JE SOFTVÉR NA PODPORU VÝVOJA SOFTVÉRU NAOZAJ PODPORUJÚCI?

Menej môže byť niekedy viac.

Matúš Michalko

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

Abstrakt. *Prostriedkov podporujúcich vývoj softvéru začína byť na trhu akosi mnoho. S ich využitím sa vývoj stáva jednoduchším, organizovanejším a príjemnejším. Zdokonaľovanie týchto softvérových produktov, pridávanie nových možností a zohľadňovanie mnohých rôznorodých požiadaviek však zvyšuje ich zložitosť, čas potrebný na ich osvojenie, čas strávený ich používaním a v konečnom dôsledku aj celkovú produktivitu tímu. Otázkou teda je, do akej miery je tento softvér užitočný a na akých projektoch, a ako si vedieť zvoliť ten správny produkt. Kým napríklad softvér umožňujúci manažment verzií zdrojového kódu a spoluprácu viacerých vývojárov na tom istom projekte sa stáva nevyhnutnosťou už aj na menších projektoch, k ostatným podporným prostriedkom sa každý stavia inak. V tejto eseji sa zamýšľam nad niektorými možnosťami a nápadmi na zlepšenia systémov na manažment verzií zdrojového kódu, systémov na sledovanie zmien a na iné technické stránky sprevádzajúce samotný vývoj softvérových produktov.*

Kľúčové slová: *manažment podpory vývoja, konfigurácia softvéru, manažment verzií zdrojového kódu, sledovanie zmien*

Ak niečo robíš sám, robíš niečo asi zle...

Každý nádejný softvérový projekt si pre svoju úspešnosť vyžaduje viac, ako len jednu zaneprázdnenú hlavu zaborenú v problémoch každodenného života. Práve preto je tímová práca v skupine tých správnych ľudí často obrovskou silou, ktorá vie uviesť do

pohybu veľké veci. Možno je to svojím spôsobom ako v športe – vo futbale, hokeji, alebo v čomsi podobnom. Akurát, pri tvorbe softvéru je často naša „lopta“ neviditeľná a družstvo bez dostatočného technického vybavenia a pomoci odsúdené na prehru...

Vývoj softvéru v tíme je pomerne náročnou úlohou. Tvorba softvéru je príliš abstraktným procesom, náročným na sledovanie, nie to ešte na správne manažovanie. Práve za týmto účelom vzniklo množstvo podporného softvéru, ktorý sa snaží tieto úlohy čo najviac zjednodušiť a zautomatizovať. Množstvo funkcií, ktoré tento softvér často so sebou prináša, však nemusí byť vždy prínosné. Čím má totiž ľahšiu úlohu projektový manažér, na tým viac nástrojov a činností sa musí sústrediť bežný programátor. Tým pádom je aj jeho produktivita niekde inde, ako by snáď mohla byť. Alebo sa mýlim? Každopádne, predošlé skúsenosti získané podieľaním sa na tvorbe projektov menších rozsahov v menších tímoch mi dali pocítiť, aké to je, keď sa vyvíja takpovediac „na kolene“, a aké to je, keď sa naopak, možno až príliš veľa času utopí činnosťami, ktoré nakoniec neprinesú očakávané ovocie. Môže byť snáď niečo menej motivujúce, ako keď človek na konci dňa po sebe nevidí prínos aj napriek množstvu odvedenej, no nie priamo prínosnej práce? Nájsť však tú správnu mieru nemusí byť vôbec ľahké. Sto ľudí, sto chutí. A napokon každému môže byť na osoh niečo úplne iné...

Poďme spolu vyvíjať

Prvá vec, ktorá ma vždy pri spoločnom vývoji softvéru napadne, je *nástroj na manažment verzií zdrojového kódu*. Z môjho pohľadu je to vec, ktorá je nevyhnutná na každom projekte, v ktorom sú zapojení čo už len dvaja ľudia. Nástroje na udržiavanie verzií zdrojového kódu tu už sú nejakých tých pár rokov, za ktorých sa hádam podarilo vychytať väčšinu múch. Spočiatku boli k dispozícii len centralizované systémy (ako napr. SVN), no v poslednej dobe sa čoraz viac do popredia dostávajú systémy založené na decentralizovanom prístupe (napríklad nástroje *GIT* alebo *Mercurial*). Pri výbere tohto softvéru tu teda možno vzniká nová dilema - po ktorom z týchto nástrojov siahnuť?

Kedysi som bol skalopevným fanúšikom *SVN*. Je to totiž pomerne jednoduché, účelné, a preto aj obľúbené a hlavne rokmi overené riešenie. Človek si k sebe raz stiahne repozitár, a potom sa už jeho život točí okolo troch jednoduchých operácií – aktualizácie lokálnej pracovnej verzie, odovzdania implementovaných zmien, no a z času na čas manuálneho vyriešenia konfliktu pri spájaní súborov. Vďaka množstvu dostupných grafických rozhraní tieto operácie prebiehajú celkom jednoducho a bezproblémovo. Vyzerá to myslím efektívne, bezstarostne a použiteľne. Prečo potom bolo potrebné vymýšľať ešte niečo nové?

Pri decentralizovanom prístupe sa život programátora komplikuje o udržiavanie lokálneho repozitára a jeho synchronizáciu s tým globálnym. Z môjho pohľadu sa zrazu tri jednoduché operácie rozšíria na päť a pribúdajú starosti s udržiavaním rôznych vývojových vetiev. Filozofia tohto prístupu je mierne iná, no mám pocit, že pri centralizovanom prístupe sa dal dosiahnuť rovnaký cieľ s oveľa menším úsilím (menším počtom krokov). A aby nebolo málo, ak človek siahne po nástroji *Git*, stráca možnosť grafickej vizualizácie jednotlivých vetiev a nejakého rozumného grafického rozhrania vôbec. Priznám sa, že si celkom dobre ani neviem predstaviť, ako je potom v tomto nástroji možné zorientovať sa vo väčšom projekte s množstvom vývojových vetiev. Existujú síce

služby ako napr. *Github*, ktoré poskytujú grafické webové rozhranie, no nie každému môže vyhovovať mať svoj repozitár umiestnený kdesi mimo vlastnej siete. Načo je teda potom celý tento chaos dobrý? Treba si uvedomiť, že určite nič na tomto svete sa nedeje len tak. Verím, že každá nevýhoda neskôr prináša svoju výhodu a každé menšie nepohodlie a ťažkosti môžu neskôr prerásť do väčšieho dobra... Siahnutie po decentralizovanom nástroji by tuším mohlo priniesť pomerne veľa ovocia: možnosť častejšieho odovzdávania zmien do lokálneho repozitára a predovšetkým zľahčenie vývoja softvéru vo viacerých vetvách. Myslím si, že práve práca s vetvami pri centralizovanom prístupe je dosť kostrbatá a nepohodlná. Používaním centralizovaného nástroja som nadobudol pocit, že sa zrejme hodí skôr na projektoch, kde sa očakáva inkrementálny vývoj nad známymi technológiami, kde sa nepredpokladá vytváranie viacerých prototypov, alternatív, alebo verzií toho istého produktu.

Vytváranie viacerých vetiev možno zo začiatku znie mierne mätúco a ako zbytočná strata času. Sú ale vetvy naozaj také neužitočné? Myslím si, že práve naopak. V istých prípadoch prinášajú mnoho zaujímavých výhod. Najmä pokiaľ je vyvíjaný softvér konfigurovateľný, vydáva sa vo viacerých verziách, prípadne pokiaľ je priebežne integrovaný. (Vývojári môžu pokojne pracovať na svojej vetve, zatiaľ čo v tej stabilnej sa môžu udržiavať len poriadne otestované verzie, dostupné verejnosti). Nemusí to teda bezpodmienečne znamenať výrazne sťaženie práce programátorov. Čím sa však jednoduchý proces odovzdávania implementovaných zmien komplikuje, tým vzniká viacej priestoru, v ktorom môžu nastať nečakané komplikácie. Napríklad zlé spojenie vetiev, prípadne vytvorenie viacerých hláv tej istej vetvy sú pomerne nepríjemné veci, pred ktorými je „sedliacky“ svet centralizovaného prístupu ochránený.

Ako možno medzi riadkami vycítiť – možno sa ani nedá s istotou povedať, ktorý nástroj je vhodný viac a ktorý menej. Existuje množstvo porovnaní a množstvo názorov, kde sa hodí ktorý prístup viac. No isté je len jedno - do procesu výberu tohto nástroja vstupuje množstvo premenných a je to už len otázkou toho správneho rozhodnutia, ktorá premenná je nakoľko dôležitá. Je to skoro ako dvojramenná váha, kde sú na jednej strane programátori a na druhej ľudia zodpovední za manažment jednotlivých verzií a konfiguráciu výsledného softvéru. Buď sa táto váha mierne nakloní jedným, alebo druhým smerom. Alebo nakoniec možno predsa len všetci ťahajú za ten istý koniec lana?

Každopádne, ak by som si mal vybrať, tak v tíme, kde nie sú skúsenosti s manažmentom verzií zdrojového kódu a kde sa predpokladá, že ľudia budú pracovať na nezávislých komponentoch, siahol by som po centralizovanom prístupe a využil jeho jednoduchosť. V prípade náročnejšieho softvéru, kde by sa možno predpokladalo experimentovanie s viacerými alternatívami a s vytváraním prototypov (chtiac-nechtiac aj tých na zahodenie), možno by som radšej umožnil každej myšlienke a každému nápadu vyrásť vo vlastnej vetve. Neskôr by to malo byť aj jednoduchšie vrátiť sa k inej alternatíve, k inému riešeniu.

Čo všetko chceme „verziovať“?

Keď sme pred pár rokmi začali vyvíjať centrálny informačný systém v jednom menšom občianskom združení tvorenom výlučne študentmi, pustili sme sa do programovania s nadšením, no bez nejakej prvotnej analýzy a tvorby UML modelov. Keď sme si po čase

uvedomili, že všetci danému systému rozumieme trochu inak, začalo nám byť jasné, že sme podcenili analýzu a nevytvorili potrebné modely. Za túto chybu sme síce všetci zaplatili vlastným časom, no bola pomerne ľahko opraviteľná. Vytvorením potrebných modelov a diagramov však celkom neskončila, ako by sme to vtedy čakali. Tie sa totiž časom menia a ich udržiavanie nemusí byť vôbec jednoduché. Práve tomuto problému sa venuje stále čoraz väčšia skupina výskumníkov, ktorí prišli s nápadom tieto modely verziovať s ohľadom na ich grafovú štruktúru [4]. Je známe, že samotný vývoj softvéru už od etáp analýzy a návrhu sprevádza množstvo rôznych modelov a diagramov na rôznej úrovni abstrakcie, ktoré sa v čase často menia. Myslím si, že je potrebné pripustiť, že súčasné verziovacie systémy, ktoré sú založené na verziovaní a spájaní textov (zdrojového kódu), nie sú práve najvhodnejšie pre uchovávanie jednotlivých verzií samotných UML modelov a diagramov, ktoré sú väčšinou grafového charakteru.

Na druhej strane, ak sa zamyslíme nad spôsobom manipulácie s jednotlivými modelmi a zdrojovým kódom, možno to až taký veľký problém nie je. Z môjho pohľadu na vývoji modelov a ich údržbe obvykle nepracuje tak veľa ľudí, ako na implementácii. Odhadujem, že na projektoch menšieho rozsahu bude za samotné modely zodpovedný jeden, nanajvýš dvaja ľudia. Verziovanie UML modelov teda bude možné aj v súčasných prostriedkoch – akurát nebude možné dynamické sledovanie zmien medzi verziami. Otázkou tiež je, nakoľko by toto aj bolo v projektoch menších rozsahov užitočné. Prikláňam sa k mottu tejto práce: „menej je niekedy viac“. Myšlienka verziovania modelov je určite na mieste, no myslím si, že nemusí byť chybou, ak sa na verziovanie použijú tradičné spôsoby. Dôležité z môjho pohľadu ale je, aby sa použili – aby mali programátori k dispozícii stále aktuálnu špecifikáciu, ideálne v adresárovej štruktúre spolu pri zdrojových kódoch a nemuseli sa zaťažovať zháňaním najaktuálnejších dokumentov.

Navyše v prípade použitia moderných softvérových rámcov založených na modelom riadenej architektúre s využitím návrhového vzoru „model - pohľad - ovládač“ (ako sú napríklad softvérové rámce založené na vzore aktívneho záznamu) je možné mnoho vyčítať už priamo zo zdrojového kódu. Už len triedy modelu zastupujúce jednotlivé entity pomerne jasne opisujú ich vzťahy k ostatným entitám. Tento opis má z môjho pohľadu určite väčšiu dôveryhodnosť ako diagram, ktorý možno nemusí byť aktuálny. Myslím si, že týmto prístupom sa takýto zdrojový kód stáva samoopisujúcim, jasným, a zároveň sa znižuje potreba manuálneho udržiavania nadbytočnej dokumentácie (čo sa ukazuje byť vhodné obzvlášť pri agilnom spôsobe vývoja softvéru). V konečnom dôsledku sa tým šetrí aj čas, aj peniaze a zvyšuje produktivita tímu. Mám pocit, že čím menej softvéru je nútený vývojár používať, tým lepšie. A to nielen pre neho, ale pre všetkých. Navyše, keď vývojári po čase zistia, že daná dokumentácia nie je dostatočne udržiavaná a konzistentná s aktuálnym stavom, stratí svoju dôveryhodnosť a účel. Jej iteratívne udržiavanie si vyžaduje ďalšie zdroje, čas a úsilie, ktorých nemusí byť pri každom projekte dostatok...

Jedným z ďalších problémov, ktoré sa nepriamo samotného verziovania kódu týkajú, je aj spätné sledovanie úsilia, ktoré bolo jednotlivými vývojármi vynaložené pri riešení daného problému. Často sa totiž stáva, že autori riešenia poskytujúci riešenie vo forme zmeny (opravy), ostanú prehliadnutí, nakoľko táto zmena bude viditeľná len v softvérovom artefakte odovzdanom programátorom, ktorý ho uplatnil. Klamlivo sa tak môže vytvárať ilúzia, že hlavný vývojár odvádza oveľa väčšie množstvo práce, keďže aplikovanie čiastkových zmien (opráv) a spájanie čiastkových kódov býva obvykle jeho

úlohou. V diele [3] autori skúmajú možnosti anotovania jednotlivých odovzdaní softvérových artefaktov a získavania z nich dodatočných informácií súvisiacich s riešením takýchto úloh. Spomínané dielo zavádza možnosť zaznačenia pôvodcu problému, ako aj jeho riešiteľa, čím sa podarilo identifikovať v priemere o 40% viac ľudí podieľajúcich sa na riešení daného problému. Vzhľadom na to, že podobná situácia už nastala aj v našom tíme, verím, že zavedenie tohto prístupu by mohlo byť užitočné už aj z pohľadu monitorovania a spätného sledovania vynaloženého úsilia. V prípade, že sa však v nástroji plánovania jednotlivé úlohy dostatočne dekomponujú, možno nemusí byť táto činnosť až taká potrebná. V konečnom dôsledku neverím, že sa všetko dá vyčítať zo sprievodných správ pri odovzdaní softvérových artefaktov. Niekedy môže aj pár riadkov kódu predstavovať komplikované riešenie náročného problému...

Je v úlohách programátorov neporiadok?

Zmenami a vylepšeniami prechádza nielen softvér na manažment verzií zdrojového kódu, ale aj samotné systémy podporujúce sledovanie zmien a prideľovanie jednotlivých úloh členom tímu. V posledných rokoch sa ich smerovanie uberá čoraz viac sociálnym smerom s cieľom podporiť produktívnu komunikáciu v tíme a umožniť riešenie úloh čo najefektívnejším spôsobom a v najvhodnejšom poradí. Podobný cieľ sledovala aj skupina výskumníkov navrhujúcich značkovanie jednotlivých úloh [5], prípadne druhá skupina, ktorá sa snažila o ich automatizované triedenie [1]. Kým prvý prístup otvára novú možnosť kategorizácie jednotlivých úloh z viacerých nezávislých pohľadov a tým pádom lepšiu organizáciu a zhlukovanie úloh do skupín, vzniká aj väčšie riziko nesprávneho zadelenia danej úlohy, prípadne vzniku duplicitných označení pre tú istú kategóriu problému. Je otázne, do akej miery je vhodné jednotlivé úlohy riadenia projektu decentralizovať medzi členov tímu a do akej miery je vhodné prenechať túto úlohu na zodpovednú osobu. Jeden človek na projektoch väčšieho rozsahu totiž nemusí mať dostatočne veľký prehľad o všetkých jeho oblastiach a všetkých čiastkových problémoch. Tým pádom kategorizácia danej úlohy (alebo reportovanej chyby) môže byť pre neho náročná. Druhý spomínaný prístup prichádza s takmer protipólnym opatrením, snažiacim sa zadeľovanie úloh a žiadostí o odstránenie chýb takmer úplne automatizovať na základe možností umelej inteligencie a extrakcií informácií z opisu úloh. V tomto prístupe sa obávam viacerých nástrah, najmä v uplatniteľnosti pri jazykoch, kde rozpoznávanie častí reči zatiaľ nedosiahlo požadovanú úroveň. Rozpoznávanie kľúčových slov v jazykoch so skloňovaním totiž nemusí byť triviálnou úlohou, navyše môže hroziť nesprávne kategorizovanie úlohy. V takom prípade hrozí jej „zabudnutie“ niekde na spodku zoznamov a odklad jej riešenia aj napriek tomu, že sa môže jednať o pomerne kritickú požiadavku na opravu chyby. Umelá inteligencia je zaujímavý smer, ktorý nám podstatne zlepšuje život, no v tomto prípade sa obávam, či nie je ešte predsa len príliš skoro. Naopak, spomínané manuálne značkovanie úloh považujem za celkom užitočnú činnosť. Nezaberie veľa času, nenúti vývojára používať ďalší softvér a hádam prinesie svoje ovocie. Na druhej strane, nápad s automatizovaným zadeľovaním úloh je určite fajn, keďže sľubuje ušetrenie času počas plánovania, ktoré sa už aj na našom projekte ukázalo byť pomerne náročnou a zdĺhavou činnosťou. Ak by sa daný prístup využil len na úrovni

odporúčania, ktoré by pomohlo rýchlejšie si utvoriť obraz o reportovanej úlohe, mohlo by sa jednať o pomerne prínosnú možnosť softvéru sledovania zmien.

Čo sa vie v systéme a čo vieme my?

Ďalšou pomerne dôležitou vlastnosťou softvéru podpory vývoja je možnosť uchovávania poznatkov. Práve tejto problematike sa venovali autori diela [2], kde sa uvádza, že často medzi tým, čo je obsiahnuté v báze poznatkov a tým, čo je vypovedané v samotnom kóde aplikácie, vo zvolených technológiách, vykonaných nastaveniach a použitých metódach je veľká priepasť. Myslím si, že počas samotného vývoja softvéru sa ľudia sústreďujú viac na produkt ako na samotnú dokumentáciu, čo možno koniec koncov ani nie je až také zlé. Po dlhšom čase životného cyklu softvéru však môžu na povrch vyplávať pôvodné zabudnuté problémy, ktorých riešenie je už zabudnuté. Hľadanie ich príčiny a spôsobov riešenia tak môže pripomínať znovuobjavovanie Ameriky. Často medzi vývojármi tiež vzniká generačná medzera. Problémom je teda často aj odovzdávanie informácií zo starších, skúsenejších členov tímu na nové, čerstvé, „programátorské uchá“. Títo noví a menej skúsení (tým nechcem povedať, že aj menej šikovní ľudia) tak musia často hľadať odpovede na otázky, ktoré už predtým raz niekto v tíme našiel. Isto, nevedieť a pýtať sa vôbec nie je hanba, no komunikácia je často dosť spomaľujúca, málo produktívna a navyše z pohľadu nových ľudí si vyžaduje aj istú dávku odvahy. Isto všetci poznáme ten nesmelý nováčikovský pocit, keď sme sa chceli poradiť s cvičiacim (starším spolužiakom), no zďaleka sme si neboli istí, či sa práve nejdeme opýtať na najsamozrejmešiu vec na svete, ktorá nám mala byť dávno jasná už pred rokmi. Tento problém nachádza svoje miesta asi skôr na väčších a dlhotrvajúcich projektoch. Náš tímový projekt zrejme nebude dostatočne dlhý na to, aby sme v ňom mali príležitosť znovuobjavovania Ameriky, no myslím si, že agilná metóda vývoja a samotný Scrum nepriamo k takémuto stavu smerujú. Dokumentovanie a činnosti neprinášajúce priamy výstup sú pri tomto spôsobe vývoja často odsúvané na druhú koľaj. Práve preto si myslím, že by mohlo byť prospešné, ak by si každý člen tímu osvojil pridávanie nových poznatkov vecným a prehľadným spôsobom do spoločnej bázy znalostí ako rutinnú činnosť nezaberajúcu dlhší čas a udržujúcu poriadok vo vývoji – možno aj po každej dokončenej úlohe.

Záver

Menej môže byť niekedy viac, no treba vedieť nájsť tú správnu rovnováhu. Aj v menšom tíme pracujúcom na menšom projekte môže zanedbanie návrhu, UML modelov a CASE nástrojov urobiť nie malú škodu. Otázkou však ostáva, či je potrebné sofistikované verziovanie modelov, alebo sa môžu modely verziovať tradične spolu so zdrojovým kódom. Odpoveď bude iste závisieť od veľkosti projektu a jasnosti požiadaviek.

Samotný Scrum uprednostňuje výsledok pred dokumentáciou, no domnievam sa, že je kľúčové nájsť správnu odpoveď na otázku, čo je jasné z kódu, a čo by bolo predsa len lepšie niekde spísať, aby sa v budúcnosti predišlo znovuobjavovaniu Ameriky.

Jedným z prvých krokov pri zakladaní projektu je aj voľba nástroja pre manažment verzií zdrojového kódu. Jedná sa o rozhodnutie, ktoré môže mať pomerne vážny dopad na

spôsob vývoja projektu. Určite je vhodné zvážiť „pre a proti“ dostupných prostriedkov a nájsť riešenie, ktoré vývoj daného produktu zjednoduší a nie naopak skomplikuje.

Samotné zadeľovanie a kategorizovanie úloh zohráva rovnako dôležitú úlohu, ako správne verziovanie kódu, či tvorba dokumentácie. Verím, že pridanie sociálneho aspektu do procesu tvorby úloh formou spoločnej kategorizácie môže byť pre tím dlhodobejšie prínosné, rovnako ako správne anotovanie odovzdaní softvérových artefaktov, ktoré tak zabráni „neviditeľnosti“ niektorých prác na projekte.

Myslím si, že pred každým rozhodnutím ohľadom ďalšieho softvéru (alebo metódy), ktorú sa chystáme v projekte využiť je dobré sa najprv zamyslieť, či prinesie požadované ovocie, keďže niekedy je možno predsa len najlepší jednoduchý sedliacky rozum, a niekedy môže byť predsa len menej viac...

Použitá literatúra

1. Antoniol K. A., Ayri K., Penta M. et al.: Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests. In: *CASCON '08 Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. New York, NY, USA, 2008, Article No. 23.
2. Liu S., Fidel R.: Managing Aging Workforce: Filling the Gap Between What We Know and What Is in the System. In: *ICEGOV '07 Proceedings of the 1st international conference on Theory and practice of electronic governance*. New York, NY, USA, 2007, s. 121-128.
3. Meneely A., Corcoran M., Willams L.: Improving Developer Activity Metrics with Issue Tracking Annotations. In: *Proceeding WETSoM '10 Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*. New York, NY, USA, 2010, s. 75 - 80.
4. Nguyen, T. N.: Model based Version and Configuration Management for a Web Engineering Lifecycle. In: *Proceedings of the 15th international conference on World Wide Web*. New York, NY, USA, 2006, s. 437-446.
5. Treude, C., Storey M-A.: How Tagging helps bridge the Gap between Social and Technical Aspects in Software Development. In: *ICSE '09 Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA, 2009, s. 12 – 22.

Annotation

Are tools for software development support really supporting?

Current market offers us a wide variety of various supporting tools that should make software development process much simpler, organized and cleaner. Amount of these tools and amount of their features is quite huge. Enriching these software products makes them more and more complex and harder to use. The more features these tools offer, the longer their learning curve is, moreover, overall time spent by their usage is increased during whole project development life cycle. Question is "which tools and which features are necessary and what's their real contribution" Some tools like version control systems are "must have" software tools even on small projects, while advantages of other tools can be discussed. The decision about right tools that will be used on a project might have serious impact on whole project development. This essay discuss features and importance of various

software tools, their possible difficulties and ideas how to make them even better and more supportive.