NĚKOLIK POZNÁMEK O METODĚ EXTREME PROGRAMMING

Petr Štěpánek

Fractal s.r.o., petr.stepanek@fractal.cz, http://www.fractal.cz

Abstrakt

Článek formou popisu vybraných vlastností seznamuje s metodou extrémního programování a vysvětluje možnosti jeho použití. Text se opírá o dvouleté praktické zkušenosti autora s touto metodou ve firmě Fractal s.r.o.

Klíčová slova: extrémní programování, požadavky na systém, implementace systému, pair programming, testování, refactoring, znovupoužitelnost, návrhové vzory

Úvod – proč vzniklo XP?

Extrémní programování (XP) vzniklo v roce 1997 jako nová metoda tvorby softwaru. Do nynějška totiž neexistovala metodika, která by dokázala využít nové vlastnosti vývojových prostředí programovacích jazyků, výhodných vlastností objektového přístupu, a vyrovnala se s faktem, že zákazníci na začátku projektu nedokáží přesně specifikovat svoje požadavky.

XP je stavěno pro svět tvorby softwaru takový jaký je, a nikoliv pro jeho ideál, jak by jej analytici rádi viděli. Současné metody, ať píší o iteračním modelu vývoje jak chtějí, provádějí analýzu, návrh, implementaci a testování celého systému a nerady připouštějí změny v průběhu implementace (velmi častá praxe, čest výjimkám). K čemu to spěje? Zákazník určí do kdy chce mít SW hotový, určí rozpočet, dodavateli sdělí, ať ukáže co umí a podepíše kontrakt. V průběhu projektu ale zákazník své představy mění. Protože ona obávaná „exponenciální“ křivka udávající vztah mezi stupněm hotovosti projektu a cenou změny pevně drží a platí, projekt se zdražuje, prodlužuje, programátoři doma nestíhají večeře, zadavatel nadává, … a nebo vývojáři tuší jak projekt může dopadnout, provádějí zdlouhavou a komplikovanou analýzu vydají ze sebe mnoho úsilí, vyrobí spoustu diagramů, se kterými obtěžují zadavatele, a k implementaci se přistupuje pozdě, se spěchem a s únavou. XP se snaží tomuto zabránit. Jak?

XP vývoj

Fakt života: *SW je živý organismus, který je velmi často měněn.* Proto se programátoři nesmí změn svého díla bát, změnami nesmí způsobovat chyby, změny musí být schopni provádět rychle, ... Potřebujeme tedy metodiku, která se dokáže vyrovnat s tím, že zákazníci na začátku projektu nemusí přesně vědět, co chtějí. Kdo se tváří učeně, že klasickou metodikou tento problém řeší, nemá pravdu. XP chápe vývoj SW jako neustálou změnu od stavu „*nic není vyvinuto*“ do stavu “*zákazník má vše co chtěl a je spokojen*”. Podstatný rozdíl oproti stávajícím metodám je ten, že při použití stávajících metod zákazník dostane to, co chtěl na začátku vývoje, kdežto při použití XP zákazník dostane to, co chtěl na konci vývoje, až poté, co se jeho požadavky vyvinuly a on si je ujasnil.

Ukazuje se, že mnoho projektů končí na nedostatku kuráže měnit kód („…já tam klidně sáhnu, ale něco rozbiji… a co pak?“ nebo „…se snad zbláznil ne? Vždyť to ještě včera chtěl úplně jinak! A už vůbec to nesedí s tím co chtěl na začátku, a to jsme si den ujasňovali jak to vlastně má být! A teď mi dokázal že jsem ho stejně pochopil špatně…“).

Jak zajistíme aby programátoři měli kuráž měnit svůj kód, aby se připravili na to, že zákazník bude měnit své zadání? Jak vlastně vypadá XP vývoj? Vzhledem k rozsahu příspěvku uvedeme jen přibližné vysvětlení (více literatura).

Zadavatel sepíše na kartičky své představy o požadovaných funkčnostech (požadavcích, oficiálně *user stories*), které má systém obsahovat. Tým společně odhadne, jak dlouho bude trvat každou z nich implementovat, a jaký dopad (např. kolik se bude muset změnit) má na stávající systém (*risk*). Zadavatel každému požadavku přiřadí *business value* (stylem “*musím mít*“, „*hodnotné, bylo by hezké mít*“ a „*nemusím mít*“) a seskupí požadavky do hromádek, kde jedna hromádka představuje cca 3 týdny reálné práce týmu. Každá hromádka představuje jednu iteraci, po které se celý proces opakuje. Toto je zjednodušeně vysvětleno proces *iteračního plánování*.

Pokud by byl zadavatel schopen dodat požadavky pro celý projekt (a my víme že není, protože jeho požadavky se budou vyvíjet spolu s projektem), mohli bychom z počtu hromádek určit odhad doby trvání projektu. S tím, jak se budou zadavatelovy požadavky měnit, bude se měnit i odhad doby trvání projektu (díky měření výkonnosti týmu se odhady budou zpřesňovat, až třeba jednu iteraci před dokončením díla budeme moci garantovat přesné datum). Toto většině zadavatelů vyhovuje, protože si nejvíce cení jistoty kvalitně odvedeného díla získané již z průběhu projektu. Jak zadavatel, tak i tým vždy vědí, co je na příští max. 3 týdny čeká.

Aby tým dosáhl kuráž, používá následující praktiky (uvedeme pouze výběr):

- *Worst things first* ⇨ požadavky s nejhorším riskem řešíme jako první. Jsme si vědomi existence exponenciální křivky, a snažíme se jí čelit aniž bychom omezovali uživatele v jeho požadavcích.
- *Unit tests* ⇨ pro každou třídu, metodu či celou komponentu musí existovat test. Programátoři si musí být jisti, že jejich kód funguje. Unit testy vlastní programátoři.
- *Functional tests* (*funkční* neboli *akceptační* testy) ⇨ těmi si zadavatel ověřuje, že program funguje tak, jak požadoval. Funkční testy vytváří a vlastní zadavatel.
- *Pair programming* ⇨ 2 oči vidí lépe, 2 mozky myslí lépe. Z dlouhodobého hlediska pracují 2 lidé u jedné klávesnice a jednoho monitoru lépe (kvalitněji, rychleji), než kdyby pracovali každý zvlášť. Jinak to není správné párové programování). Všechen produkční kód musí být tvořen párem. Párování zároveň vyučuje. (slabší se učí od zdatnějšího - ukazuje se ale, že toto platí i naopak)
- *Continuous integration* ⇨ integrace probíhá po dokončení práce na každém uživatelském požadavku. Všechny *unit* testy před i po ní musí běžet na 100%!
- *Collective ownership* ⇨ všechen kód je vlastněn týmem jako celkem, ne jednotlivými programátory. To nám umožňuje nečekat jeden na druhého, pokud potřebujeme změnu v kódu, který jsme nenapsali (testy odchytí případné kolize).
- *Do the simplest thing that could possibly work* ⇨ Neexistují obecné problémy (jen specifické), proto v naprosté většině případů nemá smysl dělat obecná řešení. Pokud projekt správě využívá vlastností OOP (hlavně návrhové vzory), je možné jej jednoduše a levně rozšiřovat v okamžiku, kdy se ukáže že nové funkce jsou třeba. Jak

poznáme co je třeba? Pokud naše testy bez *toho* běží, tak *to* potřeba není. To umožňuje stavět systém pro současné požadavky, nikoliv pro budoucí které možná přijdou, ale neví se kdy a zda vůbec.

- *Refactoring* ⇨ zde tkví kouzlo neustále udržitelného rozvoje SW. *Refactoring* je změna kódu beze změny jeho funkce. Proč to? Za účelem srozumitelného a snáze udržitelného kódu. "*Dělají dvě třídy (metody, komponenty, ...) v podstatě totéž? Kombinuj je až zbude jen jedna.* - takových pravidel existuje mnoho). Při refactoringu systém zpřeházíme a poskládáme jinak, lépe, čitelněji, zdravěji, ale jsme chráněni testy, které musí po refactoringu všechny projít na 100%!
- Důležité je používání návrhových vzorů, které jsou způsobem předávání znalostí. Návrhové vzory spolu s párovým programováním poskytují velmi výkonný nástroj komunikace znalostí mezi extrémními programátory.

Pokud připočteme schopnosti moderních programovacích prostředí (např. Smalltalk), které již zdaleka nejsou jen spojení kompilátoru s textovým editorem zdrojového kódu, jejichž vlastností XP využívá, tedy mimo jiné:

- snadný refactoring (Smalltalk má dokonce nástroj usnadňující refactoring),
- značná samodokumentace kódu (to samozřejmě závisí na nástroji, ve strojovém kódu vyrobeného klasickým kompilátorem pochopitelně žádná samodokumentace není a je třeba vytvářet dokumentaci externí),
- vysoká znovupoužitelnost hotových částí,
- snadná integrace kódu více programátorů (VCS, je součástí GUI Smalltalku) a
- známé výhody čistých OO jazyků, tak

pochopíme, že XP mění exponenciální křivku na logaritmickou. Je to naše zkušenost, je to také popsáno v literatuře.

Při neznalosti praktik XP může vzniknout dojem, že XP rovná se programování „nadivoko“. To je ovšem omyl, neboť kvalita výsledku je zajištěna mimo jiné kombinací:

- *Pair programming* (i testy jsou psány párem programátorů),
- *Unit testy* (dokud neproběhnou na 100%, tak kód nefunguje),
- *Funkční testy* - těmi si zadavatelé ověřují, že program funguje jak požadovali, a
- *Continuous integration* - vývoj probíhá po velmi malých a relativně kompaktních a jednoduchých kouscích v kooperaci se zadavatelem.

XP je především soubor vyzkoušených postupů a návodů pro etapu návrhu a implementace. V žádném případě neříká, že se nemá dělat analýza. XP není v rozporu s moderními metodami analýzy požadavků a procesního modelování (*requirement engineering, process modeling*).

XP je dokonce velmi přísná metoda, protože například:

- pro každou třídu, veřejnou metodu či komponentu musí být *unit* test,
- každý uživatelský požadavek (pokud zákazník chce, aby fungoval) musí být zajištěn funkčními testy,
- před uvolněním kódu do VCS musí všechny *unit* testy běžet na 100%,
- pokud v libovolném okamžiku sestavíme program ze zdrojových kódů z VCS, musí všechny unit testy běžet na 100%,
- chyba je pokud funkční test, který běžel, přestane fungovat a
- programátoři musí veškerý produkční kód psát v párech.

XP nespoléhá na věčně zastaralou dokumentaci. Jistě, pokud máme komponentu, která je složitá (z kódu, názvů tříd a metod není patrné, co dělá) není na škodu si pořídit konceptuální model jaké funkce realizuje. A máme i jistou šanci že tato dokumentace zastará poměrně pomalu. To ale neznamená, že je třeba komentovat každou metodu či třídu. Třídy a metody by samy měly mít takové názvy, aby z nich bylo patrné co dělají. To samo o sobě přispívá ke správné struktuře kódu. Teprve pokud to není možné, je třeba komentář a nebo i kreslit diagramy.

Závěr

Tento článek si kladl za cíl upoutat pozornost na metodu eXtreme Programming, nikoliv podat její přesný nebo výstižný výklad. XP může nezasvěcenci vypadat jako hackerství „opravdových programátorů“, a snažili jsme se ukázat, proč je takový názor mylný. XP dovoluje zákazníkovi mít kontrolu nad vývojovým procesem (konec konců jsou to jeho peníze) a rychle mu dodat SW který implementuje jeho požadavky (*Do the simplest thing that could possibly work*). Kvalita práce je v XP naprostou předností (testy). XP využívá moderních programovacích systémů (např. Smalltalk) a výhod objektového přístupu, ačkoliv jeho praktiky jsou jistě použitelně i jinde.

Závěrem bych chtěl poznamenat, že firma Fractal s.r.o. má s XP již přibližně dvouleté zkušenosti. Na XP jsem přešli po negativních zkušenostech s jinými přístupy. Také jsme si na vlastní triko vyzkoušeli co to znamená porušit některá pravidla XP, a že nikdo z týmu již nechce pracovat jiným způsobem. XP bychom pochopitelně nepoužívali, kdyby pro nás nefungovalo.

Literatura:

1. Extreme programming approach, www.xprogramming.com
2. Extreme Programming - a gentle introduction: http://www.extremeprogramming.org – jiná stránka o XP
3. Ron Jeffris: Extreme Programming Installed (The XP Series), Addison-Wesley; ISBN: 0201708426
4. Kent Beck: Extreme Programming Explained: Embrace Change (The XP Series), Addison-Wesley; ISBN: 0201616416
5. Jiří Berger, Petr Štěpánek: Extrémní programování v praxi, ve sborníku konference Objekty 2001, http://objekty.pef.czu.cz, ISBN 80-213-0829-X
6. Petr Štěpánek: Gemstone/S – popis a praktické zkušenosti při vývoji aplikací**,** ve sborníku konference Objekty 2001, http://objekty.pef.czu.cz, ISBN 80-213-0829-X