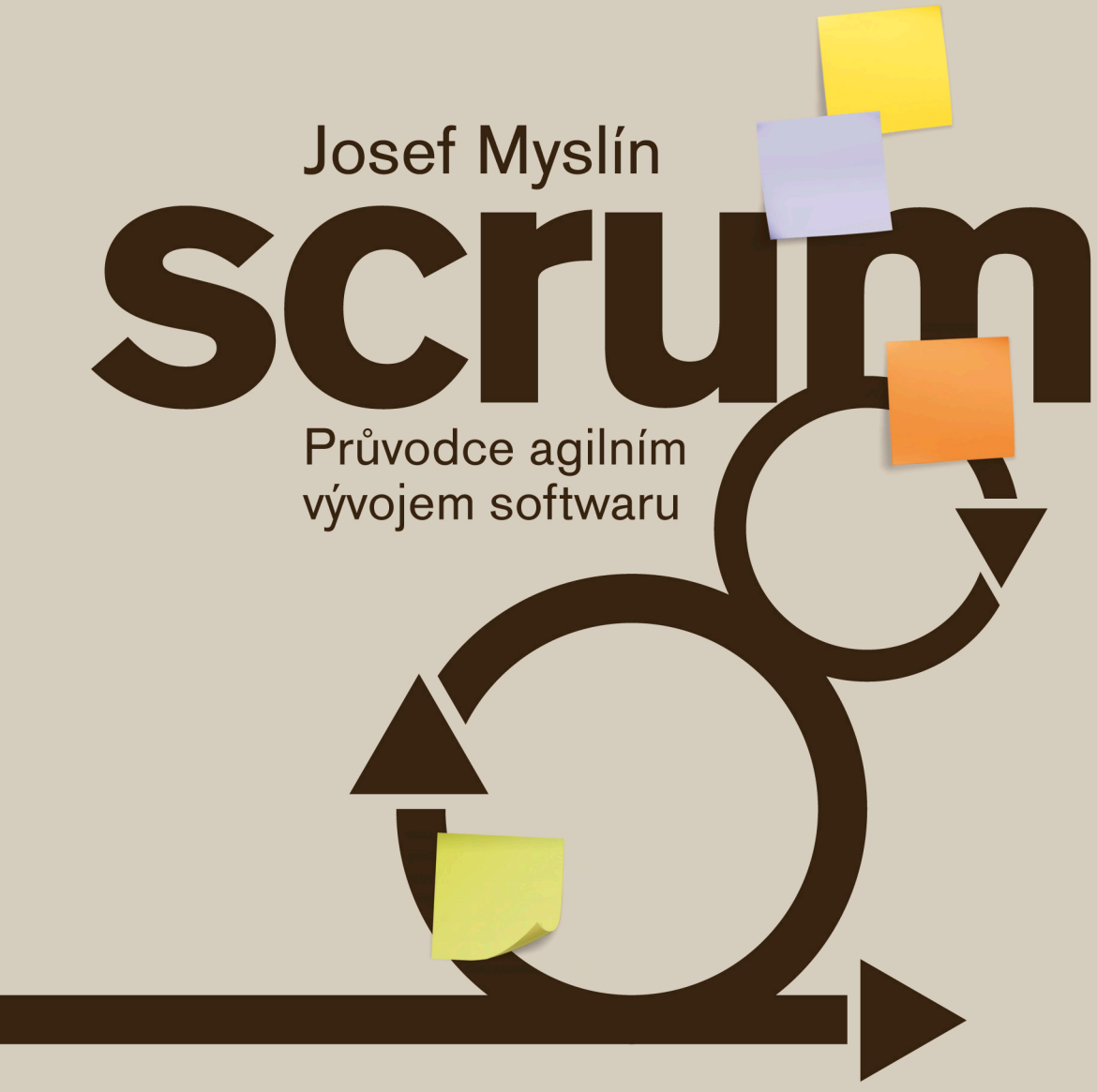
Josef Myslín

scrum

Průvodce agilním vývojem softwaru



computer press

Josef Myslín

Scrum
Průvodce agilním vývojem softwaru

Computer Press
Brno
2016

Scrum
Průvodce agilním vývojem softwaru

Josef Myslín

Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-4650-7

Vydalo nakladatelství Computer Press v Brně roku 2016 ve společnosti Albatros Media a. s.
se sídlem Na Pankráci 30, Praha 4. Číslo publikace 23 614.

1. vydání

ALBATROS MEDIA a.s.

Obsah

O čem je tato kniha?

Počítače a další prostředky informačních a komunikačních technologií se již dávno staly nedílnou součástí lidského života. Tak nedílnou, že vyhnout se jim v podstatě znamená rezignovat na výdobytky civilizace. Jsou všude – pokud bychom měli vyjmenovat, pak musíme začít s klasickými počítači v různých kabátech – stolní počítače, notebooky, netbooky, tablety, mobilní telefony. Ale to ani zdaleka není vše. Počítače jsou i tam, kde by je ti méně znalí vůbec nehledali. Jsou v automobilech, jsou v různých domácích spotřebičích, jsou součástí (či někdy spíše podstatou) mnoha lékařských přístrojů. Dnešní dobu můžeme zkrátka označit jako *dobu počítačovou* a nás, moderní lidské bytosti, jako *člověka počítačového*.

O počítačích a dalších zařízeních, které souborně označujeme jako prostředky informačních a komunikačních technologií, však můžeme říci totéž co o ohni. Jsou dobrým sluhou, ale špatným pánem. Jestliže jsou nasazeny správně, uvážlivě a s ohledem na jejich skutečnou roli, dovedou neuvěřitelným způsobem usnadnit život. Jen si zkuste představit psaní této knihy. Dříve bych musel psát knihu (byť na zcela jiné téma, protože počítače neexistovaly) rukou či v lepším případě psacím strojem. Uvažte – žádná možnost uložení a opětovného návratu, žádná možnost snadných oprav, žádná možnost nastavení písma, žádná možnost tisku libovolného množství kopií, žádná možnost zaslání dokumentu někomu jinému prostřednictvím sítě Internet. Sebemenší chyba znamenala nutnost psaní celé stránky od začátku, kopírování bylo, pokud ne přímo nemožné, pak rozhodně velmi obtížné a nepohodlné. A takto bychom mohli s výčtem problémů pokračovat.

Počítač přinesl obrovské změny. I ten nejjednodušší textový procesor, který je obsažen v operačním systému jako jeho integrální součást, nabízí při psaní textu daleko vyšší komfort. Počínaje právě možností ukládat a k uloženému se opět vracet, přes možnost snadných úprav, možnost nastavení písma, stránky a dalších vlastností dokumentu, až po možnost chrlení nekonečného množství kopií. Tady počítač slouží. A slouží dobře. A to nehovořím o pokročilých počítačových aplikacích, jakými jsou například podnikové informační systémy schopné řídit celé velké korporace od A do Z, případně moderní výpočetní systémy schopné řešit složité problémy naší doby.

Ale počítače, respektive jejich špatné či nevhodné nasazení, mohou i škodit. V posledních letech jsme měli možnost zaregistrovat několik systémů, které nejednomu člověku přivodily doslova peklo na zemi. Ať už to byl nefunkční registr motorových vozidel, díky kterému mnozí noví majitelé automobilů či motocyklů doslova kempovali před příslušnými úřady a doufali, že systém během několika hodin opět alespoň na krátkou dobu poběží a umožní jim splnit si jejich zákonnou povinnost přihlásit své auto. Nebo

můžeme zmínit rovněž nefunkční systém mající na starost výplatu sociálních dávek. I zde se nemálo lidí dostalo do velkých potíží, když například nedostali rodičovský příspěvek a nemohli tak zaplatit nezbytné a neodkladné výdaje. Oba systémy nemají náhradu. Pokud nefungují, neexistuje žádný záložní mechanismus, jak lidem umožnit přístup k příslušné službě.

Ale nemusíme jít až tak daleko. Mnohdy počítačová aplikace sama o sobě funguje, a přesto je zdrojem obtíží. To se stává tehdy, je-li aplikace špatně navržena, bez ohledu na reálné charakteristiky zákazníka a konkrétních uživatelů, na jejich potřeby, na jejich schopnosti, na jejich dovednosti, na jejich preference. My „ajťáci" máme poměrně nehezkou vlastnost, za kterou jsme mnohdy okolím odsuzováni – neumíme se vcítit do role běžného uživatele, pro kterého počítač není koníčkem, ale pouhou životní nutností.

Lidé si nekupují počítače proto, že je mají rádi, ale proto, že je potřebují. Nemají touhu se, jak my říkáme, „v počítači hrabat". Touží po jediném – aby mohli snadno a efektivně plnit úkoly, které jsou na ně v pracovním či osobním životě kladeny. Potřebují zaúčtovat fakturu, napsat dopis, přijmout nového pacienta, vyskladnit objednávku, zjistit odjezd toho správného autobusu a mnoho dalšího. A chtějí, aby počítačový systém umožnil vykonávat tyto úkoly snadno a rychle. Jenže reálná situace je často zcela opačná. Počítačové systémy jsou často velmi komplikované a i jednoduché úkoly vyžadují procházení mnoha obrazovkami, mnoho klepání myší. Často jsou postupy zcela nelogické. A důvod je poměrně prostý – vývojář nevytvářel program k obrazu zákazníkovu, ale k obrazu svému. A tak místo toho, aby uživatel v maximální možné míře pracoval tak, jak je zvyklý, a počítač mu pouze asistoval a usnadňoval jeho práci, musí se tento uživatel naopak přizpůsobovat počítači. A to je špatně. Jestliže počítač lidem nepomáhá, pak nemá morální nárok na existenci.

Jenže jak zajistit, aby počítače sloužily a pomáhaly? Každý počítačový systém se skládá z hardwaru a softwaru. Hardwarem rozumíme technické vybavení, to, nač si obvykle můžeme sáhnout. Jsou to všechny ty notebooky, tablety, síťové prvky, monitory, tiskárny, myši, kabeláž. O hardwaru tato kniha určitě nebude. Bude o softwaru, bude o tom, jak vytvářet software tak, aby opravdu odpovídal požadavkům zákazníka. Bude o jedné konkrétní metodice, kterou můžeme použít k tomu, abychom měli nad svým softwarovým projektem kontrolu a abychom byli schopni dovést své projekty ke zdárnému konci, to jest k předání a řádnému používání našeho softwaru zákazníkem. Touto metodikou je metodika SCRUM, které bude věnována převážná část této knihy.

Ale nemůžeme chápat tuto (či některou jinou) metodiku jako něco, co žije v prázdnotě prostoru. Metodika a software samotný jsou součástí širšího kontextu. A právě proto se v této knize zmíním také o obecném pojetí metodik řízení softwaru, o procesu vývoje softwaru, o roli lidí v projektu či například o způsobu měření různých charakteristik projektu. Cílem této knihy pak je, abyste po jejím přečtení byli schopni vnímat softwa-

rový projekt komplexně a abyste byli schopni společně se zákazníkem vytvářet software, který bude opravdu dobře sloužit svému účelu.

Zpětná vazba od čtenářů

Nakladatelství a vydavatelství Computer Press, které pro vás tuto knihu připravilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

Computer Press
Albatros Media a.s., pobočka Brno
IBC
Příkop 4
602 00 Brno

nebo

sefredaktor.pc@albatrosmedia.cz

Computer Press neposkytuje rady ani jakýkoli servis pro aplikace třetích stran. Pokud budete mít dotaz k programu, obraťte se prosím na jeho tvůrce.

Errata

Přestože jsme udělali maximum pro to, abychom zajistili přesnost a správnost obsahu, chybám se úplně vyhnout nelze. Pokud v některé z našich knih najdete chybu, budeme rádi, pokud nám ji oznámíte. Ostatní uživatele tak můžete ušetřit frustrace a pomoci nám zlepšit následující vydání této knihy.

Veškerá existující errata zobrazíte na adrese *http://knihy.cpress.cz/K2210* po klepnutí na odkaz Soubory ke stažení.

KAPITOLA 1

Proces vývoje softwaru

V této kapitole:

- Co je proces vývoje softwaru?
- Metodiky vývoje software
- Agilní vývoj softwaru

Jestli tato kniha něčím nemá být, pak nemá být teoretickým pojednáním. Problémem ovšem je fakt, že zcela bez teorie se prostě a jednoduše neobejdeme. Nemusíte se však bát, nemám v úmyslu zahltit vás formálními teoretickými definicemi, a pokud se tu a tam něco, co definicí zavání, přece jen objeví, pak udělám vše pro to, abych takovou definici názorně ozřejmil a abych ukázal, o co nám vlastně ve vývojářské praxi jde.

Poznámka: Zde se krátce zamyslím nad tím, proč je teorie tak neoblíbená. Pochopitelně, koho by bavilo biflovat se zpaměti nudné definice, které nijak nesouvisí s praxí. Jenže problémem je, že ty nudné definice povětšinou s tou praxí souvisí opravdu hodně. Jestliže pojedete například autem po mostě, zkuste se zamyslet nad tím, zda by konstruktér toho mostu mohl most naprojektovat, kdyby neznal základní poučky o statice, dynamice, kdyby neovládal základní (z jeho pohledu) matematické úkony a další potřebné věci. Jistě, když člověk začíná, teorie je pro něj neprobádaná oblast, které nerozumí a která se mu nezjevuje v souvislostech. Ty přichází až s mnoha nabytými znalostmi a zkušenostmi.

IT se mění, ať chceme či nikoliv. Zatímco dříve mohl být „ajťákem" člověk, který uměl například dobře programovat či zapojovat sítě, dnes to nestačí. Dnešní doba vyžaduje širší rozhled. IT pracovníci již nejsou ti neznámí, kteří někde ve sklepních kójích sedí a starají se jen o provoz sítě, i když i takoví stále existují a mají své místo. Například se ukazuje jako nezbytné mít určité povědomí o ekonomické stránce věci, o právní stránce věci atd. Rovněž existují různé moderní manažerské metody, které například využívají alespoň základy pokročilejší matematiky. Jestliže jako IT pracovníci budeme chtít zůstat v obraze, budeme se muset s tímto vyrovnat.

Co je proces vývoje softwaru?

Kapitola má název „Proces vývoje softwaru". Proto považuji za nezbytné, abychom si tento pojem vysvětlili. A pojďme na to pěkně postupně. Začněme posledním slovem, slovem **software**. Co je myšleno pojmem software? Tohle slovo zná asi každý, kdo se trošku více zabývá počítači, byť jako lehce poučený uživatel. Software můžeme jinak

nazvat programové vybavení počítače. Tedy jsou to všechny programy, které mohou být v počítači nainstalovány. Pro tuto chvíli můžeme opominout rozdělení na systémové a aplikační programy, protože to pro naše pochopení není důležité. Vše, s čím pracujeme na obrazovkách počítačů, notebooků, tabletů, mobilů a dalších zařízení, je software. Ale i zařízení, kde nevidíme žádný displej či obecně komunikační rozhraní, mohou mít software. Softwaru můžeme také chápat jako duši. Duši, která hmotnému tělu vdechuje život. Bez software by hardware, tedy technické vybavení počítače, bylo jen souborem elektronických součástek, který ovšem není schopen být jakkoliv užitečný. Se softwarem je to ovšem jinak. Software zařídí, že ty elektronické součástky začnou provádět to, co určuje vůle uživatele.

Upozornění: Občas slyšíme, že počítač zase neudělal to, co měl. Tady je ovšem problém. Počítač nemá sebemenší inteligenci. Umí udělat jen přesně to, co měl naprogramováno. Nic více a nic méně. Jestliže tedy udělal něco jiného, než co jsme očekávali, pak musí existovat člověk, který počítači tohle uložil udělat. Vinen je tedy člověk, ne počítač. Pochopitelně nebereme v úvahu situaci, kdy počítač má poruchu.

Víme tedy, co je software (a většina to věděla, aniž by musela předchozí odstavec číst, ale přesto jsem stále přesvědčen, že zde předchozí odstavec své místo má), a můžeme plynule přejít ke druhému slovu. Tím je (zachováme pořadí od konce k začátku, které je trošku neobvyklé, ale v tomto případě výstižné) slovo **vývoj**. Ano, software vyvíjíme, nikoliv tvoříme a nikoliv vyrábíme. Tvořivost patří do umění. Tam si ji nejenže můžeme dovolit, tam si ji dovolit musíme, aby to vůbec bylo umění. Jakmile totiž umění postrádá tvořivost, není uměním, je řemeslem (byť třeba velmi dobrým řemeslem). Ale software není o tvořivosti, my totiž nechceme vytvořit něco, co se líbí, něco, co nadchne davy, něco, co probudí emoce, či něco, co bude vyvolávat ty správné asociace dle umělcova záměru. My chceme vytvořit něco, co především funguje a plní požadavky zákazníka. A míra naší improvizace je tím určena.

Na druhou stranu, u softwaru nesmíme sklouznout ani k té řemeslné rutině. Pokud má software sloužit zákazníkovi dobře, či dokonce výborně, musí být zákazníkovi šit na míru. Nesmíme nikdy sklouznout k tomu, že budeme bezmyšlenkovitě aplikovat to, co jsme použili v předchozím projektu (projektech). Pochopitelně, že můžeme využívat zkušenosti i konkrétní části, konstrukce či komponenty. Ale musíme vždy zvážit, zda takové znovupoužití je v souladu se zájmy projektu, a především se zájmy zákazníka. Pokud chceme vyvíjet kvalitní software, pak z našeho slovníku nikdy nesmí vypadnout otázka „proč“, a naopak do tohoto slovníku nesmí vstoupit odpověď „protože se to takhle dělá“. Vývoj tedy stojí někde mezi tvorbou a výrobou. Od výroby se liší tím, že postrádá rutinu v tom, jaké bude konkrétní využití konkrétních nástrojů. Při výrobě je od počátku více či méně jasno, co má vzniknout. Tohle ve vývoji nemáme. Vyvíjíme něco nového, něco, co zde doposud nebylo. Zkuste si představit výrobu automobilu. Z výrobní linky

sjíždí jedno auto za druhým – a tato auta jsou stále stejná, jsou obrazem určité, předem dané šablony. Když začínáme vývoj takového automobilu, pak přesně nevíme, jaké auto bude na konci vývoje. My jej teprve vyvíjíme. Na rozdíl od umělecké tvorby však tento vývoj probíhá systematicky. Víme, že nejprve přijde jedna fáze, pak druhá, pak třetí. Víme, kdy, jak, co a kým budeme testovat, víme, jak budeme získávat jednotlivé lidi do projektu, víme, jak budeme v projektu komunikovat. Tedy vývoj postrádá rutinu ve výsledku, ale obsahuje rutinu v procesu.

A vida, lehce oklikou jsme se dostali k poslednímu slovíčku, které je potřeba definovat. Tím slůvkem je **proces**. Proces je slovo, které v posledních několika letech zní stále častěji mezi manažery všech úrovní. Procesní řízení je totiž v módě, ačkoliv mnozí vůbec nevědí, o čem je řeč, a mnozí další jen opakují naučené fráze. Co je to tedy ten proces (nemáme nyní na mysli soudní kauzu)? Zde si neodpustím definici, ale jak jsem slíbil, definici důkladně rozeberu, abyste měli jistotu, že opravdu pochopíte podstatu problematiky. Tak tedy – proces je po částech uspořádaná posloupnost aktivit vedoucí od daného počátečního stavu k danému cíli. To zní strašně, viďte? Naštěstí lze tuto větu přeložit do jazyka obyčejných smrtelníků.

Tak za prvé – proces není nic exotického. S procesy běžně pracujeme, pouze jim tak v běžné řeči neříkáme. Jestliže má někdo ve zvyku ráno si připravit šálek kávy, pak můžeme hovořit o procesu přípravy kávy. Nebo můžeme hovořit o procesu oblékání, procesu dopravy do zaměstnání atd. Každý takový proces má určitý počáteční stav a určitý cíl. Tak kupříkladu proces vaření kávy. Ten začíná zcela jednoduše tím, že máme chuť na kávu. To je impulsem, který spouští samotný proces. A proces je ukončen tím, že na stole stojí šálek horké kávy. Mezi počátkem a koncem pak musíme vykonat určité kroky – to je ona zmíněná posloupnost aktivit. Zbývá nám ještě vysvětlit, co znamená po částech uspořádaná. Samotná uspořádanost je asi jasná – kroky, tedy aktivity, mají určené pořadí. Nelze tedy například vypít kávu, která nebyla uvařena. Ale proč po částech uspořádaná? Je to proto, že ona posloupnost nemusí být vždy přísně lineární, že mnohdy musíme sice udělat jednotlivé kroky, ale u mnoha z nich není přesně dáno pořadí. Tak například můžeme nejprve nasypat do hrníčku kávu a poté cukr, ale můžeme také zvolit obrácený postup. Na druhou stranu u aktivit naplnění konvice vodou a zapnutí konvice je pořadí jednoznačně dáno.

Vyspělost procesu vývoje

Nyní je pochopitelně na místě ptát se – jak kvalitní je proces vývoje softwaru, který naše firma využívá? Lze vůbec kvalitu nějak měřit? Kdyby nešlo, byla by zde tato kapitola zcela zbytečná. Vzhledem k tomu, že zde tato kapitola je, můžeme si být jisti, že možnost posouzení vyspělosti procesu existuje. Dokonce máme několik možností – některé jsou jednodušší, jiné mají spíše formu auditu. Zájemce o složitější metody mohu pouze

odkázat na příslušnou literaturu. My se zde seznámíme s jednodušším, zato však velmi výstižným modelem hodnocení vyspělosti softwarového procesu.

Model je běžně znám pod zkratkou **CMM**. Pokud si tuto zkratku budeme chtít rozklíčovat, pak dostaneme slovní spojení **Capability Maturity Model**, tedy model vyspělosti dovedností. Model je, jak už bylo naznačeno, poměrně jednoduchý na pochopení – definuje pět úrovní vyspělosti procesu, přičemž každá tato úroveň popisuje, jak proces na této úrovni vypadá a co je nutno doplnit, aby se proces mohl posunout na vyšší úroveň. V následující tabulce pak naleznete přehled těchto pěti úrovní.

Úroveň	Název	Činnost pro postup
1	Initial (iniciální, úvodní)	Opakované postupy
2	Repeatable (opakovatelná)	Komplexní definice procesu
3	Defined (definovaná)	Měření ukazatelů
4	Measured (měřitelná)	Zpětná vazba a zlepšování
5	Optimizing (optimalizovaná)	

Úroveň 1 – Initial

Pojďme se nyní na jednotlivé úrovně podívat podrobněji. Na úrovni 1 (initial, úvodní) se organizace se svým procesem octne tím, že započne svou existenci. Proto hovoříme o iniciální úrovni, protože pro její dosažení není třeba nic vykonat. Na druhou stranu tato úroveň je velmi nízká a proces na této úrovni de facto není schopen generovat kvalitní software, možná s výjimkou těch nejjednodušších aplikací. Proces na této úrovni není de facto definován. Software je vyvíjen intuitivně (a místo vývoje celý proces spíše připomíná onu dříve zmíněnou uměleckou tvorbu), jednotlivé problémy jsou řešeny ad hoc. U větších projektů velmi brzy dojde k velkým rozporům s očekávanými ukazateli (náklady, čas, kvalita), neboť bez definovaného procesu je velmi obtížné či spíše nemožné jakékoliv smysluplné plánování a následné vyhodnocování plánu. Na této úrovni také nefunguje jakékoliv řízení lidských zdrojů a komunikace se zákazníkem je omezena na aktuální problémy a jejich přímé řešení. Komunikace navíc nemá jednotnou podobu, což může způsobovat závažné problémy. Projekt je tak odsouzen k nezdaru okamžitě po svém začátku, tímto způsobem nelze větší projekt ukočírovat ani finančně, ani časově, ale ani nelze zajistit potřebnou kvalitu. Pro organizaci je proto životně důležité urychleně postupovat na žebříčku úrovní.

Představte si, že dostanete za úkol vypočítat určité množství příkladů určitého typu. Na první úrovni pro vás bude každý příklad samostatným úkolem. Mezi jednotlivými příklady neuvidíte žádnou spojitost a každý příklad tak začínáte řešit zcela od začátku. Příklad nebudete řešit systematicky, nýbrž se vždy budete ad hoc pokoušet najít nějaké řešení.

Úroveň 2 – Repeatable

Postup na druhou úroveň (repeatable, opakovatelná) přitom nemusí být záměrný. Jestliže totiž budete na úrovni 1, pak velmi rychle zjistíte určité best practices, tedy postupy, které se osvědčily. Tyto postupy pak přijmete za své a v dalších projektech, případně v dalších fázích téhož projektu je budete používat. Postupně tak část činností nebude řešena ad hoc, ale bude řešena opakovanými postupy, které vychází ze zkušeností dotyčné organizace či projektového týmu. Ačkoliv to vypadá poměrně rozumně (a oproti první úrovni to skutečně pokrok je), není důvod se radovat. Vyskytuje se zde totiž několik podstatných ale, která je nutno brát v potaz. První ale se týká toho, že takto popsané části procesu nejsou ucelené. Není popsán proces jako takový, pouze existují určité zkušenosti s dílčími částmi. To je sice lepší než nic, ale pro praxi je to stále velmi málo. Další problém spočívá v tom, že nelze hovořit o systematicky popsaném procesu, dokonce ani o jeho částech. Platí, že jsme získali zkušenost, že konkrétní úkol lze splnit konkrétním způsobem. Nic více a nic méně. Nikde není zaručeno, že ten konkrétní způsob je opravdu optimální. Ba spíše naopak. Navíc často neexistuje ani písemný popis řešení.

Na této úrovni dojde k tomu, že v množině příkladů rozpoznáte určité podobnosti a určité opakující se podúkoly s rovněž opakujícími se dílčími řešeními. Nebudete tedy každý příklad řešit zcela ad hoc, ale budete se snažit osvědčené postupy aplikovat. Nejste však schopni říci, zda tyto postupy jsou opravdu optimální, zda jsou opravdu obecné atd. Postup také není systematizován. Některé aspekty počítání jsou přitom stále řešeny ad hoc přístupem.

Poznámka: Ve skutečnosti se přechod z první na druhou úroveň povětšinou děje samovolně a bez úmyslného přičinění vývojového týmu či organizace. Jestliže začínáme absolutně bez systému a jestliže vše řešíme ad hoc, pak v určité fázi docházíme vždy ke zjištění, že určité činnosti lze opakovat a že toto opakování povede ke znatelně lepšímu výsledku. A právě při tomto zjištění přecházíme na úroveň 2 – Repeatable. Přechody na další úrovně už ovšem vyžadují systematickou a záměrnou činnost.

Úroveň 3 – Defined

Třetí úroveň (defined, definovaná) už může být považována za počátek něčeho, čemu se říká systematický vývoj softwaru. Na této úrovni je totiž systematicky popsán celý proces vývoje software. Tedy dva obrovské rozdíly oproti předchozí úrovni. Je popsán celý proces vývoje, nikoliv pouze jeho části. Navíc je proces popsán systematicky a metodicky. Už se nejedná pouze o dílčí postupy, které nějak vycházejí, ale o předem stanovený postup, o kterém se přemýšlelo a který je v nějakém smyslu rozumný či optimální. V dnešní době je velmi rozšířená snaha firem získat certifikát o řízení systému jakosti podle normy ISO 9001 – tento certifikát mimo jiné předpokládá právě komplexní popis procesů v dané firmě. Není samozřejmě možné tvrdit, že být na úrovni 3 modelu CMM a získat certifikát podle normy ISO 9001 je totéž. To opravdu není pravda. Nicméně lze

nalézt určité analogie a je evidentní, že podrobný a systematický popis procesu je nutnou (ne však postačující, použijeme-li matematickou terminologii) podmínkou kvality.

Upozornění: Certifikace systému jakosti podle normy ISO 9001 (název normy je ve skutečnosti o něco delší a složitější, ale pro naše potřeby stačí pouhé pochopení) bohužel podlehla určité degradaci a devalvaci. Skutečným cílem této certifikace ve skutečnosti není ona certifikace, ale onen systém řízení jakosti, o kterém norma pojednává. Ve skutečnosti vznikla poptávka po certifikaci (mnoha výběrových řízení se firma bez certifikace ani nemůže zúčastnit). A tak firmy usilují o certifikaci, nikoliv o skutečné reálné zavedení systému řízení jakosti. Ten totiž předpokládá skutečné řízení, ne formální „hození na papír" a divadelní představení při certifikaci. Občas se tento postup povede a výsledkem je, že máme firmu s certifikací, ale bez systému.

Na úrovni 3 máme pro daný typ příkladů přesně stanovený postup výpočtu, který počítá se všemi známými a možnými eventualitami. Postup je ověřen, je korektní, je rozumně popsán tak, aby bylo možné jej rutinně používat. Postup je komplexní a univerzální, tzn. platí pro celou třídu problémů daného typu.

Úroveň 4 – Measured

Třetí úroveň je obecně považována za rozumnou úroveň vyspělosti. Ve skutečnosti je to první úroveň, u které lze konstatovat, že může být dostačující pro vývoj moderního softwaru. Moderní software znamená rozsáhlý informační systém, často propojený s mnoha dalšími systémy softwarovými i hardwarovými, založený na komplikovaných procesech. Moderní software je povětšinou vyvíjen většími týmy a nezřídka týmy z různých společností. A v takovém prostředí nemá vývojář jinou možnost, než postupovat systematicky a metodicky – zkrátka postupovat s využitím inženýrských metod. Třetí úroveň vyspělosti tak není v takovém případě jakousi úrovní středovou, nýbrž úrovní minimální. Je to tak – teprve na třetí úrovni je váš vývojářský tým připraven vyvíjet rozsáhlý moderní software. Nenechte se zmást tím, že se slušný software občas povede i týmům, které postupují vysloveně nemetodicky. Jednak je na zvážení, zda se opravdu jednalo o rozsáhlý sofistikovaný systém, ale především – zda se nejednalo o pouhou náhodu.

Poznámka: Pokud se dítěti podaří nastartovat auto, rozjet se, a dokonce bezpečně zastavit, jistě to nebudeme považovat za doklad způsobilosti řídit motorové vozidlo. A budeme trvat na systematické přípravě v autoškole, při splnění věkových a dalších limitů. Náhoda zkrátka není dobrým společníkem, a pokud existuje jiná cesta, je vhodné se spoléhání na náhodu vyhnout – v tomto případě právě již zmíněným inženýrským přístupem.

Přesto však přístupu, který je popsán na třetí úrovni vyspělosti, cosi chybí. A to něco je měření a vyhodnocení. Je náš postup skutečně efektivní? Nezpůsobuje zbytečné chyby, zbytečné ztráty? Je skutečně jediný možný? A pokud ne, nejsou některé jiné postupy

v něčem lepší? To jsou zcela zásadní otázky, které mohou mít zásadní dopad na úspěch (či bohužel také neúspěch) našich softwarových projektů. Jenže pokud na tyto otázky chceme odpovědět lépe než pouhým hádáním, potřebujeme mít jistá tvrdá data. A právě tato skutečnost je tím, co nás z úrovně třetí posouvá na úroveň čtvrtou. Existence určitých charakteristik projektu, které následně měříme a vyhodnocujeme. Těmito charakteristikami může být v podstatě cokoliv, co je objektivně měřitelné – procento implementovaných scénářů užití, procento úspěšných testů a mnohé další.

Těchto a podobných dalších měřitelných ukazatelů můžeme nalézt v odborné literatuře desítky, ne-li přímo stovky. Nesčetné množství dalších si můžete vymyslet v rámci svého konkrétního projektu. Jediným kritériem by mělo být to, zda daný ukazatel pro vás má nějaký smysl. Neexistuje důvod měřit něco, co neshledáváte sami pro sebe či pro svůj projekt prospěšné.

Poznámka: Zde jen pozor s hodnocením prospěšnosti. S narůstajícími zkušenostmi a znalostmi budete schopni lépe posoudit užitečnost toho či onoho postupu či přístupu – v tomto případě toho či onoho ukazatele. Dbejte na to, abyste k hodnocení užitečnosti nepřistupovali sebestředně či alibisticky – například dle toho, jak jsou jednotlivé ukazatele snadno měřitelné či snadno vyhodnotitelné. Přínosnost by měla být hodnocena zejména ve smyslu přínosu projektu, přínosu týmu, přínosu zákazníkovi. Na to se obecně často zapomíná a přínos se hodnotí individuálně a krátkodobě – na to doplácí například dokumentace, o čemž koneckonců bude pojednáno v jedné z dalších kapitol.

Jestliže tedy jste schopni definovat sadu ukazatelů (část z nich může být zcela obecná, část pak může být závislá na konkrétním projektu a jeho charakteristikách), u těchto ukazatelů pak definovat měřicí stupnice a provádět pravidelná systematická měření a tato následně rozumně vyhodnocovat, pak můžete hovořit o tom, že vyspělost vašeho softwarového procesu se nachází na čtvrté úrovni.

Upozornění: A zde se musíme zastavit. Ve skutečnosti to, že jsme na čtvrté úrovni, neznamená, že měříme charakteristické ukazatele jednoho projektu, abychom zjistili, zda jsme na tom v tomto projektu dobře či špatně. To samozřejmě provádíme také, ale nesvědčí to o vyspělosti softwarového procesu. Nám v tomto případě jde o měření toho, zda samotný proces, který používáme, je správný. Jestliže zjistíme, že se v projektu například odchylujeme od dohodnutého termínu dokončení, je to pro daný projekt sice velmi nepříjemné, ale naprosto nic to neříká o samotném procesu.

Může být chyba v našich pracovnících, kteří nezvládají své úkoly, může být chyba v komunikaci zákazníka, který adekvátně nereaguje na vzniklé problémy, mohly se vyskytnout chyby z vyšší moci. Nás však (z hlediska vyspělosti procesu) zajímá, jak se tyto ukazatele budou chovat ve více projektech či v různých fázích těchto projektů. Tedy jedině s větším množstvím systematicky získaných hodnot ukazatelů můžeme získat rozumné závěry o samotné kvalitě procesu.

Když se vrátíme k našemu případu s řidičem automobilu. Jestliže se testovaná osoba dokáže rozjet s automobilem, pak to samozřejmě nic neříká o tom, zda automobil opravdu umí ovládat. Proto opakovaně testujeme, zda frekventant autoškoly dokáže uvést automobil do stavu s nenulovou rychlostí, proto tuto skutečnost testujeme v různých podmínkách – z kopce, do kopce, na mokré vozovce, na náledí. Teprve tato opakovaná měření nám řeknou, zda samotný proces (v tomto případě postup rozjíždění se) je v pořádku.

Pro naše počítání příkladů bychom na čtvrté úrovni znali nejen postup samotného výpočtu, ale dokázali bychom hodnotit, zda náš postup skutečně vede ke správnému a úplnému výsledku, zda je tento postup efektivní, zda jej nelze nahradit jiným postupem, který by ke kýženému výsledku vedl například s vynaložením menšího úsilí.

Úroveň 5 – Optimizing

Může se zdát, že nyní už máme opravdu vše, co potřebujeme pro správný vývoj aplikací (či jinou činnost – metoda CMM byla sice navržena právě pro SW projekty, ale s menšími či většími modifikacemi je využitelná i v jiných oblastech lidské činnosti). Ale přesto ještě nejsme na konci naší cesty za kvalitním a vyspělým procesem vývoje softwaru. Co nám ještě chybí? Celý proces je podrobně popsán pomocí některé metody popisu či modelování procesů.

V celém procesu máme nastavené ukazatele, které nám dovolí hodnotit, zda daný proces je dobře nastaven, či nikoliv. Kde tedy můžeme ještě najít rezervy? Odpověď je poměrně prostá – k čemu nám je měření, pakliže toto měření nevede ke zlepšování? Dobře, naměřili jsme, že ten a ten ukazatel dosahuje té a té hodnoty. Dokonce jsme dokázali interpretovat tento stav tak, že jsme diagnostikovali určitou neefektivnost našeho procesu. S tím se však musí něco udělat. Jinak měření nemá smysl.

Upozornění: A opět je důležité pochopit, že to, co nás posune na pátou úroveň, není opravení chyby. Ve skutečnosti v podstatě každý, kdo objeví chybu, se pokusí o její nápravu. Nechme nyní stranou to, zda je tento pokus o nápravu krokem správným či špatným směrnem. Ale chybu nenechá jen tak asi nikdo. Jenže pátá úroveň se neliší pokusem opravit chybu. Pátá úroveň spočívá v tom, že v procesu jsou vytvořeny automatické mechanismy, které vyhodnocují situaci a v případě, že se začnou objevovat nesrovnalosti, začnou s nápravou. Náprava chyby není pouhé zoufalé jednání, ale je nedílnou, integrální součástí procesu.

Často se využívají principy negativní či pozitivní zpětné vazby a podobné mechanismy. V případě, že tyto nastavené mechanismy jsou nastaveny (a také používány) správně a korektně, nedochází obvykle ke stavům kritického selhání. Proč? Protože selhávání je obvykle odhaleno již ve chvíli, kdy odchylka reálného stavu od očekávaného je ještě malá. A tato malá odchylka způsobuje žádné či jen velmi malé problémy. A díky korekčním mechanismům je co nejdříve odstraněna.

Pátá úroveň dle modelu CMM je úrovní nejvyšší. Je úrovní, kdy máme popsaný proces, ve kterém dokážeme měřit důležité charakteristiky. A na základě měření máme implementovány mechanismy nápravy odchylek reálného stavu od stavu očekávaného. Tyto mechanismy nápravy přitom nejsou vnímány jako něco mimořádného, něco cizího, něco, co přišlo jen proto, že jsme „v průšvihu". Naopak, je to vnímáno jako něco, co je nedílnou součástí našeho procesu vývoje, něco, co je zcela samozřejmé a co nikoho nepřekvapuje. A co vlastně není řešením „průšvihu", ale de facto prevencí, která těm větším problémům chce zabránit.

Na naší situaci, která nás provází celým vysvětlováním jednotlivých vyspělostních úrovní, si můžeme pátou úroveň demonstrovat tak, že u příkladů, které počítáme, zároveň měříme efektivnost našeho výpočtu. Jakmile zjišťujeme nedostatky, chybné výsledky, dlouhé doby výpočtu, snažíme se okamžitě hledat příčinu a tuto také napravit. A to včas, dokud je náprava poměrně snadná a bez vážnějších následků.

Poznámka: Zkuste si představit studenta, který marně počítá první příklad písemky. A přes veškerou snahu jej nedokáže vypočítat. Uplyne doba určená k vypracování písemky, student odevzdá prázdný papír, a tudíž je logicky hodnocen nedostatečně. Přitom až dodatečně se ukáže, že další příklady byly triviální a snadno řešitelné. Proto se obvykle doporučuje, aby student v případě, že jeden příklad je nad jeho síly, po krátkém úsilí tento příklad přeskočil a věnoval se těm, které řešit dokáže. Teprve po jejich vyřešení se má vrátit k těm příkladům, které v prvním průchodu vyřešit nedokázal, a pokusit se nad nimi bádat. Triviální? Samozřejmé? Jistě, ale tohle je skromná ukázka jakéhosi optimalizujícího mechanismu. Tento je tak běžný a zažitý, že jej obvykle využíváme, aniž bychom nad ním přemýšleli.

A pro ilustraci si uvedeme ještě jeden mechanismus, snad ještě průzračnější pro pochopení. Ležíte ve vaně a napouštíte si vodu. Zkusmo nastavíte mechanismem baterie teplotu, o které odhadem předpokládáte, že bude vyhovující. V okamžiku, kdy voda začne být příliš studená, přidáte vodu teplou, naopak v situaci, kdy začnete pociťovat nepříjemné pálení, zvolíte vodu chladnější. V žádný okamžik nedochází ke krizi, kdy by došlo buď k podchlazení, či naopak k opaření těla. Tedy opět jednoduchý, ale naprosto funkční optimalizující mechanismus, jehož výsledkem je vana vody o teplotě, která je v daný okamžik optimální. Bez výrazných výkyvů, bez ohrožení, bez kritických stavů.

Všimněte si přitom, že skutečně platí to, co bylo výše popsáno – tyto optimalizační mechanismy jsou skutečně nedílnou součástí procesu jako takového. A právě takto vypadá pátá úroveň vyspělosti procesu. Samozřejmě – jedná se o procesy vývoje SW a optimalizační mechanismy jsou daleko složitější. Ale princip je velmi podobný.

Metodiky vývoje softwaru

V předchozí kapitole jsme si řekli, jak má ve zkratce vypadat proces vývoje softwaru, pokud chceme, aby byl úspěšný – to jest aby vedl (či mohl vést v případě řádné práce –

i sebelépe nastavený proces může být neúspěšný v případě, že jednotlivé práce jsou provedeny neodborně, nedbale, lajdácky) k úspěšnému cíli, kterým je kvalitně implementovaný software pro našeho zákazníka.

Otázkou pak zůstává, jak se k takto nastavenému procesu (tedy procesu na úrovni vyspělosti alespoň 3, lépe však 4 či 5 dle vyspělostního modelu CMM) dopracovat. Zkusme se vrátit zpět na kapitolu o vyspělosti softwarového procesu a zkusme odhalit, jaký vlastně proces má být, jak k němu musíme přistupovat. A naleznete zde adjektiva jako „inženýrský", „systematický", „metodický" – tato adjektiva se vztahují k procesu, k přístupu ke tvorbě softwaru. A přesně tato adjektiva jsou principiální odpovědí na otázku, kterou jsme si prve položili – jak se dopracovat ke kvalitně pojatému procesu vývoje softwaru.

Popsali jsme, že na úrovni první provádíme jednotlivé činnosti ad hoc – tedy na základě toho, jak se problémy a úkoly objevují, nacházíme znovu a znovu jejich řešení. Na druhé úrovni na základě zkušeností dokážeme jednotlivé dílčí úkoly opakovat, bohužel se však jedná o izolované dílčí úkoly, navíc bez systematického přístupu. A říkáme – tohle vše je špatně. Potřebujeme tedy sadu procesů, které nám od počátku do konce budou popisovat jednotlivé činnosti, ale také budou říkat, kdo, kdy a jak má jednotlivé činnosti v rámci těchto procesů provádět. A v ideálním případě (úroveň čtvrtá a pátá) nám umožňuje komplexně zhodnotit proces samotný a případně jej vylepšovat. A právě tato sada procesů se nazývá **metodika vývoje SW**.

Nehledejte tedy, prosím, za slovem metodika cokoliv myšlenkově složitého a cizího. Metodika nám říká, jakým způsobem budeme postupovat, jak budou rozděleny role (ve smyslu definice těchto rolí a vymezení obsahu těchto rolí). Metodika nám říká, jaká bude posloupnost jednotlivých činností, jak a kým bude projekt řízen, jak bude probíhat plánování projektu a další.

Poznámka: Zkusme si představit analogii s činností, kterou patrně každý zná a patrně každý alespoň někdy zkusil – s vařením. Můžeme vařit naprosto nemetodicky. Vložíme část surovin do hrnce, začneme vařit, nyní zjistíme, že nám něco dle receptu chybí, začneme to hledat ve spíži, mezitím se vařené suroviny začnou připalovat, opět odběhneme řešit tento problém, mezitím propásneme vhodný okamžik, kdy se měly přidat další suroviny, které jsme prozatím ve spíži nenašli. Zkrátka a dobře – vaření se nepovedlo a místo lahodného pokrmu jsme vytvořili paskvil. Všimněte si ovšem jedné nesmírně důležité skutečnosti – výše uvedený postup naznačuje, že neúspěšný kuchař velmi dobře ví, jak dobré jídlo uvařit, zná recept, zná suroviny, ví, kdy se jednotlivé suroviny mají přidat, ví, jak dlouho vařit v jednotlivých fázích. Problém ovšem je, že nepostupoval metodicky – to je důvod, proč se dostal do potíží a proč neuspěl. A jak by vlastně vypadalo vaření dle metodiky. Nejsme kuchaři, ale budeme myslet. Co takhle si nejprve pozorně pročíst recept, zjistit, co budu potřebovat, a před samotným vařením si vše připravit tak, aby to v pravou chvíli bylo po ruce a já nemusel nic zoufale hledat? To se týká jak surovin, tak také použitého nádobí a náčiní. A teprve poté se pustit do vaření, které mám ovšem předem v hlavě vymyšlené – mnohá jídla se skládají z více částí, které jsou při-

pravovány odděleně – je tedy nutné mít rovněž plán postupné přípravy těchto částí. A co teprve v restauraci, kde se připravuje například několik jídel najednou? Některé suroviny jsou shodné, jiné odlišné, některé jídlo se připravuje déle, jiné kratší dobu atd. Pokud k přípravě nepřistoupíme metodicky, pak nemůžeme uspět. A naši potenciální hosté se vytouženého jídla nedočkají.

Je proto velmi důležité pochopit, že i odborně velmi zdatný člověk může neuspět, a to jen proto, že jeho práce není systematická a metodická. Tohle je totiž velmi častý argument mnoha manažerů: „Podívejte se, já nepotřebuji ty vaše složité metodiky, mám špičkové programátory, špičkové síťaře, špičkové grafiky, špičkové všechny, my to zvládneme." Ne, opravdu nezvládnou. A kategoričnost tohoto mého tvrzení rapidně stoupá s rostoucí obtížností projektu. Ano, jednoduchý projektík skutečně dostatečně zkušený vývojář může dovést ke zdárnému konci. Ale složitější projekt? Opravdu nevěřte podobným siláckým řečem.

Přitom odmítnutí těchto siláckých řečí není zpochybněním často nezpochybnitelných kvalit jednotlivých pracovníků v projektu. Jenže problém opravdu spočívá jinde. K čemu vám bude „špičkový programátor, špičkový síťař, špičkový grafik", když nebudou vědět, co mají vlastně dělat? Když nebudou vědět, jak se má v dané chvíli postupovat? Když nebudou schopni dešifrovat dokumentaci, protože ji každý povede podle svého? Když nebudou vědět, kdo je zodpovědný za další části vývoje, a když se tedy nebudou mít na koho obrátit s dotazem, s prosbou o radu či s potřebou vysvětlit třeba důležitá rozhraní? Problém je zkrátka a dobře v tom, že moderní velké projekty jsou projekty týmové. A proto je nutné tým vést, koordinovat, řídit. Je nutné nastavit také jasná pravidla komunikace se zákazníkem, pravidla dokumentace a mnoho dalších procesů. Jen tak je možné docílit kvalitního výsledku, tedy kvalitního softwaru.

Upozornění: Zde je na místě ještě důrazné varování a upozornění. Vzhledem k tomu, že se pohybujeme v oblasti softwaru, musíme zdůraznit, že na rozdíl od vaření se nejedná o výrobu. Jedná se o vývoj. Jaký je vlastně rozdíl mezi výrobou a vývojem? Značný. U výroby předpokládáme, že známe postup, a přesně víme, co by mělo být výstupem. Například v automobilce by mělo z výrobní linky sjíždět auto, které je stejné jako auta, která z výrobní linky sjela před ním i po něm. Výroba televizorů předpokládá, že jeden vyrobený televizor je jako druhý. Rovněž při vaření očekáváme, že dnešní svíčková na smetaně bude chutnat stejně jako svíčková na smetaně uvařená před týdnem – často hosté jezdí do konkrétní restaurace právě proto, že konkrétní jídlo má konkrétní (a stabilní) nezaměnitelnou chuť – každý svíčkovou či guláš dělá jinak.

Oproti tomu vývoj softwaru funguje poněkud jinak. Každý softwarový produkt vypadá více či méně odlišně oproti jiným produktům, každý projekt má svá specifika. Proto není možné vytvořit naprosto unifikovaný postup, jak máme software vyvíjet. To pochopitelně neznamená, že nemá smysl metodika vývoje. Nejenže je možná, ale, jak bylo již uvedeno, je nutná. Ale taková metodika musí mít určitý stupeň volnosti, nemůže předpokládat zcela

pevnou posloupnost pevně daných kroků. Některé metodiky se také hodí na určitý typ projektů, jak bude ostatně v knize popsáno v další kapitole věnované jednotlivých metodikám. Vývoj tak lze postavit někam na pomezí mezi výrobu a umění. Výroba předpokládá jednoznačný postup a přesně daný výsledek. Když začínáme vyrábět auto, máme předem jasno, jaké auto bude výsledkem. Pozor – neplést si s vývojem auta – tedy procesem, ve kterém návrháři, konstruktéři, designéři a další tvoří samotný koncept auta. Tento proces je plnohodnotným vývojem stejně jako vývoj softwaru. Vývoj tedy předpokládá daný přístup (byť volnější), nicméně nikoliv přesně daný výsledek. V okamžiku, kdy za vámi přijde zákazník s tím, že chce vyvinout například systém pro evidenci docházky zaměstnanců, není nikdo schopen sdělit, jak přesně bude vypadat a fungovat. A na druhém konci je umění. Zde není definován ani přesný postup, ani přesný výsledek.

O umění se pokoušet nechceme, protože přece jen potřebujeme, aby námi vytvořený software plnil určitá měřitelná kritéria, nicméně je faktem, že některé postupy používané v informačních technologiích umění skutečně připomínají. My však zůstaneme na té „vědecké" straně bariéry, na které platí, že můžeme měřit různé objektivní charakteristiky našeho výtvoru i postupu, který jsme při jeho vývoji využili. Co a jak měřit, o tom bude pojednávat jedna z dalších kapitol.

Pro nás je však důležité pochopení slova **metodika** a především pochopení nutnosti metodiku (nějakou) využít při vývoji softwaru. Metodik existuje dnes již značné množství – tato kniha je přitom věnována jedné konkrétní z nich – proto se metodikám obecně budeme věnovat jen velmi obecně a povrchně. Přesto však není vhodné tuto kapitolu zcela vypustit, případně nebylo vhodné ji z knihy zcela vypustit. Jen tehdy, budete-li znát alespoň základní fakta o jednotlivých metodikách, jejich typech a jejich charakteristikách, budete schopni pro svůj projekt zvolit tu správnou. Ano, při čtení této knihy byste u konkrétních projektů měli být schopni dojít i ke zjištění, že popisovaná metodika SCRUM pro některé vaše projekty opravdu nebude vhodná.

Obecně o metodikách vývoje softwaru

V této krátké kapitole se zaměříme na některé aspekty, které byste měli brát v úvahu bez ohledu na to, kterou konkrétní metodiku nakonec vyberete pro svůj projekt. Jestliže byste si měli z této knihy vybrat pro svůj další vývojářský život jen několik vět, pak jednou z nich stoprocentně bude ta následující. **Metodika není náboženství, metodika není dogma.** Nyní možná váháte, jak je vlastně tato věta myšlena. Přesně tak, jak je napsaná. Mnoho lidí, kteří si přečkou některou z knih o některé z metodik, se snaží nazpaměť naučit jednotlivé poučky a poté je, a nyní mnozí prominou, tupě implementovat do projektu. Jestliže se dočtou, že některá fáze vývoje má trvat tři týdny, pak se snaží slepě podřídit vše tomu, aby opravdu trvala tři týdny.

Jenže takhle metodiky vývoje softwaru nefungují. A takto ani fungovat nebudou, protože nemohou. Odpověď na otázku, proč takto fungovat nemohou, najdete v předchozích odstavcích. Abyste nemuseli tyto odstavce znovu detailně číst, odpověď vám poskytnu přímo. Nemohou takto fungovat, protože software se vyvíjí, nikoliv vyrábí. A proto není možné, aby existoval naprosto univerzální a naprosto dokonale přesný postup. Neberte metodiku jako dogma či zákon. Ano, dogma musíte bezvýhradně ctít a zákon bezvýhradně dodržet, ale metodiku chápejte jinak. Chápejte ji jako svého průvodce procesem vývoje softwaru.

Takový průvodce vám říká, co máte dělat, ale nemůže vám říci, jak přesně a kdy přesně to máte udělat. Mimochodem, rozdíl mezi průměrným a špičkovým manažerem nespočívá v tom, že znají a používají diametrálně odlišné metody práce. Spočívá v tom, že špičkový manažer přemýšlí a umí metody použít nejen správně, ale také ve správný čas a na správném místě (a jak uvidíte dále, také se správnými lidmi). To je důvod, proč i v této knize budete mít občas pocit, že vám poskytuje informace vágní, neostré, nepřesné, že časové údaje jsou zásadně udávány jako intervaly od–do, že další hodnoty jsou udávány jen přibližně. Je to proto, že taková je realita.

Poznámka: Jestliže budu hovořit konkrétně o SCRUMu, pak brzy poznáte pojem sprintu. A základní otázka zní: Jak dlouhý má takový sprint být? A jistě budete očekávat odpověď co nejpřesnější. A tato kniha vám dá odpověď – od jednoho do šesti týdnů. A vy budete argumentovat – ale to je strašně vágní, takhle přece nemůžeme fungovat. Jenže takhle metodika skutečně funguje. Říká, že má existovat sprint, říká, co to sprint je, říká, co se má udělat na začátku sprintu, říká, co se má udělat na konci sprintu, říká, co se má dělat v průběhu sprintu. Ale neříká podrobnosti. A to je právě ona volnost metodiky, o které zde je celou dobu řeč. Jste to vy jako manažeři – v dalších kapitolách se dozvíte, jak přesně se jednotlivé role ve SCRUMu (a v jiné knize byste se dozvěděli o rolích v jiné metodice) jmenují a co je obsahem těchto rolí, kdo musí učinit rozhodnutí o délce sprintu v konkrétním projektu. V tom vám žádná kniha nepomůže. Mohu však slíbit, že se v knize pokusím poradit, jak na základě známých faktů nastavit optimální délku sprintu. Či jiné konkrétní rozhodnutí, které budete muset učinit.

Ve skutečnosti – když se v knize dočtete, že sprint má trvat dva až šest týdnů, a vy se z nějakého důvodu rozhodnete pro sprint o délce sedmi týdnů, stále se nic zásadního nemusí stát. Protože, a již se opakuji, metodika není dogma. Rozhodnutí tohoto typu samozřejmě může způsobit potíže. Ale pokud způsobí problémy, tak nikoliv proto, že jste se dopustili „zločinu nedodržení metodiky", ale proto, že jste parametry projektu nastavili špatně. Projekt nemusí fungovat ani tehdy, když nastavíte sprint třeba na délku jednoho měsíce, tzn. čtyř týdnů. Proto, že pro konkrétní projekt je tato doba nevhodná – příliš krátká nebo příliš dlouhá.

Poznámka: A nebo je také dost dobře možné, že délka sprintu (či kterýkoliv jiný nastavitelný parametr metodiky – délka sprintu, u kterého ani zatím přesně nevíme, co to je, je zde použita proto, že délka události je velmi snadno pochopitelný parametr) je nastavena dobře. Ale něco dalšího je špatně. A nyní budu zcela upřímný – nejčastější příčinou selhání projektu je nikoliv selhání metodiky, ale obyčejné selhání lidského faktoru. Tedy stav, kdy někdo v týmu (nebo celý tým) nedělá svou práci tak, jak by ji dělat měl. A v takovém případě vám nepomůže vůbec nic.

Tedy jen metodika chápaná rozumně jako průvodce může být nástrojem, který povede k úspěšnému projektu. U mnoha projektů totiž chybí to, co obvykle nazýváme zdravým selským rozumem. Ano, pro mnohé aspekty řízení projektu je klíčové obyčejné lidské přemýšlení nad tím, co děláme, jak to děláme a jaké to může mít důsledky. Proto se nedivte, že i v této knize se objeví informace, které nebudou kdo ví jak vědecké, ale které jsou praxí ověřené, přestože na první pohled mohou působit zcela triviálně.

Typy metodik

Jak již bylo naznačeno, v dnešní době se můžeme setkat s mnoha různými metodikami vývoje softwaru. Abyste se dokázali v celé té zoologické zahradě různých přístupů k vývoji softwaru zorientovat, uděláme si nyní krátký, ale výstižný exkurz do problematiky jednotlivých metodik. Těmi se budeme zabývat jen velmi povrchně a okrajově, ale přesto si zkusíme ukázat jejich základní vlastnosti. Všechny metodiky, které se využívají, jsou relativně nové (relativně je myšlen fakt, že žádné metodice není více než pár desítek let). A to zkrátka proto, že dříve ani žádný software neexistoval.

První elektronické počítače se objevily přibližně v období druhé světové války, tzn. přibližně před tři čtvrtě stoletím (byť různé návrhy mechanických počítačů existovaly mnohem dříve). To není tak dávno, dnes žije spousta lidí, kteří během svého života zažili celou éru rozvoje počítačů. A to v počátcích existence počítačů o tvorbě softwaru nemohla být ani řeč. Počítače se „programovaly" pomocí fyzického zapojení – asi jako spojování staré telefonní ústředny, což často můžeme vídat ve starých filmech.

Později došlo na programování v binárním kódu a nakonec v assembleru. Počítače jednak nebyly určeny k běžné interaktivní práci, a navíc ani nedisponovaly takovým výkonem, aby bylo možné vytvořit nějaké grafické rozhraní a programovat tak, jak jsme zvyklí. Zkrátka a dobře – do počítače se „nějak" vložil program a data, celé monstrum zabírající obrovskou místnost se spustilo, několik dní až týdnů se čekalo a poté z tiskárny vyšel papír s výsledkem. To s dnešním vývojem aplikací nemá naprosto nic společného.

S rozvojem elektroniky – zejména se to týká objevu tranzistoru a později integrovaného obvodu – se počítače zrychlovaly a zmenšovaly. Najednou měly dostatečný výkon k tomu, aby bylo možné připojit skutečnou obrazovku, aby bylo možné spustit primitivní ope-

rační systém a aby bylo možné programovat v prvních vyšších programovacích jazycích. To už jsou jazyky, o kterých většina pracovníků v oblasti informačních technologií alespoň slyšela – ALGOL, COBOL, FORTRAN, PASCAL, BASIC, C++. Větší výkon počítačů a pokročilejší programování konečně umožňovaly začít s vývojem skutečných rozsáhlejších aplikací. A s rozsáhlejšími aplikacemi přichází i potřeba prvních metodik. Tyto metodiky dnes nazýváme tradiční.

Tradiční metodiky

Tradiční metodiky označujeme jako tradiční, abychom je odlišili od dnes moderních metodik agilních (kam mimochodem spadá také naše metodika SCRUM). Tradiční tedy znamená jednak to, že tyto metodiky vznikly dříve (to však neplatí vždy, existují i moderní, nově vzniklé tradiční metodiky), ale především to, že tyto metodiky uplatňují tradiční přístupy k vývoji softwaru.

Mezi tyto tradiční přístupy patří zejména snaha o maximální možné „sešněrování" procesu. V tradičních metodikách se snažíme o co nejmenší vágnost, snažíme se, abychom co nejlépe mohli určit jednotlivé termíny, jednotlivé požadavky. Role v těchto metodikách bývají poměrně přesně dány – lidé mají svou specializaci a příliš se neangažují v těch fázích a činnostech vývoje softwaru, které jim nepřísluší. Rolí je přitom v tradičních metodikách poměrně mnoho.

Poznámka: Jako příklady rolí v tradičních metodikách (obecně, některé metodiky mají přece jen svá specifika) můžeme uvést například architekta, analytika, programátora, testera, projektového manažera, grafika, kodéra, databázistu a mnoho dalších rolí. Zastupitelnost jednotlivých rolí je velmi nízká až nulová, každá role má své přesně a striktně dané úkoly.

To na jednu stranu představuje výhodu v tom, že každý člen týmu je specializován v konkrétní činnosti, kterou dokonale ovládá, případně se v ní dále vzdělává. Jako nevýhodu pak ovšem lze uvést skutečnost, že pro splnění úkolu je nutná kooperace a komunikace často mnoha lidí, což vývojový proces prodlužuje a činí jej komplikovanějším.

Obecně přitom platí, že tradiční metodiky kladou poměrně značný důraz na to, aby bylo vše precizně dokumentováno. Od komunikace se zákazníkem, přes sběr požadavků, samotný vývoj, až po testování, předávání a údržbu. Většinou platí, že vývojáři začínají s implementací až v okamžiku, kdy mají k dispozici kompletní analýzu a návrh toho, jak by měl výsledný systém vypadat. Pro zákazníka je to často mírně frustrující v tom smyslu, že velmi dlouho od zadání nevidí žádné, ani dílčí výsledky. To rovněž nevyhovuje i mnoha ryze technicky orientovaným vývojářům, neboť jsou nuceni k častému a zdlouhavému jednání se zákazníkem, k vyplňování a čtení rozsáhlé dokumentace. Lze také říci, že implementace jakékoliv změny je při použití tradiční metodiky poměrně

zdlouhavá – často trvá několik týdnů či měsíců, než požadovaná změna projde všemi stupni rozhodování a než je odbyta potřebná byrokracie.

Nyní se vám může zdát, že tradiční metodiky mají pouze nevýhody, a že tudíž mají svá nejlepší léta za sebou. To však není pravda. Tradiční metodiky mají jednu nespornou výhodu, která jim zajistí široké využití nejspíše ještě na mnoho dalších desetiletí, s obměnami pak nejspíše po celou dobu, kdy se bude nějaký software vyvíjet. Touto nespornou výhodou je řád, pořádek, jistota a předvídatelnost. V tradiční metodice je určitá byrokracie a zkostnatělost vyvážena tím, že metodika jasně určuje role, kompetence a do značné míry i časový harmonogram. Funkcionalita je dána před zahájením implementace, změny jsou minimální a podmíněny tím, že projdou poměrně značným sítem byrokracie, schvalování a hodnocení.

Tradiční metodiky se tedy uplatní v projektech většího rozsahu, v projektech, na kterých spolupracuje více týmů, které jsou často geograficky rozděleny, v projektech, ve kterých se integruje více různých technologií a systémů. Tradiční metodiky jsou rovněž vhodné v situaci, kdy je nutné přesně dodržet funkcionalitu, termíny, kdy je jednoznačně dán rozpočet. Pro takové projekty je často i nemožné použít metodiku, která umožňuje určité rozvolnění prací či kompetencí.

Upozornění: V podstatě lze říci, že omezíme-li svoje posuzování na rozumné metodiky osvědčené praxí, nelze o žádné z nich kategoricky prohlásit, že je špatná. Takové posuzování je velmi podobné situaci, kdy bychom se snažili rozhodnout, zda jsou lepší kleště či kladivo. Metodika je nástroj, stejně jako ony kleště či kladivo. A rozhodnutí, zda použít ten či onen nástroj, přece závisí na posouzení toho, co vlastně potřebujeme. Jestliže chceme zatlouci hřebík, použijeme kladivo, jestliže chceme naopak hřebík vytáhnout, nejspíše poslouží lépe kleště. Snažit se kleštěmi zatlouci hřebík a kladivem hřebík vytáhnout svědčí o naší nekompetentnosti, ne o nekvalitě nástroje. A s metodikami je to přesně stejné – každá metodika má své vlastnosti, své charakteristiky, svá specifika. A je na nás, abychom metodiky znali, abychom posoudili projekt a abychom vhodně zvolili rozumnou a vhodnou metodiku.

Není cílem této knihy řešit výběr vhodné metodiky, proto si pouze ukážeme některé významné tradiční metodiky, aby je bylo možno posoudit.

Vodopádová metodika (Waterfall model)

Vodopádová metodika nebo také někdy vodopádový model vývoje softwaru patří mezi nejstarší metodiky vývoje softwaru. Vznikl v sedmdesátých letech a pro většinu dnešních projektů je bohužel nedostatečný. Slovo bohužel uvádím ze dvou důvodů. Tento model je jednak velmi jednoduchý a jednak i přes svou nedostatečnost je stále využíván pro takové projekty, kde by skutečně využíván být neměl.

Vodopádový model je založen na jednoduché úvaze – softwarový proces neboli proces vývoje softwaru se skládá z několika základních fází, které musí následovat jedna po druhé. Pak je tedy možné právě takto proces řídit a spravovat. Jednotlivé fáze tedy v procesu následují po fázích předchozích, proces je jednosměrný, nikdy se nevrací.

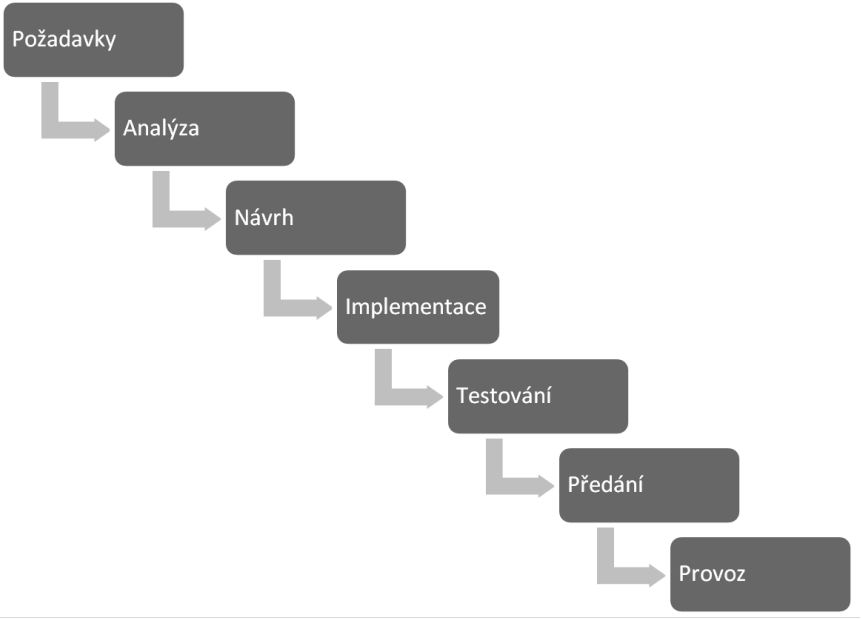


Schéma vodopádového modelu

Metodiku lze tedy charakterizovat několika body:

1. Lineární průběh – každá fáze následuje po předchozí, nikdy se nevracíme zpět
2. Jednoznačnost – vždy víme, v jaké fázi se nacházíme
3. Úplné zadání – do další fáze vstoupíme, když je předchozí fáze se svými výstupy dokončena

Výhody tohoto modelu jsou zjevné. První zásadní výhodou metodiky je její až neuvěřitelná jednoduchost. Po jedné fázi vždy následuje druhá. V každém okamžiku víme přesně, co máme dělat, víme, v jaké se nacházíme fázi. Do žádné fáze nevstupujeme, dokud není plně dokončena fáze předchozí. Pokud nemáme k dispozici specifikaci požadavků, nezačínáme analýzu, pokud nemáme k dispozici analýzu, nezačínáme s návrhem. Metodika je jednoznačná, snadno se řídí, snadno se plánuje.

Bohužel nevýhody, zejména u větších projektů, tyto výhody rapidně převyšují. Jednoduchost je sice velmi příjemná, ale je celkem k ničemu v případě, že nevede k cíli – tedy k úspěšnému softwarovému produktu. Ona zmíněná jednoduchost je totiž vykoupena tím, že v takto řízeném projektu se jen velmi obtížně odhalují a opravují chyby.

Na počátku je vytvořená specifikace. Zde ovšem narážíme na to, že zákazník má občas jen mlhavou představu, nezná možnosti, jaké se nabízí, nezná případná omezení. V této metodice se přitom nepředpokládá další zapojení zákazníka. Ten se tedy ke slovu dostává až ve fázi Předání, kdy už je obvykle dost pozdě řešit případné připomínky. Vůbec se tedy nepočítá se změnami, zákazník uvidí až výsledný produkt.

Jestliže při testování nalezneme chybu, musíme se vrátit zcela na začátek a často je nutné přepracovat v podstatě celý projekt. Často i kvůli banalitě, která je ovšem provázána s mnoha dalšími aspekty projektu. A dříve se v této metodice na chybu přijít vlastně ani nedá. Zákazník pak často velmi nelibě nese, že po celou dobu vývoje nemá žádnou možnost zasahovat do vývoje své aplikace, že dokonce ani nevidí průběžné výsledky, protože žádné průběžné výsledky ani neexistují.

Jsou to právě tyto důvody společně s velmi malou (či spíše žádnou) flexibilitou modelu, kvůli kterým je tento model vývoje (model a metodika jsou slova, která v rámci této knihy budeme považovat za synonymum, byť pochopitelně v obecném pojetí se o synonyma nejedná) dnes použitelný jen pro malé projekty tvořené spíše jedním menším týmem. A to ještě v případě, že se počítá s rychlým dokončením projektu, takže je prakticky eliminována nevýhoda dlouhé prodlevy mezi zadáním specifikace a dodáním produktu.

Jakmile se ve světě IT objevily větší projekty s vyšší provázaností s dalšími technologiemi, projevily se nedostatky vodopádové metodiky naplno. Probíhaly pochopitelně snahy vylepšit tuto metodiku, nicméně ukázalo se, že lepší cestou bude začít na zelené louce a vybudovat metodiku, která bude postavena na zcela jiné než lineární filosofii. V následující tabulce je pak vodopádová metodika stručně shrnuta.

Plusy	Mínusy
✓ Jednoduchost	✗ Velký časový rozptyl mezi zadáním a viditelným dílem
✓ V každém okamžiku přesně víme, v jaké fázi projektu se nacházíme	✗ Pozdní testování a nesnadné odhalování chyb
✓ Rozumné možnosti plánování	✗ Minimální zapojení zákazníka do procesu vývoje
✓ Jednoznačné zadání s minimem změn	✗ Přílišná linearita

Iterativní metodiky

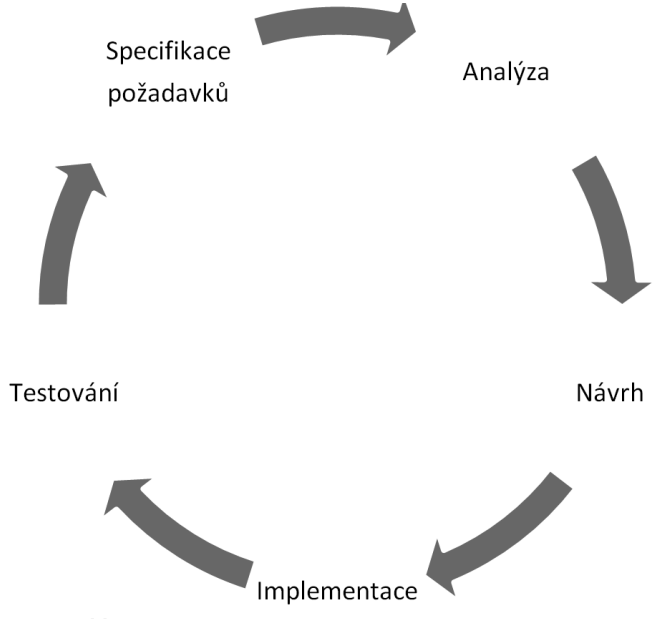
Tato kapitola nebude popisovat jednu konkrétní metodiku, ale několik metodik vyznačujících se tím, že využívají iterativní přístup. Začneme od začátku – podívejte se na tabulku nevýhod vodopádového modelu. Vadí nám velký časový rozptyl mezi zadáním a prvním viditelným výsledkem, vadí nám nemožnost rychlého nalezení problémů, vadí nám pozdní testování, vadí nám, že zákazník není příliš zapojen do procesu vývoje.

Tedy musíme tyto nešvary napravit. Jak? Zkuste si představit, že potřebujete provést určitou činnost, například řemeslnou. Tato činnost je náročná na přesnost. Jak budete postupovat? Stručně to lze vyjádřit následovně:

1. Rozmyslíte si, co vlastně potřebujete.
2. Provedete potřebná měření.
3. Navrhnete postup prací.
4. Provedete práci.
5. Zkontrolujete, zda je dosaženo cíle.
6. Pokud nikoliv, vracíte se k bodu 1.

Jinými slovy – provádíme iterace naší činnosti až do doby, než dosáhneme konečného cíle. Postupně zpřesňujeme svoji práci – první iterace bývá hrubá, další již jemnější, v těch posledních již jen jemně ladíme nejmenší detaily. A nyní vezmeme tento seznam a provedeme drobnou úpravu:

1. Rozmyslíte si, co vlastně potřebujete **(Specifikace požadavků)**.
2. Provedete potřebná měření **(Analýza)**.
3. Navrhnete postup prací **(Návrh)**.
4. Provedete práci **(Implementace)**.
5. Zkontrolujete, zda je dosaženo cíle **(Testování)**.
6. Pokud nikoliv, vracíte se k bodu 1.



Iterační cyklus

Při grafickém zpracování je to vidět více než názorně. A právě tohle je základem iterativních metodik. Těch existuje několik – jako příklady můžeme uvést spirálovou metodiku nebo Rational unified process. Společným jmenovatelem těchto metodik je:

- Spustitelný kód po každé iteraci – zákazník tak na konci každé iterace vidí dílčí výsledek, může si velmi brzy osahat systém, který se tvoří, může vznášet připomínky.
- Testování jako součást každé iterace – chyby jsou objeveny na konci každé iterace a součástí prací v rámci další iterace je oprava těchto chyb.
- Změny jako součást procesu – v každé iteraci dochází k tomu, že zákazník testuje aplikaci nejen ve smyslu hledání chyb, ale také ve smyslu posuzování toho, zda systém skutečně splňuje jeho požadavky – pokud dojde k rozporům, v další iteraci se opět pracuje na zlepšení.

Kromě toho platí, že iterativní metodiky přidávají do své standardní výbavy také další nástroje. Typickým příkladem je využívání nástrojů typu CASE (Computer Aided Software Engineering – počítačem podporované softwarové inženýrství). Tyto nástroje umožňují využívat různé grafické modely, které výrazně usnadňují orientaci ve větších projektech a umožní pochopit velmi dobře jednotlivé vazby mezi jednotlivými strukturami v rámci projektu.

CASE nástroje a grafické modely lze pochopitelně využívat v jakékoliv metodice, dokonce je lze využívat i tehdy, když žádnou metodiku nemáme. Nicméně k tomu, aby tyto nástroje mohly opravdu projevit svou sílu a užitečnost, je také nutno splnit některé další předpoklady. Zkuste si představit, že si uděláte grafické modely aplikace, kterou hodláte vyvíjet vodopádově. V tom případě musíte namodelovat celou aplikaci dříve, než vůbec začnete cokoliv vyvíjet. Problémů bude hned několik:

1. U většího projektu vznikne poměrně rozměrný model, ve kterém nebude autor mít žádné reálné možnosti ověřit jeho správnost – takový model pak nebude mít žádnou relevanci pro projekt.
2. Místo toho, aby modelování a CASE nástroje byly užitečným rozšířením možností, budou pouze dalším zdržením a prodloužením doby mezi zadáním projektu a prvním spustitelným kódem – to zcela jistě bude pro zákazníka jasné negativum, protože už tak u vodopádového modelu musíme počítat s tím, že tato časová prodleva může být u větších projektů značná.

Poznámka: Proč stále zdůrazňujeme, že máme na mysli větší projekty? Protože u projektu malého rozsahu a jednoduché konstrukce často i bez jakékoliv metodiky můžeme nakonec nějak dojít k rozumnému a použitelnému výsledku. Ne proto, že bychom byli tak dobří, ale proto, že obtíží v malém a jednoduchém projektu bylo tak málo, že jsme je dokázali ad hoc vyřešit. Takové projekty ale nejsou středem našeho zájmu. Jednak právě proto, že je lze zvládnout i bez metodiky, ale především proto, že takové projekty většinou nebudeme řešit v reálné praxi. V reálné praxi se vyskytují složitější projekty, které už bez metodického přístupu nelze

zvládnout. Ale proto, aby každému čtenáři bylo jasné, že máme na mysli ty větší projekty, to stále opakujeme. Aby náhodou někdo nenabyl dojmu, že lžeme, protože on přece dělal projekt bez metodiky a ono to vyšlo. Vyšlo a není to nic zvláštního. Ale u velkých komerčních projektů zkrátka nelze spoléhat na to, že to nějak vyjde. Zejména když víme, že to většinou bez rozumného přístupu nevyjde.

3. Kromě toho, že takový postup nám způsobuje reálné a faktické problémy v projektu samotném, dochází také k tomu, že se naprosto nesmyslně zhoršuje také pověst samotných nástrojů. Mnoho vývojářů se špatnou zkušeností vám bude tvrdit, že například „CASE je úplně zbytečné, akorát nás to zdržuje, k ničemu to není, zákazníka to akorát otravuje". Jenže těžko můžete obviňovat nástroj, že je špatný, když základním problémem je špatné použití tohoto nástroje.

Proto se modelovací a jiné CASE nástroje uplatnily zejména v iterativně orientovaných metodikách, protože tam nemusíme hned od začátku budovat obrovské modely, naopak, budujeme modely postupně, ověřujeme jejich části, namodelovaný software implementujeme, následně testujeme. A pokud se objeví problémy, upravíme model, upravíme software, opětovně otestujeme. V takovém případě jsou modely naprosto neocenitelným nástrojem pro vývoj softwaru.

Agilní vývoj softwaru

Popsali jsme si tradiční metodiky. Jak již bylo řečeno, ne vždy jsou tradiční tím, že by byly starší než metodiky jiného typu, ale jsou zcela určitě tradiční tím, že uznávají tradiční přístup k tvorbě softwaru. Existují však projekty, kde tradiční přístupy nejsou to pravé.

Například tradiční metodiky se snaží o vyčerpávající dokumentaci v podstatě jakékoliv aktivity. Změna je strašákem – snaha o změnu je většinou bagatelizována, odmítána, a pokud se ukáže jako nevyhnutelná, je postoupena do zničujícího dlouhodobého martyria schvalování. A takto vlastně fungují všechny procesy – trvají dlouho, vyžadují značnou byrokracii, aktivizují často množství lidí.

Upozornění: Jen dejte pozor a uvědomte si, co zde již padlo. Existují projekty, které se bez té „otravné" byrokracie neobejdou. Právě ta otravná byrokracie zajistí samotné fungování projektu. Je tedy na zvážení – můžeme se bez té slavné byrokracie obejít? Nebo jen odvrhneme to, co ve skutečnosti je lepidlem našeho projektu. Brzy uvidíte, že agilní principy, metodiky jsou skvělé, že umožňují parádní věci. Ale stejně jako cokoliv jiného, o čem v této knize budete číst, jsou jen a pouze nástrojem. Žádná metodika, žádný princip ani žádná pravidla váš projekt nespasí, pakliže nebudete s nástroji umět zacházet.

A prvním důkazem toho, že s nástrojem umíte zacházet, je skutečnost, že dokážete posoudit vhodnost a oblast vhodného použití daného nástroje. Byl bych velmi nerad, kdybyste po pře-

čtení této knihy nabyli přesvědčení, že tradiční metodiky a tradiční přístupy je nutno zahodit do koše a od nynější chvíle vyvíjet pouze agilně. To totiž v žádném případě není pravda. Znalost agilních metodik a agilního přístupu k vývoji softwaru vám pouze rozšíří znalost možných nástrojů. A vy budete mít tu důležitou (a také obtížnou) povinnost rozhodnout, který z vám známých nástrojů bude vhodný pro ten či onen projekt.

Agilní přístup se tedy snaží o eliminaci toho, co je výše popsáno. O eliminaci zbytečné byrokracie, zbytečného dokumentování každé sebeméně významné události či aktivity, o zjednodušení procesů změny. Ale nikoliv o eliminaci živelnou, eliminaci za každou cenu. To je velmi častý omyl mnoha příznivců agilního programování, zejména takových, kteří si přečetli několik statí a mají pocit, že nyní budou vyvíjet v podstatě v prostředí anarchie. Ne, agilní vývoj v žádném případě není návrat k anarchii či ke stavům, které jsou popisovány v kapitole o procesu na první úrovni vyspělostního modelu CMM.

Nic takového, agilně řízené projekty mohou vykazovat (a pro správné a korektní výsledky by měly vykazovat) všechny znaky toho, že vámi využívaný softwarový proces je minimálně na třetí, lépe však na vyšší úrovni. Agilní přístup neznamená nepořádek, znamená stejně pořádek jako přístup tradiční. Jen znamená jiný přístup k pořádku, jiný přístup k rozumně chápaným procesům, jiný přístup k efektivitě.

Tip: Je pochopitelné, že na mnohé z toho, co patří k základním vlastnostem agilního přístupu, si budete nějakou dobu muset zvykat. Pokud jste navíc delší dobu pracovali s tradičním přístupem, pak některé z principů, které budou v této knize popsány, budete jistě považovat za šílené a jediné, co ve vás bude agilní přístup evokovat, bude věta – tohle přece nemůže fungovat. Ale praxe ukazuje, že to opravdu funguje, a rozhodně ne špatně. Je třeba dvou základních kroků. Tím prvním je ovládnutí nástroje, abyste jej opravdu používali ve správnou dobu správným způsobem. A tím druhým je odhození předsudků.

Lidé, kteří zvládli některý z agilních přístupů a kteří jej využívají v projektech, kde je agilní přístup vhodný, jasně potvrzují, že nedošlo ani k anarchii, ani ke ztrátě kontroly nad projektem. Došlo jen k zefektivnění práce. Pochopitelně, agilní přístupy přišly před časem do módy a stalo se přesně to, co je popsáno výše. Agilních metodik se chytli lidé, kteří jim nerozuměli, a začali je aplikovat na projekty, které vůbec pro jejich použití nebyly vhodné. Zde nastalo mnoho problémů a mnoho projektů selhalo. Ale můžeme vůbec mluvit o selhání agilního vývoje? Můžeme vůbec hovořit o selhání nástroje, když nástroj byl používán nevhodně, k nevhodnému účelu, lidmi, kteří o něm vlastně téměř nic nevěděli? Nebylo to spíše selhání lidí? Dle mého názoru jednoznačně. Sebelepší nástroje vždy stojí a padají s lidmi – k tomu se ostatně ještě dostanu podrobněji.

A co tedy vlastně je agilní přístup? Na čem stojí agilní metodiky? Základním principem je – chceme dodat kvalitní software. To je víceméně jediná skutečná hodnota. Vše ostatní je pouhým nástrojem. Dokumentaci nevytváříme proto, abychom měli dokumentaci

samotnou, ale proto, aby se software lépe vyvíjel, udržoval a používal. Změnové řízení nevedeme proto, že máme rádi formuláře, ale proto, abychom udrželi pořádek v projektu a aby výsledkem tohoto projektu byl kvalitní software. A agilní přístup neříká nic jiného než onen kouzelný fakt – chceme dodat kvalitní software. Vše ostatní tomu musíme podřídit. Ale pojďme se na jednotlivé principy podívat podrobněji.

Agilní manifest

Základem celého agilního přístupu je Agilní manifest (Agile Manifesto). A paradoxně se nejedná o žádný oficiální dokument, zákon, předpis. Jedná se o jakési prohlášení, které sepsala skupina mužů věnujících se vývoji softwaru*, kteří byli přesvědčeni, že tuto činnost lze provádět lépe než doposud.

Nejprve se seznámíme s Agilním manifestem tak, jak jej uvedená skupina mužů formulovala. Český překlad naleznete v rámečku – jedná se o text běžně dostupný na internetu. Nejedná se o text složitý, přesto může u těch, kdo ho vidí poprvé, zejména pokud mají zkušenosti s tradičním přístupem k procesu vývoje softwaru, působit podivně a vyvolávat představu, že takto není možné pracovat.

Objevujeme lepší způsoby vývoje softwaru tím,
že jej tvoříme a pomáháme při jeho tvorbě ostatním.
Při této práci jsme dospěli k těmto hodnotám:

Jednotlivci a interakce před procesy a nástroji
Fungující software před vyčerpávající dokumentací
Spolupráce se zákazníkem před vyjednáváním o smlouvě
Reagování na změny před dodržováním plánu

Jakkoliv jsou body napravo hodnotné,
bodů nalevo si ceníme více.

Abychom však pochopili, co je míněno, rozebereme si jednotlivá tvrzení manifestu podrobněji.

1. Jednotlivci a interakce před procesy a nástroji – procesy a nástroje jsou důležité, nicméně pro agilní metodiky jsou důležitější jednotlivci, jejich potřeby, jejich vklad do projektu. Stejně tak interakce. Ve skutečnosti nástroje a procesy pouze slouží k tomu, aby ve výsledku uspokojily potřeby jednotlivců a aby se uskutečnily jednotlivé plánované interakce. To je první významný rozdíl oproti tradičnímu přístupu,

* Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Ken Schwaber, Jeff Sutherland a Dave Thomas

který lpí na procesech často bez ohledu na to, zda lidé v projektu jsou spokojeni (to ovšem není správně). Procesy agilního přístupu jsou celkově plánovány jako jednodušší a minimalistické, ale zároveň plnící svůj primární účel.

2. Fungující software před vyčerpávající dokumentací – fungující software je primárním cílem, vyčerpávající dokumentace je spíše nástrojem, který k danému cíli provedu. A v agilních metodikách preferujeme fungující software. Pozor – to není důvod k tomu, abychom nedokumentovali. Nedokumentovat je prostě „zločin" a žádnou metodikou nemůžete ospravedlnit fakt, že nemáte svou práci zdokumentovanou. Agilní metodiky pouze preferují dokumentaci přirozenou, založenou na tom, že vývojář průběžně dokumentuje svou činnost současně s touto činností. Nikoliv to, aby tento vývojář často seděl hodiny nad dokumentací, která navíc musí mít předepsaný formát, využívat předepsané jazykové prostředky a být psána pomocí standardních šablon a formulářů.

Upozornění: Bohužel, toto tvrzení agilního manifestu mnozí vývojáři a mnohé vývojářské týmy skutečně naprosto šíleným způsobem překrucují, aby se vyhnuli nutnosti dokumentovat. Nedokumentujeme, protože naše metodika dokumentaci zavrhuje. Takže vy již nyní víte, že tohle není pravda. Naopak. Agilní metodiky dokumentaci vyžadují, stejně jako jakékoliv jiné rozumné metodiky. Liší se ve způsobu, ve formalizaci, v tom, že nejsou tak pedantické, ale dokumentovat prostě musíte. Této své povinnosti se nevyhnete žádným způsobem, protože patří k vývoji softwaru stejně, jako hnojení patří k činnosti zemědělce. Různými přístupy k zemědělství můžete ovlivnit míru hnojení, způsob hnojení, frekvenci hnojení. Ale hnojit prostě musíte. A dokumentace je stejně taková. Je nutná. Je stejně nutná jako zdrojové kódy, i když tolik lidí tvrdí opak. Takže to, že jdeme tvrdě a nekompromisně za fungující aplikací, skutečně nemůže znamenat, že nebudete dokumentovat. Fungující aplikace totiž z principu musí být nějak zdokumentovaná.

3. Spolupráce se zákazníkem před vyjednáváním o smlouvě – různá smluvní jednání bývají často velmi komplikovaná, účastní se jich vedoucí pracovníci, účastní se jich právníci, musí se doladit nejmenší detaily. A výsledek? Často je výsledkem jen to, že sice máme precizní smlouvu, nicméně nemáme software. Proto agilní metodiky preferují rozumnou spolupráci před cizelováním dokonalých smluv, ústní komunikace je brána jako komunikace standardní. Opět tím není dotčena nutnost smluvních ujednání, jen nejsou tak protežovány a je preferována normální lidská komunikace, která umožní velmi rychle zahájit práce na samotném projektu.

Poznámka: Je tedy vidět, že agilní metodiky jsou z velké části založeny na tom, co obvykle nazýváme prostě lidské. To, na co se často zapomíná, je skutečnost, že jak vývojářský tým, tak zákazník, mají ve skutečnosti naprosto totožný cíl. A tím je úspěšný projekt zakončený úspěšným softwarovým dílem. Jen v takovém případě dostanou vývojáři zaplaceno to, co