[Draft pre ucely spatnej vazby] Softvérové inžinierstvo v otázkach a odpovediach

Mária Bieliková Jakub Šimko Marián Šimko

Február 2016

Obsah

Úvodné slovo

Kniha, ktorú držíte v rukách je učebnicou pre softvérové inžinierstvo. Je vhodná pre každého, kto študuje základy tejto oblasti.

Otázky a odpovede

Kniha nemá výkladový charakter, ale je napísaná formou otázok a odpovedí. Odpovede sú sprevádzané komentármi a snažia sa podrobne zachytávať myšlienkové pochody, aké ku konečnému riešeniu otázok vedú.

Tento formát sme zvolili z viacerých dôvodov. Predovšetkým sme chceli vytvoriť alternatívu pre študentov, ktorým výkladové materiály nevyhovujú vzhľadom na štýl učenia, majú radšej mentálne cvičenia a sledujú priebeh riešených úloh. Kniha zároveň môže slúžiť aj ako cvičebnica pre študentov, ktorí sa o softvérovom inžinierstve učia z iných zdrojov. V obidvoch prípadoch považujeme za najefektívnejšie, ak túto knihu zároveň využijú prednášajúci na predmetoch softvérového inžinierstva.

Ďalšou motiváciou bolo znovupoužiť existujúci materiál: Otázky, ktoré nájdete v tejto knihe vznikali v priebehu ostatných piatich rokov našej pedagogickej praxe na predmetoch zaoberajúcich sa softvérovým inžinierstvom. Pôvodne sme ich vytvárali ako materiál pre testy a skúšanie. Išlo najmä o jednoduchšie otázky o fundamentálnych veciach, základoch softvérového inžinierstva, v knihe však nájdete aj otázky zložitejšie, odpovede na ktoré presahujú rozsah jednej strany.

Postupom času narástol počet otázok až do tej miery, že kompletne pokryl sylaby predmetov v ktorých sme ich používali. Boli sme radi, pretože otázok bol dostatok na to, aby sme z nich mohli vyberať do testov a skúšok bez toho, aby sa pričasto opakovali. Používanie týchto otázok si vyžadovalo, že sme často rozprávali aj o odpovediach na ne. Časom sme si potrebu, mať k otázkam ucelené a podrobné odpovede uvedomili a aj preto vznikla táto kniha.

Na túto knihu by sa čitateľ určite nemal pozerať ako na prvú a poslednú zastávku v učení sa o softvérovom inžinierstve. Silno odporúčame primárne čítať alebo aspoň konzultovať výkladovú literatúru resp. navštevovať prednášky zodpovedajúceho predmetu.

Rozptyľovanie nedorozumení

Využívanie týchto otázok na testoch a skúškach zviditeľnilo jeden fenomén. Ak študenti robia pri odpovediach na otázky nejaké chyby, tieto chyby sa často opakujú, akoby boli v nejakom zmysle systematické. Podobné správanie pozorujeme aj pri diskusiách so študentmi a práci na praktických úlohách. Uvedomili sme si, že nie všetko sa dá ospravedlniť nepozornosťou alebo nedôslednou prípravou študentov pred testami a cvičeniami, ale že niekde na ceste medzi nami (učiteľmi), látkou ktorú učíme a študentami vznikajú nedorozumenia.

Uvedomili sme si zároveň, aké zhubné tieto nedorozumenia sú v predmetoch o softvérovom inžinierstve. Ide totiž o predmety, ktoré vzhľadom na predchádzajúce skúsenosti študentov akoby viseli vo vákuu. Na ich začiatku je napríklad potrebné študentom vysvetliť, prečo sa vôbec nejakou analýzou a návrhom softvéru zaoberať či prečo testovať softvér. To je zásadná komplikácia navyše napríklad oproti výučbe programovania, ktorého „produkty“ sú oveľa hmatateľnejšie a dôvody prečo sa ho učiť sa javia študentom silnejšie. Študenti majú s analýzou, návrhom, testovaním či modelovaním softvéru len malé alebo žiadne predchádzajúce skúsenosti. To si uvedomujeme a aj preto v softvérov-inžinierskych predmetoch doslova „začíname od Adama“. No po celý čas, čo princípy softvérového inžinierstva pred študentmi odvíjame, naše vysvetlenia stoja na hlinených nohách nášho vlastného výkladu, nepodporené skúsenosťami poslucháčov. Keď sa potom do takejto situácie dostanú nedorozumenia, u študentov ľahko vzniká nepochopenie, nezáujem až odpor voči týmto predmetom.

Časté testovanie študentov otázkami (prakticky každý týždeň výučby na predmetoch) nám dalo do rúk nástroj ako s nedorozumeniami bojovať. Ak v niektorom týždni dopadla niektorá otázka „zle“ s opakujúcou sa častou chybou, bola to pre nás príležitosť nedorozumenie rozbiť. Touto knihou však chceme zájsť ešte ďalej. V odpovediach na otázky sa na časté nedorozumenia vyslovene zameriavame, explicitne na ne poukazujeme a uvádzame veci na pravú mieru.

Ako čítať túto knihu

Čítať túto knihu môžete akýmkoľvek spôsobom, my však predpokladáme dva typické scenáre:

1. Čítanie ako príprava na iné študijné aktivity (v rámci predmetov o softvérovom inžinierstve), kedy sa predpokladajú len malé predchádzajúce znalosti o predmetnej študijnej látke.
2. Čítanie ako forma samoskúšania (po absolvovaní iných študijných aktivít zameraných na rovnakú látku).

Znenie a otázok, ich poradie a najmä spôsob akým sú formulované odpovede sledujú predovšetkým prvý scenár. Snažili sme sa, aby znenia odpovedí neobsahovali len minimalisticky opísané riešenia otázok, ale aby ich primerane komentovali a naznačili, ako by sa úvahy k nim vedúce mohli (mali) v hlavách softvérových profesionálov

vyvíjať. Čitateľov, ktorí hľadajú len rýchle odpovede na otázky zrejme sklameme: odpovede sú nerozlučne späté z ich vysvetleniami a bol to náš zámer. Našim cieľom totiž nie je, aby sa študenti odpovede na otázky učili naspamäť a aj takýmto spôsobom ich chceme prinútiť, aby si prečítali aj úvahy, ktoré k odpovedia vedú.

Štruktúra kapitol a sekcií je usporiadaná tak, že ak ju čitateľ začne postupne čítať a vstrebávať, nemal by mať v žiadnom momente problém s pochopením konceptov, s ktorými sa aktuálne v texte operuje. Podobne je to aj s poradím otázok v rámci jednotlivých sekcií. Ak sa čitateľ rozhodne začať knižku čítať uprostred, radíme začať aspoň na začiatku niektorej sekcie, nakoľko tam sa zvyčajne nachádzajú otázky venujúce sa definíciám pojmov. Aby sa kniha dala používať aj referenčne (napr. čitateľ hneď skočí na konkrétnu otázku), odpovede navyše často čitateľovi pripomínajú už vyslovené poznatky, ktoré sú v danom kontexte dôležité.

Keď sa do knihy začítate, zistíte, že jednotlivé otázky označujeme dvoma typmi metainformácií.

1. Prvou sú kľúčové slová, ktorých úlohou je obsahovo zaradiť otázku ešte jemnejšie, ako to definuje štruktúra knihy. Pre čitateľa to poskytuje možnosť vyhľadať všetky otázky venujúce sa určitému pojmu (užitočné najmä pre elektronickú verziu knihy).

2. Druhou informáciou je úroveň otázky v rámci Bloomovej taxonómie. Túto čitateľ môže využiť ako približné označenie náročnosti otázky a zároveň typ odpovede resp. mentálnej činnosti, aká sa od riešenia očakáva (viď nižšie).

Bloomova taxonómia je systém klasifikácie úrovne znalostí. Má 6 úrovní a každá úroveň označuje, do akej miery človek (jedinec) rozumie určitému konceptu. Čím vyššia úroveň, tým hlbšie sú znalosti človeka o danom koncepte. Úrovne sú zjednodušene definované takto:

1. *Zapamätať si*. Človek má povedomie o existencií konceptu.

2. *Porozumieť*. Človek dokáže koncept vysvetliť, chápe ako princípy s ním spojené fungujú.

3. *Aplikovať*. Človek je schopný koncept aplikovať na riešenie problémov.

4. *Analyzovať*. Človek je schopný koncept analyzovať, porovnávať s inými.

5. *Zhodnotiť*. Človek dokáže vyhodnotiť silné a slabé stránky konceptu.

6. *Tvoriť*. Človek dokáže priniesť nové poznatky ohľadom konceptu.

Každá otázka, ktorú nájdete v tejto knihe, je označená jednou z týchto úrovní. Čím vyššiu úroveň otázok čitateľ zvláda riešiť, tým hlbšie sú jeho znalosti o koncepte. Ako čitateľ uvidí, z úrovne na úroveň sa tiež mení štýl myslenia aj odpovede na otázky. Zároveň si možno všimnúť, že len odpovede na otázky prvej úrovne sa ako tak dajú naučiť naspamäť. Na druhej úrovni už je to za hranicou efektívnosti a od tretej úrovne vyššie je to prakticky nemožné. Práca s otázkami vyššej úrovne teda implicitne núti skutočne konceptom porozumieť.

Výzva

Táto kniha je v čase, keď ju čítate stále živá. Jej autori priebežne usilovne pracujú na jej zlepšeniach do budúcnosti. Existuje nielen v tlačenej podobe (v ktorej zlepšenia môžeme uplatniť žiaľ len novým vydaním), ale jej jednotlivé otázky sú pre študentov FIIT STU k dispozícií aj v elektronickej podobe vo vzdelávacích systémoch (tam sme s úpravami obsahu flexibilnejší). Preto neváhajte kontaktovať autorov s akýmikoľvek pripomienkami a námetmi na zlepšenie obsahu tejto knihy. Špecificky, ak máte námety na otázky či máte pocit, že niektoré odpovede dostatočne nevysvetľujú čo majú, ozvite sa. Želáme príjemné vzdelávanie sa.

Kapitola 1

Úvod do softvérového inžinierstva

1.0.1
softvérové inžinierstvo

Čo je to inžinierstvo?

V literatúre by sme narazili na veľa definícií. Zjednotením tých, ktoré považujeme za relevantné dostávame niečo takéto:

Inžinierstvo je systematický a kreatívny prístup k riešeniu (praktických) problémov. Inžinierstvo sa opiera sa o bázu existujúcich poznatkov a osvedčených postupov a nové problémy sa snaží riešiť ich syntézou. Inžinierstvo zároveň prispieva k rozširovaniu tejto bázy. Inžinierska práca má spravidla charakter projektov.

Poznámka: na doplnenie môžeme inžinierstvo vymedziť voči iným pojmom:

- Inžinierstvo *nie je výroba*. Aj výroba je technickou činnosťou, no definovaná je ako opakovanie presne daných postupov (ktoré ale môžu byť výsledkom inžinierskej práce). Naproti tomu je projektová práca, charakteristická pre inžinierstvo, vždy unikátna.
- Inžinierstvo *nie je remeslo*. Hoci oba koncepty zdieľajú opieranie sa o osvedčené postupy, v inžinierstve je navyše prítomná syntéza poznatkov a postupov a inovácie. Zároveň možno povedať, že „remeselné“ zručnosti sú často súčasťou výbavy inžinierov (napr. technické kreslenie, vytváranie 3D modelov ale aj zručnosti ako akademické písanie či rétorika).
- Inžinierstvo *nie je umenie*. Zdieľajú prvok kreativity, no umenie vo všeobecnosti nie je určené na riešenie problémov, tým menej praktických (výnimkou je možno kategória úžitkového umenia). Je ďaleko menej ukotvené v existujúcich poznatkoch a nie je to exaktná disciplína.
- Inžinierstvo *nie je výskum*. Výskum sa primárne zameriava na odhaľovanie nových všeobecných poznatkov a postupov, cieľom inžinierstva je riešiť špecifické problémy. Oba koncepty sú často v synergii. Výskum spravidla vyžaduje

aj inžiniersku prácu (ak napríklad skúmame správanie ľudí na sociálnej sieti, najskôr ju musíme vytvoriť). Na druhej strane, inžinierska práca stavia na výsledkoch výskumu (napr. pri tvorbe bezpečnostného kamerového systému využijeme metódy umelej inteligencie a strojového spracovania obrazu).

1.0.2 Čo všetko zahŕňame pod pojem softvér?

softvér

Intuitívne si pod pojmom softvér spravidla predstavujeme *programy* (aplikácie), resp. ich zdrojové kódy. Pojem softvér je však širší a zahŕňa aj *postupy* (niekedy celé metodológie) ako tento softvér používať. Rovnako tak zahŕňa aj *dokumentáciu* k týmto programom a postupom spolu so špecifikáciou, scenármi použitia, testovacími scenármi, opismi návrhu, testovacími údajmi a príručkami [11].

1.0.3 Uveďte, čo to je softvérové inžinierstvo.

softvérové inžinierstvo

Je to disciplína, systematicky sa zaoberajúca vývojom, údržbou, prevádzkou a vyradením softvéru. Vývoj pokrýva činnosti tvorby softvéru predtým než je daný do prevádzky. Údržba označuje činnosti administrácie a úprav softvéru počas toho, ako je v prevádzke. Vyradenie predstavuje činnosti odstránenia softvéru z prevádzky [11].

Samozrejme, keďže ide o inžiniersku disciplínu, pre softvérové inžinierstvo platia všetky charakteristiky uvedené na začiatku tejto kapitoly (otázka 1).

1.0.4 Kedy sa začal používať pojem softvérové inžinierstvo? Aká stará (mladá) je disciplína softvérové inžinierstvo?

softvérové inžinierstvo

Vznik pojmu softvérové inžinierstvo sa spája s konferenciou NATO v roku 1968 [12], kedy sa datuje aj tzv. *softvérová kríza*. V tom čase si vedci a inžinieri uvedomili, že sa priveľa softvérových projektov stáva neúspešných (vďaka ich narastajúcej zložitosti, danej vtedy najmä zvyšovaním výkonu počítačov) a že nevyhnutne musí vzniknúť disciplína, ktorá sa bude systematicky zaoberať prístupmi k tvorbe, údržbe, prevádzke a vyradeniu softvéru.

1.0.5 Čo je to softvérový projekt?

projekt, softvérový projekt

Projekt je dočasné (vopred časovo ohraničené) úsilie s cieľom vytvorenia jedinečného výrobku alebo služby. Výstupom softvérového projektu je softvér. Viac o softvérových projektoch sa dočítate v otázke 70.

Pozor: softvérový projekt nie je softvér. Projekt je v činnosť, softvér je artefakt (človekom vytvorená vec). Znie to možno triviálne, ale pojmy projekt a softvér sa zvyknú zamieňať, najmä vďaka zaužívanému označovaniu pracovných balíkov zdrojových kódov vo vývojárskych nástrojoch (IDE) ako „projektov“.

1.0.6 Kedy je softvérový projekt úspešný?

projekt, softvérový projekt

Pozor: otázka sa pýta na projekt, nie na softvér samotný.

Projekt je úspešný vtedy, ak splní všetky požiadavky do stanoveného termínu a v stanovenom rozpočte. Iba asi jedna štvrtina projektov v praxi je podľa tejto definície úspešná.

Za čiastočný úspech sa zvykne považovať to, ak projekt splní všetky požiadavky, ale prekročí stanovený čas či prostriedky (dosť ale závisí na miere prekročenia).

Treba k tomu poznamenať, že vývoj softvéru je spravidla poznačený zmenami, ktoré sa dejú v prostredí, v ktorom sa softvér vyvíja. Menia sa požiadavky na softvér a to so sebou prináša často aj zmeny termínov a rozpočtu. Preto treba softvérový projekt chápať ako dynamicky sa vyvíjajúci súbor činností. Nie je potom vždy jednoduché vyhodnotiť úspešnosť projektu podľa pravidla vyššie.

1.0.7 Kedy považujeme softvér za úspešný?

softvér

Pozor: otázka sa pýta na softvér, nie na projekt ktorý k nemu viedol. Softvér ako taký môže byť úspešný aj napriek tomu, že projekt, v ktorom vznikol úspešný nebol (nedodržal vymedzený čas, prostriedky či dokonca špecifikáciu).

Musí napĺňať potreby svojich používateľov.

1.0.8 Od čoho závisí, či bude softvérový projekt úspešný?

projekt, softvérový projekt

Rafinovaná odpoveď by znela, že od všetkého čo s projektom súvisí. Ale takáto odpoveď je príliš všeobecná a preto nepostačujúca. Vplyvov na úspech projektu je naozaj veľa. Niektoré sú prítomné v každom projekte (napr. kvalita plánu), iné vychádzajú zo špecifík predmetných oblastí (napr. projekt vývoja účtovníckeho softvéru ovplyvňujú nečakané legislatívne zmeny).

Medzi zásadné a univerzálne faktory ovplyvňujúce úspech softvérového projektu patria:

- stanovený plán (činnosti, ktoré sa majú vykonať spolu s časom a zodpovednosťami)
- stanovený rozpočet
- stanovenie rolí a zodpovedností účastníkov projektu

1.0.9
vývoj softvéru

Aký je rozdiel medzi vývojom softvéru a programovaním?

Programovanie je len jednou z mnohých aktivít vývoja softvéru.

Laici zvyknú oba pojmy vnímať ako rovnocenné, ide však o hrubý omyl. Programovanie je len jednou z činností vykonávaných v rámci vývoja softvéru. Hoci je jednoznačne nutnou podmienkou vzniku softvéru, nie je podmienkou postačujúcou. Predchádzať mu musí zber a dokumentovanie požiadaviek na vytváraný softvér a následné vytvorenie a postupné zjemňovanie návrhu softvéru spolu so spôsobom jeho overovania (testovania). Až potom je možné pristúpiť k programovaniu (implementácii), počas ktorého taktiež softvér dokumentujeme a testujeme.

Poznámka: okrem vyššie uvedeného, vývoj softvéru sa taktiež nezaobíde bez manažmentu, ktorý zahŕňa napríklad plánovanie, vyhodnocovanie rizík, komunikáciu vo vývojovom tíme či komunikáciu so zákazníkom. V rámci softvérového inžinierstva rozlišujeme vývoj softvéru a manažment vývoja ako dve podskupiny procesov. V bežnej komunikácii sa však často oba pojmy spoločne nepresne označujú ako „vývoj softvéru“.

1.0.10
vývoj softvéru, roly

Aké roly (účastníkov) rozlišujeme počas života softvéru?

Roly v softvérových projektoch prirodzene vyplývajú s rôznych typov činností a zodpovedností, ale taktiež aj zo záujmov zúčastnených strán. Neexistuje jediné rozdelenie. Prakticky každá inštitúcia vyvíjajúca softvér si ho nejak definuje (veľmi veľké spoločnosti môžu mať desiatky až stovky definovaných rolí). Predsa len však existujú niektoré typické označenia (ich prehľad nájdete na obrázku 1.1), charakterizujúce činnosti a zodpovednosti dotyčných účastníkov softvérových projektov. Ani tie však nemajú úplne jasné hranice a niekedy sa prekrývajú.

Najčastejším je rozlišovanie rolí podľa etáp životného cyklu softvéru: *analytik*, *návrhár*, *programátor* (ten čo implementuje), *tester*, *údržbár* (ten čo softvér udržiava po jeho nasadení do prevádzky, angl. *maintenance person*). Často používanou rolou v praxi je „*vývojár*“ (*developer*). Týmto pojmom sa označuje predovšetkým programátor, no často pod neho spadá aj návrh, testovanie či dokonca analýza. Po odovzdaní softvéru do prevádzky vstupujú do procesu aj *operátori, administrátori*, či ďalší podporný personál pre zabezpečenie používania softvéru.

Stretnúť sa možno aj so špecifickejšími označeniami rolí, ktoré vyplývajú z toho, že dotyčná osoba je *špecialistom* na určitú oblasť. Napríklad *architekt* je návrhárom sústrediacim sa na architektúru softvéru a vôbec celkový pohľad na softvér. *Databázový špecialista* sa sústreďuje na otázky návrhu a implementácie mechanizmov súvisiacich s uchovávaním údajov. *Návrhár používateľských rozhraní* (angl. *frontend designer*) sa venuje oblastiam ako grafický návrh a štýl rozhrania, informačná architektúra aplikácie, použiteľnosť, UX a pod.

S termínom „*špecialista*“ sa často spája termín „*konzultant*“. Konzultantom je špecialista, ktorý zastáva predovšetkým poradnú úlohu. Do projektu je spravidla zahrnutý len na obmedzený čas (aj keď veľké projekty ho vedia využiť aj „celého“). Jeho

úlohou nie je samotná realizácia záležitostí v projekte ale skôr pomoc pri dôležitých rozhodnutiach, odborný dozor, mentorovanie a odovzdávanie skúseností.

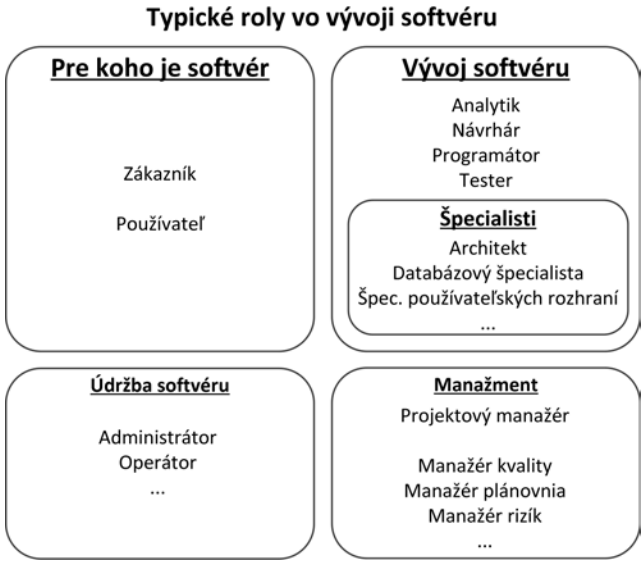
Pri definovaní rolí sa tiež možno stretnúť s rôznymi prívlastkami, ktoré sa k názvom jednotlivých rolí pridávajú. V praxi časté je napríklad používanie prívlastkov *junior* a *senior*, ktoré v sebe nesú informáciu, akú veľkú zodpovednosť daná osoba nesie, aké sú jej rozhodovacie právomoci či aké veľké skúsenosti má.

Vyššie uvedené roly sú tzv. technické. Vystihujú to, čo sa v projekte deje z pohľadu vecnej realizácie softvéru. Okrem nej je však potrebný aj manažment projektu, preto zvykneme rozlišovať viaceré manažérske roly. V menších projektoch je to univerzálny *projektový manažér*. Vo väčších sú manažéri rozlíšení podľa oblastí manažmentu, napr.: *manažér kvality*, *manažér plánovania*, *manažér rizík*.

Nesmieme opomenúť roly vyplývajúce z toho, že softvér vytvárame vždy „pre niekoho“. Preto rozlišujeme rolu *zákazníka*. Od neho primárne pochádza potreba projekt riešiť a tiež požiadavky na softvér, ktorý sa realizuje. Hoci sa to na prvý pohľad nezdá, aj on je súčasťou projektu, v niektorých prístupoch dokonca v celom jeho priebehu. Zároveň treba na strane „klientov softvéru“ rozlíšiť aj rolu *používateľa*. Zároveň si musíme uvedomiť si, že zákazník a používateľ nemusia byť nutne tie isté osoby, dokonca môžu mať aj dosť odlišné záujmy.

V projektoch všeobecne máme aj podporný personál, ktorý sa stará o vhodné prostredie pre tých pracovníkov, ktorí vytvárajú produkt.

Treba si tiež uvedomiť, že jednu rolu môže zastávať viacero ľudí (nevyhnutnosť pri väčších projektoch) a tiež, že jeden človek môže zastávať viacero rolí (napr. programátor môže byť zároveň testerom, architekt programátorom).



Obr. 1.1: V softvérových projektoch rozlišujeme niekoľko typických rolí

1.0.11
zákazník, používateľ

Aký je rozdiel medzi zákazníkom a používateľom?

Zákazníkom je osoba (alebo organizácia), ktorá má na vytvorení a fungovaní softvéru primárny záujem a zároveň doň vkladá svoje financie. Používateľom je osoba, ktorá so softvérom interaguje.

Zákazník nie je vždy používateľom softvéru, používateľ nie je vždy zákazníkom.

Základný rozdiel je v prítomnosti záujmu na existencií softvéru. Zákazník ho z definície má. Používateľ ho mať môže ale nemusí. Extrémnym prípadom môže byť softvér, keď je používateľ prinútený softvér používať, aj keď sám nechce. Napríklad zamestnanci výrobného podniku musia každý mesiac vyplniť výkazy svojej práce v nato určenej aplikácií. Z ich pohľadu (používateľov softvéru) ide o zbytočnú aktivitu. Naproti tomu ich zamestnávateľ (zákazník softvéru) má na nej eminentný záujem.

Dôsledkom otázky záujmu je samozrejme aj možnosť, že zákazník aj používateľ majú na existencií softvéru záujem, ale každý z nich trochu iný. Úlohou pri vývoji tohto softvéru je preto vždy snaha o skĺbenie týchto záujmov. Predstavme si príklad, pri ktorom banka (zákazník) zamýšľa zaviesť nový vnútropodnikový softvér pre svojich finančných analytikov (používatelia), pričom jej zámerom je jednak zrýchliť prácu analytikov a okrem toho aj zaznamenávať viac informácií o spracúvaných finančných operáciách. Analytici chápu že nové informácie získavať treba a v starom rozhraní sa to nedá a tiež by si radi zrýchlili svoju prácu, no zároveň nie sú myšlienke nového rozhrania naklonení – sú zvyknutí na to staré.

1.0.12
softvér, program

Čo je to program (v kontexte softvérového inžinierstva)?

Program je počítačom vykonateľný postup, množina inštrukcií vykonávaná počítačom na splnenie nejakej úlohy.

1.0.13
softvér, program,
softvérový produkt

Aký je rozdiel medzi programom a softvérovým produktom?

Softvérový produkt je program upravený do takej podoby, že je (pohodlne, rozumne) využiteľný človekom iným ako je autor programu. Produkt tiež môžu a často aj upravujú iní ľudia ako autor(i). Je dôležité, aby proces úpravy bol efektívny, teda aby program bol vhodne vnútorne štruktúrovaný a dokumentovaný. Produkt vnímame ako niečo, čo možno „zabaliť do krabice a predávať“.

1.0.14
softvér, systém

Čo je to systém?

V najširšom zmysle: niečo poskladané z viacerých častí.

V inžinierstve však spravidla uvažujeme, že tieto časti interagujú a spolupracujú v rámci plnenia spoločného účelu.

1.0.15
softvér, program

Čo je to programový systém?

Sústava viacerých spolupracujúcich programov.

1.0.16
softvér, program

Čo je to softvérový systém (v produktovej kvalite)?

Sústava viacerých spolupracujúcich softvérov. Dôležitý je pritom rozdiel oproti programovému systému: ten ešte z definície nemá produktovú kvalitu, softvérový systém už áno.

1.0.17
softvér, program, softvérový produkt

Porovnajte zdroje potrebné na vývoj programu a vývoj softvérového systému (v produktovej kvalite).

Ak uvažujeme, že produkt a program majú mať rovnakú funkcionalitu, potom vytvorenie produktu veľmi zhruba vyžaduje až trojnásobok úsilia oproti vytvoreniu programu. To vyplýva z faktu, že produkt musí spĺňať viaceré kvalitatívne štandardy, aby sme ho mohli reálne predávať (napr. musí byť zdokumentovaný, jednoducho použiteľný, musí sa vedieť vysporiadať s neštandardnými vstupmi a pod.) zatiaľ čo program je spravidla len „holou“ realizáciou funkcionality.

1.0.18
program, softvérový produkt

Porovnajte zdroje potrebné na vývoj jednoduchého programu a softvérového systému (v produktovej kvalite).

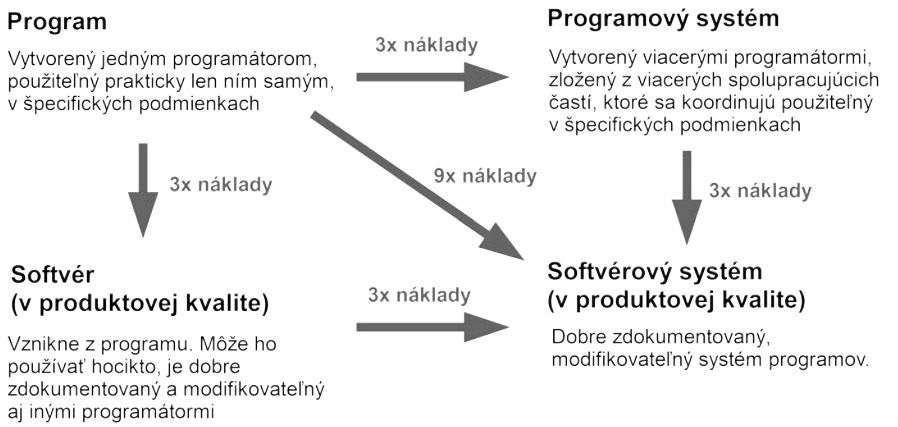
Rozdiel je v princípe až v jednom ráde. Ak je „softvérový systém-produkt“ softvérovým systémom upraveným do produktovej podoby, platí aj v tomto prípade trojnásobné zvýšenie potrebného úsilia (podobne ako pri vzťahu jednoduchého programu a z neho vytvoreného produktu). Započítať však treba aj rozdiel v úsilí na vytvorenie jedného programu (jedným človekom) a softvérového systému, kde do hry vstupuje aj potreba koordinácie práce v softvérovom projekte a koordinácie samotných programov (vytvorenie a používanie rozhraní na základe dohody). Aj tu môžeme rozdiel vnímať veľmi hrubo ako trojnásobný. Po zlúčení týchto rozdielov násobením vychádza takmer jednorádový rozdiel pri prechode od programu k softvérovému systém v produktovej kvalite (obrázok 1.2).

1.0.19
vývoj softvéru, model vývoja

Čo je to model vývoja softvéru? Aké dva základné modely rozlišujeme?

Je to hrubo definovaný postup, určujúci aké činnosti a v akom poradí vykonávame pri vývoji softvéru. Modelov vývoja softvéru poznáme viacero, no najčastejšie sa odvolávame na dva, fundamentálne odlišné typy (viac o modeloch vývoja v otázke 209):

1. *Lineárny model.* Jeho najznámejším zástupcom je *vodopádový model*. V rámci neho postupne vykonávame činnosti analýzy, špecifikácie, návrhu, implementácie a testovania softvéru. Kľúčovou črtou vodopádového vývoja je to, že každou



Obr. 1.2: Rozdiel nákladov na vytvorenie programu a softvérového systému v produktovej kvalite môže byť až jeden rád.

z týchto fáz prechádzame len raz (teda vykonáme pre celý softvér najskôr celú analýzu, potom celý návrh atď).

2. *Iteratívno-Inkrementálny model*. V rámci neho prechádzame cez rovnaké fázy ako pri vodopádovom modeli ale prechádzame cez celú postupnosť viac krát, pričom sa vždy zaoberáme len nejakou časťou softvéru.

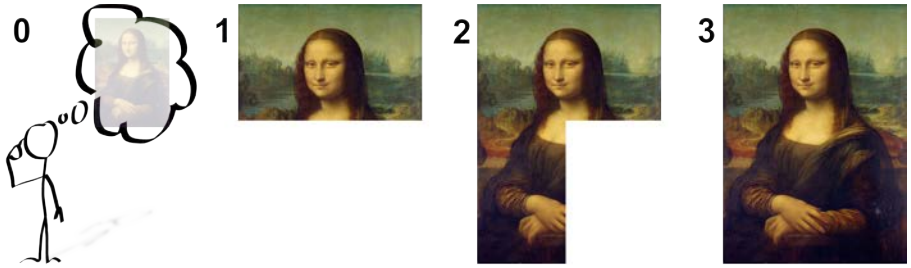
1.0.20
vývoj softvéru, inkrement

Vysvetlite pojem „inkrementálne“, resp. „inkrement“ v súvislosti *s vývojom softvéru.*

Inkrementom označujeme nejaký *prírastok* k softvéru. Môže ísť o novú funkciu, rozšírenie existujúcej, pridanie novej súčiastky, rozšírenie databázy a podobne. Inkrementy môžu nadobúdať veľkosti od veľmi malých (výsledky pár hodinovej práce jedného programátora) po veľmi veľké (výsledky niekoľkomesačnej práce tímu ľudí). Inkrementom môže byť aj niekoľko riadkov programového kódu, nová kapitola v dokumentácii alebo aj zmazanie databázovej tabuľky.

„Inkrementálny“ vývoj softvéru je vytváranie softvéru pridávaním ďalších jeho častí (viac o inkrementálnom vývoji v otázke 69).

Pojem inkrementálneho vývoja možno ilustrovať aj obrázkom 1.3.



Obr. 1.3: Inkrementálny prístup k tvorbe (čohokoľvek) možno ilustrovať ako postupné pribúdanie častí celku vždy vo finálnej podobe.

1.0.21
vývoj softvéru, iterácia

Vysvetlite pojem „iteratívne“, resp. „iterácia“ v súvislosti *s vývojom softvéru.*

Iterovaním (v softvérovom inžinierstve) označujeme opakované (cyklické) prechádzanie fázami vývoja softvéru (analýza, návrh, implementácia, ...), s cieľom jeho *postupného zlepšovania*. Iteráciou potom označujeme jeden prechod týmto cyklom.

Pojem iteratívneho vývoja možno ilustrovať aj obrázkom 1.4.

Obr. 1.4: Iteratívny prístup k tvorbe (čohokoľvek) možno ilustrovať ako postupné zmeny celku.

1.0.22
vývoj softvéru, inkrement

Prečo je výhodné pristupovať k vývoju softvéru inkrementálne?

Vyrovnávame sa tak so zložitosťou, ktorá patrí medzi vnútorné problémy softvéru (viac v otázke 36). Pokusy o vytvorenie zložitého softvéru „naraz“ v drvivej väčšine prípadov zlyhávajú. Jeho rozdelením na menšie časti (dekompozíciou) a súčasnou prácou len na jednej z nich si problém zložitosti zjednodušujeme. Zároveň umožníme veľmi skoré testovanie a overovanie softvéru po vytvorení každého prírastku. Taktiež umožníme priradiť prírastkom priority, pracovať najskôr na tých najdôležitejších a dodať tak zákazníkovi hodnotný výsledok už počas projektu. Rozdelenie tiež uľahčuje odhadovanie nákladov (ktoré pre jednotlivé prírastky vieme odhadnúť lepšie, ako pre celý softvér), plánovanie (koľko bude trvať vytvorenie prírastku) a napomáha tiež spresňovaniu návrhu softvéru ako takého.

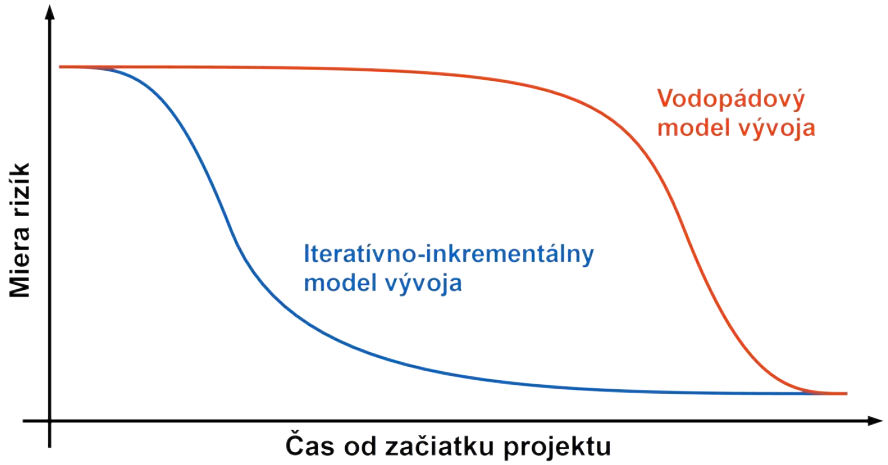
1.0.23

vývoj softvéru, iterácia, riziko

Prečo je výhodné pristupovať k vývoju softvéru iteratívne?

Málokedy máme presnú predstavu o podobe softvéru už na začiatku jeho tvorby. A ak aj nejakú predstavu máme, je rozumné ju považovať za neistú, pokiaľ preukázateľne neoveríme, že zodpovedá požiadavkám, očakávaniam a potrebám zákazníka (používateľa). Ak by sme sa totiž na jej základe pustili do tvorby softvéru, vystavujeme sa veľkému riziku minutia prostriedkov na tvorbu niečoho nepoužiteľného.

Pokiaľ sa však rozhodneme postupovať iteratívne, teda budeme vytvárať softvér viacnásobným prechodom cez všetky fázy tvorby a overovania softvéru, toto riziko sa znižuje. Už po prvom prechode, ktorý nás stojí zlomok celkových nákladov na projekt, je výsledkom prvá (zďaleka nie „dokonalá“) verzia softvéru. Túto prvú verziu možno zákazníkovi ukázať a konfrontovať s jeho očakávaniami pomerne skoro. Iteratívny prístup tak pomáha znižovať riziká, ktoré pri tvorbe softvéru hrozia – ak sme totiž vytvorili niečo, čo predstave zákazníka nezodpovedá, nestratili sme toľko prostriedkov (obrázok 1.5). Oveľa jednoduchším sa stáva aj spresnenie ďalších krokov vo vývoji, o ktorých by sme však na začiatku projektu ešte nemohli mať predstavu.



Obr. 1.5: Pri iteratívnom vývoji sa počet/miera zníži rýchlo, pri vodopádovom vývoji zostáva riziko až do posledných fáz rovnako veľké.

1.0.24

vývoj softvéru, inkrement

Aká je najvýhodnejšia veľkosť prírastkov pri inkrementálnom vývoji softvéru?

Ak je hlavným prínosom inkrementálneho prístupu k tvorbe softvéru možnosť využitia dekompozície (rozkladu na menšie časti za účelom redukcie zložitosti), potom by prírastky mali byť čo najmenšie.

Na druhej strane, prírastky nemožno zmenšovať donekonečna. V prvom rade, prírastok vždy predstavuje časť funkcionality (niečo užitočné pre používateľa) a pokiaľ už funkcie softvéru nevieme ďalej dekomponovať, ohraničuje to aj veľkosť prírastkov.

Každý prírastok by totiž mal byť zmysluplný a to z pohľadu funkcionality (užitočnosti pre používateľa) znamená pridanie nejakej novej funkcie resp. vlastnosti softvéru, ktorá sa aj dá overiť. Preto za príspevok nemožno vydávať napríklad rozšírenie dátového modelu nejakej aplikácie (hoci ako také môže vyžadovať netriviálne úsilie), bez toho aby sme implementovali aplikačnú logiku, ktorá by ho nejak zmysluplne využívala.

Okrem toho, čím menšie prírastky sú, tým ich bude viac. Viac prírastkov môže viesť k návrhu a realizácii viac softvérových súčiastok, čo zväčšuje problémy s ich zviazanosťou (zložitosťou spolupráce týchto súčiastok, ktorá ultimátne softvér príliš skomplikuje).

Veľkosť prírastkov musí byť preto vyvážená. Mali by byť dostatočne malé, aby sme ich vedeli pridávať a overovať rýchlo (význam slova rýchlo samozrejme závisí od typu projektu, ale vo všeobecnosti by malo ísť o niekoľko dní až týždňov). No zároveň všetky prírastky musia spĺňať kritérium zmysluplnosti pre zákazníka a taktiež je potrebné strážiť potenciálne problémy s príliš veľkým množstvom komponentov softvéru.

1.0.25

softvérový produkt, generický softvér, zákaznícky softvér

Aké typy softvérových produktov podľa pôvodcu špecifikácie poznáme? Aký je medzi nimi rozdiel?

Rozlišujeme zákaznícky a generický (nazývaný tiež „krabicový“) softvér. Pri zákazníckom softvéri je pôvodcom špecifikácie konkrétny zákazník (jeho požiadavky vyplývajú z jeho vlastných potrieb) a typicky daný softvér vytvárame len pre neho. Pri generickom softvéri je pôvodcom špecifikácie sama spoločnosť, ktorá softvér vyvíja, no požiadavky sú založené na potrebách ľudí (používateľoch), ktorým sa softvér má predávať.

1.0.26

softvérový produkt, generický softvér

Uveďte príklady softvérových produktov generického typu.

Predstaviť si možno akýkoľvek „krabicový“ softvér, ktorý dostať či už v „kamennej“ predajni alebo v elektronickom obchode: od operačných systémov cez kancelárske balíky až po hry.

1.0.27

softvérový produkt, generický softvér, cots, mots

Porovnajte typy softvéru COTS a MOTS. Čo majú spoločné a čo rozdielne?

COTS (commercial off the shelf) aj MOTS (modified off the shelf) sú obidva typmi generického softvéru, z čoho vyplývajú ich spoločné vlastnosti (predaj väčšiemu množstvu vopred neznámych zákazníkov). Rozdiel medzi nimi spočíva v konečnom prispôsobení podoby softvéru zákazníkovi: MOTS softvér treba pri jeho inštalácii pre zákazníka vhodne nakonfigurovať (napr. systém na správu obsahu webového portálu – *content management system,* CMS), COTS je pre všetkých zákazníkov inštalovaný rovnako (napr. kancelársky balík). Úprava softvéru pre potreby zákazníka (pri MOTS softvéri) môže byť aj náročnejšia ako konfigurácia pri inštalácii. Podstatné je, že zá-

kazník kupuje „krabicový“ softvér, len tú krabicu mu dodávateľ prispôsobí špecifickým potrebám.

Typy COTS a MOTS by sme nemali brať ako dve jasne oddelené skupiny softvérov, ide skôr o dva extrémy v rámci jedného hľadiska: dnes je možné takmer každý softvér pri inštalácii modifikovať, otázna je skôr miera a najmä to, kto to prispôsobenie robí (dodávateľ alebo zákazník).

1.0.28
softvérový produkt, zákaznícky softvér

Uveďte príklady softvérových produktov zákazníckeho typu.

Predstaviť si možno akýkoľvek softvér, ktorý si na zákazku objednáva nejaká inštitúcia. Napríklad: akademický informačný systém, obchodný register, internet banking.

1.0.29
softvérový produkt, zákaznícky softvér

Aké typy zainteresovaných (angl. stakeholder) na strane zákazníka možno rozlíšiť v prípade zákazníckeho softvéru?

V prvom rade na strane zákazníka určite nájdeme ľudí, ktorí sú pôvodcami požiadavky softvér vytvárať. To sú často manažéri na vysokých pozíciách. Ich záujem na projekte tvorby softvéru je prirodzene jeho úspešné ukončenie a prínos ich spoločnosti za použitia primeraných prostriedkov.

Druhou zvyčajnou skupinou sú zamestnanci zákazníckej spoločnosti ako budúci používatelia vytváraného softvéru. Ich záujmom je predovšetkým prínos softvéru pre ich prácu (až tak ich nezaujímajú náklady spojené s vytvorením softvéru a nemusia si byť vedomí strategického významu obstarania softvéru tak, ako by si to mal uvedomovať manažment).

Budúcimi používateľmi nemusia byť len zamestnanci zákazníckej spoločnosti. Napríklad používatelia internet banking aplikácie sú ľudia mimo prostredia banky a nie sú vopred známi (pričom však stále ide o zákaznícky softvér). Za zainteresovaných (stakeholders) ich považovať musíme, lebo aj ich záujmy (napríklad dobrú použiteľnosť) musí softvér odrážať.

Treťou skupinou zainteresovaných sú ľudia, ktorých úlohou je softvér uvádzať do života. Ide spravidla o ľudí s IT zameraním (IT oddelenia zákazníckej spoločnosti) a na projekte sa často podieľajú ako technickí konzultanti, administrátori.

1.0.30
softvérový výrobok, generický softvér

Kto plní úlohu zainteresovaného „zákazníka“ v prípade generického softvéru?

Niekto v rámci spoločnosti, ktorá softvér vyvíja, kto projekt fakticky inicializoval, pretože na trhu identifikoval príležitosť. V malej firme môže ísť o jedného človeka, spravidla manažéra. Veľké firmy majú na identifikovanie príležitostí špeciálne vytvorené oddelenia (napr. oddelenie stratégie), ktoré prichádzajú s návrhmi na vytváranie nového softvéru na základe svojich prieskumov trhu a sú tak de facto pôvodcami požiadaviek na nový softvér. Projekty tvorby generického softvéru však musia „posvä-

tiť“ a prebrať za ne zodpovednosť aj výkonné manažmenty firiem, ktoré sa tak stávajú primárnymi zainteresovanými (*stakeholdermi*).

Stáva sa tiež, že pôvodcom špecifikácie je iná spoločnosť, než tá, ktorá softvér bude vyvíjať. Hoci takáto situácia môže pripomínať skôr zákaznícky softvér (jedna spoločnosť si objedná softvér u inej spoločnosti), rozdiel je v tom, koho potreby má funkcionalita softvéru naplniť: v prípade zákazníckeho softvéru je to spoločnosť, ktorá si softvér objednáva (napr. doručovateľská spoločnosť potrebuje softvér na podporu svojich logistických operácií), no pri generickom softvéri je cieľový zákazník niekto ďalší a pôvodná organizácia, ktorá si softvér „objednala“, z neho chce benefitovať len „nepriamo“: jeho predajom alebo marketingovým využitím (napr. maloobchodný reťazec si u softvérovej firmy objedná vývoj aplikácie porovnávania cien výrobkov pre bežných zákazníkov maloobchodných reťazcov). V takomto prípade je vhodné pozerať sa na dvojicu objednávateľ-zhotoviteľ ako na konzorcium (kvázi jednu firmu), ktorá ako celok vytvára generický softvér pre skupinu zákazníkov mimo konzorcia.

1.1 Špecifiká a problémy tvorby softvéru

1.1.1
zákonité problémy

Vymenujte štyri základné podstatné vnútorné problémové vlastnosti softvéru.

Zložitosť (*complexity*), podriadenosť (*conformity*), premenlivosť (*changeability*) a neuchopiteľnosť (*invisibility,* v slovenčine doslova neuchopiteľnosť).

1.1.2
podriadenosť, tvorba softvéru, zákonité problémy

Charakterizujte problém podriadenosti v tvorbe softvéru.

Vnútorný tlak na zmenu. Podriadenosť v tvorbe softvéru charakterizuje nechcený odklon vývoja od dohodnutej špecifikácie smerom k *domnelým* potrebám používateľa resp. zmeny, ktoré softvér približujú k *aktuálnym* podmienkam prostredia, v ktorom má byť nasadený. Na prvý pohľad by sa mohlo zdať, že je to v poriadku, veď predsa chceme, aby bol používateľ spokojný a chceme aby softvér vyhovoval prostrediu, v ktorom je nasadený. Cieľom tvorby softvéru je však často *zmena* spôsobu práce používateľov a zmena podmienok v prostredí nasadenia, pretože sú pre zákazníka (pre jeho biznis) nevyhovujúce. V takom prípade podriadenosť bráni zefektívneniu biznisu, lebo budúci softvér ťahá k aktuálnemu stavu prostredia.

Príklad: Banka sa rozhodla zaviesť nový vnútrofiremný softvér pre podporu vyhodnocovania žiadostí o hypotéku. Jednou z požiadaviek je aj to, aby zamestnanec, ktorý má o udelení hypotéky v konečnom dôsledku rozhodnúť, mal všetky informácie, na základe ktorých sa rozhoduje, bez výnimky na jednej obrazovke. Zamestnanec má v obrazovke označovať, ktoré informácie ho pri rozhodovaní ovplyvnili a akým smerom. Rozhodovanie má zároveň prebiehať len v špeciálnej miestnosti, kde okrem počítača, na ktorom beží aplikácia nie sú povolené iné zariadenia ani papierová dokumentácia. Cieľom banky je takto lepšie sledovať, na základe čoho sa zamestnanec rozhodol a určiť mieru je ho zodpovednosti za rozhodnutie. Taktiež je cieľom aby banka neskôr

mohla komplexne vyhodnocovať, aké všetky informácie treba od klientov požadovať a ktoré z nich sú určujúce pri odhade, či zákazník hypotéku v danej výške bude schopný splácať.

Pri takejto špecifikácii si ľahko možno predstaviť „nesúhlas“ zamestnancov banky so zmenami oproti pre nich pohodlnejšiemu starému režimu, v ktorom mohli používať papierové podklady a v ktorom nemuseli explicitne označovať faktory svojho rozhodovania. Podriadenosť by sa v takomto prípade prejavila požiadavkami zamestnancov, aby si mohli podklady tlačiť či aby sa rozhranie čo najviac ponášalo na to v pôvodnom starom softvéri. Vývojári nového softvéru by tiež bez explicitnej a jasnej požiadavky o obmedzenej dostupnosti obrazovky s finálnym rozhodovaním mohli ľahko z doterajšej praxe usúdiť, že môže byť dostupná kdekoľvek v banke, pokiaľ sa prihlási používateľ so správnymi právami.

Poznámka: v praxi dochádza k podobnému fenoménu, ktorý však s podriadenosťou v tvorbe softvéru nesúvisí a je dôležité si ho s ňou nezamieňať: pre zákazníkov, ktorí potrebujú nový softvér je najvýhodnejšie, ak môžu kúpiť už existujúci softvér (je to rýchlejšie a často aj lacnejšie). Stáva sa však, že existujúci softvér nie úplne vyhovuje potrebám a biznis procesom zákazníka, následkom čoho zákazník svoje vlastné (fungujúce) procesy musí ohýbať, aby sa softvér dal použiť. To je rovnako negatívny fenomén, ale ako sme už povedali, s podriadenosťou v tvorbe softvéru nesúvisí.

1.1.3 Ako zmierniť problém s podriadenosťou vo vývoji softvéru?

podriadenosť, tvorba softvéru, zákonité problémy

Podriadenosť, teda odklon od špecifikácie v dôsledku pôsobenia prostredia, zmiernime *dobrým návrhom* softvéru (podrobným a jednoznačným). Keďže takýto návrh je na začiatku projektu ťažké dosiahnuť v plnej šírke bez chýb, využijeme iteratívno-inkrementálny vývoj, ktorý umožní vytvorenie návrhu v dostatočnej hĺbke po častiach.

Dôležitým nástrojom v boji s podriadenosťou je tiež vhodný *manažment zmien*, teda formálny rámec navrhovania, schvaľovania a zapracúvania zmien. Tieto zmeny môžu byť aj pripomienky používateľov, ktoré by sme nemali ignorovať, no taktiež by sme nemali nechať vývojárov, aby ich „bezhlavo“ zapracúvali do softvéru.

1.1.4 Charakterizujte problém premenlivosti v tvorbe softvéru.

premenlivosť, tvorba softvéru, zákonité problémy

Vonkajší tlak na zmenu. V máloktorom softvérovom projekte dokážeme na začiatku presne určiť aké sú na softvér požiadavky, potom ich až do konca projektu nemeniť a softvér vytvoriť podľa nich. Nemožno sa spoľahnúť, že sa existujúce požiadavky nezmenia. V priebehu riešenia projektu sa ukazujú nové skutočnosti a zákazník si uvedomuje nové požiadavky, prípadne reviduje už existujúce (ľudská predstavivosť jednoducho neumožňuje predstaviť si konečnú podobu softvéru). Navyše, aj vopred dobre preskúmané okolnosti sa počas projektu môžu zmeniť, napr. legislatíva diktujúca niektoré pravidlá, ktoré softvér musí dodržať, môže byť počas projektu zmenená a treba sa jej za behu prispôsobovať. Zapracúvanie akýchkoľvek zmien do softvéru stojí úsilie, a ak sa so zmenami nepočítalo alebo majú byť uskutočnené neskoro, môže

byť toto úsilie tak neprimerane veľké, že ohrozí rozpočet projektu a dodržanie termínov dodania.

1.1.5 Ako zmierniť problém s premenlivosťou vo vývoji softvéru?

premenlivosť, tvorba softvéru, zákonité problémy

Premenlivosť v tvorbe softvéru, teda neustálu hrozbu zmien špecifikácie softvéru tlakom „z vonka“, zmierňujeme predovšetkým *iteratívno-inkrementálnym vývojom* a *prototypovaním*. Tým urýchlime spresňovanie požiadaviek zákazníka a minimalizujeme dopad prípadných zmien.

Ďalším spôsobom (vhodne dopĺňajúcim iteratívno-inkrementálny vývoj) je *modulárny návrh* softvéru s malou zviazanosťou jednotlivých súčiastok (na všetkých úrovniach: komponentov, balíkov, tried aj metód). Takýto návrh pasívne zmierňuje úsilie potrebné na vykonávanie zmien, pretože zmeny jednej súčiastky len minimálne ovplyvňujú zmeny ďalších. Modulárnosť tak pomáha zmierňovať aj dopad zmien, ktoré dopredu nevieme ovplyvniť (externé riziká ako napr. legislatívne zmeny). Modulárnosť však treba udržiavať (v rámci prehliadok zdrojových kódov a refaktoringu), pretože softvér má v tomto ohľade (v dôsledku zapracúvania zmien) tendenciu degradovať.

Zapracúvanie zmien môže viesť k vzniku chýb a tým pádom k ďalšiemu zbytočnému vynakladaniu úsilia. Ďalším z prostriedkov zmierňovania problému premenlivosti je udržiavanie vysokého *pokrytia* softvéru automatickými *testami*, ktoré nám časť chýb pomôžu rýchlo odhaliť.

V neposlednom rade možno premenlivosť vo vývoji softvéru zmierňovať aj formálnym manažmentom zmien, ktorý (podobne ako pri zmierňovaní podriadenosti) udrží návrhy na zmeny, ich posudzovanie a samotné uskutočňovanie pod väčšou kontrolou.

1.1.6 Charakterizujte problém zložitosti v tvorbe softvéru.

zložitosť, tvorba softvéru, zákonité problémy

Softvér sa skladá z mnohých častí a prepojení (závislostí) medzi nimi. Pre človeka je nemožné sa s nimi detailne zaoberať a zároveň uvažovať o softvéri ako o celku (tento problém si možno všimnúť už pri tvorbe jednoduchých programov). Priveľké je aj množstvo dimenzií (vlastností), cez ktoré na softvér nazeráme. Záležitosti v softvéri sú (aj napriek našej snahe) prepletené: napr. jedna črta softvéru (prípad použitia) môže prechádzať cez všetky úrovne štruktúry softvéru. Zložitosť komplikuje tvorbu, údržbu a vôbec akékoľvek úvahy nad softvérom.

1.1.7 Ako zmierniť problém zložitosti vo vývoji softvéru?

zložitosť, tvorba softvéru, zákonité problémy

Dekompozíciou, teda rozložením celku na menšie časti, s ktorými sa zaoberáme separátne. Využívame ju na mnohých úrovniach a v mnohých podobách, napríklad pri formulovaní požiadaviek, analýze biznis procesov, pri návrhu algoritmov, architektúry, dátových štruktúr, v manažmente pri rozdeľovaní úloh, stanovovaní rozvrhu prác a mnohých ďalších. Viaceré hľadiská dekompozície pritom používame paralelne (pri

modelovaní softvéru napríklad udržujeme štruktúrne, funkcionálne aj behaviorálne modely).

Niekedy dekompozíciu využívame celkom nevedome a automaticky, lebo „inak sa to ani nedá“ (napr. keď programujeme). Niekedy si však nutnosť jej použitia musíme uvedomiť a aj ju vykonať (dobrým príkladom je odhadovanie času prác, či určovanie vhodnosti použitia nejakej technológie, kedy sa podcenením podrobnejšieho pohľadu na veci môžeme dostať do problémov, pretože správne neodhadneme zložitosť softvéru/úloh).

Druhou významnou technikou zmiernenia problému zložitosti je *abstrakcia,* ktorú dosahujeme skrz *modelovanie*. Výberom jedného hľadiska v rámci ktorého model tvoríme (napríklad funkcionálne opisujeme softvér modelom prípadov použitia) automaticky zanedbávame množstvo iných hľadísk, čím pohľad na softvér zjednodušujeme.

1.1.8
neuchopiteľnosť, tvorba softvéru, zákonité problémy

Vysvetlite pojem „neuchopiteľnosť softvéru“. Prečo ide o problém?

Neuchopiteľnosť softvéru znamená, že softvér nevieme vopred opísať tak, aby sme zachytili všetky jeho podstatné detaily. Jediným kompletným opisom softvéru je softvér samotný. Preto kým softvér nevytvoríme, niektoré jeho vlastnosti nevieme vyhodnotiť.

Práve nemožnosť vyhodnocovania vlastností softvéru vopred je dôvodom, prečo je neuchopiteľnosť softvéru problém. Prekáža nám napríklad pri odhadovaní úsilia: jednoducho dopredu netušíme, aké všetky problémy nás pri tvorbe nejakej časti softvéru čakajú. V kombinácií s neopodstatneným optimizmom, nedostatkom skúseností, tlakom manažmentu či zákazníka vedie neuchopiteľnosť k podceňovaniu nárokov na čas a náklady a v konečnom dôsledku k neúspechu projektov.

Častá chyba: neuchopiteľnosť neznamená, že softvér nemôžeme fyzicky vidieť. Ak už existuje, môžeme vidieť zdrojový kód, dokumentáciu, dokonca aj strojový kód. Snáď jediná súčasť softvéru, ktorá nemusí mať fyzickú podobu, sú postupy (používania softvéru). Neuchopiteľnosť však, ako sme už uviedli, chápeme v inom zmysle.

1.1.9
neuchopiteľnosť, tvorba softvéru, zákonité problémy

Čo je dôsledkom neuchopiteľnosti softvéru?

Najvážnejším dôsledkom neuchopiteľnosti („neopísateľnosti“ vopred) softvéru je sťaženie rozhodnutí v softvérových projektoch v dôsledku nedostatku informácií. Nesprávne rozhodnutia v plánovaní rozsahu, rozvrhu a zdrojov pritom môžu mať pre projekt fatálne dôsledky.

Dôsledkom neuchopiteľnosti softvéru v prípadoch, kedy boli uskutočnené zlé odhady, je aj takzvaný syndróm 90% hotovo [4]. Nazývame tak situáciu, kedy sa zdá, že softvér (alebo nejaká jeho časť) je takmer dokončený, no z nejakého dôvodu ešte správne

nefunguje. Tento stav následne trvá ešte dlho a najmä niekoľkonásobne viac ako domnelých zvyšných 10% úsilia.

1.1.10
neuchopiteľnosť, tvorba softvéru, zákonité problémy

Ako zmierniť problém s neuchopiteľnosťou vo vývoji softvéru?

Výstižným modelovaním softvéru aj procesu vývoja.

Prirodzene, žiaden model nezabezpečí kompletný opis softvéru, pomocou kombinácie viacerých modelov však vieme dostatočne presne odhadnúť vlastnosti softvéru a riadiť projekt podľa nich.

Neistotu v rozhodovaní, vyplývajúcu z neuchopiteľnosti softvéru, tiež pomáha zmierňovať prototypovanie (najmä prototypovanie na zahodenie).

Užitočným „nástrojom“ sú tiež skúsenosti s podobným softvérom, projektom či technológiou, ktorá sa má využiť. Vďaka skúsenostiam totiž ľahšie vieme predvídať potenciálne problémy, aj keď z aktuálne vytvorených modelov a prototypov ešte nie sú zrejmé.

1.1.11
neuchopiteľnosť, tvorba softvéru, zákonité problémy

Vysvetlite čo je príčinou syndrómu 90% hotovo?

Príčinou syndrómu 90% hotovo (teda situácie, v ktorej sa zdá, že 90% požiadaviek na softvér je splnených, no reálne do dokončenia softvéru zostáva násobne viac úsilia, než je tých „zvyšných“ 10%) je *neuchopiteľnosť softvéru*.

Neuchopiteľnosť (či neopísateľnosť) softvéru znamená, že softvér nevieme opísať (definovanou množinou modelov) tak, aby sme zachytili všetky jeho aspekty. To nám znemožňuje uvedomiť si, čo všetko ešte treba pre dokončenie softvéru (naplnenie špecifikácie) spraviť.

1.1.12
Škálovateľnosť, problém mierky

Čo znamená pojem „problém mierky“ v softvérovom inžinierstve?

Problém mierky (škálovateľnosti) znamená, že návrhové princípy, metódy a techniky vývoja vhodné pre malé projekty („malý“ softvér) nemusia byť nutne vhodné na veľké projekty („veľký“ softvér) a naopak.

1.1.13
Škálovateľnosť, problém mierky, scrum, manažment komunikácie, manažment plánovania, manažment verzií, manažment zmien

Uveďte príklady prístupov vo vývoji softvéru, ktorých použiteľnosť je ohraničená rozsahom projektov (teda majú problém škálovať).

Na problémy so škálovateľnosťou narazíme predovšetkým v prístupoch *manažmentu softvérových projektov*. Najvypuklejšie sa prejavia pri manažmente komunikácie: projekt na ktorom spolupracuje dohromady 5 ľudí môže byť úspešný, aj pokiaľ nebudeme ich komunikáciu nijak štruktúrovane riadiť (necháme komunikovať každého s každým ad hoc a každých pár dní zvoláme ich poradu). Pri projekte s 10, 50 či 500

vývojármi, prestáva byť takýto prístup efektívny a komunikácia musí byť podstatne viac štruktúrovaná, riadená a plánovaná.

Ďalším príkladom slabej škálovateľnosti v manažmente projektov je plánovanie prác. Napríklad metodológia Scrum, ktorá je určená pre malé tímy (do 10 členov), sa opiera o kolektívne rozhodovanie a zdieľanú tabuľu na evidenciu úloh a ich stavu. V Scrum spoločne celý tím odhaduje, koľko úsilia zaberie tá-ktorá úloha. Úspech tohto odhadovania spočíva práve v tom, že je kolektívne a že úsilie odhadujú tí istí ľudia, ktorý ho aj neskôr vynaložia. Takéto rozhodovanie je však vo väčšej mierke projektu nerealizovateľné: stretnutia, kde by sto vývojárov kolektívne odhadovalo úsilie proporcionálne väčšej množiny úloh sú nemožné. Plánovanie vo veľkých projektoch je potrebné robiť vo viacerých úrovniach, právo rozhodovania je dané hierarchiou, prizývajú sa k nemu experti z dotknutých oblastí, je podstatne viac formalizované. V opačnom garde by škálovanie nefungovalo tiež: množstvo pravidiel by malý tím len zbytočne obmedzovalo.

Ďalším príkladom je manažment verzií. Pri malom projekte často postačuje jedno centrálne úložisko s jednoduchým systémom vetvenia (jeden inkrement = jedna vetva) a všeobecné právo každého vývojára prispievať kedykoľvek do ktorejkoľvek vetvy. Naopak veľké projekty sa bez viacnásobného vetvenia ťažko zaobídu, pretože množstvo naraz pridávanej funkcionality je priveľké. Všeobecné vlastníctvo zdrojových kódov („všetci môžu všetko“) tiež nie je vo veľkých projektoch možné pre nedostatok kontroly.

Podobné obmedzenia platia aj pre manažment zmien: v menšom projekte môžu byť požiadavky komunikované zákazníkom (smerom k vývojárom) priamo na stretnutiach a rozhodnutie o ich akceptácií uskutočnené namieste. Vo veľkých projektoch je potrebné požiadavky na zmeny formálne zachytávať v špeciálnom informačnom systéme, ktorý následne sprostredkúva pomerne zložitý proces ich schvaľovania.

1.1.14
Škálovateľnosť, problém mierky, testovanie, prototyp

Uveďte príklady prístupov vo vývoji softvéru, ktorých použiteľnosť nie je ovplyvnená škálovaním projektu.

Existujú samozrejme aj metódy a techniky, ktoré škálovaním veľkosti projektu dotknuté nie sú (používajú sa rovnako, nezávisle od veľkosti projektu). Príkladom môže byť refaktoring, udržiavanie pokrytia zdrojového kódu testami či prototypovanie na zahodenie. Ďalej, mnohé techniky modelovania softvéru sa škálovaním nemusia meniť: napr. opis používateľských scenárov (prípady použitia) budeme v princípe robiť rovnako aj v projekte s piatimi vývojármi aj v projekte s piatimi tisícami vývojárov. Podobne nezmenené bude aj modelovanie životného cyklu dátovej entity (stavovým modelom) či opis biznis procesu (diagram aktivít).

1.1.15
tvorba softvéru

Čo je to syndróm druhého projektu? Stručne vysvetlite.

Pri svojom prvom projekte si vývojári (tímy, firmy) dávajú na všetko pozor a projekt sa podarí. „Opojení úspechom“ však pri druhom projekte stratia obozretnosť, veľa vecí podcenia a projekt sa dostane do problémov.

1.1.16
tvorba softvéru,
manažment komunikácie,
manažment plánovania

Softvérový systém vyvíja 50 ľudí v 3 krajinách (3 rôzne časové pásma). Aké typy problémov spojené s tvorbou softvéru môžu nastať?

Takýto tím by mal predovšetkým skomplikovanú komunikáciu. Fyzická vzdialenosť členov tímu komplikuje rýchlosť a bezprostrednosť komunikácie, potrebnú najmä pri riešení problémov (ani prostriedky ako video-konferencie či zdieľanie obrazovky nemôžu efektívnosťou priamemu kontaktu konkurovať).

Navyše, rozdiely v časových pásmach navyše môžu spôsobiť že pracovná doba vývojárov by sa mohla prekrývať len málo. To prináša problémy s plánovaním činností, no najmä predstavuje riziko v prípade vzniku problémových situácií, kedy je potrebná súčinnosť členov tímu, ktorí momentálne nie sú dostupní.

V neposlednom rade svoj vplyv zohrávajú aj jazykové a kultúrne rozdiely medzi členmi tímu, ktorí môžu komunikáciu nesprávne interpretovať. Napríklad indickí vývojári sú naproti západným oveľa menej zhovorčiví v reportovaní svojej práce a na vzniknuté problémy sa ich treba explicitne pýtať. Na druhej strane kladú menší odpor pri zavádzaní inovácií do vývojových procesov [9].

1.1.17
starnutie softvéru, údržba

Čo je to starnutie (degenerácia) softvéru?

Starnutie softvéru je postupná degradácia štruktúry v dôsledku dodatočných zmien, pridávania funkcionality a opráv chýb. Nastáva počas fázy údržby v životnom cykle softvéru.

Poznámka: ak softvér nemeníme, de facto nedegeneruje. Zdrojový kód, strojové inštrukcie ani dokumentácia či postupy sa nemôžu „opotrebovať“.

Poznámka: štruktúra softvéru môže byť vynaložením dostatočného úsilia zlepšená a degenerácia zvrátená v rámci procesu refaktoringu. Softvér teda môže byť „omladený“. Vždy však zostáva otázkou, za akú cenu.

Častá chyba: starnutie softvéru neznamená to isté ako „zastaranie“ softvéru (hoci sú tieto pojmy jazykovo veľmi podobné). Starnutím máme namysli postupnú degradáciu štruktúry softvéru v dôsledku zmien, zastaranie softvéru nastáva, keď niekto vytvorí iný softvér, ktorý náš softvér kvalitatívne prekoná.

1.1.18
starnutie softvéru, údržba

Prečo po určitom čase používania softvéru počet chýb v ňom spravidla stúpa, hoci na rozdiel od hardvéru, či iných zariadení sa programové inštrukcie neopotrebúvajú?

Je to v dôsledku zmien – opráv chýb, pridávania novej funkcionality, rozširovania či zmien existujúcej funkcionality v dôsledku zmien prostredia.

1.1.19
projekt, tvorba softvéru

Kedy a prečo vznikol pojem softvérová kríza?

V roku 1968 na konferencií NATO. V tom čase sa ukázalo, že vtedajší spôsob riadenia čoraz zložitejších softvérových projektov nestačí a projekty sú často neúspešné. Toto „uvedomenie“ dalo za vznik softvérovému inžinierstvu - novej disciplíne, aktívne sa zaoberajúcej spôsobmi, ako softvér efektívne vytvárať.

Ako môžeme aj v súčasnosti vidieť (takmer 50 rokov neskôr), problémy s efektívnosťou tvorby softvéru stále nie sú vyriešené. Znamená to, že je softvérové inžinierstvo neúspešné? Nie. Softvérové inžinierstvo (vrátane výskumu) časom vždy našlo odpoveď na konkrétne paradigmatické problémy (napr. problém prílišnej previazanosti súčiastok softvéru v 80. rokoch, spôsobený vtedy prevládajúcou procedurálnou paradigmou, bol zmiernený vznikom objektovo-orientovaného programovania). Sotva sa však nové postupy v praxi presadili, zložitosť softvéru a projektov sa opäť zvýšila a priniesla nové problémy.

1.1.20
opakovateľnosť, tvorba softvéru

Vysvetlite čo znamená slabá opakovateľnosť v tvorbe softvéru.

Prakticky každý softvér je unikát. Málokedy možno v konkrétnom projekte postupovať rovnako, ako v nejakom predchádzajúcom, pretože každý softvér (softvérový projekt) je od predchádzajúcich odlišný (inak by nebolo vôbec potrebné ho vytvárať).

Samozrejme, vždy sa snažíme osvedčené *postupy* v tvorbe softvéru opakovať, no odlišný obsah projektu nás nakoniec vždy prinúti nad konkrétnymi krokmi premýšľať a stráviť značné množstvo úsilia.

Okrem postupov sa snažíme znovupoužiť *existujúci softvér* alebo jeho *súčiastky*, no vďaka odlišným požiadavkam sú nutné ich modifikácie, ktoré môžu byť tak náročné, že je jednoduchšie softvér vytvoriť znovu „na zelenej lúke“. Možnosti znovupoužitia sa samozrejme zvyšujú, ak sa špecializujeme len na určité typy softvéru. Napríklad firma dodávajúca riešenia pre elektronické obchody môže efektívne využívať raz vytvorené „softvérové polotovary“ a pre konkrétnych zákazníkov ich len vhodne kombinovať a prispôsobovať (takmer ako MOTS). Naopak „všeobecná“ softvérová firma, riešiaca zákazky od klientov s rôznym zameraním bude mať tieto možnosti značne obmedzené. Aj špecializovaná firma sa však často môže dostať do zdĺhavého projektu, ak má jej zákazník netypické požiadavky.

Znovupoužitie postupov či súčiastok komplikuje tiež nedostatok *štandardizácie*. Ak zákazník A požaduje dodanie softvéru s rovnakou špecifikáciou ako náš predchádzajúci zákazník B, stále môžeme potrebovať uskutočniť nový projekt, pretože plat-

forma, ktorú zákazník A historicky používa je iná ako mal zákazník B a pôvodný softvér s ňou nie je kompatibilný.

1.1.21 softvérové inžinierstvo

Porovnajte softvérové inžinierstvo a stavebné inžinierstvo.

Poznámka: táto otázka je v princípe definovaná dosť všeobecne a mohli by sme ju uchopiť z mnohých hľadísk. My sa zameriame na porovnanie skrz problémy tvorby softvéru diskutované v tejto podkapitole.

Z definície majú tieto disciplíny spoločné všetky znaky „všeobecného“ inžinierstva (projektovo orientovaná práca, opieranie sa o osvedčené postupy a ich systematické rozvíjanie, budovanie znalostných báz, systematická práca a pod.).

Obe disciplíny tiež zdieľajú problémovú vlastnosť *zložitosti*. Vysporadúvajú sa s ňou tiež podobne: dekompozíciou, a to jednak horizontálnou (rozdelením predmetu projektu, softvéru resp. stavby), ako aj vertikálnou (modelovaním rôznych dimenzií a s rôznou mierou zanedbania detailov).

Stavebné inžinierstvo tiež čiastočne zdieľa so softvérovým inžinierstvom problém *premenlivosti*, teda vonkajšieho tlaku na zmeny (zo strany prostredia, špeciálne zákazníka). Aj tu častokrát dochádza k zmenám alebo pokusom o zmeny špecifikácie výsledného diela už počas jeho výstavby (hoci ich realizovateľnosť je častokrát s pochopiteľných dôvodov obmedzená). Sú podrobované schvaľovaniu a vyčísľovaniu ich nákladov, podobne ako v softvérovom inžinierstve.

Ešte menej trpí stavebné inžinierstvo *podriadenosť'ou*, teda vnútorným tlakom na zmeny (v dôsledku nechceného odklonu od špecifikácie). Stavebná dokumentácia (návrh stavby) je v čase fyzickej výstavby jednoznačne daná, podriadenosť tak môže teoreticky existovať len vo fáze návrhu stavby. Túto fázu má však spravidla zákazník pod kontrolou a tak nie je veľký priestor pre samovoľné zmeny „zvnútra“ vplyvom prostredia a budúcich „používateľov“ stavby. Naproti tomu v softvérovom inžinierstve je toto riziko prítomné neustále.

Posledným zákonitým problémom softvérového inžinierstva, ktorým je *neuchopiteľ'nosť*, stavebné inžinierstvo netrpí. Stavebné inžinierstvo má k dispozícii techniky a metódy, pomocou ktorých vie jednoznačne zachytiť všetky podstatné vlastnosti vytváraného diela a to ešte pred jeho samotným vyhotovením. To v softvérovom inžinierstve nedokážeme.

1.2 Vlastnosti softvéru

1.2.1
vlastnosti

Vymenujte a definujte vlastnosti softvéru (hľadiská podľa ktorých možno posudzovať kvalitu softvéru).

Poznámka na úvod odpovede: ako sme viackrát spomenuli, kvalitu softvéru determinuje naplnenie požiadaviek na softvér. Pod požiadavkami na softvér si väčšina z nás intuitívne predstaví najmä funkcionálne požiadavky (teda niečo čo softvér „má robiť“). Pri posudzovaní kvalitatívnych vlastností softvéru je však naplnenie funkcionálnych požiadaviek len jedným z mnohých hľadísk. Ostatné hľadiská (vlastnosti), ako možno vidieť v zozname nižšie, patria do priestoru nefunkcionálnych požiadaviek.

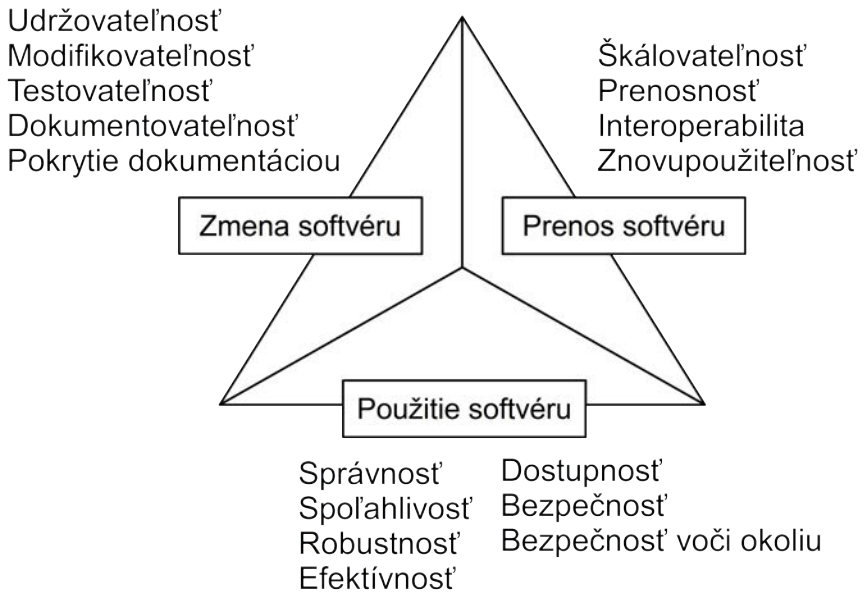
- *Správnosť* (voči špecifikácii) - miera splnenia špecifikácie, najmä *funkcionálnych* požiadaviek.
- *Spoľahlivosť* - frekvencia „nezlyhania“ softvéru (napr. koľkokrát z 1000 načítaní stránky načítanie nezlyhá). Opačnou mierou je *chybovosť* (ako frekvencia „zlyhania“).
- *Robustnosť* – schopnosť softvéru zotaviť sa z chybových stavov resp. neštandardných vstupov.
- *Efektívnosť* – (ne)náročnosť prevádzky softvéru (najmä v zmysle spotreby strojového času a pamäťového priestoru).
- *Dostupnosť* – koľko percent času prevádzky je softvér k dispozícii.
- *Škálovateľnosť* – ako dobre sa softvér dokáže vysporiadať s nárastom záťaže (napríklad v dôsledku zvýšeného počtu používateľov).
- *Použiteľnosť* – ako jednoduché je softvér používať. Ide predovšetkým o vlastnosť používateľského rozhrania.
- *Bezpečnosť voči okoliu* – miera, do akej miery softvér (ne)ohrozuje svojich používateľov. Často sa pletie so *bezpečnosťou*.
- *Bezpečnosť* – miera, do akej je softvér schopný odolávať útokom zvonku. Často sa pletie s *bezpečnosťou voči okoliu*.
- *Prenosnosť* – miera, do akej je možné softvér používať na rôznych platformách a zariadeniach, resp. miera jednoduchosti takýchto inštalácií.
- *Interoperabilita* – miera, do akej softvér podporuje spoluprácu s inými softvérmi.
- *Znovupoužiteľnosť* – miera, do akej možno softvér (alebo jeho časti) použiť ako súčasť iného softvéru.
- *Udržovateľnosť* – do akej miery je jednoduché počas prevádzky softvéru pre softvérových údržbárov softvér udržiavať v chode, opravovať chyby.
- *Modifikovateľnosť* – do akej miery je jednoduché pridávanie novej resp. zmeny existujúcej funkcionality softvéru.

- *Testovateľnosť* – do akej miery je možné (efektívne) verifikovať či softvér zodpovedá špecifikácií. Súvisiacou vlastnosťou softvéru je tiež *pokrytie testami*, ktoré označuje, aká časť štruktúry resp. funkcií softvéru má vytvorené adekvátne automatické testy.
- *Dokumentovateľnosť* – do akej miery je možné k softvéru vytvoriť dokumentáciu (má zmysel rozlišovať najmä pri tzv. „zdedenom“, angl. legacy softvéri). Súvisiacou (a častejšie používanou) vlastnosťou softvéru je *pokrytie dokumentáciou*, teda miera, do akej sú jednotlivé súčiastky zdokumentované.

1.2.2 vlastnosti

Ak máme existujúci softvér, môžeme s ním vo všeobecnosti robiť tri typy aktivít: používať ho, meniť ho a prenášať ho (do nových prostredí, kde má pôsobiť). Ako by ste vlastnosti softvéru (z otázky 52) rozdelili podľa ich relatívnej dôležitosti pre tieto aktivity? Inými slovami, ktoré vlastnosti ovplyvňujú používanie, ktoré prenos a ktoré zmenu softvéru?

Rozdelenie ilustruje obrázok 1.6.



Obr. 1.6: Vlastnosti softvéru rozdelené podľa troch okruhov činností, ktoré s hotovým softvérom typicky chceme robiť.

1.2.3
vlastnosti

Prečo spravidla netrváme na tom, aby bol softvér kvalitný vo všetkých hľadiskách?

Nie vždy sú všetky hľadiská pre zákazníka *dôležité*. Napríklad modifikovateľnosť či udržovateľnosť softvéru nemusí byť dôležitá, ak je určený len na jednorázové použitie alebo sa neplánujú jeho zmeny (napr. promo aplikácie). Interoperabilita a prenosnosť nemusia byť dôležité, ak vieme jednoznačne povedať, že je softvér určený len a len pre konkrétne prostredie (napr. firmu). Efektívnosť nemusí byť dôležitá, ak nás netlačia obmedzenia ohľadom výpočtových kapacít, ktoré máme k dispozícii. Robustnosť zasa nie je podmienkou pre úspešné fungovanie experimentálneho programu na spracovanie dát, ktorý používajú autori sami.

Ak by aj zákazník mal záujem o kvalitu vo všetkých hľadiskách, softvér by bol zrejme *neprimerane drahý*. A vo väčšine prípadov to ani nie je možné, keďže jednotlivé vlastnosti sú navzájom v protiklade (vylepšením jednej sa nám druhá zhorší). Preto si zákazníci zvyčajne veľmi rýchlo vlastnosti zoradia podľa priority a rozhodnú sa investovať do kvality len v určitých ohľadoch. Poradie samozrejme závisí od typu a účelu softvéru.

Napríklad banka vo svojom systéme určite uprednostní bezpečnosť pred interoperabilitou (dve vlastnosti, ktoré sú zvyčajne protikladné). Pri službe posielania okamžitých správ (instant messaging) by sme zrejme naopak uprednostnili interoperabilitu s inými podobnými službami aj za cenu nižších bezpečnostných štandardov.

Ako ďalší príklad môžeme zobrať iné dve protikladné vlastnosti: prenosnosť a efektívnosť. V prípade aplikácie „osobný asistent“, ktorej úlohou je na mobilnom zariadení vypočítavať optimálne trasy presunu cez mesto, preferujeme prenosnosť, čiže dostupnosť na čo najväčšom počte platforiem. Bude to však na úkor efektívnosti: predpokladá sa totiž, že pre minimalizáciu nákladov bude vytvorená jedna univerzálna implementácia potrebných algoritmov, ktorá však tým pádom nebude optimalizovaná pre všetky platformy. Opačné garde by predstavovala situácia: kedy by sme výpočet optimálnych ciest realizovali pre všetkých používateľov centrálne v serverovej časti aplikácie. Tu už by dôraz na efektívnosť implementácie žiadal viac a bol by logický: implementácia by sa optimalizovala presne na platformu servera, bez predpokladu, že by v budúcnosti mala byť prenášaná na iné serverové platformy.

1.2.4
spoľahlivosť, vlastnosti

Ako sa posudzuje spoľahlivosť softvéru?

Ak je spoľahlivosťou (alebo v opačnom garde: chybovosťou) frekvencia „nezlyhania“ softvéru, nedá spravidla sa posúdiť inak, ako štatisticky (posúdením správania sa softvéru v dostatočne veľkom množstve prípadov). To samozrejme nevylučuje, že sa spoľahlivosť nedá predpovedať nejakým iným spôsobom, to však už záleží na špecifikách softvéru (napr. pri solárne nabíjaných senzoroch umiestnených v teréne existuje šanca, že v dôsledku nedostatku energie nedodajú potrebné údaje, pravdepodobnosť čoho môže byť vyjadrená na základe skúseností s počasím v danej lokalite).

Štatistické meranie spoľahlivosti možno uskutočniť dvoma spôsobmi: umelým testom (s pripravenou vzorkou vstupných údajov) alebo sledovaním softvéru pri reálnom nasadení. Umelý test má výhodu v tom, že nemusíme „ísť s kožou na trh“ a spoľahlivosť overíme interne ešte vo fáze vývoja. Tak isto je spravidla rýchlejší a možno ho spojiť aj s tzv. záťažovými testami (ktoré okrem spoľahlivosti zvyknú testovať aj robustnosť). Na druhej strane, pri umelom teste spravidla nedokážeme pripraviť vzorku údajov tak, aby sme pokryli všetky prípady, ktoré v reálnej prevádzke môžu nastať. Spoliehať sa však len na reálnu prevádzku ako zdroj údajov pre vyhodnotenie spoľahlivosti softvéru je veľmi riskantné. Preto sa typicky obidve metódy kombinujú.

1.2.5
použiteľnosť, vlastnosti

Softvér A má 18 funkcií a zaškolenie používateľa trvá 3 dni. Softvér B má 22 funkcií a zaškolenie používateľa trvá 5 dní. Ktorý softvérový výrobok má vyššiu použiteľnosť?

Vyššiu použiteľnosť má softvér A. Rozhodujúci je čas potrebný na zvládnutie softvéru ako celku. Na počte funkcií softvéru nezáleží: jednak je niekedy ťažko povedať čo to je jedna funkcia, jednak sa funkcie môžu líšiť svojou náročnosťou.

Poznámka: otázka má praktický zmysel iba vtedy, ak porovnávame dva softvéry, ktorých biznis ciele sú rovnaké alebo aspoň podobné. Príkladom môže byť porovnanie dvoch softvérov na úpravu fotografií (napr. Gimp a Photoshop): ich ciele sú totožné, podobné sú tiež ich funkcie. Líšia sa však „filozofiou používateľského rozhrania“ ktorá bude značne ovplyvňovať ich použiteľnosť.

1.2.6
interoperabilita, vlastnosti

Softvér A ukladá výstupy do vlastného efektívneho formátu, ktorého tvar nie je známy. Softvér B ukladá výstupy do štandardného formátu (XML). Ktorý softvér má menšiu interoperabilitu?

Menšiu interoperabilitu má softvér A. Vlastný (proprietárny) formát výstupov znamená, že prípadní vývojári tretích strán budú mať viac práce s jeho integráciou s ich softvérmi. Ak by mali výstupy k dispozícii aspoň ako XML súbory, odpadla by im aspoň potreba písať vlastný „parser“.

Efektívnosť formátu výstupov softvéru A (v tomto prípade priestorová) pritom interoperabilitu nijak nezlepšuje. V najlepšom prípade na ňu nebude mať vplyv, v horšom ju

ešte zhorší: priestorová efektívnosť formátu súboru ide totiž často na úkor čitateľnosti človekom, čo sa prenáša aj do zvýšeného ľudského úsilia potrebného na integráciu takéhoto formátu do iných softvérov.

Poznámka: otázka má samozrejme zmysel len za predpokladu, že porovnávame softvéry, ktorých výstupom je zhodný obsah výstupných dát a zaoberáme sa len ich formou.

1.2.7
vlastnosti

Uveďte príklad dvojice vlastností softvéru, ktoré sú si navzájom v protiklade (zvyšovanie jednej pri zachovaní miery úsilia má tendenciu znižovať úroveň druhej).

Medzi príklady vzájomne protikladných znalostí patria (pozor, zoznam určite neobsahuje všetky dvojice):

- Efektívnosť – Prenosnosť: Ak napríklad tvoríme mobilnú aplikáciu, môžeme ju spraviť prenositeľnú na rôzne platformy, ak využijeme niektorú zjednocujúcu virtualizačnú vrstvu. Tá nám umožní aplikáciu napísať raz a potom ju používať na rôznych platformách či už s pomocou interpretácie alebo prekladu kódu do natívnych jazykov platforiem. Oba tieto prístupy zabezpečenia prenosnosti však spôsobia, že aplikácia bude menej výpočtovo efektívna (buď z dôvodu extra réžie spôsobenej interpretáciou, alebo nevhodným prekladom).
- Bezpečnosť – Interoperabilita: Schopnosť softvéru spolupracovať s inými softvérmi si spravidla vyžaduje otvorenie viacerých komunikačných kanálov či rozhraní. Čím viac rozhraní softvér má (resp. čím sú zložitejšie), tým sa zvyšuje riziko, že budú obsahovať bezpečnostné diery.
- Použiteľnosť – Modifikovateľnosť: Vysoká použiteľnosť je spravidla výsledkom prepracovaného návrhu. Aj malé zmeny rozhrania môžu použiteľnosť značne „rozhádzať“. Pre zachovanie použiteľnosti je preto pri zmenách potrebné postupovať obozretne a kroky overovať. To samozrejme znamená zníženú modifikovateľnosť softvéru.
- Správnosť – Udržovateľnosť, modifikovateľnosť: Keďže dosahovanie správnosti softvéru znamená v princípe pridávanie funkcionality, dochádza pri ňom k degradácii štruktúry a prehľadnosti softvéru, čím klesá jeho udržovateľnosť a modifikovateľnosť.

1.2.8
správnosť, vlastnosti

Je správnosť softvérového produktu postačujúcou podmienkou kvalitného softvéru?

Ťažko si predstaviť softvér, ktorý by bol správny (teda spĺňal všetky funkcionálne požiadavky) no v žiadnom ďalšom ohľade (vlastnosti) by nemal potrebnú kvalitu. Ťažko by sme napríklad označili za kvalitný e-shop, ktorý síce realizuje špecifikované scenáre nákupu, ak by dĺžka jeho odozvy bola 5 minút.

1.2.9
použiteľnosť, vlastnosti

Uveďte príklad dvoch zmysluplných, no vzájomne rozporných požiadaviek na softvér z hľadiska použiteľnosti.

Príklady dvojíc rozporných požiadaviek:

- „Pracovník musí vidieť stav svojho účtu a desať posledných transakcii na jednej obrazovke.“ – „Aplikácia musí podporovať rozlíšenia od 720 x 1280 pixelov s uhlopriečkou 4.8 palcov vyššie.“ (na tak malej obrazovke nie je možné zobraziť toľko detailov)
- „Aplikácia má responzívny dizajn a funguje aj pre smartfóny a tablety.“ – „Pre objasnenie jednotlivých funkcií zobrazuje aplikácia pri prechode kurzora ponad tlačidlo (*hover*) textovú pomôcku.“ (na dotykových displejoch neexistuje „prechod kurzora ponad“, ich používatelia by tak nemohli pomôcky vôbec využívať)
- „Rozhranie umožňuje rýchly manažment ponuky produktov.“ – „Produkt sa z ponuky vymaže výberom možnosti vymazať v kontextovom menu. Vymazanie produktu z ponuky vyvolá potvrdzovacie okno.“ (je otázne či 3 kliky sú rýchle vykonávanie údržby zoznamu; k podobným rozporom často dochádza medzi rôznymi úrovňami abstrakcie definovaných požiadaviek)

Poznámka: Takto sformulované príklady sú do očí bijúce no napriek tomu v praxi nastávajú. Problém tkvie zvyčajne v tom, že rozporné dvojice sú „pochované“ v množstve iných požiadaviek, často tvorených rôznymi ľuďmi s rôznymi záujmami.

1.2.10
vlastnosti, bezpečnosť, použiteľnosť, efektívnosť, znovupoužiteľnosť

Predstavte si, že ste začínajúca firma A ktorá sa rozhodla vyvinúť webový e-obchod pre predaj svojich výrobkov. Aké poradie priorít (a prečo) by ste priradili nasledujúcim hľadiskám kvality pri vývoji tohto e-obchodu: bezpečnosť, použiteľnosť, efektívnosť, znovupoužiteľnosť. Ako by sa zmenilo poradie, ak by išlo o firmu B, ktorá chce predávať samotný softvér na e-obchodovanie (iným malým firmám ktoré chcú predávať svoje výrobky)?

Hlavným biznis cieľom firmy A je zarobiť na predaji svojich výrobkov a jej primárnym biznis procesom je teda predaj. Pre fungovanie predaja je kľúčová *použiteľnosť* e-obchodu pre jeho klientov. Ak potenciálni klienti v prvých momentoch návštevy stránky nepochopia, ako majú vyhľadať či zaplatiť produkt, predaj sa neuskutoční. Použiteľnosťou by pravdepodobne zostala najdôležitejšou vlastnosťou aj v prípade firmy B, ktorá by chcela predávať samotný softvér e-obchodu. Pre jej zákazníkov by totiž úspešnosť predaja tiež bola prirodzenou prioritou.

Ako dôležitá je v porovnaní s použiteľnosťou *efektívnosť*? Pre malú, začínajúcu firmu je charakteristické, že má málo zákazníkov. Problémy vyplývajúce z preťaženia jej výpočtových prostriedkov preto nie sú pravdepodobné. Nedostatok efektívnosti začne byť problémom až neskôr, ak sa vôbec firme bude dariť, predávať však aspoň trochu potrebuje od začiatku takže dobrá použiteľnosť je podstatne dôležitejšia.

Čo sa týka *znovupoužiteľnosti*, tú bude mať firma A orientovaná na výrobu zrejme na poslednom mieste. Zatiaľ čo efektívnosť pre ňu v budúcnosti môže predstavovať určitý problém, znovupoužiteľnosťou sa zaoberať vôbec nebude. Jej záujmom je predávať svoje výrobky, nie softvér na predávanie výrobkov. Nechce preto venovať úsilie na udržiavanie „univerzálnosti“ a „peknej“ vnútornej štruktúry svojho softvéru. Naopak, firma B, ktorá by chcela predávať softvér samotný viacerým zákazníkom, by si na znovupoužiteľnosť dala oveľa väčší pozor a je pravdepodobné, že by táto vlastnosť dostala viac pozornosti ako efektívnosť.

Kam v rámci priorít zaradiť *bezpečnosť*? Bez ďalších informácií je táto časť otázky najmenej zrejmá. Nemáme totiž informácie o závažnosti hrozieb (pravdepodobnosti a prípadného dopadu). Útoky na elektronický obchod môžu mať charakter konkurenčného boja (napr. DDOS útoky počas kritických období predaja, napr. Vianoc) alebo „krádeží“ (napr. obchádzaním podsystému platenia). Opatrenia a miera „bezpečnostného“ úsilia by závisela od veľkosti a počtu transakcií. Veľmi by tiež záležalo, aké technológie by boli využité (napr. či by sa platby realizovali cez externú službu ako napr. PayPal). Vo vzťahu k ostatným vlastnostiam v otázke však zrejme môžeme usúdiť nasledovné: použiteľnosť by stále zrejme bola na prvom mieste – bez nej totiž nie je žiaden obchod, bez bezpečnosti bude aspoň nejaký, aj keď ohrozený. Pred znovupoužiteľnosťou a efektívnosťou by však bezpečnosť u firmy A určite dostala prednosť. U firmy B by zrejme prednosť dostala znovupoužiteľnosť, pretože ona determinuje schopnosť predávať softvér čo najviac firmám. Určite by však bezpečnosť dostala viac pozornosti ako efektívnosť.

Poznámka: zrejme sa tiež dá predpovedať, že pri porovnaní absolútneho vynaloženého úsilia na bezpečnosť medzi firmami A a B, by zrejme bezpečnosť získala viac pozornosti vo firme B. Tá by sa totiž snažila čo najviac bezpečnosť podporiť, pretože jej softvér bude nasadený u iných firiem a v hre by bola jednak dobrá povesť firmy B a jednak menej praktické riešenie bezpečnostných problémov, ktoré by boli u klienta a nie vo vlastnej firme.

1.2.11

vlastnosti, udržovateľnosť, bezpečnosť, použiteľnosť, prenosnosť

Predstavte si, že vyvíjate hru pre mobilné zariadenia s modelom jednorazového poplatku za stiahnutie. Aké poradie priorít (a prečo) by ste priradili nasledujúcim hľadiskám kvality pri vývoji tejto hry: udržovateľnosť, bezpečnosť, použiteľnosť, prenosnosť. Zmenilo by sa poradie, ak by sa model platobný model zmenil na mesačné poplatky? Zmenilo by sa poradie, ak by sme pred zakúpením hry (licencie) dali hráčom k dispozícii demo verziu hry?

Úspech počítačových hier možno merať viacerými metrikami orientovaných na hráča, predovšetkým na jeho vôľu hru hrať (opakovane, dlho). Z uvedených vlastností má tieto metriky vplyv v prvom rade *použiteľnosť*. Pokiaľ totiž hra nie je zrozumiteľná, intuitívna resp. nemá vhodný spôsob ako hráča vtiahnuť, nebude úspešná. Použiteľnosť je jediná z uvedených vlastností, ktorá je kritická (bez nej sa hra v nijakom prípade nezaobíde). V prípade, že je súčasťou marketingu zverejnenie demo verzie, stáva sa použiteľnosť ešte dôležitejšou. Bez dema sa teoreticky dajú prilákať zákazníci aj na zlú hru, s demom (ktoré je zlé) to je ťažšie.

Rola *udržovateľnosti* do veľkej miery závisí od modelu predaja hry. Zdá sa, že model jednorázovej platby neprikladá udržovateľnosti veľký význam. Ak prijmeme predpoklad, že sme hru vyvinuli dobre a hráčom sa páči, z biznis pohľadu nemá až taký zmysel ju ďalej zlepšovať, ak sme ju už raz predali. Celkom to však neplatí:

1. Ak chce firma budovať u svojich zákazníkov dôveru pre budúce obchody, musí svoje hry udržiavať aj po vydaní a vychádzať tak v ústrety potrebám a problémom svojich hráčov.

2. Navyše, predpoklad, že sa hru podarí na prvý krát vytvoriť dobre je príliš silný: hry sa málokedy podarí vytvoriť správne a použiteľne na prvý krát a musia podliehať intenzívnemu, iteratívno-inkrementálnemu vývoju. No a pri ňom je zachovanie vysokej miery udržovateľnosti veľmi dôležité.

Ešte viac pozornosti by udržovateľnosť získala, ak by bol model platenia priebežný (teda hráči by platili menšie sumy ako predplatné hry na určité obdobie). Takýto model dnes prináša herným štúdiám viac peňazí, no zároveň ich núti byť oveľa usilovnejšími pri opravách chýb a zlepšeniach hry.

Z pohľadu biznisu sa ako dôležitá javí *prenosnosť*: ovplyvňuje totiž rozsah cieľovej skupiny hry. Čím na viac mobilných platformách hra funguje, tým viac potenciálnych zákazníkov má. Má vysoká prenosnosť v praxi veľký pozitívny dopad na biznis predaja hry? Má, ale zároveň je extrémne drahá (pri hrách si častokrát vyžaduje reimplementáciu celej hry alebo jej častí). Pokiaľ herné štúdiá nedokážu prenosnosť dosiahnuť využitím nejakej existujúcej integrujúcej technológie (v hernom priemysle napr. Unity), väčšinou sa prenosnosťou nezaoberajú vôbec (resp. riešia ju na poslednom mieste).

Význam *bezpečnosti* pri vývoji a prevádzke hier vo všeobecnosti nemožno určiť – závisí od špecifík hry, ale aj modelu predaja. Napríklad v prípade hry pre jedného hráča, ktorá sa jednorazovo predáva prostredníctvom obchodov s aplikáciami, aké poznáme z mobilných platforiem (napr. Google Play, iOS AppStore), nie sú bezpečnostné riziká, ktoré by špecificky mal riešiť vývojár hry veľké. Gro bezpečného predaja a ochrany pred nelegálnym kopírovaním hry už totiž zabezpečuje platforma ako taká. Ani zmena platobného modelu na priebežný, by zrejme neznamenala zvýšenie úsilia na bezpečnosť, pokiaľ by boli platby stále realizované prostredníctvom mobilnej platformy. Zvýšenie bezpečnostných rizík by ale zrejme prišlo, ak by sa platby rozhodlo implementovať štúdio samo (napr. preto, že by mu model ponúkaný mobilnou platformou nevyhovoval). Zvýšenie hrozieb by sme tiež zrejme mohli očakávať u hier pre viac hráčov, špeciálne tzv. mnoho-hráčových online hier (MMOG – massive multiplayer online games).

1.2.12
vlastnosti, použiteľnosť, robustnosť, bezpečnosť voči okoliu, modifikovateľnosť

Predstavte si, že vyvíjate vnútrofiremnú aplikáciu na riadenie procesov výroby automobilov. Aké poradie priorít (a prečo) by ste priradili nasledujúcim hľadiskám kvality pri vývoji tohto softvéru: použiteľnosť, robustnosť, bezpečnosť voči okoliu, modifikovateľnosť.

V predchádzajúcich príkladoch hrala *použiteľnosť* neotrasiteľne najdôležitejšiu úlohu spomedzi uvedených vlastností. Hneď sa preto natíska otázka, či to tak bude aj v tomto prípade. Použiteľnosť je skutočne významná vlastnosť. Je však zakaždým kritická? Určite nie. Veď aj my sami častokrát používame softvér s ktorého použiteľnosťou nie sme úplne spokojní, no nemáme na výber a ak daný softvér svoje funkcionálne požiadavky napĺňa, ako používatelia sa s tým zmierime (príklady sú subjektívne, no je pravdepodobné, že sa čitateľ v niektorých nájde: internet banking aplikácie, akademické informačné systémy, operačné systémy, vývojárske nástroje a pod.).

To čo robí použiteľnosť v niektorých prípadoch kritickou, je stav, kedy použiteľnosť priamo ovplyvňuje biznis a finančné výnosy z predaja a používania softvéru. Platí to napríklad pre prípad e-obchodu či hry, avšak v prípade *vnútropodnikovej* aplikácie na riadenie výroby automobilov to tak nie je. To či sa vnútropodniková aplikácia bude používať nezávisí od toho, či ju používateľ (zamestnanec) na prvý pohľad pochopí a zvládne používať. Jednoducho ju bude musieť použiť a ak to bude vyžadovať týždenné školenie (nič neobvyklé v podobných prípadoch), absolvuje ho. Dá sa preto čakať, že v tomto prípade nebude mať použiteľnosť prioritu.

Softvér vo výrobných podnikoch často fyzicky manipuluje so svojím okolím. V prípade výroby áut si možno predstaviť napr. robotickú výrobnú linku. Fyzická manipulácia s okolím nesie významné riziká. Softvér môže potenciálne spôsobiť škody, ujmu na zdraví či dokonca smrť osôb (stačí si spomenúť na výbuchy rakiet či prípady smrteľného ožiarenia pacientov pri CT vyšetreniach [1]). *Bezpečnosť voči okoliu* by preto v prípade nášho softvéru bola na stole s absolútnou prioritou.

S bezpečnosťou voči okoliu je do značnej miery previazaná aj vlastnosť *robustnosti*, teda schopnosti softvéru sa zotaviť z neočakávaných resp. chybných stavov (často vznikajúcich z chybných podnetov z okolia, napr. od používateľov). Ak chceme softvér vytvoriť bezpečný pre svoje okolie, pravdepodobne sa budeme snažiť identifikovať a dôkladne ošetriť možné chybové stavy, čím do značnej miery zabezpečíme aj jeho robustnosť.

Poslednou vlastnosťou, ktorej dôležitosť treba určiť, je *modifikovateľnosť* takéhoto softvéru. Od nej závisí, či sa v budúcnosti bude dať výroba prispôsobiť novým postupom ľahko, resp. či týmto zmenám náš softvér nebude zbytočne klásť odpor pre svoju zlú štruktúru. Investovať do modifikovateľnosti sa preto môže oplatiť. Mieru dôrazu na modifikovateľnosť ale zrejme bez ďalších informácií nemôžeme správne určiť (závisí to najmä od toho, kam všade má softvér vo výrobe zasahovať). Je napríklad otázne, či sa budúce zmeny výrobných procesov nedajú postihnúť len vhodnou parametrizáciou.

Zrejme však vieme určiť relatívnu pozíciu modifikovateľnosti oproti ostatným skúmaným vlastnostiam. Nebude mať vyššiu prioritu ako bezpečnosť voči okoliu a robustnosť (ktoré sú jasne kritické pre tento typ softvéru), no vďaka svojim jasnejším

ekonomickým implikáciám by dostala prednosť pred použiteľnosťou.

1.2.13
vlastnosti, spoľahlivosť, pokrytie dokumentáciou, znovupoužiteľnosť, prenositeľnosť

Predstavte si, že vytvárate softvérovú knižnicu pre zobrazovanie grafov. Aké poradie priorít (a prečo) by ste priradili nasledujúcim hľadiskám kvality pri vývoji tejto knižnice: spoľahlivosť, pokrytie dokumentáciou, znovupoužiteľnosť, prenositeľnosť.

Ak vytvárame softvérovú knižnicu, znamená to, že chceme *znovupoužívať* vo viac ako jednom projekte. To je jej primárny účel. Znovupoužiteľnoť je teda automaticky prioritou číslo jeden. V rámci nej sa budeme snažiť, aby knižnica svojou funkcionalitou a štruktúrou vyhovovala čo najviac potenciálnym klientom.

Na zoradenie ostatných vlastností môžeme použiť hľadisko „kritickosti“ pre náš biznis. Zanedbanie ktorých vlastností by softvéru, resp. biznisu s ním spojenému uškodili najviac?

Prenositeľnosť ovplyvňuje množstvo potenciálnych zákazníkov podobne ako znovupoužiteľnosť. Tieto dve vlastnosti sa do veľkej miery prekrývajú. Rozdiel je, že prenositeľnosť chápeme ako mieru platformovej nezávislosti, zatiaľ čo znovupoužiteľnosť skôr ako funkčnú charakteristiku.

Zanedbaním prenositeľnosti by sme určite zmenšili skupinu potenciálnych zákazníkov. Zanedbaním *dokumentovania* však riskujeme, že nezískame vôbec nikoho. Bez primeranej dokumentácie je spravidla veľmi náročné porozumieť rozhraniu a spôsobu použitia softvérovej knižnice. Pokrytie dokumentáciou by preto zrejme malo dostať prednosť pred prenositeľnosťou.

Dôležitosť *spoľahlivosti* je pre softvérové knižnice vo všeobecnosti ťažko určiť. Ak však predpokladáme, že ide o vizualizačnú knižnicu, od ktorej pravdepodobne nebudú závisieť obchodné operácie ani iná logika klientskych aplikácií, dostane zrejme spomedzi štyroch spomenutých vlastností najnižšiu prioritu.

1.3 Softvérové procesy a softvérové projekty

1.3.1
softvérový proces

Čo je to softvérový proces? Čo presne znamená *vývoj softvéru*?

Softvérový proces (alebo tiež proces vývoja softvéru) vo všeobecnosti chápeme ako množinu (príbuzných) činností, ktoré vedú k vytvoreniu *softvérového produktu* [10]. Tieto činnosti zahŕňajú nielen samotné programovanie, ale aj podporné aktivity ako napríklad analýzu požiadaviek zákazníka, dokumentovanie, správu verzií vytváraného softvéru alebo jeho testovanie.

Rôzne typy softvérových produktov vyžadujú odlišné softvérové procesy, ktoré vedú k ich vytvoreniu. Neexistuje ideálny softvérový proces, teda presná postupnosť činností, ktorá zaručenie povedie ku kvalitnému softvéru. Softvérové procesy sú komplexné

intelektuálne a kreatívne činnosti, ktoré sa v každej softvérovej spoločnosti realizujú jedinečným spôsobom s ohľadom na vytváraný softvérový produkt a ľudí, ktorí na jeho tvorbe participujú.

Aj keď sa môžu softvérové procesy od seba odlišovať, vždy v sebe nejakým spôsobom zahŕňajú štyri základné vysokoúrovňové činnosti, ktorým predchádza vznik potreby softvéru a jej identifikácia:

1. Identifikácia potreby softvéru – zistenie, uvedomenie si, že potrebujeme softvér (alebo jeho aktualizáciu) na uľahčenie nejakej činnosti.
2. Špecifikáciu softvéru – definovanie toho, čo presne bude softvér robiť.
3. Návrh a implementáciu softvéru – návrh, ako bude softvér vyzerať a ako bude fungovať a následná realizácia, zhmotnenie softvéru.
4. Validáciu softvéru – overenie, či softvér robí to, čo zákazník chcel.
5. Ďalší vývin softvéru – ďalší vývin softvéru po jeho odovzdaní zákazníkovi (napr. vyplývajúci zo zmeny požiadaviek).

Vývoj softvéru (angl. software development) pokrýva množinu *všetkých* činností, ktoré vykonávame pri tvorbe softvéru. Štruktúrovanú postupnosť takýchto činností označujeme ako životný cyklus vývoja softvéru. Činnosti môžeme rozdeliť na hlavné, podporné a organizačné [5] .

S pojmom vývoj softvéru sa v praxi môžeme stretnúť aj v jeho užšom zmysle. Vtedy je to úzka skupina činností softvérového procesu, kde ide o písanie programového zdrojového kódu, „implementáciu“, t.j. činnosť, ktorú vykonávajú programátori („vývojári“). Okrem písania kódu sem patrí aj údržba kódu – dokumentovanie, testovanie, či opravu chýb v kóde.

Poznámka: Niekedy sa s pojmom vývoj softvéru stretneme aj v kontexte evolúcie softvéru (ako preklad angl. *software evolution*). V tomto prípade ide o tú časť softvérového procesu, ktorá začína odovzdaním softvéru zákazníkovi a zahŕňa jeho údržbu a reakcie na zmeny [10]. (V tomto prípade je však správnejšie hovoriť skôr o vývine ako o vývoji softvéru.)

1.3.2 Aké činnosti sa vykonávajú počas života softvéru?

vývoj softvéru, životný cyklus softvéru

Činnosti, ktoré sa vykonávajú počas života softvéru môžu byť v rôznych spoločnostiach vyvíjajúcich softvér rôzne. Súvisia s typom vyvíjaného softvérového produktu, tímom, ktorý na ňom pracuje, prípadne zavedenou *metodológiou vývoja softvéru*. Medzi najčastejšie aplikované činnosti vývoja softvéru patrí (zoznam činností nie je ani z ďaleka vyčerpaný):

- analýza problémovej oblasti (biznis analýza), zahŕňa analýzu požiadaviek zákazníka
- špecifikácia požiadaviek na softvér

- návrh softvéru (architektonický, rozhraní, komponentov),
- implementácia softvéru
- testovanie – verifikácia a validácia softvéru
- nasadenie softvéru do prevádzky
- údržba softvéru,
- používanie softvéru, prevádzka
- vyradenie softvéru.

Dôležité je uvedomiť si, že samotná implementácia softvéru (programovanie) tvorí len jednu z mnohých činností, ktoré sú so softvérom počas celého jeho života späté.

1.3.3 Akou činnosťou začína vývoj softvéru?

vývoj softvéru, štúdia vhodnosti

Vývoj softvéru začína špecifikáciou požiadaviek naň, aby bolo jasné, ČO sa bude vyvíjať. Špecifikácia požiadaviek na softvér sa vytvára popri analýze problémovej oblasti, ktorá sa často označuje ako biznis analýza. V rámci nej sa identifikujú (biznis) problémy a potreby, ktoré bude výsledný softvérový produkt riešiť, resp. napĺňať. Výstupom procesu špecifikácie požiadaviek je zoznam požiadaviek, ktorý odsúhlasil zákazník, pre ktorého softvér vytvárame.

Ak chápeme vývoj softvéru v širšom kontexte softvérových procesov, napr. v kontexte celého *softvérového projektu*, začiatočnou činnosťou môže byť inicializácia projektu, v rámci ktorej sa vykonáva tzv. štúdia vhodnosti s cieľom rozpoznať, či sa softvérovej spoločnosti požadovaný softvérový produkt vôbec oplatí vyvíjať.

1.3.4 Aký je rozdiel medzi modelom vývoja softvéru a metodológiou vývoja softvéru?

vývoj softvéru, metodológia vývoja, model softvérového procesu

Vývoj softvéru môže byť založený na vopred hrubo definovaných činnostiach a vzťahoch medzi nimi, ktoré označujeme ako *model* vývoja softvéru. Poznáme napr. vodopádový model vývoja softvéru, ktorý rozlišuje päť po sebe nasledujúcich činností (analýza požiadaviek na softvér, návrh, implementácia, verifikácia a údržba softvéru), pričom jedna začína vždy po tom, ako sa ukončí tá pred ňou.

Ak vývoj softvéru (softvérový proces, proces vývoja softvéru) realizujeme nejakým postupom osvedčeným praxou, skúsenosťami a pod., hovoríme o *metodológii* vývoja softvéru (softvérového procesu, procesu vývoja softvéru). Napríklad tzv. agilná metodológia vývoja softvéru je súbor takých činností, ktoré vedú k neustálemu dodávaniu postupne vylepšovaného softvéru, ktorý je pri každom dodaní zákazníkovi plne funkčný.

Metodológie vývoja softvéru sú často *založené na* nejakom modeli vývoja softvéru a typicky sú pre nich charakteristické niektoré metódy tvorby softvéru. Preto hovoríme napr. o „metodológii vývoja softvéru podľa vodopádového modelu" alebo o „agilnej

metodológii". Tie zahŕňajú realizáciu činností vývoja softvéru tak, ako sú definované v rámci vodopádového, resp. agileného modelu.

Modely vývoja softvéru sa často označujú aj ako modely životného cyklu vývoja softvéru alebo modely softvérového procesu. Viac o modeloch životného cyklu softvéru sa dočítate v kap. 3.

1.3.5 Aké generické modely vývoja softvéru (metodológie vývoja softvéru) poznáme?

softvérový proces, model softvérového procesu

Najznámejšie generické *modely* softvérového procesu sú vodopádový model a iteratívno-inkrementálny model [10]. Z týchto generických modelov vychádzajú aj ďalšie modely vývoja softvéru – sú od nich odvodené, rozširujú ich, vylepšujú alebo prispôsobujú pre potrebu špecifického typu softvéru.

Vodopádový model je lineárny model založený na princípe, že činnosti pri vývoji softvéru sú vykonávané sekvenčne, ďalšia činnosť začína vždy po skončení tej predošlej. V rámci modelu je identifikovaných päť základných činností: analýza požiadaviek na softvér, návrh, implementácia, verifikácia a údržba softvéru. Z tohto modelu vychádza napr. tzv. V-model.

Iteratívno-inkrementálny model hovorí, že jednotlivé činnosti, ako napovedá názov, sú vykonávané iteratívne (opakovane) a inkrementálne (po prírastkoch). Celkový vývoj softvéru potom prebieha v cykloch. Príkladmi modelov, ktoré vychádzajú z tohto modelu, sú prototypový model, špirálový model, či model agilného vývoja.

Na takýchto modeloch sú potom postavené metodológie vývoja softvéru, často s rovnomenným názvom.

1.3.6 Čo je to softvérový projekt? Ako sa líši od iných projektov?

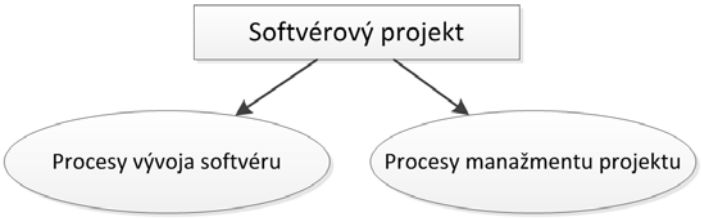
softvérový projekt

Projekt je definovaný ako *dočasné* úsilie s cieľom vytvorenia *jedinečného* výrobku alebo služby [3]. Pozor, dočasné neznamená krátke. Dočasné znamená, že každý projekt má jednoznačný začiatok a koniec. Jedinečný (výrobok alebo služba) je taký, ktorý sa vždy nejakým spôsobom odlišuje od podobných výrobkov alebo služieb. Projekt zahŕňa tvorbu niečoho, čo ešte predtým nebolo, preto výrobok alebo službu treba *postupne rozpracovať*. Príkladom projektu môže byť vývoj prototypu (alebo viacerých prototypov) nového automobilu alebo postavenie sídliska. Projektom nie je zadanie automobilu do sériovej výroby (kde už nevytvoríme jedinečný výsledok).

Softvérový projekt je taký projekt, kde vytváraný výrobok alebo služba je *softvér*. Príkladom softvérového projektu môže byť vytvorenie webovej aplikácie pre podporu doučovania matematiky alebo vývoj databázy existujúcej počítačovej hry. Za softvérový projektu už nepovažujeme vytvorenie 100 000 kusov pultového („krabicového") softvéru.

Procesy (činnosti) v rámci softvérového projektu môžeme rozdeliť na dve skupiny: *procesy vývoja softvéru* a *procesy manažmentu projektu*. Zjednodušene povedané:

všetko, čo v rámci projektu robíme, sú činnosti, vďaka ktorým vzniká *softvér* (analýza, programovanie, písanie dokumentácie, a pod.), alebo činnosti, ktorými riadime/manažujeme projekt (plánovanie, komunikácia, priraďovanie úloh programátorom, a pod.).



Obr. 1.7: Procesy v rámci softvérového projektu

1.3.7

manažment softvérového projektu, proces manažmentu softvérového projektu

Čo znamená manažovať softvérový projekt? Aké činnosti (procesy) manažment softvérového projektu zahŕňa?

Manažment projektu je vo všeobecnosti použitie vhodných znalostí, zručností, prostriedkov a techník na projektové činnosti s cieľom dosiahnutia (alebo prekročenia) potrieb a očakávaní projektu.

Manažment *softvérového* projektu zahŕňa činnosti realizované pre podporu plánovania, implementácie, monitorovania a kontroly vyvíjaného softvéru. Manažment softvérového projektu sa teda odráža do podporných činností pre *procesy vývoja softvéru*.

Príkladmi takýchto činností typicky sú: plánovanie a časový rozvrh činností v rámci projektu, riadenie komunikácie, manažment rizík v projekte, monitorovanie a reportovanie stavu projektu, zabezpečenie kvality softvéru (aj softvérového procesu) alebo manažment ľudských zdrojov.

1.3.8

manažment softvérového projektu, trojuholník manažmentu projektu

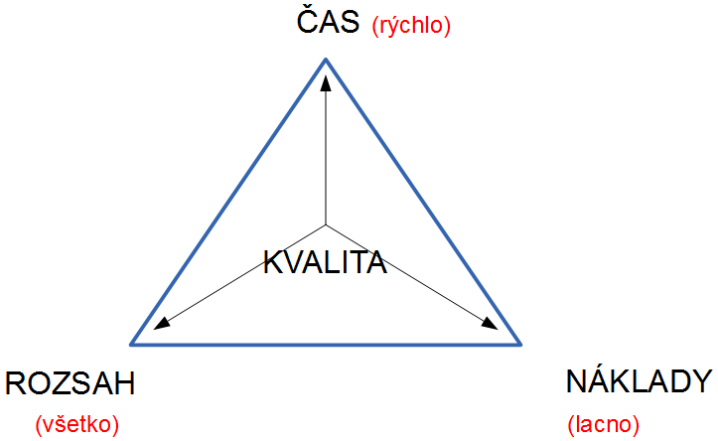
Čo je hlavným cieľom manažmentu projektu?

Hlavným cieľom manažmentu projektu je zabezpečiť, aby bol očakávaný výsledný výrobok alebo služba dodaný včas, v dohodnutom rozsahu a v rámci definovaného rozpočtu. Hovoríme, že výrobok alebo službu chceme dodať v zodpovedajúcej kvalite (ktorá sa netýka len produktu, ale aj procesu). Vzťah medzi týmito tromi veličinami nazývame aj *trojuholník manažmentu projektu* (Obr. 1.8), ktorý predstavuje akýsi model ohraničení projektu. Dodržanie týchto ohraničení v praxi nie je vôbec jednoduché. Dôvody súvisia so špecifickými *vlastnosťami softvéru* (alebo *problémami pri tvorbe softvéru*), ktoré sa odrážajú aj do *problémov manažmentu softvérového projektu* (pozri otázku 75).

Môže sa stať (aj sa často stáva), že pri riešení projektu potrebujeme zmeniť jednu z troch veličín. Reprezentácia pomocou trojuholníka ilustruje, že zmena žiadnej strany trojuholníka sa nezaobíde bez zmeny ostatných dvoch. Napríklad, ak sa zmení rozsah projektu (napr. na podnet zákazníka), nejakým spôsobom to ovplyvní aj čas a náklady

projektu. Aký veľký bude tento vplyv, závisí od projektu. Hlavným cieľom manažmentu projektu je ustriehnuť tieto tri veličiny a zabezpečiť, aby ich zmeny nemali vplyv na celkovú kvalitu (projektu).

Alternatívny spôsob čítania trojuholníka manažmentu projektu je vidieť ohraničenia ako procesné atribúty: dodať výsledok *rýchlo*, *lacno* a *všetko* (Obr. 1.8, červená farba). V takom prípade trojuholníková previazanosť atribútov reprezentuje fakt, že optimalizácia ktorýchkoľvek dvoch atribútov projektu vylučuje optimalizáciu tej tretej. Napríklad, ak chceme dodať všetko a lacno, nebude to rýchlo.



Obr. 1.8: Trojuholník manažmentu projektu

Existujú aj ďalšie modely ohraničení projektu, napr. tzv. diamantový model projektu alebo hviezda manažmentu projektu.

1.3.9 Existujú metodológie aj pre manažment (softvérového) projektu?

manažment softvérového projektu, metodológia manažmentu projektu

Áno, existujú. Podobne ako pri samotnom vývoji softvéru, aj pri jeho manažmente môžeme využiť osvedčené postupy. Medzi najznámejšie referenčné metodológie manažmentu patria metodológia podľa PMI (Project Management Institute), metodológia PRINCE2 (Projects in Controlled Environments, verzia 2) či metodológia podľa IPMA (International Project Management Association).

Viac o metodológiách manažmentu softvérového projektu sa dočítate v pokračovaniach tejto knižky.

1.3.10 Aká je úspešnosť softvérových projektov?

softvérový projekt, manažment softvérového projektu

Úspešnosť softvérových projektov sa pohybuje od 50 do 75 % [2]. Pod úspechom projektu rozumieme splnenie cieľov projektu z pohľadu manažmentu (dodanie včas, v dohodnutom rozsahu a rozpočte, tj. v zodpovedajúcej *kvalite*).

Úspešnosť softvérových projektov sa líši v závislosti od použitej metodológie vývoja softvéru. Niektoré štúdie ukazujú, že vývoj pomocou agilných metodológií vývoja softvéru vedie až k takmer o 50 % vyššej úspešnosti projektu.

Zodpovednosť za zlyhanie vývoja softvéru sa často prisudzuje aj činnostiam manažmentu projektu.

1.3.11 Aké sú najčastejšie príčiny zlyhania vývoja softvéru?

vývoj softvéru, manažment softvérového projektu

Medzi najčastejšie príčiny zlyhania patria [6, 7, 8]:

- nedostatočné zapojenie koncového používateľa (Insufficient end-user involvement)
- slabá komunikácia medzi zákazníkmi, vývojármi, používateľmi a projektovými manažérmi (Poor communication among customers, developers, users and project managers)
- nerealistické nejasne vyjadrené ciele projektu (Unrealistic or unarticulated project goals)
- nepresné odhady potrebných zdrojov(Inaccurate estimates of needed resources)
- zle definované alebo nekompletné systémové požiadavky a špecifikácia(Badly defined or incomplete system requirements and specifications)
- slabé reportovanie stavu projektu(Poor reporting of the project's status)
- slabo manažované riziká(Poorly managed risks)
- použitie nevyzretej technológie(Use of immature technology)
- neschopnosť zvládnuť zložitosť projektu(Inability to handle the project's complexity)
- nedôsledné praktiky vývoja softvéru(Sloppy development practices)
- politiky účastníkov projektu (napr. absencia výkonnej podpory, alebo politiky medzi zákazníkom a koncovými používateľmi(Stakeholder politics (e.g. absence of executive support, or politics between the customer and end-users))
- komerčné tlaky(Commercial pressures)

Uvedené príčiny zlyhania softvéru súvisia predovšetkým s *manažmentom softvérového projektu*.

Príčiny zlyhania vývoja softvéru často súvisia so špecifickými *vlastnosťami softvéru* (alebo *problémami pri tvorbe softvéru*).

1.3.12
komunikácia

Aká je najdôležitejšia súčasť práce projektového manažéra?

Komunikácia.

1.3.13
riziko

Vysvetlite pojem riziko pri tvorbe softvéru.

Riziko je možnosť utrpieť stratu, poškodenie, nevýhodu alebo zničenie. Jednoducho čokoľvek, čo nechceme, aby sa stalo [5].

Riziká môžu ohroziť projekt, produkt alebo organizáciu (biznis). Podľa toho rozlišujeme riziká projektové, produktové a tzv. biznis riziká [5].

Projektové riziká ovplyvňujú (časový) rozvrh projektu alebo zdroje. Príkladom je riziko straty skúseného dizajnéra. Nájdenie adekvátnej náhrady môže trvať dlho a celkový návrh dizajnu softvéru bude trvať dlhšie.

Produktové riziká ovplyvňujú kvalitu vyvíjaného produktu – softvéru. Príkladom môže byť riziko zlyhania hardvéru, na ktorom bude softvér v rámci komplexného riešenia pre zákazníka nasadený. Alebo chyba v knižnici od tretej strany, ktorá bola integrovaná do riešenia. Dôsledkom môže byť zníženie výkonu alebo obmedzenie poskytovanej funkcionality nášho softvéru.

Biznis riziká majú dopad na organizáciu, ktorá vyvíja softvér. Príkladom je riziko vytvorenia podobného softvéru konkurenciou. Predstavenie konkurenčného produktu môže výrazne ovplyvniť očakávané príjmy z predaja softvéru.

Typy rizík sa veľmi často prekrývajú. Napríklad strata kvalitného webového dizajnéra sa môže odraziť aj do nižšej kvality dizajnu a tým pádom celého produktu, alebo môže tiež spôsobiť nezískanie ďalšieho kontraktu, čím predstavuje aj produktové, resp. biznis riziko.

1.3.14
riziko, trojuholník manažmentu projektu

Aké sú základné riziká pri tvorbe softvéru?

Úplne základné riziká pri tvorbe softvéru súvisia s *trojuholníkom projektového manažmentu*. Ide o riziko prekročenia času, riziko prekročenia nákladov, riziko nedodania požadovanej funkcionality. Tieto tri riziká priamo súvisia s nedosiahnutím cieľa projektu, priamo ohrozujú úspešnosť projektu.

Ide o veľmi všeobecné riziká, ktoré sú v skutočnosti dekomponovateľné na konkrétnejšie, často menšie riziká.

1.4 Kvalita softvéru

1.4.1 kvalita

Voči čomu sa posudzuje kvalita softvéru?

Voči potrebám (zákazníkov, používateľov) resp. požiadavkám na softvér.

1.4.2 kvalita

Vysvetlite pojem kvalita softvéru.

Kvalitu softvéru definujeme ako mieru naplnenia potrieb kvôli ktorým softvér vytvárame. Celková kvalita softvéru je zložená z viacerých hľadísk, ktoré spravidla nájdeme medzi požiadavkami na softvér, keďže požiadavky sú v princípe formálnym vyjadrením potrieb.

Napríklad, vytvorením internet banking aplikácie napĺňame potrebu klientov banky uskutočňovať bankové operácie kdekoľvek a bez čakania a potrebu banky ušetriť personálne náklady na pobočkách. Tieto potreby sa následne pretavia do zoznamu požiadaviek (zoznam nie je kompletný, slúži na ilustráciu):

1. Aplikácia umožňuje klientovi previesť určitú sumu peňazí z jeho účtu na iný
2. Bežný beh aplikácie nesmie vyžadovať súčinnosť personálu banky
3. Aplikácia musí fungovať aj na mobilných zariadeniach
4. Aplikácia musí zvládnuť bežať naraz pre 5000 klientov
5. ...

Ako si iste vieme predstaviť, môže existovať aplikácia, ktorá splní všetky tieto požiadavky a tiež aj aplikácia, ktorá splní len niektoré. Môžeme ich teda kvantitatívne porovnať. Zároveň možno vidieť, že niektoré požiadavky, napr. 3 a 4, môžu nadobúdať viacero hodnôt vystihujúcich mieru ich splnenia (je rozdiel medzi tým, ak aplikácia zvláda 2000 alebo 20 prihlásených klientov). Celková kvalita softvéru je teda len agregovaním miery splnenia jednotlivých požiadaviek.

1.4.3 kvalita

Môžeme prehlásiť softvér obsahujúci 1000 chýb za kvalitný? Zdôvodnite.

Intuícia nám hovorí, že nie. *V skutočnosti to však prípustné je*. Kvalita sa totiž meria len a len naplnením potrieb (splnením požiadaviek na softvér). Čiže ak je (v extrémnom prípade) cieľom vytvoriť softvér s 1000 chybami, bude kvalitný práve vtedy, ak tie chyby bude mať. V realistickejšom prípade si možno predstaviť softvér, pri ktorom na tých 1000 chybách nebude záležať, pretože zhodou okolností neovplyvnia plnenie požiadaviek kladených na softvér.

Napr. v aplikácii webového porovnávača cien produktov v elektronických obchodoch je potrebné, aby jej serverová časť pravidelne preliezala dostupné webové obchody a zbierala informácie o cenách produktov. Vďaka heterogénnosti divokého Webu sa jej to v rôznych prípadoch z rôznych dôvodov nedarí v dôsledku neošetrenia množstva nečakaných prípadov štruktúry či obsahu informácií na stránkach obchodov. Pokiaľ

však porovnávač dokáže poskytnúť dostatočne presné porovnania svojim používateľom (pracujúc s informáciami, ktoré má aktuálne dostupné), na týchto chybách nezáleží a softvér tak je kvalitný.

1.4.4 Ktoré tri hľadiská determinujú kvalitu v softvérovom inžinierstve (kvalitu vývoja softvéru)?

kvalita, hľadisko kvality

Aby sme softvérový projekt mohli považovať za úspešný, musí spĺňať 3 kritériá:

1. Splnenie všetkých požiadaviek na výrobok (dosiahnutie maximálnej kvality samotného softvéru).
2. Čas použitý na vývoj nesmie presiahnuť určenú hranicu.
3. Finančné prostriedky minuté na vývoj nesmú presiahnuť určenú hranicu.

Pozor: rozlišujeme kvalitu softvéru samotného a projektu, v ktorom ho vytvárame. Jedno je súčasťou druhého. Prvé kritérium sa týka softvéru samotného, zvyšné dve procesu vývoja.

1.4.5 Diskutujte pohyblivosť a striktnosť hľadísk (čas, náklady a požiadavky) determinujúcich kvalitu v soft. inžinierstve (vývoji softvéru).

kvalita, hľadisko kvality

V prvom rade je dôležité si uvedomiť, že reálne podmienky spravidla nedovoľujú dodržanie všetkých troch hľadísk.

Vo všeobecnosti ďalej platí, že zlepšenie v jednom hľadisku sa odrazí v zhoršení jedného, alebo oboch zvyšných. Napr. ak chceme stihnúť termín dodania softvéru, musíme obetovať viac financií na urýchlenie vývoja, alebo sa vzdať splnenia niektorých požiadaviek. Alebo, ak chceme mať dodržané všetky požiadavky, musíme sa zmieriť s predražením a omeškaním projektu.

Zároveň, v závislosti od podmienok, ale často platí, že zlepšovanie jedného a zhoršovanie ďalších hľadísk kvality nemusí byť vo väčších rozsahoch zmien lineárne závislé. Napr. pridaním ďalších ľudí do (rozbehnutého projektu) a minutím väčšieho množstva prostriedkov, vývoj softvéru vždy neurýchlime. Resp. v snahe projekt naozaj urýchliť nemôžeme donekonečna zväčšovať veľkosť vývojového tímu, pretože sa jeho veľkosť postupne stane nevýhodou.

Iný príklad nelineárnosti v pohyblivosti hľadísk kvality, môžeme vidieť vo vzťahu minutých prostriedkov a času k splneniu požiadaviek. Tu často platí Paretovo pravidlo 80:20, ktoré hovorí, že približne 80% požadovanej kvality softvéru dosiahneme vynaložením 20% úsilia a na zvyšných 20% kvality dosiahneme vynaložením zvyšných 80%.

1.4.6 Ktoré hľadiská kvality vývoja softvéru sú spravidla fixné (fixované) v prípade zákazníckeho softvéru?

kvalita, hľadisko kvality

Neexistuje jednoznačná odpoveď, ale isté trendy prevažujú.

Keďže potreba obstarania nového softvéru býva často naliehavá, býva fixným ohraničením predovšetkým čas (termín dodania hotového softvéru).

Zvyčajne obmedzené sú aj finančné prostriedky, ktoré je zákazník ochotný za softvér zaplatiť.

1.4.7 Ktoré hľadisko kvality vývoja softvéru spravidla závisí (mení sa) od fixácie iných faktorov v prípade zákazníckeho softvéru?

kvalita, hľadisko kvality

Keďže fixovanými faktormi pri zákazníckom softvéri býva čas a cena, prispôsobiť sa im musí splnenie požiadaviek. Toto prispôsobenie sa spravidla realizuje zoradením požiadaviek podľa priority a zrealizovanie takého počtu z nich, na aké stačí čas a finančné prostriedky.

1.4.8 Ktoré hľadiská kvality vývoja softvéru sú spravidla fixné (fixované) v prípade generického softvéru?

kvalita, hľadisko kvality

Generický softvér musí svojou kvalitou potenciálneho zákazníka dostatočne osloviť a zároveň obstáť v prípadnej konkurencii. Preto je primárne fixovaným hľadiskom kvality najmä splnenie požiadaviek (naplnenie potrieb zákazníka) a ustupuje mu hľadisko času, za ktorý sa tieto požiadavky splnia. V konkurenčnom boji medzi softvérovými produktmi rovnakého typu síce môže skoršie uvedenie na trh poskytnúť určitú výhodu, neskôr však preváži produkt poskytujúci lepšie služby (teda lepšie uspokojujúci zákazníkov).

Ako príklad zoberme webové prehliadače, pri ktorých dnes môžeme vidieť zreteľnú konkurenciu. Vidíme, že ak aj dostane prehliadač X výhodu predinštalovania v operačnom systéme Y (ergo, z pohľadu zákazníka sa na trhu objavil ako prvý), je táto výhoda irelevantná (pre istý segment zákazníkov ktorí požadujú kvalitný prehliadač).

Zaujímavou je tiež otázka aspektu ceny a nákladov na vytvorenie softvéru. Generický softvér sa spravidla tvorí s predpokladom, že jeho opakovaný predaj niekoľkonásobne pokryje náklady spojené s jeho vývojom (v opačnom prípade je tvorba generického softvéru značne riziková). To otvára možnosti investovať viac prostriedkov do vývoja a sústrediť sa na dosiahnutie splnenia požiadaviek na softvér, bez ktorého by sa v takom rozsahu softvér nemal šancu predávať.

1.4.9
kvalita, hľadisko kvality

Porovnajte mieru fixácie úrovne splnenia požiadaviek pre generický a pre zákaznícky softvér.

Pri generickom softvéri (aby obstál v konkurencií) je zvyčajne väčší tlak na splnenie všetkých požiadaviek na úkor času a vynaložených prostriedkov.

Naproti tomu u zákazníckeho softvéru, býva skôr tlak na cenu a dodržanie termínov dodania. Býva to podčiarknuté aj faktom, že súťaž medzi dodávateľmi softvéru býva založená skôr na súťaži cien a termínov dodania, než na rozsahu požiadaviek na softvér.

1.4.10
verifikácia

Čo je to verifikácia softvéru?

Zisťovanie, či softvér zodpovedá svojej špecifikácií. Ekvivalentným pojmom je „testovanie“. Verifikácia prebieha uskutočňovaním scenárov použitia softvéru, definovaných v špecifikácií a kontrolou, či sa softvér správa tak, ako špecifikácia očakáva. Scenáre pritom môžu pokrývať funkcionalitu v rôznej miere: od jednoduchých jednotkových (angl. unit) testov, verifikujúcich jednotlivé metódy až po komplexné akceptačné testy, predstavujúce typické scenáre použitia softvéru.

Pozor: ako synonymum verifikácie sa často nesprávne používa termín „overovanie“.

1.4.11
validácia

Čo je to validácia softvéru?

Zisťovanie, či je softvér užitočný (pre svojich používateľov, pre zákazníka). Ako ekvivalentný pojem sa zvykne používať aj „overovanie“. Validácia môže mať veľa podôb, ktoré závisia od typu softvéru. Často zahŕňa tvorbu prototypov, ktoré dostávajú používatelia na posúdenie, či sú pre nich (aspoň potenciálne) užitočné (prirodzene totiž chceme softvér validovať čo najskôr, nie až keď je celý hotový). Validácia softvéru môže dokonca prebehnúť ešte skôr, ako napíšeme prvý riadok kódu: prieskumy trhu či „papierové prototypovanie“ sú vtedy typickými používanými technikami.

Pozor: ako synonymum validácie sa často nesprávne používa termín „testovanie“.

1.4.12
validácia, verifikácia

Aký je rozdiel medzi verifikáciou a validáciou softvéru?

Verifikácia je testovaním, či softvér spĺňa špecifikáciu, zatiaľ čo validácia overuje jeho skutočnú užitočnosť pre zákazníka (používateľa). Verifikácia je často formálnejšia, keďže sa opiera o „tvrdo“ spísanú špecifikáciu resp. akceptačné testy (scenáre použitia softvéru, na ktorých sa tvorca softvéru dohodne so zákazníkom vopred). Dá sa teda ľahšie argumentovať, či softvér je alebo nie je verifikovaný. Naproti tomu, validácia má vágnejšie, často len implicitne existujúce kritériá a ťažšie sa vykonáva – softvér treba často reálne nasadiť a dlhší čas nechať fungovať aby sa naplno ukázalo, či je užitočný.

Zdroje

[1] *10 historical software bugs with extreme consequences*. 2009. URL: http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/.

[2] *2013 IT Project Success Rates Survey Results*. 2013. URL: http://www.ambysoft.com/surveys/success2013.html.

[3] *A Guide To The Project Management Body Of Knowledge (PMBOK Guides)*. Project Management Institute, 2004. ISBN: 193069945X, 9781933890517.

[4] Tarek K. Abdel-Hamid. "Understanding the "90management: A simulation-based case study". In: *Journal of Systems and Software* 8.4 (1988), pp. 319 –330. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/0164-1212(88)90015-5. URL: http://www.sciencedirect.com/science/article/pii/0164121288900155.

[5] Alain Abran et al., eds. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001. ISBN: 0769510000.

[6] Robert N. Charette. "Why software fails". In: (2005). URL: http://spectrum.ieee.org/computing/software/why-software-fails/3.

[7] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005. ISBN: 0596007590.

[8] R. Frese and V. Sauter. "Improving your odds for software project success". In: *Engineering Management Review, IEEE* 42.4 (2014), pp. 125–131. ISSN: 0360-8581. DOI: 10.1109/EMR.2014.6966952.

[9] Patel S Patel D Lawson-Johnson C. *The Effect of Cultural Differences on Software Development*. 2009.

[10] Ian Sommerville. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010. ISBN: 978-0-13-703515-1.

[11] *Systems and software engineering – Vocabulary*. Dec. 2010. DOI: 10.1109/IEEESTD.2010.5733835.

[12] *The NATO Software Engineering Conferences*. 1968. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.

Kapitola 2

Etapy životného cyklu softvéru

2.0.1
etapy životného cyklu

Vymenujte a vysvetlite etapy životného cyklu softvéru (softvérového produktu).

Životný cyklus softvéru tvoria tri skupiny etáp: vývoj, prevádzka a vyradenie.

Etapy vo vývoji softvéru:

- *Analýza* (problémovej oblasti) – v nej sa snažíme čo najviac porozumieť oblasti pre ktorú softvér vytvárame a zistiť čo najpresnejšie, aké sú ciele a potreby zákazníka a používateľov. Mapujú sa biznis procesy a tiež úloha osôb a vecí v nich. Analýze zvykne predchádzať štúdia vhodnosti (či sa oplatí projekt realizovať).
- *Špecifikácia požiadaviek* – v nej na základe poznatkov z analýzy formulujeme požiadavky na softvér: *čo* má robiť a aké má mať vlastnosti. Identifikujú sa predovšetkým scenáre použitia, bežné sú aj návrhy obrazoviek. Dôležitým krokom je tvorba akceptačných testov softvéru, ktorých cieľom je definovať podmienky, za akých sa softvér bude považovať za dodaný. *Poznámka*: špecifikácia sa spolu s analýzou niekedy zvykne uvádzať ako jedna etapa.
- *Návrh* – v tejto etape sa navrhne, ako majú byť sformulované požiadavky na softvér naplnené. Typicky sú súčasťou návrhu rozhodnutia o architektúre softvéru, modeli údajov, technológiách. Navrhujú sa obrazovky a algoritmy. Architektúra sa následne postupne zjemňuje až na úroveň tried a ich rozhraní. Navrhujú a implementujú sa integračné a jednotkové testy.
- *Implementácia* – v tejto etape programujeme definované triedy tak, aby softvér zodpovedal požiadavkám. Dbáme na to, aby spĺňal vopred definované testy. Zároveň vytvárame nové testy všade tam, kde nimi softvér ešte nie je pokrytý. Súčasťou implementácie je vytvorenie fyzického modelu údajov.
- *Nasadenie* – etapa, v ktorej sa hotový softvér (alebo jeho časť) dá do prevádzky u zákazníka. Dá sa sem zahrnúť aj vykonanie akceptačného testovania zákazníkom.

Etapy v prevádzke softvéru:

- *Údržba* – táto etapa začína po nasadení softvéru a zahŕňa zmeny a opravy softvéru počas jeho prevádzky. Zmena či oprava je ako taká aktom vývoja (formálne v nej veľmi rýchlo prejdú etapy analýzy, špecfikácie, návrhu, implementácie a nasadenia), avšak tým, že je uskutočnená za prevázdky, platia pritom väčšinou odlišné pravidlá a obmedzenia (napr. treba dávať pozor na existujúce živé dáta, treba zabezpečovať šírenie zmien softvéru cez aktualizácie).

Etapy vo vyradení softvéru:

- Vyradenie – etapa nastáva potom, čo sa rozhodne, že sa softvér prestane používať. Niekedy je jej súčasťou len toto rozhodnutie a so softvérom už netreba nič robiť. Často však treba nejaké ďalšie akcie: softvér a odinštaluje (napr. kvôli ochrane know-how či licenciám), exportujú sa dáta (väčšinou do nového softvéru).

Poznámka: hranica medzi návrhom a implementáciou nie je ostrá, dá sa napr. diskutovať, či definovanie rozhraní tried či tvorba fyzického modelu patrí tam alebo tam. Až tak na tom však nezáleží, dôležitejšie je si uvedomiť umiestnenie týchto činností voči ostatným činnostiam.

Poznámka: všimnime si pozíciu testovania v celom životnom cykle: je rozptýlené po všetkých etapách. Nejde teda o nejakú samostatnú etapu (často sa však v literatúre takto uvádza).

2.0.2 Ktorá etapa životného cyklu softvéru nás stojí najviac?

Údržba

Spravidla najviac stojí údržba softvéru. Často na ňu pripadne viac ako polovica prostriedkov.

Dôvodom, prečo je údržba tak drahá, je nedokonalosť vývoja v ktorom vznikajú chyby, na ktoré prídeme až po nasadení softvéru. Nemusí ísť pritom ani o zásadné veci, úpravy softvéru sú však v neskorších fázach oveľa náročnejšie (a drahšie).

Pravidlo o najdrahšej údržbe má svoje výnimky. Môže ísť napríklad o kompletne neúspešný softvér, ktorého vývoj sme prerušili ešte pred nasadením. Pozitívnu výnimku predstavuje poctivo, iteratívno-inkrementálne vyvíjaný softvér (pri ňom navyše platí to, že sa celkové náklady majú tendenciu znížiť: dobrý vývoj si síce vyžiada viac prostriedkov, ale náklady na údržbu zníži niekoľkonásobne).

2.0.3 V ktorých etapách vývoja softvéru sa testuje/overuje?

testovanie

Skúmať či robíme softvér správny a správne má zmysel už od samého začiatku jeho tvorby. Má tiež zmysel testovať a overovať neustále – v každom momente sa predsa môžeme dopustiť chyby. Odpoveď na otázku teda je: *vo všetkých*.

Ak by sme to rozmenili na drobné (pohľad cez jednotlivé etapy), etapy implementácie a nasadenia nám na um zídu prvé. Vo etape implementácie píšeme a jednotkové

testy a zároveň vykonávame automatické testy (jednotkové, integračné). V etape nasadenia zasa vykonávame akceptačné testy. No testy vytvárame aj v etape návrhu (integračné). Zároveň vo všetkých etapách posudzujeme vytvorené artefakty manuálne – posúdením (tzv. statické testovanie). Tak ako môžeme pri implementácií sedieť nad zdrojovým kódom a posudzovať, či je dobre napísaný, tak aj v etape špecifikácie môžeme sedieť nad scenármi a posudzovať, či pokrývajú všetky situácie, ktoré môžu nastať. Ďalším príkladom je testovanie v etape analýzy. Tu sa analytici snažia preveriť, či správne pochopili problémovú doménu a ciele zákazníka. Príkladom techník sú kontrolné otázky či prezentovanie papierových prototypov.

2.0.4
dokumentácia

V ktorých etapách životného cyklu softvéru sa tvorí dokumentácia?

Dokumentáciou softvéru možno v princípe nazvať všetky písomné a grafické artefakty vytvorené v softvérovom projekte. Za dokumentáciu možno dokonca považovať aj samotný zdrojový kód softvéru pokiaľ je napísaný „pekne“ a „samovysvetľujúco“.

Každá etapa má takéto výstupy, preto odpoveď na otázku znie: *vo všetkých*.

Rozmenené na drobné (pozor: nejde o všetky možné typy dokumentácie): výsledkom analýzy je opis problémovej oblasti, cieľov a biznis procesov, doménový model. Výsledkom špecifikácie sú zoznamy požiadaviek, modely používateľských scenárov, akceptačné testy. V návrhu vytvárame napríklad opisy architektúry a v implementácii sa snažíme dokumentovať samotný zdrojový kód v referenčných príručkách. V etape nasadenia musíme dokumentovať výsledky akceptačných testov ako aj postupy inštalácie, vytvárame tiež používateľské príručky. Po nasadení (etapa prevádzky a údržby) zasa zaznamenávame výskyt chýb a požiadaviek na zmeny.

2.0.5
návrh

V akej etape životného cyklu vývoja softvéru sa rozhoduje ako bude softvér realizovaný?

Nad tým, ako sa softvér bude realizovať, premýšľame už od prvej chvíle. No až do uzavretia kontraktu so zákazníkom (vytvorenia špecfikácie) je základnou otázkou ohľadom softvéru „čo má softvér robiť?“, nie „ako to má robiť?“. Preto otázka „ako?“ prichádza až v etape *návrhu*.

Návrh zároveň nie je jedinou etapou, pri ktorej sa na otázku „ako?“ odpovedá (rozhodnutia robíme aj počas implementácie a dokonca aj počas údržby pri zapracúvaní zmien). Návrh je však určite etapou, kde sa robia najzávažnejšie rozhodnutia, preto sa otázka „ako bude softvér realizovaný“ spája predovšetkým s touto etapou.

2.0.6
analýza, špecifikácia požiadaviek

V akej etape životného cyklu vývoja softvéru sa rozhoduje, čo sa bude realizovať?

Konečné rozhodnutie, čo sa bude realizovať sa dosiahne na konci etapy špecifikácie požiadaviek. Najtvrdšie toto rozhodnutie reprezentujú akceptačné testy. Toto rozhodnutie však nepríde „odrazu“ ale cesta k nemu trvá od začiatku analýzy. Vtedy sa zisťujú ciele zákazníka a následne biznis procesy, v rámci ktorých by zákazník potreboval softvér uplatniť. Následne sa špecifikujú scenáre použitia (prípady použitia) a z nich sú potom odvodené akceptačné testy.

Odpoveď na otázku „čo sa bude realizovať“ sa preto postupne tvorí počas etáp *analýzy a špecifikácie*.

2.0.7
Údržba

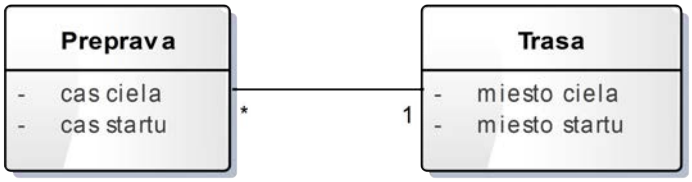
Rádovo koľko krát sa môže počas údržby softvéru predražiť oprava chyby vzniknuvšej v etape analýzy.

Náklady na opravu chyby v analýze sa môžu v rámci údržby pokojne predražiť až o dva rády (v porovnaní s úsilím, ktoré by sme museli vynaložiť pri odstránení chyby ak by sme ju odhalili už v analýze).

Pre ilustráciu zvážme nasledujúci príklad. V softvérovom systéme na organizáciu zdieľania dopravy v osobných autách (carpooling) modelujeme vo fáze analýzy (resp. skorého návrhu) konceptuálny model údajov. Jednou z kľúčových entít je aj *preprava*, ktorá reprezentuje uskutočnenú cestu autom z miesta A do miesta B v konkrétnom čase (obrázok 2.1)

Obr. 2.1: Entita preprava pred úpravou.

Takto definovaná entita vyzerá dobre. Predstavme si ale, že identifikujeme dodatočnú požiadavku a síce na znovupoužitie trás v ďalších prepravách (napríklad preto aby sme mohli využitie aplikácie lepšie analyzovať, alebo aby sme ponúkli vodičovi možnosť znovupoužitia historickej cesty či možnosť definovania viacnásobne sa opakujúcej cesty). Pravdepodobne by sme potrebovali atribúty o mieste štartu a cieľa vytiahnuť do samostatnej entity *trasa*, aby sme ju mohli napojiť na toľko prepráv, koľko potrebujeme (obrázok 2.2).



Obr. 2.2: Entita preprava po úprave.

Aké úsilie by nás zapracovanie tejto dodatočnej požiadavky stálo v rôzne pokročilých fázach projektu?

Ak by sme ju identifikovali už v analýze, pravdepodobne by stačilo upraviť model údajov prípadne doplniť do návrhu obrazoviek možnosť využitia predchádzajúcich trás (úsilie možno na 30 minút).

Ak by sme na ňu prišli počas implementácie, úsilia by bolo viac. Meniť by bolo treba aj fyzický model údajov a naň nadväzujúcu vrstvu aplikačného kódu (zmenil by sa model tried). Nie je vylúčené, že by bolo potrebné meniť aj niektoré algoritmy a taktiež obrazovky by boli už naprogramované a museli by sa prerábať (prinajlepšom dopĺňať). Takéto zmeny by už si mohli vyžiadať viac ako deň práce.

Ak by sme požiadavku chceli zrealizovať počas nasadenia, museli by sme ku všetkému pridať migráciu existujúcich údajov do novej schémy. V tomto prípade by bola ešte pomerne jednoduchá, no aj tak by si práce mohli vyžiadať niekoľko dní práce (bolo by potrebné spätne vytvoriť historické inštancie entity trasa, pravdepodobne by sme chceli duplicitné trasy odstrániť a unikáty správne mapovať, to všetko by sa navyše muselo odohrať nad „živou“ databázou).

Ak teda porovnáme úsilie počas analýzy (30 minút) a údržby (niekoľko dní), rozdiel je takmer dva rády.

2.0.8

etapy životného cyklu, model softvéru

Výstupom ktorej etapy životného cyklu softvéru sú modely vyvíjaného softvéru?

Poznámka: viac o modeloch softvéru, ich definíciách a druhoch, nájdete v otázke 229.

Väčšina modelov softvéru vznikne v etape návrhu, pretože práve vtedy sa snažíme do modelov zachytiť dôležité črty softvéru. Modelujeme tu jednak správanie softvéru (interakcie vo vnútri softvéru, algoritmy, postupy spracovania údajov...) jednak jeho štruktúru (architektúru, vnútornú štruktúru komponentov, modely tried a údajov...).

To že väčšina modelov softvéru vznikne v etape návrhu neznamená, že modely nevznikajú aj v ďalších etapách, *nie však vo všetkých*.

- V rámci *analýzy* štandardne modely budúceho softvéru nevznikajú. Iné modely tu samozrejme vytvárame, ale opisujú existujúce prostredie či biznis, v ktorom má budúci softvér pôsobiť (je naozaj dôležité si uvedomiť, že modely domény či modely procesov nie sú ešte modelmi softvéru).

- V rámci *špecifikácie* vznikajú dôležité funkcionálne modely, teda modely opisujúce, čo má softvér robiť a ako má vyzerať.
- V rámci *implementácie* modely softvéru vznikajú, predovšetkým však vzniká softvér samotný. Je pritom zaujímavou otázkou, či napríklad zdrojový kód je modelom softvéru (pozri otázku 242), každopádne však počas implementácie vznikajú modely ako fyzický model údajov.
- V rámci *nasadzovania* softvéru modely štandardne nevznikajú, resp. záleží na tom, do ktorej etapy spadá vytvorenie modelu rozmiestnenia (teda fyzickej dislokácie komponentov softvéru). Niekedy o ňom je rozhodnuté už v etape implementácie, inokedy počas nasadzovania.
- V rámci *údržby* modely softvéru štandardne nevznikajú, niektoré úpravy softvéru si však môžu vyžadovať zmeny existujúcich modelov.
- V rámci *vyradenia* modely štandardne nevznikajú.

2.0.9
analýza, návrh

Aký je rozdiel medzi analýzou a návrhom softvéru?

V analýze (spolu so špecifikáciou) sa rozhoduje, *čo* sa bude v rámci projektu realizovať. V návrhu sa rozhoduje, *ako* sa to bude realizovať.

2.0.10
Špecifikácia

V ktorej fáze vývoja softvéru sa stanovuje, čo zákazník očakáva od softvéru?

Definitívne rozhodnutia padnú v etape špecifikácie (vytvorením dokumentu špecifikácie), etapa analýzy pre tieto rozhodnutia pripraví pôdu.

2.1 Analýza

2.1.1
Štúdia vhodnosti

Čo je to štúdia vhodnosti a čo je jej výstupom?

Štúdia vhodnosti (angl. feasibility study) je prieskum okolností softvérového projektu *ešte pred jeho začiatkom*. Jej cieľom je zhodnotiť realistickosť projektu (naplnenia požiadaviek) vzhľadom na rozpočet a čas ktorý je k dispozícii.

Pozor: cieľom štúdie vhodnosti *nie je* zistiť, či je daný softvér vhodné vytvoriť (bez ohľadu na rozpočet a čas). Napríklad: účelom štúdie nie je zistiť, či trh potrebuje novú aplikáciu pre GPS navigáciu ani či navrhovaný koncept GPS navigácie, ktorý si zákazník chce dať vytvoriť, bude lepšie napĺňať potreby zákazníkov (aj takéto štúdie sa robia, ale nie sú to štúdie vhodnosti v zmysle v akom ich chápeme my). Cieľom štúdie je zistiť, či danú aplikáciu budeme schopní dodať do 6 mesiacov za 100 tisíc eur.

Výstupom štúdie je *odhad či je reálne vytvoriť softvér s danými zdrojmi*, z čoho v konečnom dôsledku vyplynie rozhodnutie, či sa projekt bude alebo nebude realizovať.

Prirodzene, v čase vykonávania štúdie nie sú známe mnohé detaily projektu, napríklad kompletné požiadavky či technológie (s ktorými sa bude pracovať). Štúdia vhodnosti je preto vždy len hrubým odhadom.

2.1.2 Aké sú najdôležitejšie vlastnosti štúdie vhodnosti?

Štúdia vhodnosti

Štúdia vhodnosti by mala byť rýchla a lacná. V porovnaní s nákladmi a úsilím na samotný projekt by nemalo ísť viac ako o jednotky percent.

Samozrejme, cieľom štúdie je primerane kvalifikovaný odhad reálnosti vytvorenia softvéru, preto ju nemôžme v záujme rýchlosti a ceny úplne zanedbať. Platí však, že s lineárne rastúcim úsilím vloženým do štúdie rastie presnosť jej odhadov len logaritmicky. Oplatí sa teda do nej investovať len malé (ale zároveň aspoň nejaké) množstvo úsilia.

2.1.3 Aké techniky sa využívajú pri štúdii vhodnosti?

Štúdia vhodnosti

Prísť na to, či je projekt realizovateľný v daných časových a nákladových obmedzeniach (rozsahu) je možné uplatnením *podobných techník, aké sa používajú v etape analýzy* softvérového projektu (tu si pripomeňme, že štúdia vhodnosti ako taká nie je súčasťou projektu samotného – predchádza mu). Napríklad:

- *Expertný odhad*. Necháme odborníka na príslušnú technologickú či aplikačnú oblasť odhadnúť rozsah prác.
- *Odhad na základe historických skúseností*. Rozsah projektu odhadneme na základe rozsahov predchádzajúcich podobných projektov, na ktorých sme pracovali.
- *Dekompozícia*. Rozdelením celku na menšie časti zvyšujeme šancu presnejších odhadov a tiež odhalenia potenciálnych problémov.
- *„Architectural spike“*. Pre lepšiu predstavu o budúcom softvéri sa pokúsime odhadnúť podobu jeho architektúry a od nej odvodzovať zložitosť projektu. Ide vlastne o špecifickú techniku dekompozície (výsledkom sú jednotlivé predpokladané moduly softvéru).
- *Prototypovanie na zahodenie*. Pokusne naprogramujeme kritickú časť softvéru. Ide o silný, ale zároveň pomerne drahý prostriedok. Umožňuje rozptýliť neistotu v prípade súčastí softvéru, u ktorých predpokladáme vysokú dôležitosť no zároveň si nie sme istí, koľko úsilia bude potrebného na ich tvorbu. Pomocou prototypu na zahodenie môžeme napríklad overiť, či bude možné znovupoužitie existujúceho algoritmu na naše údaje, alebo bude nutné vytvoriť algoritmus nový.

- *Analýza rizík*. Určia sa riziká spojené s projektom. Rizikom sa myslí nejaká neočakávaná udalosť s negatívnym dopadom na projekt a pravdepodobnosť, že nastane.
- ...

Tieto techniky vhodne kombinujeme. Napríklad dekompozíciu a „architectural spike“ použijeme na identifikáciu čiastkových problémov, ktorým sa následne venujeme separátne: prácnosť niektorých modulov odhadneme na základe historických skúseností či expertným odhadom, pri iných, kde hrozia riziká spojené s technológiami si pomôžeme prototypom na zahodenie.

2.1.4 Štúdia vhodnosti

Ako by ste postupovali pri štúdii vhodnosti na príklade univerzitného softvéru na zdieľanie prepravy v osobných autách (carpooling)? Predpokladajte, že na vytvorenie softvéru by bolo vyhradených 6 mesiacov práce troch vývojárov.

V prvom rade si treba povedať, aký rozsah práce štúdia môže vyžadovať. Štandardne ide o jednotky percent celkového úsilia (skôr bližšie k 0 ako ku 10). Povedzme teda, že chceme štúdii venovať okolo 2% celkového úsilia. K dispozícii máme celkovo 360 človekodní (3 vývojári x 6 mesiacov x 20 pracovných dní). Dve percentá toho sú necelých 11 človekodní, ktoré by sme mali na štúdiu použiť.

Rozsah jedenástich dní by nám mal umožniť, aby sme

1. Prostredníctvom komunikácie so zákazníkom a pár vybranými potenciálnymi používateľmi (vodiči a prepravované osoby) nahrubo a do šírky identifikovali všetky črty softvéru, od ktorých sa očakáva, že budú do softvéru zahrnuté (čiže vymenovali všetky scenáre použitia a funkcie softvéru) a zoradili ich. Softvér na zdieľanie prepravy by určite vyžadoval skupinu funkcionálnych čŕt na samotnú organizáciu dopravy, ale zrejme by obsahoval aj ďalšie, sekundárne funkcie ako podsystém na zaznamenávanie reputácie účastníkov či inteligentný odporúčač spolujazdcov. (2 dni)
2. Prioritné črty rozpracovali podrobnejšie (ako kvázi-špecifikáciu) zostavením scenárov a nakreslením abstraktných (*low-fidelity*) návrhov používateľských rozhraní. Išlo by zrejme o scenáre ako vytvorenie ponuky na cestu, vyhľadanie spolucestujúceho a podobne. (2 dni)
3. Vytvorili hrubý predpokladaný model údajov pre celý softvér. (1 deň)
4. Vytvorili hrubý návrh architektúry softvéru, najmä identifikovali jeho moduly a počet prepojení medzi nimi (napríklad na základe interakcií pri scenároch použitia). Pomôže nám pritom expert na architektúry. (1 deň)
5. Identifikovali externé služby, na ktorých bude softvér závislý a s ktorými by spolupracoval. Identifikujú sa možné alternatívy. Napríklad, softvér na zdieľanie prepravy by takmer určite vyžadoval súčinnosť s nejakým geografickým informačným systémom. (1 deň)

6. Identifikovali technológie, s ktorými softvér budeme vyvíjať. Pritom konzultujeme s technologickými expertmi. Budúci softvér by si určite vyžiadal vytvorenie mobilnej aplikácie pre vodičov a cestujúcich a bola by preto na stole otázka, pre aké platformy by sa vyvíjalo a aké by boli možnosti jej integrácie na ostatné potrebné technológie. (2 dni)

7. Následne na základe artefaktov vytvorených v predchádzajúcich bodoch odhadneme mieru úsilia, ktorú na vytvorenie jednotlivých čŕt softvéru budeme potrebovať. Metód odhadovania je viacero (konzultácie s expertmi, odhadovanie na základe podobnosti s minulými projektami, analytické modelovanie zložitosti...) a nebudeme ich tu rozoberať, kľúčové však je, že prakticky každá sa opiera o dekompozíciu softvéru a tú sme v predchádzajúcich krokoch vykonali. (2 dni)

8. Odhad úsilia použijeme na rozhodnutie, či a v akom rozsahu predpokladáme, že sa nám softvér podarí vytvoriť s danými prostriedkami (človekodňami).

Poznámka: uvedený postup nie je žiadna mantra, je to len jeden zo spôsobov ako pri štúdii vhodnosti postupovať a aj to len pri *podobne veľkom projekte podobného charakteru* (všimnime si, ako veľmi je táto formulácia obmedzujúca). Pri menších projektoch nebude rozsah dostatočný na to, aby bolo možné uskutočniť konzultácie s dostatočným množstvom expertov. Pri väčšom sa zrejme viac úsilia bude venovať tvorbe architektúry aby sa dosiahla primeraná miera dekompozície.

2.1.5 Aké činnosti vykonávame v etape analýzy softvérového projektu?

analýza

Analýza v softvérovom projekte spravidla zahŕňa tieto činnosti:

- Oboznamovanie sa vývojárov (analytikov) s problémovou oblasťou skúmaním jej dokumentov, pozorovaním ľudí, rozhovormi. Často sa pritom tvorí slovník pojmov domény.
- Zisťovanie biznis cieľov (ciele zákazníka, ktoré chce vytvorením softvéru dosiahnuť ale aj potreby používateľov, ktorí so softvérom majú pracovať).
- Identifikovanie biznis procesov (činností, ktoré k naplneniu biznis cieľov smerujú).
- Identifikácia vecí, zdrojov a výsledkov súvisiacich s biznis procesmi.
- Konceptuálne modelovanie domény (model údajov doménových entít).

Uskutočnenie týchto činností napokon vyústi do formulovania požiadaviek, teda špecifikácie.

2.1.6 Čo je to biznis modelovanie?

biznis modelovanie

Biznis modelovanie je opisovanie podstaty fungovania nejakej organizácie (nemusí ísť nutne o komerčnú organizáciu). Jeho výsledkom sú biznis modely. Biznis modely vyjadrujú, aké ciele organizácia má, akým spôsobom pracuje a s akými vecami pracuje. V ekonómii je pojem „biznis model“ chápaný ako spôsob akým organizácia získava prostriedky výmenou za vyprodukované hodnoty. V softvérovom inžinierstve ho chápeme skôr ako opis toho, čo organizácia robí.

2.1.7 Prečo má biznis modelovanie význam pre softvérové inžinierstvo?

biznis modelovanie

V softvérovom inžinierstve potrebujeme modelovať biznis našich zákazníkov, pre ktorých softvér vytárame. Potrebujeme poznať akým spôsobom pracujú (ich používatelia) a akú rolu v ich aktivitách bude softvér vykonávať. Činnosti a postupy, ktoré zákazník vykonáva bývajú základom scenárov samotného softvéru. Veci s ktorými pracuje zasa pravdepodobne budeme reprezentovať údajmi v našom softvéri. Nami vytvorený biznis model môže zákazník skontrolovať a opraviť nedorozumenia.

2.1.8 Čo je to biznis proces?

biznis proces, biznis modelovanie

Biznis proces je postupnosť činností (aktivít), ktorú nejaká organizácia opakovane vykonáva na dosahovanie niektorého zo svojich cieľov. Bez väzby na cieľ biznis proces nemá zmysel. S biznis procesom sú takmer vždy asociované aj osoby, ktoré v ňom hrajú nejakú rolu (*aktéri*) a veci (*artefakty*), ktoré v ňom nejakým spôsobom vystupujú (napr. vstupy, zdroje, medzivýsledky, výsledky).

2.1.9 Aké techniky používame na modelovanie biznis procesov?

biznis proces, biznis modelovanie

Biznis proces je postupnosť činností a niekedy na jeho opis stačí číslovaný zoznam činností plus ich opisy. Prechod cez biznis proces však nemusí byť vždy rovnaký: činnosti môžu byť vzhľadom na okolnosti vykonávané v rôznom poradí, opakovane alebo aj vôbec. Aj preto sú na modelovanie biznis procesov často používané techniky na zápis algoritmov ako aj grafické jazyky ako BPML alebo UML (diagram aktivít).

2.1.10 Ktoré z nasledujúcich názvov aktivít biznis procesov z príkladu na zdieľanie automobilov (carpooling) zrejme nie sú správne? Prečo? (údržba databázy, vytvorenie ponuky na prepravu, mapa trás, registrácia používateľov, preprava, zadanie cieľa cesty, vyhodnotenie prepravy)

biznis proces, biznis modelovanie

Poznámka: vždy záleží na kontexte v ktorom aktivitám dávame názvy. Hoci otázka uvádza o aký projekt ide, ako informácia to nemusí stačiť. Niektoré príklady preto môžu byť diskutabilné. Časté chyby, robené pri identifikácii aktivít v nich však určite možno vidieť.

- Údržba databázy (nesprávne, názov obsahuje pojem softvérového inžinierstva, navyše je vágny; radšej vyjadrime, čo sa v skutočnosti bude robiť, napríklad „hľadanie podobných ciest“ alebo „generovanie odporúčaní spolucestujúcich“)
- Vytvorenie ponuky na prepravu (správne)
- Mapa trás (nesprávne, názov nevyjadruje žiadnu aktivitu, neobsahuje sloveso ani slovesný tvar; správne by mohlo ísť o „prezeranie mapy trás“ alebo „výber trasy z mapy trás“)
- Registrácia používateľov (nesprávne, nejde o veľkú chybu ale názov by mohol byť výstižnejší ak by sme výraz „používateľov“ nahradili výrazom „cestujúcich“)
- Preprava (nesprávne, v danom príklade by zrejme išlo o príliš „veľkú aktivitu“ a zrejme by sme časom zistili, že ide o viac aktivít (možno celý proces), ktoré sa pod ňou skrývajú, napríklad čakanie na všetkých spolucestujúcich, jazda medzi zastávkami, prestávka, ukončenie cesty; okrem toho nie je nutné držať sa jednoslovných názvov aktivít a radšej o nich „prezradiť viac“)
- Zadanie cieľa cesty (nesprávne, tu by zrejme išlo zasa o príliš „drobnú“ aktivitu v rámci niečoho ako „vytýčenie cesty“)
- Vyhodnotenie prepravy (správne)

2.1.11 prieskum problémovej oblasti

Aké techniky prieskumu problémovej oblasti poznáme v rámci analýzy v softvérovom projekte? Stručne ich opíšte.

Tieto techniky využívajú analytici pre zber informácií (teda neuvádzame tu techniky, ktoré využijú následne na zachytenie zistení a modelovanie). Patria sem najmä, ale nielen:

- *Rozhovory* s osobami zainteresovanými v problémovej oblasti. Rozhovor možno viesť s viacerými „typmi“ ľudí (zákazníkom, budúcimi používateľmi, expertmi). Rozhovory rozlišujeme aj podľa štruktúry (od voľného rozhovoru, cez diskusie na vopred určené témy až po formálne rozhovory s tvrdo definovanými otázkami a scenárom). Rozhovory sa tiež líšia cieľom (niekedy sa zameriavajú na zisťovanie očakávaní a požiadaviek na softvér, inokedy je cieľom vôbec pochopiť, ako nejaký proces funguje či aká je štruktúra artefaktov v oblasti).
- *Pozorovanie* aktivít a artefaktov v problémovej oblasti. V tomto prípade analytik pasívne sleduje vybrané aktivity, ktoré zákazník/používateľ vykonáva alebo sú v danej oblasti vykonávané (väčšinou tie, u ktorých sa čaká, že budú pomocou budúceho softvéru podporené). Napríklad, v projekte zameranom na zefektívnenie miestneho úradu, sleduje analytik pohyb a činnosti konkrétnych klientov úradu po budove, skúma tlačivá ktoré do na rôzne oddelenia doručujú, pričom následne robí to isté aj s úradníkmi (sleduje čo robia, kedy to robia).
- *Štúdium dokumentov* o problémovej oblasti. Jednak môže ísť o dokumenty, ktoré sa v problémovej oblasti používajú (napríklad konkrétne tlačivo alebo

metodický pokyn či návod) ale aj o dokumenty ktoré hovoria o danej oblasti (tutoriály, príručky, ...).

- *Štúdium existujúceho softvéru*, fungujúceho v problémovej oblasti.
- ...

2.1.12 prieskum problémovej oblasti

Prečo vykonávame prieskum problémovej oblasti v analýze v softvérovom projekte?

Zmapovanie problémovej oblasti robíme preto, aby sme čo najlepšie spoznali okolnosti, v ktorých bude softvér fungovať a potreby zákazníkov/používateľov, ktorým má softvér slúžiť.

Poznámka: Niekedy by sa mohlo zdať, že ak má zákazník presnú predstavu čo chce (zadanie je pomerne jednoznačné a špecifické), prieskum problémovej oblasti robiť netreba. Dobrý softvérový inžinier však robí prieskum oblasti vždy a väčšinou nad rámec bezprostredne súvisiaci so zadaním. Robí to, aby pochopil čo najviac súvislostí, včas odhalil riziká alebo aby vedel klásť zákazníkom vhodné kontrolné otázky. Napríklad, je užitočné zisťovať, či má vôbec význam softvér vytvoriť alebo či nemá byť vytvorený zásadne inak, ako bolo pôvodne zamýšľané. Preto občas úsilie analytika pripomína činnosť zakladateľa startup firmy.

2.1.13 analýza, zber požiadaviek

Predstavte si, že v rámci analýzy projektu univerzitného zdieľania áut (carpooling) máte za úlohu zmapovať problémovú oblasť. Ako by ste postupovali a aké techniky by ste uplatnili?

Prvým krokom by malo byť zistenie očakávaní a cieľov zákazníka (ktorým je v tomto prípade univerzita) a používateľov, ktorými budú študenti univerzity.

Zistiť ako svoje ciele vníma univerzita. Analytik *rozhovorom* so zástupcom univerzity identifikuje biznis ciele, ktoré univerzita vytvorením softvéru chce naplniť. Rozhovor zrejme nemusí byť zvlášť sofistikovaný, postačia v ňom priame otázky, keďže je predpoklad, že na univerzite o svojich cieľoch majú jasnú predstavu (*pozor*: to nie je to isté ako jasná predstava o tom, ako má softvér vyzerať). V rozhovore sa analytik napríklad môže dozvedieť, že univerzita chce softvér zaviesť z niekoľkých dôvodov: (1) chce aby študenti trávili spolu čo najviac času, (2) chce podnecovať stretávanie sa študentov odlišných ročníkov a odborných zameraní, aby sa podnietila výmena know-how a (3) chce svojim študentom ušetriť finančné prostriedky (v rámci všeobecného záujmu na tom, aby sa študenti mali dobre).

Zistiť očakávania študentov. Zisťovanie očakávaní študentov môže prebiehať, podobne ako pri zástupcovi univerzity, rozhovorom. Analytik však zrejme takéto rozhovory bude viesť odlišným spôsobom, pretože na rozdiel od univerzity, možno predpokladať, že študenti od takéhoto softvéru zatiaľ očakávania nemajú (lebo softvér ešte neexistuje a nebol ani dôvod, prečo by sa nad niečím takým mali zamýšľať). Položiť ako prvú otázku rozhovoru „prečo by ste používali *car-pooling* softvér?“ nepripra-

venému respondentovi nie je dobrý nápad, pretože z neho budú hovoriť skôr dojmy a slabá znalosť problémovej oblasti. V takomto prípade je vhodné viesť rozhovor skôr nepriamo a najskôr zisťovať aké zvyklosti ohľadom cestovania respondent má, aké má pritom problémy, čo by chcel zlepšiť a podobne. Až potom sa respondentovi predostrie myšlienka *car-pooling* princípov (ale stále ešte bez spomínania softvéru). Akým spôsobom si dohadujú spoločné cesty, ak ich robia? Aký je ich okruh spolucestujúcich? Ako si ho vytvárajú? Aké sú pritom najčastejšie problémy? Ako veľmi „spoločnú“ musia mať cestu? Aké zachádzky sú ochotní robiť? Až v úplnom závere rozhovoru sa spýtame, ako si predstavujú ideálny softvér, ktorý by podporoval dohadovanie a riadenie spoločných ciest?

Ďalšie zisťovanie reálií o cestovaní študentov by sa mohlo uskutočniť kvantitatívne, teda s použitím *dotazníka* pre väčšiu skupinu študentov. To môže priniesť vhodné doplnkové informácie, pomocou ktorých budeme napríklad vedieť lepšie zoradiť požiadavky na softvér podľa priority. Napríklad jednoduché dve otázky „kde na Slovensku bývate?“ a „koľko krát do roka cestujete domov?“ pomôžu odhaliť či bude pri vytváraní spoločných jázd problém vôbec nájsť nejaké zhodné trasy cestovania (v prípade že je cestovania málo po rovnakých trasách) alebo je spoločných ciest naopak veľmi veľa. Vedomosť o tomto výrazne ovplyvní, či sa prvé vytvorené funkcie softvéru budú zameriavať na hľadanie kompromisov ohľadom časov a trás ciest (čo je kritické pokiaľ bude spoločných ciest málo a zároveň to môže spotrebovať veľa vývojárskeho úsilia), alebo či bude stačiť len jednoduché rozhranie na hľadanie totožných ciest, pretože takých bude dostatok (čo bude stáť menej vývojárskeho úsilia).

Okrem zberu informácií o cieľoch a očakávaniach, je potrebné *študovať problémovú oblasť ako takú*. Analytik by *štúdiom existujúcich dokumentov*, *existujúceho softvéru* prípadne *konzultáciami s expertmi*, mal nájsť odpovede na otázky a zorientovať sa v okruhoch ako:

Koľko stoja alternatívne spôsoby prepravy? Kde sú k dispozícií a kde nie sú (v ktorých mestách)? Tým pádom, kde je predpoklad, že študenti radšej cestujú autami? Je vhodné vytvoriť si takzvané persóny, čiže typické prototypy používateľov, ktoré by predstavovali rôzne motivovaných potenciálnych používateľov (viac o persónach v otázke 115).

Ako vlastne vyzerá cestná sieť v krajine, je potenciál že budú cesty zhodné?

Odkiaľ sú vlastne všetci študenti univerzity (tieto dáta môže univerzita poskytnúť).

Aké existujúce *car-pooling* aplikácie sú na trhu dnes. Prečo ich študenti nevyužívajú? Aké sú ich silné a slabé stránky (veci na inšpiráciu a veci, ktorých by sme sa aj my mali vyvarovať). Aké sú naše špecifiká, oproti nim. Napríklad: jedným z nich bude napríklad dôvera medzi používateľmi: mnohé aplikácie v tejto oblasti nefungujú pre nedostatok dôvery medzi spolucestujúcimi. Ak sú však všetci spolucestujúci z jednej univerzity, bude pravdepodobné, že základná dôvera bude vyššia.

2.1.14
zber požiadaviek

Zhodnoťte techniku pozorovania ako prostriedku pre získavanie informácií o problémovej oblasti.

Poznámka: pozorovanie je aktivitou analýzy problémovej oblasti a zberu požiadaviek. Analytik v ňom pasívne sleduje deje v problémovej oblasti, napríklad činnosť v nejakej organizácií. Nemusí ísť len o živé pozorovanie; reprodukcia dejov zo záznamu môže byť použitá tiež. Tak ako všetky techniky, má aj pozorovanie svoje výhody a nevýhody (uvedené zoznamy určite neuvádzajú všetky, len tie najdôležitejšie).

Výhody:

- Autenticita a videnie dejov v praxi. Oproti ostatným technikám (rozhovory, štúdium dokumentov, ...) má pozorovanie exkluzívnu výhodu: dokáže ukázať problémovú oblasť takú, aká naozaj je. Ostatné techniky ju analytikovi sprostredkujú len písomne či ústne (a teda potenciálne skreslene).
- Možnosť opakovania. V prípade existencie vhodných záznamov pozorovaných dejov, možno pozorovanie opakovať potrebný počet krát (napríklad keď analyzujeme, ako používatelia pracujú s už existujúcim softvérom, ku ktorému, povedzme, robíme alternatívu alebo ho plánujeme zlepšiť).
- ...

Nevýhody:

- Pozorovanie nemusí ukázať všetky dôležité skutočnosti. Napríklad rozhodovanie ľudí (pravidlá na základe ktorých rozhodnutia robia) pri ňom zostáva skryté. Niekedy si analytik môže dovoliť pozorovanie pozastaviť alebo sa ex-post spýtať, aké boli dôvody rozhodnutí (ale to už je vlastne použitie kombinácie techník, čo sa samozrejme nevylučuje).
- Nemožnosť do pozorovaných dejov zasiahnuť. Pre zachovanie autenticity niekedy nie je možné pýtať si vysvetlenia od protagonistov dejov počas ich priebehu (napríklad ak bankový úradník komunikuje s klientom o delikátnych záležitostiach, ťažko by banka mohla pripustiť, aby ich pritom „vyrušoval“ otázkami zvedavý analytik).
- Môže byť zdĺhavé. Niektoré deje bývajú v problémovej oblasti zriedkavé a čakať na ich výskyt nemusí byť efektívne. V takom prípade je jednoduchšie sa na ich priebeh opýtať v rozhovore.
- ...

Možno tiež povedať, že pozorovanie je (vďaka schopnosti priamo sprostredkovať realitu) aktivitou *predovšetkým* získavania znalostí o problémovej doméne a pri samotnom formulovaní požiadaviek na softvér hrá len pomocnú úlohu. Prebiehajúce deje a v problémovej oblasti totiž len málokedy priamo určujú požiadavky na softvér a domýšľanie si týchto požiadaviek na základe pozorovaní by dokonca mohlo viesť k nesprávnym záverom. To len podporuje tézu, že najlepšie výsledky v analýze dosiahneme vhodnou kombináciou techník.

2.1.15 Vysvetlite pojem „persóna“?

personas, analýza, zber požiadaviek

V softvérovom inžinierstve sú persóny (alebo tiež *personas*) štruktúrované opisy vybraných stereotypov (potenciálnych) používateľov vytváraného softvéru. Používajú sa preto, aby sa vývojári mohli lepšie vcítiť do ľudí, pre ktorých softvér vytvárajú. Persóna sa spravidla nevytvára jedna, ale vytvorí sa ich niekoľko tak, aby primerane pokryli najdôležitejšie cieľové skupiny používateľov. Persóna predstavuje fiktívneho človeka, no jej predobrazom môže byť samozrejme aj existujúci človek, minimálne jeho identita sa však musí zmeniť. Opis persóny by mal byť výstižný a krátky, typicky v rozsahu okolo 100 slov. Často sa využíva štruktúra zložená z troch častí:

Meno, vek a fotografia (reálneho človeka), ktorá persónu vystihuje.

Pozadie persóny. Ide o text, ktorý v krátkosti vystihuje charakteristiky a zvyky osoby. Charakteristiky bývajú jednak všeobecné (ako napríklad rodinné pozadie, práca či záľuby vo voľnom čase) a jednak súvisiace s vytváraným softvérom resp. problémovou oblasťou (napr. pri aplikácii mobilného internet bankingu by mohlo ísť o niečo takéto: „je klientom našej banky a pobočku navštevuje nepravidelne, zato každé dva týždne vyberá z bankomatu hotovosť“). V pozadí persóny sa zvyknú uvádzať aj relevantné problémy, ktoré v súčasnosti osoba má.

Ciele persóny. Ide o text charakterizujúci, aké ciele by osoba chcela naplniť používaním nového softvéru, taktiež ako by si predstavovala jeho používanie. Zvyknú sa tu uvádzať aj problémy, ktoré osoba chce pomocou softvéru riešiť.

2.1.16 Porovnajte pojmy funkcionálne a nefunkcionálne požiadavky na softvér?

požiadavky, funkcionálne požiadavky, nefunkcionálne požiadavky

Funkcionálne požiadavky definujú čo má softvér robiť a aké aktivity má vykonávať (scenáre). Definujú tiež interakciu s používateľmi a inými softvérmi. Nefunkcionálne požiadavky definujú všetky ostatné vlastnosti softvéru, ktoré sú potrebné pre jeho efektívne využívanie (napr. doba odozvy, podpora viacerých platforiem).

Spoločné charakteristiky samozrejme vyplývajú z toho, že ide o požiadavky. Z toho napr. vyplýva, že funkcionálne aj nefunkcionálne požiadavky musia byť zoradené podľa priority, merateľné, konzistentné a pod.

2.1.17 Uveďte príklad nefunkcionálnej požiadavky na softvér.

nefunkcionálne požiadavky

Odpovedí môže byť samozrejme veľa. Dôležité je nezabudnúť do požiadaviek zahrnúť aj overiteľné kritérium jej splnenia (aby boli požiadavky merateľné). V príklade univerzitného softvéru na zdieľanie automobilovej dopravy (carpooling), by sme mohli identifikovať nasledovné nefunkcionálne požiadavky. *Pozor*: nie všetky sú kompletne správne, preto je dobré si všimnúť komentáre:

- *Odozva pri všetkých akciách používateľov softvéru je maximálne 0.5 sekúnd*. Odmerať splnenie tejto požiadavky by sme mohli vykonaním všetkých scenárov, pričom by sme merali čas odozvy softvéru po každej akcií používateľa.

Tento test by sme pravdepodobne vykonali súbežne s testom nasledujúcej požiadavky.

- *Softvér musí naraz vedieť obslúžiť 1000 používateľov*. Meranie splnenia takejto požiadavky vykonávame spravidla záťažovým testom, v ktorom špeciálnym programom simulujeme požiadavky od veľkého množstva používateľov a pozorujeme, či sa softvér pri záťaži správa stále správne.
- *Aplikácia má responzívny dizajn a funguje aj pre smartfóny a tablety*. Takto sformulovaná požiadavka je zrejme príliš prísna. Nie je totiž ďalej špecifikované, aké modely smartfónov a tabletov sa požaduje podporovať. Navyše ak by sme ju chceli merať, potrebovali by sme na testy všetky existujúce modely smartfónov a tabletov. Spravidla sa preto podobné požiadavky formulujú tak, že sa uvedú konkrétne modely zariadení, ktoré sa považujú za referenčné (ak na nich aplikácia funguje, funguje pravdepodobne aj na mnohých ďalších).
- *Softvér je prístupný len pre študentov a zamestnancov univerzity*. Pri tejto požiadavke by sme so sformulovaním kritéria splnenia mali trochu problém: je totiž implicitné. Že softvér nemôže okrem študentov a zamestnancov nikto iný využívať je dané jeho návrhom.

2.1.18 Aký je hlavný rozdiel medzi dokumentmi „špecifikácia požiadaviek (na softvér)“ a „špecifikácia softvéru“.

Špecifikácia, návrh

Špecifikácia softvéru je v podstate len iný názov pre návrh softvéru. Svojou etymológiou však viac zdôrazňuje záväznosť daného artefaktu (výraz „návrh“ má v jazyku predsa len nižšiu silu ako výraz „špecifikácia“).

Rozdiel teda vychádza z rozdielu medzi špecifikáciou a návrhom. Špecifikácia odpovedá na otázku *čo* od softvéru čakáme, návrh na otázku *ako* to softvér dosiahne. Tento rozdiel sa prejaví aj v jazyku dokumentu: požiadavky budú sformulované v jazyku domény, teda tak, aby im rozumel zákazník. Dokument návrhu (špecifikácie softvéru) už je podkladom pre vývojárov a bude obsahovať najmä softvérovo-inžiniersku terminológiu.

Poznámka: oba pojmy môžu veľmi ľahko zameniť. Naše odporúčanie je pojem „špecifikácia softvéru“ radšej nepoužívať a radšej používať „návrh softvéru“.

2.1.19 Čo je to externá požiadavka na softvér.

požiadavky

Ide o požiadavku, ktorá nevychádza od potrieb zákazníka či používateľov, ale je výsledkom vplyvu tretej strany, ktorá na softvéri nemá priamy záujem alebo iných okolností. Príkladom je legislatíva (zákon napríklad môže určovať minimálne bezpečnostné štandardy, ktoré musia aplikácie spracúvajúce osobné údaje spĺňať).

2.1.20
požiadavky

Čo môže vyvolať externú požiadavku na softvér (uveďte príklady).

Príkladom „vyvolávačov“ externých požiadaviek sú (zoznam nie je kompletný):

- *Legislatívne zmeny*. Znenia zákonov môžu definovať obmedzenia týkajúce sa vyvíjaného softvéru, napr. povinnú mieru zabezpečenia pred hrozbami zvonku, povinnosť anonymizácie uchovávaných údajov a podobne.
- *Kultúra, spoločenské normy, etika*. Podobne ako legislatíva, ovplyvňujú podobu softvéru aj ďalšie spoločenské pravidlá v podobe nepísaných noriem. Opäť možno spomenúť tlak na ochranu súkromia používateľov.
- *Nepredvídané* a nekontrolované situácie vo všeobecnosti (napríklad živelné katastrofy, nehody) ktorým sa treba prispôsobiť.

2.1.21
požiadavky, vlastnosti požiadaviek

Uveďte vlastnosti dobrej špecifikácie požiadaviek.

Môžeme hodnotiť každú požiadavku zvlášť, vtedy rozlišujeme vlastnosti:

- *Merateľnosť*. Znamená, že vieme objektívne vyhodnotiť, že bola požiadavka splnená.
- *Správnosť*. Znamená, že požiadavka je v súlade s cieľmi a záujmami zákazníka.
- *Sledovateľnosť*. Znamená, že v priebehu ďalšieho vývoja vieme požiadavku priradiť vývojovým aktivitám, ktoré smerujú k jej naplneniu (a v opačnom garde teda vieme povedať, či niečo nerobíme zbytočne).
- *Modifikovateľnosť*. Znamená, že vieme jednoducho požiadavku zmeniť bez toho, aby negatívne dopady na projekt boli veľké.
- *Jednoznačnosť*. Znamená, že požiadavka je sformulovaná tak, že nedovoľuje viaceré interpretácie (teda sa nemôže stať, že by ju dvaja vývojári inak pochopili).
- *Zrozumiteľnosť*. Znamená, že je požiadavke vývojári jednoducho rozumejú a požiadavka nezvádza vďaka svojej formulácii k nesprávnemu pochopeniu.

Na špecifikáciu požiadaviek sa môžeme pozerať ako celok a vyššie uvedené vlastnosti agregovať. Ako celok má špecifikácia aj ďalšie vlastnosti, ktoré zložením individuálnych vlastností nedostaneme:

- *Zoradenie podľa dôležitosti*. Požiadavky sú jednoznačne zoradené a poradie indikuje ich dôležitosť pre zákazníka.
- *Úplnosť*. Požiadavky spoločne pokrývajú všetky záujmy a ciele, ktoré chce zákazník vytvorením a používaním softvéru dosiahnuť.
- *Konzistentnosť*. Žiadne požiadavky navzájom nie sú v rozpore.

2.1.22

požiadavky, vlastnosti požiadaviek, merateľnosť požiadaviek

Prečo je dôležitá merateľnosť požiadaviek?

Aby sme mohli objektívne vyhodnotiť, či softvér dané požiadavky spĺňa. Bez merateľnosti sa nedá uzavrieť kontrakt so zákazníkom, o ktorý sa dá reálne oprieť. Nie je ani možné rozumne monitorovať priebeh projektu a určovať tempo, s akým napreduje.

2.1.23

požiadavky, vlastnosti požiadaviek, merateľnosť požiadaviek

Aká je základná/nevyhnutná vlastnosť každej požiadavky na softvér? Prečo?

Musí byť *merateľná*. Bez merateľnosti nie je možné určiť, či bola alebo nebola požiadavka splnená.

Ak by sme sa namiesto individuálnej požiadavky pýtali na celú špecifikáciu (zloženú z viacerých požiadaviek) potom je rovnako dôležité *zoradenie požiadaviek podľa priorít*. Požiadaviek je spravidla viac než je na projekt zdrojov a zároveň sú pre zákazníka rôzne dôležité (rôznou mierou prispievajú k naplneniu jeho potrieb). Preto ak chceme od zákazníka aj ďalší projekt, nemali by sme len splniť kontrakt s ním ale ten kontrakt navrhnúť tak, aby napĺňal jeho potreby.

Aj ďalšie vlastnosti (vzťahujúce sa na jednotlivé požiadavky alebo všetky naraz) sú dôležité, ale vyššie uvedené nepredbehnú. Vlastnosti ako správnosť, jednoznačnosť, úplnosť, konzistentnosť či sledovateľnosť bezpochyby napomáhajú dobrému priebehu vytvárania softvéru. Avšak, bez merateľnosti nie je vôbec možné uzavrieť funkčnú zmluvu medzi objednávateľom a zhotoviteľom softvéru. Zoradenie podľa priorít zas maximalizuje potenciálnu pridanú hodnotu, ktorú zákazník pri obmedzených zdrojoch dostane (a obmedzené sú takmer vždy). Nedostatok v ostatných vlastnostiach veci komplikuje, ale v menšej miere.

2.1.24

vlastnosti požiadaviek, sledovateľnosť požiadaviek, úplnosť požiadaviek

Ktorá vlastnosť špecifikácie požiadaviek je dôležitejšia – úplnosť alebo sledovateľnosť? Vysvetlite prečo.

Úplnosť požiadaviek spravidla nevieme zabezpečiť. Väčšinou sa k nej však vieme dostatočne priblížiť takže nám budú požiadavky v tomto ohľade stačiť aj keď sme úplnosť formálne nedosiahli. Naproti tomu sledovateľnosť možno ľahko dosiahnuť, stačí formulovať požiadavky primerane podrobne a tak, aby sa neprekrývali. Sledovateľnosti tiež pomáha, ak požiadavky namapujeme na v analýze identifikované biznis procesy a doménové entity.

Dosiahnuť úplnosť požiadaviek je spravidla snahou každého analytika pri zbere požiadaviek, naproti tomu na sledovateľnosť požiadaviek sa myslí menej. Na úplnosť totiž prirodzene tlačí aj (a najmä) zákazník, zatiaľ čo sledovateľnosť zákazníka vôbec nezaujíma. Sledovateľnosť dokonca nezaujíma ani analytika, pokiaľ nepatrí k manažmentu projektu a nie je zodpovedný za jeho riadenie.

Napriek tomu je sledovateľnosť kľúčová pre správny manažment a teda priebeh projektu. Bez nej sa vôbec nedá monitorovať postup prác na projekte a ak, tak nahrubo

vďaka inej vlastnosti – merateľnosti. Bez sledovateľnosti napríklad programátor nevie povedať, prečo programuje nejakú triedu. Celý projekt tak je o vytváraní softvéru naslepo alebo v hmle s nejasnými obrysmi požiadaviek. To v konečnom dôsledku spôsobí neistotu, zdržania a prácu na zbytočnostiach. Pokiaľ sú požiadavky len neúplné, a v priebehu riešenia projektu sa identifikujú ďalšie (pozor, nie modifikujú existujúce), nie je spravidla problém ich dodatočne zapracovať. A aj keby na to nebol priestor, je pravdepodobné, že nepôjde o prioritné záležitosti.

Môžeme teda povedať, že dôležitejšou vlastnosťou je sledovateľnosť, napriek pozornosti, ktorá sa venuje úplnosti.

Poznámka: V odpovedi je vyslovená téza, že aj keď máme špecifikáciu len „takmer úplnú“ je rozumné predpokladať, že nepôjde o zásadné doplnky, ktoré by softvér kompletne zmenili. Hoci aj to sa môže stať, šlo by o výnimku. Predpoklad sa opiera o to, že zásadné požiadavky sa v analýze zvyknú identifikovať skrátka preto, že sú dôležité a neúplnosť špecifikácie je skôr vecou neidentifikovania „drobností“. Výnimka môže byť spôsobená tým, že nie vždy platí korelácia, že čím zásadnejšia požiadavka, tým väčší má vplyv na vnútornú realizáciu softvéru.

2.1.25
vlastnosti požiadaviek, merateľnosť požiadaviek, konzistentnosť požiadaviek

Ktorá vlastnosť špecifikácie požiadaviek je dôležitejšia – merateľnosť alebo konzistentnosť? Vysvetlite prečo.

Jednoznačne merateľnosť. Bez tejto vlastnosti nemôžeme vyhodnotiť splnenie jednotlivých požiadaviek a projekt sa tak prakticky nedá riešiť už len preto, že nemôžme so zákazníkom uzavrieť rozumný kontrakt. Akákoľvek špecifikácia je bez merateľnosti nepoužiteľná.

Konzistentnosť je dôležitá vlastnosť, ale pri reálnych projektoch s obmedzenými zdrojmi sa spravidla nedá dosiahnuť. To neznamená, že sa neusilujeme o konzistentnosť, ale konzistentnosť nie je nevyhnutná a problémy, ktoré nám jej nedostatok spôsobí spravidla vieme vyriešiť.

2.1.26
vlastnosti požiadaviek, sledovateľnosť požiadaviek, zoradenie požiadaviek podľa dôležitosti

Ktorá vlastnosť špecifikácie požiadaviek je dôležitejšia – sledovateľnosť alebo zoradenie požiadaviek podľa dôležitosti? Vysvetlite prečo.

Zoradenie požiadaviek podľa dôležitosti. To je základná vlastnosť každej špecifikácie požiadaviek. Ak nemáme prioritu, nevieme stanoviť postup v projekte a vystavujeme sa veľkému riziku, že dôležité črty softvéru nevytvoríme.

Sledovateľnosť je tiež dôležitá (jej dôležitosť je opísaná aj v otázke 124). Oproti zoradeniu je však druhoradá.

Predstavme si dilemu výberu v hypotetickom projekte, v ktorom máme 10 požiadaviek ale máme obmedzené zdroje a zrejme nebudeme vedieť naplniť všetky (viac menej typická situácia v softvérovom inžinierstve). Ak by sme si mali vybrať len jednu vlastnosť špecifikácie, vyberáme si medzi dvoma prípadmi:

1. Uprednostníme zoradenie. Budeme vedieť, čo je pre zákazníka najdôležitejšie, ale budeme tápať pri vývoji. Splníme 3 požiadavky z 10, ale budú to tie najdôležitejšie pre zákazníka.

2. Uprednostníme sledovateľnosť. Budeme efektívni pri vývoji, ale nebudeme mať predstavu, čo je pre zákazníka dôležité. Implementujeme 5 náhodne vybraných požiadaviek z 10. S veľmi vysokou pravdepodobnosťou neimplementujeme jednu z dvoch najdôležitejších požiadaviek.

Ktorá možnosť je lepšia? Chýba nám veľa informácií, ale pre jednoduchosť predpokladajme, že miera potrebného úsilia na jednotlivé požiadavky je rovnaká, ale prínos splnenia požiadaviek je veľmi závislý od zákazníkovej priority. Tak to naznačuje aj Paretovo pravidlo 80:20, ktoré v praxi znamená, že prvé dve najprioritnejšie požiadavky z 10 predstavujú 80% hodnoty softvéru. V prípade prvej možnosti ich určite splníme. Dodaná hodnota teda bude aspoň 80%. Pri druhej možnosti je pravdepodobné, že to najdôležitejšie nezrealizujeme a dodaná hodnota bude hlboko pod 80%.

2.1.27

požiadavky, validácia

Kto vykonáva validáciu požiadaviek na softvér?

Zákazník. On je ten, koho predstavy by požiadavky mali odrážať.

2.1.28

požiadavky, vlastnosti požiadaviek, konzistentnosť požiadaviek

Ako zabezpečujeme konzistentnosť požiadaviek na softvér?

Proces by sme mohli rozdeliť do dvoch fáz, *kontroly* a *odstraňovania* nekonzistentností.

Konzistentnosť kontrolujeme posudzovaním dvojíc alebo menších skupín požiadaviek a zisťovaním, či nie sú v rozpore. Môžeme takto posúdiť požiadavky po ich zozbieraní naraz, alebo posudzujeme priebežne, teda každú novú požiadavku skontrolujeme voči existujúcim.

Keďže pri väčších projektoch môže byť požiadaviek dosť veľa a počet kontrol konzistentnosti by bol voči počtu požiadaviek kvadratický, posudzujú sa iba požiadavky, ktoré nejakým spôsobom súvisia. Môžu sa týkať jedného okruhu funkcionality (napríklad sa týkajú jednej obrazovky v používateľskom rozhraní) alebo sa očakáva, že budú realizované spoločnými súčiastkami a podobne.

Je treba povedať, že identifikácia nekonzistentností vždy vyžaduje istú mieru skúseností analytika, ktorý ju vykonáva. Skúsený analytik problémy s nekonzistentnosťou zažil a typické, problémy generujúce situácie tak dokáže odhaliť skôr. Hĺbka technologických znalostí pomáha tiež. Čím hlbšie si vie analytik premietnuť požiadavky do podoby budúceho softvéru, tým ľahšie si všimne potenciálne problémy.

Odstraňovanie nekonzistentností. Ak rozpor medzi požiadavkami identifikujeme, diskutujeme so zákazníkom, skúmame alternatívy a vyberieme takú, ktorá rozpor neobsahuje. Zákazník je vedený potrebami a požiadavky sú len ich priemetom, preto je

častokrát možné, že k potrebe existuje alternatívny spôsob ako ju naplniť. V opačnom prípade musí zákazník v niektorej z požiadaviek ustúpiť (podľa svojich priorít).

Príklad: Zákazník môže požadovať, aby sa objednávkový formulár jeho e-obchodu zmestil na jednu obrazovku a aby obchodné podmienky (pomerne dlhý text) boli v plnom znení zobrazené zákazníkovi v čase, keď ich potvrdzuje. Zároveň požaduje aby sa podmienky potvrdzovali pri objednávke. Vyššie uvedené požiadavky sú vzájomne v rozpore, pretože obchodné podmienky sa na jednu obrazovku nezmestia ako také, nieto ešte s ďalšími súčasťami objednávkového formulára. Zákazník sa preto musí rozhodnúť čomu dá prednosť. Objednávku môže mať buď viackrokovú, alebo podmienky na formulári nezobrazí celé, iba ponúkne náhľad cez rolovacie okno prípadne iba cez odkaz. Alternatívou pre neho tiež môže byť umiestnenie podmienok a ich potvrdenia úplne mimo objednávky, napríklad na vstup do e-obchodu.

2.1.29

požiadavky, vlastnosti požiadaviek, jednoznačnosť požiadaviek

Ako zabezpečujeme jednoznačnosť požiadaviek na softvér?

Jednoznačnosť požiadaviek dosahujeme ich detailným opisom. Čím viac detailov požiadavky obsahujú, tým menšia je pravdepodobnosť, že budú viacznačné, pretože viac detailov vylučuje možnosti alternatívnych interpretácií (samozrejme za predpokladu, že udržujeme aj inú dôležitú vlastnosť požiadaviek - zrozumiteľnosť).

Analytik tiež môže využiť techniku „diablovho advokáta“ a úmyselne si začať predstavovať možné dezinterpretácie toho, čo práve vytvoril a následne opravovať formulácie tak, aby už možné neboli.

Jednoznačnosť sa často kryštalizuje postupne, najmä po tom, čo s požiadavkou prichádza do styku viacero účastníkov projektu. Napríklad slovný opis scenáru použitia, ktorý vytvoril a videl analytik, má menšiu šancu byť jednoznačný ako scenár, ktorý prečítal a na jeho základe vytvoril návrh rozhrania návrhár a grafik. V druhom prípade nastáva šanca, že analytikova predstava bude od návrhárovej odlišná, čo nevyhnutne povedie k diskusii o konkrétnych záležitostiach a počas tejto diskusie sa časť nejednoznačností odstráni. Podobný efekt nastáva aj v prípade posúdenia požiadaviek zákazníkom. Z toho vidno, že veľkú časť odstraňovania nejednoznačností zabezpečuje iteratívny prístup k tvorbe špecifikácie, v ktorom prebieha viacnásobná kontrola zúčastnenými vývojármi a zákazníkom.

2.1.30

požiadavky, vlastnosti požiadaviek

Uveďte problémy vznikajúce pri špecifikácii požiadaviek na softvér.

Predovšetkým rozlišujme *problémy procesu tvorby požiadaviek* od *problémov dokumentu špecifikácie*, ktorý je produktom tohto procesu.

Medzi typické problémy procesu tvorby požiadaviek patrí:

- Celková malá pozornosť a energia, ktorá sa tejto etape vývoja venuje.
- *Nedostatočné zapojenie používateľov do tvorby požiadaviek*. Jedna vec je diskutovať so zákazníkom a vypočuť si jeho potreby. Softvér však často budú používať iní ľudia, ktorí zákazníkmi nie sú a aj ich potreby treba vypočuť. Hľa-

danie týchto ľudí môže byť náročné (niekedy sú vopred neznámi) a aj vedenie rozhovorov s nimi (nemajú apriórnu motiváciu sa s nami vôbec rozprávať).

- *Spoliehanie sa na iba na jednoduché techniky akými je napríklad rozhovor*. V rozhovore síce z úst zákazníka či používateľov požiadavky zaznievajú („to je predsta to čo chceme, nie?“), no aj tieto treba vedieť nejakým spôsobom potvrdiť, pretože zákazník ani používateľ často sám poriadne nevie, čo by potreboval a treba počítať s tým, že sám nemá dostatočne do hĺbky svoj zámer premyslený. Preto sú, techniky ako pozorovanie či štúdium existujúcich dokumentov nevyhnutné pre odstraňovanie rizík (hoci sú časovo náročnejšie).

- *Chyby vo vedení rozhovorov*. Výsledok rozhovoru, v ktorom sa snažíme zisťovať požiadavky môže byť negatívne ovplyvnený javmi ako napr. sugestívne otázky (ktorými nechtiac ovplyvňujeme úsudok opýtaných), priame otázky položené v nevhodných momentoch (opýtaní často vedome alebo nevedome klamú) či sociálny tlak (budúci používateľ a súčasne zamestnanec firmy, ktorá si softvér objednáva, nemusí odpovedať pravdivo na všetky otázky ak v miestnosti počas rozhovoru s ním sedí jeho šéf).

- *Používanie nesprávneho žargónu*. Chybou, ktorá trochu súvisí s predchádzajúcim bodom, je ak analytik (ako IT profesionál) neprispôsobí svoj slovník k biznis slovníku svojho zákazníka. Na základe tohto vznikajú nedorozumenia a taktiež klesá chuť oboch strán komunikovať, čo je vždy na škodu projektu (celého, nielen zberu požiadaviek), keďže od nedostatku komunikácie sa odvíja množstvo problémov.

- *Absencia manažmentu očakávaní*. Veľmi často podceňovaný aspekt jednania so zákazníkom. Zákazník má od prvej chvíle, kedy identifikuje potrebu nejaký softvér vytvoriť, nejakú predstavu, ako má softvér vyzerať a čo všetko mu softvér prinesie. Tieto očakávania môžu byť príliš veľké. Až tak, že ich ani najlepšia softvérová firma za dané peniaze nebude vedieť uspokojiť. Očakávania sa však môžu meniť a je jednou z prvých úloh analytika aby po tom, čo zistí, že očakávania zákazníka sú objektívne neprimerané, tieto čo najviac skorigoval a ak to nebude možné, z projektu vycúval. Pokiaľ k takejto korekcii nedôjde, môže ľahko nastať veľmi nepríjemný scenár, kedy síce vznikne realistická špecifikácia, v ktorej nebude vidieť problém ani zákazník ani zhotoviteľ, no v skutočnosti bude existovať diskrepancia v očakávaniach a veľké rozčarovanie na konci projektu. V takom prípade je častým javom to, že firma už druhý krát od rovnakého zákazníka zákazku nedostane.

- ...

Problémy dokumentu špecifikácie (zjednodušene: zoznamu jednotlivých požiadaviek) odrážajú problémy v procese tvorby požiadaviek a ide v podstate o opačné vlastnosti, aké by mala mať špecifikácia podľa otázky 121. Predovšetkým:

- nejasná a neúplná formulácia požiadaviek,

- nejednoznačnosť spojená s častou špecifikáciou požiadaviek v prirodzenom jazyku,

- nestálosť a protirečivosť požiadaviek,
- prirodzená neúplnosť a nepresnosť pri špecifikácii veľkých softvérových systémov
- ...

2.1.31
požiadavky, akceptačné testovanie

Aká činnosť je, z hľadiska testovania, neoddeliteľnou časťou tvorby požiadaviek na softvér?

Vytvorenie a naplánovanie akceptačných testov.

Akceptačné testy zo svojej podstaty musia vzniknúť pred uzavretím kontraktu so zákazníkom, pretože definujú za akých okolností zákazník uzná, že softvér bol vytvorený a mal by ho prebrať. Akceptačné testy potvrdzujú merateľnosť požiadaviek (bez nej by sa ani nedali vytvoriť). Mnohé kontrakty sa opierajú práve o akceptačné testy ako o „tvrdý“ dokument špecifikácie a „klasickú“ špecifikáciu (zoznam funkcionálnych a nefunkcionálnych požiadaviek) vnímajú len ako doplnkový dokument.

2.1.32
funkcionálny test

Vysvetlite prečo je dobré vytvárať funkcionálne testovacie scenáre počas špecifikácie. Má to aj nejaké nevýhody?

Hlavná výhoda a dôvod prečo vytvárame funkcionálne testy ešte pred návrhom a implementáciou je, aby sme odhalili problémy v požiadavkách. Vytváranie testov, napríklad konkrétnych vstupov a výstupov nás núti nad požiadavkami podrobnejšie premýšľať. Nevýhodou je, že v prípade zmien požiadaviek treba modifikovať aj testovacie scenáre.

Poznámka 1: funkcionálne testy sú také, pri ktorých testujeme funkcionalitu softvéru alebo nejakej jeho súčiastky zadávaním vstupov a porovnávaním reálnych výstupov s očakávanými (tzv. čierna skrinka). Spravidla sú automatické (viac o typoch testovania v otázke 171).

Poznámka 2: z definície v poznámke 1 nepriamo vyplýva, že *nie všetky* funkcionálne testy, ktoré v softvérovom projekte možno vytvoriť a použiť, možno vytvoriť počas špecifikácie: zahŕňajú totiž aj testy komponentov softvéru, ktoré v čase špecifikácie ešte nemôžeme poznať.

2.2 Návrh

2.2.1 Aké činnosti vykonávame vo etape návrhu v softvérovom projekte?

návrh

Návrhom sú tie činnosti, v ktorých vznikajú rozhodnutia ako má byť softvér zrealizovaný, no zároveň ešte nejde o realizáciu samotnú (ktorá už spadá pod implementáciu). Dôležitými činnosťami etapy návrhu sú najmä:

- Navrhovanie architektúry (identifikácia komponentov, rozhraní a prepojení medzi nimi)
- Rozhodnutia o použití technológií pre implementáciu (pokiaľ neboli dané už v špecifikácii)
- Tvorba integračných testov (ako zodpovedajúci typ testovania komponentov v architektúre)
- Práce na modeloch údajov (prechod od konceptuálnych modelov založených predovšetkým na biznis analýze, k logickým modelom údajov, ktoré sú viac technické)
- Identifikácia a návrh budúcich programových štruktúr (tried, ich vzťahov, rozhraní, dátových typov) a algoritmov (táto činnosť už istým spôsobom hraničí s implementáciou)
- Tvorba jednotkových testov (taktiež na hranici s etapou implementácie)
- Navrhovanie používateľských rozhraní (resp. dotváranie predstáv, ktoré už sú dané špecifikáciou)

Všimnime si, že hranica etapy návrhu s etapou implementácie nie je vždy zreteľná. To je dôsledkom aj toho, že návrh softvéru je spravidla postupným zjemňovaním a konkretizovaním modelov softvéru (napr. najskôr robíme „veľké“ rozhodnutia, napr. o architektúre a až neskôr sa dostávame k detailom, ako napríklad rozhodnutie o použití dátového typu pre istý atribút, ktoré už spravidla nemá význam zachytávať len dokumentačne, ale rovno sa zapíše na úrovni zdrojového kódu).

2.2.2 Čo je výstupom etapy návrhu v softvérovom projekte (aké artefakty)?

návrh

V prvom rade ide o dokumentáciu:

- Architektúra softvéru (identifikované komponenty, rozhrania, často spolu s platformami, na ktorých budú postavené)
- Logické modely údajov
- Modely tried, rozhrania tried
- Návrhy používateľských rozhraní (*wireframes,* môžu už byť veľmi presné)
- Algoritmy a postupy, ktoré bude softvér realizovať

Výstupom sú však aj prvé programové štruktúry:

- Integračné testy, jednotkové testy

- Základ programových reprezentácií tried

2.2.3 Čo musí obsahovať každý návrh softvéru (aké dva kľúčové výstupy)?

návrh

Každý návrh softvéru musí obsahovať architektonický návrh (identifikované komponenty, rozhrania a vzťahy medzi nimi) a návrh jednotlivých komponentov (vnútornú štruktúru).

Prečo práve tieto časti? Pretože tvoria spojivo celého softvéru. Len veľmi ťažko sa dá na softvéri ďalej pracovať, ak tieto artefakty nie sú vytvorené.

2.2.4 Čo je to architektúra softvéru? Čo ju tvorí?

[1 - Zapamätať si]

Architektúra je základnou organizáciou softvéru. Tvoria ju komponenty (súčiastky), vzťahy medzi nimi a vzťahy k prostrediu, v ktorom má softvér existovať. Tieto vzťahy sú najčastejšie definované programovými rozhraniami (niektoré komponenty ich vystavujú, iné sa k nim pripájajú a využívajú ich). Súčasťou architektúry sú aj princípy jej vytvárania a vývoja. To znamená, že pri vytváraní architektúry vedome využívame nejakú (pokiaľ možno štandardnú a overenú) stratégiu, napríklad uplatňujeme nejaký architektonický štýl (napr. vrstvy či klient-server).

Častá chyba: často krát sa za architektúru vydáva už štruktúra, ktorá identifikuje komponenty a „kreslí“ medzi nimi nepomenované vzťahy (napríklad ako na obrázku 2.3). Také niečo je veľmi nepresné a neprípustné. Architektúra nie je kompletná, pokiaľ okrem názvov komponentov neexistuje aj ich vnútorný opis a najmä rozhrania (vo forme jednotlivých operácií, ktoré komponenty poskytujú). Taktiež musí byť jasné, ktorý komponent sa pripája na ktoré rozhranie a ktorý iný komponent toto rozhranie implementuje. Iba takýto opis totiž skutočne definuje, aké sú „kompetencie“ jednotlivých modulov.

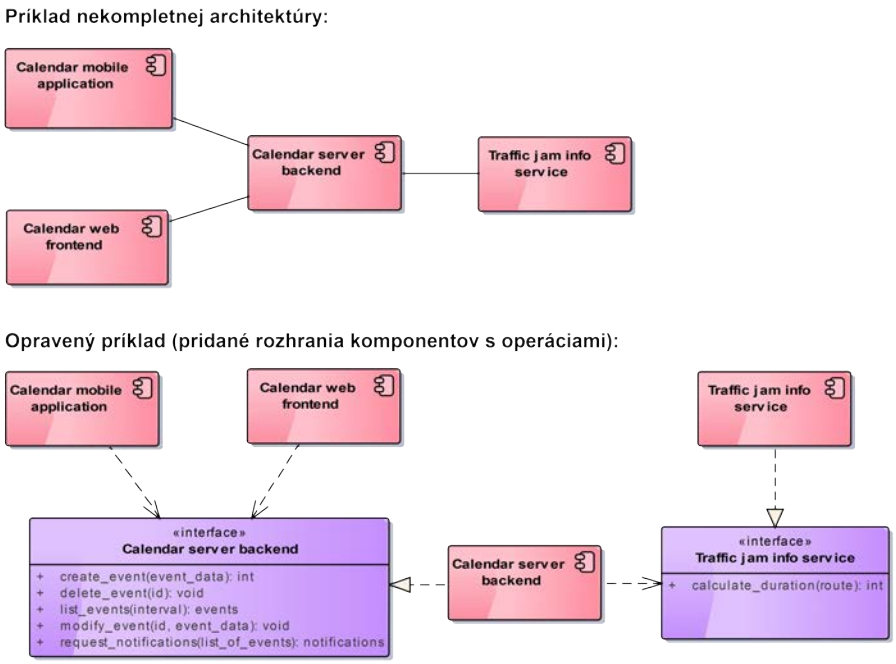
Poznámka: architektúra môže mať a často má viacero úrovní. Určitý komponent, ktorý je súčasťou architektúry celého softvéru môže mať vnútorne ešte vlastnú architektúru, zloženú z menších súčastí.

2.2.5 Na čo nám slúži architektonický návrh softvéru (prečo ho vytvárame)?

architektúra

Vytvorenie architektúry softvéru má niekoľko funkcií zároveň:

- Dekompozícia. Architektúra rozdeľuje softvér na menšie časti, nad ktorými sa separátne ľahšie uvažuje a taktiež sa dajú separátne vytvárať. Zároveň možno na jej základe rozdeľovať prácu medzi súčasti vývojového tímu.
- Spresní sa rozsah softvéru. Vytvorením architektúry získavame lepšiu predstavu, čo všetko vnútorne bude softvér robiť, často prídeme aj na veci, ktoré



Obr. 2.3: Horná časť obrázku je príkladom častej chyby pri tvorbe architektúry, kedy návrhár vytvorí iba názvy komponentov, ktoré pospája nepomenovanými vzťahmi. Hlavným problémom je, že takto „definovaná“ architektúra je veľmi vágna a reálne nepomáha v dekompozícii softvéru. V dolnej časti je tento problém odstránený presným definovaním rozhraní komponentov. Ku kompletnému opisu architektúry tiež patrí slovný opis jednotlivých komponentov, ktorý v tomto obrázku naznačený nie je.

sme predtým neuvažovali (napríklad že budeme potrebovať netriviálne transformovať dáta pri ich prechode z jedného modulu do druhého, alebo že prídeme na to, že niektorá zo služieb, ktorú sme predpokladali že ju využijeme ako externú, bude musieť byť implementovaná v rámci nášho softvéru).

- Ukážu sa závislosti. Vzťahy medzi komponentmi indikujú, ktoré súčasti softvéru bude treba vytvoriť skôr ako iné a taktiež bude vidno, ktoré sú pre softvér kritické, lebo od nich veľa závisí.
- Je to nástroj komunikácie. Tak ako všetky modely, aj architektúra je nástrojom komunikácie [2].
- Podporuje sa znovupoužitie. Identifikovaním komponentov a definovaním ich správania zvyšujeme šancu, že niektoré z nich budú znovupoužité, pretože sa ľahšie identifikuje, že dané služby už sú vytvorené [2].

2.2.6
súdržnosť

Čo je to súdržnosť softvérovej súčiastky a o čo sa v súvislosti s ňou snažíme pri dobrom návrhu?

Sila vnútorných väzieb (závislostí) v rámci softvérovej súčiastky. Vystihuje, nakoľko sú od seba jej časti závislé a ako intenzívna je ich interakcia. Snažíme sa aby bola čo najvyššia (vtedy má zmysel aby bol komponent komponentom a jeho súčasti držané a menené spoločne).

2.2.7
zviazanosť

Čo je to zviazanosť softvérových súčiastok a o čo sa v súvislosti s ňou snažíme pri dobrom návrhu?

Sila väzieb (závislostí) medzi softvérovými súčiastkami. Snažíme sa aby bola čo najnižšia (aby zmeny jednej súčiastky čo najmenej ovplyvňovali súčiastky iné, čím sa následne vyhneme zbytočne komplikovaným zmenám, na ktorých sa musí podieľať zbytočne veľa ľudí).

2.2.8
dekompozícia

Prečo je v návrhu softvéru dôležitá dekompozícia?

Dekompozícia v návrhu softvéru, teda rozdeľovanie softvéru a prác na ňom na menšie časti, nám umožňuje vysporiadať sa so zložitosťou. Ak softvér rozdelíme na malé časti, úsilie na ich vývoj bude v sume menšie, než keby sme sa pokúšali vytvoriť softvér ako monolit.

2.2.9
dekompozícia, zviazanosť

Ak je v súčte lacnejšie vytvorenie softvéru po menších súčiastkach (oproti vývoju monolitu), prečo nemôžeme donekonečna znižovať veľkosť súčiastok?

Kvôli rastu nákladov na integráciu súčiastok navzájom. Je totiž dôvodné sa domnievať, že ďalším rozbíjaním niektorých celkov (súčiastok) nedosiahneme dostatočne voľne zviazané súčiastky. Príliš vysoká zviazanosť začne spôsobovať také problémy, že sa ďalšie delenie neoplatí.

2.2.10
dekompozícia

Aké druhy dekompozície softvéru v návrhu poznáte? Vysvetlite, čo znamenajú.

Druhy dekompozície sú hľadiská, podľa ktorých možno softvér rozdeliť na menšie časti:

- *Funkcionálna*. Softvér rozdeľujeme podľa funkcií, ktoré má poskytovať. Môžeme napríklad zobrať jednotlivé prípady použitia a venovať sa návrhu softvéru vždy iba pre jeden z nich (samozrejme s predpokladom, že budeme neskôr jednotlivé časti spájať).
- *Udalostná*. Softvér navrhujeme tak, že najskôr identifikujeme všetky možné podnety z vonka (vstupy do softvéru či už od používateľov alebo externých

systémov) a následne sa pri každom zaoberáme tým, čo pre ich „vybavenie“ bude softvér robiť.

- *Štruktúrna*. Softvér rozdeľujeme podľa jeho „statickej“ štruktúry. Na najvyššej úrovni podľa navrhnutej architektúry, na nižších podľa štruktúry jednotlivých súčiastok (až na úroveň tried)
- *Používateľské rozhranie*. Špeciálny, no významný prípad štruktúrnej dekompozície, pri ktorej softvér delíme a navrhujeme jednotlivo pre súčasti používateľského rozhrania (napr. jednotlivých obrazoviek).
- *Údajová*. Po vytvorení modelu údajov tento rozdelíme na menšie časti (jednotlivé entity alebo skupiny entít) a následne sa zaoberáme iba tými časťami softvéru, ktoré s jednotlivými entitami súvisia.
- *Objektová*. Softvér delíme podľa tried a metód.

Pozor: To že sa rozhodneme uplatniť nejaký druh dekompozície neznamená, že sa ostatným aspektom softvéru nevenujeme, práve naopak. Ak si napríklad vyberieme funkcionálnu dekompozíciu, pre každý scenár budeme uvažovať s akými údajmi bude pracovať, aké triedy využívať, aké časti používateľského rozhrania zahrnie a pod.

Poznámka: nie každý druh dekompozície rozdeľuje softvér „čisto“ na vzájomne nezávislé a neprekrývajúce sa súčasti. Napríklad dva prípady použitia môžu zdieľať niektoré algoritmy či údaje. Znamená to, že je taká dekompozícia neužitočná? Nie. Hoci by sa mohlo zdať, že bude jednoduchšie sa zaoberať takýmito časťami softvéru spoločne, väčšinou to spôsobí problémy, pretože daný problém ako celok zostane príliš zložitý. Je spravidla jednoduchšie navrhnú softvér pre každú časť separátne a potom v ďalšom prechode (iterácii) hľadať spoločné znaky a návrhy revidovať.

2.2.11 dekompozícia

Aký je rozdiel medzi funkcionálnou a udalostnou dekompozíciou softvéru pri návrhu?

Podobnosť je pomerne veľká, oba typy dekompozície sú orientované na procesy, ktoré sa v softvéri odohrávajú.

Rozdiel je v tom, že pri funkcionálnej dekompozícií sú kritériom rozdelenia identifikované služby softvéru (zmysluplné scenáre s hodnotou pre zákazníka/používateľa), čo nie je to isté čo udalosti (akékoľvek podnety, ktoré do softvéru prichádzajú z vonka). Scenár služby softvéru (prípadu použitia) môže obsahovať aj viacero udalostí (nielen iniciálnu), naopak, nejaká udalosť môže byť súčasťou viacerých scenárov.

Dá sa povedať, že udalosti predstavujú „jemnejšie“ rozdelenie softvéru ako služby (funkcie).

2.2.12 Prečo pri tvorbe jedného softvéru spravidla potrebujeme uplatniť viaceré druhy dekompozície?

dekompozícia

Medzi dôvody patrí:

- Rôzne druhy dekompozície majú rôznu hrubosť, spravidla najskôr potrebujeme uplatniť tie „hrubšie“ (funkcionálnu, štruktúrnu) až potom možno pracovať s jemnejšími (objektová).
- Aby sme softvér vytvorili správne, mali by sme mať „poriadok“ v každom možnom uhle pohľadu naň. Ťažko si napríklad predstaviť, že by sme softvér navrhli čisto skrz funkcionálnu dekompozíciu. Pravdepodobne by sme nevytvorili efektívne používateľské rozhranie ani správny model údajov.
- Uplatnenie rôznych druhov dekompozície pôsobí aj ako „skúška správnosti“ návrhu. Ak napríklad uvažujeme dekompozíciu podľa používateľského rozhrania, pri návrhu jednotlivých častí by sme postupne mali naraziť na všetky služby softvéru, ak na ne nenarazíme, pravdepodobne niektorá zo služieb nebola z pohľadu používateľského rozhrania navrhnutá.

2.2.13 Čo je to návrhový vzor?

návrhový vzor

Návrhový vzor je všeobecné, abstraktné, praxou overené riešenie často sa opakujúceho problému v softvérovom inžinierstve. V užšom zmysle sú návrhové vzory znovupoužiteľné princípy organizácie tried softvéru. Pojem označuje predovšetkým vzory identifikované gangom štyroch [4].

Účelom používania návrhových vzorov je udržiavanie softvéru vhodne flexibilného a pripraveného na prípadné zmeny a rozšírenia.

Medzi známejšie vzory patria napríklad: *singleton*, *strategy*, *composite* či *facade*.

2.2.14 Čo je to architektonický štýl?

architektonický štýl

Architektonický štýl je znovupoužiteľný (a opakovane používaný, praxou preverený) spôsob, akým navrhujeme architektúru softvéru a definuje niektoré znaky, ktoré takáto architektúra vykazuje.

Zvolením architektonického štýlu do značnej miery definujeme „filozofiu“ organizácie celého softvéru a zároveň architektúru štandardizujeme.

Medzi známejšie architektonické štýly patria napríklad: *klient-server*, *dátovody a filtre*, *vrstvy* či *tabuľa*.

2.2.15
návrhový vzor,
architektonický štýl

Aký je rozdiel medzi architektonickým štýlom a návrhovým vzorom?

Oba pojmy označujú preverené, znovupoužiteľné, všeobecné a abstraktné riešenia návrhových problémov v softvéri.

Líšia sa však úrovňou pôsobnosti. Architektonické štýly sa týkajú celého softvéru, prejavujú sa na úrovni architektúry a určujú vzťahy medzi celými komponentmi. Návrhové vzory predpisujú vzťahy medzi triedami, spravidla v rámci jedného komponentu.

Rozdiel je ešte aj v spôsobe, akým vznikli. Vzory vznikli na základe odpozorovania osvedčených štruktúr, štýly boli primárne najskôr vymyslené a až potom overované.

2.2.16
architektonický štýl

Aké architektonické štýly poznáme? Stručne ich opíšte.

Z existujúcich štýlov vyberáme:

- *Klient-server*. Používateľské aplikácie (klienti) sú oddelené od ústredného servera, ktorý pre ich fungovanie poskytuje potrebné služby. Rozlišujeme pritom ešte pojmy tučný a tenký klient, ktorými označujeme (relatívne) koľko aplikačnej logiky sa nachádza na strane klientskej aplikácie.
- *Model-view-controller*. V softvéri je prísne oddelená aplikačná logika, údaje a používateľské rozhranie. Tento štýl je veľmi významne prítomný v architektúrach webových aplikácií, požíva sa v kombinácii so štýlom klient server. Je základom webových rámcov (*frameworks*).
- *Vrstvy*. Komponenty si virtuálne predstavme zoradené v zástupe. Závislosti medzi komponentmi pri použití štýlu vrstvy potom existujú len medzi susednými komponentmi. Príkladom použitia štýlu vrstvy môže byť celý softvér počítača, pozostávajúci z vrstiev bezprostredne ovládajúcich hardvér, ďalej vrstiev operačného systému a programových platforiem (napr. Java) až po používateľské aplikácie.
- *Dátovody a filtre*. Filtre sú komponenty softvéru, ktoré priebežne transformujú údaje prichádzajúce na ich vstup na výstupné údaje, ktoré sú následne poskytované ako výsledky alebo presúvané prostredníctvom dátovodov do ďalších filtrov. Príkladom sú systémy prúdového spracovania dát, či spôsob, akým na seba napájame programy na spracovanie textu v UNIX operačných systémoch.
- *Servisne orientovaná architektúra (SOA)*. Systémy budované ako SOA, pozostávajú z viacerých služieb a klientskych aplikácií, ktoré tieto služby rôzne využívajú. Ich spoločnú integráciu zabezpečuje tzv. zbernica služieb (enterprise service bus), na ktorú sú všetky služby a klientske aplikácie pripojené a ktorá zabezpečuje vhodný výber a kombináciu služieb podľa zadaných požiadaviek z klientskych aplikácií. SOA je typická pre veľké, podnikové informačné systémy.

- *Tabuľa*. Ústredným integračným prvkom softvéru je pasívne úložisko dát (tabuľa), informácií a znalostí. Aktívne časti softvéru čítajú z tohto úložiska informácie, spracúvajú ich a odvodené informácie opäť zapisujú „na tabuľu“. Iné aktívne časti iba získavajú informácie z okolia softvéru a ďalšie zobrazujú (s prípadnou transformáciou) stav tabule používateľovi.

2.2.17
model view controller

Uveďte príklad alebo typ aplikácie kedy NIE JE vhodné používať architektonický štýl MVC (neuvažujte triviálne príklady a la kalkulačka, kde to nemá zmysel z dôvodu jednoduchosti).

Faktom je, že štýl MVC pomáha dobre oddeľovať záležitosti, ktoré by v softvéri mali byť čo najmenej prepletené. Preto po ňom intuitívne siahame často. MVC však neznamená len rozdelenie softvéru na údaje, používateľské rozhranie a aplikačnú logiku (to robíme všade), ale aj spôsob, akým tieto časti spolupracujú, napr: že spúšťacie udalosti pochádzajú od používateľa, že controller trvalo neuchováva žiaden stav, že úložisko údajov je pasívne a podobne (webové aplikácie a rámce na ktorých stoja, napr. Ruby on Rails sú toho dobrým príkladom).

Práve spôsob, spracovania udalostí v softvéri je často dôvod, že pri vývoji niektorých softvérov nemusí byť MVC vhodným architektonickým štýlom, napr.:

- *Autoritatívny server*. Ide o aplikácie, pri ktorých je z nejakého dôvodu potrebná „iniciatíva“ pri generovaní udalostí zo strany servera. Typickými predstaviteľmi sú sieťové hry, pri ktorých server šíri aktualizácie stavu aplikácie na klientke stanice.
- *Inteligentný agent*. Ak je softvér umelo inteligentný, autonómny agent (interagujúci s prostredím), spravidla je riadený nekonečným cyklom vyhodnocujúcim aktuálne dostupné informácie zo senzorov a fakty zapamätané si z minulosti. V prípade potreby ovplyvňuje prostredie navonok, prostredníctvom efektorov. Do koncepcie MVC takýto prístup nezapadá jednak preto, že vnemy z prostredia len ťažko možno vnímať ako „požiadavky okolia“ (iniciatíva, čo na ich základe softvér spraví závisí od softvéru samého) jednak pre kontinuálnu aktivitu agenta (neustále vyhodnocovanie aktuálneho stavu).
- *Softvéry založené na stavových strojoch*. Niektoré softvéry, ako napríklad softvér bankomatu, či automatu na lístky sú riadené skôr pravidlami a aktuálnym stavom (ktorý držia v pamäti) a neuplatňujú procedurálny či objektovo-orientovaný prístup k zápisu algoritmov, ktorý by umožnil oddelenie aplikačnej logiky od údajov.
- *Softvéry transformujúce údaje*. Napríklad shell skripty v operačnom systéme či skripty v nástrojoch na matematické výpočty. Nemajú spravidla interaktívne používateľské rozhranie. Ich účelom je spracovať údaje ktoré dostanú na vstupe a transformovať na požadované výsledky. Jedno spustenie takéhoto softvéru spravidla znamená vybavenie jednej požiadavky.
- ...

2.2.18
model view controller

Akú funkciu v MVC má model? Čo ho tvorí?

Modelom rozumieme modul softvéru, ktorý uchováva údaje a definuje ich štruktúru. Je jediným miestom softvéru, v ktorom sú perzistentne uchovávané údaje.

Kľúčovou súčasťou realizácie modelu je databáza, prostredníctvom ktorej sú údaje uchovávané na fyzické médium. Nebýva však jedinou súčasťou modulu modelu v softvéri – do modelu spravidla zahŕňame aj obaľujúci programový kód (napísaný v programovacom jazyku aplikačnej vrstvy), ktorý databázu sprostredkúva aplikačnej vrstve.

Úlohou *obalu* je zjednodušiť prístup k údajom – z pohľadu aplikačnej vrstvy skrýva implementačné detaily. V prípade použitia relačných databáz veľmi často úplne skrýva SQL príkazy. Mnohé objektovo orientované rámce na to využívajú objektovo-relačné mapovanie, ktoré je dokonca využívané ako nástroj samotného definovania modelu údajov, takže vývojár s databázou samotnou (a písaním SQL príkazov) nemusí vôbec prísť do styku.

2.2.19
model view controller

Akú funkciu v MVC má view?

View je modul softvéru, definujúci rozhrania pre používateľov, teda všetko čo súvisí so zobrazovaním informácií a ovládania softvéru. View predstavuje bránu pre používateľa do softvéru. Sprostredkuje požiadavky používateľa controlleru. Zobrazuje stav *modelu*.

To zahŕňa definovanie statických charakteristík používateľského rozhrania: štýlu, kompozície, informačnej architektúry.

View definuje aj dynamické charakteristiky používateľského rozhrania ako animácie a interaktivitu prvkov rozhrania. Obsahuje teda aj „vykonateľný kód“, ktorý sa však striktne týka len logiky zobrazovania a je oddelený od aplikačnej logiky *controllera*.

2.2.20
model view controller

Akú funkciu v MVC má controller?

Controller je modul softvéru, ktorý spracúva udalosti vykonané používateľom, mení stav modelu a realizuje aplikačnú logiku.

2.2.21
tučný klient, tenký klient

Vysvetlite pojmy tučný a tenký klient?

Pri softvéri architektúry klient-server označuje pojem „tenký klient“ prípad, kedy klientska aplikácia (bežiaca lokálne na pracovnej stanici používateľa) neobsahuje (takmer) žiadnu aplikačnú logiku a slúži len ako prezentačná vrstva, ktorá prijaté príkazy používateľa obratom posiela na server, kde sú spracované a následne len zobrazuje výsledky spracovania.

„Tučný klient“ označuje prípad, kedy je aplikačná logika realizovaná na strane klienta a server sa stáva len podpornou službou. Bežné pre tučného klienta je tiež perzis-

tentné uchovávanie údajov aj na strane klienta. Tučný klient často dokáže pracovať autonómne, bez pripojenia na serverové služby.

Poznámka: Uvedené pojmy predstavujú extrémy a málokedy sa dnes stretneme s aplikáciami, ktoré by ortodoxne zodpovedali jednému alebo druhému. Reálne aplikácie sa nachádzajú v spektre medzi týmito pojmami, niektoré sú tenšie, niektoré tučnejšie.

2.2.22

tučný klient, tenký klient

Má pri webových aplikáciách zmysel hovoriť o tučnom klientovi?

Má. Hoci pôvodnou filozofiou webu je, že klient je tenký (i.e., že akákoľvek dynamika je realizovaná skrz server odoslaním požiadavky a vykreslením HTML dokumentu, ktorý sa vráti v odpovedi). Nástup jazyka Javascript však toto zmenil. Javascript umožňuje skriptovanie na strane klienta a v momente keď toto skriptovanie prestáva slúžiť iba na realizáciu dynamiky zobrazovania informácií ale zasahuje aj do aplikačnej logiky (zvyčajne za účelom nezaťažovať server a redukovať dobu odozvy aplikácie realizáciou niektorých výpočtov lokálne), prestáva byť aplikácia tenkým klientom.

„Najtučnejšími príkladmi“ môžu byť napríklad HTML5 hry.

Poznámka: Striktne vzaté možno pod pojmom webová aplikácia chápať len aplikácie postavené na HTML. V širšom ponímaní by sme mohli uvažovať aj tzv. bohaté internetové aplikácie (*rich internet applications*), bežiace na špeciálnych platformách ako Flash alebo Silverlight. Takéto aplikácie zo svojej podstaty tiež spadajú skôr do kategórie tučných klientov.

2.2.23

tučný klient, tenký klient

Aké sú výhody a nevýhody použitia tučného a tenkého klienta? Ako je to špecificky v prípade webových aplikácií?

Tenšie klientske aplikácie obsahujú menej funkcionality, z čoho vyplýva niekoľko výhod:

- Nedochádza k nekonzistentnostiam stavov na klientovi a na serveri (prechod do iného stavu klientskej aplikácie prebehne výlučne skrz požiadavku servera, ktorý klientovi „oznámi“ v akom stave sa po požiadavke nachádza).
- Je šanca, že ich (klientske aplikácie) budeme musieť menej aktualizovať, keďže aktualizácie sa mnohokrát obmedzia len na serverovú časť.
- Vieme lepšie kontrolovať, čo sa v aplikácií deje. Opäť najmä preto, že sa aktivita odohráva na serveri, na ktorý máme lepší dosah.
- Budú mať menej závislostí na technológiách, ktorými musia byť vybavené klientske pracovné stanice.
- Klientske stanice nemusia byť zvlášť výkonné (musia zvládať len úlohu zobrazovania informácií).

- Lepšia bezpečnosť aplikácií (klientom sa posiela len to čo je bezpodmienečne nutné, aplikačná logika môže zostať na serveri spolu s algoritmami, ktoré nechceme potenciálne zverejniť).

V kontexte webu majú webové aplikácie navyše tieto výhody:

- Aktualizácie klientskych aplikácií sa stávajú triviálne
- Praktická platformová nezávislosť

Tučný klient však môže v niektorých prípadoch predstavovať výhodu:

- Klientske aplikácie môžu byť do určitej miery autonómne.
- Bremeno náročných výpočtov nespočíva len na serveri (prínos z hľadiska škálovateľnosti).
- Klientske aplikácie môžu byť interaktívnejšie a mať rýchlejšiu odozvu.

V kontexte webu o tučných klientoch zároveň platí:

- Aktualizácie tučných webových klientskych aplikácií nie sú také náročné ako pri ich desktop náprotivkom.

2.2.24

web, webové aplikácie

Aké výhody majú webové aplikácie oproti desktop aplikáciám?

Výhody zahŕňajú:

- *Vyššia dostupnosť*. Banálne vyzerajúca vlastnosť, no k používaniu webových aplikácií sa používateľ dostane rýchlejšie, nakoľko odpadá potreba inštalácie.
- *Multiplatformovosť*. Webové prehliadače zjednocujú špecifiká pracovných staníc (rôzne operačné systémy, hardvér) a interpretujú zdrojový kód webových aplikácií všade rovnako. Webové aplikácie tak s jednou implementáciou môžu bežať na oveľa viac strojoch, ako desktop aplikácie.
- *Jednoduchšie aktualizácie*. Aktualizácia klientskej časti aplikácie je pri webe implicitná (stránka sa načíta zakaždým nanovo), zatiaľ čo z pohľadu používateľa pohodlné aktualizácie desktop aplikácií vyžadujú dotvorenie extra infraštruktúry na podporu aktualizácií.

Poznámka: možno sa pýtať, prečo nie sú všetky aplikácie webové? Web a prehliadače nezvládajú robiť efektívne všetko, napr. náročné výpočty (matematické a analytické nástroje), spracovanie zložitej grafiky (dizajnérske aplikácie, hry) či podporu samotného vývoja softvéru (vývojové prostredia). Dôvodom je najmä neefektívne využívanie hardvéru prehliadačmi a webovými aplikáciami, čo je priama daň za multiplaftormovosť (platí to všeobecne: čím viac virtualizačných vrstiev využívame, tým viac prostriedkov „zhltnú“ a softvér je výpočtovo menej efektívny).

2.2.25
softvérová knižnica, softvérový rámec

Aký je rozdiel medzi softvérovou knižnicou (library) a softvérovým rámcom (framework)?

Oba pojmy označujú softvér (nazvime ho jednotne *znovupoužitý softvér*), ktorý vytvoril niekto iný a my ho (znovu)využívame v nami vyvíjanom softvéri (ktorý by sme analogicky mohli nazvať *klientsky softvér*; používa sa však skôr pojem *klientsky kód*).

Hlavným rozdiel medzi je v určení autority (z angličtiny vzťah *slave-master*) vo vzťahu znovupoužitého softvéru (knižnice alebo rámca) a klientskeho kódu:

- Pri knižnici má autoritu klientsky kód nad knižnicou. On obsahuje hlavný algoritmus, ktorý celý softvér riadi a na knižnicu sa obracia keď potrebuje jej služby. Napríklad aplikácia na plánovanie jázd (*klientsky kód, master*) môže pre výpočet optimálnej cesty využívať špecializovanú knižnicu implementujúcu rôzne grafové algoritmy (*slave*).
- Pri rámci má autoritu rámec nad klientskym kódom. Čiže, rámec definuje hlavný tok kontroly v aplikácií a klientsky kód volá podľa potreby. Dôvodom tejto „inverzie“ je odlišná úloha rámcov oproti knižniciam. Kým knižnice majú za úlohu realizovať čiastkové úlohy softvéru čo najefektívnejším spôsobom, úlohou rámca je dať celej aplikácií čo najštandardnejšiu kostru a odbremeniť vývojárov od opakovanej implementácie rovnakých vecí v každom projekte. Vývojári sa tak môžu sústrediť na esenciu softvéru – reálnu funkcionalitu, ktorá tvorí hodnotu softvéru. Typickým príkladom tohto sú webové rámce ako Ruby on Rails. Pri ich použití nezačíname vývoj vytváraním kostry zabezpečujúcej komunikáciu so sieťovou infraštruktúrou, napojenie na databázu, objektovo-relačné mapovanie, bezpečnosť komunikácie či riadenie toku udalostí v rámci MVC. Všetky tieto esenciálne prvky webovej aplikácie sú už pripravené a možno okamžite pracovať na „zmysluplnej“ funkcionalite. Aby však táto „zmysluplná funkcionalita“ mohla fungovať, musí sa pravidlám rámca podriadiť.
- *Poznámka*: v praxi sa vyskytujúce softvérové rámce sú často dodávané spolu s knižnicami. Klientsky kód je teda podriadený rámcu, ale na mnohé (očakávané) úlohy môže využiť pomoc predpripravených knižníc.

2.2.26
prípad použitia

Prečo triedy vo vytváranom systéme viažeme na prípady použitia?

Sledujeme tým, či vývojom smerujeme k naplneniu špecifikácie, pomáha to pri kontrole správnosti návrhu. Ak triedu nedokážeme priradiť k žiadnemu prípadu použitia, je otázne, načo sme ju vôbec vytvorili. Skúška správnosti platí aj naopak: ak sa na prípad použitia neviažu žiadne triedy, pravdepodobne sme ho ešte neimplementovali.

2.2.27
návrh

Aké sú charakteristiky dobrého návrhu softvéru (na mysli máme výstupy, nie samotnú etapu)?

Poznámka: táto otázka by na vás mala pôsobiť odstrašujúco. Nemá krátku odpoveď a taktiež nemá zmysel sa uvedený zoznam nižšie učiť naspamäť (okrem toho určite nie je kompletný, obsahuje len najdôležitejšie charakteristiky). Neznamená to ale, že sa ňou nemáme zaoberať. Naopak, dá sa povedať, že je z celej podkapitoly najdôležitejšia a dobrý vývojár si ju vo vzťahu k softvéru, ktorý vyvíja, kladie často.

Medzi charakteristiky dobrého návrhu softvéru patrí:

- Jasný „návod“ ako softvér implementovať. Dobrý návrh by nemal nechávať otvorené zásadné otázky, len detaily.
- Vysoká súdržnosť a nízka zviazanosť súčiastok (komponentov) softvéru. Vhodná zložitosť súčiastok.
- Konzistentnosť v rámci a aj medzi jednotlivými modelmi (metódy zodpovedajúce scenárom, triedy zodpovedajúce údajom, údaje zodpovedajúce scenárom a používateľským rozhraniam atď).
- Dekompozícia v takej miere, že je pomerne presne možné odhadnúť mieru úsilia potrebného na dokončenie projektu.
- Jasné mapovanie navrhnutých artefaktov na požiadavky.
- Zrozumiteľná dokumentácia (skúškou správnosti je príchod nových ľudí do projektu a rýchlosť ich oboznamovania sa s ním).
- Rozšíriteľnosť návrhu, pripravenosť na zmeny.

2.3 Implementacia

2.3.1
implementácia

Aké činnosti vykonávame v etape implementácie softvéru?

Činnosti v etape implementácie možno rozdeliť do troch skupín:

1. Realizácia softvérových súčiastok (programovanie, tvorba používateľských rozhraní v produkčnej kvalite, transformácia modelu údajov do fyzickej podoby, konfigurovanie znovupoužitých súčiastok, ...).
2. Tvorba dokumentácie k softvérovým súčiastkam.
3. Testovanie implementovaných softvérových súčiastok (vykonávanie jednotkových a integračných testov, prípadne programovanie ďalších).

2.3.2 Čo je výstupom etapy implementácie softvéru?

implementácia

Artefakty, vznikajúce počas implementácie zodpovedajú činnostiam v nej vykonávaným:

1. Zdrojové kódy (tvoriace alebo preložiteľné priamo na fungujúci softvér)
2. Dokumentácia (zahŕňa jednak „inline“ dokumentáciu priamo v zdrojových kódoch, jednak externé dokumenty ako opisy a návody na použitie súčiastok, inštalačné a používateľské príručky a podobne).
3. Testy a záznamy o testovaní

2.3.3 Prečo na vytvorenie softvérového systému potrebujeme spravidla viac softvérových (programovacích) jazykov?

programovací jazyk

Žiaden programovací jazyk nie je univerzálne vhodný na všetko. Pokusy taký jazyk vytvoriť zvyčajne končia tak, že jazyk zostáva vhodný len na niektoré veci a pri iných je práca s ním mimoriadne ťažkopádna. Na svete sú preto desiatky (reálne používaných) jazykov.

Niekedy ani nemáme na výber, pretože použitie viacerých jazykov diktuje architektúra, ekosystém alebo historické súčasti softvéru (tzv. zdedený, angl. *legacy* kód). Napríklad vytvorenie *view* webovej aplikácie potrebujeme hneď tri jazyky (HTML+CSS+Javascript), na *controller* nejaký všeobecný OO jazyk (napr. python, ruby, java) a na *model* dopytovací jazyk do databázy (SQL). Rozhodovanie, či použiť viac jazykov však stále zostáva, iba sa presúva na úroveň jednotlivých komponentov.

Na ilustráciu uvádzame 3 príklady, kedy je možné potenciálne použiť len jeden jazyk, no nie je to výhodné:

1. Pri webových aplikáciách, pri ktorých očakávame veľkú záťaž, niekedy potrebujeme jeden komponent najvyššej úrovne, napríklad *controller* vnútorne štruktúrovať na niekoľko sub-komponentov a každý napísať v inom jazyku. Napríklad, naším rámcovým jazykom môže byť python (pretože v ňom je napísaný *framework* v ktorom sa dá rýchlo vyvíjať), no niektoré algoritmy môžeme kvôli výpočtovej efektívnosti chcieť implementovať v C (ktoré by sme ale na celú aplikáciu nepoužili, lebo jej vývoj by bol neúmerne prácnejší).
2. Iný príklad výhodného využitia viacerých jazykov sú expertné systémy (systémy na podporu rozhodovania ľudí vo vedúcich pozíciách). Tie spravidla potrebujú pracovať zo sadou pravidiel, ktorými sa ich rozhodnutia riadia (napríklad môže ísť o sadu pravidlá, za akých okolností má byť zákazníkovi banky pridelená hypotéka). Takéto pravidlá je výhodné programovať v logickom programovacom jazyku (napr. Prolog), pretože sa ľahšie udržujú a kód je čistejší. Avšak programovať *celý* controller v logickom programovacom jazyku by možno trvalo ešte dlhšie ako v predchádzajúcom príklade. Preto je v takomto prípade logicky programovať len rozhodovací stroj a ostatnú obaľujúcu a podpornú funkcionalitu programovať v niečom inom.

3. Iný príklad zbytočnej ťažkej práce spôsobenej použitím jedného jazyka je generovanie kódu používateľských rozhraní pri webových aplikáciách. Dnešné webové aplikácie sa zvyčajne neskladajú len zo statických stránok. O konkrétnej podobe stránky rozhoduje aplikačný kód na serveri. A práve tu je k dispozícií už viac možností, ako výsledný HTML kód pripraviť. Prvá možnosť je, že by aplikačný kód, písaný v niektorom „všeobecnom programovacom jazyku“ (napr. Java, C#, Ruby či Python) mohol žiadané HTML dokumenty vygenerovať kompletne algoritmicky spájaním reťazcov obsahujúcich kusy HTML kódu. Vtedy by sme mohli víťazoslávne prehlásiť, že na všetko používame jeden jazyk (zanedbajme teraz fakt, že ešte treba nejak vyriešiť prístup k dátam, kde spravidla potrebujeme opäť iný jazyk, SQL). Na druhej strane, takto programovať generovanie rozhrania by bolo zbytočne zložité a namáhavé. Práve preto sa využíva druhá možnosť, a síce špeciálne jazyky, ktoré majú deklaratívny charakter (napr. mutácie XML). Tieto jazyky oveľa prirodzenejšie definujú podobu používateľského rozhrania no zároveň povoľujú vkladanie kúskov aplikačného kódu na tých miestach štruktúry, ktoré sú dynamické (teda až v čase požiadavky sa pomocou nich dopočíta, ako majú vyzerať). Príkladom takéhoto prístupu ku generovaniu kódu používateľského rozhrania je napríklad *Embedded Ruby*.

2.3.4 Čo by sme pri výbere programovacích jazykov pre projekt mali zvažovať?

programovací jazyk

Naše úvahy nad voľbou jazykov budú v konečnom dôsledku vyhodnocovaním ich kladov a záporov v rámci viacerých kritérií. Nasledujúce zoznamy uvádzajú tie najdôležitejšie z nich. Keďže ich je veľa, oplatí sa ich zoskupiť.

Kritériá týkajúce sa znalostí jazykov vývojármi:

- **Skúsenosti programátorov s daným jazykom**. Veľmi často ide o určujúce, no zároveň krátkozraké kritérium. Problémy a zvýšené úsilie, ktoré môže nevhodne zvolený jazyk spôsobiť často prevýšia úsilie spojené s učením sa nového jazyka.
- **Rýchlosť s akou sa dá nový jazyk naučiť**.
- **Stratégia rozvoja našej organizácie**. Vývojárska firma môže napríklad dlhodobo preferovať určité vývojové prostredie či jazyk. Ak má napríklad firma viacero projektov, určite bude zo strany manažmentu tlak na to, aby sa všetky vyvíjali v rámci čo najmenšej skupiny jazykov (resp. v tomto prípade skôr celých ekosystémov, resp. *technology stack*-ov).

Technologické charakteristiky jazyka:

- **Výpočtová efektívnosť jazyka**. Všeobecná ako aj vo vzťahu k špecifickým úlohám, ktoré v konkrétnom projekte potrebujeme riešiť.
- **Existujúce knižnice pre špecifickú funkcionalitu**, ktorú v softvéri ideme riešiť. Prirodzene máme tendenciu si vyberať jazyk, v ktorom máme najviac predpripravenej funkcionality pre náš softvér.
- **Dostupnosť a cena dobrých vývojových nástrojov** (napr. IDE Eclipse, Visual

Studio) pre vývoj softvéru v danom jazyku.

Komunita okolo jazyka:

- **Rozšírenosť jazyka a aktivita vývojárskej komunity okolo neho**. S nástupom portálov komunitného odpovedania na otázky (napr. Stack Overflow) zásadne narástla dôležitosť aktívnej komunity pre úspešnosť vývoja v danom jazyku. Ak pri vývoji narazíme na problém, komunitu možno požiadať o radu a pomoc. Ak takúto možnosť nemáme, zostáva nám jedine existujúca dokumentácia, práca s ktorou je namáhavejšia a odpovede na naše otázky nemusíme vôbec dostať.
- **Je jazyk živý?** Každý jazyk má nejakých autorov. Nie každý jazyk ich však má stále aktívnych a je stále vylepšovaný. Pokiaľ jazyk žije, je väčšia šanca, že nám v prípade problémov bude mať kto poradiť, prípadne odstrániť chyby v jazyku.

Kritériá kladené z vonka (požiadavky zákazníka, kontext použitia):

- **Potreba kompatibility s konkrétnymi platformami**. Výber jazykov sa môže zúžiť, ak zákazník špecifikuje, že softvér musí využívať určité platformy.
- **Preferencia jazyka zákazníkom**. Zákazník môže z nejakého dôvodu preferovať implementáciu v niektorom jazyku. Napríklad aby softvér mohol v budúcnosti svojpomocne rozvíjať.

Poznámka: Zaujímavosťou uvedených zoznamov je, že niektoré kritériá aplikujú v praxi úplne prirodzene už aj vývojári-začiatočníci (znalosť jazyka), iné sú charakteristické skôr pre expertov a manažment (ako rozvinutá je komunita či ekosystém jazyka).

2.3.5 Čo sú to nástroje na správu verzií zdrojových kódov (source control tools)?

verzie zdrojového kódu

Softvérové nástroje, ktoré umožňujú uchovávať zdrojový kód vyvíjaného softvéru, pričom zároveň (a pohodlne pre vývojára) umožňujú

- replikáciu zdrojových kódov vo vzdialených úložiskách,
- uchovávanie histórie zmien zdrojového kódu,
- spravovanie paralelne existujúcich verzií zdrojového kódu (vetvy),
- manažovanie prístupu viacerých vývojárov k zdrojovému kódu,
- riešenie konfliktov vo verziách.
- ...

Navyše, okolo týchto nástrojov často vznikajú natívne alebo integrované prostredia pre manažment úloh, plánovanie a komunikáciu členov vývojových tímov. Tieto funkcie nezvykneme považovať za súčasť kľúčovej funkcionality nástrojov na správu

verzií, no výhodnosť ich napojenia na tieto nástroje ukazuje prax na portáloch ako *github* alebo *bitbucket*.

2.3.6
verzie zdrojového kódu

Prečo uchovávame zdrojový kód v nástrojoch na správu verzií zdrojových kódov?

Nástroje na správu verzií zdrojových kódov prinášajú viaceré výhody:

- **Zálohovanie**. Možnosť straty zdrojového kódu, napríklad v dôsledku ľudskej chyby alebo zlyhania hardvéru sa výrazne minimalizuje, lebo existujú vzdialené kópie, ktorých priebežné vytváranie je veľmi jednoduché (ak by sme to chceli robiť manuálne, môžeme, ale s oveľa väčším úsilím).
- **Možnosť návratu po neúmyselnom zavedení obsahových chýb**. Nástroje umožňujú ľahký návrat k predchádzajúcej funkčnej verzií softvéru v prípade, ak sme niečo pokazili (opäť úloha, ktorá sa síce dá vykonávať manuálne, ale je veľmi prácna).
- **Propagácia zmien zdrojového kódu v rámci tímu**. Manuálne veľmi prácna úloha náchylná na vznik chýb.
- **Možnosť experimentovať**. Vytvorením novej vetvy z existujúceho zdrojového kódu možno ľahko a bez obáv vyskúšať implementačné riešenia, o ktorých úspechu a vhodnosti nie sme dopredu presvedčení (a preto ich nechceme do softvéru zavádzať bez možnosti návratu).
- **Ľahšie monitorovanie postupu projektu**. Ak dôsledne využívame *commit* správy (komentáre pripájané ku každej zmene verzie, charakterizujúce túto zmenu), vieme spätne ľahko zhodnotiť, čo a kto na softvéri v minulosti zmenil.
- **Ľahšie hľadanie chýb**. *Commit* správy tiež slúžia na ľahšie lokalizovanie zmien, ktoré mohli spôsobiť neúmyselné zavádzanie chýb.

2.3.7
verzie zdrojového kódu

Aké dva druhy nástrojov na správu verzií zdrojových kódov poznáme z hľadiska umiestnenia úložiska?

Centralizované a *decentralizované*.

Centralizované sú historicky staršie a fungujú na princípe jedného úložiska (umiestneného na serveri), na ktoré sa jednotliví vývojári pripájajú a prispievajú do neho zmenami svojho zdrojového kódu.

Decentralizované nástroje majú úložísk viac a tieto úložiská sú navyše organizované v dvoch vrstvách: lokálnej (každý vývojár má svoje vlastné lokálne úložisko) a spoločnej (jedno alebo viac úložísk zdieľaných viacerými vývojármi). Zmeny zdrojového kódu sú každým vývojárom vykonávané nad lokálnym úložiskom a až následne propagované do spoločných úložísk.

2.3.8 Aké spôsoby na zabezpečenie toho, aby zmeny jednej časti zdrojového kódu dvoma programátormi neboli v konflikte poznáme v nástrojoch na správu verzií zdrojových kódov?

verzie zdrojového kódu

Nástroje na uchovávanie zdrojových kódov ponúkajú spravidla dva spôsoby: *uzamykanie* a *post-hoc riešenie konfliktov*.

- Uzamykanie spočíva v explicitnom zákaze vykonávania zmien nad časťou zdrojového kódu s výnimkou jediného vývojára, ktorý kód uzamkol. Po odomknutí kódu týmto vývojárom, môže uzamknutie využiť iný vývojár. Tento prístup je spoľahlivejší, je však veľmi ťažkopádny.
- Post-hoc riešenie konfliktov ponecháva všetok zdrojový kód voľne meniteľný všetkým vývojárom. Ak pritom nastanú konfliktné situácie (v prípade že jednu časť v rovnakom čase upravili dvaja vývojári), rozhodne jeden z vývojárov, ktoré zmeny z ktorej verzie sa prijmú, prípadne opraví chyby, ktoré zlúčením verzií vznikli. V praxi sa zvykne využívať skôr táto možnosť.

Vždy je snahou vývoj plánovať tak, aby programovanie nemuselo prebiehať súčasne nad rovnakými časťami kódu.

Treba si tiež uvedomiť, že toto rozdelenie je z pohľadu funkcií nástrojov. Z hľadiska manažmentu vývoja možno tieto nástroje využiť „na mnoho spôsobov“. Záleží na pravidlách, ktoré pre vývoj v organizácií platia. Napríklad vlastníctvo zdrojového kódu môže byť spoločné (každý má právo upravovať hociktorú časť) alebo rozdelené (za každú časť je primárne zodpovedný nejaký vývojár). Z toho napríklad môže vyplynúť, či má konflikty riešiť vývojár, ktorý nane prišiel, alebo vývojár, ktorý je za kód zodpovedný.

2.3.9 Čo je to „commit“ (v súvislosti s nástrojmi na správu verzií zdrojových kódov)?

verzie zdrojového kódu, commit

Pri vývoji softvéru s použitím nástroja na správu verzií zdrojových kódov, upravujú vývojári vždy svoje lokálne kópie zdrojového kódu. Po uskutočnení zmien v zdrojovom kóde ich vývojár „odovzdá“ do centrálneho úložiska (v prípade decentralizovaných nástrojov predtým ešte do lokálneho úložiska), čím sa kód v centrálnom úložisku aktualizuje a zmeny môžu „vidieť“ aj iní vývojári. Toto odovzdanie nazýva *commit*.

Commit by mal byť sprevádzaný krátkou správou od vývojára, ktorá by mala opisovať, aké zmeny sa v kóde udiali.

Malo by tiež byť snahou vývojárov robiť odovzdania kompaktné, ideálne jedna k jednej mapovateľné na logické zmeny a prírastky v zdrojovom kóde.

2.3.10
verzie zdrojového kódu, commit

Nástroje na správu verzií zdrojových kódov sa často používajú nesprávne. Aké chyby či nedbalosti sa v praxi často vyskytujú?

Medzi časté chyby a „zlozvyky“ patrí:

- **Neuvádzanie žiadnych (alebo uvádzanie nevhodných) *commit* správ**. Každé odovzdanie zdrojového kódu prirodzene reprezentuje nejakú zmenu oproti predchádzajúcemu stavu. *Commit* správa slúži na opis tejto zmeny. Pokiaľ tieto opisy nevytvárame, prichádzame o prehľad zmien.
- **Odovzdávanie „viacerých vecí naraz“**. *Commit* by mal byť ucelený (nemali by sme odovzdávať polovičaté zmeny), no zároveň by sa mal venovať len jednej záležitosti. Niekedy sa však stáva, že do jedného odovzdania vývojári zapracujú aj viacero záležitostí, ktoré spolu vzájomne až tak nesúvisia. Ak potom chceme vrátiť zmeny v niektorej z týchto záležitostí, musíme vrátiť aj zmeny v záležitostiach, ktoré sú s ňou zviazané, ale inak ich vracať nepotrebujeme.
- **Nepoužívanie vetiev**. Vetvenie verzií zdrojového kódu je po odovzdaniach (*commits*) ďalším užitočným nástrojom ako oddeliť zmeny zdrojového kódu, ktoré spolu nesúvisia. Poskytujú možnosť paralelného vývoja a poskytujú tak väčšiu flexibilitu aj čo sa týka prijímania zmien do hlavnej vetvy. Ak sa vetvenie nepoužíva, je správa zdrojového kódu podstatne menej flexibilná a môže dochádzať k častejším konfliktom. Rozpracované zmeny často nemôžu byť uzatvárané dostatočne rýchlo a môžu blokovať ostatných vývojárov, ktorí na projekte pracujú.
- ...

2.4 Testovanie

2.4.1
testovanie

Prečo softvér testujeme?

Najdôležitejšie dôvody zahŕňajú (nemusia byť nutne jediné, zoznam je otvorený):

- Testovaním nachádzame v softvéri chyby.
- Testovaním demonštrujeme, že softvér funguje (aspoň do určitej miery). Sila tvrdenia, že softvér funguje zodpovedá pokrytiu softvéru testami.
- Testovanie dáva vývojárom istotu. Ak máme zdrojové kódy pokryté automatickými testami, tieto nám včas pomôžu odhaliť chyby, ktoré ďalším programovaním do softvéru nechtiac zanášame.
- Testovanie nám umožňuje sústrediť sa na to čo je podstatné. Testy možno vnímať ako formu špecifikácie (akceptačné testy ňou koniec koncov sú, ale rovnako môžeme brať aj integračné či jednotkové testy)

2.4.2

testovanie, hľadiská testovania

Uveďte hľadiská podľa ktorých delíme techniky testovania softvéru.

Testovanie môžeme deliť podľa nasledovných hľadísk:

- **Manuálne verzus automatizované**. Manuálne testovanei (keďže je vykonávané človekom) môže ísť do veľkej hĺbky a ľahko zachytiť aj neočakávané situácie, no je veľmi drahé, práve kvôli cene ľudského času. Automatizované testovanie je na druhej strane ľahko opakovateľné, lacné (po vytvorení testu ho môžeme využiť tisíce krát), no pokryť ním všetky aspekty softvéru nedokážeme.
- **Dynamické verzus statické**. Dynamické testovanie znamená, že softvér je testovaný počas svojho behu. Statické testovanie vykonávame nad nečinným softvérom (predovšetkým nad zdrojovými kódmi a dokumentáciou).
- **Čierna skrinka, Biela skrinka, Šedá skrinka**. Tieto tri pojmy označujú rôzne spôsoby, ako pripravovať testovacie vstupy a výstupy. Testovanie čiernou skrinkou vychádza iba zo špecifikácie testovaného softvéru (súčiastky). Testovanie bielou skrinkou vychádza z vnútornej štruktúry existujúceho programu (údaje pri ňom pripravujeme tak, aby sme overili správnosť fungovania všetkých možných prechodov cez graf podmienok programoch). Šedá skrinka označuje prípad podobný bielej skrinke, avšak tu vychádzame iba z *predpokladanej* vnútornej štruktúry programu (používame návrhové dokumenty obsahujúce informácie o algoritmoch a vnútorných dátových typoch).
- **Funkcionálne verzus nefunkcionálne**. Toto rozdelenie zodpovedá rozdeleniu požiadaviek na softvér. Funkcionálne testovanie je testovanie funkcionálnych a nefunkcionálne nefunkcionálnych požiadaviek (napr. overenie priebehu scenára je funkcionálne testovanie, overenie záťaže softvéru veľkým množstvom používateľov je nefunkcionálne testovanie).
- **Podľa (úrovne resp. fázy vývoja): Akceptačné, integračné, jednotkové testy**. Akceptačné testy zodpovedajú scenárom použitia, integračné testy testujú celé moduly, jednotkové testy zodpovedajú metódam.

2.4.3

testovanie, hľadiská testovania

Uvažujme dve dimenzie prístupov k testovaniu: manuálne-automatizované a dynamické-statické. Uveďte príklad testovania pre každú kombináciu „hodnôt“ z týchto dvoch dimenzií?

Manuálne dynamické: Napríklad akceptačné testovanie, ale aj hocijaké spustenie časti programu počas vývoja spojené s pozorovaním, či sa správa správne.

Manuálne statické: Napríklad prehliadky zdrojových kódov, prehliadky dokumentácie, kladenie kontrolných otázok o softvéri.

Automatizované dynamické: Napríklad jednotkové testovanie, integračné testovanie (okrem iného zahŕňa aj simulovanie ľudského správania nad používateľským rozhraním).

Automatizované statické: Sem patrí napríklad výpočet metrík zdrojového kódu za účelom získania kvantitatívnych ukazovateľov o zdrojovom kóde (pokrytie kódu automatickými testami, zložitosť metód, prepojenosť modulov, výskyt pachov a podobne).

2.4.4 Ktoré vlastnosti softvéru sa zvyknú najviac testovať?

testovanie, vlastnosti

Ako už otázka napovedá, *nebudú to všetky vlastnosti*, hoci teoreticky by testovateľné mali byť všetky (pretože ak v rámci nich formulujeme požiadavky, najmä nefunkcionálne, mali by byť merateľné, ergo, mali by sa pre ne dať napísať testy).

V praxi sa testujú najmä tieto vlastnosti:

- **Správnosť**. Najdôraznejšie testovaná. Spadá sem totiž testovanie funkcionality: akceptačné, intergačné a jednotkové testy. Mnohé projekty sa obmedzujú len na testovanie správnosti.
- **Spoľahlivosť**. Vedľajším produktom testovania správnosti je testovanie pomeru úspešných a chybových prípadov správania sa softvéru – spoľahlivosti. V najjednoduchšom prípade pôjde o spustenie testov správnosti dostatočný počet krát v čo najreprezentatívnejšej množine situácií použitia softvéru (aby sa ukázali prípadné chyby). Napríklad modul na generovanie notifikácií o udalostiach necháme bežať v zrýchlenom čase „desať ročia“, aby sme pokryli aj menej časté udalosti, ktoré môžu s časovaním vyvolávať problémy, napríklad korekcie času (v Unixových systémoch vykonávané raz za niekoľko rokov), prechodné roky, sviatky a podobne.
- **Robustnosť**. Od testovania správnosti možno ľahko prejsť k testovaniu robustnosti. Testy v tomto prípade pripravujeme s chybnými alebo neštandardnými vstupnými údajmi a okolnosťami.
- **Efektívnosť**. Vyhodnocuje sa pomerne často, pretože ide o relatívne jednoduché testovanie. Často stačí opäť len spustiť existujúce funkcionálne testy a merať časovú a pamäťovú záťaž, ktorú softvér vyvoláva.

Prečo sa teda taká pozornosť nevenuje ostatným vlastnostiam? Predovšetkým kvôli úsiliu, ktoré na testovanie treba vynaložiť (všimnime si, že vyššie uvedené vlastnosti možno testovať automatickými testami). Naproti tomu u vlastností ako je udržovateľnosť či modifikovateľnosť je automatické vyhodnocovanie nepostačujúce (výpočty metrík zdrojových kódov, sú pritom pri vyhodnocovaní týchto vlastností užitočné, no mali by mať len poradný charakter). U iných vlastností je potrebné manuálne testovanie (napr. použiteľnosť). Niektoré vlastnosti netreba testovať vždy, alebo niekedy na nich nezáleží (napr. bezpečnosť). No a často je dôvodom prosto len to, že niektoré vlastnosti nie sú dostatočne docenené (napr. použiteľnosť).

2.4.5
statické testovanie, dynamické testovanie

Porovnajte statické a dynamické testovanie softvéru.

Deliacou čiarou je, či pri testovaní potrebujeme, aby softvér bežal a mohli sme tak sledovať jeho správanie (dynamické testovanie) alebo nám stačí jeho „statická podoba“ (zdrojové kódy, dokumentácia).

Tento hlavný rozdiel potom determinuje techniky, ktoré pri testovaní využívame. Dynamické testovanie využíva často automatické testy, ktoré z definície nevieme použiť pri statickom testovaní, pretože využívajú bežiaci softvér. Pri statickom testovaní možno automatické prístupy využiť jedine pri výpočtoch metrík zdrojového kódu.

Z definície tiež vyplýva ďalšia odlišnosť: staticky možno testovať od samého začiatku projektu (už výstupy analýzy možno manuálne posudzovať), zatiaľ čo pre dynamické testovanie potrebujeme mať niečo naprogramované.

2.4.6
statické testovanie

V ktorých etapách životného cyklu softvéru používame statické testovanie?

Vo všetkých. Manuálny prístup totiž možno uplatniť hneď vo fáze analýzy, kedy možno posudzovať rôzne vlastnosti vytváranej dokumentácie.

2.4.7
statické testovanie

Je pravdivé tvrdenie: „statické testovanie sa používa najmä pri integrácii softvérového systému“?

Nie. Existujú etapy vývoja softvéru, kde je význam a využitie statického testovania väčšie: analýza a návrh. V nich nemáme inú možnosť ako testovať softvér staticky. Naopak pri integrácií, ktorá prichádza počas implementácie resp. pri nasadzovaní softvéru poslúžia lepšie dynamické, najmä automatické techniky testovania (najmä integračné testy).

2.4.8
statické testovanie

Je pravdivé tvrdenie: „statické testovanie sa používa iba pri analýze a návrhu“?

Nie. Staticky testujeme v každej etape vývoja. Staticky možno testovať akékoľvek softvérové artefakty, napríklad aj zdrojové kódy (už len tým, že zdrojový kód pri programovaní čítame a kontrolujeme, je statickým testovaním). Je však pravdou, že pre analýzu a návrh má statické testovanie oveľa väčší význam, ako pre iné etapy.

2.4.9
automatizované testovanie

Je pravdivé tvrdenie: „Automatizované testovanie môžeme realizovať vo všetkých etapách životného cyklu softvéru“?

Nie. V etapách analýzy a návrhu spravidla nevznikajú také artefakty, ktoré by sa dali automaticky testovať (požiadavky či návrhové diagramy ťažko automaticky otestujeme).

2.4.10
automatizované testovanie, nefunkcionálna požiadavka

Uveďte príklad nefunkcionálnej požiadavky na softvér, ktorú možno testovať automatizovaným testom.

Nefunkcionálne vlastnosti, ktoré možno testovať automaticky sú napríklad efektívnosť, robustnosť, spoľahlivosť. Príkladom konkrétnej požiadavky môže byť napríklad:

- *Požiadavka na efektívnosť*: 1000 po sebe idúcich žiadostí o výpočet sadzby za dopravu musí služba vypočítať za menej ako jednu sekundu. *Test*: v cykle tisíc krát za sebou požiadame aplikáciu o výpočet sadzby s náhodne vygenerovanými lokalitami dodania tovaru.
- *Požiadavka na robustnosť* (viac príkladov na robustnosť tu [3]): Pri zadaní dátumu v nesprávnom tvare aplikácia nespadne s neošetrenou výnimkou, ale pokúsi sa správny dátum odhadnúť zo zadaného vstupu s použitím alternatívnych dátumových vzorov. Zároveň upozorní používateľa. *Test*: podobný ako pri funkcionálnom testovaní. De facto totiž pôjde o overenie alternatívneho scenára.
- *Požiadavka na spoľahlivosť*: Zo sto požiadaviek o stiahnutie súboru zlyhá v priemere len jedna. *Test*: sto krát spustíme požiadavku o stiahnutie a skontrolujeme, či bol súbor správne prenesený. Eventuálne sa môžeme snažiť o spúšťanie s časovým odstupom a pri iných podmienkach.

2.4.11
testovanie

Čo v testovaní softvéru znamená pojem „fixture“?

Ide o východiskový stav, do ktorého softvér dostaneme pred začatím testovania. Niekedy sa týmto pojmom označuje aj procedúra, akým softvér do tohto stavu dostaneme.

Je dôležité softvér pred každým testovaním dostať do rovnakého stavu, pretože inak sa spoľahlivo nedajú testy replikovať (opakované spúšťanie testov by mohlo dávať rôzne výsledky). Fixture procedúra často manipuluje s databázou, resp. s akýmikoľvek perzistentnými údajmi, ktoré upravuje do východiskového tvaru (inak by vždy rôzne údaje v databáze mohli ovplyvňovať správanie softvéru). Časté je tiež kompletné reštartovanie a inicializácia softvéru, s následným naplnením databázy s testovacími údajmi.

Poznámka: pre niektoré testy fixture procedúry nepotrebujeme. Ide o prípady, kedy testovaná súčiastka nezávisí od nastavenia softvéru ani perzistentných údajov. Príkladom môže byť jednotkový test metódy s algoritmom, ktorý závisí len na vstupných údajoch (napríklad nájdenie mediánu v poli čísiel).

2.4.12
automatické testovanie

Ako sa dosahuje nezávislosť vykonávania jednotlivých automatických testov?

Znovunastavením do východiskového stavu softvéru pre testovanie pomocou *fixture* procedúry. Spravidla ak vykonávame viac testov, celý proces je riadený testovacím rámcom, ktorý procedúru spolu s testami spúšťa.

2.4.13
automatické testovanie, mutačné testovanie

Máme hotový softvér a tiež sadu testov, ktoré ho automaticky testujú, pričom všetky testy skončili úspešne. Ako sa môžeme dodatočne preveriť, či sú testy skutočne vytvorené správne?

Použijeme techniku zvanú mutačné testovanie. Zavedieme do zdrojového kódu umelé chyby jeho malými modifikáciami na náhodných miestach (podobne ako keď mutuje genetický kód). Príkladom môže byť výmena parametrov pri volaní funkcie alebo zmena logického súčinu na súčet a podobne (bežná chyba, ktorú robia aj programátori). Ak sú naše testy napísané dobre, mali by tieto chyby identifikovať. Ak prejdú, máme v testoch diery.

Poznámka: treba povedať, že použitie tejto techniky v praxi príliš nevidieť. Je totiž veľmi tvrdá. Okrem toho nedokáže emulovať všetky typy chýb, ktoré by pri vývoji softvéru mohli nastať (hlavne chyby vyššej úrovne abstrakcie).

2.4.14
verifikácia, validácia

Vysvetlite v čom spočíva porovnávacia stratégia verifikácie a validácie softvéru.

Necháme rovnakú súčiastku vytvoriť dvom nezávislým tímom, výsledky testujeme rovnakou množinou údajov a sledujeme či dávajú rovnaké výstupy.

Poznámka: v praxi sa takýto prístup používa málokedy - je veľmi drahý. Mieru istoty však poskytuje veľkú a ak je úlohou vytvorenie obzvlášť spoľahlivej súčiastky (napríklad pre oblasti ako energetika či kozmické lety), predstavuje jednu z možností zabezpečenia správnosti softvéru.

2.4.15
testovanie, čierna skrinka, biela skrinka

Aký je rozdiel medzi technikami testovania čierna skrinka (black-box) a biela skrinka (white-box) testovaním?

Testovanie čiernou skrinkou vychádza iba zo špecifikácie testovaného softvéru (súčiastky). Testovanie bielou skrinkou vychádza z vnútornej štruktúry existujúceho programu (údaje pri ňom pripravujeme tak, aby sme overili správnosť fungovania všetkých možných prechodov cez graf podmienok programoch).

2.4.16
testovanie, triedy ekvivalencie

Vysvetlite pojem „triedy ekvivalencie“ (v súvislosti s testovaním softvéru).

Pojem súvisí s prípravou testovacích údajov (sád vstupných a výstupných údajov), pomocou ktorých overujeme správanie softvéru alebo niektorého jeho komponentu. Pri vytváraní týchto údajov sa snažíme, aby čo najviac pokryli predpokladané scenáre, podľa ktorých sa softvér má správať, a to nielen tie „happy day“ scenáre (ktoré čakáme že budú najčastejšie) ale aj rôzne chybové scenáre, pri ktorých sa softvér tiež musí správne zachovať.

Mohlo by teda platiť tvrdenie, že čím viac sád pripravíme, tým lepšie pokryjeme testovaný softvér? Čo ak sme údaje zvolili tak, že v skutočnosti všetky opisujú podobné prípady situácií a odlišné situácie sa medzi nimi nenachádzajú? Predstavme si napríklad sadu testovacích údajov pre funkciu na výpočet logaritmu. Ak by sme pripravili veľa príkladov údajov iba s kladnými číslami ako vstupmi no opomenuli otestovať správanie funkcie v prípade dodania záporného argumentu, v skutočnosti sme testom pokryli ani nie polovicu prípadov (nezabúdajme na nulový argument, ktorý by sa teoreticky mohol byť definovaný ako nekonečno).

Uvedený príklad ilustruje, čo je to trieda ekvivalencie: je to taká skupina testovacích údajov, ktoré testujú podobné situácie, ktoré sú z pohľadu dôležitých čŕt rovnaké. Inými slovami, softvér by sa pri nich mal správať podobne.

Ako z uvedeného príkladu tiež vyplýva: nemá veľký zmysel testovať softvér viac než jednou sadou testovacích údajov z jedne triedy ekvivalencie (môžeme, ale spravidla tým len zbytočne predlžujeme testovanie). Na druhej strane, je úplne zásadné mať medzi používanými testovacími údajmi zástupcov zo všetkých tried ekvivalencie.

2.4.17
testovanie, triedy ekvivalencie, čierna skrinka, biela skrinka

Je pravdivé tvrdenie: „Pri určovaní testovacích vstupov technikou biela skrinka sa vychádza z rozdelenia vstupov/výstupov programu do tried ekvivalencie“?

V typickom chápaní „triedy ekvivalencie“ vznikajú na základe špecifikácie, ktorá definuje, aké scenáre a ich alternatívy majú v softvéri existovať. Z pohľadu špecifikácie triedy ekvivalencie zodpovedajú týmto alternatívam. Príprava testov na základe takýchto tried ekvivalencie skôr pripomína testovanie čiernou skrinkou.

Triedy ekvivalencie je spätne možné vytvoriť aj zo štruktúry existujúceho programu, čiže pri testovaní biela skrinka. To už ale rozhodne nie je prípad, že by sme z nich „vychádzali“.

Odpoveď na otázku závisí od toho, akú definíciu pojmu „trieda ekvivalencie“ použijeme. Celkom určite je ale tento termín spojený skôr s technikou čiernej, ako bielej skrinky.

2.4.18

testovanie, biela skrinka, čierna skrinka

Ktorá technika testovania je presnejšia? Biela skrinka vs. čierna skrinka

Žiadna. Účinnosť testovania nezávisí od toho, akú techniku použijeme, ale koľko úsilia do prípravy testov vložíme a ako hlboko do detailov sme ochotní ísť pri stanovovaní testovacích údajov.

Charakter oboch typov testovania by nás pritom mohol zvádzať k odpovedi, že presnejšia je biela skrinka, keďže o softvéri pri nej „vieme viac“, a tým pádom by sme mali lepšie vedieť pripraviť testovacie údaje. Lenže, ak by sme použili *naozaj iba* bielu skrinku, tak síce otestujeme náš súčasný program dobre, ale vôbec si nebudeme istí, či robí to čo má (teda že napĺňa špecifikáciu).

Poznámka: použitie oboch techník zároveň je presnejšie ako ktorákoľvek z nich samostatne.

2.4.19

tdd

Vysvetlite pojem testom riadený vývoj (angl. Test Driven Development, TDD).

Pred vytvorením časti softvéru sa najskôr vytvoria testy, ktoré overia správnosť budúceho nového prírastku.

Tento princíp začína už v skorých etapách vývoja (niektoré testy píšeme už pri špecifikácii - akceptačné) a prechádza všetkými úrovňami granularity softvéru (vopred píšeme testy testujúce celé komponenty aj testy testujúce jednotlivé metódy).

2.4.20

tdd, čierna skrinka

Pri vývoji riadenom testom (test driven development) sa používa technika testovania biela alebo čierna skrinka? Zdôvodnite.

Čierna. Biela potrebuje existujúci program, aby sa dal analyzovať a následne aby sa vytvorili testovacie vstupy. Čierna vychádza zo špecifikácie, tá je k dispozícií od začiatku.

2.4.21

akceptačné testovanie

Čo je to akceptačné testovanie?

Akceptačné testovanie je proces vykonávania a vyhodnocovania akceptačných testov. Akceptačné testy sú konkrétne scenáre a merateľné kritériá, ktoré musí softvér spĺňať, aby zákazník považoval kontrakt vytvorenia softvéru za splnený. Inými slovami, ide o kritériá, na základe ktorých zákazník *akceptuje* softvér a zaplatí zaň.

Najčastejšie akceptačné testy testujú funkcionalitu, teda či softvér robí to, čo je definované v špecifikovaných scenároch (prípadoch použitia). Väčšina akceptačných testov tak nápadne pripomína scenáre v prípadoch použitia (akceptačné testy sú však prísne konkrétne čo sa týka vstupných údajov a očakávaných výstupov).

Menej časté, no stále potrebné sú akceptačné testy nefunkcionálnych požiadaviek. Aj tieto testy sú v konečnom dôsledku scenáre (takmer všetky akceptačné testy sú dynamické), no nie sú nutne odvodené zo scenárov použitia softvéru skôr sa zameriavajú

na to, aby sa požadovaná nefunkcionálna vlastnosť softvéru dala „odmerať“. Napríklad to, že sa pri bezpečnostnom záťažovom teste nezmyselne veľa krát prihlásime alebo zadáme úplne nezmyselné údaje nikto nikdy v scenároch použitia neopisoval.

2.4.22
akceptačné testovanie

Kto vykonáva akceptačné testovanie?

Akceptačné testovanie realizuje vždy zákazník, alebo je vykonávané pod dozorom zákazníka.

2.4.23
akceptačné testovanie, životný cyklus softvéru

Kedy (v ktorej etape vývoja softvéru) sa navrhujú akceptačné testy?

Vo fáze špecifikácie požiadaviek.

Iná možnosť ani nie je, akceptačné testy sú v skutočnosti *kontraktom*, ktorý zákazník uzatvára s vývojárom. Kontrakt sa spravidla uzatvára na začiatku projektu.

2.4.24
akceptačné testovanie, životný cyklus softvéru

Kedy sa v životnom cykle realizuje akceptačné testovanie (kedy sa testy vykonávajú)?

Pri odovzdávaní softvéru do používania.

Nie je samozrejme vylúčené, naopak, je prirodzené aby si vývojári nanečisto prechádzali scenáre akceptačných testov už počas vývoja. Vtedy však o akceptačné testovanie, v zmysle definície ešte nejde.

2.4.25
alfa testovanie, beta testovanie

Čo je to alfa-testovanie a čo je to beta-testovanie?

Oba typy predstavujú špeciálny typ akceptačného testovania, určeného pre generický (krabicový) softvér, teda softvér, ktorý si nikto priamo neobjednal ale je predpoklad že bude po uvedení na trh kupovaný a používaný väčším množstvom používateľov. V prípade absencie objednávateľa totiž niekto iný musí zastúpiť jeho úlohu pri akceptačnom testovaní. Niekto musí povedať, že daný softvér spĺňa potreby *budúcich* zákazníkov.

Pri alfa testovaní túto úlohu plnia vývojári ktorí softvér vyvíjajú. Sami sa snažia vžiť do úlohy používateľov, priebežne hľadajú chyby a odstraňujú ich. Takéto testovanie spravidla trvá dlho (pri iteratívno-inkremetálom vývoji je prakticky neustále). Zhruba 80% nedostatkov softvéru sa odhalí v tejto fáze.

Pri beta testovaní sa softvér nasadí do reálneho sveta a nechá sa testovať ohraničenou skupinou používateľov – vytipovaných potenciálnych zákazníkov, ktorí o svojich skúsenostiach so softvérom podajú správu. Beta testovanie trvá vopred určenú dobu. Zistené nedostatky sa spravidla opravujú až po jeho skončení.

Alfa a beta testovanie, na rozdiel od ostatných akceptačných testov nie je vedené nejakými rigidne stanovenými a zazmluvnenými scenármi. Pri beta testovaní by nakoniec

ani nebolo možné ich uplatniť, pretože používatelia v reálnom svete sa správajú podľa vlastnej vôle. Skôr je snahou dávať testovaniu voľný priebeh, sledovať spokojnosť používateľov a dokonca upravovať softvér podľa ich požiadaviek tak, aby ich lepšie uspokojoval.

2.4.26 Aký typ testov sa používa pri testovaní použiteľnosti softvéru?

[2 - Porozumieť]

Takýmto testom sa jednoducho hovorí „testy použiteľnosti“ alebo „používateľské testovanie“. Podstatné však je, aké vlastnosti má (najmä z pohľadu hľadísk uvedených na začiatku kapitoly, ale nielen tých):

- Vždy ide o testovanie s reálnymi alebo potenciálnymi používateľmi softvéru. Ide tým pádom o manuálne testovanie.
- Takmer vždy ide o dynamické testovanie (softvér musí byť vytvorený a počas testu spustený). „Takmer“ preto, lebo za používateľský test sa teoreticky dá považovať už test dizajnu používateľského rozhrania (nakresleného na kuse papiera), pri ktorom iba necháme používateľa zhodnotiť, čo si o návrhu myslí.
- Z definície ide o nefunkcionálne testovanie (napriek tomu že prebieha na scenároch, teda funkciách softvéru). Testuje totiž nefunkcionálnu vlastnosť softvéru, teda použiteľnosť.
- Úrovňou používateľské testovanie zodpovedá akceptačnému testovaniu, pretože sa testuje softvér ako celok. Pozor: nie každé používateľské testovanie, alebo test použiteľnosti je akceptačným testovaním.

Poznámka: či ide o bielu, čiernu alebo šedú skrinku nemá zmysel uvažovať, nakoľko ide o manuálne testovanie. Najbližšie by však bolo zrejme testovanie čiernou skrinkou (používateľovi je jedno, ako je softvér vnútorne realizovaný a určite na to nebude pri testovaní prihliadať).

2.4.27 Čo je to jednotkové testovanie (unit tests)?

jednotkový test

Ide o automatické, dynamické testovanie softvéru na úrovni jednotlivých metód. Jednotkový test je vlastne malý program, ktorý volaním jednej metódy, v jednej konkrétnej situácií danej vstupnými údajmi overuje jej výstupy.

2.4.28 Kto spravidla vykonáva jednotkové testovanie?

jednotkový test

Vývojári. V prvom rade je to vývojár ktorý kód sám naprogramoval, ale aj iní vývojári, ktorí s kódom pracujú alebo pracujú na kóde, ktorý je na ňom závislý (testy spúšťajú preto, aby sa ubezpečili, že všetko stále funguje správne aj po ich zmenách, kľudne aj v zdanlivo nesúvisiacich moduloch).

2.4.29
integračné testovanie

Čo je to integračné testovanie?

Ide o automatické, dynamické testovanie komponentu softvéru. Podobne ako pri jednotkovom testovaní, môže ísť o volanie metód, avšak nie hocijakej triedy, programového rozhrania komponentu. Ak má komponent používateľské rozhranie, zvykne sa správanie používateľa (používateľské vstupy) emulovať. V obidvoch prípadoch je ale princíp rovnaký: zistiť či pre zadané vstupy vracia komponent požadované výstupy, či už ako návratovú hodnotu metódy alebo správne prejavy používateľského rozhrania.

Poznámka: možno sa čitateľ už zamyslel, kde je hranica medzi jednotkovým a integračným testom? Metódy predsa zvyknú v programoch volať iné metódy (*call stack*, čiže zásobník volaní metód, môže byť aj pri jednoduchších programoch dosť hlboký). Pravdou je, že priestor testov medzi týmito pojmami je (matematici odpustia) spojitý. Striktne by sme možno mohli integračným testom nazývať len testy metód rozhraní komponentov, a všetko ostatné nazvať jednotkovými testami, no celkom určite potom testy mnohonásobne „zložených“ metód nemajú typické vlastnosti jednotkového ale skôr integračného testu. Každopádne, písať testy treba ideálne na všetky metódy a je jedno ako ich nazývame, ak si uvedomujeme ich vlastnosti.

2.4.30
integračné testovanie,
jednotkové testovanie

Uvažujte nasledovné: „Integračné testy by sme mohli pripraviť tak, aby pokrývali celú funkcionalitu príslušného komponentu a ušetrili by sme tým pádom veľa práce s tvorbou (a udržiavaním) jednotkových testov v rámci komponentu. Pre otestovanie komponentu by sme vždy spustili integračné testy a ak sme v poslednej iterácii niečo spravili zle, ukáže sa to.“ Prečo teda nepoužívame iba integračné testovanie, ale aj jednotkové testovanie?

Hoci úvaha o úsilí je platná, proti takejto praxi sú dôležitejšie dôvody:

1. Vytvoriť integračné testy zvyčajne vieme len pomocou prístupu čierna skrinka. Zapojenie metód bielej skrinky je pre veľký komponent resp. jeho vnútornú zložitosť príliš náročné. Lenže aj pre čiernu skrinku je veľký komponent rovnakým problémom: s narastajúcou zložitosťou sa čoraz ťažšie dajú predpovedať triedy ekvivalencie. Z tohto vyplýva, že len samotné integračné testy nikdy nebudú mať schopnosť dostatočne strážiť pred niektorými chybami. Preto si radšej v zmysle hesla „rozdeľuj a panuj“, dekomponujeme komponent na menšie časti a testujeme ich jednotlivo (jednotkovo), pretože menšie časti vieme ľahšie bielo-skrinkovým prístupom analyzovať.

2. Integračné testy, keďže spúšťajú programy v celej svojej zložitosti, spravidla trvajú dlho (špeciálne ak zahŕňajú emulovanie správania používateľa na používateľskom rozhraní). A ešte dlhšie by trvali, ak by sme sa predsa len pokúsili prekonať problém z bodu 1 a naozaj poctivo a dôsledne by sme sa snažili prístupom bielej skrinky nadefinovať čo najviac sád testovacích údajov. Reálne by nám ich počet prerástol cez hlavu *veľmi rýchlo*: veď pridanie čo i len jednej podmienky v programe v skutočnosti znamená zdvojnásobenie všetkých možností, ktoré treba preskúmať. Rozdelením na menšie časti v rámci jednotkového testovania opäť tento problém zmierňujeme.

3. S jednotkovými testami vieme lepšie lokalizovať, kde nastali problémy v dôsledku nedávnych zmien v softvéri. Ak by sme používali len integračné testy, po vykonaní zmien v komponente by sme sa síce po ich spustení dozvedeli, že sme niečo pokazili, no nevedeli by sme presne hneď, kde.

Poznámka: integračné testy sú samozrejme stále dôležité. Len s nimi môžeme preveriť, či jednotlivé súčasti komponentu spolu správne „hrajú“.

2.4.31
jednotkový test

Ako by sme mali postupovať pri tvorení jednotkových testov pri zložených metódach (metódach, ktoré volajú iné metódy)?

V predchádzajúcich otázkach sme spomenuli, že ak chceme vytvoriť testovacie údaje pre príliš zložitú časť softvéru, dostaneme sa pred problém exponenciálne narastajúceho množstva testovacích údajov, ktoré by sme potrebovali na pokrytie všetkých alternatív.

Zložené metódy, teda metódy volajúce vo svojom tele nejaké iné metódy sú zložitou časťou softvéru. Ako teda pre také metódy písať testy?

Skúsme začať od najjednoduchšieho príkladu: majme metódu, ktorá žiadne ďalšie metódy nevolá, iba vykonáva nejaké aritmetické operácie, napríklad počíta sumu čísiel v zozname. Pri písaní testov pre túto metódu by sme sa zrejme cítili komfortne, je to jednoduchá metóda.

Potom by nám však niekto povedal, že vlastne ide o zloženú metódu, pretože v nej voláme iné metódy. Je to pravda, aj aritmetické operácie sú predsa metódami, iba prostredie jazyka v ktorom programujeme nám ich ponúka s menej typickou syntaxou. Znervózneli by sme ale, vediac zrazu, že máme dočinenia so zloženou metódou? Nie. Ani by sme horúčkovito nezačali pripravovať ďalšie sady údajov, ktoré by overovali nielen logiku nášho kódu ale aj logiku metód, ktoré voláme. Prečo? Lebo dôverujeme tomu, že oné volané metódy sú naimplementované správne.

Rovnako ako v tomto prípade, by sme mali postupovať aj v prípade skladania s použitím metód, ktoré sme sami naprogramovali. Mali by sme im dôverovať a ich dôveryhodnosť podložiť ich vlastnými testami. Test zloženej metódy by sa tak mal zameriavať len na logiku nej samotnej a mal by veriť, že volané metódy si svoje potenciálne problémy vyriešia vo vlastných testoch.

2.5 Údržba

2.5.1 Čo je to údržba softvéru?

Údržba

Modifikácia softvérového produktu po jeho dodaní (odovzdaní) zákazníkovi kvôli oprave chýb, zlepšeniu výkonu alebo iných vlastností [6]. Údržba softvéru typicky zahŕňa opravu chýb, pridanie nových funkcií, odstránenie nadbytočných funkcií alebo optimalizáciu v už nasadenom/odovzdanom softvéri. V literatúre bolo identifikovaných až 21 generických činností, ktoré spadajú do údržby softvéru [7]. V žiadnom prípade nejde len o odstraňovanie chýb, ako sa mnohí mylne domnievajú.

Z pohľadu životného cyklu softvéru chápeme údržbu softvéru ako jeho samostatnú etapu, ktorá nasleduje po vývoji softvéru (jeho odovzdaní do prevádzky) a predchádza jeho vyradeniu.

S údržbou softvéru súvisí aj jedna z jeho dôležitých nefunkcionálnych vlastností: udržiavateľnosť. Ide o schopnosť softvérového produktu byť zmenený (pozri otázku 52).

2.5.2 Aké typy údržby softvéru poznáme?

Údržba

Údržbu môžeme rozdeliť podľa toho, či sa v rámci nej vykonávajú opravy alebo rozšírenia pôvodného softvéru. Ďalej môžeme údržbu rozdeliť na reaktívnu a proaktívnu. Kombináciou týchto pohľadov získame štyri hlavné typy [6]:

1. korektívna údržba (angl. corrective maintenance) – modifikácia softvérového produktu po jeho odovzdaní s cieľom odstránenia (vnútorných) problémov softvéru, ktoré bránia jeho riadnemu používaniu.
2. adaptívna údržba (angl. adaptive maintenance) – modifikácia softvérového produktu po jeho odovzdaní s cieľom udržať softvérový produkt použiteľný kvôli meniacemu sa (alebo už zmenenému) prostrediu. Zmena sa môže týkať hardvéru či platformy, na ktorých produkt beží (napr. vyjde nová verzia operačného systému, na ktorom je produkt používaný).
3. perfektívna údržba (angl. perfective maintenance) – modifikácia softvérového produktu po jeho odovzdaní s cieľom rozšíriť funkcionalitu, zlepšiť dokumentáciu, zvýšiť výkonnosť softvéru alebo inak vylepšiť iné atribúty softvérového produktu (napr. zákazník zistil, že vo svojom elektronickom obchode chce podporovať platbu bitcoinami).
4. preventívna údržba (angl. preventive maintenance) – modifikácia softvérového produktu po jeho odovzdaní s cieľom odhaliť a odstrániť *skryté* chyby predtým, ako sa ukážu v reálnej prevádzke.

V rámci korektívnej a preventívnej údržby sa snažíme *opraviť* (prejavené alebo skryté chyby), pri adaptívnej a perfektívnej vytvárame *rozširujeme* pôvodný softvér. Korektívna a adaptívna údržba spadajú do kategórie *reaktívnej* údržby (reagujú na vzniknuté okolnosti, podnety), perfektívna a preventívna do kategórie *proaktívnej* údržby (na vykonanie nepotrebujú podnety).

2.5.3
Údržba

Prečo je údržba softvéru dôležitá? (Prečo je dôležité sa ňou zaoberať? Líšia sa aktivity počas údržby od typického vývoja softvéru?)

Zmena v softvéri je nevyhnutná a neodvratná. Je súčasťou samotnej podstaty softvéru, priamo vyplýva z dvoch jeho *kľúčových vlastností*: podriadenosti a menlivosti. Mení sa náš svet, mení sa aj softvér, ktorý mu slúži. Logicky: čím viac softvéru na svete je, tým väčšie úsilie treba venovať jeho údržbe. Dalo by sa povedať, že hlavným cieľom údržby softvéru je zachovanie hodnoty softvéru v čase.

Už na začiatku tohto tisícročia sa viac ako 50 % celkovej softvérovej „populácie“ zaoberalo viac modifikovaním existujúceho softvéru ako vytváraním nového [7]. Tento podiel sa zvyšuje ako priamy dôsledok stále narastajúceho množstva softvéru používaného vo svete. Celkový pomer nákladov na údržbu v porovnaní s nákladmi na vývoj môže byť až 4:1 [8]. .

Tie môžeme do určitej miery znížiť, najmä dôrazom na kvalitu softvéru. I keď je to veľmi náročné na odhad, podiel korektívnej údržby (ktorá je často zodpovednosťou tvorcov softvéru a vieme ju najľahšie ovplyvniť), môže okolo 20 % z celkových nákladov na údržbu [5]. Toto je dôležité uvedomiť si, ak si spomenieme na vzťah nákladov na opravu chyby a etapy vývoja softvéru, v ktorom bola nájdená.

Údržba softvéru sa na životnom cykle softvéru podieľa obrovským dielom a vyžaduje veľa nielen ľudského, ale aj finančného kapitálu. Preto je systematický prístup k údržbe softvéru veľmi dôležitý. A to jednak z pohľadu technického (samotná zmena softvéru) a jednak z pohľadu manažmentu vývoja softvéru (riadenie zmien v rámci softvérového projektu; *manažment zmien* je dokonca samostatne vyčlenená oblasť projektového manažmentu).

V rámci údržby účastníci vývoja softvéru často čelia *novým* výzvam/úlohám, ktoré v predošlých etapách neriešili: napr. hľadanie chýb v kóde (často vytvorenom inými vývojármi), tvorba tzv. pohotovostných záplat, či – z pohľadu manažmentu – plánovanie opravných vydaní softvéru alebo odhadovanie ceny za údržbu, ktoré v prípade niektorých softvérových produktov môžu viesť k vzniku samostatných jednotiek údržby – tzv. help-desk [1]. Nezriedka sa v praxi na údržbe softvérového produktu podieľajú iné tímy ako tie, ktoré produkt vyvíjali.

2.5.4
vývoj softvéru, Údržba softvéru

Aký je pomer nákladov na vývoj a na údržbu softvérového systému?

Veľa ľudí si neuvedomuje, že náklady na údržbu softvéru môžu byť niekedy až 4-krát vyššie.

Údržba softvéru zahŕňa odstraňovanie chýb, ktoré sa odhalili *po* odovzdaní softvérového systému, rozšírenie služieb systému na základe novoidentifikovaných požiadaviek či vylepšenie softvéru kvôli zlepšeniu jeho škálovateľnosti pre narastajúci počet používateľov alebo kvôli zabezpečeniu lepšej bezpečnosti. Nakoľko je zmena v softvérovom inžinierstve neodvratná, náklady spojené s údržbou softvéru môžu narásť veľmi rýchlo.

Možno povedať, že v súčasnosti sú náklady na údržbu priemerného softvérového systému vyššie ako náklady na vývoj. Pomer nákladov na vývoj a na údržbu môže byť až niekde medzi 40:60 a 20:80 [10, 5]. Zaujímavá je aj historická perspektíva tohto pomeru. V ranom období softvérového inžinierstva totižto náklady na údržbu boli omnoho nižšie a nedosahovali výšku nákladov na vývoj [8].

Agilná metodológia vývoja implicitne redukuje riziká potreby korektívnej – stále fungujúci softvér – a perfektívnej údržby – vďaka zahrnutiu zákazníka do procesu vývoja klesá pravdepodobnosť objavenia novoidentifikovaných požiadaviek na softvér po odovzdaní. Na druhej strane, aplikovanie agilnej metodológie vo fáze údržby môže byť niekedy náročné. Niektoré techniky sú veľmi ťažko použiteľné, napr. fixná dĺžka šprintu (chybu je potrebné odstrániť hneď), *refaktoring* kódu medzi jednotlivými iteráciami (často nemožné pri pohotovostnej oprave), a pod. Podmienky sú ešte náročnejšie, ak tím pracuje na cudzom kóde, príp. ak ide o údržbu nejakého zastaraného systému.

2.5.5 Čo je to tzv. „odkázaný“ systém (angl. legacy system)?

legacy, údržba

Odkázaný systém (angl. legacy system) je označenie existujúceho, väčšinou zastaraného systému, ktorý je používaný zákazníkom a nezriedka kľúčový pre jeho biznis [12], str. 245, kap. 9.4. Príkladom takéhoto softvéru je jadro bankového systému alebo evidenčný systém obyvateľstva. Ich výmena by predstavovala veľké riziko ohrozujúce biznis alebo fungovanie iných softvérových systémov.

Základnými charakteristikami odkázaných systémov sú vysoký stupeň znehodnotenia a slabá až žiadna zdokumentovanosť. Takéto systémy zvyčajne prešli rukami viacerých tímov softvérových inžinierov. Mohli byť často modifikované a sú nesystematicky „pozliepané“. Sú založené na zastaraných technológiách či komponentoch, ktorých vývoj už skončil, a nemajú žiadnu podporu. Nezriedka tieto systémy vnímame len ako *čiernu skrinku*.

Tieto systémy majú kriticky slabú udržiavateľnosť, ktorá je problémom pre nové vyvíjané softvérové systémy, ktoré s nimi spolupracujú/interoperujú.

2.5.6 Prečo majú tzv. „odkázané“ systémy slabú udržiavateľnosť?

legacy, údržba

Existuje hneď niekoľko dôvodov, prečo majú „odkázané“ systémy slabú udžiavateľnosť, resp. prečo sú náklady na udržiavanie špeciálne vysoké [11]:

1. Rôzne časti systému boli implementované rôznymi tímami. Štýl programovania nie je konzistentný naprieč systémom.
2. Časti systému môžu byť implementované v zastaranom, neudržiavanom jazyku. Môže byť náročné nájsť personál, ktorý pozná tento jazyk a zároveň drahé, ak je potrebné získať takýchto ľudí prostredníctvom tzv. outsourcingu.
3. Dokumentácia je často neprimeraná a stará. Nezriedka jedinou dokumentáciou je samotný zdrojový kód. V hraničných prípadoch môže byť dokonca stratený

a len skompilovaná/vykonateľná verzia systému je dostupná.

4. Predošlé roky údržby narušili štruktúru systému, čo sťažuje jeho pochopiteľnosť. Špeciálne to platí pre zmeny, ktoré boli vykonané kvôli optimalizácii výkonu. Nové súčasti mohli byť pridané a prepojené ad hoc.
5. Dáta spracovávané systémom môžu byť udržiavané v rôznych typoch, vo formátoch s nekompatibilnou štruktúrou, môžu byť duplikované, nekompletné či inak zastarané.

Údržba odkázaných systémov je veľmi nákladná. Pri vývoji nového softvéru treba vyhodnotiť, či má vôbec zmysel odkázaný systém ďalej udržiavať, alebo ho prerobiť či dokonca úplne vymeniť.

2.5.7 Čo je to refaktoring kódu?

refaktoring

Zmena štruktúry kódu bez zmeny jeho vonkajšieho správania. Je to proces vylepšovania programu pre spomalenie jeho degradácie vyplývajúcej z čelenia (nevyhnutnej a neodvratnej) zmene. Refaktoring nepridáva do programového kódu žiadnu novú funkcionalitu, tá sa vôbec nemení. Mení sa len štruktúra kódu s cieľom zlepšenia jeho čitateľnosti a pochopiteľnosti.

Niekoľko príkladov techník refaktoringu [9]:

- premenovanie metódy (alebo atribútu triedy), aby mala výstižnejší názov,
- extrakcia časti existujúcej triedy do samostatnej triedy,
- zapuzdrenie atribútu pre donútenie prístupu k nemu cez tzv. mutátory (metódy typu get, set),
- rozdelenie pridlhej metódy na dve, príp. viac častí.

Refaktoring sa najčastejšie využíva pri objektovo orientovanom programovaní, ale jeho princípy môžu byť uplatnené aj pri iných prístupoch.

Refaktoring môžeme považovať za techniku *preventívnej údržby* softvéru.

2.5.8
kultivované programovanie

Vysvetlite (stručne) pojem kultivované programovanie (angl. literate programming).

Vytváranie zdrojového kódu aj dokumentácie jedným nástrojom, na jednom mieste. Vytvára sa tým dobre čitateľný kód.

Zdroje

[1] Alain Abran et al., eds. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001. ISBN: 0769510000.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154959.

[3] *Examples for Software Robustness Requirements*. 2005. URL: `http://www.it-checklists.com/Examples_Robustness_Requirements.html`.

[4] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[5] R.L. Glass. "Frequently forgotten fundamental facts about software engineering". In: *Software, IEEE* 18.3 (2001), pp. 112–111. ISSN: 0740-7459. DOI: `10.1109/MS.2001.922739`.

[6] ISO. *Software Engineering – Software Life Cycle Processes – Maintenance*. ISO/IEC 14764:2006. Geneva, Switzerland: International Organization for Standardization, 2006.

[7] Capers Jones. *The Economics of Software Maintenance in the Twenty First Century*. 2006.

[8] Jussi Koskinen. *Software Maintenance Costs*. 2010. URL: `http://web.archive.org/web/20120803081704/http://users.jyu.fi/~koskinen/smcosts.htm`.

[9] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[10] R. Schach. *Software Engineering, Fourth Edition*. McGraw-Hill, 1999, p. 11.

[11] Ian Sommerville. *Increasing costs of legacy system maintenance*. 2008. URL: `https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/LegacySys/Costs.html`.

[12] Ian Sommerville. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010. ISBN: 978-0-13-703515-1.

Kapitola 3

Modely životného cyklu vývoja softvéru

3.0.1
model životného cyklu vývoja softvéru

Vysvetlite, čo je to model životného cyklu vývoja softvéru.

Model životného cyklu vývoja softvéru (ďalej len model vývoja softvéru) je schéma definujúca procesy/činnosti, ktoré treba vykonať počas životného cyklu vývoja softvéru, ich vstupy a výstupy a časovú následnosť. Model vývoja softvéru *neurčuje* dĺžku trvania činností.

Modely vývoja softvéru vznikali ako zovšeobecnenie procesov/činností, ktoré boli realizované v rôznych typoch softvérových projektov. Rôzne modely vývoja softvéru vznikli z dôvodu dosiahnutia odlišných cieľov, ktoré mal ten-ktorý softvér napĺňať. Veľmi dôležitá je aj historická perspektíva, keď novšie modely vývoja typicky reagujú na nedostatky starších a adresujú aktuálne/súčasné problémy vývoja softvéru.

Modely vývoja softvéru môžeme rozdeliť na lineárne a inkrementálno-iteratívne [1]. Pri lineárnych modeloch sú jednotlivé etapy životného cyklu softvéru usporiadané sekvenčne – jedna začína vždy potom, keď skončí predošlá. Pri inkrementálnych modeloch je softvér vyvíjaný v menších cykloch, výstupom ktorých sú softvérové inkrementy – prírastky softvéru. Hranica medzi oboma typmi modelov nie je striktná a v praxi sú často zaužívané modely, ktoré obsahujú črty oboch typov.

Medzi najznámejšie modely životného cyklu vývoja softvéru patria:

- Vodopádový model,
- V-model,
- Inkrementálny model,
- Iteratívny model,
- Evolučný model,

- Špirálový model,
- Agilný model

Vodopádový model a V-model sa zaraďujú medzi lineárne modely, ostatné modely sú vo väčšej či menšej miere považované za inkrementálno-iteratívne.

Rozlišujúcou črtou modelov životného cyklu vývoja softvéru je spôsob prístupu k riadeniu používateľských požiadaviek. Lineárne modely opisujú tvorbu celej množiny používateľských požiadaviek na začiatku životného cyklu softvéru. Ich rozsah je dôsledne kontrolovaný a dodatočné požiadavky sú vnímané už ako zmeny softvéru. Zmeny v softvéri sú riešené (vyžiadané, vyhodnocované, schvaľované) v rámci tzv. manažmentu zmien (tam je často zapojený aj vyšší manažment, takéto zmeny predstavujú dodatočné náklady pre zákazníka). Inkrementálno-iteratívne modely opisujú produkciu inkrementov softvéru, ktoré bývajú predmetom priebežného vyhodnocovania zákazníkom a so zmenami v softvéri pracujú omnoho voľnejšie.

Modely vývoja softvéru sa často odlišujú aj na základe etapy testovania softvéru, tj. podľa toho čo, kedy a ako sa testuje. Etapa testovania má významný vplyv na celkovú kvalitu vyvíjaného softvéru.

V rámci modelovania životného cyklu softvéru niekedy rozlišujeme medzi životným cyklom vývoja softvéru (angl. software development life cycle, SDLC) a životným cyklom softvérového produktu (angl. software product life cycle, SPLC) [1]. V prvom prípade sa typicky neuvažujú etapy, ktoré nasledujú po vývoji plne fungujúceho softvérového produktu, ako napr. údržba softvéru či jeho vyradenie, v tom druhom áno.

3.0.2

model životného cyklu softvéru

Čo hovoria modely životného cyklu vývoja softvéru o dĺžke trvania jednotlivých etáp?

Model životného cyklu softvéru nedefinuje dĺžku trvania jednotlivých etáp, iba ich poradie.

3.0.3

model životného cyklu softvéru, vodopádový model

Ktorý model životného cyklu vývoja softvéru považujeme za referenčný?

Za referenčný model považujeme vodopádový model. Patrí medzi historicky najstaršie a je najjednoduchší na pochopenie, pretože základné etapy vývoja softvéru sú jasne oddelené a zoradené sekvenčne za sebou.

3.0.4

model životného cyklu softvéru, vodopádový model

Vysvetlite základný princíp vodopádového modelu vývoja softvéru.

Ďalšia etapa životného cyklu softvéru začne vždy až po tom, ako sa predchádzajúca skončila.

3.0.5

model životného cyklu softvéru, vodopádový model

Aké sú výhody vodopádového modelu vývoja softvéru?

Výhodami sú jednoduchosť a ľahká pochopiteľnosť. Jednotlivé etapy sa neprekrývajú a neprebiehajú súbežne. Majú presne definované výstupy a podmienky ukončenia, čím je proces vývoja softvéru viditeľný a ľahko manažovateľný. Tento model je vhodný len pre malé projekty s veľmi dobre definovanými požiadavkami na softvérový produkt.

3.0.6

model životného cyklu softvéru, vodopádový model

Čo je základnou nevýhodou vodopádového modelu vývoja softvéru?

Príliš neskoro vidíme výsledok (fungujúci softvér). Model nie je dobre pripravený na zmenu požiadaviek – veľmi ťažko pracujeme s akýmikoľvek zmenami v neskorších etapách, treba prepracovať výstupy z predošlých etáp. Cena za zmeny v neskorších etapách je vysoká. K testovaniu softvéru dochádza až na konci, oprava väčších chýb býva často drahá.

3.0.7

model životného cyklu softvéru, v-model

Vysvetlite princíp V-modelu životného cyklu softvéru.

V-model je vo svojej podstate určitým vylepšením vodopádového modelu. Základným princípom je rozdelenie životného cyklu na dve fázy: fázu vývoja (v užšom zmysle) a fázu testovania. Jednotlivé etapy životného cyklu sú podobne ako pri vodopádovom modeli vykonávané sekvenčne, pričom model rozlišuje viacero etáp testovania (pozri obr. 3.1). Etapy vývoja a etapy testovania sú navzájom prepojené, každá z etáp testovania má priradenú etapu vývoja, ktorej výstupy sú v nej testované. Pozor – plán a návrh samotných testov sa vytvára vo fáze vývoja.

Verifikačno-validačný charakter tohto modelu životného cyklu softvéru mu s spolu s tvarom zoradenia jednotlivých etáp dal názov – V-model.

3.0.8

model životného cyklu softvéru, v-model

Aký druh testovania je vo V-modeli prepojený na fázu analýzy používateľských požiadaviek (a zároveň uzatvára životný cyklus softvéru)?

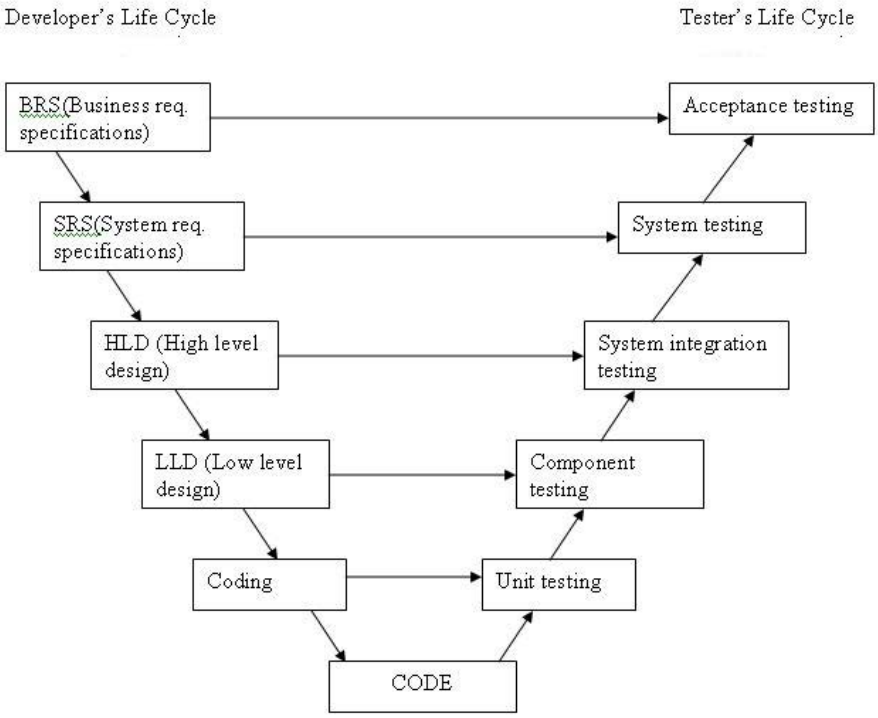
Akceptačné testovanie. V tejto etape sa overuje, či vytvorený softvérový systém naozaj spĺňa požiadavky zákazníka na základe vykonania *akceptačných testov*. Tie vykonáva samotný zákazník, resp. používateľ systému v takmer produkčnom prostredí.

3.0.9

model životného cyklu softvéru, inkrementálny model

V čom spočíva inkrementálny model vývoja softvéru? Prečo/Kedy sa používa?

Jednotlivé etapy životného cyklu softvéru vedú k tvorbe malých prírastkov (inkrementov) softvéru. Celkový softvér je vytváraný postupným pridávaním plne dokončených prírastkov, až kým nie je kompletný (pozri obr. 3.2). Prírastky sú ľahšie manažovateľné menšie časti softvéru. Na vytvorenie jedného inkrementu sa najčastejšie využíva

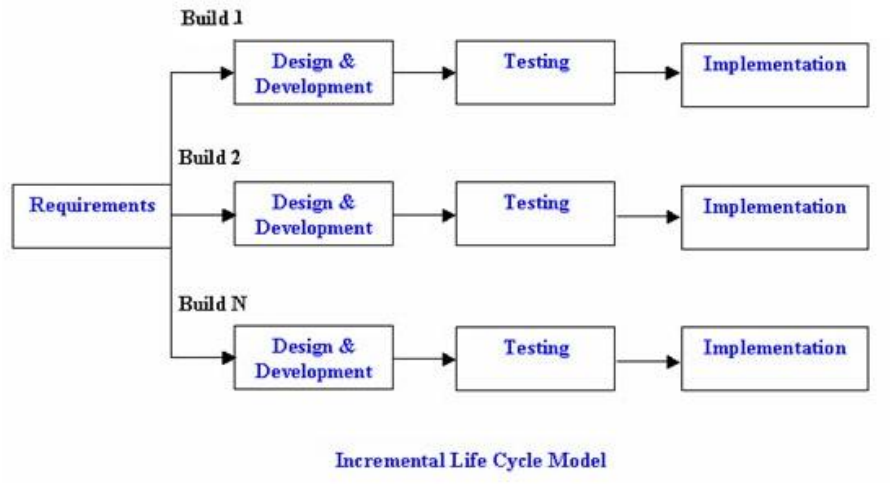


Obr. 3.1: Schéma V-modelu

vodopádový model, ktorý je vhodný na tvorbu softvéru malého rozsahu. V procese vývoja sa pridávajú vždy nové malé časti, ide o tzv. *add* princíp.

Použitie tohto modelu je vhodné ak vytvárame softvér, kde je na začiatku dobre zadefinovaná a jasne pochopená väčšina požiadaviek na systém (menšie zmeny sú možné). Tento model pomáha znížiť riziko neúspechu projektu kvôli vysokej cene za zmeny – v porovnaní s lineárnymi modelmi životného cyklu dokážeme ľahšie pracovať so zmenami požiadaviek. Zákazník má možnosť priebežne vidieť fungujúce časti softvéru.

Takéto vlastnosti modelu sú vhodné, ak sa vytvára produkt, ktorý je potrebné dostať na trh čo najskôr, ak sa pri vývoji pracuje s novou menej známa technológia alebo ak nie sú hneď na začiatku dostupné zdroje na tvorbu všetkých častí systému.



Obr. 3.2: Schéma inkrementálneho modelu

3.0.10

model životného cyklu softvéru, inkrementálny model

Porovnajte výhody a nevýhody inkrementálneho modelu vývoja softvéru (voči vodopádovému modelu).

Výhody: Využitie tohto modelu vedie k vždy fungujúcemu softvéru (aj keď len jeho častí). Zákazník sa môže vyjadriť ku každej novej verzii, čo môže pomôcť odhaliť nedostatky existujúcich požiadaviek. Vývoj na základe tohto modelu je flexibilný a dokáže lepšie reagovať na zmeny. Manažment rizík je jednoduchší. Kontrola nad menšou časťou softvérom z pohľadu testovania a odstraňovania chýb je lepšia. Cena prvotného dodania produktu zákazníkovi je nižšia.

Nevýhody: Identifikácia všetkých požiadaviek na začiatku je náročná. Nároky na dobré plánovanie a dobrú dekompozíciu systému sú vyššie. V porovnaní s vodopádovým modelom je celková cena vývoja vyššia.

3.0.11

model životného cyklu softvéru, inkrementálny model

Uveďte príklady modelov životného cyklu softvéru založených na inkrementálnom modeli vývoja softvéru.

RAD model, agilný model. Čiastočne z neho vychádza aj špirálový model.

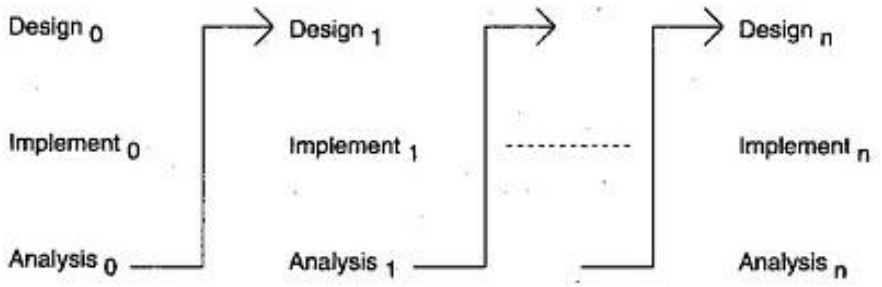
3.0.12

model životného cyklu softvéru, iteratívny model

Aká je základná charakteristika iteratívneho prístupu k vývoju softvéru? Prečo/Kedy sa používa?

Jednotlivé etapy vývoja softvéru vedú k postupne vylepšovanému softvéru. Opakujú sa cyklicky (iteratívne) tak, že sa postupne vylepšuje celý softvér – opakujú sa postupne jednotlivé etapy vývoja softvéru, najčastejšie podľa vodopádového modelu, na celom softvéri (pozri obr. 3.3). V prvej iterácii sa začína s hrubou špecifikáciou po-

žiadaviek, hrubým návrhom riešenia a jeho implementáciou, v ďalších vedú všetky etapy k lepším, podrobnejším výstupom. Jednotlivé časti softvéru sa postupne prerábajú , ide o tzv. *redo* princíp.



Obr. 3.3: Schéma iteratívneho modelu

Tento model je vhodný, keď je známych a jasných veľa požiadaviek na softvér (ďalšie zmeny však sú možné vo väčšej miere ako pri inkrementálnom modeli). Je veľmi výhodný, keď je vytváraný softvérový produkt veľký.

3.0.13
model životného cyklu softvéru, iteratívny model

Porovnajte výhody a nevýhody iteratívneho modelu vývoja softvéru (voči vodopádovému modelu?).

Výhody: Zlepšovanie produktu krok za krokom. Možnosť vytvorenia počiatočnej zjednodušenej verzie produktu. Zákazník môže poskytovať priebežnú spätnú väzbu, čo môže pomôcť redukovať nepochopené požiadavky už v začiatočnej fáze vývoja. Potenciál skorého odhalenia chýb, bez nutnosti opakovania už raz vykonaných procesov vývoja (návrh, implementácia, testovanie) alebo spätného negatívneho ovplyvnenia už hotových častí (napr. aj vrátane dokumentácie).

Nevýhody: Jednotlivé iterácie sa nedajú ľahko presne zadefinovať, procesy nemusia byť dobre viditeľné. Keďže nemusia byť všetky požiadavky známe na začiatku, v neskorších iteráciách sa môže ukázať potreba zmeny architektúry alebo veľkej časti návrhu, čo môže predražiť celkový vývoj.

3.0.14
model životného cyklu softvéru, evolučný model

Aká je základná charakteristika evolučného modelu vývoja softvéru?

Jednotlivé etapy vývoja softvéru sú realizované s dôrazom na rýchle dodanie verzie s vysokou hodnotou pre zákazníka a kontinuálne získavanie a využívanie jeho spätnej väzby.

Evolučný model sa považuje sa špeciály prípad iteratívneho modelu vývoja, kde výstupy jednotlivých etáp tvorby softvéru (požiadavky, plán, odhady a samotné riešenie) sa jednotlivými iteráciami postupne vyvíjajú a vylepšujú z prvotného prototypu.

Evolučný model je vhodný pre vývoj softvéru, kde je prítomná neistota v požiadavkách (napr. nepoznáme všetky požiadavky) a môže dôjsť k nepredvídateľným zmenám.

3.0.15
model životného cyklu softvéru

Podľa akých dvoch kritérií by sme sa mali rozhodovať medzi evolučným a „grand design“ prístupom k vývoju softvéru?

Naša znalosť domény a veľkosť projektu.

3.0.16
model životného cyklu softvéru, špirálový model

Porovnajte výhody a nevýhody špirálového modelu vývoja softvéru.

Výhody: Zo svojej podstaty vývoj softvéru na báze tohto modelu redukuje výskyt škodlivých udalostí (ako dôsledok analýzy rizík). Je vhodný najmä pre veľké projekty. Jeho iteratívno-inkrementálny charakter umožňuje rozširovanie funkcionality softvéru počas vývoja.

Nevýhody: Je nevhodný pre malé projekty. Analýza a vyhodnocovanie rizík často vyžaduje experta. Vývoj na základe tohto modelu je relatívne drahý.

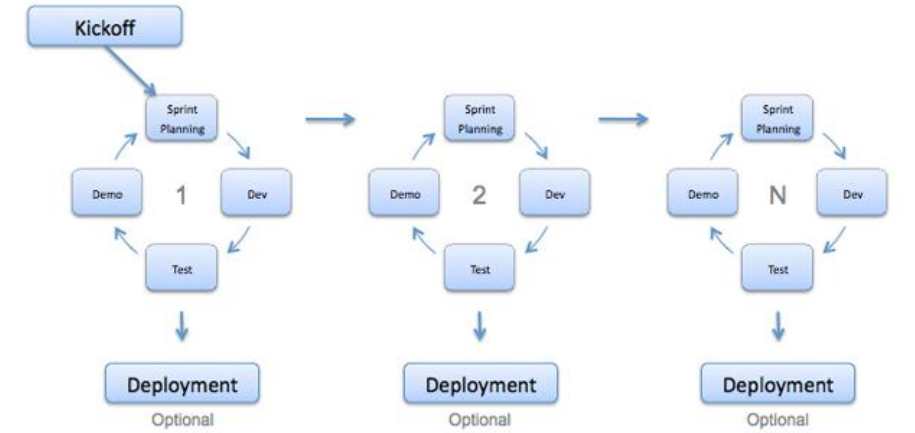
3.0.17
model životného cyklu softvéru, agilný model

Čo je najväčší prínos agilného modelu vývoja oproti všeobecnému iteratívno-inkrementálnemu modelu?

Vykonávané iterácie (cykly) sú *rýchle/svižné*, typicky časovo ohraničené a vedú k *malým* a *častým* prírastkom. Kladie sa veľký dôraz na kvalitu, dosahovanú najmä skrz zodpovedný prístup k testovaniu. Do vývoja je priamo zakomponovaný zákazník, ktorý má úlohy vo viacerých etapách vývoja, je integrálnou súčasťou modelu.

V rámci vývoja sa celkovo kladie väčší dôraz na ľudí a interakcie medzi nimi ako na procesy a nástroje, fungujúci softvér je dôležitejší ako rozsiahla dokumentácia, spolupráca so zákazníkom má prednosť pred dohadovaním kontraktu a rýchla reakcia na zmenu má väčšiu hodnotu ako postupovanie podľa plánu. Tieto kritériá boli sformulované v tzv. agilnom manifeste [2]. Agilný model tak priamo ovplyvňuje aj manažment softvérového projektu, a to najmä na úrovni komunikácie (v rámci softvérových tímov, so zákazníkom).

Je dôležité uvedomiť si historickú perspektívu vzniku agilných metód, ktoré na konci minulého storočia boli reakciou na ťažkopádny, zložito regulovaný prístup k vývoju softvéru s veľkým podielom manažmentu reprezentovaný vodopádovým modelom. Tieto vlastnosti sa snažia agilné metódy redukovať alebo úplne odstrániť.



Obr. 3.4: Priebeh projektu s agilným modelom

3.0.18

model životného cyklu softvéru, agilný model

Porovnajte výhody a nevýhody agilného modelu vývoja softvéru.

Výhody: Vyššia spokojnosť zákazníka vďaka kontinuálnemu dohľadu nad odovzdávaným softvérom. Stále fungujúci softvér s častými prírastkami. Neustále vylepšovanie vedie k technickej kvalite riešenia. Jednoduchá reakcia na zmenu požiadaviek a ľahké prispôsobenie na zmenu podmienok vývoja.

Nevýhody: Náročné odhadovanie úsilia na začiatku vývojového cyklu, obzvlášť pre veľké časti (črty) softvéru. Niekedy je problémom slabý dôraz na rozpracovanie návrhu a dokumentáciu. Softvér je závislý od zástupcu zákazníka, ak nie je dostatočne zapájaný, príp. ak nemá jasnú predstavu o výslednom softvéri, výsledok nemusí byť uspokojivý. Potreba skúsených pracovníkov v agilných tímoch kvôli náročným rozhodnutiam, ktoré treba vykonávať počas procesu vývoja.

3.0.19

model životného cyklu softvéru, inkrementálny model, iteratívny model, agilný model

Vysvetlite rozdiel medzi inkrementálnym, iteratívnym a agilným modelom vývoja softvéru z pohľadu stavu fungujúceho softvéru.

Rozdiely demonštruje obr. 3.5. Hlavný rozdiel je v aktuálnom 1) rozsahu čŕt vytváraného softvéru a 2) ich hodnovernosti (úplnosti voči tomu, čo zákazník požaduje). Jednotlivé grafy predstavujú jednotlivé vydania softvéru (angl. release). Vtedy softvér uvidí a hodnotí/pripomienkuje zákazník. Inkrementálny model vedie k vydaniam, ktoré obsahujú stále nové črty s vysokou hodnovernosťou (úplné). Iteratívny model vedie k vydaniam, ktoré pokrývajú všetky črty, avšak s jednotlivými vydaniami rastie hodnovernosť každej z nich. Agilný model kombinuje prvky oboch predošlých. Všimnime si, že pri agilnom modeli je a počet vydaní softvéru najvyšší (v skutočnosti je relatívne ešte vyšší, ako demonštruje obrázok). Pre porovnanie, pri použití vodopádového modelu máme fungujúci softvér až na konci. Vtedy softvér prvýkrát uvidí aj zákazník. Aké sú dopady, keď zistíme, že nejaká požiadavka bola zle pochopená a

črta je implementovaná zbytočne, sme opísali v predchádzajúcich otázkach.

Poznámka: Na obrázku horizontálna os pri jednotlivých modeloch nereprezuntuje rovnakú absolútnu dĺžku času (napr. agilným modelom možno dosiahnuť rýchlejšie odovzdanie finálneho produktu). Výsledkom využitia agilného modelu môže byť tiež skutočnosť, že miera hodnovernosti niektorých čŕt nemusí byť oproti pôvodnej predstave zákazníka vysoká/úplná, pričom produkt už bude finálny. Ten si to môže uvedomiť vďaka často predvádzanému softvéru).

3.0.20
model životného cyklu softvéru

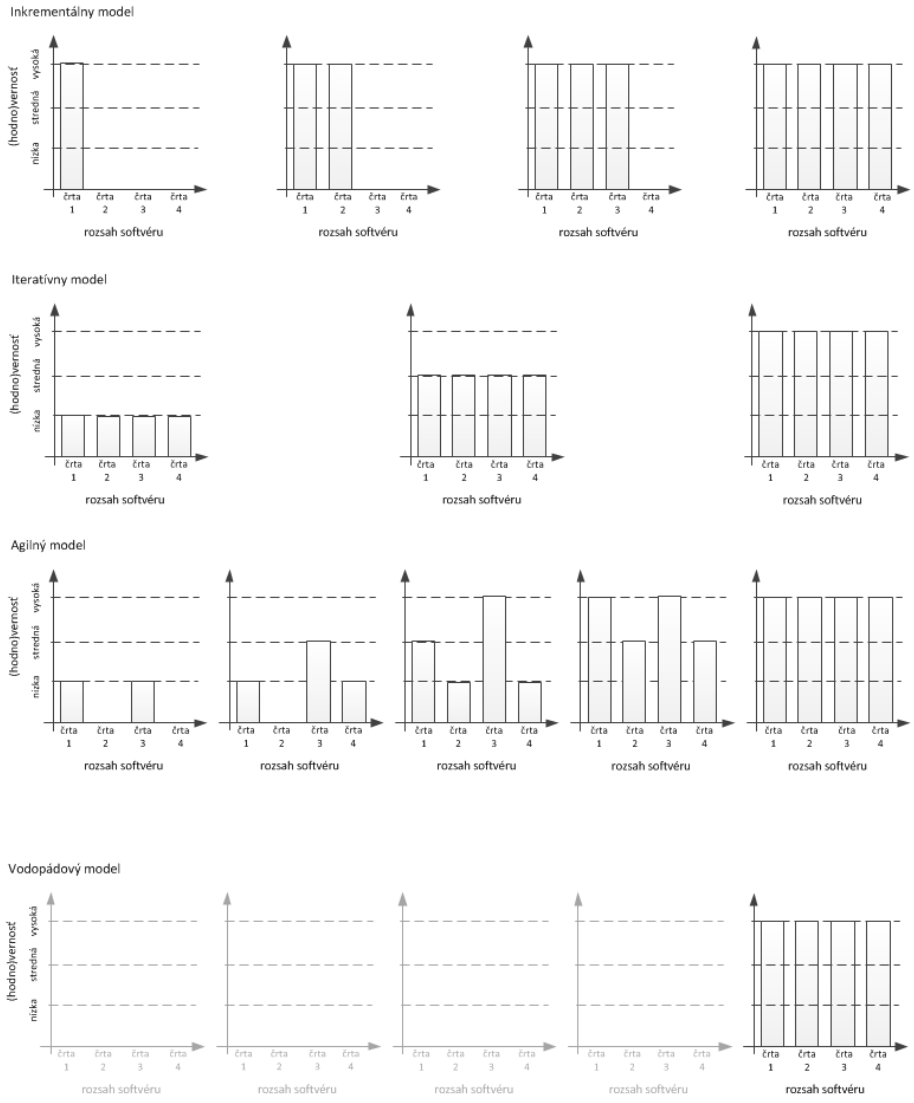
Akým modelom životného cyklu vývoja softvéru sa v súčasnosti v praxi vyvíja najviac? Odpoveď stručne zdôvodnite.

Iteratívno-inkrementálnym. Od vodopádového modelu sa pre jeho evidentné nedostatky upúšťa, aj keď reziduálne niektoré zvyklosti pretrvávajú (napríklad snaha špecifikovať celý softvér podrobne už na začiatku projektu). Agilné metodológie sú síce atraktívne, ale v skutočnosti aj náročné na sebadisciplínu vývojových tímov. Mnoho spoločností, ktoré robili pokusy vyvíjať softvér agilne v týchto pokusoch zlyhalo najmä kvôli nedôslednému uplatňovaniu agilných princípov a vrátili sa k tradičnejšiemu iteratívno-inkrementálnemu prístupu.

Zdroje

[1] Alain Abran et al., eds. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001. ISBN: 0769510000.

[2] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: `http://www.agilemanifesto.org/`.



Obr. 3.5: Porovnanie modelov vývoja

Kapitola 4

Modelovanie softvéru

4.0.1 Čo je to model?

model

Model (vo všeobecnosti) je abstrakcia, čiže „zjednodušenina“ nejakej veci. Vzniká tak, že si vyberieme niektoré charakteristiky veci a zrealizujeme ich, zatiaľ čo iné zanedbáme.

4.0.2 Čo je to model softvéru?

model softvéru

Ako vyplýva z definície pojmu model všeobecne, model softvéru je zjednodušeninou (abstrakciou) softvéru. Spravidla sa zameriava na určitú skupinu charakteristík, ktoré chce o softvéri vizualizovať, pričom ostatné potláča (nezobrazuje).

Napríklad model údajov softvéru zobrazuje iba štruktúru a v rámci nej iba štruktúru údajov. Nedozvieme sa z nej nič o tom ako sa softvér správa, ani akú má architektúru (to sú práve tie „potlačené“ charakteristiky, ktoré sa v danom modeli nezobrazujú).

4.0.3 Prečo vytvárame modely softvéru?

model softvéru

Modely softvéru potrebujeme predovšetkým pri jeho vytváraní, potrebné sú aj neskôr pri údržbe. Dôvodov je viacero a väčšina vyplýva z faktu, že vytvorenie modelu je podstatne rýchlejšie a lacnejšie ako samotného softvéru:

1. Modely možno staticky skúmať alebo použiť na simulácie a zisťovať vlastnosti (budúceho) softvéru.
2. Tým že modely vytvárame, dozvedáme sa čoraz viac o probléme, ktorý riešime (typickým príkladom je vytváranie modelu údajov, pri ktorom sa často dodatočne prichádza na nedostatky špecifikácie a návrhu a formulujú sa spresňujúce otázky pre zákazníka).
3. Pomocou modelov komunikujeme naše myšlienky.

4. Pomocou modelov špecifikujeme softvér. Deklarujeme tak, čo má softvér robiť.
5. Pomocou modelov navrhujeme softvér. Deklarujeme tak, ako má softvér naplniť špecifikáciu.
6. Modelovanie ako abstrakcia, redukuje zložitosť úlohy akou je tvorba softvéru. V podstate ide o typ dekompozície softvéru. Namiesto práce so softvérom ako celkom sa zaoberáme jeho časťami ale nie v zmysle „horizontálne“ oddelených súčiastok, ale „vertikálne“ oddelených aspektov (pohľadov).
7. Modely používame pri uzatváraní dohôd.
8. ...

4.0.4 V ktorých etapách životného cyklu vývoja softvéru vytvárame modely?

model softvéru

Modely softvéru vytvárame vo všetkých etapách životného cyklu softvéru. Od modelovania biznis procesov v analýze, cez špecifikáciu funkcionálnych požiadaviek (ktoré sú sami o sebe modelom), pokračujúc cez návrh architektúry, údajov, komunikácie či algoritmov v návrhu a implementácií až po dokumentovanie rozmiestnenia komponentov pri nasadení softvéru, pri všetkých týchto činnostiach vytvárame modely či už formou grafických diagramov alebo obyčajným textom (áno, aj obyčajný text môže slúžiť ako model softvéru).

4.0.5 Aká je základná charakteristika dobrého modelu softvéru?

model softvéru

Model by mal mať správnu úroveň podrobnosti a zachytávať iba to, čo práve potrebujeme. Inými slovami, nemali by sme zbytočne rozpracúvať detaily, ktoré rozpracované ešte nemusia byť (tzv. *lean* prístup), no zároveň musíme spraviť model taký podrobný, aby splnil účel, pre ktorý sme ho vytvárali.

Poznámka: táto otázka a odpoveď na ňu znejú takmer triviálne, no vývojári sa prekvapivo často uvedeného odporúčania nedržia. Nedostatočná úroveň detailov modelov je napríklad charakteristická pre prípady, kedy vývojár vytvára model ako podklad pre iného vývojára. Vtedy vzniká problém nejednoznačnosti. Na druhej strane, k prílišnému „predbiehaniu“ dochádza napríklad z dôvodov zlého manažmentu práce alebo z nedostatočnej disciplinovanosti vývojárov. Zbytočné, dopredu rozpracované detaily, sa potom často musia meniť, pretože neboli dostatočne overené základy, na ktorých stoja.

4.0.6
model softvéru

Uveďte (vymyslite) príklad softvéru ktorý je modelom.

Poznámka: Aj keď celý čas hovoríme o modelovaní softvéru, je zaujímavé si uvedomiť, že aj softvér samotný je veľmi často modelom – reálneho sveta.

Prípustných je mnoho odpovedí. Hocijaký softvér, ktorý v nejakom zmysle reprezentuje reálny svet je modelom. Napríklad akademický informačný systém je modelom univerzity. Letecký simulátor je modelom vzdušného priestoru a lietadiel.

4.0.7
dimenzie modelov

Uveďte hľadiská (dimenzie, aspekty), podľa ktorých delíme/nazeráme na modely softvéru.

Poznámka: tieto dimenzie sú dôležité predovšetkým z terminologického pohľadu. Čitateľ si ušetrí veľa námahy a zmätku, ak si tieto termíny osvojí.

1. **Funkcionálny – správania – štruktúrny**. V tejto dimenzií prvé dva typy označujú dynamické modely softvéru, teda modely opisujúce procesy prebiehajúce v softvéri v čase (napr. scenáre používania, algoritmy). Výrazom „funkcionálny model“ označujeme modely opisujúce interakcie softvéru navonok, predovšetkým vo vzťahu k používateľovi. Výrazom „model správania“ označujeme modely opisujúce, čo sa deje vo vnútri softvéru. Štruktúrne modely sú statické, teda zachytávajú tie aspekty softvéru, ktoré sa v čase nemenia (schéma údajov, architektúra).
2. **Dynamický** – **statický**. Je vlastne alternatívou prvej dimenzie, ktorá akurát pri modeloch opisujúcich procesy (dynamických modeloch) nerozlišuje či ide o správanie navonok alebo dovnútra.
3. **Logický - fyzický**. Táto dimenzia označuje pomyselnú mieru konkrétnosti modelu, resp. tomu, ako verne model opisuje podmienky, za akých bude softvér nasadený. Logické modely sú flexibilnejšie a lepšie sa s nimi pracuje, fyzické je zase nevyhnutné vypracovať ak má softvér naozaj fungovať (efektívne). Najlepšie rozdiel vidieť pri modeloch údajov (ktoré nazývame týmito prívlastkami) alebo pri dekompozícií na komponenty (tu je viac logickým model balíkov, viac fyzickým model rozmiestnenia).
4. **Etapa životného cyklu**. Jednotlivé modely resp. techniky modelovania nie sú rovnomerne zastúpené v celom životnom cykle softvéru, ale naopak, spravidla zodpovedajú jednej konkrétnej etape životného cyklu kde sú vytvárané alebo využívané.
5. **Použitá notácia (diagram)**. Každý model musí byť vyjadrený nejakým formalizmom a v softvérovom inžinierstve ide najčastejšie o diagramy. Jazyk UML nám poskytuje dostatok diagramov na pokrytie všetkých typov modelov, ktoré by sme mohli potrebovať, existuje však samozrejme množstvo alternatívnych notácií aj techník.

4.0.8
model softvéru

Čo rozhoduje o rozsahu modelovania v softvérovom projekte?

Už intuitívne zrejme vieme povedať, že rozsah modelovania bude predovšetkým závisieť od rozsahu projektu ako takého (pre jednoduchosť si rozsah možno predstaviť ako počet funkcií, ktoré má softvér poskytovať, či objem financií, ktoré sú na projekt vyčlenené). Závislosť rozsahu projektu a modelovania v ňom potom pre zjednodušenie možno považovať za lineárnu (nie je žiaden dôvod sa domnievať, že by sa mal rast množstva potrebného modelovania spomaľovať alebo zrýchľovať s narastajúcou veľkosťou projektu).

Okrem toho však cítime, že rozsah modelovania bude ovplyvnený aj charakterom projektu (ak by sme pokračovali v matematickej analógii, potrebujeme určiť smernicu lineárnej funkcie, ktorá závislosť určuje). Závisí od otázok ako napríklad:

- Je nám doména riešenia známa alebo neznáma? Ak je známa, nepotrebujeme toľko modelovať, lebo mnohé veci vieme povedať hneď.
- Máme v projekte pracovať s pre nás novými technológiami? Ak áno, budeme potrebovať viac modelovať aby sme nabrali istotu.
- Existuje už podobný softvér? Ak áno, môžeme ho využiť ako príklad, ktorý nahradí niektoré modely, resp. uľahčí ich tvorbu.
- Riešili sme už podobný projekt? Ak áno, budeme modelovať zrejme menej resp. rýchlejšie, pretože už máme skúsenosti.
- ...

Uvedené otázky majú spoločného menovateľa: modelovanie je v nich prostriedkom na odstraňovanie neistoty a predchádzaniu rizikám. Platí teda, že čím rizikovejší a neistejší je charakter projektu, tým môžeme čakať viac modelovania.

Poznámka: Mieru modelovania môže ovplyvniť ešte jedna dôležitá črta projektu, a to je metodológia vývoja: napríklad pri tradičných metodológiách (založených na vodopádovom či iteratívno-inkrementálnom prístupe) možno očakávať viac modelovania ako pri agilných.

4.0.9
model softvéru

Čo rozhoduje o tom, aké modely v softvérovom projekte použijeme?

Poznámka: predpokladajme, že projekt nemá triviálne malý rozsah, teda vylúčme prípad, že niektorý model nevytvoríme len preto, že si vieme softvér celý predstaviť aj bez toho.

Začnime vyslovením tézy: existujú také modely, bez ktorých sa žiaden projekt nezaobíde a potom také, ktoré využívame situačne (keď sa to pre daný softvér/projekt „hodí“). Ak táto téza platí, otázka sa prirodzene bude týkať len druhej skupiny modelov.

Skúsme teda určiť, ktoré modely sú v ktorej skupine a pre modely druhej skupiny, skúsme opísať okolnosti, ktoré rozhodnú o tom, či sa použijú.

Ak by sme sa na otázku pozreli cez *typy* (hľadiská, dimenzie) modelov, asi by sme pre každú dimenziu a každú jej hodnotu mohli povedať, že ju určite bezpodmienečne potrebujeme. Môžeme napríklad úplne vynechať funkcionálne, štruktúrne alebo behaviorálne (modely správania) modely? Nie. Zaobídeme sa úplne bez logického alebo fyzického modelovania? Nie. Môžeme si dovoliť vynechať modelovanie v niektorej etape vývoja? Nie. Jedinou výnimkou je dimenzia „použitá notácia“, ktorej hodnoty sú podstatne jemnejšie rozdelené a je možné, že niektoré typy diagramov nebudeme potrebovať.

Musíme teda zísť na úroveň jednotlivých modelov a určiť nutnosť ich použitia každému zvlášť. Potrebujeme ich bezpodmienečne?

- **Model biznis procesov?** Teoreticky nie, ale musíme mať alternatívny spôsob, ako porozumieť procesom v doméne.
- **Model domény?** Teoreticky nie, ale musíme mať alternatívny spôsob ako porozumieť, aké veci a vzťahy v doméne existujú.
- **Model funkcionálnych požiadaviek (používateľských scenárov)?** Áno. Ťažko sa nám bude vytvárať softvér ktorý nemá definované, čo má robiť.
- **Logický model údajov?** Áno. Mohli by sme síce argumentovať, že niektoré softvéry trvalo neuchovávajú žiadne údaje, no aj v ich prípade musíme nejakým spôsobom reprezentovať aspoň tie údaje, ktoré uchovávajú dočasne.
- **Fyzický model údajov?** Nie je potrebný vždy v „plnom znení“. Dnešné nástroje na podporu vývoja a databázové technológie urobia za vývojárov veľa zo „špinavej práce“ spojenej s konkretizovaním logického modelu. Ak však chceme mať pod kontrolou výpočtovú efektívnosť softvéru, fyzickému modelovaniu údajov sa nevyhneme.
- **Stavový model?** Využíva sa situačne. Vnímať ho možno ako určitú nadstavbu modelu údajov. Využíva sa len v prípadoch, kedy niektorá entita alebo komponent počas svojho života nadobúda viaceré stavy a zároveň tieto stavy ovplyvňujú jej správanie a zároveň je model týchto stavov dostatočne zložitý na to, aby stálo za to ho vôbec modelovať. Takéto situácie v skutočnosti nastávajú veľmi často a skôr dochádza k tomu že nie sú docenené a stavový systém sa nemodeluje, no zároveň neplatí, že stavové mechanizmy máme hľadať úplne všade.
- **Model architektúry?** Áno. Bez architektúry nevieme, v akom štruktúrnom rámci bude softvér existovať.
- **Model interakcií a komunikácie?** Áno. Tento model je dynamickou dvojičkou architektonického modelu. Bez neho by neexistoval rámec, v ktorom bude softvér vykonávať potrebné procesy.
- **Model procesov a algoritmov?** Áno. Potrebujeme vedieť detaily ako budú procesy v softvéri definované a aké algoritmy sa majú použiť. Tu však treba povedať, že pri menších projektoch (hoci stále nie triviálnych) sa často vieme zaobísť s tým, že tieto rozhodnutia robíme až priamo pri implementácií.

- **Model rozmiestnenia softvéru?** Teoreticky nie. Ak by sme vyvíjali monolitický softvér (teda softvér o jednom komponente, bežiaci na jednom fyzickom stroji) nemusíme modelovať fyzické rozmiestnenie jeho komponentov.
- **Model tried?** Prakticky áno. Vyhnúť sa objektovo-orientovanému programovaniu dnes prakticky nie je možné.
- **Model štruktúry používateľského rozhrania?** Nie. Prirodzene, nepotrebujeme ho ak softvér nemá používateľské rozhranie (napríklad v prípade programových knižníc).

4.0.10 Čo je to UML?

uml

Skratka pochádza z anglického *unified modeling language*, teda zjednotený modelovací jazyk. UML je grafický, poloformálny jazyk určený a modelovanie softvéru.

To že je UML jazyk poloformálny znamená, že síce má definovanú syntax, ale nie vždy sú jeho pravidlá jednoznačné, presné a detailné. Okrem toho tieto pravidlá nepokrývajú celý priestor toho, čo pri modelovaní potrebujeme robiť a čo chceme zachytiť. V praxi teda nie je rigidný ako napríklad programovacie jazyky a možno ho úspešne používať aj keď v modeloch v ňom vytvorených máme chyby.

UML sa skladá z viacerých diagramov (v UML 2.0 ich je 14). Každý diagram predstavuje unikátnu notáciu (formálny spôsob zápisu informácií), a zameriava sa vždy na špecifický aspekt softvéru (tak ako sa aj model vždy zameriava na špecifické aspekty).

4.0.11 Ako delíme diagramy v rámci UML?

uml

Diagramy v UML podobne ako modely softvéru delíme na štruktúrne (statické) a behaviorálne (dynamické), pričom v rámci behaviorálnych diagramov špeciálne rozlišujeme ešte skupinu interakčných diagramov.

4.0.12 Na čo slúži UML ?

uml

UML slúži na modelovanie softvéru. Pomocou neho softvér:

- **Vizualizujeme**. Základnou funkciou každého modelu softvéru je vizualizovať nejaké vlastnosti softvéru.
- **Komunikujeme.** Hoci je UML len poloformálnym jazykom, je podstatne presnejším a efektívnejším pri komunikácií našich myšlienok iným vývojárom než prirodzený jazyk.
- **Špecifikujeme**. UML notácia, napriek tomu že je len poloformálna, stačí na jednoznačné vyjadrenie toho čo má softvér spĺňať.

- **Konštruujeme**. Pri navrhovaní softvéru slúži jazyk UML ako efektívny nástroj na vyjadrenie vysokoúrovňových čŕt softvéru, napr. architektúry, modelu údajov či modelov interakcií a procesov. Slúži tak ako medzistupeň medzi veľmi nejasnou myšlienkou v hlave vývojára o tom ako má softvér vyzerať a tvrdým a detailným zdrojovým kódom, ku ktorému sa „na jeden skok“ je z obyčajnej myšlienky len veľmi ťažké dostať.
- **Dokumentujeme**. Modely vyjadrené pomocou UML aj po dokončení softvéru naďalej opisujú jeho vysokoúrovňové črty (samozrejme za predpokladu, že sa pri zmenách softvéru dôsledne udržiavajú) a umožňujú tak novým vývojárom ľahšie oboznamovanie sa s daným softvérom.

4.0.13 Aká je základná „filozofia“ modelovania v UML (súvisí s iteratívno-inkrementálnym prístupom)

uml

Základnou filozofiou modelovania v UML je postupná konkretizácia a modelovanie čoraz jemnejších detailov softvéru. Začína sa vždy s najhrubšími a najdôležitejšími črtami a detaily sa pokiaľ možno riešia až nakoniec. Nie je prekvapujúce, že takto sa pokúšame vytvárať softvér ako taký, nielen jeho modely – ide totiž o najbezpečnejší postup v ktorom nás najmenej budú bolieť prípadné zmeny špecifikácie.

Príkladom tejto stratégie môže byť napríklad modelovanie údajov diagramom tried. Ako prvé identifikujeme, aké entity vôbec budeme uvažovať. Následne (až keď máme rozumnú mieru istoty, že sme na žiadnu nezabudli) určíme vzťahy medzi nimi a atribúty a ich typy určíme až nakoniec. Ak by sme k detailom (vzťahom ale najmä atribútom a typom) prešli hneď, zbytočne sa vystavíme riziku ich zmien v dôsledku identifikovania nových entít.

4.0.14 Je zdrojový kód modelom softvéru?

zdrojový kód

Za model softvéru možno formálne považovať aj zdrojový kód programov. Prakticky ho ale za model možno považovať len ťažko (nemá väčšinou potrebné vlastnosti ako abstraktnosť a nízku cenu vytvorenia).

Kapitola 5

Metódy tvorby softvéru

5.0.1

metóda tvorby softvéru, funkcionálna metóda, dátovo-orientovaná metóda, objektovo-orientovaný

Aké spôsoby vývoja softvéru poznáme z pohľadu toho, na čo sa kladie pri modelovaní dôraz? (čím pri modelovaní začíname?)

Poznámka: Tieto pojmy berte skôr ako prívlastky či kategórie, ktorými charakterizujeme metodológie vývoja softvéru. Nie sú to metodológie ako také.

Rozlišovať môžeme 3 základné spôsoby:

1. **Funkcionálny prístup**. Ide o prípad, kedy modelovanie softvéru vedieme predovšetkým ako modelovanie jeho procesov. Vychádza sa z namodelovaných biznis procesov a funkcionálnych požiadaviek a postupne sa rozvíjajú a čoraz podrobnejšie modelujú aktivity, algoritmy, interakcie v rámci softvéru. Možno čakať, že sa vzhľadom na to budú využívať zodpovedajúce notácie diagramov: diagramy aktivít, sekvenčné diagramy, diagramy tokov údajov, komunikačné diagramy, diagramy časovania a pod. Modelovanie údajov je v tomto prístupe až druhoradé.

2. **Dátovo-orientovaný prístup**. V tomto prípade začíname modelovaním údajov v softvéri. Vychádza sa z poznatkov o problémovej doméne, identifikujú sa jej entity a vzťahy medzi nimi. Následne sa rozhoduje o tom, ktoré entity má zmysel reprezentovať aj vo vytváranom softvéri. Vývoj následne zvykne pokračovať vytvorením úložiska údajov, na ktoré sa až potom začnú nabaľovať funkcie. Kľúčovými technikami modelovania sú diagramy na modelovanie údajov, najmä diagram tried, prípadne staršie techniky ako entitno-relačný diagram a diagram modelu údajov.

3. **Objektovo-orientovaný prístup**. Je kombináciou vyššie uvedených prístupov. Podobne ako v dátovom prístupe sú v centre pozornosti entity (domény, neskôr softvéru) no do úvahy sa berie aj ich správanie (údaje so správaním spolu tvoria triedy). Vývojári striedavo modelujú triedy a procesy (a interakcie) ktoré sa v softvéri majú vykonávať a snažia sa, aby boli v harmónií (kľúčovým diagramom ktorý sa tu využíva je sekvenčný diagram).

5.0.2 Kedy je lepšie dať prednosť funkcionálnemu a kedy k dátovo-orientovanému prístupu k modelovaniu a vývoju softvéru?

metóda tvorby softvéru, funkcionálna metóda, dátovo-orientovaná metóda

Toto rozhodnutie je často *subjektívnou* preferenciou vývojárov. Tí majú skôr sklony radšej modelovať údaje, než procesy (dôvodom je pravdepodobne fakt, že aj pre neznámu doménu sa údaje modelujú relatívne ľahšie, než procesy). Okrem toho vývojárom často imponuje to, že modely údajov vedia byť „čistejšie“ (po tom čo ich normalizujú je jednoduchšie v nich dodržovať princíp DRY – *don't repeat yourself*). Naproti tomu modelovanie procesov a funkcionality sa zvyčajne nezaobíde bez opakovania a modely sú ťažšie udržiavateľné a menej prehľadné.

Najsilnejším *objektívnym* indikátorom je zvyčajne komparatívna dôležitosť procesov resp. údajov v softvéri. Napríklad, v softvéri na manažment online obchodu bude zrejme dôležitejšie správne riadiť celý priebeh obchodovania a logistiky. Bude preto výhodnejšie pochopiť a správne určiť (namodelovať) postupnosť činností a priebehy rôznych alternatív (autentifikácie, predaja, dopravy a pod.), ako začať modelovať objednávky a tovary (o atribútoch ktorých bez znalosti procesov aj tak na začiatok nevieme povedať veľa). Naopak v prípade softvéru akým je akademický informačný systém, ktorého primárnou úlohou je uchovávanie údajov a nositeľom procesov sú predovšetkým ľudia, ktorí systém používajú (oni rozhodujú, čo sa s údajmi bude diať), je možno výhodnejšie začať s modelovaním údajov a postupne prejsť na objektovo-orientovaný prístup modelovania.

O výbere stratégie tiež môže rozhodovať miera vedomostí vývojárov o procesoch resp. údajoch. Inými slovami, je výhodné najskôr modelovať niečo, o čom vieme viac a čím sme si istí.

5.0.3 Aké paradigmy programovania poznáte?

programovanie, paradigma programovania

Najčastejšie spomínané paradigmy uvádzame v nasledujúcom zozname:

- **Imperatívne programovanie**. Dnes len minimálne používaná paradigma, ktorá vznikla ako historicky prvá. Program mení svoj stav postupným vykonávaním zoznamu príkazov. Program však nutne nemusí byť štruktúrovaný, charakteristické je v ňom používanie príkazu GOTO. Dnes sa imperatívne programovanie používala najmä pre programovanie v assembleroch, vo vyšších jazykoch bolo nahradené procedurálnym prístupom.
- **Procedurálne programovanie**. Ide o typ imperatívneho programovania, ktoré však zavádza do programov prísne štruktúrne pravidlá, postavené predovšetkým na funkciách.
- **Objektovo-orientované programovanie**. Snaží sa o poňatie programu ako metafory reálneho sveta, v ktorom objekty majú svoje vlastnosti a interagujú s inými objektami prostredníctvom volaní metód.
- **Funkcionálne programovanie**. Programom je výpočet (zloženej) matematickej funkcie.

- **Logické programovanie**. Program je súborom logických výrokov z ktorých sú následne odvodzované ďalšie tvrdenia.
- **Deklaratívne programovanie**. Paradigma, v ktorej program definuje charakteristiky svojho výsledku, no nedefinuje, ako sa k výsledku má dopracovať (to je ponechané na interpreter) Predstaviteľom tejto paradigmy je napríklad jazyk SQL.
- **Distribuované programovanie**. V tejto paradigme sa počíta s ďalšou úrovňou zložitosti programu: možnosťou paralelne prebiehajúcich výpočtov na viacerých výpočtových jednotkách (procesoroch). Spravidla v kombinácii s ďalšími paradigmami, princípy distribuovaného programovania sa sústredia na problémy súvisiace so synchronizáciou, vhodným rozdeľovaním úloh do jednotlivých procesov a spájaním ich výsledkov.

Poznámka: Niektoré z paradigiem sa prekrývajú a majú spoločné prvky. Taktiež máloktorý programovací jazyk prislúcha striktne iba k jednej paradigme. Existujú samozrejme „čisté“ jazyky, ich praktické použitie však väčšinou vyžaduje „primiešanie“ prvkov iných paradigiem.

5.0.4
komponent

Aká je základná charakteristika komponentového prístupu k vývoju softvéru? Prečo je výhodné ho použiť?

Skladanie softvéru z existujúcich súčiastok v čo najväčšej miere. Reálny vývoj sa následne prenáša do vytvárania prepojení medzi súčiastkami. Motiváciou k takémuto prístupu je samozrejme úspora úsilia, je však vždy treba zvážiť, či nám existujúce súčiastky vyhovujú, či budeme schopní ich v prípade chýb opraviť a či nás to všetko nebude stáť viac úsilia, než vytvorenie softvéru „na zelenej lúke“.

5.0.5
zhora-nadol

Vysvetlite pojem „návrh zhora-nadol“. Kedy volíme túto stratégiu?

Je to stratégia, kedy najskôr navrhujeme zásadné črty softvéru a postupne sa prepracúvame k detailom (inými slovami, ideme od abstraktného ku konkrétnemu). Z pohľadu štruktúry softvéru je pre túto stratégiu typické, že najskôr vypracúvame architektonický návrh a až následne jednotlivé komponenty a ich súčasti. Z pohľadu dynamiky softvéru začíname od procesov a funkcionálnych požiadaviek a postupne odkrývame a navrhujeme detailnejšie, aké procesy a algoritmy majú prebiehať vo vnútri softvéru až sa dostaneme na úroveň jednotlivých metód.

Po tejto stratégií siahame predovšetkým vtedy, keď je jasná špecifikácia celého softvéru. Stratégia má blízko k vodopádovému modelu vývoja, preto si musíme byť vedomí aj jej rizík. Jedným z nich je aj problematické odhadovanie úsilia (práce) a predvídanie problémov hlavne v prvých fázach vývoja, kedy pracujeme s príliš veľkými, zatiaľ neznámymi komponentmi. Je preto veľmi výhodné, ak riešime dobre známy problém resp. máme skúsenosti s podobným projektom a môžeme tak lepšie odhadovať a predvídať problémy.

5.0.6
zdola-nahor

Vysvetlite pojem „návrh zdola-nahor“. Kedy volíme túto stratégiu?

Je to stratégia, kedy najskôr navrhujeme jednotlivé súčiastky softvéru, so všetkými detailmi a až neskôr ich spájame do celkového návrhu softvéru. Nemusí ísť len o súčiastky (štruktúrny pohľad) ale aj o funkcie (služby) softvéru, teda najskôr navrhneme a zrealizujeme jednu funkciu, potom k nej pridáme ďalšiu a ďalšiu a celkový návrh (a realizácia) softvéru vznikne postupne.

Po tejto stratégií viac siahame, keď nemáme toľko istoty ohľadom špecifikácie a ohľadom znalostí problémovej domény. Taktiež je výhodná, ak máme veľa už hotových súčiastok. Pozorný čitateľ si iste všimol, že tvorí základ agilnejších prístupov k tvorbe softvéru.

Prečo je stratégia zdola-nahor na toto vhodná? Ak softvér vytvárame po menších častiach, sme schopní ich rýchlejšie validovať a rozptyľovať tak neistotu.

5.0.7
prototyp

Čo je to prototyp?

Význam pojmu vo všeobecnosti znamená „prvý svojho druhu“. Prototypom v oblasti softvéru je tým pádom prakticky každý softvér, ktorý práve vyvíjame.

V užšom zmysle za prototyp označujeme taký softvér, ktorým sa snažíme preveriť, či kľúčové princípy našich návrhov fungujú. Preto prototypy nebývajú po všetkých stránkach dotiahnutými softvérmi.

5.0.8
prototyp

Prečo je prototypovanie dôležité pre softvérové inžinierstvo?

Prototypovanie je kľúčovým nástrojom na odstraňovanie neistoty vo vývoji softvéru. Môže ísť o neistotu vo viacerých zmysloch. Požiadavky zákazníka môžu byť nepresné, vágne alebo neúplné. Technológie na ktorých chceme riešenie postaviť pre nás nemusia byť známe alebo môžeme mať pochybnosti o tom, či budú fungovať.

Skorým prototypovaním kľúčových vecí softvér čiastočne zhmotníme, môžeme ho vyskúšať a dať vyskúšať používateľom, vyjasníme detaily, ukazuje čo možné je a čo nie.

5.0.9
prototyp

Uveďte príklad prototypu softvéru, ktorý netreba programovať?

Jeden príklad sú tzv. papierové prototypy, čiže grafické návrhy používateľských rozhraní, ktoré ukazujeme zákazníkovi či už vo fyzickej alebo papierovej podobe. Nástroje na tvorbu týchto prototypov dokonca umožňujú aj čiastočne nadefinovať dynamiku rozhrania aby jeho skúšanie pôsobilo vernejšie.

Iným typom prototypov, ktoré sa nemusia programovať, sú tzv. *mockups*, teda akési makety softvéru, ktoré sa navonok tvária ako funkčný softvér, no v skutočnosti reagujú na vstupy len „natvrdo“ definovanými odpoveďami. Používajú sa napríklad pri návrhu programových rozhraní (API) a je užitočné vytvárať ich počas návrhu architektúry.

Vývojári, zákazníci či subdodávatelia ich tak môžu využívať, včas pripomienkovať a uľahčiť tak integráciu softvéru.

5.0.10
prototyp, prototyp na zahodenie, evolučný prototyp

Aké dva druhy prototypovania poznáme? Stručne uveďte v čom sa líšia.

Rozlišujeme dva základné typy:

1. **Prototyp na zahodenie**. Prototyp ktorý po splnení jeho úlohy (napr. overení nejakej technológie či algoritmu) nijak nezapojíme do vytváraného softvéru. Prototypy na zahodenie sú veľmi lacné, pretože pri nich nemusíme dodržovať žiadne štandardy kvality, nepotrebujeme dokumentovať ani nijak zvlášť udržiavať zdrojový kód programu – snažíme sa čisto o dodanie potrebnej funkcionality.
2. **Evolučný prototyp**. Je opakom prototypu na zahodenie. Evolučný prototyp je určený na zapojenie do vytváraného softvéru (resp. vytváraný, cieľový softvér vznikne modifikovaním – evolúciou prototypu). Hoci zozačiatku sa tiež snažíme vytvoriť predovšetkým funkcionalitu, o ktorú nám ide, neskôr vytvorený „technický dlh“ dobehneme refaktorovaním a dokumentovaním prototypu.

5.0.11
prototyp, prototyp na zahodenie, evolučný prototyp

Kedy volíme prototypovanie na zahodenie a kedy radšej evolučné prototypovanie?

Prototypy na zahodenie využívame na overenie tých najmenej istých a jasných čŕt vyvíjaného softvéru, teda na veci ktoré sú najrizikovejšie. Následne keď sú všetky rizikové črty overené, vytvoríme celý softvér „už poriadne“.

Evolučné prototypujeme tie črty, u ktorých máme veľkú istotu (ohľadom toho ako ich realizovať). Zároveň pritom dúfame, že sa počas tejto realizácie vyjasnia aj otázky ohľadom rizikovejších častí softvéru.

Pokiaľ má softvér priveľa neistých čŕt a zároveň tieto črty predstavujú prioritnú funkcionalitu softvéru, potom treba aj v projekte začať prototypovať na zahodenie. Naopak evolučným prototypovaním zvyčajne začíname vtedy, keď prioritne požadovanú funkcionalitu vieme s istotou zrealizovať a rizikové črty sú až ďalšie v pláne.

5.0.12
prípad použitia

Vysvetlite čo znamená, že proces tvorby softvéru je vedený prípadmi použitia?

Vedenie vývoja prípadmi použitia znamená, že každý vytváraný artefakt v modeli či v zdrojovom kóde mapujeme na jeden alebo viac prípadov použitia, ku ktorých realizácií ho potrebujeme. Inými slovami, sledujeme, či skutočne každá časť softvéru, ktorú vo vývoji vytvárame prispieva k napĺňaniu funkcionálnych požiadaviek (vyjadrených prípadmi použitia). Artefakty, ktoré nevieme namapovať na nijaký prípad použitia sú vo svojej podstate zbytočné.

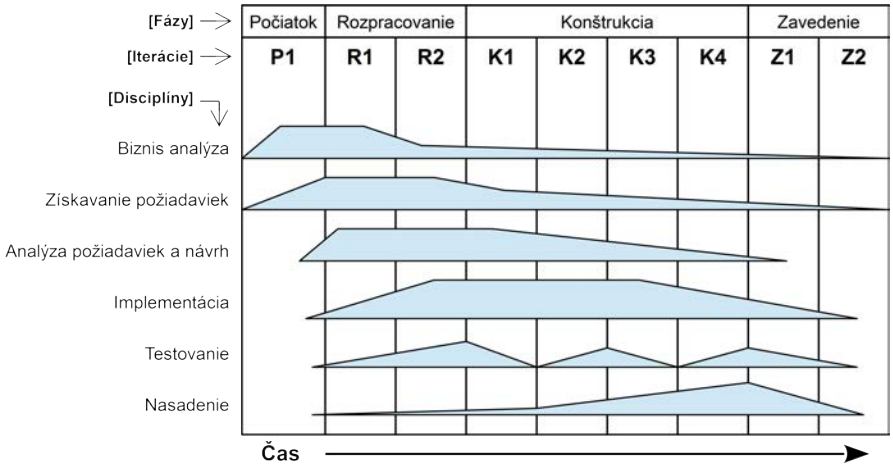
5.0.13
unified process

Charakterizujte pojem „Unified Process“ – čo to je a na čo sa používa v softvérovom inžinierstve

Unified Process (UP) je rámec tvorby softvéru. Používa sa pri definovaní činností a ich postupností v rámci softvérového projektu. Ako možno vidieť na obrázku 5.1, UP definuje 4 fázy, ktorými každý projekt prejde (*inception, elaboration, construction, transition*) a v rámci ktorých vždy prebehne jedna alebo viac iterácií cez tzv. disciplíny (*disciplines*), ktoré zhruba zodpovedajú etapám životného cyklu softvéru (ide teda o iteratívno-inkrementálny rámec).

Unified Process je ako taký abstraktný a nedefinuje príliš veľa detailov, ako presne má byť vykonávaný. Organizácie, ktoré sa ho rozhodnú využiť tak majú značné možnosti prispôsobiť si ho. Medzi konkrétnejšie metodológie, postavené na UP patrí Rational Unified Process, OpenUP a Agile Unified Process.

Unified Process je zameraný predovšetkým na projekty väčšieho rozsahu. Nevylučuje sa jeho používanie aj v menších projektoch, tam však väčšinou dáva prednosť agilným metodológiám ako je SCRUM (ktoré naopak pre použitie vo veľkých projektoch vyžadujú značnú adaptáciu alebo sú nepoužiteľné).



Obr. 5.1: Unified process je iteratívno-inkrementálny rámec tvorby softvéru. Iterácie rozdeľuje do fáz a vývojové činnosti združuje do disciplín, ktorých intenzita vykonávania sa mení v priebehu projektu. Počet iterácii pritom nie je daný, počty na obrázku slúžia len na ilustráciu.

5.0.14
unified process

O aké princípy sa opiera Unified Process?

Unified Process sa opiera o 4 kľúčové princípy:

1. **Iteratívno-inkrementálny vývoj**.
2. **Vývoj vedený prípadmi použitia**.

3. **Postavený na architektúre**. Architektúra softvéru je vypracúvaná a zvalidovaná veľmi skoro a venuje sa jej značné úsilie. Až potom sa pristupuje k implementácií komponentov.
4. **Orientovaný na riziká**. Kladie sa veľký dôraz na identifikáciu rizík projektu a na odstraňovanie, zmierňovanie najvážnejších rizík už od začiatku projektu (napríklad prototypovaním na zahodenie).

5.0.15 unified process

Aké štyri fázy má Unified Process?

Poznámka: Iteratívno-inkrementálny charakter Unified Process je zrejmý, no predsa len sa pozorný čitateľ nemusí ubrániť dojmu, že uvedené štyri fázy pôsobia „vodopádovo“ (napríklad až v poslednej fáze sa používatelia dostávajú do kontaktu so softvérom). S tým sa v rámci UP vieme vysporiadať, najjednoduchšie zapojením menšieho množstva budúcich používateľov do vývoja aj v predchádzajúcich fázach (najmä pre etapy testovania).

Poznámka: Na opise jednotlivých fáz dobre vidno, že Unified Process je niečo viac, než len životný cyklus softvéru ale že definuje aj aktivity týkajúce sa manažmentu projektu (plánovanie, manažment rizík, ...).

5.0.16 unified process

Aké hlavné skupiny činností (disciplín) definuje Unified Process?

Unified process definuje nasledovné hlavné (čiže na vývoj orientované) skupiny činností (nazývaných tiež disciplíny):

1. Biznis analýza
2. Získavanie požiadaviek
3. Analýza požiadaviek a návrh
4. Implementácia
5. Testovanie
6. Nasadenie

Poznámka: Všimnime si, že tieto činnosti (disciplíny) sa s miernymi odlišnosťami pomerne dobre mapujú na náš referenčný rámec životného cyklu softvéru. Konkrétne, Unified Process definuje dve „analýzy“. Prvou je „biznis analýza“, ktorá viac menej zodpovedá našej referenčnej etape „analýzy“, teda činností, ktoré prebiehajú ešte pred formulovaním požiadaviek na softvér a ktoré vykonávame aby sme porozumeli problémovej doméne. Ekvivalentom „analýzy požiadaviek a návrhu“ tak ako ju definuje UP v našom referenčnom rámci je etapa návrhu. Slovíčko „analýza“ v návrhu znamená, že definované požiadavky analyzujeme v rámci úsilia o vypracovanie čo najlepšieho návrhu softvéru (čo je niečo iné ako analyzovanie domény ako takej).

5.0.17
unified process

Aké podporné skupiny činností (disciplín) definuje Unified Process?

Unified process definuje nasledovné podporné skupiny činností:

1. Konfigurácie a manažment zmien (aktivity zodpovedajúce najmä údržbe softvéru)
2. Manažment projektu (ľuských zdrojov, plánovania, rizík, kvality, ...)
3. Prostredie (podporné aktivity vývoja ako zabezpečovanie nástrojov, nastavenie vývojových prostredí, štandardov a pod.)

5.0.18
unified process

Vysvetlite vzťah cyklu, iterácie, fáz (angl. phase) a skupín činností (angl. discipline) v metodike Unified Process

Cyklus, fáza aj iterácia sú časové úseky projektu. Celý projekt sa skladá najmenej z jedného cyklu. Môže ich mať aj viac, ale zvyčajne má len jeden.

Cyklus sa skladá z fáz a v jednom cykle sú vždy presne štyri fázy (počiatok, rozpracovanie, konštrukcia, zavedenie).

Každá fáza sa skladá z jednej, alebo viacerých iterácií. Iterácia sa teda súčasťou práve jednej fázy.

Skupiny činností (disciplíny) sú vzhľadom na časové hľadisko „kolmé“. Všetky prebiehajú v každom cykle, fáze resp. iterácií, avšak vždy s rôznou intenzitou, podľa toho, ako pokročilý je projekt.

5.0.19
unified process

Ktoré z hlavných činností (disciplín) sú v Unified Process spravidla dominantné pre fázu inicializácie (inception)?

Biznis modelovanie a získavanie požiadaviek.

5.0.20
unified process

Ktoré z hlavných činností (pracovných tokov) sú v Unified Process spravidla dominantné pre fázu rozpracovania (elaboration)?

Analýza požiadaviek a návrh.

5.0.21
unified process

Ktoré z hlavných činností (pracovných tokov) sú v Unified Process spravidla dominantné pre fázu vytvorenia softvéru (construction)?

Analýza požiadaviek a návrh, Implementacia a Testovanie.

5.0.22
unified process

Ktoré z hlavných činností (pracovných tokov) sú v Unified Process spravidla dominantné pre fázu zavedenia (transition)?

Testovanie, nasadenie.

5.0.23
unified process

Najmenej koľko iterácií musí mať projekt podľa Unified Process?

Najmenej štyri. Ak sa projekt skladá z aspoň jedného cyklu, musí mať aspoň 4 fázy. Ak má každá z nich mať aspoň jednu iteráciu, znamená to, že celý projekt musí mať najmenej 4 iterácie.

5.0.24
unified process, uml

Aký je rozdiel medzi UML a UP (Unified process)?

UML je špecifikačný jazyk. UML je množinou *techník* na opisovanie (modelovanie) softvéru.

Naproti tomu Unified Process opisuje proces tvorby softvéru, je to *metodológia* resp. metodologický rámec.