ÚDRŽBA SOFTVÉRU – NÁROČNÁ ALEBO ZANEDBÁVANÁ?

Zdĺhavé procesy nemusia byť objektívne zdĺhavé.

Máté Fejes

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava
matefejes13[zavináč]gmail[.]com

Abstrakt. *Podľa všeobecných poznatkov manažmentu softvérových projektov pokročilé etapy vývoja sú oproti úvodným časovo náročné. Tento fakt môže byť aj dôsledkom prístupu k tvorbe softvéru a kladenia veľkého dôrazu na vybrané činnosti. Možno sústredenie sa na aktivity, ktoré sú dnes považované za dôležité, má na výsledok odhadovania potrebného času a úsilia negatívny dopad. Existujú nástroje, ktoré síce slúžia na podporu inej oblasti, majú potenciál aj vo vylepšovaní kvality údržby softvéru a redukovaní jej časovej náročnosti. Modelovaním evolúcie na základe histórie zmien predchádzajúcich projektov by sme dokázali predpokladať a vopred eliminovať výskyt najpravdepodobnejších chýb, resp. zmien, ktoré by celkový potrebný čas, úsilie a tým aj cenu produktu zbytočne navyšovali.*

Kľúčové slová: *údržba softvéru, modelovanie evolúcie, história zmien, systémy pre manažment verzií, manažment podpory vývoja*

Bežný prístup k projektu – na čo sa sústreďujeme

Väčšina softvérových projektov trvá niekoľko mesiacov, v mnohých prípadoch aj viac rokov. Kým úvodné etapy projektu prebiehajú pomerne rýchlo, v pokročilejších etapách sú časové nároky tímu čoraz väčšie. Po rýchlej analýze, návrhu a implementácii prvých verzií nasleduje zdĺhavé opravovanie kódu, dopĺňanie nových funkcionalít a údržba systému. Je nesporné, že najviac času, úsilia a iných prostriedkov zaberajú práve tieto posledné úkony.

V súčasnosti existuje a bežne sa používa množina metód pre návrh softvéru. Rozličné modely vývoja, modelovacie prostriedky a ďalšie poznatky tvoria vedu softvérového

inžinierstva, ktoré sa zameriava na oblasť vytvárania počítačových aplikácií. Táto veda je dnes na špičkovej úrovni, pomocou nej dokážeme podrobne rozoberať a riešiť relevantné problematiky. Rôzne existujúce metódy slúžia na získanie najlepšieho možného výsledku v rámci projektov, väčšina z nich je však zameraná na prvé etapy vývoja – analýza, návrh a implementácia.

Čo zanedbávame?

Ak sa zamyslíme nad doteraz diskutovanými skutočnosťami, môžeme si všimnúť paradox: metódy a prostriedky návrhu a implementácie softvéru sa neustále zdokonaľujú, pričom nikoho netrápi, že revolúciu v softvérovom inžinierstve a projektovom manažmente by mohlo urobiť aj niečo iné. Neexistuje totiž overená metóda, ktorá by dokázala predvídať chyby a požiadavky, ktoré v budúcnosti pravdepodobne vzniknú. Takáto metóda by mohla priniesť značnú úsporu času a úsilia potrebného v posledných etapách vývoja (refaktoring, oprava chýb, údržba) a tým zaručiť výrazné zefektívnenie softvérových projektov.

Dôležité vs. náročné

Pre pochopenie môjho názoru je potrebné ujasniť si, čo vlastne považujeme počas vytvárania softvéru za dôležité a čo za náročné. Ako aj v každodennom živote, za dôležité považujeme veci, ktoré vykonávame často, preto sa snažíme urobiť všetko, aby sme si ich uľahčili. Venujeme veľa času a úsilia vytváraniu prostriedkov, pomocou ktorých sa dôležité činnosti budú ľahko vykonávať a takto sa vlastne stávajú nenáročným činnostiam. Jednoduchým príkladom je doprava: od počiatku čias bolo pre človeka dôležité dostať sa z jedného miesta do druhého. Vďaka vynálezom a neustálemu vývoju máme dnešné dopravné prostriedky, pomocou ktorých sa presun človeka stal nenáročnou činnosťou.

Myslím si, že táto analógia plne platí aj pre softvérové inžinierstvo. Vývojári za najdôležitejšiu činnosť vždy považovali podrobnú špecifikáciu, návrh a implementáciu aplikácií. V zmysle vyššie opísanej teórie tomu zodpovedá aj existencia dnešných moderných nástrojov, ktoré tieto činnosti podporujú.

Toto však platí aj naopak. Vytváraniu metód a pomocných nástrojov pre podporu opravy a údržby softvéru sa doposiaľ venovalo omnoho menej odborníkov. Je to spôsobené tým, že vývojári stále kladú väčší dôraz na návrh a implementáciu, a tak ani nevznikol nárok na podporu ostatných činností.

Úvaha v ďalších častiach eseje nadväzuje na túto teóriu. Aby som ju v jednej vete zhrnul: čím viac sa daná činnosť považuje za dôležitú, tým viac úsilia sa venuje vývoju podporných nástrojov, vďaka ktorým je diskutovaná činnosť v konečnom dôsledku nenáročná a naopak.

Problém nerovnomerného rozdelenia času

V tejto eseji by som sa chcel prehĺbiť v doteraz pomerne zanedbanej problematike podpory údržby softvéru. Mojou snahou je poukázať na skutočnosť, podľa ktorej sa kladie omnoho väčší dôraz na návrh a implementáciu systému, čo však vo všeobecnosti dopadne práve

naopak – väčšina času a úsilia sa venuje tým aktivitám, ktoré mali pôvodne nižšiu prioritu. Uvažujem riešenie, ktoré by do istej miery spôsobilo vyrovnanie potrebného času a úsilia medzi jednotlivými etapami tvorby softvéru.

Prečo je údržba časovo náročná?

Údržba softvérového produktu zahŕňa všetky druhy zmien, ktoré treba vykonať medzi implementáciou a nasadením, resp. po nasadení programu. Počas údržby sa riešia všetky ovplyvňujúce faktory, ktoré boli zistené počas behu projektu, preto sa na ich výskyt ťažko pripravuje. Je to etapa projektu, ktorá nikdy nie je podrobne naplánovaná [1].

Pred nasadením sa koná testovanie, počas ktorého sa odhalí množina chýb a nedostatkov. Keďže množstvo chýb a trvanie ich odstránenia sa nedá vopred presne odhadnúť, ich zisťovanie a opravovanie je v mnohých prípadoch príčinou prekročenia časového, resp. finančného limitu, čo vedie k zlyhaniu, prípadne k neúspešnému ukončeniu projektu.

Treba počítať aj s ďalšími ovplyvňujúcimi faktormi, ktoré môžu nastať kedykoľvek, dokonca aj po nasadení systému počas prevádzky. Sem patria predovšetkým zmena požiadavky zadávateľa, alebo prepracovanie existujúceho návrhu a implementácie, t.j. refaktoring.

Vyššie opísané faktory sú zdrojmi neplánovaných zmien počas vývoja a prevádzky softvéru. Ako už bolo spomenuté, ide o činnosti, ktoré vyžadujú najväčšie úsilie a množstvo času. V zmysle teórie o dôležitosti a náročnosti považujem za zarážajúce, že hľadanie potenciálnych spôsobov na zníženie náročnosti údržby softvéru – a tým na všeobecné zefektívnenie softvérových projektov – je naďalej zanedbávané, t.j. údržba softvéru sa bežne považuje za nedôležitú.

Keby sme údržbu prehodnotili

Som presvedčený o tom, že na diskutovanú problematiku existuje riešenie, i keď len čiastočné. Nakoľko sa jedná o skutočnosť, ktorá jednoznačne obmedzuje prácu v danej oblasti a zároveň nejde o perpetuum mobile, alebo inú moc mimo zákony fyziky, myslím si, že venovať sa tejto problematike sa oplatí. Podmienkou je však, aby si ľudia dali záležať na efektívnej údržbe softvéru a tým vznikol nárok na jej podporu.

Na jednej strane nerozumiem tomu, prečo sa nerieši problém príliš náročnej údržby, na druhej strane je pochopiteľné, že návrh modelu softvéru a zvládnutie opravy nečakanej chyby sú dve výzvy odlišného charakteru. Kým pri vytváraní modelu máme dané možnosti, dokonca je povolené do istej miery tvarovať výsledok podľa vlastného, neznáma chyba je neviditeľná, neexistuje explicitný spôsob na jej odstránenie. Vzhľadom na toto si myslím, že ak existuje spôsob na vylepšenie diskutovanej situácie, riešenie nespočíva v urýchlení opravy chýb, ale v prevencii. Pokiaľ by sme boli schopní sa poučiť z našich krokov, ktoré mali negatívny dopad, a tieto praktické skúsenosti aplikovať pri vytvorení nového diela, množstvo vzniknutých chýb by pravdepodobne bolo omnoho nižšie.

Jeden z existujúcich spôsobov, ktoré majú do istej miery pozitívny vplyv na množstvo vzniknutých nedostatkov, je použitie návrhových vzorov. Návrhové vzory sa požívajú na modelovanie, resp. implementáciu situácií, ku ktorým pri vytváraní softvéru často dochádza. Vďaka tomu, že tieto vzory ponúkajú optimálne riešenie pre časté situácie,

určitú časť potenciálnych chýb pravdepodobne eliminujú. Napriek tomu môžem tvrdiť, že zníženie počtu chýb je len vedľajším účinkom aplikovania návrhových vzorov, pretože sú to taktiež nástroje, ktorých primárnym cieľom je uľahčenie prvých etáp vývoja.

Ideálne riešenie by mohla priniesť metóda, ktorá by bola používaná počas návrhu a implementácie, ale s cieľom znížiť riziko negatívneho dopadu jednotlivých častí systému. Ako primárny zdroj považujem za vhodné predošlé projekty, ktoré z určitého pohľadu môžu byť relevantné (projekty podobného charakteru, projekty danej skupiny riešiteľov a pod.). Získavaním, modelovaním a využívaním skúseností a vedomostí z predchádzajúcich činností, myslím si, že vývojári by mali možnosť na predvídanie potenciálnych chýb. Zníženie času venovaného na údržbu by vo veľkej miere vplývalo na celkový plán projektu, čo by malo mimoriadne pozitívny dopad v projektovom manažmente.

Kde hľadať riešenie?

Ako sme spomínali, riešením by mohla byť metóda, ktorá sa použije ako pomôcka pri návrhu a implementácii, zameraná na zníženie počtu chýb a ideálne by mala vychádzať z iných projektov. Po prečítaní predchádzajúcej vety a zamyslením sa nad jej významom, črtá sa nám primárny zdroj potrebných informácií: systémy na manažovanie verzií (ďalej len *VCS – Version Control System*).

Udržiavajú presne to, čo potrebujeme – množinu dát, ktoré zachytávajú a presne opisujú každú verziu systému, poukazujú na každú drobnú vykonanú zmenu [3]. V ešte lepšom prípade máme navyše k dispozícii informácie o zmenách v závislosti na iných udalostiach, ako požiadavka na zmenu, hlásenie chyby a pod.

Prečo sú systémy pre manažment verzií relevantné?

Dáta, ktoré systémy na manažovanie verzií udržiavajú, nemajú relevantnú informačnú hodnotu v základnom stave. Musíme si povedať, presne ktoré údaje, resp. ich kombinácie sú pre nás vzácne. Taktiež si treba uvedomiť, že samotné získanie informácií z histórie zmien ešte nezabráni výskytu chýb počas vývoja, preto je potrebné navrhnúť vhodnú reprezentáciu a spôsob využitia daných údajov. Výsledkom má byť metóda, ktorá využitím týchto informácií slúži ako podporný nástroj, ktorý sa používa pri návrhu a implementácii, ale je zameraný na vylepšenie dopadu údržby. Všetky informácie uvedené nižšie by sa mali získavať z predchádzajúcich, ideálne dokončených projektov, ktoré sú z určitého pohľadu podobné aktuálne riešenému projektu.

- Čas pridania návrhového vzoru – často dochádza k situácii, že pri návrhu dátových modelov kvôli zdanlivej jednoduchosti sa analytik rozhodne nepoužiť návrhový vzor. V mnohých prípadoch sa neskôr ukáže, že použitie vzoru je predsa len potrebné. Dôsledkom je opätovné vytvorenie príslušnej časti modelu, v prípade, že sa projekt nachádza v neskoršej etape aj implementácie. Takáto modifikácia systému vyžaduje významné množstvo času, preto by sme sa mali snažiť predísť takéto situácie. História zmien vo VCS udržiava čas a presné kroky modifikácie kódu. Táto zmena sa dokonca aj ľahko identifikuje, nakoľko sa v podobnej situácii k zmene priraďuje správa (*commit message*), ktorá presne

definuje, že ide o pridanie návrhového vzoru. Ak pri návrhu modelu berieme do úvahy takéto zmeny z predchádzajúcich projektov podobného charakteru, môžeme si vopred uvedomiť potrebu použitia vzoru.

- Vybavenie požiadavky na zmenu – požiadaviek na zmenu (*change request*) sa týka pridanie, resp. modifikácia funkcionalít. Pre každú funkcionalitu je charakteristická daná oblasť zdrojového kódu. Pri upravovaní danej funkcionality musia vývojári pracovať nad určitou množinou súborov. Zdrojové súbory patriace k jednej funkcionalite sú vďaka tomu relevantné. Využitím tejto vlastnosti môžeme vytvárať relácie medzi časťami kódu, ktoré boli modifikované v rámci jednej zmeny. Takto získané množiny môžeme priraďovať k jednotlivým funkcionalitám, prípadne požiadavkám na zmenu. Pomocou tohto mapovania dokážeme redukovať čas potrebný na vykonanie zmeny, pretože pri riešení požiadavky máme vopred danú oblasť kódu, ktorú treba meniť. Ďalej pri riešení projektu, ktorého časti sú podobné, prípadne do istej miery totožné s iným projektom, informáciu o relevantnosti zdrojových súborov môžeme využiť na organizáciu kódu, napr. pri vytváraní balíkov tried.
- Hlásenia chýb, refaktoring – pokiaľ je projektový manažment na dostatočne vysokej úrovni, k jednotlivým zmenám vo VCS je priradená aj príslušná úloha, ktorá sa danou zmenou riešila. Taktiež pri nich môžu byť uvedené nahlásené chyby, ktorých odstránenie vyžadovalo zmenu príslušnej časti kódu. Pokiaľ máme informácie takéhoto charakteru o iných podobných projektoch, môžeme sa z chýb poučiť a pokúšať sa vytvoriť systém tak, aby sa známe nedostatky nevyskytli. Platí toto aj pre refaktoring návrhu, resp. implementácie. Ak vieme, že sa v podobných situáciách stáva, že hotový kód treba skrátiť, rozdeliť na viac častí alebo vykonať nad ním inú zmenu, môžeme vopred eliminovať potreby modifikácie.

Ako podávať?

Samotné informácie získané z histórie zmien ešte nemôžeme považovať za podporný nástroj, preto relevantné údaje musíme spracovať a vytvoriť pomocou nich prostriedok, ktorý môžeme reálne použiť na prevenciu chýb pri návrhu a implementácii.

Za ideálne riešenie považujem model evolúcie softvéru. Modelovanie evolúcie je zatiaľ čiastočne neprebádaná, alebo prinajmenšom nečinná oblasť softvérového inžinierstva. Model evolúcie obsahuje najpravdepodobnejšie a najdôležitejšie zmeny, ktorými softvér počas vývoja prejde. Optimálne by sa mal vytvárať počas návrhu, ktorý by mohol byť priamo ovplyvnený predpokladaním zmien. Princíp modelovania evolúcie spočíva v porovnávaní aktuálne riešeného projektu so starším, pričom sa snažíme nájsť tie najrelevantnejšie [2]. Východiskom je história zmien týchto podobných projektov.

Hoci modelovaniu evolúcie softvéru sa v súčasnosti venuje, počas štúdovania tejto techniky som mal dojem, že sa nachádza iba v experimentálnom štádiu. Aj keď zatiaľ neexistuje overený spôsob vytvárania a používania takýchto modelov, myslím si, že táto oblasť má potenciál a na jej vývoj je evidentný dopyt – pokiaľ sa nenájde iný spôsob na zefektívnenie údržby.

Vízia budúcnosti

Podľa mojich skúseností, ktoré zodpovedajú aj všeobecným poznatkom a štatistikám, údržba je práve tou oblasťou softvérového inžinierstva, ktorá v súčasnosti potrebuje najviac zlepšenia. Myslím si, že oproti nej, podpora ostatných činností je na dostatočnej úrovni, preto by sa oplatilo venovať väčší význam vývoju nástrojov na podporu neskorých etáp. Je však potrebné, aby ľudia nepovažovali údržbu softvéru objektívne za „najťažšiu" a aby si uvedomili, že ako všetko, aj toto sa dá vylepšiť.

Čo sa týka blízkej budúcnosti, viem si predstaviť, že popri klasických UML, CASE nástrojoch a ostatných prostriedkoch budeme pri návrhu bežne používať históriu zmien predchádzajúcich projektov a modelovanie evolúcie. Konečný dopad by pravdepodobne spočíval vo zvýšenej náročnosti analýzy, návrhu a implementácie, toto by ale prinieslo revolúciu v softvérovom inžinierstve tým, že by výrazne znížilo náročnosť údržby. Možno čoskoro nastane doba – nová kapitola v dejinách projektového manažmentu, kedy skrátenie potrebných úsilí bude globálne pozitívne vplývať aj na cenu softvérových produktov. Je zrejmé, že sa asi nikdy nedopracujeme do stavu, že implementáciou a otestovaním softvéru bude projekt ukončený, ale je viac než pravdepodobné, že takáto metóda by vo veľkej miere mohla vyrovnať súčasný veľký rozdiel náročnosti skorých a neskorých etáp vývoja softvéru.

Použitá literatúra

1. Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., Tan, W.-G.: Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution Research and Practice*, John Wiley & Sons, Ltd (2001), 3-30.
2. Ducasse, S., Girba, T., Favre, J.: Modeling Software Evolution by Treating History as a First Class Entity. *Electronic Notes in Theoretical Computer Science*, Elsevier (2005), 75-86.
3. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution Research and Practice*, John Wiley and Sons (2006), 207-236.
4. Ying, A. T. T., Murphy, G. C., Ng, R., Chu-Carroll, M. C.: Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, Published by the IEEE Computer Society (2004), 574-586.

Annotation

Software Maintenance – Complex or Neglected?

The main point of this essay is a reflection on the causes and possible solutions to the complexity of software maintenance. Work includes subjective opinions on the general approach to the development. Errors in ways of solving of individual tasks are searched. The idea of a fictitious future support instrument is discussed. The content of this text is the result of research and personal experiences of the author.