolený programovací jazyk a prostředí, použité knihovny a nástroje
  - použité algoritmy
  - struktura programu a podprogramů a jejich propojení
- **Případně:**
  - Návrh, jak by se měl projekt modifikovat při té které předpokládané změně zadání
  - Diskuse výběru algoritmů, o kterých řešeních jste uvažovali, a případně i proč jsme je zavrhli
  - Reprezentace vstupních dat a jejich příprava
  - Reprezentace výstupních dat a jejich interpretace
  - Průběh prací, Co nebylo doděláno, Závěrečný povzdech

Testování a ladění kódu

- je nezbytné, přitom časově náročné
  (80% kódu je implementování za 20% času)
  - jak kód přibývá, trávíme čím dál více času odstraňováním chyb
  - na odstranění chyby zabere nejvíc času její lokalizace
- základem úspěchu jsou tzv. **unit testy**
  $\rightarrow$ mohou výrazně urychlit vývoj kódu a zvýšit jeho kvalitu

Testování a ladění kódu

Unit testy

- kdykoliv dokončíme (nebo než začneme psát) jeden kus kódu (např. funkci), napíšeme kód, který testuje, zda funkce pro dané vstupní parametry vrací správné výsledky
- spuštění testů by mělo být jednoduché / automatizované
- **Výhody:**
  - po každé úpravě kódu vidím, zda se neporušila funkcionalita
  - unit testy mohou sloužit jako ukázka použití funkce
- **Nevýhody:**
  - prakticky žádnou funkci nelze otestovat kompletně v rozumném čase
  - ne vždy je jasné, které vstupní parametry jsou klíčové, nebo jaký má být výsledek funkce

Uživatelská dokumentace

- musí být stručná a jasná (uživatel nechce ztrácet čas)
- měla by obsahovat informace o tom, jak s programem pracovat

Uživatelská dokumentace, která uživatele nepopudí:

- najdeme typické a **jednoduché** úlohy a sepíšeme tutoriál, jak tyto úlohy řešit
- přidáme podrobnější referenční příručku s detaily (např. jaký je požadovaný formát vstupních dat)
- případně sestavíme FAQ

Uvolnění první verze (production run)

- do první verze implementujeme jen nezbytné minimum, ale snažíme se o dobrou kvalitu
- ne zcela kvalitní produkt uživatele odradí
  $\rightarrow$ ne zcela doladěné funkce je lepší v první verzi vypnout a vydat až s updaty
- můžeme uvolnit beta verzi a využít k testování dobrovolníky

Údržba

- odstraňování chyb a pročišťování kódu
- doplňování funkcí
  - na přání uživatelů
  - nepřeháníme to
  - dbáme na to, aby se náš produkt dobře a snadno používal a novými funkcemi se nepokazil
- někdy se můžeme rozhodnout přepracovat celý projekt zcela od začátku