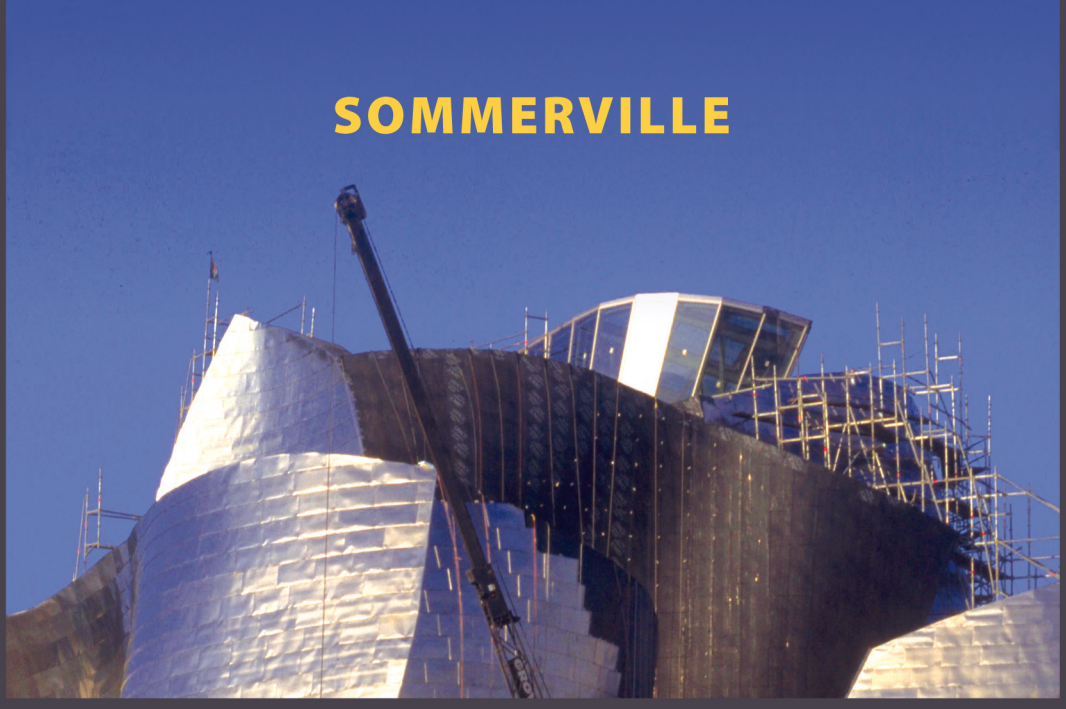
SOMMERVILLE



SOFTWAROVÉ INŽENÝRSTVÍ

computer press

Addison Wesley

Ian Sommerville

Softwarové inženýrství

Computer Press
Brno
2013

Softwarové inženýrství

Ian Sommerville

Překlad: Jakub Goner
Odpovědný redaktor: Libor Pácl
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-3826-7

Vydalo nakladatelství Computer Press v Brně roku 2013 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 17 989.

1. vydání

ALBATROS MEDIA a.s.

Stručný obsah

Obsah

Předmluva

Když jsem v létě roku 2009 psal poslední kapitoly této knihy, uvědomil jsem si, že softwarové inženýrství už má za sebou 40 let. Pojem „softwarové inženýrství" byl navržen roku 1969 na konferenci NATO, kde probíhaly diskuze nad problémy s vývojem softwaru – velké softwarové systémy měly zpoždění, neposkytovaly funkce požadované svými uživateli, stály více, než se očekávalo, a nebyly spolehlivé. Té konference jsem se sice neúčastnil, ale o rok později jsem napsal svůj první program a začal jsem svou profesionální kariéru v oblasti softwaru.

Během mého pracovního života došlo v softwarovém inženýrství ke značnému pokroku. Naše společnost by bez velkých a profesionálních softwarových systémů nemohla fungovat. Při tvorbě obchodních systémů lze nyní využít mnohé technologie s nesrozumitelnými zkratkami – J2EE, .NET, SaaS, SAP, BPEL4WS, SOAP, CBSE atd. – které umožňují vývoj a nasazení velkých podnikových aplikací. Rozvodné podniky a poskytovatelé infrastruktury – energetické, komunikační a dopravní firmy – se vesměs spoléhají na složité a zpravidla spolehlivé počítačové systémy. Software nám umožnil zkoumat vesmír a vytvořit web, který představuje nejvýznamnější informační systém v lidské historii. Lidstvo nyní čelí řadě nových problémů – změně klimatu a extrémním výkyvům počasí, vyčerpávání přírodních zdrojů, rostoucí světové populaci, kterou je třeba nakrmit a ubytovat, mezinárodnímu terorismu nebo potížím se zajištěním uspokojivého a naplněného života pro starší generaci. Abychom dokázali tyto problémy řešit, potřebujeme nové technologie, a ústřední roli v těchto technologiích nepochybně sehraje software.

Softwarové inženýrství je proto kriticky důležitá technologie pro budoucnost lidstva. Musíme pokračovat ve výchově softwarových inženýrů a rozvíjet tuto disciplínu, abychom mohli vytvářet stále složitější softwarové systémy. Softwarové projekty mají samozřejmě i nadále své problémy. I v současnosti dochází ke zpožděním při vývoji softwaru a náklady překračují očekávání. Kvůli těmto potížím bychom však neměli přehlédnout skutečné úspěchy softwarového inženýrství a jeho působivé metody a technologie, které se podařilo vyvinout.

Oblast softwarového inženýrství je nyní natolik rozsáhlá, že není možné celé téma pojmout v jedné knize. Zaměřujeme se proto na klíčová témata, která jsou zásadní pro všechny vývojové procesy, a na témata týkající se vývoje spolehlivých distribuovaných systémů. Stále větší důraz je kladen na agilní metody a opakované použití softwaru. Jsem pevně přesvědčen o tom, že agilní metody mají své místo, ale platí to rovněž i o „tradičním" plánovaném softwarovém inženýrství. Chceme-li tvořit lepší softwarové systémy, musíme zkombinovat ty nejlepší aspekty těchto přístupů.

Knihy nevyhnutelně odrážejí názory a předsudky svých autorů. Někteří čtenáři s mými názory určitě nebudou souhlasit a zvolené příklady jim nebudou vyhovovat. Tyto přirozené názorové rozdíly svědčí

o rozmanitosti disciplíny a jsou klíčové pro její další rozvoj. Přesto doufám, že všichni softwaroví inženýři i studenti tohoto oboru najdou v této knize něco zajímavého.

Integrace s webem

Na webu je k dispozici mimořádné množství informací o softwarovém inženýrství a někteří lidé pochybují o tom, zda jsou stále potřebné učebnice, jako je tato. Kvalita dostupných informací však bývá značně kolísavá, informace jsou někdy prezentovány nevhodným způsobem a občas je těžké najít právě ten údaj, který potřebujeme. Z těchto důvodů se domnívám, že učebnice mají při učení i nadále velký význam. Slouží jako podrobný průvodce tematikou a umožňují informace o metodách a technikách uspořádat a prezentovat souvislým a čitelným způsobem. Poskytují také výchozí bod k hlubšímu zkoumání výzkumné literatury a materiálu na webu.

Rozhodně věřím, že učebnice mají budoucnost, ale pouze za předpokladu, že budou integrovány s webem a budou oproti němu poskytovat přidanou hodnotu. Tuto knihu jsem tedy sestavil jako kombinaci tištěných a webových textů, kde klíčové informace z tištěného vydání jsou propojeny s doplňkovými materiály na webu. Téměř všechny kapitoly obsahují speciálně napsané „webové sekce", které téma dané kapitoly dále rozvíjejí. K dispozici jsou také čtyři „webové kapitoly" v anglickém jazyce o tématech, kterými jsem se v tištěné verzi knihy nezabýval.

Web knihy má adresu:

http://www.SoftwareEngineering-9.com

Kniha se skládá ze čtyř hlavních částí:

1. *Webové sekce* – jedná se o dodatečné sekce, které doplňují obsah prezentovaný v jednotlivých kapitolách. Na tyto webové sekce odkazují rámečky v každé kapitole.
2. *Webové kapitoly* – na webu se nacházejí čtyři webové kapitoly, které se zabývají formálními metodami, návrhem interakcí, dokumentací a aplikačními architekturami. Během životnosti knihy se mohou objevit i další kapitoly o nových tématech.
3. *Materiál pro instruktory* – cílem materiálu v této části je pomoci lidem, kteří učí softwarové inženýrství. Další informace najdete v sekci „Doplňkové materiály" této předmluvy.
4. *Případové studie* – tyto části poskytují další informace o případových studiích použitých v knize (inzulínová pumpa, systém péče o psychiatrické pacienty, automatizovaná meteorologická stanice) a také údaje o jiných případových studiích, například ohledně selhání nosiče Ariane 5.

Kromě těchto částí jsou také k dispozici odkazy na různé weby s užitečnými materiály k softwarovému inženýrství, další zdroje, blogy, zpravodaje atd.

Uvítám vaše konstruktivní komentáře a návrhy týkající se knihy a webu. Můžete mě kontaktovat na e-mailu *ian@SoftwareEngineering-9.com*. V předmětu zprávy prosím uveďte [SE9]. Jinak moje antispamové filtry vaši zprávu pravděpodobně odmítnou a nedostanete žádnou odpověď. Nemám čas pomáhat studentům s jejich domácími úkoly, takže mě o to prosím nežádejte.

Okruh čtenářů

Kniha je určena zejména pro studenty univerzit, kteří si zapisují úvodní a pokročilé kurzy softwarového a systémového inženýrství. Softwarovým inženýrům z praxe může kniha poskytnout obecný přehled oboru a možnost aktualizovat své znalosti témat, jako je opakované použití softwaru, návrh architektury,

spolehlivost a zabezpečení a zlepšování procesů. Předpokládám, že čtenáři mají za sebou úvodní kurz programování a jsou obeznámeni s programátorskou terminologií.

Změny oproti předchozím vydáním

Toto vydání zachovává základní materiál o softwarovém inženýrství, který byl součástí předchozích vydání, ale všechny kapitoly byly revidovány a aktualizovány a kniha obsahuje nový materiál k mnoha různým tématům. Nejdůležitější jsou tyto změny:

1. Došlo k přechodu od pouze tištěné knihy k hybridní tištěné a webové knize, v níž je webový materiál úzce integrován s částmi knihy. Díky tomu jsem mohl omezit počet kapitol v knize a v každé kapitole se soustředit na klíčová témata.
2. Proběhla kompletní restrukturalizace, která usnadňuje použití knihy při výuce softwarového inženýrství. Kniha nyní místo osmi zahrnuje pouze čtyři části a každou z nich lze jako základ kurzu softwarového inženýrství použít samostatně nebo v kombinaci s libovolnými jinými částmi. Jedná se o následující čtyři části: úvod do softwarového inženýrství, spolehlivost a zabezpečení, pokročilé softwarové inženýrství a management softwarového inženýrství.
3. Některá témata z předchozích vydání jsou prezentována stručněji v jediné kapitole a dodatečné texty jsou přesunuty na web.
4. Na webu jsou dostupné další webové kapitoly, které jsou založeny na kapitolách z předchozích vydání, které jsem do tohoto vydání nezahrnul.
5. Obsah všech kapitol jsem aktualizoval a revidoval. Odhaduji, že bylo úplně přepsáno od 30 % do 40 % textu.
6. Přidal jsem nové kapitoly o agilním vývoji softwaru a integrovaných systémech.
7. Kromě těchto nových kapitol kniha obsahuje nový materiál týkající se inženýrství řízeného modely, vývoje open source, vývoje řízeného testování, Reasonova ementálového modelu, architektur spolehlivých systémů, statické analýzy a kontroly modelů, opakovaného použití komerčních krabicových systémů, softwaru jako služby a agilního plánování.
8. V několika kapitolách se používá nová případová studie se systémem záznamů o pacientech, kteří podstupují léčbu psychiatrických nemocí.

Použití knihy při výuce

Knihu jsem sestavil tak, aby ji bylo možné uplatnit ve třech různých typech kurzů softwarového inženýrství:

1. *Obecné úvodní kurzy softwarového inženýrství* – první část knihy byla úmyslně navržena tak, aby mohla sloužit jako základ jednosemestrálního kurzu úvodu do softwarového inženýrství.
2. *Úvodní až středně pokročilé kurzy konkrétních témat softwarového inženýrství* – pomocí kapitol v částech 2–4 lze vytvořit celou řadu pokročilejších kurzů. Svůj kurz inženýrství kritických systémů jsem například postavil na kapitolách v části 2 doplněných kapitolami o kontrole kvality a správě konfigurace.
3. *Pokročilejší kurzy konkrétních témat softwarového inženýrství* – v tomto případě tvoří základ kurzu kapitoly z knihy. Tyto kapitoly poté doplňují další zdroje informací, v nichž jsou témata rozebrána do větších podrobností. Kurz opakovaného použití softwaru je například možné sestavit z kapitol 16, 17, 18 a 19.

Další informace o aplikaci této knihy při výuce včetně porovnání s předchozími vydáními jsou dostupné na webu knihy.

Doplňkové materiály

K dispozici je mnoho různých doplňkových materiálů, které pomáhají při použití knihy při výuce kurzů softwarového inženýrství: Patří k nim:

- prezentace PowerPoint pro všechny kapitoly v knize,
- obrázky ve formátu PowerPoint,
- příručka instruktora, která poskytuje rady o použití knihy v různých kurzech a popisuje vztahy mezi kapitolami v tomto vydání a předchozích vydáních,
- další informace o případových studiích z knihy,
- dodatečné případové studie, které lze použít v kurzech softwarového inženýrství,
- doplňkové prezentace PowerPoint o systémovém inženýrství,
- čtyři webové kapitoly, které se zabývají formálními metodami, návrhem interakcí, aplikačními architekturami a dokumentací.

Veškerý uvedený materiál je pro čtenáře knihy dostupný zdarma na webu knihy nebo na webu podpory nakladatelství Pearson (viz dále). Omezeně pouze pro akreditované instruktory jsou k dispozici doplňkové instruktážní materiály:

- modelové odpovědi na vybraná cvičení na koncích kapitol,
- kvizové otázky a odpovědi pro každou kapitolu.

Všechny doplňkové materiály, včetně těch omezeně dostupných, jsou v anglickém jazyce k dispozici na adrese:

http://www.pearsonhighered.com/sommerville/

Instruktoři, kteří používají knihu k výuce, mohou získat heslo pro přístup k omezeným materiálům, když se zaregistrují na webu nakladatelství Pearson nebo kontaktují místního zástupce tohoto nakladatelství. Případně si heslo mohou vyžádat e-mailem na adrese computing@aw.com. Autor knihy hesla neposkytuje.

Poděkování

K vývoji této knihy v průběhu let přispělo mnoho lidí a chtěl bych poděkovat všem (recenzentům, studentům a čtenářům knihy), kteří komentovali předchozí vydání a poskytli mi konstruktivní návrhy na změny.

Zejména děkuji své rodině (Anne, Ali a Jane) za jejich pomoc a podporu při psaní knihy. Velké díky patří hlavně mé dceři Jane, která v sobě objevila talent ke korekturám a redakční práci. Mimořádně mi pomohla tím, že přečetla celou knihu a odvedla skvělou práci při opravách mnoha překlepů a gramatických chyb.

Ian Sommerville

ČÁST 1

Úvod do softwarového inženýrství

Cílem této části knihy je poskytnout obecný úvod do softwarového inženýrství. Představíme důležité pojmy, jako jsou softwarové procesy a agilní metody, a popíšeme klíčové aktivity vývoje softwaru od počáteční specifikace softwaru po vývoj systémů. Kapitoly v této části byly sestaveny tak, aby je bylo možné použít jako základ jednosemestrálního kurzu softwarového inženýrství.

Kapitola 1 slouží jako obecný úvod, který představuje profesionální softwarové inženýrství a definuje některé pojmy tohoto oboru. Obsahuje také krátkou diskuzi týkající se etických otázek v softwarovém inženýrství. Podle názoru autora je důležité, aby se softwaroví inženýři zamýšleli nad širšími důsledky své práce. Tato kapitola také uvádí tři případové studie, které budeme používat v dalších částech knihy. Konkrétně se jedná o systém na správu záznamů o pacientech, kteří postupují léčení duševních problémů, řídicí systém přenosné inzulinové pumpy a automatizovanou meteorologickou stanici.

Kapitoly 2 a 3 se zabývají procesy softwarového inženýrství a agilním vývojem. V kapitole 2 zavedeme běžně používané obecné modely procesů vývoje softwaru, jako je vodopádový model, a budeme diskutovat základní aktivity, které do těchto procesů patří. Kapitola 3 toto téma dále rozvíjí při diskusi agilních vývojových metod softwarového inženýrství. Jako příklad agilní metody poslouží hlavně extrémní programování, ale stručně se také zmíníme o metodě Scrum.

Zbývající kapitoly této části rozšiřují popisy aktivit softwarových procesů, které jsou zmíněny v kapitole 2. Kapitola 4 se zabývá kriticky důležitým tématem inženýrství požadavků, kdy dochází k definování požadavků na funkce systému. Kapitola 5 představuje systémové modelování pomocí jazyka UML. V této kapitole se zaměříme na použití diagramů případů použití, diagramů tříd, sekvenčních diagramů a stavových diagramů při modelování softwarového systému. V kapitole 6 se dostaneme k návrhu architektury a vysvětlíme význam architektury a práci s architektonickými vzory při návrhu softwaru.

Kapitola 7 je zaměřena na objektově orientovaný návrh a použití návrhových vzorů. Zmíníme se také o důležitých otázkách implementace – opakovaném použití, správě konfigurace a vývoji zaměřeném na hostitele – a analyzujeme vývoj open source. Kapitola 8 se soustřeďuje na softwarové testování od testování jednotek při vývoji systému až po testování kompletních vydání softwaru. Popíšeme také postupy vývoje řízeného testováním. Tento přístup se nejdříve prosadil v agilních metodách, ale lze jej aplikovat i v jiných scénářích. Nakonec v kapitole 9 poskytneme přehled otázek souvisejících s evolucí softwaru. Budeme se zabývat evolučními procesy, údržbou softwaru a správou starších systémů.

1 Úvod

Cíle

Cílem této kapitoly je představit softwarové inženýrství a poskytnout rámec, který umožní porozumět zbývajícím částem knihy. V této kapitole:

- dozvíte se, co to je softwarové inženýrství a proč je důležité,
- porozumíte tomu, že vývoj různých typů softwarových systémů může vyžadovat různé metody softwarového inženýrství,
- pochopíte některé etické a profesionální otázky, které jsou pro softwarové inženýry důležité,
- seznámíte se se třemi systémy různého typu, které budeme používat jako příklady v dalších částech knihy.

Moderní svět nemůže fungovat bez softwaru. Veřejnou infrastrukturu a rozvodné sítě řídí počítačové systémy a většina elektrických produktů zahrnuje počítač a řídicí software. Na počítače se kompletně spoléhá průmyslová výroba, distribuce i finanční systém. Software se intenzivně používá také v zábavě, včetně hudebního průmyslu, vývoji počítačových her a filmu či televizi. Softwarové inženýrství je tedy pro fungování národních i globální společnosti zcela klíčové.

Softwarové systémy jsou abstraktní a nehmotné. Nejsou omezeny vlastnostmi materiálů, neřídí se fyzikálními zákony ani je nesvazují výrobní procesy. Tím se softwarové inženýrství usnadňuje, protože potenciál softwaru nemá žádné přirozené limity. Na druhou stranu právě proto, že softwarové systémy nemusí zohledňovat fyzická omezení, mohou se rychle dostat do stavu, kdy jsou mimořádně složité, těžko srozumitelné a jejich změny velmi drahé.

Existuje mnoho typů softwarových systémů od jednoduchých integrovaných systémů po komplexní informační systémy s celosvětovým dosahem. U softwarového inženýrství nemá smysl hledat univerzální terminologii, metody či postupy, protože různé typy softwaru vyžadují odlišné přístupy. Vývoj podnikových informačních systémů se zásadně liší od vývoje řadiče vědeckého přístroje. Ani jeden z obou systémů se přitom příliš nepodobá počítačové hře náročné na grafiku. Všechny uvedené aplikace potřebují softwarové inženýrství, ale zvolené metody mohou být přitom různé.

Stále se objevuje mnoho zpráv o neúspěšných softwarových projektech a „softwarových selháních". Softwarové inženýrství čelí kritice, že neodpovídá požadavkům vývoje moderního softwaru. Podle názoru autora jsou však mnohé z takzvaných softwarových selhání důsledkem dvou faktorů:

1. *Rostoucí nároky* – spolu s tím, jak nové metody softwarového inženýrství pomáhají budovat větší a složitější systémy, mění se i požadavky. Systémy je nutné vytvářet a dodávat rychleji, zákazníci požadují větší a ještě složitější systémy a tyto systémy musí mít nové funkce, o kterých se dříve předpokládalo, že je nelze implementovat. Existující metody softwarového inženýrství těmto nárokům nestačí a je potřeba vyvinout nové techniky, které budou novým potřebám odpovídat.
2. *Nízká očekávání* – počítačové programy lze psát relativně snadno i bez použití metod a postupů softwarového inženýrství. Mnoho společností se pustilo do vývoje softwaru, když se změnily jejich produkty a služby. Ve své běžné činnosti však nepoužívají metody softwarového inženýrství. Jejich software proto bývá dražší a méně spolehlivý, než by mohl být. Chceme-li tento problém vyřešit, potřebujeme lepší osvětu a školení na poli softwarového inženýrství.

Softwaroví inženýři mohou být na své výsledky právem hrdi. Samozřejmě se při vývoji složitého softwaru i nadále potýkáme s problémy, ale bez softwarového inženýrství bychom nedokázali zkoumat vesmír, neměli bychom Internet ani moderní telekomunikace. Všechny formy cestování by byly nebezpečnější a dražší. Softwarové inženýrství poskytuje značné výhody a podle názoru autora bude v 21. století jeho přínos ještě větší.

HISTORIE SOFTWAROVÉHO INŽENÝRSTVÍ

Pojem „softwarové inženýrství" byl poprvé navržen roku 1968 na konferenci, kde se diskutoval jev, kterému se tehdy říkalo „softwarová krize" (Naur a Randell, 1969). Jasně se ukázalo, že individuální přístupy k vývoji softwaru nelze škálovat na velké a složité softwarové systémy. Tyto nové systémy byly nespolehlivé, stály více, než se původně očekávalo, a jejich vývoj obvykle nabíral zpoždění.

Během 70. a 80. let vzniklo mnoho nových technik a metod softwarového inženýrství, jako například strukturované programování, skrývání informací a objektově orientovaný vývoj. Vyvinuly se nástroje a standardní formy zápisu, které se nyní rozsáhle využívají.

http://www.SoftwareEngineering-9.com/Web/History/

1.1 Profesionální vývoj softwaru

Programy píše mnoho lidí. Zaměstnanci komerčních firem si svou práci zjednodušují tvorbou programů pro tabulkové procesory, vědci a technici píší programy na zpracování svých experimentálních dat a amatéři vytvářejí programy pro svou zábavu i poučení. Většinou se však software vyvíjí jako profesionální aktivita, kdy vzniká software pro konkrétní podnikové použití, pro zahrnutí do jiných zařízení nebo jako softwarové produkty typu informačních systémů, nástrojů CAD atd. Profesionální software, jehož cílovým uživatelem je někdo jiný než jeho vývojář, obvykle nevyvíjí jednotlivci, ale týmy. Tento software je nutné během jeho životního cyklu udržovat a měnit.

Softwarové inženýrství má podporovat profesionální vývoj softwaru spíše než programování jednotlivců. Zahrnuje techniky, které usnadňují specifikaci, návrh a evoluci programů. Tyto techniky přitom zpravidla nejsou pro individuální vývoj softwaru relevantní. Tabulka na obrázku 1.1 shrnuje některé nejčastější dotazy, které se týkají softwarového inženýrství.

Mnoho lidí se domnívá, že pojem software je pouze synonymem pro počítačové programy. V kontextu softwarového inženýrství však software nezahrnuje pouze samotné programy, ale také související

dokumentaci a konfigurační data, která jsou potřebná ke správnému fungování těchto programů. Profesionálně vyvíjený softwarový systém se často neomezuje na jediný program. Systém obvykle obsahuje více samostatných programů a konfiguračních souborů, které slouží ke správnému nastavení těchto programů. Součástí může být systémová dokumentace, která popisuje strukturu systému, uživatelská dokumentace, která popisuje používání systému, a webové stránky pro uživatele, kde je možné najít aktuální informace o produktu.

Jedná se o jeden z hlavních rozdílů mezi profesionálním a amatérským vývojem softwaru. Pokud píšete program sami pro sebe, nebude jej používat nikdo jiný a nemusíte se zabývat psaním programových příruček, dokumentací návrhu programu atd. Jestliže však tvoříte software, se kterým budou pracovat jiní uživatelé a který budou měnit jiní programátoři, obvykle musíte poskytnout další informace i kód programu.

Otázka	Odpověď
Co to je software?	Počítačové programy a související dokumentace. Softwarové produkty je možné vytvářet pro konkrétního zákazníka, nebo mohou být vyvíjeny pro obecný trh.
Jaké atributy má kvalitní software?	Kvalitní software by měl uživateli poskytovat požadované funkce a výkon a měl by být spravovatelný, spolehlivý a snadno použitelný.
Co to je softwarové inženýrství?	Softwarové inženýrství je technická disciplína, která se zabývá všemi aspekty produkce softwaru.
Jaké jsou základní aktivity softwarového inženýrství?	Specifikace softwaru, vývoj softwaru, validace softwaru a evoluce softwaru.
Jaký je rozdíl mezi softwarovým inženýrstvím a informatikou?	Informatika se zaměřuje na teorii a základy. Softwarové inženýrství se zabývá praktickými otázkami vývoje a poskytování užitečného softwaru.
Jaký je rozdíl mezi softwarovým inženýrstvím a systémovým inženýrstvím?	Systémové inženýrství se zabývá všemi aspekty vývoje počítačových systémů, včetně hardwarového, softwarového a procesního inženýrství. Softwarové inženýrství je součástí tohoto obecnějšího procesu.
Jaké jsou klíčové výzvy, před nimiž stojí softwarové inženýrství?	Zvládání stále větší rozmanitosti, požadavků na kratší lhůty dodání a vývoje důvěryhodného softwaru.
Jaké jsou náklady softwarového inženýrství?	Přibližně 60 % nákladů na software představují náklady na vývoj, 40 % jsou náklady na testování. U zakázkového softwaru platí, že evoluční náklady jsou často vyšší než náklady na vývoj.
Jaké jsou nejlepší techniky a metody softwarového inženýrství?	Všechny softwarové projekty je sice nutné řídit a vyvíjet profesionálně, ale pro různé typy systémů se hodí odlišné techniky. Například hry je vždy vhodné vyvíjet pomocí řady prototypů, zatímco u bezpečnostně kritických řídicích systémů je nutné vyvinout úplnou a analyzovatelnou specifikaci. Nelze tedy říci, že jedna metoda je lepší než jiná.
Jaké změny softwarového inženýrství způsobil web?	Web zvýšil dostupnost softwarových služeb a umožnil vyvíjet vysoce distribuované systémy založené na službách. Vývoj webových systémů vedl k důležitým pokrokům v oblasti programovacích jazyků a opakovaného použití softwaru.

Obrázek 1.1 – Nejčastější dotazy týkající se softwaru

Softwaroví inženýři se zabývají vývojem softwarových produktů (tj. softwaru, který lze prodávat zákazníkům). Existují dva typy softwarových produktů:

1. *Obecné produkty* – jedná se o samostatné systémy, jejichž autorem je vývojářská firma, která je na otevřeném trhu prodává všem zákazníkům, kteří si je mohou dovolit. Jako příklady tohoto typu

produktu lze uvést programy pro počítače PC, jako jsou databáze, textové procesory, grafické balíčky a nástroje na řízení projektů. Patří sem také takzvané vertikální aplikace, které jsou určeny pro speciální účely, jako například knihovní informační systémy, účetní systémy nebo systémy na správu zubních záznamů.

2. *Přizpůsobené (neboli zakázkové) produkty* – jedná se o systémy, které si objednal určitý zákazník. Softwarový kontraktor vyvíjí software pro tohoto konkrétního zákazníka. K příkladům tohoto typu softwaru patří řídicí systémy elektronických zařízení, systémy napsané kvůli podpoře určitého obchodního procesu a systémy na řízení letového provozu.

Důležitý rozdíl mezi těmito typy softwaru spočívá v tom, že u obecných produktů řídí specifikaci softwaru organizace, která jej vyvíjí. V případě zakázkových produktů specifikaci obvykle vyvíjí a kontroluje organizace, která software kupuje. Softwaroví vývojáři se musí touto specifikací řídit.

Rozdíl mezi těmito typy systémových produktů se však stále více stírá. Stále více systémů nyní vzniká na základě obecného produktu, který je poté přizpůsoben tak, aby vyhovoval požadavkům konkrétního odběratele. Dokonalým příkladem tohoto přístupu jsou systémy ERP (Enterprise Resource Planning), jako je SAP. V tomto případě je velký a složitý systém adaptován pro určitou společnost tak, že zahrne informace o podnikových pravidlech a procesech, požadovaných sestavách atd.

Při diskusi o kvalitě profesionálního softwaru je potřeba vzít v úvahu, že software používají a mění jiné osoby než jeho vývojáři. Kvalita tedy nesouvisí pouze s tím, jakou má software funkci. Místo toho musí zahrnovat chování softwaru během jeho činnosti a struktur a organizaci systémových programů a přidružené dokumentace. To se odráží v hledisku, které se označuje jako kvalita mimofunkčních softwarových atributů. Jako příklad těchto atributů lze uvést dobu odezvy softwaru na uživatelský dotaz a srozumitelnost programového kódu.

Specifická sada atributů, které lze v softwarovém systému očekávat, samozřejmě závisí na jeho použití. Bankovní systém tedy musí být bezpečný, interaktivní hra musí rychle reagovat, systém telefonní ústředny musí být spolehlivý atd. Tyto požadavky lze zobecnit formou sady atributů uvedených na obrázku 1.2, které podle názoru autora patří ke klíčovým vlastnostem profesionálního softwarového systému.

1.1.1 Softwarové inženýrství

Softwarové inženýrství je technická disciplína, která se zabývá všemi aspekty produkce softwaru od počátečních fází specifikace systému až po údržbu systému, který se již používá. Tato definice obsahuje dvě klíčové fráze:

1. *Technická disciplína* – díky technikům věci fungují. Technici aplikují vhodné teorie, metody a nástroje. Používají je však selektivně a řešení problémů se snaží nacházet i tam, kde nejsou odpovídající teorie a metody k dispozici. Technici si také uvědomují, že musí zohlednit organizační a finanční omezení, takže hledají řešení v rámci těchto omezení.

Vlastnosti produktu	Popis
Schopnost údržby	Software by měl být napsán takovým způsobem, aby se dokázal vyvíjet s ohledem na proměnlivé požadavky zákazníků. Jedná se o kritický atribut, protože změny softwaru představují v proměnlivém podnikovém prostředí nevyhnutelnou nutnost.
Spolehlivost a bezpečnost	Spolehlivost softwaru zahrnuje mnoho parametrů včetně stability a bezpečnosti. Spolehlivý software by v případě selhání systému neměl způsobit fyzickou nebo ekonomickou škodu. Systém by neoprávněným uživatelům neměl umožnit, aby k němu získali přístup nebo jej poškodili.

Vlastnosti produktu	Popis
Efektivita	Software by neměl plýtvat systémovými prostředky, jako je kapacita paměti a cykly procesoru. Efektivita tedy zahrnuje rychlost reakce, dobu zpracování, využití paměti atd.
Přijatelnost	Software musí být přijatelný pro typ uživatelů, pro které je určen. To znamená, že musí být srozumitelný, použitelný a kompatibilní s jinými systémy, které používají.

Obrázek 1.2 – Klíčové atributy kvalitního softwaru

2. *Všechny aspekty produkce softwaru* – softwarové inženýrství se nezabývá pouze technickými procesy softwarového vývoje. Zahrnuje také aktivity typu řízení softwarových projektů a vývoj nástrojů, metod a teorií na podporu produkce softwaru.

Inženýrství usiluje o dosažení výsledků v požadované kvalitě a v rámci schváleného časového plánu a rozpočtu. Často je přitom nutné přijímat kompromisy – inženýři nemohou být perfekcionisty. Lidé, kteří píší programy sami pro sebe, však mohou vývojem strávit tolik času, kolik chtějí.

Obecně řečeno přijímají softwaroví inženýři systematický a organizovaný přístup ke své práci, protože se často jedná o nejvíce efektivní způsob, jak produkovat vysoce kvalitní software. Inženýrství je však založeno na výběru nejvhodnější metody pro danou sadu okolností. V některých případech je proto při vývoji lepší zvolit kreativnější a méně formální přístup. Méně formální vývoj se hodí hlavně při tvorbě webových systémů, které vyžadují kombinaci dovedností programátorů a grafických návrhářů.

Softwarové inženýrství je důležité ze dvou důvodů:

1. Jednotlivci i celé společnosti často spoléhají na pokročilé softwarové systémy. Spolehlivé a důvěryhodné systémy potřebujeme tvořit ekonomicky a rychle.
2. Z dlouhodobého hlediska je často levnější pro softwarové systémy používat metody a techniky softwarového inženýrství a nepsat programy způsobem, jako by se jednalo o osobní programátorský projekt. V případě většiny systémů platí, že většinu nákladů tvoří náklady na změny softwaru poté, co se začal používat.

Systematický přístup, který se uplatňuje v softwarovém inženýrství, se někdy označuje jako softwarový proces. Softwarový proces je sekvence aktivit, které vedou k produkci softwarového produktu. Pro všechny softwarové procesy jsou společné čtyři základní aktivity. Jedná se o tyto aktivity:

1. Specifikace softwaru, kde zákazníci a technici definují vyvíjený software a omezení jeho činnosti.
2. Vývoj softwaru, kdy dochází k návrhu a programování softwaru.
3. Validace softwaru, kdy probíhá kontrola softwaru, zda odpovídá požadavkům zákazníka.
4. Evoluce softwaru, kdy se software upravuje tak, aby odrážel proměnlivé požadavky zákazníků a trhu.

Různé typy systémů vyžadují odlišné vývojové procesy. Například software fungující v letadle v reálném čase musí mít kompletní specifikaci před zahájením vývoje. V systémech elektronického obchodování se specifikace obvykle vyvíjí společně s programem. Z toho vyplývá, že tyto obecné aktivity lze uspořádat různými způsoby a popsat na různých úrovních podrobností v závislosti na typu vyvíjeného softwaru. Softwarové procesy podrobněji popíšeme v kapitole 2.

Softwarové inženýrství souvisí s informatikou i systémovým inženýrstvím:

1. Informatika se zabývá teoriemi a metodami, na nichž je založeno fungování počítačů a softwarových systémů, zatímco softwarové inženýrství řeší praktické problémy při produkci softwaru. Softwaroví inženýři potřebují určité znalosti informatiky obdobně jako se zase elektrotechnici

neobejdou bez základních znalostí fyziky. Teorie informatiky se však často lépe aplikují na relativně malé programy. Elegantní teorie informatiky nelze pokaždé uplatnit na velké a složité problémy, které vyžadují softwarové řešení.

2. Systémové inženýrství analyzuje všechny aspekty vývoje a evoluce komplexních systémů, ve kterých hraje významnou úlohu software. Systémové inženýrství se proto zaměřuje na vývoj hardwaru, návrh zásad a procesů a vývoj systému a také na softwarové inženýrství. Systémoví inženýři se podílejí na specifikaci systému, definují jeho celkovou architekturu a poté integrují různé části do výsledného systému. Poněkud méně se zajímají o konstrukci systémových komponent (hardware, software atd.).

Jak vysvětlíme v další části, software patří do mnoha různých kategorií. Neexistuje žádná univerzální metoda nebo technika softwarového inženýrství, kterou by bylo možné aplikovat na všechny typy softwaru. Přesto však lze zmínit tři obecné otázky, které ovlivňují mnoho různých typů softwaru:

1. *Heterogenita* – od systémů se stále častěji vyžaduje, aby fungovaly distribuovaným způsobem v sítích, které jsou tvořeny různými typy počítačů a mobilních zařízení. Kromě toho, že musí software fungovat v univerzálních počítačích, je také potřeba, aby jej bylo možné spouštět v mobilních telefonech. Často je nutné integrovat nový software se staršími systémy, které jsou napsány v odlišných programovacích jazycích. Problém spočívá ve vývoji technik na tvorbu spolehlivého softwaru, který je dostatečně flexibilní, aby se s touto heterogenitou vyrovnal.
2. *Podnikové a společenské změny* – podniky i celá společnost procházejí mimořádně rychlými změnami, jak se rozvíjejí nové ekonomiky a zvyšuje se dostupnost nových technologií. Uživatelé potřebují měnit svůj stávající software a požadují rychlý vývoj nového softwaru. Mnoho tradičních technik softwarového inženýrství je časově náročných a vývoj nových systémů často trvá déle, než se plánovalo. Tyto metody se musí zlepšit, aby software svým zákazníkům poskytoval rychlejší návratnost jejich investic.
3. *Bezpečnost a důvěra* – vzhledem k tomu, že software zasahuje do všech aspektů našeho života, je zásadně důležité, abychom mu mohli důvěřovat. Platí to hlavně pro vzdálené softwarové systémy, které poskytují přístup přes webové stránky nebo rozhraní webových služeb. Je nutné zajistit, aby útočníci nemohli software úspěšně napadnout a aby zůstala zachována bezpečnost informací.

Tyto aspekty samozřejmě nejsou nezávislé. Může být například potřeba provést rychlé změny staršího systému, aby získal rozhraní pro webové služby. Kvůli těmto problémům potřebujeme nové nástroje a techniky a také inovativní způsoby, jak kombinovat a používat stávající metody softwarového inženýrství.

1.1.2 Rozmanitost softwarového inženýrství

Softwarové inženýrství je založeno na systematickém přístupu k produkci softwaru, který bere v úvahu praktické otázky nákladů, časového plánu a spolehlivosti stejně jako požadavky zákazníků a producentů softwaru. Konkrétní způsob implementace tohoto systematického přístupu se značně liší podle organizace, která software vyvíjí, typu softwaru a osob, které se na procesu vývoje podílejí. Neexistují žádné univerzální metody a techniky softwarového inženýrství, které by se hodily pro všechny systémy a každou společnost. V uplynulých 50 letech se objevila celá škála metod a nástrojů softwarového inženýrství.

Jako pravděpodobně nejdůležitější faktor, který určuje nejdůležitější metody a techniky softwarového inženýrství, lze uvést typ vyvíjené aplikace. Existuje mnoho různých typů aplikací, k nimž patří:

1. *Samostatné aplikace* – jedná se o aplikační systémy, které fungují v místním počítači (například počítači PC). Zahrnují všechny potřebné funkce a nevyžadují připojení k síti. Jako příklady takových aplikací lze uvést kancelářské programy pro PC, programy CAD, software na zpracování fotografií atd.
2. *Interaktivní aplikace založené na transakcích* – jde o aplikace, které se spouštějí ve vzdáleném počítači a uživatelé k nim přistupují z vlastních počítačů nebo terminálů. Do této skupiny samozřejmě patří webové aplikace, jako jsou aplikace pro elektronické obchodování, které umožňují interagovat se vzdáleným systémem při nákupu zboží a služeb. Tato třída aplikací zahrnuje také obchodní systémy, kde podnik poskytuje přístup ke svým systémům prostřednictvím webového prohlížeče nebo speciálního klientského programu, a služby založené na cloudu, jako například webová pošta a sdílení fotografií. Interaktivní aplikace často obsahují velké datové úložiště, ke kterému při každé transakci přistupují a aktualizují jej.
3. *Integrované řídicí systémy* – jsou to softwarové řídicí systémy, které kontrolují a spravují hardwarová zařízení. Co se týče počtu, integrované systémy pravděpodobně překonávají kterýkoli jiný typ systému. Jako příklady integrovaných systémů lze uvést software v mobilních telefonech, software na řízení systémů ABS v automobilech a software v mikrovlnných troubách, který řídí proces ohřívání pokrmů.
4. *Systémy dávkového zpracování* – tyto podnikové systémy jsou určeny ke zpracování dat ve velkých dávkách. Přijímají mnoho jednotlivých vstupů a generují odpovídající výstupy. Jako příklady dávkových systémů můžeme zmínit systémy pravidelné fakturace, jako jsou systémy telefonních operátorů, a systémy na zpracování mezd.
5. *Zábavní systémy* – jedná se o systémy, které jsou určeny zejména pro osobní použití a jejichž účelem je svého uživatele pobavit. Většina z těchto systémů patří do různých herních kategorií. Zábavní systémy se odlišují hlavně zpracováním uživatelské interakce.
6. *Systémy na modelování a simulaci* – jde o systémy, které vyvíjejí vědci a technici, aby mohli modelovat fyzické procesy či situace, které zahrnují mnoho samostatných interagujících objektů. Tyto aplikace bývají výpočetně náročné a k jejich spouštění jsou potřebné vysoce výkonné paralelní systémy.
7. *Systémy shromažďování dat* – jsou to systémy, které sbírají data ze svého prostředí pomocí sady senzorů a odesílají tato data ke zpracování jiným systémům. Příslušný software musí interagovat se senzory a často se instaluje do nepříznivého prostředí, jako je vnitřek motoru, nebo musí fungovat na odlehlém místě.
8. *Systémy systémů* – tyto systémy se skládají z několika jiných softwarových systémů. Některé z nich mohou být obecné softwarové produkty typu tabulkového procesoru. Jiné systémy v rámci celé sady mohou být pro dané prostředí speciálně vytvořeny.

Hranice mezi těmito typy systémů samozřejmě nejsou zcela ostré. Vyvíjíme-li hru pro mobilní telefon, musíme vzít v úvahu stejná omezení (napájení, interakce s hardwarem) jako vývojáři jiného softwaru pro telefony. Systémy dávkového zpracování se často používají v kombinaci s webovými systémy. Jistá společnost může například svým pracovníkům umožnit, aby vyúčtování cestovních náhrad zadávali do webové aplikace, ale příslušná data může odesílat dávkové aplikaci k měsíčnímu zpracování mezd.

Pro každý typ systému se používají jiné techniky softwarového inženýrství, protože software mívá značně odlišné vlastnosti. Například u integrovaného řídicího systému v automobilu je kriticky důležitá jeho bezpečnost a tento systém se při instalaci do vozidla trvale zapisuje do paměti ROM. Jeho změna je proto velmi drahá. Takový systém vyžaduje velmi rozsáhlou verifikaci a validaci, aby se minimalizovala pravděpodobnost, že bude nutné kvůli opravě softwarové chyby stáhnout již prodaná auta.

Uživatelská interakce je minimální (nebo vůbec neexistuje), takže není nutné používat vývojový proces, který vychází z prototypování uživatelského rozhraní.

U webového systému může být vhodný přístup založený na iterativním vývoji a dodávání, přičemž systém je tvořen opakovaně použitelnými komponentami. Takový přístup však nemusí být praktický v případě systému systémů, kde je nutné předem definovat podrobné specifikace systémových interakcí, aby bylo možné všechny dílčí systémy vyvíjet samostatně.

Přesto však můžeme uvést několik základních pouček softwarového inženýrství, které se vztahují na všechny typy softwarových systémů:

1. Systémy je potřeba vyvíjet na základě řízeného a srozumitelného vývojového procesu. Organizace vyvíjející software by měla vývojový proces plánovat a měla by mít jasnou představu o tom, co bude produkovat a kdy to bude dokončeno. Pro různé typy softwaru se samozřejmě používají odlišné procesy.
2. U všech typů systémů je důležitý jejich výkon a spolehlivost. Software by se měl chovat podle očekávání, neměl by selhávat a měl by být svým uživatelům k dispozici tehdy, kdy jej potřebují. Jeho fungování by mělo být bezpečné a v maximální míře by měl být odolný proti externímu útoku. Systém by měl fungovat efektivně a neměl by plýtvat prostředky.
3. Důležité je rozumět specifikaci softwaru a požadavkům na software (co má software dělat) a udržovat tyto dokumenty aktuální. Měli bychom vědět, co od systému očekávají různí zákazníci a uživatelé, a měli bychom jejich očekávání zohledňovat, abychom dokázali dodat užitečný systém s dodržením časových i finančních plánů.
4. Existující prostředky bychom měli využívat co nejefektivněji. Tam, kde je to vhodné, bychom tedy měli opakovaně používat již vyvinutý software a nepsat nový kód.

Tyto základní principy zpracování, spolehlivosti, požadavků, správy a opakovaného použití představují důležité motivy celé knihy. V různých metodách se tyto principy odrážejí různými způsoby, ale tvoří základ veškerého profesionálního vývoje softwaru.

Je potřeba si všimnout, že tyto základy se nevztahují na implementaci a programování. V této knize nerozebíráme konkrétní programátorské metody, protože se mezi jednotlivými typy systémů zásadně liší. K programování webových systémů se například uplatňuje skriptovací jazyk Ruby, který by se však vůbec nehodil pro tvorbu integrovaných systémů.

1.1.3 Softwarové inženýrství a web

Rozvoj webu značně ovlivňuje životy nás všech. Zpočátku se web používal hlavně jako univerzálně dostupné úložiště informací a na softwarové systémy měl jen malý vliv. Tyto systémy fungovaly v místních počítačích a bylo možné k nim přistupovat pouze v rámci příslušné organizace. Kolem roku 2000 se však web začal vyvíjet novým směrem a prohlížeče postupně dostávaly stále nové funkce. To znamenalo, že bylo možné vyvinout webové systémy, které byly namísto speciálního uživatelského rozhraní dostupné prostřednictvím webového prohlížeče. Díky tomu vzniklo mnoho nových systémových produktů, které dokázaly na webu poskytovat inovativní služby. Tyto služby jsou často financovány z reklam, které se zobrazují na obrazovkách uživatelů, a nevyžadují od nich za používání přímé platby.

Kromě těchto systémových produktů se díky pokroku webových prohlížečů, které nově dokázaly spouštět malé programy a zajišťovat omezené místní zpracování dat, začaly vyvíjet nové podnikové a organizační programy. Místo toho, aby se vytvořené programy instalovaly do uživatelských počítačů, nasazoval se tento software na webové servery. Díky tomu bylo možné software mnohem levněji měnit

a aktualizovat, protože již nebylo nutné aktualizované verze instalovat do každého počítače. Uvedený přístup také umožnil snížit náklady, protože vývoj uživatelského rozhraní je obzvláště drahý. Kdykoli to tedy bylo možné, podniky v důsledku tohoto vývoje přecházely na webovou interakci se svými softwarovými systémy.

Další fázi vývoje webových systémů představovala koncepce webových služeb. Webové služby jsou softwarové komponenty, které poskytují konkrétní užitečné funkce a ke kterým lze přistupovat přes web. Aplikace se tvoří integrací těchto webových služeb, které přitom mohou poskytovat různé společnosti. Propojování webových služeb přitom může být dynamické, takže aplikace může při každém svém spuštění pracovat s jinými webovými službami. Tímto přístupem k vývoji softwaru se budeme zabývat v kapitole 19.

V posledních několika letech se rozvíjí koncepce „softwaru jako služby". Její zastánci navrhují, že software by se neměl spouštět v místních počítačích, ale místo toho ve „výpočetních cloudech", které jsou přístupné přes Internet. Používáme-li službu typu webové pošty, pracujeme se systémem založeným na cloudu. Výpočetní cloud sestává z velkého počtu propojených počítačových systémů, které sdílí mnoho uživatelů. Uživatelé si software nekupují, ale platí za něj podle toho, nakolik jej používají, případně mohou získat bezplatný přístup výměnou za to, že se na jejich monitorech objevují reklamy.

Příchod webu tedy způsobil značné změny v organizaci podnikového softwaru. Před nástupem webu byly podnikové aplikace většinou monolitické jednotlivé programy, které fungovaly v jednotlivých počítačích nebo jednotlivých počítačových clusterech. Komunikace probíhala pouze místně v rámci organizace. V současnosti je software vysoce distribuován, někdy po celém světě. Podnikové aplikace se neprogramují od nuly, ale do značné míry spoléhají na opakované použití komponent a programů.

Tato radikální změna organizace softwaru samozřejmě vedla ke změnám ve způsobu, jakým se webové systémy konstruují. Například:

1. Dominantním přístupem při tvorbě webových systémů se stalo opakované použití softwaru. Při budování těchto systémů uvažujeme o tom, nakolik je dokážeme sestavit z již existujících softwarových komponent a systémů.
2. Obecně se nyní uznává, že není praktické specifikovat všechny požadavky na takové systémy předem. Webové systémy je vhodné vyvíjet a poskytovat inkrementálně.
3. Uživatelská rozhraní jsou omezena možnostmi webových prohlížečů. Technologie jako AJAX (Holdener, 2008) sice dovolují vytvářet v rámci webového prohlížeče bohatá uživatelská rozhraní, ale tyto technologie se dosud používají jen obtížně. Častěji se uplatňují webové formuláře s místním skriptováním. Aplikační rozhraní u webových systémů bývají chudší než speciálně navržená uživatelská rozhraní produktů pro operační systémy PC.

Základní myšlenky softwarového inženýrství, které jsme rozebrali v předchozí části, se na webový software vztahují stejně jako na jiné typy softwarových systémů. Pro webový software jsou stále relevantní zkušenosti, které byly získány při vývoji velkých systémů ve 20. století.

1.2 Etika softwarového inženýrství

Podobně jako jiné technické disciplíny také softwarové inženýrství funguje ve společenském a právním rámci, jenž omezuje volnost lidí, kteří v této oblasti pracují. Softwaroví inženýři si musí uvědomit, že odpovědnost jejich profese nesouvisí pouze s aplikací technických dovedností. Pokud chtějí získat profesionální respekt, musí se také chovat etickým a morálně odpovědným způsobem.

Není potřeba zdůrazňovat, že by měli zachovávat normální standardy poctivosti a integrity. Neměli by své dovednosti a schopnosti používat nečestným způsobem nebo takovým způsobem, který by

poškozoval dobré jméno celé jejich profese. V některých oblastech však standardy přijatelného chování nejsou určeny zákony, ale poněkud mlhavějšími principy profesionální odpovědnosti. Patří k nim:

1. *Důvěrnost* – obvykle byste měli respektovat důvěrnost komunikace zaměstnanců či klientů bez ohledu na to, zda byla podepsána formální smlouva o utajovaných skutečnostech.
2. *Kompetentnost* – neměli byste přeceňovat úroveň svých schopností. Neměli byste vědomě přijmout práci, na kterou nestačíte.
3. *Práva k duševnímu vlastnictví* – měli byste znát místní předpisy, kterými se řídí použití duševního vlastnictví, jako jsou například zákony týkající se patentů a autorských práv. Měli byste dbát na bezpečnost duševního vlastnictví svých zaměstnanců i klientů.
4. *Zneužití počítačů* – své technické schopnosti byste neměli používat ke zneužívání cizích počítačů. Zneužití počítačů může mít různou podobu od relativně triviální (například hraní her v pracovním počítači) až po mimořádně závažnou (šíření virů nebo jiného malwaru).

ETICKÝ KÓD A KÓD PROFESIONÁLNÍ PRAXE SOFTWAROVÉHO INŽENÝRA

Jednotná pracovní skupina ACM/IEEE-CS k etickému kódu a kódu profesionální praxe softwarového inženýra

PREAMBULE

Krátká verze kódu shrnuje cíle na vyšší úrovni abstrakce. Klauzule uvedené v úplné verzi poskytují příklady a podrobnosti toho, jak tyto cíle ovlivňují způsob, kterým se chováme jako profesionální softwaroví inženýři. Bez těchto cílů bychom zabíhali do legalistických a nudných podrobností, naopak bez konkrétních podrobností by mohly cíle znít vznešeně, ale prázdně. Společně pak tyto cíle a podrobnosti tvoří soudržný kód.

Softwaroví inženýři by měli usilovat o to, aby činnosti analýzy, specifikace, návrhu, vývoje, testování a údržby softwaru tvořily obsah užitečné a respektované profese. V souladu se svou snahou o veřejné zdraví, bezpečnost a prosperitu by se softwaroví inženýři měli řídit následujícími osmi principy:

1. VEŘEJNOST – softwaroví inženýři by měli jednat podle veřejného zájmu.
2. KLIENT A ZAMĚSTNAVATEL – softwaroví inženýři by měli jednat způsobem, který je v nejlepším zájmu jejich klienta a zaměstnance s ohledem na veřejný zájem.
3. PRODUKT – softwaroví inženýři by měli zajistit, že jejich produkty a příslušné modifikace vyhovují nejvyšším možným profesionálním standardům.
4. ÚSUDEK – softwaroví inženýři by si měli při svém profesionálním rozhodování zachovávat integritu a nezávislost.
5. MANAGEMENT – manažeři a vedoucí v oblasti softwarového inženýrství by měli přijímat a prosazovat etický přístup k řízení vývoje a údržby softwaru.
6. PROFESE – softwaroví inženýři by měli budovat integritu a pověst profese v souladu s veřejným zájmem.
7. KOLEGOVÉ – softwaroví inženýři by měli být spravedliví ke svým kolegům a podporovat je.
8. SOFTWAROVÍ INŽENÝŘI – softwaroví inženýři by se měli účastnit celoživotního vzdělávání ve své profesi a měli by propagovat etický přístup k vykonávání své profese.

Obrázek 1.3 – Etický kód ACM/IEEE (© IEEE/ACM 1999)

Důležitou roli při určování etických standardů hrají profesionální společnosti a instituce. Organizace jako ACM, IEEE (Institute of Electrical and Electronic Engineers) a British Computer Society publikují své kódy profesionálního chování nebo etické kódy. Členové těchto organizací se při svém vstupu zavazují k tomu, že se budou příslušným kódem řídit. Tyto kódy chování se obecně zabývají základním etickým chováním.

Profesionální asociace, konkrétně ACM a IEEE, společně vytvořily etický kód a kód profesionální praxe. Tento kód existuje jak v krátké formě, která je uvedena na obrázku 1.3, tak v delší formě (Gotterbarn et al., 1999), která kratší formu doplňuje o podrobnosti a další aspekty. Důvod, proč tento kód vznikl, shrnují první dva odstavce jeho delší formy:

> *Počítače mají ústřední a stále rostoucí roli v obchodu, průmyslu, státní správě, medicíně, vzdělávání, zábavě a společnosti jako celku. Softwaroví inženýři buď přímo nebo výukou přispívají k analýze, specifikaci, návrhu, vývoji, certifikaci, údržbě a testování softwarových systémů. Vzhledem k jejich roli při vývoji softwarových systémů mají softwaroví inženýři značné možnosti činit dobro nebo působit škody, dávat tyto možnosti jiným nebo přitom jiné ovlivňovat. Pokud chtějí softwaroví inženýři v nejvyšší možné míře zajistit, že bude jejich úsilí využito k dobrým účelům, musí se zavázat k tomu, že budou softwarové inženýrství provozovat jako prospěšnou a respektovanou profesi. V souladu s tímto závazkem by se softwaroví inženýři měli řídit následujícím Etickým kódem a kódem profesionální praxe.*
>
> *Kód obsahuje osm principů týkajících se chování a rozhodování profesionálních softwarových inženýrů včetně praktiků, školitelů, manažerů, supervizorů a tvůrců předpisů stejně jako nováčků a studentů této profese. Zásady identifikují eticky odpovědné vztahy, na nichž se účastní jednotlivci, skupiny a organizace, a primární povinnosti v rámci těchto vztahů. Klauzule jednotlivých zásad ilustrují některé závazky, které z těchto vztahů vyplývají. Tyto závazky jsou založeny na humanismu softwarového inženýra, na speciální péči věnované lidem ovlivněným prací softwarových inženýrů a na jedinečných prvcích praxe softwarového inženýrství. Kód je stanovuje jako závazek všech, kdo se považují za softwarové inženýry nebo na tuto práci aspirují.*

Etickým dilematům budete nejspíš vystaveni v každé situaci, kde mají různí lidé odlišné pohledy a cíle. Jak byste například měli reagovat, pokud z principu nesouhlasíte se zásadami výše postaveného managementu společnosti? Je jasné, že to závisí na konkrétních jednotlivcích a povaze sporu. Je lepší prosazovat svůj postoj v rámci organizace, nebo z principiálního hlediska odejít? Pokud se domníváte, že má softwarový projekt potíže, kdy je sdělíte managementu? Pokud je budete probírat ve fázi, kdy se jedná jen o vaše podezření, může se jednat o přehnanou reakci na celou situaci. Jestliže si své pochybnosti ponecháte do doby, než bude příliš pozdě, řešení problémů již nemusí být možné.

S takovými etickými dilematy se ve svém profesionálním životě setkáváme všichni. Ve většině případů jsou naštěstí relativně nevýznamná nebo je můžeme relativně snadno vyřešit. V případě, že je vyřešit nelze, čelí inženýr možná dalšímu problému. Principiálním krokem může být rezignace na svou pozici, ale tento krok může ovlivnit jiné osoby, jako je životní partner nebo děti.

Mimořádně obtížná situace pro profesionální inženýry nastává tehdy, když se jejich zaměstnavatel chová neetickým způsobem. Řekněme, že společnost odpovídá za vývoj bezpečnostně kritického systému a kvůli časovému tlaku falšuje záznamy o bezpečnostní validaci. Měl by inženýr zachovat důvěrnost informací, nebo by měl upozornit zákazníka či nějakým způsobem zveřejnit, že dodávaný systém nemusí být bezpečný?

Problém spočívá v tom, že z hlediska bezpečnosti neexistují žádné absolutní standardy. Systém sice nemusí být validovaný podle předem určených kritérií, ale tato kritéria mohou být příliš přísná. Systém

může ve skutečnosti v rámci svého životního cyklu fungovat spolehlivě. Naopak platí, že systém může selhat a způsobit nehodu i v případě, že byl úspěšně validován. Předčasné odhalení problémů může způsobit škodu zaměstnavateli a jiným zaměstnancům, ale na druhou stranu utajení problémů může způsobit škody jiným osobám.

Svůj postoj k těmto záležitostem si každý musí ujasnit sám. Vhodná etická pozice zde závisí výhradně na názorech člověka, který situaci řeší. V tomto případě by mělo rozhodnutí záviset na potenciálu škody, jejím rozsahu a okruhu osob, na které by měla vliv. Jestliže je případ velmi nebezpečný, může být oprávněné jej zveřejnit například v celostátním tisku. Měli byste se však vždy pokusit situaci vyřešit s ohledem na práva svého zaměstnavatele.

Další etické dilema představuje účast na vývoji vojenských a jaderných systémů. Někteří lidé k těmto záležitostem zaujímají jednoznačný postoj a nechtějí se podílet na žádném vývoji, který souvisí s vojenskými systémy. Jiní jsou ochotni pracovat na vojenských systémech s výjimkou zbraňových systémů. Další skupina se zase domnívá, že princip národní bezpečnosti zde má přednost, a nemají ohledně vývoje zbraňových systémů žádné zábrany.

V této situaci je důležité, aby si zaměstnanci i zaměstnavatelé své postoje vzájemně ujasnili předem. Když se organizace účastní na projektech vojenského nebo jaderného typu, měla by svým zaměstnancům dát najevo, že se od nich očekává, že přijmou libovolné pracovní zadání. Podobně platí, že pokud dá zaměstnanec v jednom případě jasně najevo, že si na takových systémech pracovat nepřeje, zaměstnavatel by na něj neměl později vyvíjet tlak, aby toto rozhodnutí změnil.

Obecná oblast etiky a profesionální odpovědnosti je stále důležitější, jak systémy intenzivně využívající software prostupují všechny aspekty pracovního a soukromého života. K celé problematice se můžeme postavit z filozofického hlediska, kdy zvažujeme základní principy etiky a následně diskutujeme etiku softwarového inženýrství na základě těchto základních principů. Tento přístup volí Laudon (1995) a v menší míře Huff a Martin (1995). Také Johnsonův text o počítačové etice (2001) pojímá toto téma z filozofické perspektivy.

Podle názoru autora je však filozofický přístup příliš abstraktní a lze jej jen obtížně uvést do souvislosti s každodenními úkoly. Výhodnější je podle něj konkrétnější přístup, který vychází ze zásad chování a praxe. Je tedy nejlepší etiku rozebírat v kontextu softwarového inženýrství a nikoli jako samostatný předmět. V této knize se tedy nebudeme pouštět do abstraktních etických diskusí, ale tam, kde je to vhodné, zahrneme do cvičení příklady, které mohou posloužit jako základ pro skupinovou diskusi o etických záležitostech.

1.3 Případové studie

K ilustraci principů softwarového inženýrství nám v rámci celé knihy poslouží příklady tří různých typů systémů. Jedinou případovou studii nepoužíváme proto, že jedním z klíčových motivů této knihy je závislost postupů softwarového inženýrství na typu produkovaného systému. Při diskusi o koncepcích typu bezpečnosti a spolehlivosti, systémového modelování, opakovaného použití apod. budeme proto volit odpovídající příklady.

Jako případové studie poslouží následující tři typy systémů:

1. *Integrovaný systém* – jedná se o systém, kde software umožňuje řídit hardwarové zařízení, ve kterém je uložen. U integrovaných systémů je obvykle nutné zohlednit fyzickou velikost, dobu odezvy, řízení spotřeby atd. Jako příklad integrovaného systému použijeme softwarový systém na řízení zdravotnického zařízení.

2. *Informační systém* – primárním účelem tohoto systému je spravovat databázi informací a poskytovat k ní přístup. Informační systémy se zabývají otázkami bezpečnosti, použitelnosti, soukromí a zachování integrity dat. Jako příklad informačního systému se uplatní systém správy zdravotních záznamů.
3. *Systém shromažďování dat pomocí senzorů* – tento systém má primárně shromažďovat data ze sady senzorů a nějakým způsobem tato data zpracovávat. Klíčovým požadavkem na takové systémy je spolehlivost – i v nepříznivých podmínkách prostředí – a možnost správy. V této knize je jako příklad systému shromažďování dat zvolena automatická meteorologická stanice.

Všechny uvedené systémy představíme v této kapitole a podrobnější informace o každém z nich jsou dostupné na webu.

1.3.1 Řídicí systém inzulínové pumpy

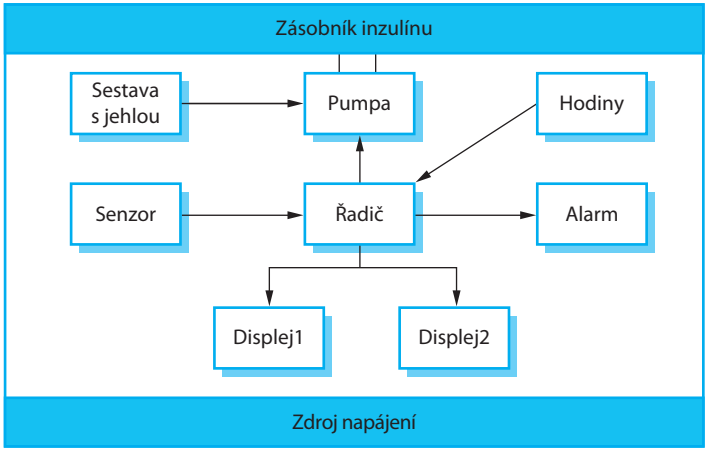
Inzulínová pumpa je zdravotnický systém, který simuluje fungování slinivky břišní. Software na řízení tohoto systému představuje integrovaný systém, který dostává informace ze senzoru a ovládá pumpu, která uživateli poskytuje řízenou dávku inzulínu.

Uživateli tohoto systému jsou pacienti s cukrovkou. Cukrovka (diabetes) je poměrně běžný syndrom, při kterém lidská slinivka nedokáže produkovat dostatečné množství hormonu zvaného inzulín. Inzulín umožňuje metabolizovat cukr glukózu v krvi. Konvenční léčba cukrovky spočívá v pravidelných injekcích inzulínu, který se získává metodami genetického inženýrství. Diabetici měří svou hladinu cukru v krvi pomocí externího přístroje a poté vypočítávají dávku inzulínu, kterou si mají píchnout.

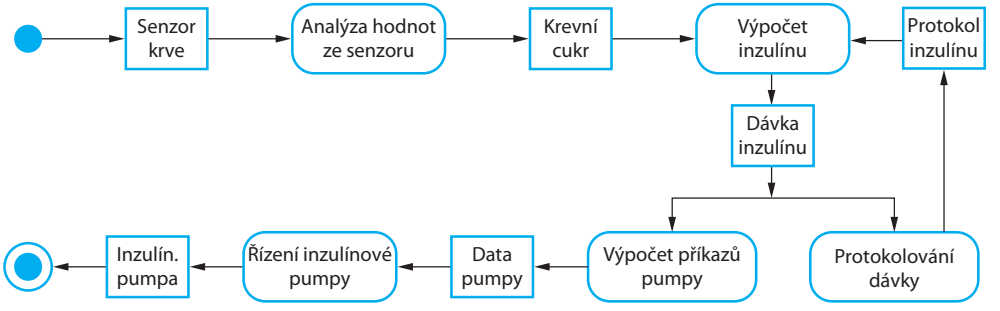
Problém této léčby spočívá v tom, že potřebné množství inzulínu nezávisí pouze na aktuální hladině glukózy v krvi, ale také na tom, kdy pacient dostal poslední injekci inzulínu. Hladina glukózy v krvi proto může značně poklesnout (pokud je inzulínu příliš mnoho), nebo se může výrazně zvýšit (jestliže je inzulínu příliš málo). Nízká hladina glukózy je z krátkodobého hlediska nebezpečnější, protože může vést k dočasné poruše fungování mozku a následně i bezvědomí a smrti. Z dlouhodobé perspektivy však trvale vysoká hladina glukózy v krvi může způsobovat poškození oční sítnice a ledvin a problémy se srdcem.

Díky současným pokrokům při vývoji miniaturizovaných senzorů lze nyní vyvinout automatizované systémy na dávkování inzulínu. Tyto systémy monitorují hladinu cukru v krvi a poskytují odpovídající dávku inzulínu podle potřeby. Systémy na dávkování inzulínu podobného typu se již používají při léčení pacientů v nemocnicích. V budoucnu bude možné, aby mělo mnoho diabetiků takové systémy trvale připojeno ke svému tělu.

Systém dávkování inzulínu řízený softwarem funguje na základě voperovaného mikrosenzoru. Tento senzor měří určitý parametr krve, který je přímo úměrný hladině cukru. Příslušná hodnota se poté odesílá do řadiče pumpy. Řadič vypočítá hladinu cukru v krvi a potřebné množství inzulínu. Následně odešle signály miniaturizované pumpě, která poskytuje inzulin přes trvale připojenou jehlu.



Obrázek 1.4 – Hardware inzulínové pumpy



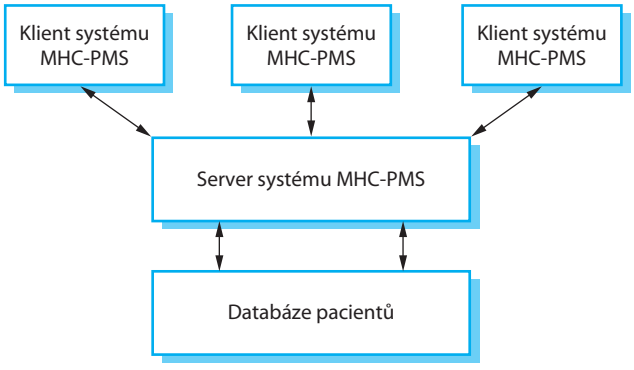
Obrázek 1.5 – Model aktivity inzulínové pumpy

Obrázek 1.4 znázorňuje hardwarové komponenty a organizaci inzulínové pumpy. Abychom porozuměli příkladům v této knize, stačí vědět, že senzor krve měří elektrickou vodivost krve za různých podmínek a tyto hodnoty lze korelovat s hladinou cukru v krvi. Inzulínová pumpa poskytuje v reakci na jediný pulz z řadiče jednu dávku inzulínu. Pokud je tedy potřeba poskytnout 10 jednotek inzulínu, řadič pumpě odešle 10 pulzů. Obrázek 1.5 je model aktivity UML, který ilustruje, jak software převádí vstupní hladinu cukru v krvi na sekvenci příkazů k pohonu inzulínové pumpy.

Je jasné, že se jedná o bezpečnostně kritický systém. Pokud pumpa selže nebo nefunguje správně, může dojít k poškození zdraví uživatele, který může v případě extrémních hladin cukru v krvi dokonce upadnout do kómatu. Tento systém tedy musí splnit dva zásadní obecné požadavky:

1. Systém je k dispozici pro dávkování inzulínu, kdykoli je potřeba.
2. Systém by měl fungovat spolehlivě a poskytovat správné množství inzulínu, které by odpovídalo aktuální hladině cukru v krvi.

Systém tedy musí být navržen a implementován tak, aby bylo zaručeno, že bude vždy těmto požadavkům vyhovovat. Podrobnější rozbor požadavků a analýzy toho, jak zajistit bezpečnost systému, naleznete v dalších kapitolách.



Obrázek 1.6 – Organizace systému MHC-PMS

1.3.2 Informační systém o pacientech v psychiatrické péči

Informační systém na podporu psychiatrické péče je zdravotnický informační systém, který uchovává informace o pacientech s duševními problémy a o léčbě, kterou absolvovali. Většina psychiatrických pacientů nepotřebuje lůžkovou léčbu, ale musí pravidelně navštěvovat speciální kliniku, kde se jim věnuje lékař s podrobnými znalostmi o jejich problémech. Aby byla docházka pro pacienty jednodušší, tato zařízení nefungují pouze při nemocnicích. Mohou být provozována také v místních zdravotních střediscích nebo komunitních centrech.

Systém MHC-PMS (Mental Health Care-Patient Management System, tj. psychiatrická zdravotní péče – systém správy pacientů) je informační systém určený pro použití v příslušných klinikách. Je založen na centralizované databázi informací o pacientech, ale zároveň jej lze spouštět v počítačích PC. Umožňuje tedy přístup a používání i v lokalitách, které nemají zabezpečenou síťovou konektivitu. Tam, kde mají místní systémy zabezpečený přístup k síti, používají informace o pacientech v databázi, ale mohou také stáhnout a používat místní kopie záznamů o pacientech v situacích, kdy konektivita není k dispozici. Nejedná se o úplný systém na správu zdravotnických záznamů, takže neumožňuje ukládat informace o jiných nemocech. Může však interagovat s jinými klinickými informačními systémy a vyměňovat si s nimi data. Uspořádání systému MHC-PMS je znázorněno na obrázku 1.6.

Systém MHC-PMS má dva obecné cíle:

1. Generovat přehledné informace, které umožní manažerům zdravotnických zařízení hodnotit výkony s ohledem na místní a celostátní měřítka.
2. Poskytovat zdravotnickému personálu aktuální informace, které jim pomohou při léčbě pacientů.

Z povahy psychiatrických problémů vyplývá, že tito pacienti si často nedokážou uspořádat život, takže mohou zmeškat schůzky, záměrně nebo náhodou ztrácet recepty a léky, zapomínat instrukce a mohou na zdravotnický personál klást nereálné požadavky. Mohou se také v klinice objevit bez předchozího objednání. V menším počtu případů mohou také představovat nebezpečí pro sebe nebo druhé osoby. Mohou často měnit adresy nebo mohou být dlouhodobě či krátkodobě bez domova. Pokud jsou pacienti nebezpeční, může být nutné je izolovat, tj. umístit na uzavřené nemocniční oddělení, kde je lze léčit a sledovat.

K uživatelům systému patří kliničtí pracovníci jako lékaři, sestry a zdravotničtí terénní pracovníci (sestry, které navštěvují pacienty doma a kontrolují postup jejich léčby). Mezi jiné než zdravotnické

uživatele se řadí recepční, kteří vyřizují telefonické objednávky na schůzku, pracovníci, kteří udržují systém záznamů, a administrativní síly, které generují sestavy.

Systém se používá k zaznamenávání informací o pacientech (jméno, adresa, věk, nejbližší příbuzný atd.), konzultacích (datum, navštívený lékař, subjektivní dojem z pacienta atd.), nemocech a léčebných postupech. V pravidelných intervalech se generují sestavy pro zdravotnický personál a manažery zdravotnického zařízení. Sestavy pro zdravotnický personál se obvykle zaměřují na informace o jednotlivých pacientech, zatímco sestavy pro management jsou anonymizované a týkají se statistiky nemocí, nákladů na léčbu atd.

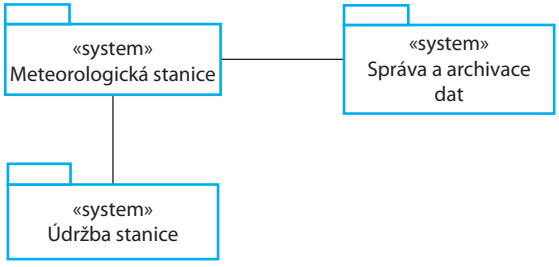
Systém má následující klíčové vlastnosti:

1. *Správa péče o jednotlivce* – kliničtí pracovníci mohou vytvářet záznamy o pacientech, upravovat informace v systému, zobrazovat historii pacientů atd. Systém podporuje datové souhrny. Lékaři, kteří se s pacientem zatím nesetkali, se díky tomu mohou rychle dozvědět o hlavních problémech pacienta a o dosud nasazené léčbě.
2. *Sledování pacientů* – systém pravidelně monitoruje záznamy o pacientech, kteří se účastní léčby, a při zjištění možných komplikací poskytuje varování. Takové varování se může objevit například v případě, že pacient nějakou dobu nenavštívil lékaře. Mezi nejdůležitější prvky monitorovacího systému patří sledování hospitalizovaných pacientů a zajištění toho, že budou ve vhodnou dobu provedeny zákonem nařízené kontroly.
3. *Administrativní vykazování* – systém každý měsíc generuje sestavy pro management, které informují o počtu pacientů léčených v každé klinice, počtu pacientů, kteří do systému péče vstoupili a kteří jej opustili, počtu hospitalizovaných pacientů, předepisovaných lécích a nákladech na ně atd.

Fungování systému závisí na dvou typech zákonů. Jde o zákony na ochranu osobních údajů, které definují důvěrnost osobních informací, a zákony o psychiatrické péči, kterými se řídí povinná hospitalizace pacientů, kteří představují nebezpečí pro sebe či jiné. Psychiatrická péče je v tomto ohledu výjimečná, protože se jedná o jedinou zdravotnickou specializaci, která může doporučit léčbu pacientů proti jejich vůli. Tyto postupy podléhají velmi přísné zákonné kontrole. Jedním z cílů systému MHC-PMS je zajistit, že bude personál vždy postupovat v souladu se zákonem a příslušná rozhodnutí budou zaznamenána pro případné soudní přezkoumání.

Stejně jako u všech zdravotnických systémů patří ke kritickým požadavkům na systém ochrana soukromí. Je zásadně důležité, aby informace o pacientech zůstaly důvěrné a nikdy k nim neměli přístup nepovolané osoby mimo autorizovaný zdravotnický personál a samotné pacienty. Systém MHC-PMS rovněž patří mezi bezpečnostně kritické systémy. Někteří pacienti s duševními chorobami mohou mít sklon k sebevraždám nebo mohou ohrožovat jiné osoby. Kdykoli je to možné, systém by měl varovat zdravotnické pracovníky před potenciálně sebevražednými nebo agresivními pacienty.

Celkový návrh systému musí zohlednit požadavky na soukromí a bezpečnost. Systém musí být dostupný v případě potřeby, jinak může dojít k ohrožení bezpečnosti a lékaři nemusí být schopni pacientům předepsat potřebné léky. V návrhu systému se objevuje potenciální konflikt – soukromí se udržuje nejsnáze, když existuje pouze jediná kopie systémových dat. Kvůli zajištění dostupnosti v případě výpadku serveru nebo odpojení od sítě je však vhodné uchovávat více kopií dat. Kompromisy mezi těmito požadavky se budeme zabývat v dalších kapitolách.



Obrázek 1.7 – Prostředí meteorologické stanice

1.3.3 Automatizovaná meteorologická stanice

Kvůli monitorování změn klimatu a zvýšení přesnosti předpovědí počasí v odlehlých regionech se může vláda země s velkými řídce obydlenými oblastmi rozhodnout, že v těchto místech instaluje několik set meteorologických stanic. Tyto meteorologické stanice shromažďují data ze sady přístrojů, které měří teplotu, tlak, délku slunečního svitu, srážky a rychlost a směr větru.

Automatizované meteorologické stanice tvoří součást většího systému (viz obrázek 1.7). Jedná se o systém informací o počasí, který sbírá data z meteorologických stanic a zpřístupňuje je ke zpracování jiným systémům. Na obrázku 1.7 jsou znázorněny tyto systémy:

1. *Systém meteorologické stanice* – odpovídá za shromažďování dat o počasí, vykonává předběžné zpracování dat a přenáší tato data do systému správy dat.
2. *Systém správy a archivace dat* – tento systém shromažďuje data ze všech automatizovaných meteorologických stanic, provádí zpracování a analýzu dat a archivuje data ve formě, kterou mohou načítat jiné systémy, jako například systémy na předpovídání počasí.
3. *Systém údržby stanic* – tento systém může pomocí satelitu komunikovat se všemi automatizovanými meteorologickými stanicemi kvůli monitorování jejich funkčního stavu a může operátorům poskytovat zprávy o problémech. Dokáže také aktualizovat integrovaný software v těchto systémech. V případě systémových potíží lze pomocí tohoto systému také systémy automatizovaných meteorologických stanic vzdáleně řídit.

Na obrázku 1.7 je pomocí symbolů sady UML uvedeno, že každý systém je tvořen sadou komponent. S použitím stereotypu UML «system» jsou také identifikovány samostatné systémy. Asociace mezi balíčky informují o tom, že dochází k výměně informací, ale v této fázi je není nutné definovat podrobněji.

Každá meteorologická stanice obsahuje více přístrojů, které měří parametry počasí, jako je například rychlost a směr větru, teplota vzduchu ve výšce a při zemi, tlak vzduchu a srážky za období 24 hodin. Všechny uvedené přístroje jsou řízeny softwarovým systémem, který pravidelně odečítá jednotlivé parametry a spravuje data shromážděná z přístrojů.

Systém meteorologické stanice funguje tak, že sbírá pozorování počasí v častých intervalech – například teploty se měří každou minutu. Vzhledem k tomu, že datová kapacita připojení přes satelit je poměrně malá, však meteorologická stanice zajišťuje určité místní zpracování a agregaci dat. Poté přenáší tato agregovaná data na základě požadavku systému shromažďování dat. Jestliže se z jakéhokoli důvodu nelze připojit, meteorologická stanice uchovává data místně do doby, než lze komunikaci obnovit.

Všechny meteorologické stanice jsou napájeny bateriemi a musí být zcela samostatné – nejsou k dispozici žádné externí zdroje napájení ani síťové kabely. Veškerá komunikace probíhá prostřednictvím relativně pomalého satelitního spojení a meteorologická stanice musí být vybavena nějakým mechanismem (solárními panely nebo větrnou turbínou) na dobíjení svých baterií. Vzhledem k tomu, že stanice se instalují v divočině, jsou vystaveny náročným povětrnostním podmínkám a mohou je poškozovat zvířata. Software stanice se tedy neomezuje jen na shromažďování dat. Musí také:

1. Monitorovat přístroje, napájení a komunikační hardware a oznamovat veškeré chyby systému správy.
2. Kontrolovat systémové napájení a zajistit, že se budou baterie dobíjet vždy, když to podmínky prostředí dovolí, ale při potenciálně nebezpečných povětrnostních podmínkách (například za silného větru) budou generátory vypnuty.
3. Umožnit dynamickou změnu konfigurace, kdy budou části softwaru nahrazeny novými verzemi nebo budou k systému v případě selhání jeho částí připojeny záložní přístroje.

Meteorologické stanice musí být samostatné a musí fungovat bez obsluhy. Z toho vyplývá, že instalovaný software je dosti složitý, i když samotné funkce shromažďování dat jsou poměrně jednoduché.

HLAVNÍ BODY

- Softwarové inženýrství je technická disciplína, která se zabývá všemi aspekty produkce softwaru.
- Software není pouze program či programy, ale zahrnuje také dokumentaci. Základními atributy softwarového produktu je možnost údržby, spolehlivost, bezpečnost, efektivita a přijatelnost.
- Softwarový proces obsahuje všechny aktivity, které jsou součástí vývoje softwaru. Všechny softwarové procesy zahrnují vysokoúrovňové aktivity specifikace, vývoje, validace a evoluce.
- Základní pojmy softwarového inženýrství lze univerzálně aplikovat na všechny typy vývoje systémů. K těmto základům se řadí softwarové procesy, spolehlivost, bezpečnost, požadavky a opakované použití.
- Existuje mnoho různých typů systémů a každý z nich při svém vývoji vyžaduje odpovídající nástroje a techniky softwarového inženýrství. Málokteré (pokud vůbec nějaké) konkrétní techniky návrhu a implementace se hodí pro všechny druhy systémů.
- Základní myšlenky softwarového inženýrství lze aplikovat na všechny typy softwarových systémů. K těmto základům se řadí řízené softwarové procesy, spolehlivost a bezpečnost softwaru, inženýring požadavků a opakované použití softwaru.
- Softwaroví inženýři mají odpovědnost vůči své profesi a celé společnosti. Neměli by se zabývat pouze technickými otázkami.
- Profesionální společnosti publikují své kódy chování, které určují standardy chování, jaké očekávají od svých členů.

DALŠÍ ZDROJE INFORMACÍ

„No silver bullet: Essence and accidents of software engineering“ (Univerzální řešení neexistuje: podstata a náhody softwarového inženýrství). Navzdory svému věku tento článek představuje dobrý obecný úvod do problematiky softwarového inženýrství. Základní vyznění článku zůstává v platnosti. (F. P. Brooks, *IEEE Computer*, **20** (4), duben 1987.) *http://doi.ieeecomputersociety.org/10.1109/MC.1987.1663532*.

„Software engineering code of ethics is approved“ (Byl schválen etický kód softwarového inženýrství). Tento článek se zabývá souvislostmi vývoje Etického kódu ACM/IEEE a zahrnuje jeho krátkou i dlouhou verzi. (*Comm. ACM*, D. Gotterbarn, K. Miller a S. Rogerson, říjen 1999.) *http://portal.acm.org/citation.cfm?doid=317665.317682*.

Professional Issues in Software Engineering (Profesionální otázky softwarového inženýrství). Tato vynikající kniha diskutuje právní a profesionální hlediska i etické aspekty. Svým praktickým přístupem podle autora této knihy překonává teoretičtější texty o etice. (F. Bott, A. Coleman, J. Eaton a D. Rowland, 3. vyd., 2000, Taylor and Francis.)

IEEE Software, březen/duben 2002. Toto speciální vydání časopisu je věnováno vývoji webového softwaru. Tato oblast se velmi rychle mění, takže některé články již poněkud zastaraly, ale většina z nich je stále relevantní. (*IEEE Software*, **19** (2), 2002.) *http://www2.computer.org/portal/web/software*.

„A View of 20th and 21st Century Software Engineering“ (Pohled na softwarové inženýrství 20. a 21. století). Pohled na minulost i budoucnost softwarového inženýrství od jednoho z prvních a nejvíce zasloužilých softwarových inženýrů. Barry Boehm identifikuje nadčasové principy softwarového inženýrství, ale zároveň naznačuje, že některé běžně používané postupy jsou již zastaralé. (B. Boehm, *Proc. 28th Software Engineering Conf.*, Shanghai. 2006.) *http://doi.ieeecomputersociety.org/10.1145/1134285.1134288*.

„Software Engineering Ethics" (Etika softwarového inženýrství). Zvláštní číslo časopisu IEEE Computer s mnoha články týkajícími se tohoto tématu. (*IEEE Computer*, **42** (6), červen 2009.)

CVIČENÍ

1.1. Vysvětlete, proč profesionální software nejsou pouze programy, které jsou vyvíjené pro zákazníka.

1.2. Jaký je nejdůležitější rozdíl mezi vývojem obecných softwarových produktů a vývojem zákaznického softwaru? Co to může v praxi znamenat pro uživatele obecných softwarových produktů?

1.3. Které čtyři důležité atributy by měl mít veškerý profesionální software? Navrhněte čtyři další atributy, které by mohly být v některých případech významné.

1.4. Vedle problémů heterogenity, podnikových a společenských změn a důvěry a bezpečnosti uveďte další problémy a výzvy, kterým bude softwarové inženýrství pravděpodobně čelit v 21. století. (Nápověda: nezapomeňte na životní prostředí.)

1.5. Na základě svých znalostí některých typů aplikací, které jsme diskutovali v části 1.1.2, vysvětlete na příkladech, proč podpora návrhu a vývoje různých typů aplikací vyžaduje specializované techniky softwarového inženýrství.

1.6. Vysvětlete, proč lze základní myšlenky softwarového inženýrství aplikovat na všechny typy softwarových systémů.

1.7. Vysvětlete, jak univerzální používání webu proměnilo softwarové systémy.

1.8. Diskutujte, zda by měli být profesionální inženýři certifikováni stejným způsobem jako lékaři nebo právníci.

1.9. Pro každou klauzuli v Etickém kódu ACM/IEEE uvedeném na obrázku 1.3 uveďte vhodný příklad, který bude danou klauzuli ilustrovat.

1.10. Kvůli účinnější prevenci terorismu mnoho zemí plánuje nebo vyvíjí počítačové systémy, které sledují mnoho občanů a jejich aktivit. Tento vývoj má pochopitelně dopady na ochranu soukromí. Diskutujte etické aspekty práce na vývoji takového typu systému.

CITACE

Gotterbarn, D., Miller, K. a Rogerson, S. (1999). Software Engineering Code of Ethics is Approved (Byl schválen etický kód softwarového inženýrství). Comm. ACM, **42** (10), 102–7.

Holdener, A. T. (2008). *Ajax: The Definitive Guide* (Ajax: nejobsáhlejší průvodce). Sebastopol, Ca.: O'Reilly and Associates.

Huff, C. a Martin, C. D. (1995). Computing Consequences: A Framework for Teaching Ethical Computing (Důsledky používání počítačů: schéma výuky etického uplatnění počítačů). Comm. ACM, **38** (12), 75–84.

Johnson, D. G. (2001). *Computer Ethics* (Počítačová etika). Englewood Cliffs, NJ: Prentice Hall.

Laudon, K. (1995). Ethical Concepts and Information Technology (Etické koncepce a informační technologie). Comm. ACM, **38** (12), 33–9.

Naur, P. a Randell, B. (1969). Software Engineering (Softwarové inženýrství): zpráva o konferenci sponzorované vědeckou radou NATO, Garmisch, Německo. 7. – 11. října 1968.

Softwarové procesy

2

Cíle

Cílem této kapitoly je představit myšlenku softwarového procesu – koherentní sady aktivit při produkci softwaru. V této kapitole:

- porozumíte principům softwarových procesů a modelů softwarových procesů,
- seznámíte se se třemi obecnými modely softwarových procesů a tím, kdy je lze použít,
- dozvíte se o základních aktivitách procesů při inženýringu požadavků na software, vývoje, testování a evoluce softwaru,
- pochopíte, proč je potřeba procesy uspořádat s ohledem na zvládání změn požadavků na software a návrhu,
- porozumíte tomu, jak proces RUP (Rational Unified Process) integruje kvalitní postupy softwarového inženýrství a vytváří přizpůsobitelné softwarové procesy.

Softwarový proces je sekvence souvisejících aktivit, které vedou k produkci softwarového produktu. Tyto aktivity mohou zahrnovat vývoj softwaru od nuly ve standardním programovacím jazyce jako Java nebo C. Podnikové aplikace však není nutné vyvíjet tímto způsobem. Nový podnikový software se nyní často vyvíjí rozšiřováním a úpravami stávajících systémů nebo konfigurací a integrací standardně prodávaného softwaru nebo systémových komponent.

Existuje mnoho různých softwarových procesů, ale všechny musí zahrnovat čtyři aktivity, které jsou pro softwarové inženýrství klíčové:

1. *Specifikace softwaru* – je nutné definovat funkce softwaru a omezení jeho činnosti.
2. *Návrh a implementace softwaru* – vytvořený software musí splňovat specifikace.
3. *Validace softwaru* – software je nutné validovat, aby bylo zajištěno, že jeho vlastnosti odpovídají požadavkům zákazníka.
4. *Evoluce softwaru* – software se musí vyvíjet, aby splňoval proměnlivé potřeby zákazníka.

Tyto aktivity jsou v určité formě součástí všech softwarových procesů. V praxi se samozřejmě jedná o složité aktivity samy o sobě, které zahrnují dílčí aktivity typu validace požadavků, návrhu architektury, testování jednotek atd. Některé aktivity také slouží k podpoře procesů – jedná se například o správu dokumentace a konfigurace softwaru.

Při popisování procesů obvykle diskutujeme aktivity v těchto procesech, jako je například specifikace datového modelu, návrh uživatelského rozhraní atd., a uspořádání těchto aktivit. Kromě aktivit mohou tedy popisy procesů obsahovat také:

1. Produkty, které jsou výsledkem aktivity procesu. Například výsledkem aktivity návrhu architektury může být model softwarové architektury.
2. Role, které odrážejí odpovědnosti osob, které se procesu účastní. Jako příklady rolí lze uvést projektového manažera, manažera konfigurace, programátora atd.
3. Předběžné a následné podmínky, což jsou tvrzení, která platí před nebo po provedení aktivity procesu nebo vytvoření produktu. Před zahájením návrhu architektury může být například stanovena předběžná podmínka, že všechny požadavky musí schválit zákazník. Po dokončení této aktivity může platit následná podmínka, že je nutné zkontrolovat modely UML s popisem architektury.

Softwarové procesy jsou složité a stejně jako všechny duševní a kreativní procesy jsou závislé na lidech, kteří přijímají jednotlivá rozhodnutí. Neexistuje žádný ideální proces a většina organizací si vytvořila vlastní procesy vývoje softwaru. Procesy se vyvíjejí tak, aby využily schopností pracovníků organizace a konkrétních vlastností vyvíjených systémů. U některých systémů, jako jsou bezpečnostně kritické systémy, je nutné vytvořit velmi strukturovaný proces vývoje. V případě podnikových systémů s rychle se měnícími požadavky bude pravděpodobně efektivnější méně formální a pružnější proces.

Softwarové procesy se někdy dělí na plánované a agilní. Plánované jsou takové procesy, kde jsou všechny aktivity procesu předem naplánovány a jejich postup se porovnává s tímto plánem. V agilních procesech, kterými se budeme zabývat v kapitole 3, je plánování inkrementální a proces lze snáze změnit tak, aby odrážel změněné požadavky zákazníka. Jak diskutují Boehm a Turner (2003), každý přístup se hodí pro odlišný typ softwaru. Obecně je nutné najít rovnováhu mezi plánovanými a agilními procesy.

Ačkoli neexistuje žádný „ideální" softwarový proces, v mnoha organizacích je možné softwarový proces do jisté míry zdokonalit. Procesy mohou obsahovat zastaralé techniky nebo nemusí využívat optimálních postupů průmyslového softwarového inženýrství. Mnoho organizací při vývoji softwaru skutečně s metodami softwarového inženýrství vůbec nepracuje.

Softwarové procesy lze zlepšit pomocí standardizace procesů, kdy se omezuje rozmanitost softwarových procesů v rámci organizace. Tento postup pomáhá zlepšit komunikaci a zkrátit čas školení a zvyšuje ekonomičnost automatizované podpory procesů. Standardizace také představuje důležitý první krok při zavádění nových metod, technik a optimálních postupů softwarového inženýrství. Zdokonalováním softwarového procesu se budeme podrobněji zabývat v kapitole 26.

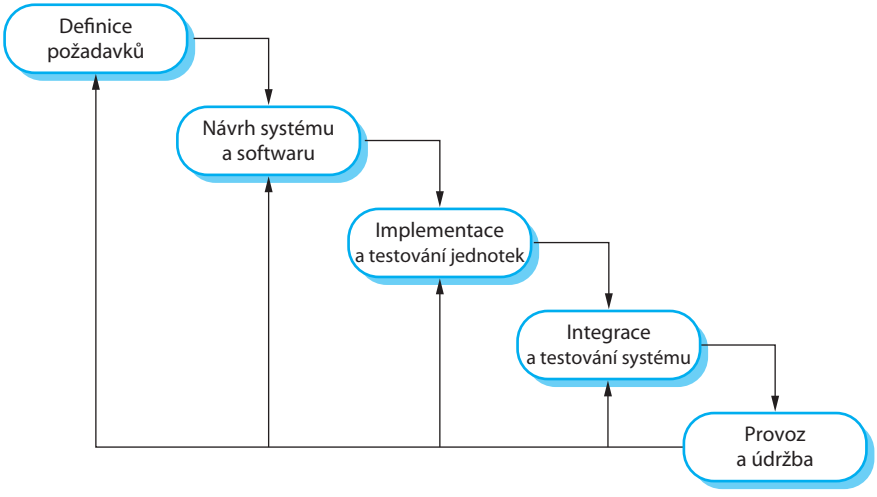
2.1 Modely softwarových procesů

Jak jsme vysvětlili v kapitole 1, model softwarového procesu zjednodušeně reprezentuje příslušný softwarový proces. Každý model procesu znázorňuje proces z určité perspektivy, a podává o něm tedy jen částečné informace. Například model aktivity procesu ukazuje aktivity a jejich pořadí, ale nemusí informovat o rolích osob, které se na těchto aktivitách podílejí. V této části představíme několik značně obecných modelů procesů (někdy se označují jako „paradigmata procesů") a předvedeme je z hlediska architektury. To znamená, že se zaměříme na rámcovou strukturu procesu, ale nikoli na podrobnosti jednotlivých aktivit.

Tyto obecné modely neslouží jako kompletní popis softwarových procesů. Jedná se spíše o abstrakci procesů, pomocí nichž je možné vysvětlit různé přístupy k vývoji softwaru. Můžeme si je představit jako strukturu procesu, kterou lze rozšířit a adaptovat tak, abychom dostali konkrétnější procesy softwarového inženýrství.

Budeme se zde zabývat těmito modely procesů:

1. *Vodopádový model* – tento model zpracovává základní aktivity procesu (specifikaci, vývoj, validaci a evoluci) a reprezentuje je jako samostatné fáze procesu, jako je specifikace požadavků, návrh softwaru, implementace, testování atd.



Obrázek 2.1 – Vodopádový model

2. *Inkrementální vývoj* – tento přístup prokládá aktivity specifikace, vývoje a validace. Systém se vyvíjí jako řada verzí (inkrementů), kdy každá další verze přidává funkce k předchozí verzi.
3. *Softwarové inženýrství orientované na opakované použití:* tento přístup vychází z existence značného počtu opakovaně použitelných komponent. Proces vývoje systému se soustřeďuje na integraci těchto komponent do systému místo jejich vyvíjení od začátku.

Tyto modely se vzájemně nevylučují a často se používají společně, zejména při vývoji velkých systémů. V případě velkých systémů je rozumné kombinovat hlavní výhody vodopádového modelu a modelu inkrementálního vývoje. Chceme-li navrhnout softwarovou architekturu, která bude kompatibilní se základními požadavky na systém, potřebujeme tyto požadavky znát. Tento systém nelze vyvíjet inkrementálně. Podsystémy většího systému lze vyvíjet na základě různých přístupů. Části systému, které jsou dobře analyzované, lze specifikovat a vyvíjet procesem založeným na vodopádovém modelu. Části systému, které lze předem specifikovat jen obtížně (například uživatelské rozhraní), je vhodné vždy vyvíjet pomocí inkrementálního přístupu.

2.1.1 Vodopádový model

První publikovaný model procesu vývoje softwaru byl odvozen z obecnějších procesů systémového inženýrství (Royce, 1970). Tento model je znázorněn na obrázku 2.1. Vzhledem k tomu, že postupuje formou kaskády jednotlivých fází, označuje se tento model jako „vodopádový model“ neboli životní cyklus softwaru. Vodopádový model představuje příklad plánovaného procesu. Principiálně vzato je nutné naplánovat všechny aktivity procesu dříve, než se na nich začne pracovat.

Hlavní fáze vodopádového modelu přímo odrážejí základní vývojové aktivity:

1. *Analýza a definice požadavků* – na základě konzultací s uživateli systému se určují systémové služby, omezení a cíle. Poté se definují podrobněji a slouží jako specifikace systému.
2. *Návrh systému a softwaru* – proces návrhu systému přiděluje požadavky na hardwarové či softwarové systémy tak, že určuje celkovou systémovou architekturu. Návrh softwaru zahrnuje identifikaci a popis základních abstrakcí softwarového systému a jejich vztahů.
3. *Implementace a testování jednotek* – během této fáze se realizuje návrh systému jako sada programů nebo programových jednotek. Při testování jednotek se ověřuje, zda každá jednotka splňuje příslušnou specifikaci.
4. *Integrace a testování systému* – jednotlivé programové jednotky nebo programy se integrují a testují jako kompletní systém, aby bylo zajištěno, že vyhovují požadavkům na software. Po testování se softwarový systém předává zákazníkovi.
5. *Provoz a údržba* – normálně (ačkoli to nemusí platit vždy) se jedná o nejdelší fázi životního cyklu. Systém se instaluje a předává k praktickému používání. Údržba zahrnuje opravu chyb, které nebyly zjištěny v dřívějších fázích životního cyklu, zlepšování implementace systémových jednotek a zdokonalování systémových služeb s ohledem na nové požadavky.

Výsledkem každé fáze je zpravidla jeden nebo více schválených (podepsaných) dokumentů. Následující fáze by neměla začít dříve, než je dokončena fáze předchozí. V praxi se tyto stupně často překrývají a dochází mezi nimi k přenosu informací. Během návrhu se identifikují problémy týkající se požadavků. Při kódování jsou nalezeny problémy v návrhu atd. Softwarový proces nelze popsat jednoduchým lineárním modelem, ale zahrnuje zpětnou vazbu mezi jednotlivými fázemi. Dokumenty připravené v každé fázi pak může být nutné upravit s ohledem na provedené změny.

Vzhledem k nákladům na produkci a schvalování dokumentů mohou být iterace nákladné a vyžadovat zásadní přepracování. Po malém počtu iterací se proto běžně postupuje tak, že se části vývoje (jako například specifikace) zmrazí a proces pokračuje pozdějšími fázemi vývoje. Problémy se ponechávají k pozdějšímu řešení, ignorují se nebo se programátorsky obcházejí. Toto předčasné zmrazení požadavků může způsobit, že systém nebude plnit požadavky uživatelů. Může také vést k chybně strukturovaným systémům, protože se problémy návrhu obcházejí různými implementačními triky.

Během závěrečných fází životního cyklu (provoz a údržba) se software začíná používat. Objevují se chyby a opomenutí původních požadavků na software. Ukazují se programové a návrhové chyby a identifikují se požadavky na nové funkce. Má-li tedy systém zůstat užitečný, musí se dále vyvíjet. Provádění těchto změn (údržba softwaru) může vyžadovat opakování předchozích fází procesu.

SOFTWAROVÉ INŽENÝRSTVÍ TYPU CLEANROOM

Příkladem formálního procesu vývoje, který původně vytvořila společnost IBM, je proces Cleanroom. V procesu Cleanroom je každý inkrementální krok vývoje softwaru formálně specifikován a tato specifikace se převádí na implementaci. Správnost softwaru se demonstruje formálním přístupem. Jednotky se v rámci procesu nijak netestují na chyby a testování systému se zaměřuje na hodnocení jeho spolehlivosti.

Cílem procesu Cleanroom je vytvořit software bez chyb, aby měly poskytované systémy vysokou úroveň spolehlivosti.

http://www.SoftwareEngineering-9.com/Web/Cleanroom/

Vodopádový model je konzistentní s jinými modely technických procesů a v každé fázi vzniká dokumentace. Tím se proces zviditelňuje, takže vedoucí pracovníci mohou sledovat postup vzhledem k plánu vývoje. Hlavním problémem tohoto modelu je nepružné rozdělení projektu na samostatné fáze. V raných fázích procesu je nutné pevně stanovit určité vlastnosti, takže lze obtížněji reagovat na měnící se požadavky zákazníků.

Vodopádový model je obecně řečeno vhodné používat pouze v situacích, kdy jsou požadavky dobře srozumitelné a není pravděpodobné, že se v průběhu vývoje budou zásadně měnit. Vodopádový model však odráží typ procesu, který je součástí jiných technických projektů. Vzhledem k tomu, že je jednodušší na celý projekt nasadit společný model správy, softwarové procesy založené na vodopádovém modelu se dodnes běžně používají.

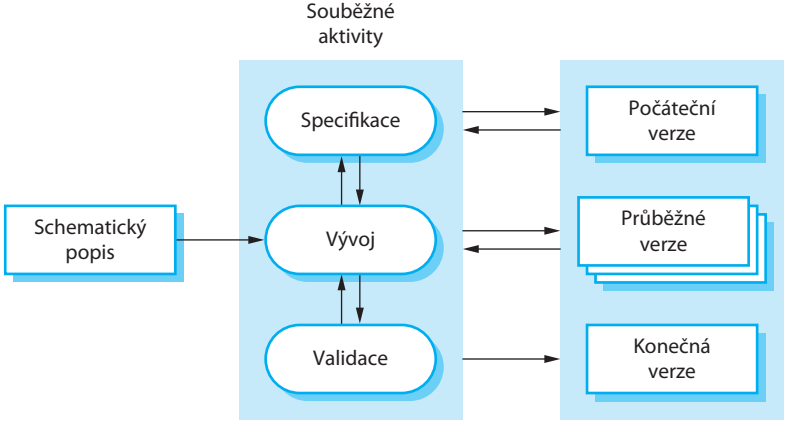
Důležitou variantou vodopádového modelu je formální vývoj systému, kdy se vytváří matematický model specifikace systému. Tento model se poté při zachování své konzistence pomocí matematických transformací konkretizuje do spustitelného kódu. Na základě předpokladu, že matematické transformace jsou správné, lze tedy poté se značnou jistotou tvrdit, že program generovaný tímto způsobem je konzistentní se svou specifikací.

Procesy formálního vývoje, jako je proces založený na metodě B (Schneider, 2001; Wordsworth, 1996), jsou mimořádně vhodné k vývoji systémů, které mají přísné požadavky na bezpečnost, zabezpečení či spolehlivost. Formální přístup zjednodušuje produkci případu bezpečnosti nebo zabezpečení. Zákazníci nebo regulační orgány se tak mohou přesvědčit, že systém skutečně splňuje své požadavky na bezpečnost nebo zabezpečení.

Procesy založené na formálních transformacích se obecně používají pouze při vývoji systémů s kritickými požadavky na bezpečnost či zabezpečení. Při jejich vývoji je nutné uplatnit speciální znalosti. U většiny systémů tento proces oproti jiným přístupům k vývoji systému nenabízí zásadní nákladové výhody.

2.1.2 Inkrementální vývoj

Inkrementální vývoj je založen na principu, že nejdříve se tvoří počáteční implementace, předkládá se uživatelům ke komentování a vyvíjí se přes několik verzí, dokud nevznikne odpovídající systém (obrázek 2.2). Procesy specifikace, vývoje a validování nejsou samostatné, ale vzájemně se prolínají. Mezi jednotlivými aktivitami přitom funguje rychlá zpětná vazba.



Obrázek 2.2 – Inkrementální vývoj

Inkrementální vývoj softwaru, který tvoří základní složku agilních přístupů, je pro většinu podniků, elektronických obchodů a osobních systémů vhodnější než vodopádový přístup. Inkrementální vývoj odráží způsob, jakým lidé běžně řeší problémy. Málokdy předem zpracováváme úplné řešení problému, ale postupujeme k němu pomocí řady kroků. Když si přitom uvědomíme, že jsme udělali chybu, vrátíme se o několik kroků zpět. Díky inkrementálnímu vývoji softwaru se při vývoji snáze a levněji zavádějí změny.

Každý inkrement či verze systému zahrnuje některé funkce, které zákazník požaduje. Obecně platí, že první inkrementy systému zahrnují nejdůležitější a nejakutněji potřebné funkce. Zákazník tedy může systém zhodnotit v relativně rané fázi vývoje a přesvědčit se, zda systém poskytuje to, co od něj očekává. V opačném případě stačí změnit pouze aktuální inkrement a případné nové funkce definovat pro pozdější inkrementy.

Inkrementální vývoj poskytuje v porovnání s vodopádovým modelem tři zásadní výhody:

1. Snižují se náklady na zavedení měnících se požadavků zákazníka. Analýza a dokumentace, které je potřeba přepracovat, mají mnohem menší rozsah než při stejné situaci u vodopádového modelu.
2. Zpětnou vazbu od zákazníka lze snáze získat pro již provedenou vývojovou práci. Zákazníci mohou komentovat demonstrační verze softwaru a sledovat, kolik požadavků zatím bylo implementováno. Zákazníci jen obtížně posuzují postup na základě dokumentů návrhu softwaru.
3. Lze rychleji poskytnout a nasadit užitečný software u zákazníka, i když tyto verze softwaru ještě nezahrnují všechny potřebné funkce. Zákazníci mohou se softwarem pracovat a využívat jeho přínosy dříve, než je tomu možné u vodopádového modelu.

PROBLÉMY S INKREMENTÁLNÍM VÝVOJEM

Inkrementální vývoj má sice mnoho výhod, ale zároveň přináší i jisté potíže. Primární důvod problémů spočívá v tom, že velké organizace postupně vyvinuly své byrokratické postupy, které nemusí být v souladu s méně formálním iterativním či agilním procesem.

Tyto postupy mohou mít dobrý důvod – mohou například existovat postupy, které zajišťují, že software odpovídajícím způsobem implementuje jisté zákony (například účetní pravidla podle zákona Sarbanes-Oxley v USA). Změna těchto postupů nemusí být možná, takže dochází k nevyhnutelnému konfliktu s procesem vývoje.

http://www.SoftwareEngineering-9.com/Web/IncrementalDev/

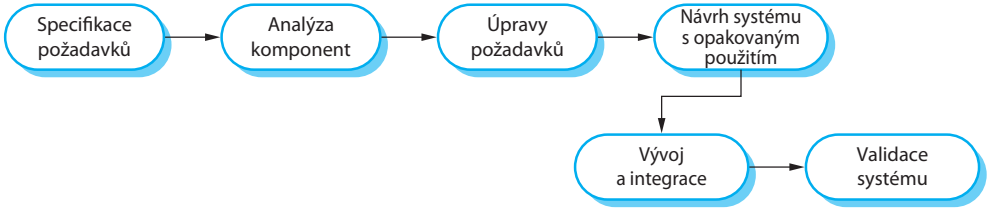
Inkrementální vývoj určitého typu nyní představuje nejrozšířenější přístup k vývoji aplikačních systémů. Tento přístup může být plánovaný, agilní, nebo nejčastěji zahrnuje směs obou těchto přístupů. Při plánovaném přístupu jsou inkrementy systému určeny předem. Jestliže je přijat agilní přístup, identifikují se rané inkrementy, ale vývoj pozdějších inkrementů závisí na postupu a prioritách zákazníka.

Z hlediska managementu má inkrementální přístup dva problémy:

1. Proces není viditelný. Aby mohli manažeři sledovat postup, potřebují pravidelné výstupy. Jestliže se systém vyvíjí rychle, není ekonomicky efektivní vytvářet dokumenty, které odrážejí každou verzi systému.
2. Spolu s přidáváním nových inkrementů se struktura systému zpravidla degraduje. Bez časových a finančních investic do refaktoringu, který by software zdokonalil, se při častých změnách obvykle poškozuje struktura softwaru. Zahrnutí dalších softwarových změn je poté stále obtížnější a dražší.

Problémy inkrementálního vývoje se zvýrazňují hlavně u velkých a složitých systémů s dlouhým životním cyklem, kde části systému vyvíjejí různé týmy. Velké systémy potřebují stabilní strukturu nebo architekturu a s ohledem na tuto architekturu je nutné jasně definovat odpovědnosti různých týmů, které pracují na jednotlivých částech systému. Tato pravidla je nutné plánovat dopředu a nikoli vyvíjet inkrementálně.

Systém lze vyvíjet inkrementálně a nabídnout jej zákazníkovi ke komentování, aniž by byl systém skutečně dodán zákazníkovi a nasazen v jeho prostředí. Inkrementální dodávání a nasazování znamená, že se software používá ve skutečných provozních podmínkách. To není vždy možné, protože experimentování s novým programem může narušit normální podnikové procesy. Výhody a nevýhody inkrementálního dodávání budeme rozebírat v části 2.3.2.



Obrázek 2.3 – Softwarové inženýrství orientované na opakované použití

2.1.3 Softwarové inženýrství orientované na opakované použití

U většiny softwarových projektů se do jisté míry opakovaně používají předchozí kódy. Často k tomu dochází neformálně, když pracovníci na projektu vědí o návrhu nebo kódu, který je podobný tomu požadovanému. Vyhledají příslušná data, podle potřeby je upraví a zahrnou je do nového systému.

Toto neformální opakované použití probíhá bez ohledu na to, který vývojový proces se používá. Ovšem v 21. století se široce uplatňují procesy vývoje softwaru, které se zaměřují na opakované použití existujícího softwaru. Přístupy orientované na opakované použití jsou založeny na rozsáhlé základně opakovaně použitelných softwarových komponent a integrujícím rámci, který umožňuje tyto komponenty skládat. Uvedené komponenty jsou někdy systémy samy o sobě (systémy krabicového softwaru), které mohou poskytovat specifické funkce – například zpracování textu nebo tvorbu tabulek.

Obecný model procesu pro vývoj založený na opakovaném použití je znázorněn na obrázku 2.3. Počáteční fáze specifikace požadavků a fáze validace jsou sice srovnatelné s jinými softwarovými procesy, ale mezilehlé fáze se v procesu orientovaném na opakované použití liší. Jedná se o tyto fáze:

1. *Analýza komponent* – na základě analýzy požadavků probíhá hledání komponent, které umožní danou specifikaci implementovat. Obvykle se nepodaří najít přesnou shodu a komponenty, které lze použít, poskytují pouze část požadovaných funkcí.
2. *Úpravy požadavků* – během této fáze probíhá analýza požadavků na základě informací o nalezených komponentách. Tyto požadavky se poté upravují, aby odpovídaly dostupným komponentám. Tam, kde úpravy nejsou možné, může dojít k opakované analýze komponent, aby bylo možné vyhledat alternativní řešení.
3. *Návrh systému s opakovaným použitím* – v této fázi je navržena struktura systému nebo se opakovaně používá existující struktura. Návrháři zohledňují opakovaně používané komponenty a organizují strukturu tak, aby jim odpovídala. Pokud nejsou opakovaně použitelné komponenty k dispozici, může být nutné navrhnout nový software.

4. *Vývoj a integrace* – dochází k vývoji softwaru, který nelze pořídit externě, a integrují se komponenty s krabicovými systémy za vzniku nového systému. Systémová integrace v tomto modelu může být součástí vývojového procesu a nikoli samostatná aktivita.

V procesu orientovaném na opakované použití lze použít tři typy softwaru:

1. Webové služby, které se vyvíjejí podle standardů služeb a které umožňují vzdálené volání.
2. Kolekce objektů, které se vyvíjejí jako balíček pro integraci s architekturou komponent, jako je .NET či J2EE.
3. Samostatné softwarové systémy, které se konfigurují pro použití v konkrétním prostředí.

Softwarové inženýrství orientované na opakované použití samozřejmě poskytuje tu výhodu, že omezuje rozsah vyvíjeného softwaru a snižuje tedy náklady a rizika. Obvykle také vede k rychlejšímu dodávání softwaru. Je však nutné přijímat kompromisy týkající se požadavků, což může vést k systému, který nesplňuje skutečné požadavky uživatelů. Kromě toho dochází ke ztrátě části kontroly nad evolucí systému, protože organizace, která používá nové verze opakovaně použitelných komponent, nemá na jejich podobu vliv.

Opakované použití systému je velmi důležité, a tomuto tématu proto věnujeme několik kapitol ve třetí části knihy. Obecnými otázkami týkajícími se opakovaného použití zakázkového a krabicového softwaru se budeme zabývat v kapitole 16, softwarovým inženýrstvím založeným na komponentách v kapitolách 17 a 18 a systémy orientovanými na služby v kapitole 18.

2.2 Aktivity procesů

Reálné softwarové procesy střídají technické, kolaborativní a manažerské aktivity s celkovým cílem určit, navrhnout, implementovat a otestovat softwarový systém. Softwaroví vývojáři při své práci používají různé softwarové nástroje. Nástroje jsou užitečné hlavně k podpoře editování různých typů dokumentů a ke správě mimořádného objemu podrobných informací, které se během velkého softwarového projektu generují.

Čtyři základní aktivity procesů – specifikace, vývoj, validace a evoluce – jsou v různých vývojových procesech uspořádány odlišně. Ve vodopádovém modelu jsou organizovány v řadě za sebou, zatímco při inkrementálním vývoji dochází k jejich prokládání. Způsob provádění těchto aktivit závisí na použitém typu softwaru, osobách a organizačních strukturách. Například při extrémním programování se specifikace píší na kartičky. Testy lze spouštět a vyvíjejí se před samotným programem. Evoluce může zahrnovat zásadní restrukturalizaci nebo refaktoring systému.

2.2.1 Specifikace softwaru

Specifikace softwaru či inženýring požadavků je proces, který umožňuje určit a definovat, které služby budou od systému požadovány, a identifikovat omezení na fungování a vývoj systému. Inženýring požadavků představuje zvláště kritickou fázi softwarového procesu, protože chyby v této fázi nevyhnutelně vedou k pozdějším problémům při návrhu a implementaci systému.

NÁSTROJE NA VÝVOJ SOFTWARU

Nástroje na vývoj softwaru (někdy označované jako nástroje CASE – Computer-Aided Software Engineering) jsou programy, které slouží k podpoře aktivit procesů softwarového inženýrství. Tyto nástroje tedy zahrnují návrhové editory, datové slovníky, kompilátory, ladicí programy, nástroje na konstrukci systému atd.

Softwarové nástroje podporují procesy tím, že automatizují některé aktivity procesů a poskytují informace o vyvíjeném softwaru. Automatizovat lze například tyto aktivity:

- Vývoj grafických modelů systému jako součást specifikace požadavků nebo návrhu softwaru
- Generování kódu z těchto grafických modelů
- Generování uživatelských rozhraní z popisu uživatelského rozhraní, které interaktivně vytvářejí uživatelé
- Ladění programu na základě poskytnutých informací o spouštěném programu
- Automatizovaný překlad programů napsaných ve starší verzi programovacího jazyka na novější verzi

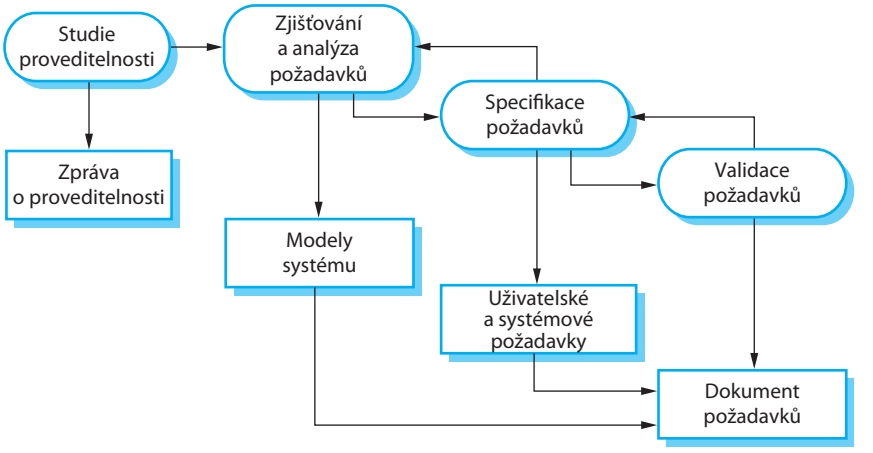
Nástroje se mohou kombinovat do architektury, která se označuje jako prostředí IDE (Interactive Development Environment). Toto prostředí poskytuje společnou sadu funkcí, s nimiž mohou nástroje pracovat. Nástroje tedy mohou snáze komunikovat a fungovat integrovaným způsobem. Široce se používá například integrované vývojové prostředí ECLIPSE, které bylo navrženo tak, aby zahrnovalo mnoho různých softwarových nástrojů.

http://www.SoftwareEngineering-9.com/Web/CASE/

Cílem procesu inženýringu požadavků (viz obrázek 2.4) je vytvořit odsouhlasený dokument požadavků, který specifikuje požadavky vyhovující všem zainteresovaným osobám. Požadavky se obvykle předkládají se dvěma úrovněmi podrobností. Koncoví uživatelé a zákazníci požadují obecný přehled požadavků, zatímco vývojáři systému potřebují podrobnější specifikaci systému.

Proces inženýringu požadavků obsahuje čtyři hlavní aktivity:

1. *Studie proveditelnosti* – zpracuje se odhad, zda lze identifikované požadavky uživatelů uspokojit pomocí aktuálních softwarových a hardwarových technologií. Studie analyzuje, zda navržený systém bude ekonomicky efektivní z podnikového hlediska a zda jej lze vyvinout s ohledem na aktuální rozpočtová omezení. Studie proveditelnosti by měla být relativně levná a rychlá. Výsledek by měl vést k rozhodnutí, zda bude proces pokračovat podrobnější analýzou.
2. *Zjišťování a analýza požadavků* – jedná se o proces odvozování požadavků na systém z pozorování stávajících systémů, diskusí s potenciálními uživateli a zákazníky, analýzy úkolů atd. Může zahrnovat vývoj jednoho nebo více modelů a prototypů systému. Tyto modely pomáhají specifikovanému systému lépe porozumět.
3. *Specifikace požadavků* – jedná se o aktivitu, kdy dochází k převodu informací získaných při aktivitě analýzy do dokumentu, který definuje sadu požadavků. Tento dokument může obsahovat dva typy požadavků. Uživatelské požadavky jsou abstraktní požadavky na systém od zákazníků a koncových uživatelů. Systémové požadavky představují podrobnější popis poskytovaných funkcí.



Obrázek 2.4 – Proces inženýringu požadavků

4. *Validace požadavků* – tato aktivita kontroluje realističnost, konzistenci a úplnost požadavků. Během tohoto procesu se nevyhnutelně zjišťují chyby v dokumentu požadavků. Tento dokument je poté nutné upravit, aby se tyto problémy vyřešily.

Aktivity procesu požadavků samozřejmě nepostupují přísně sekvenčně. Analýza požadavků pokračuje i při definování a specifikaci a v celém procesu se objevují stále nové požadavky. Aktivity analýzy, definice a specifikace se tedy vzájemně prolínají. U agilních metod, jako je extrémní programování, se požadavky vyvíjejí inkrementálně podle uživatelských priorit a požadavky se zjišťují od uživatelů, kteří patří do vývojového týmu.

2.2.2 Návrh a implementace softwaru

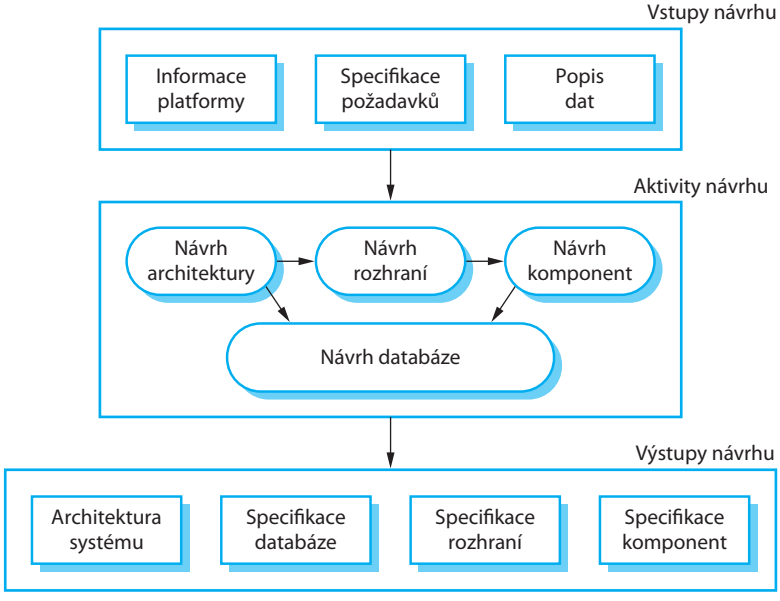
Fáze implementace při vývoji softwaru zahrnuje proces převodu specifikace systému do spustitelné formy. Vždy obsahuje procesy návrhu a programování softwaru, ale pokud se používá inkrementální přístup k vývoji, může být její součástí také upřesňování specifikace softwaru.

Návrh softwaru popisuje strukturu implementovaného softwaru, datové modely a struktury používané systémem, rozhraní mezi systémovými komponentami a někdy i použité algoritmy. Návrháři nedospívají ke konečnému návrhu okamžitě, ale vyvíjejí jej iterativně. Při vývoji systému doplňují formální prvky a podrobnosti a přitom se neustále vracejí zpět, aby opravili předchozí části návrhu.

Obrázek 2.5 znázorňuje abstraktní model tohoto procesu, který obsahuje vstupy procesu návrhu, aktivity procesu a produkované dokumenty, které představují výstup tohoto procesu. Diagram naznačuje, že jednotlivé fáze procesu návrhu jsou uspořádány sekvenčně. V praxi se ovšem aktivity procesu návrhu prokládají. Všechny procesy návrhu vyžadují zpětnou vazbu mezi jednotlivými fázemi a následné přepracování návrhu.

Většina softwaru obsahuje rozhraní s jinými softwarovými systémy. Jedná se operační systém, databázi, middleware a další aplikační systémy. Tvoří „softwarovou platformu" čili prostředí, ve kterém má software fungovat. Informace o této platformě představují zásadně důležitý vstup do procesu návrhu, protože návrháři se musí rozhodnout, jak software co nejlépe integrovat s jeho prostředím. Specifikace požadavků představuje popis funkcí, které musí software poskytovat, a požadavků na jeho výkon a spolehlivost. Pokud má systém zpracovávat existující data, je nutné do specifikace platformy zahrnout popis

těchto dat. Jinak musí být popis dat vstupem procesu návrhu, aby bylo možné definovat organizaci systémových dat.



Obrázek 2.5 – Obecný model procesu návrhu

Aktivity procesu návrhu se liší v závislosti na vyvíjeném systému. Například systémy fungující v reálném čase vyžadují návrh časování, ale nemusí obsahovat databázi, takže ji není nutné ani navrhovat. Obrázek 2.5 představuje čtyři aktivity, které mohou být součástí procesu návrhu informačních systémů:

1. *Návrh architektury*, kde se identifikuje celková struktura systému, jeho základní komponenty (někdy označované jako podsystémy nebo moduly), jejich vztah a způsob jejich distribuce.
2. *Návrh rozhraní*, kde se definují rozhraní mezi komponentami systému. Tato specifikace rozhraní musí být jednoznačná. Díky přesnému popisu rozhraní lze komponentu používat, aniž by jiné komponenty musely vědět, jak je implementována. Po schválení specifikací rozhraní lze komponenty navrhovat a vyvíjet souběžně.

STRUKTUROVANÉ METODY

Strukturované metody představují přístup k návrhu softwaru, kdy se definují grafické modely, které je potřeba vyvíjet v rámci procesu návrhu. Metoda může zároveň definovat proces vývoje modelů a pravidla, která se vztahují na jednotlivé typy modelů. Strukturované metody vedou ke standardizované dokumentaci systému a hodí se zejména k tomu, aby poskytovaly vývojovou strukturu méně zkušeným softwarovým vývojářům.

http://www.SoftwareEngineering-9.com/Web/Structured-methods/

3. *Návrh komponent*, kde lze postupně popsat jednotlivé komponenty systému a navrhnout, jak budou fungovat. Může se jednat o jednoduchý popis předpokládaných implementovaných funkcí, kdy konkrétní návrh zůstane na programátorovi. Alternativně může být tento dokument tvořen

seznamem změn, které je nutné provést u opakovaně použitelné komponenty, nebo se může jednat o podrobný model návrhu. Z modelu návrhu lze automaticky generovat implementaci.

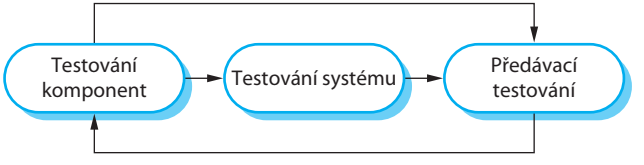
4. *Návrh databáze*, ve kterém se navrhují systémové datové struktury a způsob, jakým budou tyto struktury reprezentovány v databázi. I u této aktivity platí, že konkrétní činnosti závisí na tom, zda se bude opakovaně používat existující databáze, nebo bude vytvořena databáze nová.

Tyto aktivity vedou k sadě výstupů návrhu, které jsou rovněž znázorněny na obrázku 2.5. Výstupy se značně liší svou podrobností a vzhledem. U kritických systémů je nutné vytvořit podrobné návrhové dokumenty, které budou obsahovat přesný a podrobný popis systému. Pokud se používá přístup založený na modelu, mohou mít výstupy zpravidla podobu diagramů. Jestliže se uplatňují agilní metody vývoje, výstupy procesu návrhu nemusí být samostatné dokumenty specifikace, ale mohou mít formu poznámek v kódu programu.

Strukturované metody návrhu vznikaly v 70. a 80. letech a předcházely použití jazyka UML a objektově orientovaného návrhu (Budgen, 2003). Spoléhají se na tvorbu grafických modelů systému a v mnoha případech na automatické generování kódu z těchto modelů. Evoluci strukturovaných metod představuje vývoj založený na modelech (MDD – model-driven development) nebo inženýring založený na modelech (Schmidt, 2006), kde vznikají modely softwaru na různých úrovních abstrakce. Při vývoji založeném na modelech se klade větší důraz na modely architektury, kdy se rozlišuje mezi abstraktními modely nezávislými na implementaci a modely specifickými pro konkrétní implementaci. Modely se vyvíjejí s dostatečnou úrovní detailů, aby z nich bylo možné generovat spustitelný systém. Tímto přístupem k vývoji se budeme zabývat v kapitole 5.

Vývoj programu implementujícího systém přirozeně vyplývá z procesů návrhu systému. Některé třídy programů, jako například bezpečnostně kritické systémy, se sice obvykle navrhují podrobně ještě před zahájením implementace, ale častěji dochází k tomu, že se pozdější fáze návrhu prolínají s vývojem programu. Nástroje na vývoj softwaru umožňují generovat z návrhu základ programu. Tato základní struktura obsahuje kód na definování a implementaci rozhraní a v mnoha případech vývojáři stačí, když doplní podrobnosti fungování jednotlivých programových komponent.

Programování je osobní aktivita a neexistuje obecně přijímaný postup, kterým by se programátoři museli řídit. Někteří programátoři začínají od komponent, kterým rozumějí, vyvíjejí je a poté přecházejí k méně známým komponentám. Jiní volí opačný přístup a ponechávají si známé komponenty na konec, protože vědí, jak je vyvinout. Někteří vývojáři dávají přednost definování dat v prvních fázích procesu a z této definice vycházejí při dalším vývoji programu. Jiní ponechávají data nespecifikovaná tak dlouho, jak je to možné.



Obrázek 2.6 – Fáze testování

Programátoři obvykle provádějí určité testování kódu, který vyvinuli. Často se přitom objevují vady, které je nutné z programu odstranit. Tato aktivita se označuje jako ladění. Testování vad a ladění jsou odlišné procesy. Při testování se zjišťuje výskyt vad. Ladění se zabývá vyhledáním a opravou těchto vad.

Při ladění je potřeba vytvářet hypotézy o pozorovaném chování programu a poté tyto hypotézy testovat s nadějí, že bude nalezena chyba, která je příčinou abnormálního výstupu. Testování hypotéz může

vyžadovat ruční trasování kódu programu. Může být nutné vytvářet nové testovací případy, které problém lokalizují. Při procesu ladění mohou pomoci interaktivní ladicí nástroje, které zobrazují průběžné hodnoty programových proměnných a trasují prováděné příkazy.

2.2.3 Validace softwaru

Validace softwaru nebo obecněji řečeno verifikace a validace (V&V) má ukázat, že systém vyhovuje své specifikaci a zároveň splňuje očekávání svých zákazníků. Jako základní technika validace slouží testování programu, kdy se systém spouští se simulovanými testovacími daty. Validace může také zahrnovat kontrolní procesy, jako například inspekce a revize v každé fázi softwarového procesu od definice uživatelských požadavků až po vývoj programu. Vzhledem k dominantní roli testování je většina nákladů na validaci generována během implementace a po ní.

S výjimkou malých programů by se systémy neměly testovat jako jediná monolitická jednotka. Obrázek 2.6 znázorňuje třífázový proces testování, kdy se testují systémové komponenty, poté integrovaný systém a nakonec se systém testuje se zákaznickými daty. V ideálním případě se již v raných fázích tohoto procesu zjistí vady komponent a po integraci systému se odhalí problémy s rozhraními. Po nalezení defektů je však nutné program ladit a to může vyžadovat opakování předchozích fází procesu testování. Chyby v komponentách programu se například mohou objevit teprve při testování systému. Jedná se tedy o iterativní proces, kde se informace z pozdějších fází předávají zpět předchozím fázím.

Proces testování má tyto fáze:

1. *Vývojové testování* – komponenty, které tvoří systém, jsou testovány vývojáři systému. Každá komponenta se testuje nezávisle bez jiných systémových komponent. Komponenty mohou být jednoduché entity jako funkce nebo třídy objektů nebo se může jednat o koherentní skupiny těchto entit. Běžně se používají nástroje na automatizaci testování, jako je například JUnit (Massol a Husted, 2003), které mohou opakovaně spouštět testy komponent po vytvoření jejich nových verzí.
2. *Testování systému* – systémové komponenty se integrují za vzniku kompletního systému. Cílem tohoto procesu je najít chyby, které jsou způsobeny nepředpokládanými interakcemi mezi komponentami, a problémy s rozhraním komponent. Zároveň se zjišťuje, zda systém splňuje své funkční a mimofunkční požadavky, a testují se nově objevené vlastnosti systému. U velkých systémů se může jednat o vícefázový proces, kde se komponenty integrují do podsystémů, které se testují jednotlivě a poté jsou samy integrovány do výsledného systému.
3. *Předávací testování* – jde o závěrečnou fázi procesu testování, než je systém předán k používání v reálném provozu. Místo simulovaných testovacích dat se systém testuje s daty, která dodal jeho zákazník. Předávací testování může odhalit chyby a opomenutí v definici požadavků na systém, protože reálná data zkoumají systém jiným způsobem než testovací data. Předávací testování může také objevit problémy požadavků, kdy funkce systému v praxi nevyhovují potřebám uživatelů nebo systém poskytuje nedostatečný výkon.

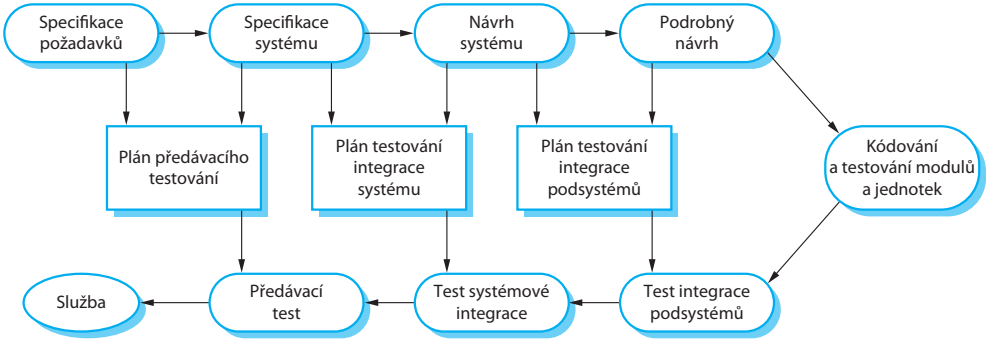
Procesy vývoje a testování komponent se obvykle prokládají. Programátoři sami sestavují testovací data a inkrementálně kód testují již během vývoje. Jedná se o ekonomicky rozumný přístup, protože programátoři znají jednotlivé komponenty a mohou tedy optimálně generovat testovací případy.

Jestliže se používá inkrementální přístup k vývoji, měl by se každý inkrement testovat při svém vývoji a tyto testy by měly být založeny na požadavcích na daný inkrement. Při extrémním programování se testy vyvíjejí spolu s požadavky před započetím vývoje. Díky tomu mohou testeři i vývojáři lépe porozumět požadavkům a je zajištěno, že kvůli vytváření testovacích případů nevzniknou žádná zpoždění.

Jestliže se používá plánovaný softwarový proces (např. u vývoje kritických systémů), testování je založeno na sadě testovacích plánů. Na těchto předem formulovaných testovacích plánech, které byly vyvinuty ze specifikace a návrhu systému, pracuje nezávislý tým testerů. Obrázek 2.7 dokumentuje, jak testovací plány propojují aktivity testování a vývoje. Tento model vývoje se někdy označuje jako V-model (písmeno V připomíná po otočení na bok).

Předávací testování se někdy označuje jako „alfa testování". Zákaznické systémy se vyvíjejí pro jediného klienta. Proces alfa testování pokračuje, dokud se vývojáři systému a klient neshodnou, že dodávaný systém představuje přijatelnou implementaci požadavků.

U systémů, které se dodávají na trh, se často používá proces označovaný jako „beta testování". Při beta testování se systém poskytuje více potenciálním zákazníkům, kteří jsou ochotni jej používat. Tito zákazníci pak oznamují zjištěné problémy vývojářům systému. Produkt je tak vystaven reálnému používání a lze najít chyby, které by vývojáři systému nedokázali odhadnout. Na základě této zpětné vazby dochází k úpravám systému, který je poté uvolněn k dalšímu beta testování nebo do volného prodeje.



Obrázek 2.7 – Fáze testování v plánovaném softwarovém procesu

2.2.4 Evoluce softwaru

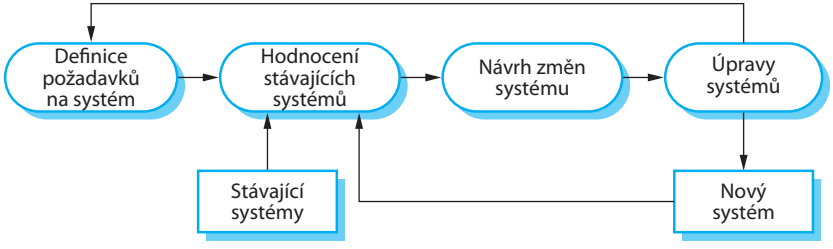
Flexibilita softwarových systémů patří mezi hlavní důvody, proč se stále více programů stává součástí větších a komplexních systémů. Jakmile se začne vyrábět určitý hardware, je změna jeho návrhu velmi drahá. Oproti tomu software lze měnit v libovolné fázi vývoje systému nebo později. Dokonce i rozsáhlé změny jsou pořád mnohem levnější než odpovídající změny systémového hardwaru.

Historicky vždy existovaly rozdíly mezi procesy vývoje softwaru a procesy evoluce softwaru (jeho údržby). Lidé si představují, že vývoj softwaru je kreativní činnosti, kdy softwarový systém směřuje od počáteční koncepce až k fungujícímu systému. Údržba softwaru se však často považuje za nudnou a nezajímavou činnost. I když náklady na údržbu často několikrát převyšují původní vývojové náklady, procesy údržby se často pokládají za méně náročné než počáteční vývoj softwaru.

Dělicí čára mezi vývojem a údržbou je však stále méně relevantní. Málokterý softwarový systém je zcela nový, a proto je rozumnější pohlížet na vývoj a údržbu jako na kontinuum. Místo dvou samostatných procesů je realističtější pojímat softwarové inženýrství jako evoluční proces (viz obrázek 2.8), kde se software během svého životního cyklu neustále mění v reakci na proměnlivé požadavky a potřeby zákazníků.

2.3 Zvládání změn

Změnám se v žádném velkém softwarovém projektu nelze vyhnout. Požadavky na systém se mění, jak podnik, který si systém objednal, reaguje na externí tlaky a mění se priority jeho managementu. S dostupností nových technologií se objevují nové možnosti návrhu a implementace. Bez ohledu na použitý model softwarových procesů je tedy zásadně důležité, aby model dokázal pracovat se změnami vyvíjeného softwaru.



Obrázek 2.8 – Evoluce systému

Změny zvyšují náklady na vývoj softwaru, protože obvykle znamenají, že je nutné znovu provést práci, která již byla dokončena. Označuje se to jako přepracování. Pokud například proběhne analýza vztahů mezi požadavky na systém a následně jsou identifikovány nové požadavky, je nutné analýzu požadavků částečně nebo kompletně opakovat. Poté může být nutné změnit návrh systému tak, aby poskytoval nové požadavky, změnit programy, které již byly vyvinuty, a znovu systém otestovat.

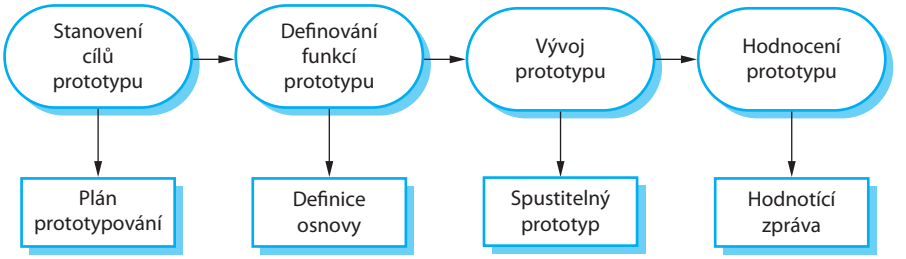
Při snižování nákladů na přepracování lze využít dva související přístupy:

1. Prevenci změn, kdy softwarový proces zahrnuje aktivity, které mohou předpokládat možné změny dříve, než je nutné zásadnější přepracování. Lze například vyvinout prototyp systému, který zákazníkům předvede některé jeho klíčové funkce. Zákazníci mohou s prototypem experimentovat a upřesňovat své požadavky dříve, než se zavážou uhradit vysoké náklady na produkci softwaru.
2. Toleranci změn, kdy je proces navržen tak, aby bylo možné změny zahrnout s relativně nízkými náklady. Obvykle zahrnuje nějakou formu inkrementálního vývoje. Navržené změny lze implementovat v inkrementech, které zatím nebyly vyvinuty. Jestliže to není možné, pak může k zahrnutí změny postačovat změna jediného inkrementu (malé části systému).

V této části budeme diskutovat dva způsoby zvládání změn a změny požadavků na systém. Jedná se o:

1. Prototypování systému, kdy je rychle vyvinuta verze systému nebo jeho část, která umožní testovat požadavky zákazníka a proveditelnost některých voleb návrhu. Tento způsob je v souladu s prevencí změn, protože uživatelé mohou experimentovat se systémem před jeho dodáním a díky tomu mohou upřesňovat své požadavky. Díky tomu je obvykle možné snížit počet žádostí o změnu požadavků po dodání systému.
2. Inkrementální dodávání, kdy zákazník dostává inkrementální verze systému, které může komentovat a zkoušet. Tento způsob podporuje jak prevenci, tak toleranci změn. Zabraňuje předčasnému potvrzení požadavků na celý systém a dovoluje relativně levně integrovat změny do pozdějších inkrementů.

Dalším důležitým mechanismem, který pomáhá při toleranci změn, je koncepce refaktoringu, konkrétně zlepšování struktury a organizace programu. Tímto aspektem se budeme zabývat v kapitole 3, která se týká agilních metod.



Obrázek 2.9 – Proces vývoje prototypu

2.3.1 Prototypování

Prototyp je raná verze softwarového systému, která umožňuje demonstrovat principy, vyzkoušet možnosti návrhu a zjistit více informací o problému a jeho možných řešeních. Prototyp je potřeba vyvíjet rychle a iterativně, aby se omezily náklady na vývoj a osoby zainteresované na systému mohly s prototypem experimentovat již na počátku softwarového procesu.

Softwarový prototyp může v procesu vývoje softwaru pomoci odhadovat změny, které mohou zákazníci požadovat:

1. V procesu inženýringu požadavků může prototyp usnadnit zjišťování a validaci systémových požadavků.
2. V procesu návrhu systému lze pomocí prototypu zkoumat konkrétní softwarová řešení a podpořit návrh uživatelského rozhraní.

Systémové prototypy umožňují uživatelům zjistit, nakolik jim systém dokáže pomoci při jejich práci. Mohou dostat nápady na nové funkce a určit silné a slabé stránky vyvíjeného softwaru. Mohou poté navrhnout nové požadavky na systém. Vyvíjený prototyp může navíc odhalit chyby a opomenutí v již navržených požadavcích. U funkce popsané ve specifikaci se může zdát, že je užitečná a dostatečně definovaná. Když se však stejná funkce zkombinuje s jinými funkcemi, uživatelé často zjistí, že jejich počáteční představa byla nesprávná či neúplná. Poté lze systémovou specifikaci upravit tak, aby odrážela změny v tom, jak uživatelé svým požadavkům rozumějí.

Systémový prototyp použitý při návrhu systému může sloužit k návrhovým experimentům, které ověří proveditelnost navrženého návrhu. Lze například prototypovat návrh databáze a vyzkoušet, zda podporuje efektivní přístup k datům u nejběžnějších uživatelských dotazů. Prototypování je také klíčovou součástí procesu návrhu uživatelského rozhraní. Vzhledem k dynamické povaze uživatelských rozhraní nestačí k vyjádření požadavků na uživatelské rozhraní jen textové popisy a diagramy. Jediný rozumný postup při vývoji grafických uživatelských rozhraní softwarových systémů tedy představuje rychlé prototypování se zapojením koncových uživatelů.

Model procesu pro vývoj prototypu je znázorněn na obrázku 2.9. Cíle prototypování by měly být zřejmé již na začátku celého procesu. Může se jednat o vývoj systému na prototypování uživatelského rozhraní, vývoj systému k ověření funkčních požadavků na systém nebo vývoj systému k demonstraci proveditelnosti aplikace pro manažery. Jediný prototyp nemůže splňovat všechny cíle. Pokud cíle nejsou stanoveny, nemusí vedoucí pracovníci či koncoví uživatelé funkci prototypu porozumět správně. V důsledku toho nemusí vývoj prototypu poskytnout očekávané výhody.

V další fázi procesu je potřeba zvolit, co bude systém prototypu obsahovat. Možná ještě důležitější rozhodnutí se týká toho, co v tomto systému vynechat. Kvůli omezení nákladů na prototypování a zkrácení

plánu dodávek je možné některé funkce prototypu vypustit, Můžeme se rozhodnout, že prototyp nebude přísně dodržovat mimofunkční požadavky typu doby odezvy a využití paměti. Lze ignorovat obsluhu a správu chyb, pokud není cílem prototypu tvorba uživatelského rozhraní. Je přípustné snížit standardy spolehlivosti a kvality programu.

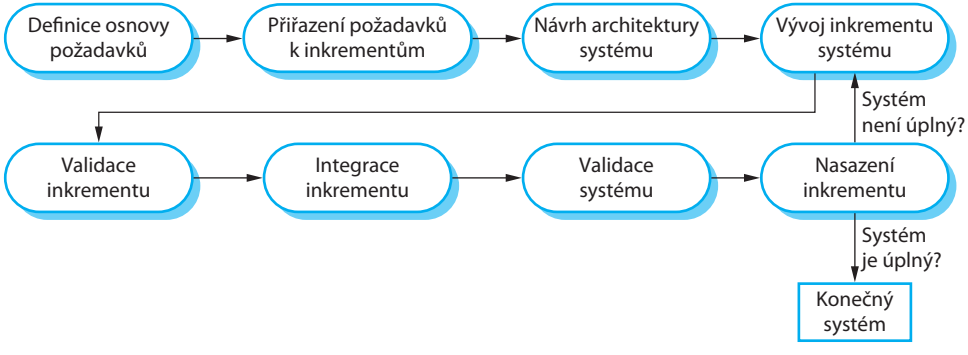
Závěrečnou fází procesu je hodnocení prototypu. V této fázi je nutné zajistit uživatelské školení a na základě cílů prototypu by měl být odvozen plán hodnocení. Uživatelé potřebují určitý čas, aby se s novým systémem obeznámili a dostali se do běžného rytmu používání. Jakmile začnou systém normálně používat, mohou objevit chyby v požadavcích a chybějící požadavky.

Obecný problém s prototypováním spočívá v tom, že prototyp se nutně nemusí používat stejným způsobem jako konečný systém. Tester prototypu nemusí patřit mezi typické uživatele systému. Někdy není při hodnocení prototypu dostatek času na školení. Pokud je prototyp pomalý, mohou hodnotící uživatelé změnit způsob své práce a vyhýbat se těm systémovým funkcím, které mají dlouhou dobu odezvy. Jestliže výsledný systém reaguje rychleji, mohou s ním pracovat odlišným způsobem.

Manažeři někdy vyvíjejí na vývojáře tlak, aby dodali prototyp na jedno použití, zejména v situacích, kdy dochází ke zpoždění v dodávce výsledné verze softwaru. Tento postup však obvykle není rozumný:

1. Někdy není možné vyladit prototyp tak, aby vyhověl mimofunkčním požadavkům, například na výkon, zabezpečení, robustnost a spolehlivost, které se při vývoji prototypu ignorovaly.
2. Rychlé změny při vývoji nevyhnutelně vedou k tomu, že prototyp není dokumentován. Jedinou specifikací návrhu je kód prototypu. To není z hlediska dlouhodobé údržby vhodné.
3. Změny provedené při vývoji prototypu s velkou pravděpodobností zhoršují strukturu systému. Údržba systému bude obtížná a nákladná.
4. Při vývoji prototypů se obvykle uvolňují organizační standardy kvality.

Prototypy mohou být užitečné, i když je nelze spustit. Papírové modely uživatelského rozhraní systému (Rettig, 1994) mohou efektivně pomoci uživatelům, aby vyladili návrh rozhraní a prošli jednotlivé scénáře použití. Vývoj těchto modelů je velmi levný a lze je sestavit během několika dní. Rozšířením této metody je prototyp „Čaroděje ze země Oz“, kde se vyvíjí pouze uživatelské rozhraní. Uživatelé interagují s tímto rozhraním, ale jejich požadavky jsou předávány osobě, která je interpretuje a poskytuje vhodnou reakci.



Obrázek 2.10 – Inkrementální dodávání

2.3.2 Inkrementální dodávání

Inkrementální dodávání (obrázek 2.10) je přístup k vývoji softwaru, kdy se některé z vyvinutých inkrementů dodávají zákazníkovi a nasazují se k používání v provozním prostředí. V procesu inkrementálního dodávání mohou zákazníci identifikovat osnovu služeb, které bude systém poskytovat. Zákazníci určují, které služby jsou pro ně nejdůležitější a které nejméně důležité. Poté je definováno několik dodávaných inkrementů, z nichž každý poskytuje dílčí sadu systémových funkcí. Přidělování služeb inkrementům závisí na prioritě služeb, přičemž služby s nejvyšší prioritou se implementují a dodávají jako první.

Po identifikaci inkrementů systému lze podrobně definovat požadavky na služby obsažené v prvním inkrementu a zahájit vývoj tohoto inkrementu. Během vývoje lze provádět další analýzu požadavků pro pozdější inkrementy, ale změny požadavků na aktuální inkrement se již nepřijímají.

Jakmile je inkrement dokončen a dodán, mohou jej zákazníci nasadit do provozu. To znamená, že mohou již brzy začít využívat část systémových funkcí. Mohou se systémem experimentovat a díky tomu si mohou ujasnit své požadavky na pozdější inkrementy systému. Po svém dokončení jsou následné inkrementy integrovány se stávajícími, takže se s každým dodaným inkrementem funkce systému postupně rozšiřují.

Inkrementální dodávání poskytuje několik výhod:

1. Zákazníci mohou používat rané inkrementy jako prototypy a získat zkušenosti, které jim umožní informovaně zvolit požadavky na pozdější inkrementy systému. Oproti prototypům se jedná o součást reálného systému, takže po zpřístupnění úplného systému není nutné, aby se jej uživatelé učili používat znovu.
2. Zákazníci nemusí čekat na dodání celého systému, ale mohou některé jeho funkce využívat již dříve. První inkrement uspokojuje nejkritičtější požadavky, aby mohli zákazníci software ihned nasadit.
3. Proces zachovává výhody inkrementálního dodávání v tom, že by mělo být relativně snadné do systému zavádět změny.
4. Když se služby s nejvyšší prioritou dodávají jako první a poté se integrují další inkrementy, nejdůležitější systémové služby procházejí nejpodrobnějším testováním. Díky tomu se snižuje pravděpodobnost, že zákazníci najdou chyby softwaru v nejdůležitějších částech systému.

Inkrementální dodávání je však spojeno s jistými problémy:

1. Většina systémů vyžaduje sadu základních funkcí, které se používají v různých částech systému. Když nejsou požadavky definovány podrobně, než je implementován příslušný inkrement, může být obtížné identifikovat společné funkce, které jsou potřebné pro všechny inkrementy.
2. Iterativní vývoj se také může zkomplikovat v situacích, kdy probíhá vývoj náhradního systému. Uživatelé požadují všechny funkce starého systému a často nejsou ochotni s neúplným novým systémem experimentovat. Kvůli tomu je těžké od nich získat užitečnou zpětnou vazbu.
3. Podstata iterativních procesů spočívá v tom, že se specifikace vyvíjí spolu se softwarem. To je však v rozporu s modelem, na jehož základě mnoho organizací objednává zboží a služby. V tomto modelu je úplná specifikace systému součástí smlouvy na jeho dodání. V případě inkrementálního přístupu neexistuje žádná úplná specifikace systému, dokud není specifikován poslední inkrement. To vyžaduje nový typ smlouvy, který nemusí být přijatelný pro velké zákazníky, jako jsou vládní úřady.

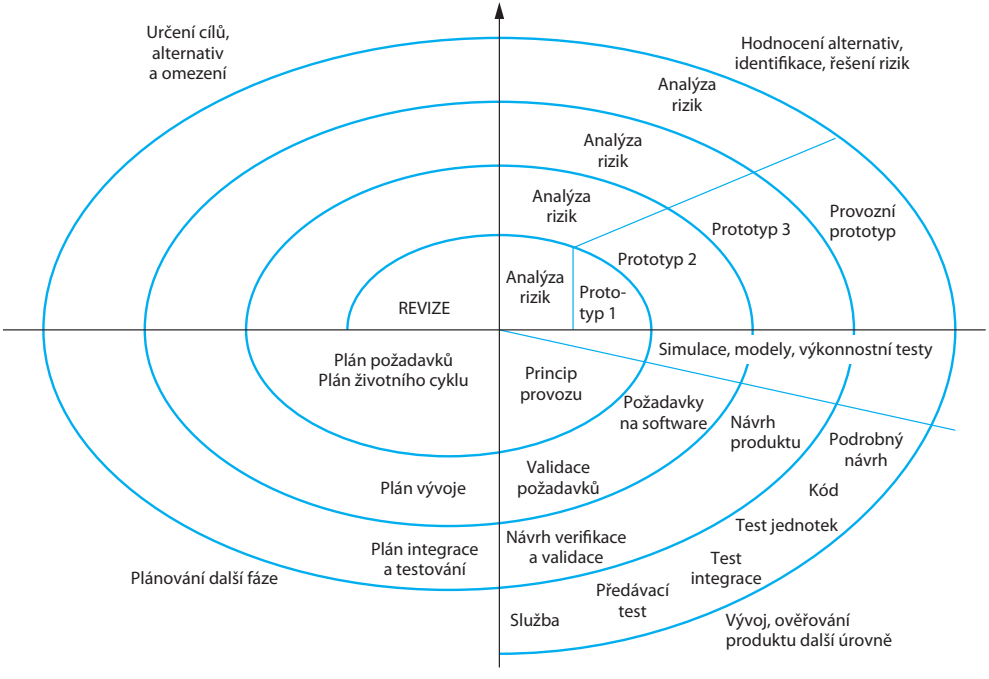
U některých typů systémů není přístup založený na inkrementálním vývoji a dodávání nejvhodnější. Jedná se o velmi rozsáhlé systémy, které může vyvíjet více týmů pracujících v různých lokalitách, některé

integrované systémy, kde software závisí na vývoji hardwaru, a některé kritické systémy, kde je nutné analyzovat všechny požadavky, aby bylo možné zkontrolovat, zda některé interakce nenarušují bezpečnost či zabezpečení systému.

Tyto systémy se samozřejmě musí vypořádat se stejným problémem nejasných a proměnlivých požadavků. Aby tedy bylo možné tyto problémy vyřešit a získat určité výhody inkrementálního vývoje, lze použít proces, kdy se iterativně vyvíjí prototyp systému, který slouží jako platforma pro experimenty s požadavky na systém a jeho návrhem. Díky zkušenostem získaným u prototypu je poté možné schválit definitivní požadavky.

2.3.3 Boehmův spirálový model

Boehm (1988) navrhl architekturu softwarového procesu založenou na riziku (spirálový model). Tento model je znázorněn na obrázku 2.11. Softwarový proces je zde reprezentován pomocí spirály, nikoli jako posloupnost aktivit s určitými zpětnými posuny mezi jednotlivými aktivitami. Každá smyčka spirály představuje jednu fázi softwarového procesu. Nejvnitřnější smyčka se tedy může týkat studie proveditelnosti systému, další smyčka definice požadavků, následující smyčka návrhu systému atd. Spirálový model kombinuje prevenci změn s tolerancí změn. Předpokládá, že změny jsou výsledkem rizik projektu, a zahrnuje explicitní aktivity řízení rizik, které mají tato rizika omezovat.



Obrázek 2.11 – Boehmův spirálový model softwarového procesu (©IEEE 1988)

Každá smyčka ve spirále se dělí na čtyři sektory:

1. *Nastavení cílů* – jsou definovány konkrétní cíle pro danou fázi projektu. Identifikují se omezení procesu a produktu a vytváří se podrobný plán řízení. Jsou určena rizika projektu. Případně se mohou v závislosti na těchto rizicích naplánovat alternativní strategie.

2. *Hodnocení a omezení rizik* – pro každé z identifikovaných rizik projektu se provádí podrobná analýza. Přijímají se kroky na omezení rizika. Pokud například existuje riziko, že požadavky neodpovídají možnostem, lze vyvinout prototyp systému.
3. *Vývoj a validace* – po hodnocení rizik dochází k volbě modelu vývoje systému. Jestliže například dominují rizika související s uživatelským rozhraním, může být nejvhodnějším přístupem k vývoji tvorba prototypů na jedno použití. Pokud je kladen důraz na bezpečnostní rizika, může nejlépe vyhovovat proces vývoje založený na formálních transformacích. V případě, že hlavní identifikované riziko spočívá v integraci podsystémů, může být pro vývoj nejvhodnější vodopádový model.
4. *Plánování* – projekt se reviduje a přijímá se rozhodnutí, zda pokračovat další smyčkou spirály. Pokud padne rozhodnutí o pokračování, jsou sestaveny plány na další fázi projektu.

Hlavní rozdíl spirálního modelu a jiných modelů softwarových procesů leží v explicitním rozpoznání rizik. Cyklus spirály začíná rozborem cílů, například z hlediska výkonu a funkcí. Poté se sepisují alternativní způsoby, jak těchto cílů dosáhnout a jak zvládat omezení jednotlivých postupů. Každá alternativa se hodnotí vzhledem k příslušnému cíli a identifikují se zdroje projektových rizik. Dalším krokem je řešení těchto rizik díky aktivitám shromažďování informací, jako je podrobnější analýza, prototypování a simulace.

Po vyhodnocení rizik proběhne první část vývoje, po které následuje aktivita plánování další fáze procesu. Neformálně se pojmem riziko označuje cokoli, co se může pokazit. Pokud například existují plány na použití nového programovacího jazyka, riziko spočívá v tom, že dostupné kompilátory nebudou spolehlivé nebo neposkytnou dostatečně efektivní objektový kód. Rizika vedou k navrženým změnám softwaru a problémům s projektem, jako jsou zpoždění oproti plánu a překročení rozpočtu. Minimalizace rizik je proto velmi důležitou aktivitou při řízení projektu. Řízením rizik, které představuje klíčovou složku řízení projektů, se budeme zabývat v kapitole 22.

2.4 Proces RUP

Proces RUP (Rational Unified Process – Krutchen, 2003) je příkladem moderního modelu procesů. Odvozuje se z práce na jazyce UML a souvisejícím procesu Unified Software Development Process (Rumbaugh, et al., 1999; Arlow a Neustadt, 2005). Jeho popis zahrnujeme do této kapitoly, protože se jedná o dobrý příklad hybridního modelu procesů. Kombinuje elementy ze všech obecných modelů procesů (část 2.1), ilustruje optimální postupy při specifikaci a návrhu (část 2.2) a podporuje prototypování a inkrementální dodávání (část 2.3).

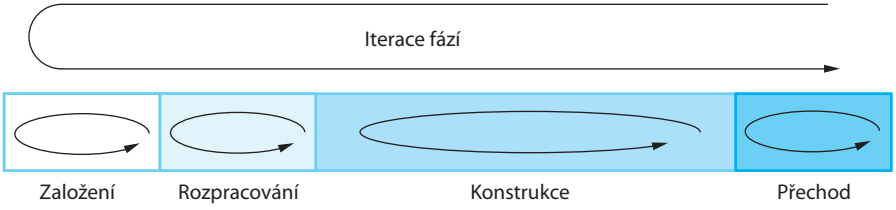
Proces RUP vychází z toho, že konvenční modely procesů představují jediný pohled na celý proces. Oproti tomu proces RUP se obvykle popisuje ze třech perspektiv:

1. Dynamická perspektiva, která znázorňuje fáze modelu v průběhu času.
2. Statická perspektiva, která zobrazuje odehrávající se aktivity procesu.
3. Perspektiva postupů, která doporučuje optimální postupy pro nasazení v procesu.

Většina popisů procesu RUP se pokouší kombinovat statickou a dynamickou perspektivu do jediného diagramu (Krutchen, 2003). Podle názoru autora se tím komplikuje porozumění celému procesu, takže v této knize budeme používat samostatné popisy každé z uvedených perspektiv.

RUP je fázový model, který identifikuje čtyři oddělené fáze softwarového procesu. Oproti vodopádovému modelu, kde fáze odpovídají aktivitám procesu, však fáze procesu RUP úžeji souvisejí spíše s podnikovými než technickými hledisky. Fáze procesu RUP jsou znázorněny na obrázku 2.11. Jedná se o:

1. *Založení* – cílem fáze založení je vytvořit podnikový případ příslušného systému. Je potřeba identifikovat všechny externí entity (osoby a systémy), které budou se systémem interagovat, a definovat tyto interakce. Na základě těchto informací lze potom vyhodnotit příspěvek systému k fungování podniku. Pokud tento příspěvek není významný, může být projekt po této fázi zrušen.



Obrázek 2.12 – Fáze procesu RUP

2. *Rozpracování* – fáze rozpracování by měla rozvinout znalosti problémové domény, vytvořit schéma architektury systému, vyvinout projektový plán a identifikovat klíčová projektová rizika. Po dokončení této fáze by měl existovat model požadavků na systém, což může být sada případů použití UML, popis architektury a plán vývoje softwaru.
3. *Konstrukce* – fáze konstrukce zahrnuje návrh systému, programování a testování. V této fázi se paralelně vyvíjejí a integrují části systému. Na konci této fáze by měl být k dispozici funkční softwarový systém a související dokumentace v podobě, která umožňuje dodání uživatelům.
4. *Přechod* – závěrečná fáze procesu RUP řeší přenesení systému od vývojové komunity k uživatelské komunitě a zprovoznění systému v reálném prostředí. Tento aspekt většina modelů softwarových procesů ignoruje, ale v praxi se jedná o nákladnou a často i problematickou aktivitu. Po dokončení této fáze bychom měli mít dokumentovaný softwarový systém, který správně funguje ve svém provozním prostředí.

Proces RUP podporuje iteraci dvěma způsoby. Každou fázi lze provést iterativním způsobem, kdy se výsledky vyvíjejí inkrementálně. Kromě toho je možné inkrementálně realizovat i celou sadu fází, jak je patrné z vracející se šipky od fáze přechodu k fázi založení na obrázku 2.12.

Statický pohled na proces RUP se zaměřuje na aktivity, které probíhají při procesu vývoje. V terminologii procesu RUP se jedná o pracovní toky. Proces identifikuje šest klíčových pracovních toků procesu a tři klíčové podpůrné pracovní toky. Proces RUP byl navržen s ohledem na jazyk UML, takže je popis pracovního toku orientován na přidružené modely UML, jako jsou sekvenční modely, objektové modely atd. Klíčové technické a podpůrné pracovní toky jsou znázorněny na obrázku 2.13.

Výhoda prezentace dynamických a statických pohledů spočívá v tom, že fáze vývojového procesu nejsou přiřazeny ke konkrétním pracovním tokům. Alespoň teoreticky mohou být všechny pracovní toky procesu RUP aktivní ve všech fázích celého procesu. V počátečních fázích procesu bude většina úsilí pravděpodobně vynaložena na pracovní toky typu obchodního modelování a požadavků a v pozdějších fázích na testování a nasazení.

Pracovní tok	Popis
Obchodní modelování	Obchodní procesy se modelují podle obchodních případů použití.
Požadavky	Jsou identifikováni aktéři, kteří se systémem interagují, a vyvíjejí se případy použití na modelování systémových požadavků.
Analýza a návrh	Při tvorbě a dokumentaci modelu návrhu se používají modely architektury, modely komponent, modely objektů a sekvenční modely.
Implementace	Komponenty v systému jsou implementovány a strukturovány do implementace podsystémů. Tento proces lze urychlit automatickým generováním kódu z modelů návrhu.
Testování	Testování je iterativní proces, který probíhá spolu s implementací. Po dokončení implementace následuje testování systému.
Nasazení	Je vytvořeno vydání produktu, distribuuje se uživatelům a instaluje na jejich pracovišti.
Správa konfigurace a změn	Tento podpůrný pracovní tok řídí změny systému (viz kapitola 25).
Řízení projektu	Tento podpůrný pracovní tok řídí vývoj systému (viz kapitoly 22 a 23).
Prostředí	Tento pracovní tok zpřístupňuje vývojovému týmu softwaru příslušné softwarové nástroje.

Obrázek 2.13 – Statické pracovní toky v procesu RUP

Perspektiva postupů na proces RUP popisuje optimální postupy softwarového inženýrství, které se při vývoji systémů doporučují. Doporučuje se šest základních optimálních postupů:

1. *Iterativní vývoj softwaru* – inkrementy systému je vhodné plánovat na základě priorit zákazníka a vyvíjet funkce systému s nejvyšší prioritou na začátku procesu vývoje.
2. *Správa požadavků* – doporučuje se explicitně dokumentovat požadavky zákazníků a sledovat jejich změny. Před přijetím požadavků na změny systému je vhodné analyzovat jejich dopad.
3. *Použití architektury založené na komponentách* – architekturu systému je vhodné strukturovat do komponent, jak jsme rozebrali v předchozí části této kapitoly.
4. *Vizuální modelování softwaru* – k prezentování statických a dynamických pohledů na software se hodí grafické modely UML.
5. *Verifikace kvality softwaru* – je nutné zkontrolovat, zda software splňuje podnikové standardy kvality.
6. *Správa změn softwaru* – změny softwaru je vhodné spravovat pomocí systému na správu změn a pomocí postupů a nástrojů správy konfigurace.

Proces RUP se nehodí pro všechny typy vývoje, například pro vývoj integrovaného softwaru. Reprezentuje však přístup, který potenciálně kombinuje tři obecné modely procesů, kterými jsme se zabývali v části 2.1. Nejdůležitější inovace procesu RUP spočívá v oddělení fází a pracovních toků a v uvědomění si toho, že součástí procesu je nasazení softwaru v uživatelském prostředí. Fáze jsou dynamické a mají své cíle. Pracovní toky jsou statické a jedná se o technické aktivity, které nejsou pevně spjaty s jedinou fází, ale lze je použít v rámci celého vývoje, aby bylo možné dosáhnout cílů jednotlivých fází.

HLAVNÍ BODY

- Softwarové procesy jsou aktivity, které se podílejí na produkci softwarového systému. Modely softwarových procesů představují abstraktní reprezentace těchto procesů.
- Obecné modely procesů popisují organizaci softwarových procesů. K příkladům těchto obecných modelů patří vodopádový model, inkrementální vývoj a vývoj orientovaný na opakované použití.
- Inženýring požadavků je proces vývoje specifikace softwaru. Účelem specifikací je sdělovat požadavky zákazníků na systém vývojářům systému.
- Procesy návrhu a implementace se týkají převodu specifikace požadavků do spustitelného softwarového systému. V rámci této transformace lze použít systematické metody návrhu.
- Validace softwaru je proces kontroly, zda je systém v souladu se svou specifikací a zda splňuje reálné požadavky svých uživatelů.
- Evoluce softwaru probíhá, když se existující softwarové systémy mění tak, aby vyhověly novým požadavkům. Změny jsou průběžné, a aby zůstal software užitečný, musí se vyvíjet.
- Procesy by měly zahrnovat aktivity na zvládání změn. Může se jednat o fázi prototypování, která pomáhá předcházet chybným rozhodnutím ohledně požadavků a návrhu. Procesy lze strukturovat s ohledem na iterativní vývoj a dodávání, aby bylo možné provádět změny bez narušení systému jako celku.
- Proces RUP je moderní obecný model procesů, který je uspořádán do fází (založení, rozpracování, konstrukce a přechod), ale odděluje aktivity (požadavky, analýzu, návrh atd.) od těchto fází.

DALŠÍ ZDROJE INFORMACÍ

Managing Software Quality and Business Risk (Správa kvality softwaru a obchodních rizik). Tato kniha je primárně o správě softwaru, ale obsahuje vynikající kapitolu (kapitolu 4) o modelech procesů. (M. Ould, John Wiley and Sons Ltd, 1999.)

Process Models in Software Engineering (Modely procesů v softwarovém inženýrství). Jedná se o skvělý přehled široké škály modelů procesů softwarového inženýrství, které byly dosud navrženy. (W. Scacchi, *Encyclopaedia of Software Engineering* (Encyklopedie softwarového inženýrství), ed. J. J. Marciniak, John Wiley and Sons, 2001.) *http://www.ics.uci.edu/~wscacchi/ Papers/SE-Encyc/ Process-Models-SE-Encyc.pdf*.

The Rational Unified Process: An Introduction (3rd edition) – Proces RUP: úvod (3. vydání). V době psaní této knihy šlo o nejčtivější dostupnou knihu o procesu RUP. Krutchen velmi dobře popisuje celý proces, ale v knize poněkud chybí informace o praktických potížích při používání procesu. (P. Krutchen, Addison-Wesley, 2003.)

CVIČENÍ

2.1. Navrhněte nejvhodnější obecný model softwarových procesů, který by bylo možné použít jako základ pro správu vývoje následujících systémů, a uveďte důvody pro své odpovědi s ohledem na typ vyvíjeného systému:

Systém na řízení systému ABS v automobilu

Systém virtuální reality na podporu údržby softwaru

Univerzitní účetní systém, který má nahradit stávající systém

Interaktivní systém plánování cest, který uživatelům umožní plánovat dovolenou s nejmenším dopadem na životní prostředí

2.2. Vysvětlete, proč inkrementální vývoj představuje nejefektivnější přístup pro vývoj podnikových softwarových systémů. Proč je tento model méně vhodný při inženýringu systémů pracujících v reálném čase?

2.3. Zamyslete se nad procesem založeným na opakovaném použití, který je znázorněn na obrázku 2.3. Vysvětlete, proč tento proces musí obsahovat dvě samostatné aktivity inženýringu požadavků.

2.4. Odhadněte, proč je v procesu inženýringu požadavků důležité rozlišovat mezi vývojem uživatelských požadavků a vývojem systémových požadavků.

2.5. Popište hlavní aktivity procesu návrhu softwaru a výstupy těchto aktivit. Pomocí diagramu znázorněte možné vztahy mezi výstupy těchto aktivit.

2.6. Vysvětlete, proč se složité systémy musí měnit, a uveďte příklady (mimo prototypování a inkrementálního dodávání) aktivit softwarových procesů, které pomáhají předpovídat změny a zajišťují větší odolnost vyvíjeného softwaru ke změnám.

2.7. Vysvětlete, proč systémy vyvíjené jako prototypy obvykle není vhodné používat jako produkční systémy.

2.8. Vysvětlete, proč je Boehmův spirálový model přizpůsobitelným modelem, který dokáže zajistit aktivity prevence změn i tolerance změn. V praxi se tento model široce nepoužívá. Odhadněte, proč tomu tak může být.

2.9. Jaké jsou výhody poskytování statických a dynamických pohledů na softwarový proces jako v procesu RUP?

2.10. Zavádění nových technologií v minulosti mnohokrát způsobilo zásadní změny na trhu práce a alespoň dočasně připravovalo lidi o práci. Rozhodněte, zda zavádění rozsáhlé automatizace procesů bude mít pravděpodobně stejný dopad na softwarové inženýry. Pokud si myslíte, že nikoli, vysvětlete svou odpověď. Jestliže se domníváte, že možnosti pracovního uplatnění sníží, je etické, aby dotčení inženýři nasazení této technologie pasivně nebo aktivně odporovali?

CITACE

Arlow, J. a Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)* – UML 2 a proces RUP: praktická objektově orientovaná analýza a návrh (2. vydání). Boston: Addison-Wesley.

Boehm, B. a Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed* (Rovnováha mezi agilitou a disciplínou: průvodce pro bezradné). Boston: Addison-Wesley.

Boehm, B. W. (1988). „A Spiral Model of Software Development and Enhancement" (Spirálový model vývoje a zdokonalování softwaru). *IEEE Computer*, **21** (5), 61–72.

Budgen, D. (2003). *Software Design (2nd Edition)* – Návrh softwaru (2. vydání). Harlow, UK.: Addison-Wesley.

Krutchen, P. (2003). *The Rational Unified Process: An Introduction (3rd Edition)* – Proces RUP: úvod (3. vydání). Reading, MA: Addison-Wesley.

Massol, V. a Husted, T. (2003). *JUnit in Action* (JUnit v praxi). Greenwich, Conn.: Manning Publications Co.

Rettig, M. (1994). „Practical Programmer: Prototyping for Tiny Fingers" (Praktický programátor: prototypování pro malé prsty). *Comm. ACM*, **37** (4), 21–7.

Royce, W. W. (1970). „Managing the Development of Large Software Systems: Concepts and Techniques" (Správa vývoje velkých softwarových systémů: koncepce a techniky). IEEE WESTCON, Los Angeles CA: 1–9.

Rumbaugh, J., Jacobson, I. a Booch, G. (1999). *The Unified Software Development Process* (Proces Unified Software Development Process). Reading, Mass.: Addison-Wesley.

Schmidt, D. C. (2006). „Model-Driven Engineering" (Inženýrství založené na modelech). *IEEE Computer*, **39** (2), 25–31.

Schneider, S. (2001). *The B Method* (Metoda B). Houndmills, UK: Palgrave Macmillan.

Wordsworth, J. (1996). *Software Engineering with B* (Softwarové inženýrství a metoda B). Wokingham: Addison-Wesley.

Agilní vývoj softwaru

3

Cíle

Cílem této kapitoly je představit metody agilního vývoje softwaru. V této kapitole:

- seznámíte se s východisky agilních metod vývoje softwaru, manifestem agilního vývoje a rozdíly mezi agilním a plánovaným vývojem,
- poznáte klíčové postupy extrémního programování a dozvíte se, jak souvisejí s obecnými principy agilních metod,
- porozumíte přístupu Scrum k řízení agilních projektů,
- dozvíte se o komplikacích a problémech při škálování metod agilního vývoje na vývoj velkých softwarových systémů.

Podniky nyní fungují v globálním a rychle se měnícím prostředí. Musí reagovat na nové příležitosti a trhy, proměnlivé ekonomické podmínky a vznik konkurenčních produktů a služeb. Software tvoří součást téměř všech podnikových činností. Proto je nutné nový software vyvíjet rychle, aby mohly podniky využít nové příležitosti a přizpůsobit se konkurenčnímu tlaku. Mezi nejdůležitější požadavky na softwarové systémy proto nyní často patří rychlost vývoje a dodávání. V praxi je mnoho podniků ochotno slevit z kvality softwaru a přijímat kompromisy ohledně požadavků, aby mohly potřebný software nasadit rychleji.

Vzhledem k tomu, že tyto podniky fungují v proměnlivém prostředí, často prakticky není možné odvodit úplnou sadu stabilních požadavků na software. Počáteční požadavky se nevyhnutelně mění, protože zákazníci nedokážou předvídat, jak systém ovlivní pracovní postupy, jak bude interagovat s jinými systémy a které uživatelské operace by měl automatizovat. Důležité požadavky se vyjasní teprve poté, co je systém dodán a uživatelé získají zkušenosti s jeho používáním. I poté se požadavky pravděpodobně budou rychle a nepředvídatelně měnit vzhledem k externím faktorům. Software může být tedy zastaralý již v okamžiku svého dodání.

Procesy vývoje softwaru, které plánují kompletní specifikaci požadavků a následný návrh, budování a testování systému, se proto k rychlému vývoji nehodí. Spolu se změnou požadavků nebo zjištěním jejich problémů je nutné přepracovat a znovu testovat návrh či implementaci systému. V důsledku toho

se konvenční vodopádový proces nebo proces založený na specifikaci zpravidla protahuje a výsledný software je zákazníkovi dodán značně dlouho po sepsání první specifikace.

U některých typů softwaru, jako jsou bezpečnostně kritické řídicí systémy, kde je nezbytné systém kompletně analyzovat, představuje plánovaný přístup správnou cestu. V rychle se měnícím podnikovém prostředí však může způsobit značné problémy. Než lze software začít používat, může se počáteční důvod pro jeho pořízení někdy změnit natolik zásadně, že je software prakticky zbytečný. Zejména v případě podnikových systémů mají tedy klíčovou roli vývojové procesy, které se zaměřují na rychlý vývoj softwaru.

Potřeba rychlého vývoje systémů a procesů, které dokážou zvládat proměnlivé požadavky, se obecně uznává již delší dobu. Společnost IBM zavedla inkrementální vývoj v 80. letech (Mills et al., 1980). Také příchod programovacích jazyků takzvané čtvrté generace (rovněž v 80. letech) podpořil myšlenku rychlého vývoje a dodávání softwaru (Martin, 1981). Celá myšlenka se však skutečně prosadila koncem 90. let s vývojem koncepce agilních přístupů jako DSDM (Stapleton, 1997), Scrum (Schwaber a Beedle, 2001) a extrémního programování (Beck, 1999; Beck, 2000).

Procesy rychlého vývoje softwaru jako navrženy tak, aby dokázaly rychle vyprodukovat užitečný software. Software se nevyvíjí jako jediná jednotka, ale v řadě inkrementů, kdy každý inkrement obsahuje nové funkce systému. Existuje sice mnoho přístupů k rychlému vývoji softwaru, ale všechny sdílejí stejné základní vlastnosti:

1. Procesy specifikace, návrhu a implementace se prolínají. Neexistuje žádná podrobná specifikace systému a dokumentace návrhu se minimalizuje nebo generuje automaticky z programovacího prostředí, které se používá k implementaci systému. Dokument s uživatelskými požadavky definuje pouze nejdůležitější vlastnosti systému.
2. Systém se vyvíjí v řadě verzí. Na specifikaci a hodnocení každé verze se podílejí koncoví uživatelé a jiné osoby zainteresované na systému. Mohou navrhovat změny softwaru a nové požadavky, které lze implementovat v pozdější verzi systému.
3. Uživatelská rozhraní systému se často vyvíjejí formou interaktivního vývoje, který umožňuje rychle vytvářet návrh rozhraní kreslením a rozmisťováním ikon. Systém pak může generovat webové rozhraní pro prohlížeč nebo rozhraní pro konkrétní platformu, jako je Microsoft Windows.

Agilní metody patří mezi metody inkrementálního vývoje, kde jsou inkrementy malé a nová vydání systému se vytvářejí a zpřístupňují zákazníkům obvykle každé dva či tři týdny. Tyto metody zapojují zákazníky do rozhodovacího procesu, aby bylo možné získat rychlou zpětnou vazbu týkající se proměnlivých požadavků. Minimalizují dokumentaci díky tomu, že upřednostňují neformální komunikaci před formálními poradami s písemnými zápisy.

3.1 Agilní metody

V 80. a 90. letech se všeobecně věřilo, že nejlepší způsob tvorby kvalitního softwaru je založen na pečlivém plánování projektu, formalizované kontrole kvality, použití analytických a návrhových metod dostupných v nástrojích CASE a řízených a přesných procesech vývoje softwaru. Tento pohled pocházel od komunity softwarových inženýrů, kteří odpovídali za vývoj velkých a dlouhodobě používaných softwarových systémů (například v leteckých a vládních projektech).

Tento software vyvíjely velké týmy pracující pro různé společnosti. Týmy často bývaly rozptýleny po světě a pracovaly na softwaru dlouhou dobu. Příkladem tohoto typu softwaru jsou řídicí systémy moderního letadla, jejichž vývoj od počáteční specifikace k nasazení může trvat až 10 let. Tyto plánované přístupy jsou spojeny se značnou režií na plánování, návrh a dokumentaci systému. Tato režie je

oprávněná, pokud je nutné koordinovat práci více vývojových týmů, pokud se jedná o kritický systém a pokud se na údržbě softwaru během jeho životního cyklu bude podílet mnoho lidí.

Když se však tento komplikovaný plánovaný přístup aplikuje na malé a středně velké podnikové systémy, je související režie natolik velká, že celému procesu vývoje softwaru dominuje. Diskusemi o způsobu vývoje se stráví více času než samotným vývojem a testováním programu. Jak se mění požadavky na systém, je nutné práci předělat a alespoň z principu je spolu s programem nutné změnit i specifikaci a návrh.

Nespokojenost s těmito robustními přístupy k softwarovému inženýrství vedla mnoho softwarových vývojářů v 90. letech k tomu, aby navrhli nové „agilní metody". Tyto metody vývojovým týmům umožňují, aby se namísto návrhu a dokumentace zaměřily na samotný software. Agilní metody univerzálně spoléhají na inkrementální přístup ke specifikaci, vývoji a dodávání softwaru. Nejlépe se hodí k vývoji aplikací, kde se systémové požadavky během procesu vývoje rychle mění. Jejich cílem je rychle dodat fungující software zákazníkům, kteří pak mohou navrhnout nové a změněné požadavky, které budou zohledněny v pozdějších iteracích systému. Metody usilují o omezení byrokratického procesu. Odstraňují práci, která má pochybnou dlouhodobou hodnotu, a eliminují dokumentaci, která se pravděpodobně nikdy nebude používat.

Filozofie v pozadí agilních metod se odráží v jejich manifestu, na kterém se shodli mnozí významní vývojáři těchto metod. Tento manifest prohlašuje:

> *Objevujeme nové způsoby vývoje softwaru, které používáme sami a pomáháme je používat i jiným. Při své práci jsme se naučili upřednostňovat:*
>
> *jednotlivce a jejich interakce před procesy a nástroji,*
>
> *funkční software před úplnou dokumentací,*
>
> *spolupráci se zákazníky před dojednáváním smluv,*
>
> *reagování na změny před sledováním plánu.*
>
> *Ačkoli tedy uznáváme, že položky na pravé straně mají svou hodnotu, oceňujeme spíše položky vlevo.*

Pravděpodobně nejznámější agilní metodou je extrémní programování (Beck, 1999; Beck, 2000), které popíšeme v další části kapitoly. K dalším agilním přístupům patří Scrum (Cohn, 2009; Schwaber, 2004; Schwaber a Beedle, 2001), Crystal (Cockburn, 2001; Cockburn, 2004), adaptivní vývoj softwaru (Highsmith, 2000), DSDM (Stapleton, 1997; Stapleton, 2003) a vývoj založený na funkcích (Palmer a Felsing, 2002). Úspěch těchto metod vedl k částečné integraci s tradičními metodami vývoje založenými na modelování systému. Výsledkem je princip agilního modelování (Ambler a Jeffries, 2002) a agilní varianty procesu RUP (Larman, 2002).

Všechny tyto agilní metody jsou sice založeny na principu inkrementálního vývoje a dodávání, ale navrhují odlišné procesy, jak toho dosáhnout. Sdílejí však sadu principů založených na manifestu agilního vývoje, a mají tedy hodně společných prvků. Tyto principy jsou znázorněny na obrázku 3.1. Různé agilní metody příslušné principy konkretizují různými způsoby, v této kapitole však nemáme dost prostoru, abychom se zabývali všemi agilními metodami. Místo toho se zaměříme na dvě nejčastěji používané metody: extrémní programování (část 3.3) a Scrum (část 3.4).

Agilní metody jsou velmi úspěšné pro některé typy vývoje systémů:

1. Vývoj produktů, kde softwarová společnost vyvíjí malý nebo středně velký produkt k prodeji.

2. Vývoj zákaznických systémů v rámci organizace, kde si zákazník uvědomuje nutnost zapojit se do vývojového procesu a kde neexistuje mnoho externích pravidel a předpisů, které by měly vliv na software.

Princip	Popis
Zapojení zákazníka	Zákazníci by se měli úzce zapojit do vývojového procesu. Jejich role spočívá v poskytování nových požadavků na systém a určování jejich priorit a hodnocení iterací systému.
Inkrementální dodávání	Software se vyvíjí v inkrementech, kde zákazník specifikuje požadavky, které budou zahrnuty do vývoje každého inkrementu.
Lidé namísto procesu	Je potřeba rozpoznat a využít schopností vývojového týmu. Členové týmu by měli dostat možnost, aby vyvinuli své vlastní způsoby práce bez předem definovaných procesů.
Přijímání změn	Je nutné očekávat, že se požadavky na systém budou měnit, a navrhnout systém tak, aby tyto změny dokázal pojmout.
Zachování jednoduchosti	Je potřeba se soustředit na jednoduchost jak vyvíjeného softwaru, tak vývojového procesu. Kdekoli je to možné, měli by se členové týmu snažit, aby snižovali složitost systému.

Obrázek 3.1 – Principy agilních metod

Jak budeme diskutovat v závěrečné části této kapitoly, díky úspěchu agilních metod existuje značný zájem o jejich nasazení v jiných typech vývoje softwaru. Avšak vzhledem k tomu, že se tyto metody soustřeďují na malé a úzce integrované týmy, objevují se problémy při jejich škálování do větších systémů. Proběhly také experimenty s použitím agilních přístupů při vývoji kritických systémů (Drobna et al., 2004). Vzhledem k požadavkům na analýzu zabezpečení, bezpečnosti a spolehlivosti v kritických systémech však agilní metody vyžadují zásadní úpravy, než je lze rutinně použít v inženýringu kritických systémů.

Principy, na kterých jsou agilní metody založeny, se v praxi někdy realizují jen obtížně:

1. Myšlenka zapojit zákazníka do procesu vývoje je sice atraktivní, ale její úspěch vyžaduje zákazníka, který je ochoten a schopen trávit čas s vývojovým týmem a který dokáže reprezentovat všechny osoby zainteresované na systému. Zástupci zákazníka mají často jiné úkoly a nemohou se na procesu vývoje softwaru plně podílet.

2. Jednotliví členové týmu nemusí mít vhodné osobní vlastnosti, aby se dokázali intenzivně zapojit, jak to agilní metody vyžadují, a nemusí proto dobře interagovat s ostatními členy týmu.

3. Stanovení priorit změn může být velmi obtížné, zejména u systémů, které mají mnoho zainteresovaných osob. Každá z nich obvykle dává jednotlivým změnám odlišné priority.

4. Zachování jednoduchosti vyžaduje dodatečné úsilí. Pod tlakem plánu dodávek nemusí mít členové týmu dost času, aby prováděli žádoucí zjednodušení systému.

5. Mnoho organizací (zejména větších společností) strávilo mnoho let změnami své podnikové kultury a definováním a prosazováním jednotlivých procesů. Pro takové organizace je obtížné přejít na pracovní model, jehož procesy nejsou formální a definují je vývojové týmy.

Další mimotechnický problém – tj. obecný problém u inkrementálního vývoje a dování – se objevuje, když zákazník systému objednává vývoj systému u jiné organizace. Součástí smlouvy mezi zákazníkem a dodavatelem obvykle bývá dokument s požadavky na software. Vzhledem k tomu, že nedílnou částí agilních metod je inkrementální specifikace, může se komplikovat tvorba smluv na tento typ vývoje.

V důsledku toho se agilní metody musí spoléhat na smlouvy, kdy zákazník platí za čas potřebný k vývoji systému a nikoli za vývoj konkrétní sady požadavků. Dokud vše probíhá dobře, vyhovuje tento princip zákazníkovi i vývojáři. Jestliže se však objeví problémy, mohou nastat složité spory o to, kdo je za ně odpovědný a kdo by měl nést náklady na dodatečný čas a prostředky, které jsou k řešení problémů potřeba.

Většina knih a článků zabývajících se agilními metodami a zkušenostmi s nimi popisuje použití těchto metod při vývoji nových systémů. Jak ale vysvětlujeme v kapitole 9, velká část projektů softwarového inženýrství se týká údržby a evoluce existujících softwarových systémů. Existuje pouze málo zpráv o zkušenostech s použitím agilních metod při údržbě softwaru (Poole a Huisman, 2001). Při úvahách o agilních metodách a údržbě je také potřeba zvážit dvě otázky:

1. Lze systémy vyvíjené pomocí agilního přístupu udržovat, vzhledem k tomu, že vývojový proces klade důraz na minimalizaci formální dokumentace?
2. Mohou agilní metody efektivně umožnit evoluci systému v reakci na měnící se požadavky zákazníků?

Formální dokumentace by měla popisovat systém a usnadnit pochopení systému lidem, kteří jej mění. V praxi se však formální dokumentace často neudržuje aktuální, a proto ani přesně neodráží programový kód. Příznivci agilních metod proto tvrdí, že psaní dokumentace představuje plýtvání časem. Klíčem k implementaci softwaru, který lze udržovat, je podle nich tvorba vysoce kvalitního a čitelného kódu. Agilní procesy proto zdůrazňují význam psaní dobře strukturovaného kódu a investování energie do zlepšování kódu. Nedostatek dokumentace by tedy při údržbě systémů vyvinutých pomocí agilního přístupu neměl představovat problém.

Ze zkušeností autora s údržbou systémů však vyplývá klíčový význam dokumentu se systémovými požadavky, který softwarovému inženýrovi sděluje, co se od systému očekává. Bez těchto znalostí lze dopad navržených změn systému těžko vyhodnotit. Mnoho agilních metod shromažďuje požadavky neformálně a inkrementálně a nevytváří koherentní dokument s požadavky. V tomto ohledu může použití agilních metod pravděpodobně zkomplikovat a prodražit následnou údržbu systému.

Agilní postupy použité v samotném procesu údržby budou pravděpodobně efektivní bez ohledu na to, zda se agilní přístup používal při vývoji systému. Při změnách systému dává smysl použít principy inkrementálního dodávání, navrhování s ohledem na změny i zachování jednoduchosti. V praxi si proces agilního vývoje můžeme představit jako proces evoluce softwaru.

Hlavním problémem po dodání softwaru však nejspíš bude udržet zapojení zákazníka do celého procesu. Zákazník sice může odůvodnit zapojení svého zástupce na plný úvazek během procesu vývoje systému, ale tento přístup je méně pravděpodobný při údržbě, kdy ke změnám nedochází průběžně. Zástupci zákazníka obvykle o systém ztrácejí zájem. Při vytváření nových požadavků na systém tedy může být nutné využít alternativních mechanismů, jako jsou návrhy na změny, kterými se budeme zabývat v kapitole 25.

Běžně se vyskytuje i další problém, který souvisí se zachováním kontinuity vývojového týmu. Agilní metody se spoléhají na to, že členové týmu rozumí aspektům systému, aniž by se museli obracet na dokumentaci. Pokud se tým agilního vývoje rozpadne, jsou tyto implicitní vědomosti ztraceny a pro nové členy týmu je obtížné dosáhnout stejného stupně znalostí systému a jeho komponent.

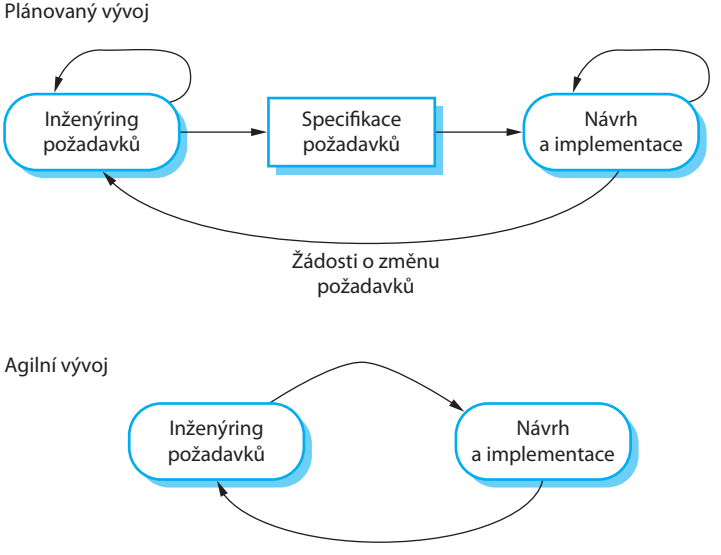
Zastánci agilních metod propagují tyto metody značně hlasitě a obvykle přehlížejí jejich nedostatky. Vzbuzují tím podobně krajní reakci, která podle názoru autora přehání problémy související s tímto přístupem (Stephens a Rosenberg, 2003). Uměřenější kritikové jako DeMarco a Boehm (DeMarco a Boehm, 2002) upozorňují jak na výhody, tak i na nevýhody agilních metod. Navrhují, že optimální

cestu vpřed může představovat hybridní přístup, kde agilní metody zahrnují některé techniky plánovaného vývoje.

3.2 Plánovaný a agilní vývoj

Agilní přístupy k vývoji softwaru považují návrh a implementaci za ústřední aktivity softwarového procesu. Do návrhu a implementace zahrnují i další aktivity, jako například zjišťování požadavků a testování. Oproti tomu plánovaný přístup k softwarovému inženýrství identifikuje samostatné fáze softwarového procesu, kdy s každou fází souvisí určité výstupy. Výstup jedné fáze slouží jako základ pro plánování následující aktivity procesu. Obrázek 3.2 znázorňuje rozdíly mezi plánovaným a agilním přístupem ke specifikaci systému.

U plánovaného přístupu dochází k iteracím v rámci aktivit a výměnu informací mezi fázemi procesu zajišťují formální dokumenty. Dochází například k vývoji požadavků a nakonec může být vytvořena specifikace požadavků. Tato specifikace pak slouží jako vstup procesů návrhu a implementace. V případě agilního přístupu přesahují iterace více aktivit. Požadavky a návrh se tedy nevyvíjejí samostatně, ale společně.



Obrázek 3.2 – Plánovaná a agilní specifikace

Plánovaný softwarový proces může podporovat inkrementální vývoj a dodávání. Není problém přidělit požadavky a naplánovat fáze návrhu a vývoje jako řadu inkrementů. Agilní proces nemusí být nutně soustředěn na kód a může produkovat určité návrhové dokumenty. Jak rozebereme v následující části, může se tým agilního vývoje rozhodnout, že vytvoří dokumentační „špičku“ a namísto nové verze systému připraví systémovou dokumentaci.

Většina softwarových projektů v praxi zahrnuje postupy plánovaných i agilních přístupů. Při rozhodování mezi plánovaným a agilním přístupem je nutné odpovědět na několik technických, osobních a organizačních otázek:

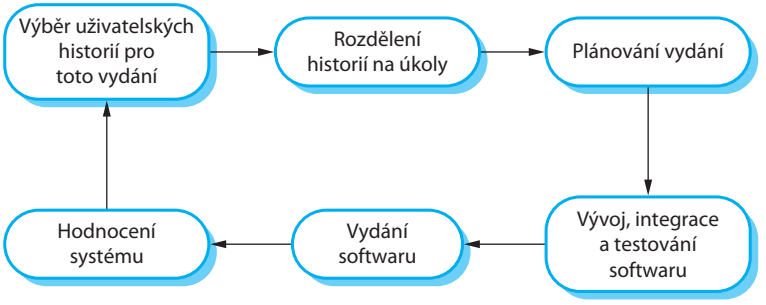
1. Je pro vás důležité mít před přechodem k implementaci velmi podrobnou specifikaci a návrh? Pokud ano, pravděpodobně potřebujete plánovaný přístup.
2. Je realistické nasadit strategii inkrementálního dodávání, kde budete dodávat software zákazníkovi a dostávat od něj rychlou zpětnou vazbu? Pokud ano, uvažujte o použití agilních metod.
3. Jak velký je vyvíjený systém? Agilní metody jsou nejefektivnější, když lze systém vyvinout v malém týmu na jediném pracovišti, který může neformálně komunikovat. Tato volba nemusí být dostupná u velkých systémů, které vyžadují větší vývojové týmy. V takových případech může být nutné zvolit plánovaný přístup.
4. Jaký typ systému je vyvíjen? Systémy, které před implementací vyžadují rozsáhlou analýzu (např. systémy fungující v reálném čase se složitými požadavky na časování), obvykle k provedení takové analýzy potřebují poměrně podrobný návrh. Za těchto okolností může být nejlepší plánovaný přístup.
5. Jakou má systém očekávanou životnost? Systémy s dlouhým životním cyklem mohou vyžadovat rozsáhlejší návrhovou dokumentaci, aby se tým podpory mohl seznámit s původními záměry systémových vývojářů. Zastánci agilních metod však oprávněně upozorňují, že dokumentace se často neudržuje aktuální a při dlouhodobé údržbě systému není příliš užitečná.
6. Které technologie na podporu vývoje systému jsou k dispozici? Agilní metody se často spoléhají na kvalitní nástroje, které dokáží vyvíjející se návrh sledovat. Pokud vyvíjíte systém v prostředí IDE, které neobsahuje vhodné nástroje na vizualizaci a analýzu programu, můžete potřebovat rozsáhlejší návrhovou dokumentaci.
7. Jaká je organizace vývojového týmu? Jestliže je vývojový tým distribuovaný nebo se část vývoje řeší formou outsourcingu, může být nutné vytvořit návrhové dokumenty, které zajistí komunikaci mezi vývojovými týmy. Strukturu těchto dokumentů může být potřeba naplánovat předem.
8. Mohou vývoj systému ovlivnit kulturní aspekty? Tradiční inženýrské organizace mívají kulturu plánovaného vývoje, protože v inženýrství představuje normu. Tento vývoj obvykle vyžaduje rozsáhlou návrhovou dokumentaci a nikoli neformální znalosti, které se využívají v agilních procesech.
9. Jak dobří jsou návrháři a programátoři vývojového týmu? Někdy se tvrdí, že agilní metody vyžadují vyšší úroveň schopností než plánované přístupy, kde programátoři pouze převádějí podrobný návrh do podoby kódu. Máte-li tým s relativně nízkými zkušenostmi, může být nutné nasadit nejlepší lidi na vývoj návrhu a zbytek přidělit na úkoly programování.
10. Podléhá systém externím předpisům? Jestliže musí být systém schválen externím regulačním úřadem (například v USA úřad FAA – Federal Aviation Authority – schvaluje software, který hraje kritickou roli při provozu letadla), pak bude nejspíš nutné vytvořit podrobnou dokumentaci v rámci bezpečnostního hodnocení systému.

Otázka, zda bude projekt označen jako plánovaný nebo agilní, ve skutečnosti není příliš důležitá. Zákazníci softwarového systému se zajímají hlavně o to, zda dostanou funkční softwarový systém, který bude splňovat jejich požadavky a bude přínosný pro jednotlivé uživatele v organizaci. Mnohé společnosti, které se přihlásily k použití agilních metod, v praxi přijaly některé agilní postupy a integrovaly je do svých plánovaných procesů.

3.3 Extrémní programování

Extrémní programování (XP) pravděpodobně představuje nejznámější a nejvíce používanou agilní metodu. S názvem přišel Beck (2000), protože jeho přístup posunoval uznávané optimální metody, jako

je iterativní vývoj, na „extrémní“ úroveň. V XP může například několik programátorů během jediného dne vyvinout několik nových verzí systému a tyto verze integrovat a otestovat.



Obrázek 3.3 – Cyklus vydání v extrémním programování

V extrémním programování se požadavky vyjadřují jako scénáře (označované jako uživatelské historie), které se implementují přímo jako řada úkolů. Programátoři pracují ve dvojicích a před psaním kódu vyvíjejí testy pro každý úkol. Při integraci nového kódu do systému musí být všechny testy provedeny úspěšně. Jednotlivá vydání systému odděluje pouze malá časová mezera. Obrázek 3.3 dokládá, jakým způsobem proces XP produkuje inkrement vyvíjeného systému.

Extrémní programování zahrnuje mnoho postupů shrnutých na obrázku 3.4, které odrážejí principy agilních metod:

1. Podporu inkrementálního vývoje zajišťují malá a častá vydání systému. Požadavky jsou založeny na jednoduchých zákaznických historiích či scénářích, podle nichž lze rozhodnout, které funkce bude inkrement systému zahrnovat.
2. Zapojení zákazníků se dosahuje díky trvalé spolupráci zástupce zákazníka s vývojovým týmem. Zástupce zákazníka se vývoje přímo účastní a odpovídá za definování předávacích testů systému.
3. Princip „lidé, nikoli procesy“ vychází z programování ve dvojicích, kolektivního vlastnictví systémového kódu a udržitelného procesu vývoje bez příliš dlouhé pracovní doby.
4. Přijímání změn je založeno na pravidelném vydávání systému zákazníkovi, vývoji s testováním hned na začátku, refaktoringu, který zabraňuje degenerování kódu, a na trvalém integrování nových funkcí.
5. Zachování jednoduchosti zajišťuje neustálý refaktoring, který zlepšuje kvalitu kódu, a použití jednoduchých návrhových vzorů, které zbytečně nepředjímají budoucí změny systému.

V procesu XP se zákazníci úzce podílejí na specifikaci a stanovení priorit systémových požadavků. Požadavky nejsou definovány formou seznamů požadovaných systémových funkcí. Zákazník systému spíše plní roli člena vývojového týmu a diskutuje scénáře s jinými členy týmu. Společně vyvíjejí „kartu historie“, která zahrnuje zákaznické potřeby. Vývojový tým se poté snaží daný scénář implementovat v budoucím vydání softwaru. Příkladem je karta historie systému správy záznamů o psychiatrických pacientech, kterou znázorňuje obrázek 3.5. Jedná se o stručný popis scénáře pro předepsání léku pacientovi.

Princip nebo postup	Popis
Inkrementální plánování	Požadavky se zaznamenávají na karty historie. Historie, které budou součástí vydání, se vybírají podle dostupného času a své relativní priority. Vývojáři rozdělují tyto historie do vývojových „úkolů“. Viz obrázky 3.5 a 3.6.

Princip nebo postup	Popis
Malá vydání	Nejdříve se vyvíjí minimální užitečná sada funkcí, která poskytuje obchodní hodnotu. Vydání systému následují rychle po sobě a inkrementálně doplňují funkce prvního vydání.
Jednoduchý návrh	Návrh je pouze natolik složitý, aby vyhovoval aktuálním požadavkům.
Vývoj s testováním hned na začátku	Automatizovaná struktura testování jednotek umožňuje vytvářet testy na nové funkce ještě dříve, než jsou příslušné funkce implementovány.
Refaktoring	Od všech vývojářů se očekává, že budou neustále provádět refaktoring kódu ihned poté, co najdou jeho možná zlepšení. Tento postup zachovává jednoduchost kódu a umožňuje jej udržovat.
Programování ve dvojicích	Vývojáři pracují v párech, vzájemně kontrolují svou práci a podporují se v tom, aby neustále odváděli kvalitní práci.
Kolektivní vlastnictví	Dvojice vývojářů pracují na všech oblastech systému, takže se nevyvíjejí jakési izolované znalostní oblasti a všichni vývojáři přijímají odpovědnost za celý kód. Každý může cokoli změnit.
Trvalá integrace	Ihned po dokončení práce jsou všechny úkoly integrovány do celého systému. Po každé takové integraci musí úspěšně proběhnout testy všech jednotek v systému.
Udržitelné tempo	Časté přesčasy se netolerují, protože jejich čistým efektem často bývá snížení kvality kódu a střednědobé produktivity.
Zákazník u vývojáře	Zástupce koncového uživatele systému (Zákazník) by měl být týmu XP k dispozici na plný úvazek. V procesu extrémního programování je zákazník členem vývojového týmu a jeho úkolem je předkládat týmu k implementaci jednotlivé systémového požadavky.

Obrázek 3.4 – Postupy extrémního programování

Karty historie jsou hlavním vstupem procesu plánování XP neboli „plánovací hry". Jakmile jsou tyto karty vytvořeny, vývojový tým rozdělí historie na úkoly (viz obrázek 3.6) a odhadne objem práce a prostředků na implementaci každého úkolu. Obvykle se přitom diskutuje se zákazníkem a požadavky se upřesňují. Zákazník pak určí prioritu historií k implementaci a vybere historie, které lze při poskytování užitečných podnikových funkcí využít okamžitě. Cílem je identifikovat užitečné funkce, které je možné implementovat asi do dvou týdnů, kdy zákazník dostane další vydání systému.

Jak se požadavky mění, mohou se neimplementované historie samozřejmě také měnit nebo mohou být úplně vypuštěny. Jestliže je nutné provést změny systému teprve po jeho dodání, jsou vytvořeny nové karty historie a zákazník se znovu rozhoduje, zda by tyto změny měly mít prioritu před novými funkcemi.

Ve fázi plánování se někdy objeví otázky, na které nelze snadno odpovědět, a prozkoumání možných řešení vyžaduje více úsilí. Tým může vytvořit určité prototypy nebo provést zkušební vývoj, aby problému a řešení lépe porozuměl. V terminologii XP se jedná o „špičku" (spike), tj. inkrement, kde nedochází k žádnému programování. Lze také provést „špičky" k návrhu systémové architektury nebo vývoji systémové dokumentace.

Extrémní programování volí „extrémní" přístup k inkrementálnímu vývoji. Nové verze softwaru lze sestavovat několikrát denně a jednotlivá vydání jsou zákazníkovi poskytována přibližně každé dva týdny. Termíny verzí se nikdy nezpožďují. Pokud při vývoji dochází k problémům, zákazník je konzultován a z plánovaného vydání jsou odstraněny některé funkce.

Předepsání léku

Katka je lékařka, která chce předepsat lék pacientovi navštěvujícímu kliniku. Na obrazovce počítače má již zobrazen pacientův záznam, takže klepne na pole léku a vybere možnost „aktuální lék", „nový lék" nebo „lékopis".

Jestliže zvolí možnost „aktuální lék", systém ji požádá, aby zkontrolovala dávku. Chce-li dávku změnit, zadá novou hodnotu a poté recept potvrdí.

Pokud vybere možnost „nový lék", systém předpokládá, že předepisovaný lék zná. Zadá několik prvních písmen názvu léku. Systém zobrazí seznam možných léků, které začínají těmito písmeny. Zvolí požadovaný lék. Systém reaguje tím, že ji požádá, aby zkontrolovala, zda je vybraný lék správný. Lékařka zadá dávku a poté recept potvrdí.

Pokud vybere možnost „lékopis", systém zobrazí pole pro vyhledávání v lékopisu. Poté může vyhledat požadovaný lék. Vybere lék a zobrazí se dotaz, zda je příslušná medikace správná. Lékařka zadá dávku a poté recept potvrdí.

Systém vždy kontroluje, zda dávka patří do schváleného rozsahu. Pokud tomu tak není, systém lékařku požádá, aby dávku změnila.

Potvrzený recept se pro kontrolu zobrazí znovu. Lékařka může klepnout na tlačítko „OK" nebo „Změnit". Klepne-li na tlačítko „OK", recept je zaznamenán do databáze k auditu. Jestliže klepne na tlačítko „Změnit", vrátí se zpět na začátek procesu „předepsání léku".

Obrázek 3.5 – Historie „předepsání léku"

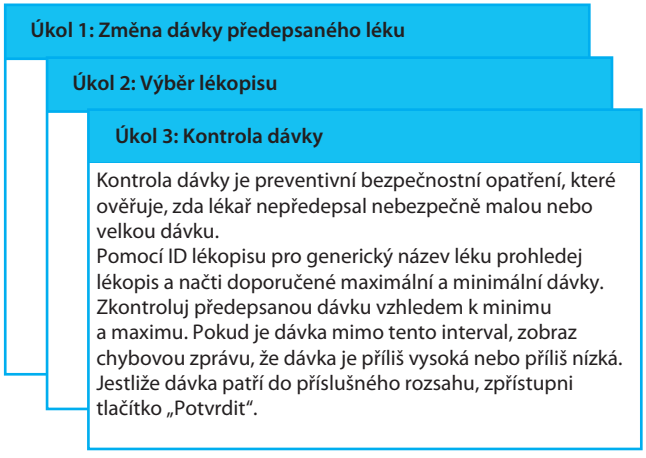
Když programátor při vytváření nové verze sestaví systém, musí spolu s testy na novou funkčnost provést všechny automatizované testy. Nové sestavení softwaru je přijato pouze v případě, že všechny testy proběhnou úspěšně. Toto sestavení je pak základem další iterace systému.

Základní zásada tradičního softwarového inženýrství spočívá v tom, že je potřeba navrhovat s ohledem na změny. To znamená, že vývojáři by měli počítat s budoucími změnami softwaru a navrhovat je tak, aby bylo možné tyto změny snadno implementovat. Extrémní programování však tento princip nepřijímá. Argumentuje přitom tím, že navrhování s ohledem na změny často plýtvá úsilím. Nestojí za to věnovat čas na zobecnění programu, aby jej bylo možné lépe měnit. Předpokládané změny se často nikdy neobjeví a místo nich může být nutné realizovat požadavky na zcela odlišné změny. Přístup XP tedy akceptuje, že změny nastávají, a mění organizaci softwaru teprve tehdy, když k těmto změnám skutečně dojde.

Obecný problém s inkrementálním vývojem spočívá v tom, že zpravidla degraduje strukturu softwaru. Změny softwaru se proto časem implementují stále obtížněji. Vývoj v zásadě pokračuje tak, že hledá způsoby, jak problémy obejít. Výsledkem je kód, který je často duplikovaný, části softwaru se znovu používají nevhodnými způsoby a celková struktura swe degraduje spolu s tím, jak v systému přibývá nový kód.

Extrémní programování řeší tento problém tak, že doporučuje neustálý refaktoring kódu. To znamená, že programátorský tým hledá možná zlepšení softwaru a okamžitě je implementuje. Když člen týmu najde část kódu, kterou lze zdokonalit, provede tyto změny i v situacích, kdy nejsou bezprostředně potřeba. Jako příklady refaktoringu je možné uvést reorganizaci hierarchie tříd, aby se odstranil duplicitní kód, vyčištění a přejmenování atributů a metod a nahrazení kódu s voláními metod, které jsou definovány v programové knihovně. Vývojová prostředí typu Eclipse (Carlson, 2005) obsahují nástroje

pro refaktoring, které zjednodušují proces hledání závislostí mezi částmi kódu a provádění globálních úprav kódu.



Obrázek 3.6 – Příklady karet úkolů pro předepsání léku

V principu by tedy měl být software vždy snadno srozumitelný a měl by umožňovat změny související s implementací nových historií. V praxi to však pokaždé neplatí. Vývojové tlaky někdy způsobují, že se refaktoring zpožďuje, protože se čas investuje na implementaci nových funkcí. Některé nové funkce a změny nelze snadno zajistit pomocí refaktoringu na úrovni kódu a vyžadují úpravy architektury celého systému.

Mnoho společností, které přijaly vývoj XP, nepoužívá všechny postupy extrémního programování uvedené na obrázku 3.4. Vybírají z nich s ohledem na svůj specifický styl práce pouze část. V některých společnostech se například osvědčuje programování ve dvojicích, jiné dávají přednost individuálnímu programování a revizím. Vzhledem k odlišné úrovni zkušeností někteří programátoři neprovádí refaktoring částí systému, které sami nevyvíjeli, a namísto uživatelských historií se používají konvenční požadavky. Většina společností, které přijaly variantu XP, však používá principy malých vydání, vývoje s testováním hned na začátku a průběžné integrace.

3.3.1 Testování v XP

Jak jsme diskutovali v úvodu této kapitoly, jeden z důležitých rozdílů mezi inkrementálním a plánovaným vývojem spočívá ve způsobu testování systému. U inkrementálního vývoje neexistuje žádná specifikace systému, podle níž by mohl externí testovací tým vyvinout systémové testy. V důsledku toho mají některé přístupy k inkrementálnímu vývoji v porovnání s plánovaným testováním velmi neformální testovací proces.

Jako prevenci některých problémů s testováním a validací systému zdůrazňuje metoda XP význam testování programu. XP zahrnuje přístup k testování, který omezuje pravděpodobnost, že se do aktuální verze systému dostanou nezjištěné chyby.

Testování v XP má následující klíčové funkce:

1. Vývoj s testováním hned na začátku
2. Inkrementální vývoj testů ze scénářů

3. Zapojení uživatelů do vývoje a validace testů
4. Použití automatických testovacích architektur

Vývoj s testováním hned na začátku patří mezi nejdůležitější inovace XP. Místo toho, aby programátoři vytvořili nějaký kód a poté psali testy tohoto kódu, píší nejdříve testy a teprve poté programový kód. To znamená, že lze testy spouštět při práci na kódu a zjišťovat problémy již během vývoje.

Psaní testů implicitně definuje rozhraní i specifikaci chování pro vyvíjenou funkci. Omezuje se rozsah problémů s nedostatečným porozuměním požadavkům a rozhraní. Tento přístup lze aplikovat na libovolný proces, kde existuje jasný vztah mezi požadavkem na systém a kódem, který tento požadavek implementuje. V XP je možné tuto vazbu sledovat vždy, protože karty historie reprezentující požadavky se dělí na úkoly a tyto úkoly jsou základní jednotkou implementace. Přijetí vývoje s testováním hned na začátku v XP vedlo k obecnějším přístupům k vývoji založeným na testech (Astels, 2003). Tyto přístupy rozebereme v kapitole 8.

Při vývoji s testováním hned na začátku musí programátoři, kteří implementují konkrétní úkol, dokonale rozumět jeho specifikaci, aby mohli napsat testy systému. To znamená, že nejasnosti a opomenutí ve specifikaci je nutné vysvětlit ještě před zahájením implementace. Kromě toho odpadá problém se „zpožděným testováním". K tomu může dojít, když vývojář systému pracuje rychleji než jeho tester. Implementace získává stále větší náskok před testováním a vzniká tendence k vynechávání testů, aby bylo možné dodržet plán vývoje.

Uživatelské požadavky se v XP vyjadřují jako scénáře nebo historie a uživatel určuje priority vývoje. Vývojový tým hodnotí jednotlivé scénáře a dělí je na úkoly. Na obrázku 3.6 jsou například znázorněny některé karty úkolů vyvinuté z karty historie pro předepsání léku (viz obrázek 3.5). Každý úkol generuje jeden nebo více testů jednotek, které kontrolují implementaci popsanou v daném úkolu. Obrázek 3.7 představuje zkrácený popis testovacího případu, který byl vytvořen k testování toho, zda předepsaná dávka léku neleží mimo známé bezpečné limity.

Test 4: Kontrola dávky

Vstup:
1. Číslo v miligramech, které udává jedinou dávku léku.
2. Číslo, které určuje počet jednotlivých dávek za den.

Testy:
1. Testování vstupů, kde jediná dávka je správná, ale frekvence je příliš vysoká.
2. Testování vstupů, kde jediná dávka je příliš vysoká a příliš nízká.
3. Testování vstupů, kde součin jediné dávky a frekvence je příliš vysoký a příliš nízký.
4. Testování vstupů, kde se součin jediné dávky a frekvence nachází v povoleném rozsahu.

Výstup:
Zpráva typu OK nebo chybová zpráva, která informuje o tom, že dávka leží mimo bezpečný rozsah.

Obrázek 3.7 – Popis testovacího případu na kontrolu dávky

Role zákazníka na procesu testování spočívá v tom, že pomáhá vyvinout předávací testy pro historie, které mají být implementovány v dalším vydání systému. Jak analyzujeme v kapitole 8, předávací testování je proces, kdy se systém testuje s použitím zákaznických dat, aby se zjistilo, zda splňuje skutečné potřeby zákazníka.

V XP probíhá předávací testování stejně jako vývoj inkrementálně. Zákazník, který je součástí týmu, píše testy podle toho, jak pokračuje vývoj. Veškerý nový kód je proto validován, aby bylo zajištěno, že

odpovídá požadavkům zákazníka. U historie na obrázku 3.5 by předávací test zahrnoval scénáře, kde (a) dávka léku se změnila, (b) byl vybrán nový lék a (c) proběhlo vyhledání léku v lékopisu. V praxi se namísto jediného testu často vyžaduje řada předávacích testů.

Spoléhání na to, že zákazník podpoří vývoj předávacích testů, často představuje zásadní potíž procesu testování XP. Osoby v roli zástupce zákazníka mívají velmi málo dostupného času a nemusí být k dispozici, aby s vývojovým týmem pracovaly na plný úvazek. Zákazník se může domnívat, že k vývoji již dostatečně přispěl poskytnutím požadavků, a se zapojením do testovacího procesu může váhat.

Při vývoji testování hned na začátku má zásadní význam automatizace testů. Testy se píší jako spustitelné komponenty před implementací úkolu. Tyto testovací komponenty by měly být samostatné, měly by simulovat zadávání testovaného vstupu a měly by zjišťovat, zda výsledky splňují specifikaci výstupu. Automatizovaná testovací architektura je systém, který usnadňuje psaní spustitelných testů a předávání sady testů ke spuštění. Příkladem často používané automatizované testovací architektury je nástroj Junit (Massol a Husted, 2003).

Díky automatizaci testování vždy existuje sada testů, které lze snadno a rychle spustit. Testy lze spustit při každém přidání funkcí do systému a okamžitě tak zachytit problémy, které nový kód zanesl.

Díky vývoji s testováním hned na začátku a automatizovanému testování se obvykle píše a spouští velký počet testů. Tento přístup však nutně nevede ke kompletnímu testování programu. Je to způsobeno třemi důvody:

1. Programátoři dávají přednost programování před testováním a někdy si tvorbu testů zjednodušují. Mohou například psát neúplné testy, které nekontrolují všechny výjimky, jež mohou nastat.
2. Některé testy může být velmi obtížné napsat inkrementálně. Například ve složitém uživatelském rozhraní je často těžké napsat testy jednotek pro kód, který implementuje „zobrazovací logiku" a přechody mezi obrazovkami.
3. Jen obtížně lze hodnotit úplnost sady testů. Může sice existovat mnoho systémových testů, ale sada testů nemusí poskytovat úplné pokrytí. Někdy nejsou spuštěny klíčové části systému, které tak zůstávají netestované.

Ačkoli tedy rozsáhlá sada často spouštěných testů může vzbuzovat dojem, že systém je úplný a správný, nemusí to být pravda. Jestliže testy nejsou revidovány a po dokončení vývoje nejsou vytvořeny další testy, může vydání systému obsahovat nezjištěné chyby.

3.3.2 Programování ve dvojicích

Vývoj XP zavádí další inovativní postup: programátoři při psaní softwaru pracují ve dvojicích. V praxi při vývoji softwaru sedí vedle sebe u stejné pracovní stanice. Stejná dvojice však neprogramuje společně pokaždé. Dvojice spíše vznikají dynamicky, aby během vývojového procesu spolupracovali všichni členové týmu.

Použití programování ve dvojicích přináší několik výhod:

1. Podporuje myšlenku kolektivního vlastnictví a odpovědnosti za systém. V tom se odráží Weinbergova (1971) idea nesobeckého programování, kde software vlastní tým jako celek a za problémy s kódem neodpovídají jednotlivci. Místo toho přebírá kolektivní odpovědnost za řešení problému celý tým.
2. Funguje jako neformální proces revize, protože každý řádek kódu sledují alespoň dvě osoby. Inspekce a revize kódu (budeme se jim věnovat v kapitole 24) dokážou s vysokou úspěšností odhalit velké procento softwarových chyb. Jejich organizace je však časově náročná a obvykle do procesu

vývoje zavádějí zpoždění. Proces programování ve dvojicích je sice méně formální a pravděpodobně nenajde tolik chyb jako inspekce kódu, ale jedná se oproti formálním inspekcím programu o mnohem levnější kontrolu.

3. Pomáhá podpořit refaktoring, což je proces zlepšování softwaru. V normálním vývojovém prostředí naráží refaktoring na potíže, protože investice na něj vynakládané se vyplatí teprve v dlouhodobém výhledu. Jednotlivec, který se věnuje refaktoringu, může být hodnocen jako méně efektivní než ten, který se zabývá pouze vývojem kódu. Tam, kde se uplatňuje programování ve dvojicích a kolektivní vlastnictví, mají jiní vývojáři z refaktoringu okamžité výhody, takže celý proces budou pravděpodobně podporovat.

Mohli bychom očekávat, že programování ve dvojicích bude méně efektivní než individuální práce. Za daný čas vytvoří dvojice vývojářů poloviční objem kódu než dva jednotlivci, kteří pracují samostatně. Proběhly různé studie týkající se produktivity placených programátorů. Tyto studie přitom poskytly smíšené výsledky. U studentských dobrovolníků zjistila Williams se svými spolupracovníky (Cockburn a Williams, 2001; Williams et al., 2000), že produktivita programování ve dvojicích je srovnatelná se dvěma osobami, které pracují nezávisle. Výzkumníci navrhli vysvětlení, že dvojice diskutují o softwaru ještě před jeho vývojem, takže dochází k menšímu počtu falešných startů a omezuje se přepracování. Kromě toho se při takové neformální inspekci opraví tolik chyb, že je nutné vynaložit méně času na opravu chyb zjištěných v procesu testování.

Ve studiích se zkušenějšími programátory (Arisholm et al., 2007; Parrish et al., 2004) se však tyto výsledky nepodařilo zopakovat. Studie zjistily, že v porovnání se dvěma programátory pracujícími samostatně došlo k významné ztrátě produktivity. Projevily se sice určité výhody z hlediska kvality, které však režii spojenou s programováním ve dvojicích plně nekompenzovaly. V každém případě je však velmi důležité sdílení znalostí, ke kterému při programování ve dvojicích dochází, protože se tím snižuje celkové riziko, jaké pro projekt znamená odchod členů týmu. Tato výhoda sama o sobě může jako argument ve prospěch programování ve dvojicích stačit.

3.4 Řízení agilních projektů

Základní odpovědnost vedoucích softwarových projektů leží v řízení projektu, kdy je nutné zajistit, aby byl software dodán včas a aby byl dodržen plánovaný rozpočet projektu. Vedoucí dohlížejí na práci softwarových inženýrů a monitorují, nakolik vývoj softwaru postupuje.

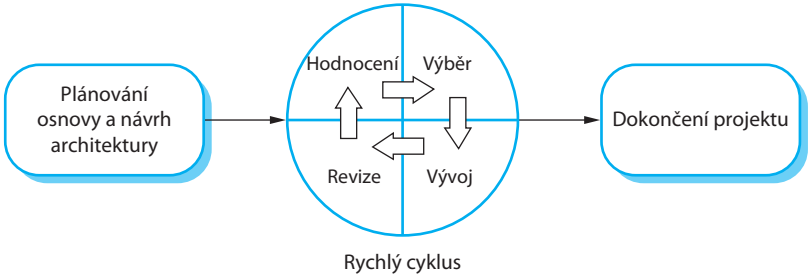
Standardní přístup k řízení projektů je založen na plánech. Jak rozebereme v kapitole 23, vedoucí vytvoří plán projektu s konkrétními údaji o tom, co se bude dodávat, kdy to má být dodáno a kdo bude na vývoji projektových výstupů pracovat. Plánovaný přístup skutečně vyžaduje vedoucího, který má neustále přehled o všem, co je nutné vyvinout, a o vývojových procesech. Tento přístup však dobře nefunguje u agilních metod, kde se požadavky vyvíjejí inkrementálně, software se dodává v krátkých a rychlých inkrementech a změny požadavků a softwaru jsou spíše pravidlem než výjimkou.

Stejně jako všechny ostatní vývojové procesy profesionálního softwaru je také agilní vývoj nutné řídit, aby bylo možné optimálně využít dostupného času a prostředků týmu. Přitom je nutné zvolit odlišný přístup k řízení projektů, který je přizpůsoben inkrementálnímu dodávání a projevuje své výhody u agilních metod.

Obecnou agilní metodu představuje přístup Scrum (Schwaber, 2004; Schwaber a Beedle, 2001). Zaměřuje se však na řízení iterativního vývoje a nikoli na konkrétní technické přístupy k agilnímu softwarovému inženýrství. Na obrázku 3.8 je znázorněn diagram řídicího procesu Scrum. Scrum nepředepisuje

používání programátorských postupů, jako je programování ve dvojicích a testování hned na začátku. Lze jej tedy nasadit spolu s techničtějšími agilními přístupy, jako je XP, kdy poskytuje rámec řízení projektu.

Přístup Scrum má tři fáze. První je fáze plánování osnovy, kde se určují obecné cíle projektu a navrhuje se softwarová architektura. Poté následuje řada rychlých cyklů, kdy každý cyklus vyvíjí jeden inkrement systému. V poslední fázi dokončení projektu je celý projekt uzavřen, dopracuje se požadovaná dokumentace, jako jsou okna systémové nápovědy a uživatelské příručky, a vyhodnotí se poznatky získané při práci na projektu.



Obrázek 3.8 – Proces Scrum

Inovativní částí přístupu Scrum je jeho centrální fáze, konkrétně rychlé cykly. Rychlý cyklus Scrum je plánovací jednotka, kdy se hodnotí plánovaná práce, vybírají se funkce pro vývoj a implementuje se software. Na konci rychlého cyklu se zainteresovaným osobám dodává kompletní sada funkcí. Tento proces má následující klíčové vlastnosti:

1. Rychlé cykly mají pevnou délku, obvykle 2–4 týdny. Odpovídají vývoji vydání systému v XP.
2. Plánování začíná produktovými resty, což je seznam prací, které je na projektu potřeba provést. Během fáze hodnocení rychlého cyklu se tyto resty revidují a přiřazují se priority a rizika. Na tomto procesu se úzce podílí zákazník, který může na začátku každého rychlého cyklu zavést nové požadavky nebo úkoly.
3. Na fázi výběru se podílí celý projektový tým, který spolu se zákazníkem vybírá vlastnosti a funkce, které se budou v rychlém cyklu vyvíjet.
4. Po dohodnutí těchto parametrů se tým uspořádá s ohledem na vývoj softwaru. Pořádají se krátké denní schůzky s účastí všech členů týmu, které rekapitulují postup a v případě nutnosti mění prioritu prací. Během této fáze je tým izolován od zákazníka a organizace a veškerá komunikace probíhá přes takzvaného „Scrum-šéfa“ (Scrum master). Rolí Scrum-šéfa je chránit vývojový tým před rozptylováním zvnějšku. Způsob, jakým to probíhá, závisí na problému a na týmu. Na rozdíl od přístupu XP neposkytuje Scrum konkrétní doporučení, jak psát požadavky, testovat hned na začátku atd. Pokud se však tým domnívá, že jsou tyto postupy XP vhodné, může je použít.
5. Na konci rychlého cyklu se provedená práce reviduje a předkládá zainteresovaným osobám. Poté začíná další rychlý cyklus.

Idea přístupu Scrum spočívá v tom, že celý tým by měl mít dostatečné pravomoci, aby mohl přijímat rozhodnutí, takže se úmyslně přestává používat termín „vedoucí projektu“. „Scrum-šéf“ je spíše asistent, který organizuje každodenní schůzky, sleduje pracovní resty, které je potřeba zvládnout, zapisuje rozhodnutí, měří postup vzhledem k restům a komunikuje se zákazníky a managementem mimo tým.

Celý tým se účastní denních schůzkek, které se často pořádají ve stoje, aby zůstaly krátké a věcné. Během schůzek všichni členové týmu sdílejí informace, popisují svůj postup od poslední schůzky, zmiňují problémy, které se vyskytly, a seznamují se s plánem na následující den. To znamená, že všichni členové týmu vědí, co se děje, a pokud se objeví potíže, mohou změnit krátkodobé pracovní plány, aby tyto potíže vyřešili. Na krátkodobém plánování se podílejí všichni – neexistuje žádná hierarchie, na jejímž vrcholu by se nacházel Scrum-šéf.

Na webu lze najít mnoho neoficiálních zpráv o úspěšném uplatnění přístupu Scrum. Rising a Janoff (2000) diskutují úspěšné nasazení tohoto přístupu v prostředí vývoje softwaru pro telekomunikace. Uvádějí přitom jeho následující výhody:

1. Produkt se dělí na řadu kontrolovatelných a srozumitelných částí.
2. Nestabilní požadavky neblokují postup.
3. Celý tým má přehled o všem a v důsledku toho se zlepšuje týmová komunikace.
4. Zákazníci vidí včasné dodávání inkrementů a získávají zpětnou vazbu o tom, jak produkt funguje.
5. Buduje se důvěra mezi zákazníky a vývojáři a vytváří se pozitivní kultura, kde všichni očekávají úspěch projektu.

Přístup Scrum tak, jak byl původně navržen, byl určen k použití ve společně pracujících týmech, kde se všichni členové mohou každý den setkávat na schůzkách ve stoje. Většinu vývoje softwaru však nyní zajišťují distribuované týmy, jejichž členové se nacházejí na různých místech světa. V důsledku toho se objevují různé experimenty, které se snaží přizpůsobit přístup Scrum pro distribuované vývojové prostředí (Smits a Pshigoda, 2007; Sutherland et al., 2007).

3.5 Škálování agilních metod

Agilní metody vznikly proto, aby je mohly používat malé programátorské týmy, které mohou pracovat ve stejné místnosti a neformálně komunikovat. Agilní metody se proto nejčastěji uplatňují při vývoji malých a středně velkých systémů. Potřeba rychlejšího vývoje softwaru, který by lépe vyhovoval požadavkům zákazníků, se však vztahuje i na větší systémy. V důsledku toho je věnována velká pozornost škálování agilních metod, aby dokázaly zvládnout větší systémy, jaké se vyvíjejí ve velkých organizacích.

Denning et al. (2008) tvrdí, že jediný způsob, jak se vyhnout běžným problémům softwarového inženýrství, jako jsou systémy nesplňující potřeby zákazníků a překročení rozpočtu, spočívá v nalezení cest, jak agilní metody přizpůsobit pro velké systémy. Leffingwell (2007) diskutuje, které agilní postupy lze škálovat na vývoj větších systémů. Moore and Spens (2008) informují o svých zkušenostech při nasazení agilního přístupu na vývoj velkého zdravotnického systému se 300 vývojáři, kteří pracovali v geograficky distribuovaných týmech.

Vývoj velkých softwarových systémů se liší od vývoje malých systémů několika způsoby:

1. Velké systémy obvykle sestávají z více samostatných komunikujících systémů, kde každý systém vyvíjí jiný tým. Tyto týmy často pracují na různých místech, někdy v odlišných časových pásmech. Prakticky není možné, aby měl každý tým přehled o celém systému. Prioritou každého týmu proto zpravidla bývá dokončení jeho části systému bez ohledu na širší systémové aspekty.
2. Velké systémy vznikají jako „systémy typu brownfield“ (Hopkins a Jenkins, 2008). To znamená, že se nebudují od základů, ale zahrnují mnoho existujících systémů a interagují s nimi. Mnohé systémové požadavky souvisejí s těmito interakcemi, a proto v praxi neumožňují flexibilitu a inkrementální vývoj. Důležitý vliv mohou mít i politické otázky – nejsnadnější řešení problému často

spočívá ve změně existujícího systému. To však vyžaduje jednání s vedoucími příslušného systému, kdy je nutné tyto osoby přesvědčit, že lze dané změny implementovat bez rizika pro fungování jejich systému.

3. Když dochází k integraci několika systémů do nového systému, je významná část vývoje namísto vývoje původního kódu zaměřena na konfiguraci systému. To nemusí být často kompatibilní s inkrementálním vývojem a častou integrací systému.
4. Velké systémy a jejich vývojové procesy jsou často omezeny externími pravidly a předpisy, které omezují možné metody vývoje, vyžadují tvorbu jistého typu systémové dokumentace atd.
5. Velké systémy se vyznačují dlouhou dobou pořizování a vývoje. Je obtížné po celou tuto dobu zachovat koherentní týmy se znalostmi systému, protože členové týmu se nevyhnutelně přesunují na jiné úkoly a projekty.
6. Velké systémy zpravidla mívají rozmanitou skupinu zainteresovaných osob. Koncovými uživateli zdravotnického systému například mohou být sestry a administrativní pracovníci, ale na systému jsou zainteresováni také primáři, nemocniční manažeři a další lidé. Účast všech těchto osob na vývojovém procesu prakticky není možné zajistit.

Na škálování agilních metod lze pohlížet ze dvou perspektiv:

1. Perspektiva „škálování zdola nahoru" se zabývá použitím těchto metod při vývoji velkých softwarových systémů, jaké nemůže vyvíjet malý tým.
2. Perspektiva „škálování do šířky" se zaměřuje na způsob, jak je možné zavést agilní metody ve velkých organizacích, které mají s vývojem softwaru dlouholeté zkušenosti.

Mají-li agilní metody zvládnout inženýring velkých systémů, je nutné je přizpůsobit. Leffingwell (2007) prohlašuje, že zásadní význam má zachování základních prvků agilních metod – pružného plánování, častých vydání systému, průběžné integrace, vývoje založeného na testování a dobré týmové komunikace. Autor této knihy se domnívá, že je potřeba zavést následující zásadní přizpůsobení:

1. Při vývoji velkých systémů se není možné zaměřit pouze na kód systému. Je potřeba více investovat do předběžného návrhu a systémové dokumentace. Je nutné navrhnout softwarovou architekturu a vytvořit dokumentaci, která bude popisovat kritické aspekty systému, jako například databázová schémata, rozdělení prací mezi týmy atd.
2. Je nutné vyvinout a aplikovat mechanismy komunikace mezi týmy. Měly by k nim patři pravidelné telefonické a videotelefonní konference mezi členy týmu a časté krátké elektronické schůzky, kdy se jednotlivé týmy navzájem informují o aktuálním postupu. Komunikaci je vhodné zajistit formou více komunikačních kanálů, jako je e-mail, instant messaging, wiki a systémy sociálních sítí.
3. V případech, kdy je k vytvoření systému nutné integrovat několik samostatných programů, prakticky není možná průběžná integrace, kde je pokaždé sestaven celý systém a vývojář registruje změnu. Je však zásadně důležité zachovat častá sestavení systému a jeho pravidelná vydání. Z tohoto důvodu může být nutné zavést nové nástroje pro správu konfigurace, které podporují vývoj softwaru ve více týmech.

Agilní metody nejochotněji přijímají malé softwarové společnosti, které vyvíjejí softwarové produkty. Tyto firmy nejsou omezeny organizační byrokracií ani standardy procesů a mohou se rychle měnit a aplikovat nové myšlenky. Velké společnosti samozřejmě také experimentují s agilními metodami u konkrétních projektů, ale pro ně je mnohem obtížnější škálovat tyto metody do šířky v rámci celé organizace. Lindvall, et al. (2004) se zabývají některými problémy při škálování agilních metod do šířky ve čtyřech velkých technologických společnostech.

Agilní metody se ve velkých společnostech zavádějí obtížně z několika důvodů:

1. Vedoucí projektů, kteří nemají s agilními metodami zkušenosti, mohou neochotně přijímat rizika nového přístupu, protože nevědí, jak tyto metody ovlivní právě jejich projekty.
2. Velké organizace často používají postupy a standardy kvality, kterými by se měly řídit všechny projekty. Vzhledem k jejich byrokratické povaze tyto postupy často nejsou kompatibilní s agilními metodami. Někdy jsou k jejich podpoře určeny softwarové nástroje (např. nástroje na správu požadavků), jejichž použití je pro všechny projekty povinné.
3. Ukazuje se, že agilní metody fungují nejlépe, když jsou členové týmu relativně zkušení. Ve velkých organizacích však pravděpodobně pracují lidé s velmi rozmanitými znalostmi a schopnostmi. Osoby s nižší úrovní zkušeností přitom nemusí být efektivními členy týmu, který používá agilní procesy.
4. Může se vyskytovat kulturní odpor k agilním metodám, zejména v těch organizacích, které mají dlouhou historii používání konvenčních procesů systémového inženýringu.

Jako příklady podnikových postupů, které nemusí být kompatibilní s agilními metodami, lze uvést postupy správy změn a testování. Správa změn je proces kontroly změn v systému, aby bylo možné předpovídat dopad změn a kontrolovat náklady. Všechny změny je nutné před provedením předem schválit a tento princip je v rozporu s požadavkem na refaktoring. V přístupu XP může každý vývojář zlepšit libovolnou část kódu, aniž by k tomu potřeboval povolení zvnějšku. U větších systémů existují také standardy testování, kdy se sestavení systému předává externímu testovacímu týmu. Tento postup může být v rozporu s přístupy testování hned na začátku a častého testování, které se uplatňují v XP.

Zavedení a trvalé používání agilních metod v rámci velké organizace představuje proces kulturní změny. Implementace kulturních změn trvá dlouho a ke svému dokončení často vyžaduje změnu managementu. Společnosti, které si přejí nasadit agilní metody, potřebují jejich propagátory, kteří budou změny prosazovat. Na proces změny musí vyhradit značné prostředky. V době psaní této knihy pouze málo velkých společností úspěšně přešlo na agilní vývoj v celé své struktuře.

HLAVNÍ BODY

- Agilní metody patří mezi metody inkrementálního vývoje, které se soustřeďují na rychlý vývoj, častá vydání softwaru, snížení režie procesu a produkci vysoce kvalitního kódu. Zákazníka přímo zapojují do procesu vývoje.
- Rozhodnutí o tom, zda použít agilní nebo plánovaný přístup k vývoji, by mělo záviset na typu vyvíjeného softwaru, schopnostech vývojového týmu a kultuře společnosti, která systém vyvíjí.
- K nejznámějším agilním metodám patří extrémní programování. Integruje více optimálních programátorských postupů, jako jsou častá vydání softwaru, trvalé zlepšování softwaru a účast zákazníka ve vývojovém týmu.
- K silným stránkám extrémního programování patří zejména vývoj automatizovaných testů ještě před vytvořením programové funkce. Při integraci inkrementu do systému je nutné úspěšně provést všechny testy.
- Scrum je agilní metoda, která poskytuje rámec řízení projektů. Soustřeďuje se kolem sady rychlých cyklů, což jsou pevná časová období, kdy se vyvíjí inkrement systému. Plánování je založeno na stanovení priorit pracovních restů a výběru úkolů s nejvyšší prioritou, které se budou zpracovávat v nejbližším rychlém cyklu.
- Agilní metody se obtížně škálují na velké systémy. Velké systémy vyžadují předběžný návrh a neobejdou se bez dokumentace určitého typu. Průběžnou integraci prakticky nelze zajistit v případě, že na projektu souběžně pracuje několik samostatných vývojových týmů.

DALŠÍ ZDROJE INFORMACÍ

Extreme Programming Explained (Úvod do extrémního programování). Jedná se o první knihu o přístupu XP, která i nadále patří mezi nejčtivější. Vysvětluje celý přístup z perspektivy jednoho ze svých tvůrců a jeho nadšení je patrné z celé knihy. (Kent Beck, Addison-Wesley, 2000.)

„Get Ready for Agile Methods, With Care" (Opatrně se připravte na agilní metody). Promyšlená kritika agilních metod, která diskutuje jejich silné i slabé stránky. Napsal ji mimořádně zkušený softwarový inženýr. (B. Boehm, IEEE Computer, leden 2002.) *http://doi.ieeecomputersociety.org/10.1109/2.976920*.

Scaling Software Agility: Best Practices for Large Enterprises (Škálování softwarové agility: optimální postupy pro velké společnosti). Kniha se sice zaměřuje na otázky škálování agilního vývoje, ale obsahuje také souhrn klíčových agilních metod, jako je XP, Scrum a Crystal. (D. Leffingwell, Addison-Wesley, 2007.)

Running an Agile Software Development Project (Vedení projektu agilního vývoje softwaru). Většina knih o agilních metodách se soustřeďuje na konkrétní metodu, ale tato kniha volí odlišný přístup a diskutuje, jak prakticky uplatnit přístup XP ve vývojovém projektu. Obsahuje praktické a užitečné rady. (M. Holcombe, John Wiley and Sons, 2008.)

CVIČENÍ

3.1. Vysvětlete, proč je rychlé dodávání a nasazení nových systémů pro podniky často důležitější než propracované funkce těchto systémů.

3.2. Vysvětlete, jak principy v základech agilních metod umožňují rychlejší vývoj a nasazení softwaru.

3.3. Kdy byste *nedoporučili* použití agilních metod při vývoji softwarového systému?

3.4. Extrémní programování vyjadřuje uživatelské požadavky formou historií, kdy každá historie je napsána na jedné kartě. Diskutujte výhody a nevýhody tohoto přístupu k popisu požadavků.

3.5. Vysvětlete, proč vývoj s testováním hned na začátku pomáhá programátorům získat lepší přehled o systémových požadavcích. Jaké jsou potenciální potíže při vývoji s testováním hned na začátku?

3.6. Navrhněte čtyři důvody, proč může být úroveň produktivity programátorů pracujících ve dvojicích vyšší než poloviční oproti dvěma samostatně pracujícím programátorům.

3.7. Porovnejte přístup Scrum s řízením projektu pomocí klasických plánovaných přístupů, kterými se budeme zabývat v kapitole 23. Porovnání by mělo vycházet z efektivity každého přístupu k plánování alokace lidí na projekty, odhadu nákladů na projekty, zachování soudržnosti týmu a řízení změn členství v projektovém týmu.

3.8. Pracujete jako softwarový manažer ve společnosti, která vyvíjí kritický řídicí software pro letadla. Odpovídáte za vývoj systému podpory softwarového návrhu, který umožňuje převod požadavků na software do formální specifikace softwaru (popis naleznete v kapitole 13). Popište výhody a nevýhody následujících vývojových strategií:

a. Shromážděte požadavky na takový systém od softwarových inženýrů a externích zainteresovaných osob (jako je regulační certifikační orgán) a vyviňte systém pomocí plánovaného přístupu.

b. Vyviňte prototyp pomocí skriptovacího jazyka, jako je Ruby nebo Python, vyhodnoťte prototyp se softwarovými inženýry a dalšími zainteresovanými osobami a poté revidujte systémové požadavky. Výsledný systém znovu vyviňte v jazyce Java.

c. Vyviňte systém v jazyce Java pomocí agilního přístupu, kdy se uživatel bude účastnit práce vývojového týmu.

3.9. Podle některých pozorování jeden problém související s tím, že se uživatel úzce podílí na práci vývojového týmu softwaru, spočívá v tom, že zástupce uživatele „přejde na druhou stranu". To znamená, že přijme pohled vývojového týmu a přestane hájit potřeby svých kolegů uživatelů. Navrhněte tři způsoby, jak lze tomuto problému předejít, a diskutujte výhody a nevýhody každého způsobu.

3.10. Kvůli snížení nákladů a ekologických dopadů dojíždění se vaše společnost rozhodne uzavřít několik poboček a zajistit podporu pracovníků, aby mohli pracovat z domova. Vrcholné vedení, které tuto politiku zavedlo, však neví o tom, že software se vyvíjí pomocí agilních metod, které spoléhají na úzkou týmovou spolupráci a programování ve dvojicích. Diskutujte obtíže, které tato nová politika může způsobit, a navrhněte, jak by bylo možné příslušné problémy vyřešit.

CITACE

Ambler, S. W. a Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process.* (Agilní modelování: efektivní postupy pro extrémní programování a proces Unified Process). New York: John Wiley & Sons.

Arisholm, E., Gallis, H., Dyba, T. a Sjoberg, D. I. K. (2007). „Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise" (Hodnocení programování ve dvojicích s ohledem na složitost systému a zkušenosti programátorů). *IEEE Trans. on Software Eng.*, **33** (2), 65–86.

Astels, D. (2003). *Test Driven Development: A Practical Guide* (Vývoj řízený testováním: praktický průvodce). Upper Saddle River, NJ: Prentice Hall.

Beck, K. (1999). „Embracing Change with Extreme Programming" (Přijetí změn v extrémním programování). *IEEE Computer*, **32** (10), 70–8.

Beck, K. (2000). *Extreme Programming explained* (Úvod do extrémního programování). Reading, Mass.: Addison-Wesley.

Carlson, D. (2005). *Eclipse Distilled* (Přehled systému Eclipse). Boston: Addison-Wesley.

Cockburn, A. (2001). *Agile Software Development* (Agilní vývoj softwaru). Reading, Mass.: Addison-Wesley.

Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams* (Crystal: osobní metodika pro malé týmy). Boston: Addison-Wesley.

Cockburn, A. a Williams, L. (2001). „The costs and benefits of pair programming" (Náklady a výhody programování ve dvojicích). *In Extreme programming examined (Analýza extrémního programování). (ed.).* Boston: Addison-Wesley.

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum* (Úspěch s agilními metodami: vývoj softwaru přístupem Scrum). Boston: Addison-Wesley.

Demarco, T. a Boehm, B. (2002). „The Agile Methods Fray" (Boje o agilní metody). *IEEE Computer*, **35** (6), 90–2.

Denning, P. J., Gunderson, C. a Hayes-Roth, R. (2008). „Evolutionary System Development" (Evoluční vývoj systémů). *Comm. ACM*, **51** (12), 29–31.

Drobna, J., Noftz, D. a Raghu, R. (2004). „Piloting XP on Four Mission-Critical Projects" (Pilotní nasazení XP u čtyř kritických projektů). *IEEE Software*, **21** (6), 70–5.

Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems* (Adaptivní vývoj softwaru: kolaborativní přístup ke zvládání komplexních systémů). New York: Dorset House.

Hopkins, R. a Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield* (Konzumace IT slona: přechod od vývoje na zelené louce k vývoji na brownfieldu). Boston, Mass.: IBM Press.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process* (Aplikace UML a vzorů: úvod do objektově orientované analýzy a návrhu a procesu Unified Process). Englewood Cliff, NJ: Prentice Hall.

Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises* (Škálování softwarové agility: optimální postupy pro velké podniky). Boston: Addison-Wesley.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. a Kahkonen, T. (2004). „Agile Software Development in Large Organizations" (Agilní vývoj softwaru ve velkých organizacích). *IEEE Computer*, **37** (12), 26–34.

Martin, J. (1981). *Application Development Without Programmers* (Vývoj aplikací bez programátorů). Englewood Cliffs, NJ: Prentice-Hall.

Massol, V. a Husted, T. (2003). *JUnit in Action* (JUnit v praxi). Greenwich, Conn.: Manning Publications Co.

Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. a Quinnan, R. E. (1980). „The Management of Software Engineering" (Řízení softwarového inženýrství). *IBM Systems.* J., **19** (4), 414–77.

Moore, E. a Spens, J. (2008). „Scaling Agile: Finding your Agile Tribe" (Škálování agilních metod: najděte svůj agilní kmen). *Proc. Agile 2008 Conference*, Toronto: IEEE Computer Society. 121–124.

Palmer, S. R. a Felsing, J. M. (2002). A *Practical Guide to Feature-Driven Development* (Praktický průvodce do vývoje založeného na funkcích). Englewood Cliffs, NJ: Prentice Hall.

Parrish, A., Smith, R., Hale, D. a Hale, J. (2004). „A Field Study of Developer Pairs: Productivity Impacts and Implications" (Terénní studie vývojářských dvojic: dopady na produktivitu a důsledky). *IEEE Software*, **21** (5), 76–9.

Poole, C. a Huisman, J. W. (2001). „Using Extreme Programming in a Maintenance Environment" (Použití extrémního programování v prostředí údržby). *IEEE Software*, **18** (6), 42–50.

Rising, L. a Janoff, N. S. (2000). „The Scrum Software Development Process for Small Teams" (Proces vývoje softwaru Scrum pro malé týmy). *IEEE Software*, **17** (4), 26–32.

Schwaber, K. (2004). *Agile Project Management with Scrum* (Řízení agilních projektů s přístupem Scrum). Seattle: Microsoft Press.

Schwaber, K. a Beedle, M. (2001). *Agile Software Development with Scrum* (Agilní vývoj softwaru přístupem Scrum). Englewood Cliffs, NJ: Prentice Hall.

Smits, H. a Pshigoda, G. (2007). „Implementing Scrum in a Distributed Software Development Organization" (Implementace přístupu Scrum v distribuované organizaci vyvíjející software). Agile 2007, Washington, DC: IEEE Computer Society.

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method* (Dynamická metoda vývoje systémů DSDM). Harlow, UK: Addison-Wesley.

Stapleton, J. (2003). *DSDM: Business Focused Development, 2nd ed.* (DSDM: obchodně zaměřený vývoj), 2. vyd. Harlow, UK: Pearson Education.

Stephens, M. a Rosenberg, D. (2003). *Extreme Programming Refactored* (Refaktoring extrémního programování). Berkley, Calif.: Apress.

Sutherland, J., Viktorov, A., Blount, J. a Puntikov, N. (2007). „Distributed Scrum: Agile Project Management with Outsourced Development Teams" (Distribuovaný přístup Scrum: řízení agilních projektů s outsourcovanými vývojovými týmy). 40th Hawaii Int. Conf. on System Sciences, Hawaii: IEEE Computer Society.

Weinberg, G. (1971). *The Psychology of Computer Programming* (Psychologie programování počítačů). New York: Van Nostrand.

Williams, L., Kessler, R. R., Cunningham, W. a Jeffries, R. (2000). „Strengthening the Case for Pair Programming" (Argumenty ve prospěch programování ve dvojicích). *IEEE Software*, **17** (4), 19–25.

Inženýrství požadavků

4

Cíle

V této kapitole představíme koncepci požadavků na software a rozebereme procesy, které se při zjišťování a dokumentaci těchto požadavků uplatňují. V této kapitole:

- porozumíte principu uživatelských a systémových požadavků a tomu, proč je vhodné tyto typy požadavků zapisovat odlišně,
- seznámíte se s rozdíly mezi funkčními a mimofunkčními požadavky na software,
- dozvíte se, jak lze požadavky uspořádat do dokumentu požadavků na software,
- naučíte se pracovat se základními aktivitami inženýrství požadavků: zjišťováním, analýzou a validací, a porozumíte rozdílům těchto aktivit,
- pochopíte, proč je nezbytná správa požadavků a jak podporuje jiné aktivity inženýrství požadavků.

Požadavky na systém popisují, co má systém dělat – jaké služby poskytuje a jaká jsou omezení jeho činnosti. Tyto požadavky odrážejí potřeby zákazníka na systém, který bude sloužit určitému účelu, například řízení jistého zařízení, zadávání objednávek nebo vyhledávání informací. Proces zjišťování, analýzy, dokumentace a kontroly těchto služeb a omezení je označován jako inženýrství požadavků (RE – requirements engineering).

Termín „požadavek“ se v softwarovém oboru nepoužívá konzistentně. V některých případech je požadavek pouze vysokoúrovňový abstraktní popis služby, kterou by měl systém poskytovat, nebo omezení systému. Druhý extrém představuje podrobná a formální definice systémové funkce. Proč tyto rozdíly existují, vysvětluje Davis (1993):

> *Jestliže chce společnost zadat velký projekt na vývoj softwaru, musí své potřeby definovat dostatečně abstraktním způsobem, aby předem neomezovala způsob řešení. Požadavky je nutné zapisovat tak, aby se mohlo o zakázku ucházet několik kontraktorů, kteří mohou teoreticky nabízet odlišné způsoby, jak potřeby klientské organizace uspokojit. Po udělení kontraktu musí kontraktor pro klienta napsat podrobnější definici systému, aby mu klient porozuměl a dokázal funkce softwaru validovat. Oba tyto dokumenty lze označovat jako dokumenty požadavků na systém.*

Některé problémy, které se vyskytují v rámci procesu inženýrství požadavků, vyplývají z toho, že se tyto odlišné úrovně popisu dostatečně neoddělují. V této knize mezi nimi budeme rozlišovat tak, že termínem „uživatelské požadavky" označíme vysokoúrovňové abstraktní požadavky a termínem „systémové požadavky" budeme mít na mysli podrobný popis toho, co má systém dělat. Uživatelské požadavky a systémové požadavky lze definovat takto:

1. Uživatelské požadavky jsou věty přirozeného jazyka doplněné diagramy, které popisují, jaké služby se od systému očekávají a za jakých omezení musí fungovat.
2. Systémové požadavky podrobněji popisují funkce, služby a provozní omezení softwarového systému. Dokument se systémovými požadavky (někdy označovaný jako funkční specifikace) by měl přesně definovat, co se bude implementovat. Může být součástí smlouvy mezi zákazníkem systému a softwarovými vývojáři.

Je vhodné využít různé úrovně požadavků, protože poskytují informace o systému různému typu čtenářů. Obrázek 4.1 ilustruje rozdíly mezi uživatelskými a systémovými požadavky. Tento příklad ze systému správy záznamů o psychiatrických pacientech (MHC-PMS – Mental Health Care-Patient Management System) znázorňuje, jak lze uživatelské požadavky rozšířit do několika systémových požadavků. Z obrázku 4.1 je patrné, že uživatelské požadavky jsou poměrně obecné. Systémové požadavky poskytují podrobnější informace o službách a funkcích systému, které mají být implementovány.

Definice uživatelských požadavků

1. Systém správy záznamů o psychiatrických pacientech bude měsíčně generovat sestavy pro management, které budou uvádět náklady na léky předepsané v každé klinice za daný měsíc.

Specifikace systémových požadavků

1.1 Poslední pracovní den každého měsíce bude generován souhrn předepsaných léků, jejich cen a předepisujících klinik.

1.2 Systém bude automaticky generovat sestavu k tisku po 17.30 posledního pracovního dne v měsíci.

1.3 Pro každou kliniku se bude vytvářet sestava, která bude obsahovat názvy jednotlivých léků, celkový počet receptů, počet předepsaných dávek a celkové náklady na předepsané léky.

1.4 Pokud jsou léky dostupné v různém dávkování (např. 10 mg, 20 mg), budou pro každou dávku léku vytvořeny samostatné sestavy.

1.5 Přístup ke všem nákladovým sestavám bude omezen na autorizované uživatele, kteří jsou uvedeni na seznamu řízení přístupu managementu.

Obrázek 4.1 – Uživatelské a systémové požadavky

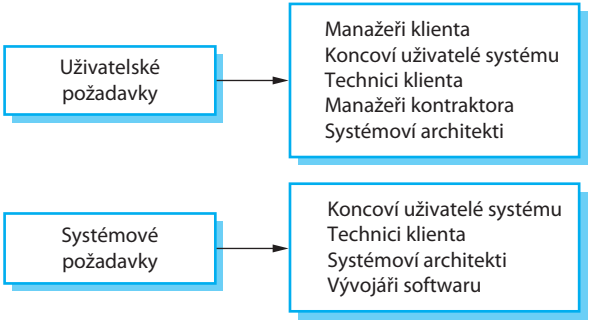
Požadavky je nutné uvádět s různými úrovněmi podrobností, protože různí čtenáři je budou používat různými způsoby. Obrázek 4.2 znázorňuje možné čtenáře uživatelských a systémových požadavků. Čtenáři uživatelských požadavků se obvykle nezabývají tím, jak bude systém implementován. Může se jednat o manažery, kteří se nezajímají o podrobnosti funkcí systému. Čtenáři systémových požadavků potřebují znát funkce systému přesněji, protože je pro ně důležité, jak bude systém podporovat podnikové procesy, nebo protože se podílejí na jeho implementaci.

V této kapitole představíme „tradiční“ pohled na požadavky, nikoli požadavky v agilních procesech. U většiny velkých systémů stále platí, že implementaci systému předchází jednoznačně identifikovaná fáze inženýrství požadavků. Výsledkem je dokument požadavků, který může být součástí smlouvy na vývoj systému. Samozřejmě platí, že požadavky se obvykle později mění a uživatelské požadavky lze rozšířit na podrobnější systémové požadavky. Agilní přístup, který získává požadavky souběžně s vývojem systému, se však při vývoji velkých systémů používá jen zřídka.

4.1 Funkční a mimofunkční požadavky

Systémové požadavky na software se často dělí na funkční a mimofunkční požadavky:

1. *Funkční požadavky* – jedná se o popis služeb, které by měl systém poskytovat, reakce systému na určité vstupy a chování systému v konkrétních situacích. V některých případech mohou funkční požadavky také explicitně určovat, co systém nesmí provést.



Obrázek 4.2 – Čtenáři různých typů specifikace požadavků

2. *Mimofunkční požadavky* – jsou to omezení služeb či funkcí, které systém poskytuje. Patří sem omezení časování, omezení vývojového procesu a omezení založená na standardech. Mimofunkční požadavky se často týkají systému jako celku a nikoli jeho jednotlivých funkcí či služeb.

Rozdíly mezi jednotlivými typy požadavků ve skutečnosti nejsou tak jednoznačné, jak by se zdálo podle těchto jednoduchých definic. Uživatelský požadavek týkající se zabezpečení, jako je omezení přístupu na autorizované uživatele, může vypadat jako mimofunkční požadavek. Když se však podrobněji rozvine, může tento požadavek generovat další požadavky, které jsou jednoznačně funkční, jako například nutnost zahrnout do systému funkce uživatelské autentizace.

Z toho je patrné, že požadavky nejsou nezávislé a jeden požadavek často generuje nebo omezuje jiné požadavky. Systémové požadavky tedy pouze neurčují požadované služby či funkce systému, ale specifikují také nezbytné vlastnosti, které zajišťují správné poskytování těchto služeb či funkcí.

4.1.1 Funkční požadavky

Funkční požadavky na systém popisují, co má systém provádět. Tyto požadavky závisejí na typu vyvíjeného softwaru, jeho předpokládaných uživatelích a obecném přístupu organizace, která požadavky sepisuje. Při vyjádření v podobě uživatelských požadavků se funkční požadavky obvykle uvádějí abstraktním způsobem, který je srozumitelný pro uživatele systému. Konkrétnější funkční systémové požadavky však podrobně popisují funkce systému, jeho vstupy a výstupy, výjimky atd.

Funkční systémové požadavky mohou sahat od obecných požadavků, které uvádějí, co má systém dělat, až po velmi specifické požadavky, které se týkají lokálních pracovních metod nebo stávajících systémů v organizaci. Uveďme například ukázky funkčních požadavků na systém MHC-PMS, který slouží k ukládání informací o pacientech léčených kvůli psychiatrickým problémům:

DOMÉNOVÉ POŽADAVKY

Doménové požadavky se odvozují z aplikační domény systému a nikoli z konkrétních potřeb systémových uživatelů. Může se jednat o funkční požadavky samy o sobě či omezení na stávající funkční požadavky, případně mohou určovat, jak lze provádět jisté výpočty.

Problém s doménovými požadavky spočívá v tom, že softwaroví inženýři nemusí rozumět charakteristikám domény, kde systém funguje. Často nedokážou určit, zda nebyl doménový požadavek vynechán nebo zda není v konfliktu s jinými požadavky.

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

1. Uživatel bude mít možnost prohledávání seznamu schůzek na všech klinikách.
2. Systém bude každý den pro každou kliniku generovat seznam pacientů, kteří jsou na příslušní den objednáni na schůzku.
3. Každý pracovník používající systém bude jednoznačně identifikován pomocí svého osmimístného čísla zaměstnance.

Tyto funkční uživatelské požadavky definují konkrétní funkce poskytované systémem. Pocházejí z dokumentu uživatelských požadavků a dokládají, že funkční požadavky lze zapisovat s různými úrovněmi podrobností (porovnejte požadavky 1 a 3).

Nepřesnosti ve specifikaci požadavků jsou příčinou mnoha problémů softwarového inženýrství. Je přirozené, že vývojáři systému interpretují nejednoznačné požadavky způsobem, kterým se implementace zjednodušuje. Tento postup však často neodpovídá tomu, co zákazník chce. Je nutné stanovit nové požadavky a provést změny systému. Tím se pochopitelně zdržuje dodání systému a zvyšují náklady.

První ukázkový požadavek na systém MHC-PMS například udává, že uživatel bude mít možnost prohledávání seznamu schůzek na všech klinikách. Důvod tohoto požadavku spočívá v tom, že pacienti s duševními problémy bývají někdy zmatení. Mohou se objednat v jedné klinice, ale ve skutečnosti navštívit jinou. Pokud mají schůzku, zaznamená se u nich, že se dostavili, bez ohledu na konkrétní kliniku.

Člen zdravotnického týmu, který tento požadavek zadal, může mít „prohledáváním" na mysli, že po zadání jména pacienta systém vyhledá toto jméno ve všech schůzkách na všech klinikách. To však z požadavků přímo nevyplývá. Vývojáři systému mohou tento požadavek interpretovat odlišným způsobem a mohou vyhledávání implementovat tak, že uživatel musí před vlastním vyhledáváním nejdříve vybrat určitou kliniku. Tento postup by od uživatele samozřejmě vyžadoval více zadávání, a trval by tedy déle.

Specifikace funkčních požadavků na systém by v zásadě měla být zároveň úplná i konzistentní. Úplnost znamená, že jsou definovány všechny služby, které uživatel požaduje. Konzistence znamená, že definice požadavků by si neměly vzájemně odporovat. V praxi je u velkých a složitých systémů prakticky nemožné této konzistence a úplnosti požadavků dosáhnout. Jeden důvod spočívá v tom, že při psaní specifikace komplexních systémů je snadné dopustit se chyb a opomenutí. Dalším důvodem je to, že na velkém systému je zainteresováno mnoho osob. Zainteresovaná osoba je konkrétní osoba nebo role,

kterou systém nějakým způsobem ovlivňuje. Zainteresované osoby mají odlišné a často protichůdné potřeby. Tyto inkonzistence nemusí být při první specifikaci požadavků zjevné, takže se tyto nekonzistentní požadavky dostanou do specifikace. Problémy se mohou objevit teprve po hlubší analýze nebo poté, co je systém dodán zákazníkovi.

4.1.2 Mimofunkční požadavky

Mimofunkční požadavky, jak vyplývá z jejich názvu, se přímo netýkají konkrétních služeb, které systém poskytuje svým uživatelům. Mohou se vztahovat na emergentní vlastnosti systému, jako je spolehlivost, doba odezvy a vytíženost. Alternativně mohou definovat omezení na implementaci systému, jako například možnosti vstupně-výstupních zařízení nebo reprezentace dat, které se používají v rozhraních s jinými systémy.

Mimofunkční požadavky, jako je výkon, zabezpečení nebo dostupnost, obvykle určují či omezují vlastnosti systému jako celku. Mimofunkční požadavky bývají často kritičtější než jednotlivé funkční požadavky. Uživatelé systému mohou obvykle najít způsoby, jak obejít funkce systému, které v praxi nevyhovují jejich požadavkům. Jestliže však vývoj nerespektuje mimofunkční požadavek, může to znamenat, že nebude použitelný celý systém. Pokud například systém letadla nevyhovuje požadavky na spolehlivost, nebude certifikován jako bezpečný k provozu. Nesplní-li integrovaný řídicí systém své výkonnostní požadavky, nebudou jeho řídicí funkce pracovat správně.

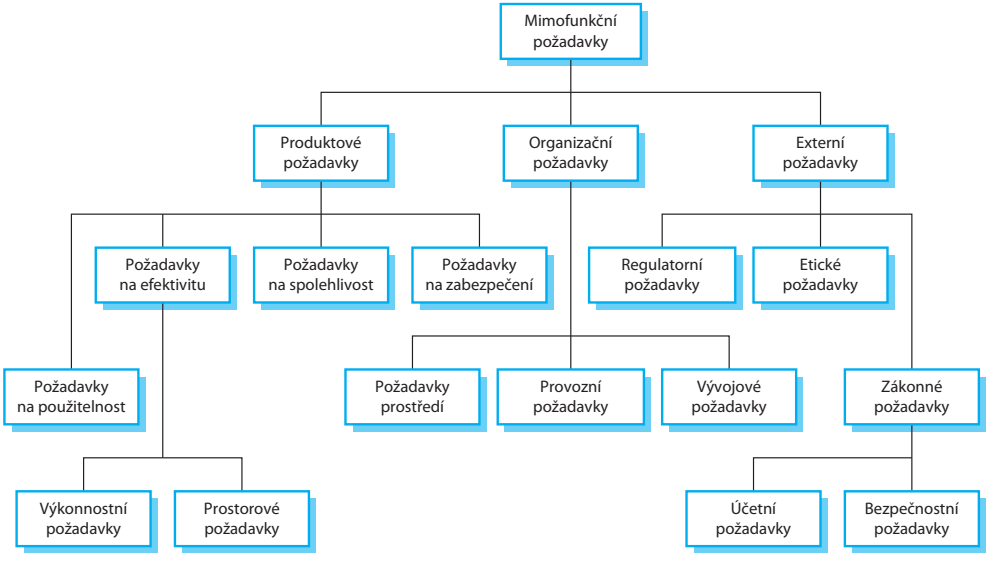
Často je sice možné určit, které systémové komponenty implementují jistý funkční požadavek (například mohou existovat formátovací komponenty, které implementují požadavky na vykazování), ale najít komponenty související s mimofunkčními požadavky bývá často obtížnější. Implementace těchto požadavků může být rozptýlena po celém systému. Je to způsobeno dvěma důvody:

1. Mimofunkční požadavky mohou ovlivnit celkovou architekturu systému namísto jeho jednotlivých komponent. Chceme-li například zajistit, že budou splněny výkonnostní požadavky, může být nutné uspořádat systém tak, aby se minimalizovala komunikace mezi komponentami.
2. Jediný mimofunkční požadavek (například ohledně zabezpečení) může generovat více souvisejících funkčních požadavků, které definují nově potřebné systémové funkce. Kromě toho se mohou také objevit požadavky, které kladou omezení na stávající požadavky.

Mimofunkční požadavky vyplývají z potřeb uživatelů, rozpočtových omezení, organizačních zásad, potřeby interoperability s jinými softwarovými či hardwarovými systémy nebo z externích faktorů, jako jsou bezpečnostní předpisy nebo zákony na ochranu osobních údajů. Obrázek 4.3 představuje klasifikaci mimofunkčních požadavků. Z tohoto diagramu je patrné, že mimofunkční požadavky mohou pocházet z požadovaných vlastností softwaru (produktové požadavky), od organizace vyvíjející software (organizační požadavky) nebo z vnějších zdrojů:

1. *Produktové požadavky* – tyto požadavky určují nebo omezují chování softwaru. Jako příklady lze uvést výkonnostní požadavky týkající se rychlosti práce systému a jeho spotřeby paměti, požadavky na spolehlivost, které stanovují přijatelnou četnost chyb, požadavky na zabezpečení a požadavky na použitelnost.
2. *Organizační požadavky* – do této kategorie patří obecné systémové požadavky, které se odvozují od zásad a postupů v organizaci zákazníka a vývojáře. K příkladům patří požadavky na provozní procesy, které definují, jak se bude systém používat, požadavky na proces vývoje, které určují programovací jazyk, vývojové prostředí nebo použité procesní standardy, a požadavky prostředí, které definují operační prostředí systému.

3. *Externí požadavky* – tato široká třída zahrnuje všechny požadavky, které jsou odvozeny od faktorů, které leží vně systému a procesu jeho vývoje. Může se jednat o požadavky oborových předpisů, které stanovují, jak musí systém fungovat, aby jeho použití schválil regulátor (například centrální banka), zákonné požadavky, které je nutné dodržet, aby systém fungoval v souladu se zákony, a etické požadavky, které zajišťují, že bude systém přijatelný pro své uživatele a jiné občany.



Obrázek 4.3 – Typy mimofunkčních požadavků

PRODUKTOVÉ POŽADAVKY

Systém MHC-PMS bude ve všech klinikách dostupný během normální pracovní doby (po–pá, od 08.30 do 17.30). Odstávky v rámci normální pracovní doby nepřekročí během libovolného dne pět sekund.

ORGANIZAČNÍ POŽADAVKY

Uživatelé systému MHC-PMS se budou autentizovat pomocí svých služebních identifikačních karet.

EXTERNÍ POŽADAVKY

Systém bude implementovat požadavky předpisu HStan-03-2006-priv na ochranu osobních údajů pacientů.

Obrázek 4.4 – Příklad mimofunkčních požadavků na systém MHC-PMS

Obrázek 4.4 představuje příklady produktových, organizačních a externích požadavků na systém MHC-PMS, jehož uživatelské požadavky jsme představili v části 4.1.1. Produktové požadavky se týkají dostupnosti systému a definují, kdy bude systém k dispozici a jaká je přípustná doba výpadků během jednoho dne. Nijak nepopisují funkčnost systému MHC-PMS a jednoznačně identifikují omezení, která musí návrháři systému zohlednit.

Organizační požadavky specifikují, jak se budou autentizovat uživatelé systému. Zdravotnická organizace, která systém provozuje, přechází pro veškerý software na standardní proceduru autentizace. Místo

aby uživatelé zadávali přihlašovací jména a hesla, identifikují se protažením své identifikační karty čtečkou. Externí požadavky se odvozují z požadavku, aby systém vyhovoval zákonům na ochranu soukromí. Jedná se pochopitelně o velmi důležitý aspekt systémů zdravotní péče. Požadavky určují, že při vývoji systému je nutné dodržet národní standardy na ochranu osobních údajů.

Běžný problém s mimofunkčními požadavky spočívá v tom, že uživatelé nebo zákazníci tyto požadavky často navrhují jako obecné cíle – například snadné používání, schopnost systému zotavit se ze selhání nebo rychlá odezva na uživatelské vstupy. Tyto cíle jsou sice míněny dobře, ale mohou vývojářům systému způsobit problémy. Ponechávají totiž prostor pro různou interpretaci a později při dodání systému může docházet ke sporům. Následující systémový cíl je například typickou ukázkou toho, jak může požadavky na použitelnost vyjádřit manažer.

> *Systém bude snadno použitelný pro zdravotníky a bude uspořádán takovým způsobem, který minimalizuje výskyt uživatelských chyb.*

Toto zadání můžeme přepsat tak, aby vyjadřovalo „testovatelný" mimofunkční požadavek. Systémový cíl není možné objektivně verifikovat, ale do následujícího popisu lze alespoň zahrnout softwarové nástroje na počítání chyb, kterých se uživatelé při testování systému budou dopouštět.

> *Zdravotníci dokážou používat všechny funkce systému po čtyřech hodinách školení. Průměrný počet chyb, kterých se po tomto školení dopustí zkušení uživatelé, nepřekročí dvě chyby za hodinu práce se systémem.*

Kdykoli je to možné, měli bychom mimofunkční požadavky zapisovat kvantitativně, aby je bylo možné objektivně testovat. Obrázek 4.5 představuje metriky, pomocí nichž lze specifikovat mimofunkční vlastnosti systému. Tyto vlastnosti je možné měřit při testování systému a zkontrolovat, zda systém své mimofunkční požadavky splňuje.

Vlastnost	Metrika
Rychlost	Zpracované transakce za sekundu Doba odezvy pro uživatele nebo událost Čas aktualizace obrazovky
Velikost	Megabajty Počet čipů paměti ROM
Snadné použití	Délka školení Počet obrazovek nápovědy
Spolehlivost	Průměrná doba mezi chybami Pravděpodobnost nedostupnosti Frekvence výskytu chyb Dostupnost
Robustnost	Čas restartu po chybě Procento událostí způsobujících chybu Pravděpodobnost poškození dat při chybě
Přenositelnost	Procento příkazů závislých na cílovém systému Počet cílových systémů

Obrázek 4.5 – Metriky ke specifikování mimofunkčních požadavků

Zákazníci systému mají v praxi často problémy převést své cíle do měřitelných požadavků. U některých cílů, jako je možnost správy, ani použitelné metriky neexistují. V jiných případech je sice kvantitativní specifikace možná, ale zákazníci nemusí být schopni své potřeby příslušnou formou vyjádřit. Nerozumí tomu, co například z hlediska jejich každodenní práce s výpočetními systémy znamená určité číslo definující požadovanou spolehlivost. Kromě toho mohou být náklady na objektivní verifikaci měřitelných mimofunkčních požadavků velmi vysoké a zákazníci, kteří za systém platí, mohou o oprávněnosti těchto dodatečných nákladů pochybovat.

Mimofunkční požadavky často bývají v konfliktu a různě interagují s jinými funkčními či mimofunkčními požadavky. Požadavek na autentizaci z obrázku 4.4 například zjevně vyžaduje, aby byla u každého počítače připojeného do systému nainstalována čtečka karet. Jiný požadavek však může určovat mobilní přístup k systému z notebooků lékařů nebo sester. Tyto přenosné počítače obvykle nebývají čtečkami karet vybaveny. Za těchto okolností tedy může být povolena alternativní metoda autentizace.

V praxi je obtížné v dokumentu požadavků oddělit funkční a mimofunkční požadavky. Jestliže jsou mimofunkční požadavky uvedeny odděleně od funkčních požadavků, je někdy obtížné porozumět tomu, jak spolu souvisejí. Měli bychom však explicitně zdůraznit požadavky, které jasně souvisejí s emergentními vlastnostmi systému, jako je výkon či spolehlivost. Můžeme to provést tak, že je umístíme do samostatné části dokumentu požadavků nebo je nějakým způsobem odlišíme od jiných systémových požadavků.

STANDARDY DOKUMENTŮ POŽADAVKŮ

Mnoho velkých organizací, jako je americké ministerstvo obrany a organizace IEEE, definovalo své standardy na dokumenty požadavků. Tyto standardy obvykle bývají velmi obecné, ale přesto jsou užitečné jako základ pro vývoj podrobnějších organizačních standardů. Americká organizace IEEE (Institute of Electrical and Electronic Engineers) patří mezi nejznámější tvůrce standardů. Vyvinula standard, který definuje strukturu dokumentů požadavků. Tento standard je nejvhodnější pro systémy typu systému vojenského velení a kontroly, které mají dlouhou životnost a obvykle je vyvíjí skupina organizací.

http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html

Mimofunkční požadavky, jako je spolehlivost, bezpečnost a důvěrnost, jsou zvláště důležité v kritických systémech. Těmito požadavky se budeme zabývat v kapitole 12, kde popíšeme konkrétní techniky na specifikaci požadavků na spolehlivost a zabezpečení.

4.2 Dokument požadavků na software

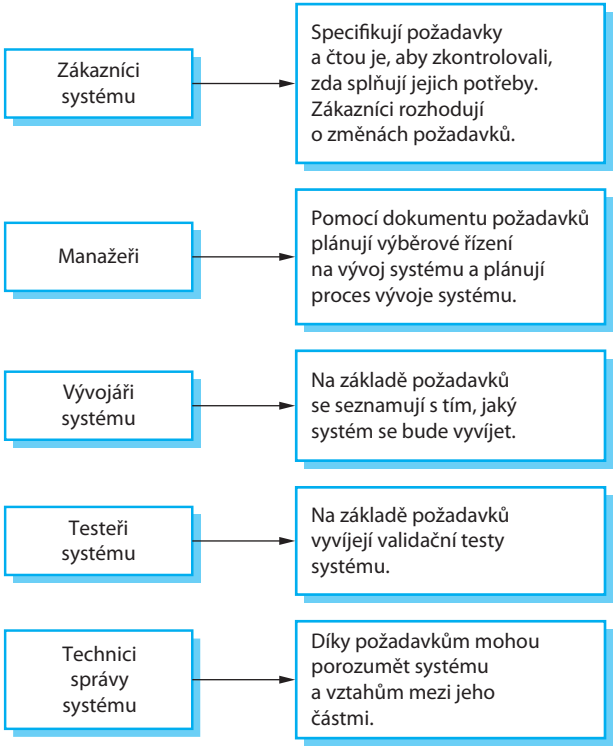
Dokument požadavků na software (někdy se označuje jako specifikace požadavků na software) je oficiální přehled toho, co mají vývojáři softwaru implementovat. Měl by zahrnovat jak uživatelské požadavky na systém, tak podrobnou specifikaci systémových požadavků. Uživatelské a systémové požadavky se někdy shrnují do jediného popisu. V jiných případech se uživatelské požadavky definují v úvodu ke specifikaci systémových požadavků. Pokud je požadavků mnoho, mohou být podrobné systémové požadavky umístěny do samostatného dokumentu.

Dokumenty požadavků jsou nezbytné, když softwarový systém vyvíjí externí kontraktor. Zastánci agilních vývojových metod však upozorňují, že požadavky se mění tak rychle, že dokument požadavků je zastaralý ihned po svém vzniku, takže úsilí na jeho tvorbu je z větší části zbytečné. Přístupy jako extrémní programování (Beck, 1999) místo formálního dokumentu shromažďují uživatelské požadavky

inkrementálně a zapisují je na kartičky jako uživatelské historie. Uživatel pak určuje prioritu požadavků k implementaci v dalším inkrementu systému.

U podnikových systémů s málo stabilními požadavky se podle názoru autora jedná o vhodný přístup. Přesto je však i nadále vhodné sepsat krátký pomocný dokument, který definuje podnikové požadavky a požadavky na spolehlivost systému. Pokud se totiž pozornost soustřeďuje na funkční požadavky následujícího vydání systému, lze snadno ztratit ze zřetele požadavky, které se týkají systému jako celku.

Dokument požadavků má různorodou skupinu uživatelů – od vrcholového vedení organizace, která vývoj systému platí, až po programátory, kteří odpovídají za vývoj softwaru. Obrázek 4.6, který pochází z knihy o inženýrství požadavků (Kotonya a Sommerville, 1998), znázorňuje možné uživatele dokumentů a způsob, jakým s dokumentem pracují.



Obrázek 4.6 – Uživatelé dokumentu požadavků

Kvůli rozmanitosti možných uživatelů musí být dokument požadavků kompromisem mezi informováním zákazníků o požadavcích, podrobným definováním požadavků pro vývojáře a testery a zahrnutím informací o možné evoluci systému. Informace o předpokládaných změnách mohou vývojářům systému pomoci, aby se vyhnuli restriktivním vývojovým rozhodnutím, a usnadnit práci technikům údržby systému, kteří potřebují přizpůsobit systém novým požadavkům.

Úroveň podrobností, které je vhodné zahrnout do dokumentu požadavků, závisí na typu vyvíjeného systému a použitém vývojovém procesu. Kritické systémy musí mít podrobné požadavky, protože otázky bezpečnosti a zabezpečení je nutné analyzovat do detailů. Když systém vyvíjí samostatná firma (například při outsourcingu), musí být specifikace systému podrobná a přesná. Jestliže se uplatňuje

interní iterativní proces vývoje, je dokument požadavků mnohem méně podrobný a případné nejasnosti lze vyřešit během vývoje systému.

Obrázek 4.7 představuje jednu možnou podobu dokumentu požadavků, která je založena na standardu IEEE pro tyto dokumenty (IEEE, 1998). Jedná se o obecný standard, který lze přizpůsobit pro konkrétní nasazení. V tomto případě jsme standard rozšířili tak, aby dokument obsahoval informace o odhadované evoluci systému. Tyto informace pomáhají při údržbě systému a umožňují návrhářům zahrnout podporu budoucích vydání systému.

Údaje obsažené v dokumentu požadavků samozřejmě závisí na typu vyvíjeného softwaru a použitém přístupu k vývoji. Jestliže se při vývoji softwarového produktu uplatňuje například evoluční přístup, nebude dokument požadavků obsahovat mnohé podrobné kapitoly, které jsme uvedli výše. Zaměří se na definování uživatelských požadavků a vysokoúrovňových mimofunkčních systémových požadavků. V tomto případě se návrháři a programátoři podle svého uvážení rozhodují, jak rámcové uživatelské požadavky na systém splnit.

Kapitola	Popis
Předmluva	Tato část definuje, komu je dokument určen, a popisuje historii jeho verzí spolu s důvody vytvoření nové verze a souhrnem změn, ke kterým v každé verzi došlo.
Úvod	Tato část popisuje požadavky na systém. Stručně shrnuje funkce systému a vysvětluje, jak budou spolupracovat s jinými systémy. Popisuje také, jak systém zapadá do celkových podnikových nebo strategických cílů organizace, která software pořizuje.
Slovníček pojmů	Definuje technické termíny, které se v dokumentu vyskytují. Není vhodné předem předpokládat, že čtenáři dokumentu budou na určité úrovni zkušeností nebo znalostí.
Definice uživatelských požadavků	Zde popisujeme služby, které uživateli poskytujeme. V této části je vhodné popsat i mimofunkční systémové požadavky. K popisu lze použít přirozeného jazyka, diagramy nebo jiné typy zápisu, kterým zákazníci rozumějí. Je potřeba specifikovat standardy produktů a procesů, které se na software vztahují.
Architektura systému	Tato kapitola předkládá celkový přehled předpokládané architektury systému a znázorňuje distribuci funkcí mezi systémovými moduly. Je potřeba zvýraznit komponenty architektury, které se budou používat opakovaně.
Specifikace systémových požadavků	Tato část by měla podrobněji popisovat funkční a mimofunkční požadavky. Je-li to nutné, lze také k mimofunkčním požadavkům doplnit další podrobnosti. Případně mohou být definována rozhraní k jiným systémům.
Modely systému	Může se jednat o grafické modely systému, které znázorňují vztahy mezi komponentami systému, systémem a jeho prostředím. Jako příklady možných modelů lze uvést objektové modely, modely datového toku nebo sémantické datové modely.
Evoluce systému	Tato část uvádí základní předpoklady, na kterých je systém založen, a případné předpokládané změny s ohledem na evoluci hardwaru, měnící se potřeby uživatelů atd. Tato část je užitečná pro systémové návrháře, protože se díky ní mohou při návrhu vyhnout rozhodnutím, která by zkomplikovala budoucí změny systému.
Přílohy	Přílohy poskytují podrobné a konkrétní informace týkající se vyvíjené aplikace, například popis hardwaru a databáze. Hardwarové požadavky definují minimální a optimální konfigurace pro práci se systémem. Databázové požadavky definují logickou organizaci dat, která systém používá, a vztahy mezi datovými položkami.
Rejstřík	Dokument lze doplnit několika rejstříky. Spolu s normálním abecedním rejstříkem se může jednat o rejstřík diagramů, funkcí atd.

Obrázek 4.7 – Struktura dokumentu požadavků

Když je však software součástí projektu velkého systému, který zahrnuje interakci hardwaru a softwarových systémů, je zpravidla nutné definovat požadavky velmi podrobně. To znamená, že dokument požadavků pravděpodobně bude velmi dlouhý a bude obsahovat většinu, nebo dokonce všechny kapitoly uvedené na obrázku 4.7. Dlouhé dokumenty je zvláště důležité doplnit podrobným obsahem a rejstříkem, aby mohli čtenáři najít informace, které potřebují.

PROBLÉMY PŘI POUŽITÍ PŘIROZENÉHO JAZYKA KE SPECIFIKACI POŽADAVKŮ

Pružnost přirozeného jazyka, která je při specifikaci velmi výhodná, často způsobuje potíže. Poskytuje prostor k psaní nejasných požadavků a čtenáři (návrháři) mohou tyto požadavky interpretovat chybně, protože mají jiné zkušenosti než uživatelé. Několik požadavků je snadno možné shrnout do jediné věty a uspořádání požadavků v přirozeném jazyce může být obtížné.

http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html

4.3 Specifikace požadavků

Proces specifikace požadavků spočívá v sepisování uživatelských a systémových požadavků do dokumentu požadavků. Uživatelské a systémové požadavky by měly být v ideálním případě jasné, jednoznačné, snadno srozumitelné, úplné a konzistentní. V praxi se tohoto stavu dosahuje těžko, protože zainteresované osoby interpretují požadavky různými způsoby a často se v požadavcích nelze vyhnout konfliktům a inkonzistencím.

Uživatelské požadavky na systém popisují funkční a mimofunkční požadavky tak, aby jim rozuměli uživatelé systému bez hlubších technických znalostí. Tyto požadavky by optimálně měly specifikovat pouze vnější chování systému. Dokument požadavků by neměl zahrnovat podrobnosti o architektuře nebo návrhu systému. Při psaní uživatelských požadavků bychom tedy neměli používat programátorský žargon nebo strukturovaný či formální zápis. Uživatelské požadavky je vhodné zaznamenat přirozeným jazykem spolu s jednoduchými tabulkami, formuláři a intuitivními diagramy.

Systémové požadavky představují rozšířenou verzi uživatelských požadavků. Používají je softwaroví inženýři jako výchozí bod návrhu systému. Tyto požadavky doplňují další podrobnosti a vysvětlují, jakým způsobem by měl systém splnit uživatelské požadavky. Mohou tvořit součást smlouvy na implementaci systému a mělo by se proto jednat o úplnou a podrobnou specifikaci celého systému.

V ideálním případě by systémové požadavky měly popisovat pouze vnější chování systému a jeho provozní omezení. Neměly by se zabývat tím, jak má být systém navržen či implementován. Na úrovni podrobností, jakou vyžaduje úplná specifikace komplexního softwarového systému, je však téměř nemožné všechny návrhové informace vyloučit. Je to způsobeno několika důvody:

1. Někdy je nutné navrhnout počáteční architekturu systému, na níž lze založit strukturu specifikace požadavků. Systémové požadavky jsou uspořádány podle různých podsystémů, ze kterých se systém skládá. Jak rozebereme v kapitolách 6 a 18, tato definice architektury má klíčovou roli, chceme-li při implementaci systému opakovaně používat softwarové komponenty.

Zápis	Popis
Věty přirozeného jazyka	Požadavky se zapisují pomocí číslovaných vět přirozeného jazyka. Každá věta by měla vyjadřovat jeden požadavek.
Strukturovaný přirozený jazyk	Požadavky se zapisují v přirozeném jazyce pomocí standardní formy nebo šablony. Každé pole poskytuje informace o jednom aspektu požadavků.
Jazyky popisu návrhu	Tento přístup specifikuje požadavky pomocí jazyka podobného programovacímu jazyku, který má ovšem abstraktnější prvky. Přitom definuje provozní model systému. Uvedený přístup se nyní používá jen zřídka, ačkoli může být užitečný při specifikaci rozhraní.
Grafické znázornění	Grafické modely doplněné textovými poznámkami umožňují definovat funkční požadavky na systém. Běžně se uplatňují případy použití UML a sekvenční diagramy.
Matematické specifikace	Tyto zápisy jsou založeny na matematických principech, jako jsou stroje nebo množiny s konečným počtem stavů. Tyto jednoznačné specifikace sice mohou omezit nejasnosti v dokumentu požadavků, ale zákazníci většinou formálním specifikacím nerozumějí. Nedokážou zkontrolovat, zda odpovídají jejich požadavkům, a nejsou ochotni je přijmout jako součást smlouvy o dodávce systému.

Obrázek 4.8 – Způsoby psaní specifikace systémových požadavků

2. Systémy musí ve většině případů spolupracovat s již existujícími systémy, což klade na návrh nového systému omezení a vynucuje nové požadavky.
3. Někdy je nutné použít určitou architekturu, aby bylo možné vyhovět mimofunkčním požadavkům (jedná se například o programování N verzí kvůli dosažení spolehlivosti – viz kapitola 13). Externí regulátor, který musí certifikovat bezpečnost systému, může určit, že je nutné zvolit již certifikovaný návrh architektury.

Uživatelské požadavky se téměř vždy zapisují přirozeným jazykem. Text v dokumentu požadavků přitom doplňují příslušné diagramy a tabulky. Také systémové požadavky lze uvádět pomocí přirozeného jazyka, ale lze použít také jiné formy zápisu založené na formulářích, grafických modelech systému nebo matematických modelech systému. Obrázek 4.8 shrnuje možné zápisy, které lze při psaní systémových požadavků zvolit.

Grafické modely jsou nejužitečnější v situacích, kdy je potřeba znázornit, jak se mění stav, nebo je nutné popsat sekvenci akcí. Sekvenční grafy UML a stavové grafy (viz popis v kapitole 5) představují řadu akcí, k nimž dochází v reakci na určitou zprávu nebo událost. Při popisu požadavků na systémy, kde je kritická bezpečnost nebo zabezpečení, se někdy používají formální matematické specifikace. Za jiných okolností se však tento typ zápisu uplatňuje jen zřídka. Tento přístup k psaní specifikace vysvětlíme v kapitole 12.

3.2. Systém měří hladinu cukru v krvi a v případě potřeby poskytne inzulín každých 10 minut. *(Hladina cukru v krvi se mění relativně pomalu, takže není nutné ji měřit častěji. Méně časté měření by mohlo vést ke zbytečně vysokým hladinám cukru v krvi.)*

3.6. Systém bude každou minutu přecházet do rutiny vlastního testování, kde se budou testovat podmínky a provádět příslušné akce definované v tabulce 1. *(Rutina vlastního testování může zjistit hardwarové nebo softwarové problémy a upozornit uživatele na to, že systém nedokáže fungovat normálním způsobem.)*

Obrázek 4.9 – Příklady požadavků na softwarový systém inzulínové pumpy

4.3.1 Specifikace v přirozeném jazyce

Přirozený jazyk se při psaní požadavků na software používá již od začátků softwarového inženýrství. Dokáže předat mnoho informací, je intuitivní a univerzální. Na druhou stranu může být také vágní a nejasný a význam sdělení závisí na znalostech jejich adresátů. V důsledku toho se objevilo mnoho návrhů na alternativní způsoby, jak psát požadavky. Žádný z nich se však nesetkal s obecným přijetím a přirozený jazyk bude i nadále nejčastěji používaným způsobem, jak specifikovat systémové a softwarové požadavky.

Kvůli minimalizaci nedorozumění při psaní požadavků v přirozeném jazyce je vhodné dodržovat několik jednoduchých pokynů:

1. Vytvořte standardní formát a zkontrolujte, zda tomuto formátu odpovídají všechny definice požadavků. Standardizace formátu snižuje pravděpodobnost opomenutí a usnadňuje kontrolu požadavků. Je vhodné, aby formát vyjadřoval požadavky jednou větou. Ke každému uživatelskému požadavku je vhodné doplnit důvod, který popisuje, proč byl příslušný požadavek navržen. Důvod může také zahrnovat informaci o tom, kdo požadavek navrhl (zdroj požadavku). Díky tomu lze zjistit, koho je vhodné konzultovat v případě, že je požadavek nutné změnit.
2. Používejte jazyk konzistentně, abyste rozlišili povinné a nepovinné požadavky. Povinné požadavky jsou takové, které musí systém podporovat. Obvykle se zapisují v budoucím čase. Nepovinné požadavky nejsou nezbytné a uvádějí se formou „mělo by".
3. Klíčové části požadavků zvýrazněte (tučně, kurzívou nebo barevně).
4. Nepředpokládejte, že čtenáři budou rozumět technickému jazyku softwarového inženýrství. Slova jako „architektura" a „modul" je snadné vykládat chybně. Měli byste se tedy vyhýbat žargonu, zkratkám a akronymům.
5. Kdykoli je to možné, měli byste ke každému uživatelskému požadavku doplnit jeho důvod. Důvod by měl vysvětlovat, proč se požadavek do dokumentu dostal. Je to vhodné zvláště v případech, kdy se požadavky změní, protože se lze lépe rozhodnout, které změny nejsou žádoucí.

Možné uplatnění těchto pokynů je znázorněno na obrázku 4.9. Na tomto obrázku jsou uvedeny dva požadavky na integrovaný software automatizované inzulínové pumpy, kterou jsme představili v kapitole 1. Úplná specifikace požadavků na inzulínovou pumpu je k dispozici ke stažení na webu knihy.

4.3.2 Strukturované specifikace

Strukturovaný přirozený jazyk umožňuje psát systémové požadavky standardním způsobem, který omezuje volnost při jejich specifikaci. Tento přístup z větší části zachovává vyjadřovací možnosti a srozumitelnost přirozeného jazyka, ale zajišťuje, že specifikace získá určitou uniformitu. Zápisy strukturovaným jazykem specifikují systémové požadavky pomocí šablon. Specifikace mohou znázorňovat alternativní cesty a cykly pomocí konstrukcí programovacího jazyka a mohou zvýrazňovat klíčové prvky stínováním nebo odlišným písmem.

Robertsonovi (Robertson a Robertson, 1999) ve své knize o metodě inženýrství požadavků VOLERE doporučují zapisovat uživatelské požadavky zpočátku na jednotlivé karty. Navrhují na každou kartu umístit několik polí, jako je důvod požadavku, závislost na dalších požadavcích, zdroj požadavků, podpůrné materiály atd. Tento přístup se podobá příkladu strukturované specifikace, který jsme předvedli na obrázku 4.10.

INZULÍNOVÁ PUMPA/ŘÍDICÍ SOFTWARE/SRS/3.3.2

Funkce	Výpočet dávky inzulínu: bezpečná hladina cukru v krvi.
Popis	Vypočítá dávku inzulínu, kterou pumpa poskytne, když se aktuálně naměřená hladina cukru pohybuje v bezpečné zóně mezi 3 a 7 jednotkami.
Vstupy	Aktuální naměřená hodnota cukru v krvi (r2), dvě předchozí měření (r0 a r1).
Zdroj	Aktuální vstup ze senzoru cukru. Další hodnoty z paměti.
Výstupy	VypocDavka – poskytovaná dávka inzulínu.
Cíl	Hlavní řídicí cyklus.
Akce	VypocDavka je nulová, pokud je hladina cukru stabilní či klesá nebo pokud se hladina zvyšuje, ale klesá rychlost jejího přírůstku. Jestliže hladina roste a rychlost jejího nárůstu se zvyšuje, pak software vypočítá hodnotu VypocDavka tak, že vydělí rozdíl mezi aktuální hladinou cukru a předchozí hladinou čtyřmi a výsledek zaokrouhlí. Pokud je výsledek zaokrouhlen na nulu, je VypocDavka nastavena na minimální dávku, kterou lze podat.
Požadavky	Dvě předchozí naměřené hodnoty, aby bylo možné vypočítat rychlost změny hladiny cukru v krvi.
Předběžná podmínka	Zásobník inzulínu obsahuje alespoň maximální povolenou dávku inzulínu pro jednorázovou aplikaci.
Následná podmínka	Hodnotu r0 nahrazuje hodnota r1 a hodnotu r1 pak nahrazuje hodnota r2.
Vedlejší účinky	Žádné.

Obrázek 4.10 – Strukturovaná specifikace požadavků na inzulínovou pumpu

Chceme-li systémové požadavky specifikovat strukturovaným způsobem, definujeme jednu nebo více standardních šablon požadavků a reprezentujeme tyto šablony jako strukturované formuláře. Specifikace lze uspořádat kolem objektů, se kterými systém manipuluje, funkcí, které systém plní, nebo událostí, které zpracovává. Příklad specifikace založené na formuláři definuje, jak vypočítat dávku inzulínu, když se hladina cukru v krvi nachází v bezpečném pásmu (viz obrázek 4.10).

Podmínka	Akce
Hladina cukru klesá (r2 < r1)	VypocDavka = 0
Hladina cukru je stabilní (r2 = r1)	VypocDavka = 0
Hladina cukru se zvyšuje a rychlost zvyšování klesá ((r2 – r1) < (r1 – r0))	VypocDavka = 0
Hladina cukru se zvyšuje a rychlost zvyšování je stabilní nebo roste ((r2 – r1) ≥ (r1 – r0))	VypocDavka = zaokrouhli ((r2 – r1)/4) Pokud zaokrouhlený výsledek = 0, pak VypocDavka = MinimalniDavka

Obrázek 4.11 – Tabulková specifikace výpočtu inzulínové pumpy

Když se funkční požadavky specifikují pomocí standardního formuláře, je potřeba zahrnout následující informace:

1. Popis specifikované funkce nebo entity.
2. Popis příslušných vstupů a jejich původu.
3. Popis příslušných výstupů a jejich směřování.

4. Informace o informacích, které jsou potřebné k výpočtu, nebo jiných použitých entitách v systému (část „vyžaduje“).
5. Popis realizované akce.
6. Pokud se používá funkční přístup, je uvedena předběžná podmínka, která určuje, co musí být splněno před voláním funkce, a následná podmínka, která určuje, co musí platit po volání funkce.
7. Popis případných vedlejších účinků operace.

Použití strukturovaného jazyka odstraňuje některé problémy specifikací v přirozeném jazyce. Snižuje se variabilita specifikací a požadavky jsou uspořádány efektivněji. Přesto bývá i nadále obtížné psát požadavky jasným a jednoznačným způsobem, zejména v případech, kdy je nutné specifikovat složité výpočty (například výpočet dávkování inzulínu).

Kvůli řešení tohoto problému lze požadavky v přirozeném jazyce doplnit dodatečnými informacemi, například pomocí tabulek nebo grafických modelů systému. Tyto dodatečné informace mohou znázornit, jak postupují výpočty, jak se mění stavy systému, jak se systémem interagují uživatelé a jak probíhají sekvence akcí.

Tabulky jsou vhodné zvláště v případech, kdy existuje několik možných alternativních situací a potřebujeme popsat, které akce proběhnou v každé uvedené situaci. Inzulínová pumpa odvozuje požadavky na inzulín z rychlosti změny hladiny cukru v krvi. Rychlost změny se počítá pomocí aktuálních a dříve naměřených hodnot. Obrázek 4.11 ve formě tabulky shrnuje, jak lze z rychlosti změny hladiny cukru v krvi určit velikost dávky inzulínu.

4.4 Proces inženýrství požadavků

Jak jsme diskutovali v kapitole 2, proces inženýrství požadavků může zahrnovat čtyři vysokoúrovňové aktivity. Zaměřují se na hodnocení toho, zda je systém pro podnik užitečný (studie proveditelnosti), hledání požadavků (zjišťování a analýza), převod těchto požadavků do určité standardní formy (specifikace) a kontrolu toho, zda požadavky skutečně definují systém, který zákazník požaduje (validace). Na obrázku 2.6 jsme tyto aspekty znázornili jako sekvenční proces. V praxi však inženýrství požadavků představuje iterativní proces, kde se jednotlivé aktivity prolínají.

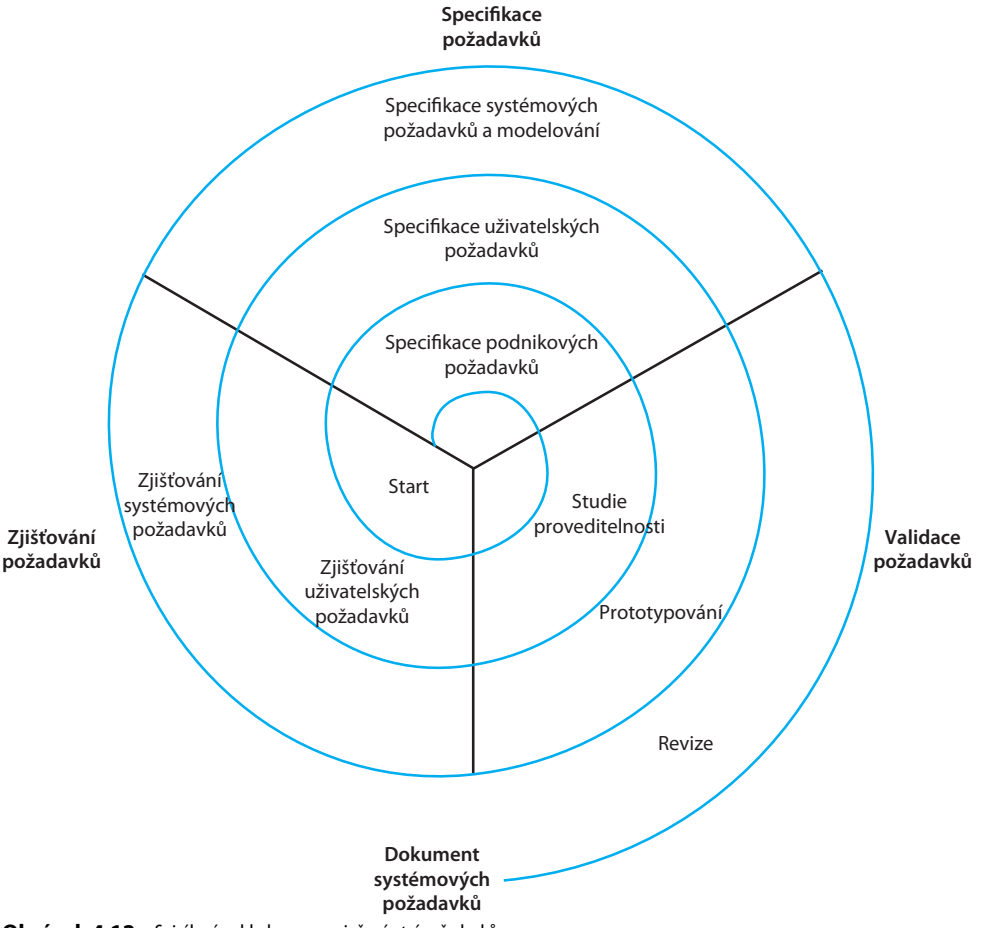
Toto překrývání je patrné na obrázku 4.12. Aktivity jsou uspořádány jako iterativní proces kolem spirály, kde výstupem je dokument systémových požadavků. Množství času a práce věnované jednotlivým aktivitám v každé iteraci závisí na fázi celkového procesu a na typu vyvíjeného systému. V rané etapě procesu se většina energie investuje do seznámení s vysokoúrovňovými podnikovými a mimofunkčními požadavky a uživatelskými požadavky na systém. V pozdější části procesu (na vnějších úsecích spirály) se více úsilí investuje do zjišťování a analýzy podrobných systémových požadavků.

STUDIE PROVEDITELNOSTI

Studie proveditelnosti je krátká a cílená studie, kterou je vhodné provést na začátku procesu inženýrství požadavků. Měla by odpovědět na tři klíčové otázky: a) přispívá systém k dosažení hlavních cílů organizace? b) lze systém implementovat v plánovaném čase a s vymezeným rozpočtem pomocí aktuální technologie? a c) je možné systém integrovat s jinými používanými systémy?

Jestliže je odpověď na libovolnou z těchto otázek záporná, pravděpodobně není vhodné na projektu pokračovat.

http://www.SoftwareEngineering-9.com/Web/Requirements/FeasibilityStudy.html



Obrázek 4.12 – Spirálový pohled na proces inženýrství požadavků

Tento spirálový model vychází z přístupu k vývoji, kde se požadavky vyvíjejí na různých úrovních podrobností. Počet iterací kolem spirály se může lišit, takže lze spirálu opustit, jakmile jsou zjištěny některé nebo všechny uživatelské požadavky. Místo prototypování lze zvolit agilní vývoj, takže se požadavky i implementace systému vyvíjí společně.

Někteří lidé považují inženýrství požadavků za proces, který aplikuje metodu strukturované analýzy, jako je například objektově orientovaná analýza (Larman, 2002). Jedná se o analýzu systému a vývoj sady grafických systémových modelů, jako jsou modely případů použití, které pak slouží jako specifikace systému. Sada modelů popisuje chování systému a je doplněna dalšími informacemi, které popisují například požadovaný výkon či spolehlivost systému.

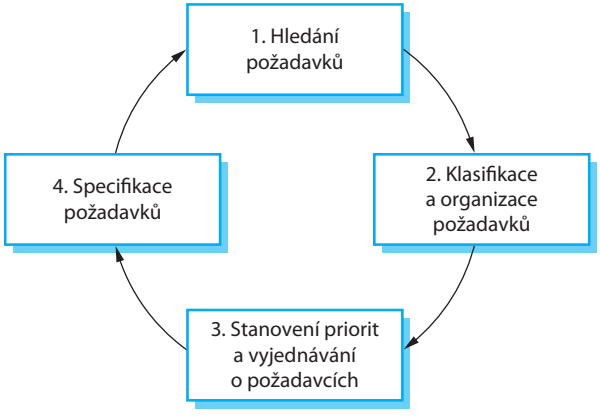
Strukturované metody sice v procesu inženýrství požadavků plní svou roli, ale inženýrství požadavků se zdaleka nevyčerpává pouze těmito metodami. Konkrétně zjišťování požadavků je osobně zaměřená aktivita a lidé nemají rádi omezení, jaká kladou rigidní systémové modely.

Téměř ve všech systémech se požadavky mění. Lidé, kteří na projektu pracují, si postupně ujasňují, co od softwaru očekávají, mění se organizace kupující systém a dochází ke změnám v systémovém hardwaru,

softwaru a organizačním prostředí. Proces správy těchto proměnlivých požadavků se označuje jako správa požadavků a budeme se jím zabývat v části 4.7.

4.5 Zjišťování a analýza požadavků

Po úvodní studii proveditelnosti pokračuje proces inženýrství požadavků svou další fází – zjišťováním a analýzou požadavků. Přitom softwaroví inženýři spolu se zákazníky a koncovými uživateli systému hledají aplikační doménu, služby, které má systém poskytovat, požadovaný výkon systému, hardwarová omezení atd.



Obrázek 4.13 – Proces zjišťování a analýzy požadavků

Na zjišťování a analýze požadavků se mohou podílet různí pracovníci organizace. Osoba zainteresovaná na systému je kdokoli, kdo má přímý nebo nepřímý vliv na systémové požadavky. K zainteresovaným osobám patří koncoví uživatelé, kteří budou se systémem interagovat, a kterýkoli jiný zaměstnanec organizace, kterého systém ovlivní. Mezi další osoby zainteresované na systému lze zařadit techniky, kteří vyvíjejí nebo spravují jiné související systémy, podnikové manažery, experty na příslušnou problematiku a zástupce odborů.

Procesní model zjišťování a analýzy je znázorněn na obrázku 4.13. Každá organizace bude mít vlastní konkrétní verzi tohoto obecného modelu, která závisí na místních faktorech typu zkušeností zaměstnanců, typu vyvíjeného systému, použitých standardech atd.

Proces má tyto aktivity:

1. *Hledání požadavků* – jedná se o proces interakce s osobami zainteresovanými na systému, kdy se ujasňují jejich požadavky. Během této činnosti se také určují doménové požadavky zainteresovaných osob a dokumentace. Při hledání požadavků lze zvolit několik doplňkových technik, kterými se budeme zabývat dále v této části.
2. *Klasifikace a organizace požadavků* – tato aktivita přijímá nestrukturovanou sadu požadavků, seskupuje související požadavky a uspořádává je do koherentních shluků. Při seskupení požadavků se nejčastěji používá model systémové architektury, který identifikuje podsystémy a přidružuje požadavky jednotlivým podsystémům. Aktivity inženýrství požadavků a návrhu architektury nelze v praxi zcela oddělit.
3. *Stanovení priorit a vyjednávání o požadavcích* – při zapojení více zainteresovaných osob nevyhnutelně dochází ke konfliktům požadavků. Cílem této aktivity je určit prioritu požadavků a poté najít

a vyřešit konflikty požadavků na základě vyjednávání. Obvykle je nutné, aby se zainteresované osoby sešly, prodiskutovaly svá odlišná stanoviska a dohodly se na kompromisní podobě požadavků.

4. *Specifikace požadavků* – požadavky se dokumentují a předávají do dalšího kola spirály. Lze produkovat formální nebo neformální dokumenty požadavků, jak jsme probrali v části 4.3.

Z obrázku 4.13 je zřejmé, že zjišťování a analýza požadavků je iterativní proces, kdy jednotlivé aktivity neustále poskytují zpětnou vazbu dalším aktivitám. Cyklus procesu začíná hledáním požadavků a končí jejich dokumentováním. V každém kole tohoto cyklu rozumí analytik požadavkům stále lépe. Cyklus končí, když je hotový dokument požadavků.

Zjišťování a analýza požadavků, které pocházejí od zainteresovaných osob, je z několika důvodů obtížný proces:

1. Zainteresované osoby často nevědí, co mají od počítačového systému očekávat, a dokážou své požadavky vyjádřit jen v nejobecnější formě. Mohou mít problémy artikulovat, co od systému chtějí, a mohou mít nerealistické požadavky, protože nevědí, co lze prakticky implementovat a co nikoli.
2. Osoby zainteresované na systému přirozeně vyjadřují své požadavky pomocí vlastní terminologie a s implicitními znalostmi své vlastní práce. Inženýři požadavků, kteří nemají zkušenosti v doméně zákazníka, těmto požadavkům nemusí rozumět.
3. Různé zainteresované osoby mají různé požadavky a mohou je vyjadřovat odlišnými způsoby. Inženýři požadavků někdy musí vyhledat všechny možné zdroje požadavků a poté analyzovat jejich společné prvky a konflikty.
4. V požadavcích na systém se někdy může projevovat vliv politických faktorů. Manažeři mohou klást konkrétní systémové požadavky proto, aby zvýšili svůj vliv v organizaci.
5. Ekonomické a podnikové prostředí, kde analýza probíhá, je dynamické. Během procesu analýzy se zpravidla mění. Může se měnit i význam konkrétních požadavků. Někdy se objeví nové požadavky od nových zainteresovaných osob, které původně nebyly konzultovány.

Nevyhnutelně dochází k tomu, že různé zainteresované osoby mají odlišný pohled na význam a prioritu požadavků a tyto pohledy jsou často v rozporu. Během tohoto procesu je potřeba organizovat pravidelná jednání zainteresovaných osob, kdy lze přijímat kompromisy. Není možné zcela uspokojit všechny zainteresované osoby, ale pokud mají některé z nich pocit, že jejich pohled nebyl dostatečně zohledněn, mohou celý proces inženýrství požadavků cíleně narušovat.

Ve fázi specifikace požadavků jsou dosud zjištěné požadavky dokumentovány takovým způsobem, který usnadňuje hledání dalších požadavků. V této fázi lze vytvořit koncept dokumentu systémových požadavků s chybějícími částmi a neúplnými požadavky. Případně je možné požadavky dokumentovat zcela jiným způsobem (například v tabulce tabulkového procesoru nebo na kartách). Karty s požadavky jsou v některých případech velmi efektivní, protože s nimi mohou zainteresované osoby snadno manipulovat, měnit je a uspořádávat.

HLEDISKA

Hledisko je způsob shromažďování a organizace sady požadavků od skupiny zainteresovaných osob, které mají něco společného. Každé hledisko tedy zahrnuje sadu systémových požadavků. Hlediska mohou pocházet od koncových uživatelů, vedoucích atd. Pomáhají identifikovat osoby, které mohou poskytnout informace o svých požadavcích, a vytvořit strukturu požadavků k analýze.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

4.5.1 Hledání požadavků

Proces hledání požadavků (někdy se označuje jako zjišťování požadavků) spočívá v shromažďování informací o požadovaném systému a existujících systémech. Z těchto informací následně extrahuje uživatelské a systémové požadavky. Mezi zdroje informací během fáze hledání požadavků patří dokumentace, osoby zainteresované na systému a specifikace podobných systémů. Se zainteresovanými osobami lze interagovat formou rozhovoru a pozorování jejich činnosti. Budoucí podobu systému je možné zainteresovaným osobám přiblížit pomocí scénářů a prototypů.

Mezi zainteresované osoby se řadí koncoví uživatelé systému, manažeři i externí zainteresované osoby, jako jsou regulační orgány, které certifikují přijatelnost systému. Mezi osoby zainteresované na informačním systému o psychiatrických pacientech mohou například patřit:

1. Pacienti, jejichž informace se do systému zaznamenávají.
2. Lékaři, kteří odpovídají za stanovení diagnóz a léčení pacientů.
3. Sestry, které koordinují konzultace s lékaři a podávají některé léky.
4. Nemocniční recepční, kteří domlouvají návštěvy pacientů.
5. Pracovníci oddělení IT, kteří mají na starosti instalaci a správu systému.
6. Manažer pro zdravotnickou etiku, který musí zajistit, že systém vyhoví aktuálním etickým pravidlům péče o pacienty.
7. Manažeři zdravotnické péče, kteří ze systému získávají informace potřebné k řízení.
8. Zaměstnanci pracující se zdravotnickými záznamy, kteří odpovídají za to, že bude možné spravovat a uchovávat systémové informace, a za správnou implementaci postupů ukládání záznamů.

Již jsme viděli, že kromě osob zainteresovaných na systému mohou požadavky pocházet i z aplikační domény a z jiných systémů, které se specifikovaným systémem interagují. Všechny je nutné v procesu zjišťování požadavků zvážit.

Tyto různé zdroje požadavků (zainteresované osoby, doména, systémy) lze vesměs reprezentovat jako systémová hlediska, kdy každé hledisko znázorňuje podmnožinu požadavků na systém. Odlišná hlediska ukazují problém z odlišných stran. Jejich perspektiva však není zcela nezávislá, ale obvykle se překrývají, takže mívají některé společné požadavky. Pomocí těchto hledisek můžete určit strukturu hledání systémových požadavků i jejich dokumentace.

4.5.2 Rozhovory

Formální nebo neformální rozhovory s osobami zainteresovanými na systému jsou součástí většiny procesů inženýrství požadavků. Při těchto rozhovorech kladou týmy inženýrství požadavků zainteresovaným osobám otázky týkající se systému, se kterým aktuálně pracují, a vyvíjeného systému. Z odpovědí na tyto otázky pak odvozují jednotlivé požadavky. Může se jednat o rozhovory dvou typů:

1. Uzavřené rozhovory, kde zainteresovaná osoba odpovídá na předem definovanou sadu otázek.
2. Otevřené rozhovory, které nemají žádnou předem definovanou agendu. Tým inženýrství požadavků probírá s osobami zainteresovanými na systému řadu otázek a díky tomu získává lepší přehled o jejich potřebách.

Rozhovory se zainteresovanými osobami se v praxi vyznačují kombinací obou přístupů. Může být nutné shromáždit odpovědi na určité otázky. Tyto odpovědi však obvykle vedou k dalším aspektům, které je možné probrat méně strukturovaným způsobem. Zcela otevřené diskuse málokdy fungují podle

očekávání. Obvykle je nutné položit několik počátečních otázek a poté vést rozhovor tak, aby se neodchyloval od problematiky vyvíjeného systému.

Rozhovory se hodí k získání celkového přehledu o práci zainteresovaných osob, o tom, jak mohou s novým systémem interagovat, a o potížích, které mají tyto osoby se svými současnými systémy. Lidé rádi mluví o své práci, takže se obvykle rozhovorů ochotně účastní. Rozhovory však nejsou příliš užitečné k seznámení s požadavky z aplikační domény.

Znalosti domény může být pomocí rozhovorů těžké získat ze dvou důvodů:

1. Všichni aplikační specialisté používají terminologii a žargon specifické pro svou doménu. Bez této terminologie o doménových požadavcích nedokážou komunikovat. Terminologii obvykle používají přesným a jemným způsobem, který si inženýři požadavků mohou snadno vykládat špatně.
2. Některé znalosti domény jsou zainteresovaným osobám natolik vlastní, že pro ně může být těžké je vysvětlit, nebo je považují za natolik základní, že nestojí za zmínku. Pro knihovníka je například samozřejmé, že všechny přírůstky se musí před přidáním do knihovny katalogizovat. Z hlediska tazatele to však samozřejmé být nemusí, takže se tento požadavek v seznamu neobjeví.

Rozhovory také nepředstavují efektivní metodu získávání znalostí o organizačních požadavcích a omezeních, protože mezi jednotlivými osobami v organizaci existují jemné vlivové vztahy. Publikovaná organizační schémata málokdy odpovídají skutečné rozhodovací struktuře v organizaci. Dotazované osoby však nemusí být ochotné vnějšímu tazateli odhalit místo teoretického stavu ten skutečný. Obecně lze říci, že většina osob obvykle váhá, má-li diskutovat o politických a organizačních otázkách, které mohou mít vliv na požadavky.

Efektivní tazatelé se vyznačují dvěma vlastnostmi:

1. Jsou otevření, nevytvářejí si předem pevnou představu o požadavcích a jsou ochotni zainteresovaným osobám naslouchat. Pokud zainteresovaná osoba přijde s překvapivými požadavky, dokážou svou představu o systému změnit.
2. Vyzývají dotazované k diskusi pomocí odrazových otázek a žádostí o požadavky nebo tak, že s nimi spolupracují na prototypu systému. Pokud lidi požádáme: „řekněte mi, co chcete“, málokdy dostaneme užitečné informace. Pro lidi je mnohem snadnější mluvit v definovaném kontextu než na obecné úrovni.

Informace z rozhovorů doplňují další informace o systému od dokumentace popisující podnikové procesy nebo existující systémy, pozorování uživatelů atd. Kromě údajů v systémových dokumentech mohou být někdy data z rozhovorů jediným zdrojem informací o systémových požadavcích. U samotných rozhovorů však hrozí, že nezachytí klíčové informace, a proto je vhodné použít je v kombinaci s jinými technikami zjišťování požadavků.

4.5.3 Scénáře

Lidé obvykle lépe rozumí praktickým příkladům než abstraktním popisům. Dokážou pochopit a rozebrat scénář možné interakce se softwarovým systémem. Inženýři požadavků mohou na základě informací získaných z této diskuse formulovat skutečné systémové požadavky.

Scénáře mohou být užitečné zejména při doplňování podrobností k obecnému popisu požadavků. Jedná se o popis ukázkových interakčních relací. Každý scénář se obvykle zabývá pouze malým počtem možných interakcí. Vyvíjejí se různé formy scénářů, které o systému poskytují odlišné typy informací na různých úrovních podrobností. Typem scénáře požadavků jsou i historie používané v extrémním programování, kterými jsme se zabývali v kapitole 3.

Scénář začíná osnovou interakce. V procesu zjišťování požadavků se do této osnovy doplňují podrobnosti, takže vzniká úplný popis dané interakce. Na nejobecnější rovině může scénář zahrnovat:

1. Popis toho, co systém a uživatelé očekávají na začátku scénáře.
2. Popis normálního toku událostí ve scénáři.
3. Popis možných problémů a způsobu jejich zvládání.
4. Informace o jiných aktivitách, které mohou probíhat ve stejnou dobu.
5. Popis stavu systému při dokončení scénáře.

POČÁTEČNÍ PŘEDPOKLAD:

Pacient se setkal s nemocničním recepčním, který vytvořil záznam v systému a uložil osobní informace pacienta (jméno, adresa, věk atd.). Do systému se přihlásí sestra a shromažďuje zdravotnickou historii.

NORMÁLNĚ:

Sestra vyhledá pacienta podle příjmení. Pokud je pacientů se stejným příjmením více, identifikuje se pacient pomocí křestního jména a data narození.

Sestra zvolí z nabídky možnost doplnění zdravotnické historie.

Sestra pak postupuje podle systémových dotazů, které požadují zadání informací o jiných místech, kde pacient konzultoval své duševní problémy (volný textový vstup), o stávajícím zdravotním stavu (sestra vybere stav z nabídky), aktuálně užívaných lécích (výběr z nabídky), alergiích (volný text) a osobním životě (formulář).

MOŽNÉ PROBLÉMY:

Pacientův záznam neexistuje nebo jej nelze najít. Sestra vytvoří nový záznam a zapíše osobní informace.

Nabídka neobsahuje stav pacienta ani medikaci. Sestra zvolí možnost „Jiné" a zadá volný text s popisem stavu nebo medikace.

Pacient nemůže nebo nechce poskytnout informace o své zdravotní historii. Sestra zadá volný text, že pacient nedokáže nebo není ochoten poskytnout informace. Systém vytiskne standardní vylučovací formulář s textem, že kvůli nedostatku informací může být léčba omezena nebo opožděna. Tento formulář je potřeba předat pacientovi a nechat jej podepsat.

JINÉ AKTIVITY:

Při zadávání mohou jiní zaměstnanci záznamy číst, ale nikoli upravovat.

STAV SYSTÉMU PŘI DOKONČENÍ:

Uživatel je přihlášen. Záznam pacienta včetně zdravotnické historie je uložen do databáze, do systémového protokolu je přidán záznam obsahující počáteční a koncový čas relace a identifikaci příslušné sestry.

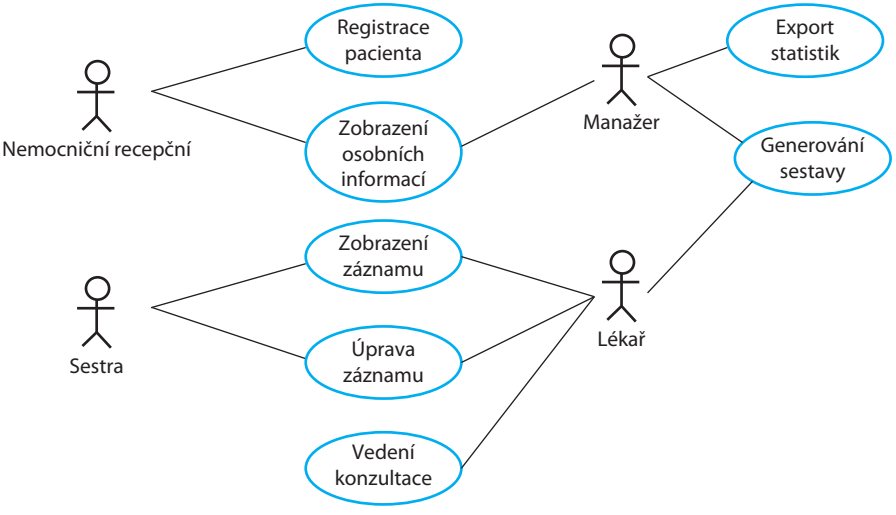
Obrázek 4.14 – Scénář shromažďování zdravotnické historie v systému MHC-PMS

Při zjišťování požadavků založených na scénářích se spolupracuje se zainteresovanými osobami na identifikaci scénářů a zachycování podrobností, které budou tyto scénáře zahrnovat. Scénáře lze zapsat v textové formě, doplnit diagramy, snímky obrazovek atd. Případně je možné zvolit strukturovanější přístup, který je založen například na scénářích událostí.

Jako příklad jednoduchého textového scénáře uveďme způsob, kterým systém MHC-PMS umožňuje zadat data o novém pacientovi (obrázek 4.14). Když pacient navštíví kliniku, nemocniční recepční vytvoří nový záznam a přidá do něj osobní informace (jméno, věk atd.). Sestra se pak dotazuje pacienta na jeho zdravotní historii. Pacient následně absolvuje první konzultaci s lékařem, který stanoví diagnózu a v případě potřeby doporučí postup léčby. Scénář znázorňuje, co se stane při shromáždění zdravotní historie.

4.5.4 Případy použití

Případy použití představují techniku hledání požadavků, která se poprvé objevila v metodě Objectory (Jacobson et al., 1993). V současnosti se řadí mezi základní funkce jazyka UML. Ve své nejjednodušší formě případ použití identifikuje aktéry, kteří se podílejí na interakci, a pojmenovává typ interakce. Následně se doplňují další informace, které popisují interakce se systémem. Další informace mohou mít podobu textového popisu či jednoho nebo více grafických modelů, jako jsou sekvenční nebo stavové grafy UML.



Obrázek 4.15 – Případy použití pro systém MHC-PMS

Případy použití se dokumentují pomocí vysokoúrovňového diagramu případů použití. Sada případů použití reprezentuje všechny možné interakce, které budou popsány v systémových požadavcích. Aktéry v tomto procesu (může se jednat o osoby nebo jiné systémy) zde představují schematické figurky. Každou třídu interakcí znázorňuje pojmenovaná elipsa. Aktéry a interakce propojují čáry. Čáry lze volitelně doplnit šipkami, které ukazují, kdo interakci inicializuje. To je patrné na obrázku 4.15, který znázorňuje některé případy použití pro systém správy informací o pacientech.

Mezi scénáři a případy použití neexistuje ostrá hranice. Někteří odborníci považují každý případ použití za samostatný scénář, zatímco jiní, jak navrhují Stevens a Pooley (2006), zahrnují do jediného případu použití celou sadu scénářů. Každý scénář představuje samostatný průchod případem použití. Může tedy existovat scénář normální interakce a vedle něj scénáře pro každou možnou výjimku. V praxi je můžeme používat oběma způsoby.

Případy použití identifikují jednotlivé interakce mezi systémem a jeho uživateli nebo jinými systémy. Každý případ použití je vhodné dokumentovat textovým popisem. Poté je lze propojit s jinými modely v jazyku UML, které scénář rozvinou podrobněji. Stručný popis případu použití Vedení konzultace z obrázku 4.15 může například vypadat takto:

> *Vedení konzultace umožňuje dvěma nebo více lékařům, kteří pracují v různých ordinacích, aby zobrazili stejný záznam současně. Jeden lékař inicializuje konzultaci tak, že zvolí zúčastněné osoby z rozevírací nabídky lékařů, kteří jsou online. Záznam o pacientovi se poté zobrazí na jejich obrazovkách, ale může jej upravovat pouze lékař, který konzultaci inicializoval. Kromě toho se otevře okno textové konverzace, která pomáhá koordinovat aktivitu. Předpokládá se, že bude samostatně navázána telefonní konference pro hlasovou komunikaci.*

Scénáře a případy použití jsou efektivní metody na získávání požadavků od zainteresovaných osob, které interagují přímo se systémem. Každý typ interakce lze reprezentovat jako případ použití. Vzhledem k tomu, že se zaměřují na interakci se systémem, nejsou však tolik efektivní při zjišťování omezení nebo vysokoúrovňových podnikových a mimofunkčních požadavků či při hledání doménových požadavků.

Jazyk UML je de facto standard v oblasti objektově orientovaného modelování. Případy použití se proto nyní při zjišťování požadavků uplatňují velmi často. Případy použití se budeme dále zabývat v kapitole 5, kde ukážeme, jak je lze použít k dokumentování návrhu systému vedle jiných modelů systému.

4.5.5 Etnografie

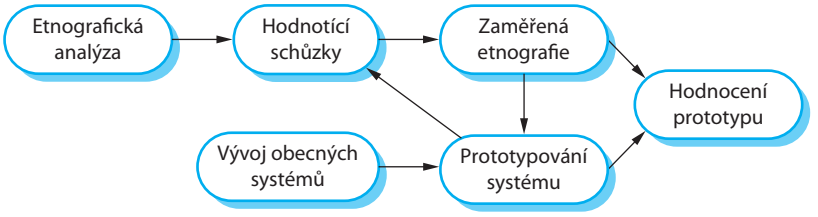
Softwarové systémy neexistují izolovaně. Používají se ve společenském a organizačním kontextu a systémové požadavky na software se mohou z tohoto kontextu odvozovat nebo jím mohou být omezeny. Uspokojení těchto společenských a organizačních požadavků je často klíčové, aby byl systém úspěšný. Jeden z důvodů, proč se mnoho softwarových systémů po svém dodání nikdy nepoužívá, spočívá v tom, že vývoj těchto systémů dostatečně nezohledňuje vliv společenského a organizačního kontextu na praktické fungování systému.

Etnografie je pozorovací technika, která umožňuje porozumět provozním procesům a pomáhá odvodit požadavky podpory těchto procesů. Analytik se ponoří do pracovního prostředí, kde se systém bude používat. Pozoruje každodenní práci a pořizuje poznámky týkající se vlastních úkolů, na nichž se účastníci podílejí. Hodnota etnografie spočívá v tom, že pomáhá odhalit implicitní systémové požadavky, které odrážejí skutečný způsob práce a nikoli formální procesy, jak je organizace definovala.

Pro lidi často bývá velmi obtížné popsat podrobnosti své práce, protože se pro ně stala druhou přirozeností. Své práci sice rozumějí, ale nemusí rozumět jejímu vztahu k práci jiných zaměstnanců organizace. Společenské a organizační faktory, které ovlivňují práci, ale které nejsou pro jednotlivce zjevné, se mohou ukázat teprve tehdy, když si jich všimne nezaujatý pozorovatel. Pracovní skupina se například může sama organizovat, takže její členové znají práci všech ostatních a mohou se vzájemně zastoupit, když někdo z nich chybí. Při rozhovoru to nemusí vyjít najevo, protože skupina to nemusí vnímat jako integrální část své práce.

Pionýrem etnografie při studiu kancelářské práce byla Suchman (1987). Zjistila, že skutečné pracovní postupy byly mnohem bohatší, mnohem složitější a mnohem dynamičtější než jednoduché modely, jaké předpokládaly systémy kancelářské automatizace. Rozdíly mezi předpokládanou a skutečnou prací představovaly nejdůležitější důvod toho, proč tyto kancelářské systémy neměly žádný významný dopad na produktivitu. Crabtree (2003) diskutuje širokou škálu následných studií a obecně popisuje uplatnění etnografie při návrhu systémů. Autor při svém vlastním výzkumu analyzoval metody, které umožňují integrovat etnografii do procesu softwarového inženýrství jejím propojením s metodami inženýrství

požadavků (Viller a Sommerville, 1999; Viller a Sommerville, 2000) a dokumentováním vzorů interakce v kooperativních systémech (Martin et al., 2001; Martin et al., 2002; Martin a Sommerville, 2004).



Obrázek 4.16 – Etnografie a prototypování při analýze požadavků

Etnografie je efektivní zejména při zjišťování dvou typů požadavků:

1. Požadavky odvozené ze způsobu, jakým lidé skutečně pracují, namísto způsobu, o kterém definice procesů tvrdí, že se tak má pracovat. Například kontroloři letového provozu mohou vypnout výstražný systém, který detekuje letadla, jejichž letové dráhy se protínají, ačkoli normální postupy řízení letového provozu určují, že se má tento systém používat. Kontroloři záměrně na krátkou dobu umisťují letadla na konfliktní dráhy, aby mohli letový prostor lépe využít. Jejich řídicí strategie zajišťuje, že se příslušná letadla dostanou od sebe dříve, než dojde k problémům, a systém varování před konflikty je při jejich práci rozptyluje.
2. Požadavky odvozené od spolupráce a povědomí o aktivitách jiných osob. Kontroloři letového provozu mohou například díky přehledu o činnosti svých kolegů odhadovat, kolik letadel vstoupí do jejich řídicího sektoru. Potom v závislosti na tomto předpovězeném vytížení upravují svou řídicí strategii. Automatizovaný systém řízení letového provozu by měl tedy kontrolorům určitého sektoru dovolit, aby mohli sledovat část provozu v sousedních sektorech.

Etnografii lze kombinovat s prototypováním (obrázek 4.16). Etnografie poskytuje informace pro vývoj prototypu, takže postačuje menší počet cyklů ladění prototypu. Prototypování kromě toho přispívá k lepšímu zacílení etnografie, protože identifikuje problémy a otázky, které lze potom diskutovat s etnografem. Etnograf by měl následně hledat odpovědi na tyto otázky během další fáze svého studia systému (Sommerville et al., 1993).

Etnografické studie mohou odhalit kritické podrobnosti o procesech, které jiné metody zjišťování požadavků často nezachytí. Vzhledem ke svému zaměření na koncového uživatele se však tento přístup pokaždé nehodí k zjišťování organizačních nebo doménových požadavků. Nedokážou vždy identifikovat nové funkce, které je potřeba do systému přidat. Etnografie sama o sobě tedy nepředstavuje kompletní přístup k zjišťování požadavků a měla by se používat jako doplněk jiných přístupů, jako je analýza případů použití.

REVIZE POŽADAVKŮ

Revize požadavků je proces, při kterém skupina pracovníků zákazníka systému a vývojáře systému podrobně čte dokument požadavků a hledá v něm chyby, anomálie a inkonzistence. Jakmile jsou určité problémy zjištěny a zaznamenány, musí se zákazník s vývojářem dohodnout, jak se budou identifikované problémy řešit.

http://www.SoftwareEngineering-9.com/Web/Requirements/Reviews.html

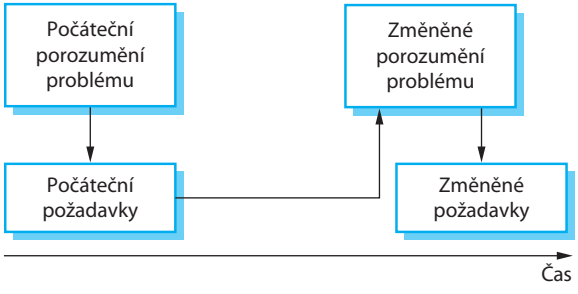
4.6 Validace požadavků

Proces validace požadavků spočívá v kontrole, zda požadavky skutečně definují systém, který zákazník požaduje. Překrývá se s analýzou, protože se zaměřuje na hledání problémů v požadavcích. Validace požadavků je důležitá proto, že chyby v dokumentu požadavků mohou vést k značným nákladům na přepracování, když se příslušné potíže objeví teprve během vývoje nebo po uvedení systému do provozu.

Náklady na opravu problému s požadavky pomocí změny systému obvykle značně převyšují náklady na opravu chyb návrhu nebo programového kódu. Je to dáno tím, že změna požadavků obvykle přináší nutnost změny návrhu a implementace systému. Kromě toho je potřeba systém znovu testovat.

Během procesu validace požadavků je nutné provést různé kontroly požadavků v dokumentu požadavků. Jedná se o tyto kontroly:

1. *Kontroly validity* – uživatel se může domnívat, že systém je potřebný k zajištění určitých funkcí. Další úvahy a analýzy však mohou identifikovat další nebo odlišné požadované funkce. Systémy mají různorodé zainteresované osoby s odlišnými požadavky a libovolná sada požadavků nevyhnutelně představuje kompromis v rámci komunity zainteresovaných osob.
2. *Kontroly konzistence* – požadavky v dokumentu by neměly být v konfliktu. To znamená, že stejná systémová funkce by neměla mít rozporná omezení nebo odlišné popisy.
3. *Kontroly úplnosti* – dokument požadavků by měl obsahovat požadavky, které definují všechny funkce a omezení, které uživatelé systému předpokládají.
4. *Kontroly realističnosti* – požadavky je potřeba zkontrolovat se znalostí stávající technologie, zda je lze skutečně implementovat. Tyto kontroly by měly rovněž zohlednit rozpočet a časový rozvrh vývoje systému.
5. *Verifikovatelnost* – kvůli omezení potenciálu sporů mezi zákazníkem a smluvním partnerem je vhodné požadavky vždy zapisovat tak, aby je bylo možné verifikovat. To znamená, že by mělo být možné zapsat sadu testů, které dokážou demonstrovat, že poskytnutý systém splňuje každý specifikovaný požadavek.



Obrázek 4.17 – Evoluce požadavků

K dispozici jsou různé techniky validace požadavků, které lze použít individuálně nebo ve vzájemné kombinaci:

1. *Revize požadavků* – tým kontrolorů systematicky analyzuje požadavky a hledá v nich chyby a inkonzistence.

2. *Prototypování* – při tomto přístupu k validaci se koncovým uživatelům a zákazníkům předvádí spustitelný model příslušného systému. Uživatelé mohou s modelem experimentovat, aby ověřili, zda odpovídá jejich reálným potřebám.
3. *Generování testových případů* – požadavky by mělo být možné testovat. Pokud jsou testy požadavků odvozeny jako součást procesu validace, často se přitom odhalí problémy s požadavky. Jestliže je návrh testů obtížný nebo nemožný, obvykle to znamená, že požadavky bude složité implementovat a bylo by vhodné je přehodnotit. Vývoj testů z uživatelských požadavků ještě před psaním jakéhokoli kódu je integrální součástí extrémního programování.

Problémy s validací požadavků bychom neměli podceňovat. Nakonec je obtížné dokázat, že sada požadavků skutečně odpovídá potřebám uživatelů. Uživatelé si musí představit, jak bude systém fungovat a jak ovlivní jejich práci. Taková abstraktní analýza je náročná dokonce i pro zkušené počítačové profesionály, nemluvě o pouhých uživatelích systému. V důsledku tohoto se během procesu validace požadavků obvykle neukážou všechny problémy s požadavky. Po schválení dokumentu požadavků se budou požadavky nevyhnutelně měnit, aby se napravila opomenutí a nedorozumění.

4.7 Správa požadavků

Požadavky na velký softwarový systém se neustále mění. Jeden důvod spočívá v tom, že tyto systémy se obvykle vyvíjejí v reakci na „záludné“ problémy – tj. problémy, které nelze úplně definovat. Vzhledem k tomu, že problémy není možné definovat kompletně, jsou požadavky na software z principu neúplné. V procesu vývoje softwaru se povědomí zainteresovaných osob o problému neustále vyvíjí (obrázek 4.17). Aby systémové požadavky tomuto měnícímu se pohledu na problém odpovídaly, musí se vyvíjet také.

TRVALÉ A PŘECHODNÉ POŽADAVKY

Některé požadavky mají větší tendenci ke změnám než jiné. Trvalé jsou takové požadavky, které souvisejí s klíčovými a málo proměnnými aktivitami organizace. Trvalé požadavky souvisejí se základními pracovními činnostmi. Pravděpodobnost změny přechodných požadavků je vyšší. Zpravidla se týkají podpůrných aktivit, které odrážejí, jak organizace své aktivity vykonává, nikoli vlastní činnost.

http://www.SoftwareEngineering-9.com/Web/Requirements/EnduringReq.html

Poté co je systém nainstalován a pravidelně se používá, nevyhnutelně se objeví nové požadavky. Pro uživatele i zákazníky systému je obtížné předpokládat, jaký vliv bude mít nový systém na jejich podnikové procesy a na způsob, kterým plní své úkoly. Jakmile koncoví uživatelé získají zkušenosti se systémem, objeví nové potřeby a priority. Změny jsou nevyhnutelné z několika důvodů:

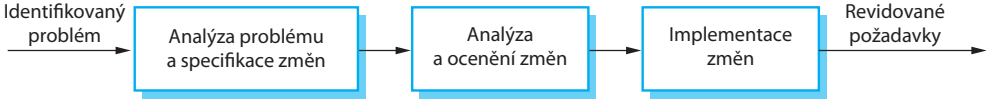
1. Podnikové a technické prostředí systému se po instalaci vždy změní. Někdy se začne používat nový hardware, může být nutné vytvořit rozhraní systému s jinými systémy, mohou se změnit podnikové priority (s následnou nutností změn systémové podpory) a mohou se objevit nové zákony a předpisy, kterým se systém musí přizpůsobit.
2. Za systém obvykle platí někdo jiný než jeho uživatelé. Zákazníci systému kladou požadavky, které vycházejí z organizačních a rozpočtových omezení. Tyto požadavky mohou být v rozporu s potřebami koncových uživatelů. Aby systém splnil svůj účel, může být nutné po dodání systému doplnit nové funkce uživatelské podpory.

3. Velké systémy zpravidla mívají různorodou uživatelskou komunitu, kde mnozí uživatelé mají odlišné požadavky a priority, které mohou být vzájemně v rozporu. Výsledné systémové požadavky nevyhnutelně představují kompromis mezi nimi. Ze zkušeností vyplývá, že je často nutné upravit úroveň podpory poskytované různým uživatelům.

V procesu správy požadavků je potřeba porozumět změnám systémových požadavků a tyto změny dostat pod kontrolu. Je nutné sledovat jednotlivé požadavky a udržovat vazby mezi závislými požadavky, aby bylo možné vyhodnotit dopad změn jednotlivých požadavků. Je nezbytné ustavit formální proces zadávání návrhů na změny a jejich propojení se systémovými požadavky. Formální proces správy požadavků je vhodné zahájit ihned poté, kdy je dostupný koncept dokumentu požadavků. Postupy správy proměnlivých požadavků je však vhodné začít plánovat již v rámci procesu zjišťování požadavků.

4.7.1 Plánování správy požadavků

Plánování je nezbytnou první fází procesu správy požadavků. Ve fázi plánování se definuje požadovaná úroveň podrobností správy požadavků. Během fáze správy požadavků je nutné přijmout rozhodnutí ohledně těchto faktorů:



Obrázek 4.18 – Správa změn požadavků

1. *Identifikace požadavků* – každý požadavek je potřeba samostatně identifikovat, aby jej bylo možné křížově propojit s jinými požadavky a použít při hodnocení sledovatelnosti.
2. *Proces správy změn* – jedná se o sadu aktivit, které umožňují hodnotit dopady změn a jejich náklady. Tímto procesem se budeme podrobněji zabývat v následující části.
3. *Zásady sledovatelnosti* – tyto zásady definují vztahy mezi jednotlivými požadavky a požadavky a návrhem systému, které je potřeba zaznamenat. Zásady sledovatelnosti by měly také definovat, jak se budou tyto záznamy udržovat.
4. *Podpora nástrojů* – součástí správy požadavků je zpracování velkého množství informací o požadavcích. Škála vhodných nástrojů sahá od specializovaných systémů na správu požadavků po tabulkové procesory a jednoduché databázové systémy.

Správa požadavků vyžaduje automatizovanou podporu a softwarové nástroje k tomuto účelu je potřeba zvolit ve fázi plánování. Podpora nástrojů je potřebná pro:

1. *Ukládání požadavků* – požadavky je vhodné uchovávat v zabezpečeném a spravovaném datovém úložišti, které je přístupné všem, kteří se podílejí na procesu inženýrství požadavků.
2. *Správa změn* – proces správy změn (viz obrázek 4.18) lze zjednodušit, je-li k dispozici aktivní podpora nástrojů.
3. *Správa sledovatelnosti* – jak jsme zmínili výše, podpora nástrojů pro sledovatelnost umožňuje vyhledávat související požadavky. Některé dostupné nástroje používají k zjišťování možných vztahů mezi požadavky techniky zpracování přirozeného jazyka.

U malých systémů nemusí být nutné používat specializované nástroje pro správu požadavků. Podporu procesu správy požadavků lze zajistit pomocí funkcí, které poskytují textové procesory, tabulkové procesory a databáze pro počítače PC. U velkých systémů se však nelze obejít bez podpory specializovanějších

nástrojů. Odkazy na informace o nástrojích pro správu požadavků jsou uvedeny na webových stránkách knihy.

4.7.2 Správa změn požadavků

Správu změn požadavků (viz obrázek 4.18) je nutné aplikovat na všechny navržené změny systémových požadavků, poté co byl schválen dokument požadavků. Změna požadavků má zásadní význam, protože se musíme rozhodnout, zda výhody implementace nových požadavků převažují nad náklady jejich implementace. Nasazení formálního procesu správy změn je výhodné proto, že všechny požadavky na změny jsou zpracovány konzistentně a dokument požadavků se mění kontrolovaným způsobem.

SLEDOVATELNOST POŽADAVKŮ

Je potřeba sledovat vztahy mezi požadavky, jejich zdroji a návrhem systému, aby bylo možné analyzovat důvody navržených změn a dopad, jaké budou tyto změny mít na jiné části systému. Potřebujeme mít možnost sledovat, jak se změna postupně projevuje v různých částech systému. Proč?

http://www.SoftwareEngineering-9.com/Web/Requirements/ReqTraceability.html

Proces správy změn zahrnuje tři základní fáze:

1. *Analýza problému a specifikace změn* – proces začíná identifikací problému s požadavky, případně v některých případech konkrétním návrhem na změnu. V této fázi probíhá analýza validity problému nebo návrhu na změnu. Tato analýza se pak předává zpět žadateli o změnu, který může reagovat konkrétnějším návrhem na změnu požadavků nebo se může rozhodnout, že svůj požadavek stáhne.
2. *Analýza a ocenění změn* – efekt navržené změny se vyhodnocuje na základě informací sledovatelnosti a obecných znalostí o systémových požadavcích. Náklady na změnu se odhadují z hlediska úprav dokumentu požadavků a případně i návrhu a implementace systému. Po dokončení této analýzy padne rozhodnutí, zda bude práce na požadavku na změnu pokračovat.
3. *Implementace změn* – dochází k úpravě dokumentu požadavků, a je-li to nutné, také návrhu a implementace systému. Dokument požadavků je vhodné uspořádat tak, aby jej bylo možné měnit bez rozsáhlého přepisování nebo přeskupování. Stejně jako v případě programů lze možnost změny dokumentů zajistit minimalizací externích odkazů a tím, že jsou sekce dokumentu co nejvíce modulární. Jednotlivé části lze poté změnit nebo nahrazovat, aniž by to ovlivnilo jeho jiné části.

Jestliže je potřeba naléhavě implementovat nový požadavek, často to svádí k tomu, aby se nejdříve změnil systém a zpětně se upravoval dokument požadavků. Je lepší se tomuto postupu vyhnout, protože téměř nevyhnutelně vede k tomu, že se specifikace požadavků začne od implementace systému odchylovat. Jakmile dojde ke změnám systému, snadno můžeme zapomenout tyto změny zanést do dokumentu požadavků nebo můžeme do tohoto dokumentu přidat informace, které nejsou s implementací konzistentní.

Agilní vývojové procesy, jako je extrémní programování, byly navrženy tak, aby umožnily zvládat změny požadavků během vývojového procesu. Když při agilním vývoji uživatel navrhne změnu požadavků, tato změna neprochází formálním procesem správy změn. Místo toho musí uživatel určit prioritu dané změny, a pokud této změně přiřadí vysokou prioritu, musí se rozhodnout, které systémové funkce plánované pro další iteraci budou vypuštěny.

HLAVNÍ BODY

- Požadavky na softwarový systém určují, jaké funkce bude systém poskytovat, a definují omezení jeho provozu a implementace.
- Funkční požadavky definují služby, které musí systém nabízet, nebo popisují, jakým způsobem bude provádět některé výpočty.
- Mimofunkční požadavky často kladou omezení na vyvíjený systém a použitý vývojový proces. Může se jednat o produktové, organizační nebo externí požadavky. Často souvisejí s emergentními vlastnostmi systému a týkají se proto systému jako celku.
- Dokument požadavků na software představuje schválený popis systémových požadavků. Měl by být uspořádán tak, aby jej mohli používat zákazníci systému i vývojáři softwaru.
- Proces inženýrství požadavků zahrnuje studii proveditelnosti, zjišťování a analýzu požadavků, specifikaci požadavků, validaci požadavků a správu požadavků.
- Zjišťování a analýza požadavků je iterativní proces, který můžeme reprezentovat jako spirálu aktivit – hledání požadavků, klasifikace a organizace požadavků, vyjednávání o požadavcích a dokumentace požadavků.
- Proces validace požadavků spočívá v kontrole validity, konzistence, úplnosti, realističnosti a verifikovatelnosti požadavků.
- Podnikové, organizační a technické změny nevyhnutelně vedou ke změnám požadavků na softwarový systém. Správa požadavků je proces, který umožňuje tyto změny spravovat a kontrolovat.

DALŠÍ ZDROJE INFORMACÍ

Software Requirements, 2nd edition (Softwarové požadavky, 2. vyd.). Tato kniha určená pro tvůrce a uživatele požadavků rozebírá doporučené postupy inženýrství požadavků. (K. M. Weigers, 2003, Microsoft Press.)

„Integrated requirements engineering: A tutorial" (Kurz integrovaného inženýrství požadavků). V tomto výukovém článku jsem diskutoval aktivity inženýrství požadavků a způsoby, kterými je lze přizpůsobit tak, aby odpovídaly moderním postupům softwarového inženýrství. (I. Sommerville, IEEE Software, 22(1), leden–únor 2005.) http://dx.doi.org/10.1109/MS.2005.13.

Mastering the Requirements Process, 2nd edition (Správa procesu požadavků, 2. vydání). Dobře napsaná a snadno čitelná kniha, která vychází z konkrétní metody (VOLERE), ale zahrnuje také mnoho užitečných obecných rad ohledně inženýrství požadavků. (S. Robertson a J. Robertson, 2006, Addison-Wesley.)

„Research Directions in Requirements Engineering" (Směry výzkumu v inženýrství požadavků). Jedná se o kvalitní shrnutí výzkumu inženýrství požadavků. Upozorňuje na výzkumné oblasti, které vyžadují další pozornost, aby bylo možné řešit otázky škálování a agility. (B. H. C. Cheng a J. M. Atlee, Proc. Conf on Future of Software Engineering, IEEE Computer Society, 2007.) http://dx.doi.org/10.1109/FOSE.2007.17.

CVIČENÍ

4.1. Identifikujte a stručně popište čtyři typy požadavků, které lze pro počítačový systém definovat.

4.2. Najděte nejasnosti nebo opomenutí v následujících definicích požadavků na část systému výdeje jízdenek.

Automatizovaný systém výdeje jízdenek prodává jízdenky na vlak. Uživatelé vyberou svůj cíl, vloží platební kartu a zadají své osobní identifikační číslo. Systém vydá jízdenku na vlak a zaúčtuje příslušnou částku na vrub platební karty. Když uživatel stiskne tlačítko start, aktivuje se nabídka potenciálních cílů spolu s výzvou uživateli, aby vybral svůj cíl. Po výběru cíle se uživatelům zobrazí výzva, aby vložili svou platební kartu. Zkontroluje se platnost karty a uživatel je potom požádán, aby zadal osobní identifikátor. Po validaci transakce platební kartou je vydána jízdenka.

4.3. Přepište výše uvedený popis pomocí strukturovaného přístupu, který jsme popsali v této kapitole. Vhodným způsobem vyřešte zjištěné nejasnosti.

4.4. Napište sadu mimofunkčních požadavků na systém výdeje jízdenek, přičemž stanovíte očekávanou spolehlivost a dobu odezvy systému.

4.5. Pomocí zde navržené techniky, kde jsou popisy v přirozeném jazyce uvedeny ve standardním formátu, napište věrohodné uživatelské požadavky na následující funkce:

- Samoobslužný systém na čerpací stanici, který zahrnuje čtečku platebních karet. Zákazník protáhne kartu čtečkou a poté určí, kolik paliva chce natankovat. Systém poskytne příslušný objem paliva a zatíží zákazníkův účet.
- Funkce výběru hotovosti z bankomatu.
- Kontrola pravopisu a automatické opravy v textovém procesoru.

4.6. Navrhněte, jak může technik odpovědný za vytvoření specifikace systémových požadavků sledovat vztahy mezi funkčními a mimofunkčními požadavky.

4.7. Na základě svých zkušeností s používáním bankomatu vyviňte sadu případů použití, která by mohla sloužit jako základ pro seznámení s požadavky na systém bankomatu.

4.8. Kdo by se měl podílet na revizi požadavků? Načrtněte model procesů, který znázorní možné uspořádání revize požadavků.

4.9. Když je potřeba provést naléhavé změny systému, může být nutné modifikovat systémový software ještě před schválením změn požadavků. Navrhněte model procesu na provedení těchto úprav, který zajistí, že se neobjeví inkonzistence mezi dokumentem požadavků a implementací systému.

4.10. Pracujete pro uživatele softwaru, který si objednal vývoj systému u vašeho předchozího zaměstnavatele. Zjistíte, že interpretace požadavků ve vaší společnosti se liší od způsobu, jakým požadavky interpretoval předchozí zaměstnavatel. Diskutujte, jak byste se měli v takové situaci zachovat. Víte, že pokud se nejasnosti nevyřeší, zvýší se náklady vašeho stávajícího zaměstnavatele. Na druhou stranu byste však zároveň měli zachovat důvěrnost informací, které jste získali ve své předchozí práci.

CITACE

Beck, K. (1999). „Embracing Change with Extreme Programming" (Přijetí změn v extrémním programování). *IEEE Computer*, **32** (10), 70–8.

Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography* (Návrh kolaborativních systémů: praktický průvodce etnografií). Londýn: Springer-Verlag.

Davis, A. M. (1993). *Software Requirements: Objects, Functions and States* (Softwarové požadavky: objekty, funkce a stavy). Englewood Cliffs, NJ: Prentice Hall.

IEEE. (1998). „IEEE Recommended Practice for Software Requirements Specifications" (Doporučené postupy IEEE pro specifikace softwarových požadavků). In *IEEE Software Engineering Standards Collection*. Los Alamitos, Ca.: IEEE Computer Society Press.

Jacobson, I., Christerson, M., Jonsson, P. a Overgaard, G. (1993). *Object-Oriented Software Engineering* (Objektově orientované softwarové inženýrství). Wokingham: Addison-Wesley.

Kotonya, G. a Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques* (Inženýrství požadavků: procesy a techniky). Chichester, UK: John Wiley and Sons.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process* (Aplikace UML a vzorů: úvod do objektově orientované analýzy a návrhu a procesu Unified Process). Englewood Cliff, NJ: Prentice Hall.

Martin, D., Rodden, T., Rouncefield, M., Sommerville, I. a Viller, S. (2001). „Finding Patterns in the Fieldwork" (Hledání vzorů v terénní práci). *Proc. ECSCW'01*. Bonn: Kluwer. 39–58.

Martin, D., Rouncefield, M. a Sommerville, I. (2002). „Applying patterns of interaction to work (re) design: E-government and planning" (Aplikace vzorů interakce na pracovní návrhy: e-government a plánování). *Proc. ACM CHI'2002*, ACM Press. 235–42.

Martin, D. a Sommerville, I. (2004). „Patterns of interaction: Linking ethnomethodology and design" (Vzory interakce: propojení etnografické metodiky a návrhu). *ACM Trans. on Computer-Human Interaction*, **11** (1), 59–89.

Robertson, S. a Robertson, J. (1999). *Mastering the Requirements Process* (Zvládnutí procesu požadavků). Harlow, UK: Addison-Wesley.

Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. a Twidale, M. (1993). „Integrating ethnography into the requirements engineering process" (Integrace etnografie do procesu inženýrství požadavků). *Proc. RE'93*, San Diego CA.: IEEE Computer Society Press. 165–73.

Stevens, P. a Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components, 2nd ed* (Použití UML: softwarové inženýrství s objekty a komponentami, 2. vyd.). Harlow, UK: Addison Wesley.

Suchman, L. (1987). *Plans and Situated Actions* (Plány a situované akce). Cambridge: Cambridge University Press.

Viller, S. a Sommerville, I. (1999). „Coherence: An Approach to Representing Ethnographic Analyses in Systems Design" (Koherence: přístup k reprezentaci etnografických analýz při návrhu systémů). *Human-Computer Interaction*, **14** (1 & 2), 9–41.

Viller, S. a Sommerville, I. (2000). „Ethnographically informed analysis for software engineers" (Etnograficky informovaná analýza pro softwarové inženýry). *Int. J. of Human-Computer Studies*, **53** (1), 169–96.

Systémové modelování

5

Cíle

Cílem této kapitoly je představit některé typy systémových modelů, které lze vyvinout v rámci procesů inženýrství požadavků a návrhu systému. V této kapitole:

- porozumíte tomu, jak lze reprezentovat softwarové systémy pomocí grafických modelů,
- seznámíte se s různými typy modelů a základními perspektivami systémového modelování: kontextovou, interakční, strukturní a behaviorální,
- setkáte se s některými typy diagramů jazyka UML (Unified Modeling Language) a dozvíte se, jak lze tyto diagramy uplatnit při systémovém modelování,
- získáte představu o idejích, z nichž vychází inženýrství řízené modely, kde je systém automaticky generován ze strukturních a behaviorálních modelů.

Proces systémového modelování umožňuje vyvíjet abstraktní modely systému, kdy každý model prezentuje odlišný pohled či perspektivu daného systému. Systémové modelování postupně začalo reprezentovat systémy určitým způsobem grafického zápisu, který je v současnosti téměř vždy založen na jazyku UML. Je však rovněž možné vyvinout formální (matematické) modely systému, obvykle jako podrobné specifikace systému. Grafickým modelováním pomocí jazyka UML se budeme zabývat v této kapitole a formálním modelováním v kapitole 12.

V procesu inženýrství požadavků pomáhají modely odvodit požadavky na systém, během procesu návrhu umožňují popsat systém pro techniky, kteří systém implementují, a po implementaci slouží k dokumentování struktury a fungování systému. Lze vyvinout modely stávajících systémů i nově vyvíjených systémů:

1. Modely existujícího systému se používají při inženýrství požadavků. Pomáhají vyjasnit funkce stávajícího systému a mohou sloužit jako základ pro diskusi o jeho silných a slabých stránkách. Tyto diskuse pak mohou vést k požadavkům na nový systém.
2. Modely nového systému pomáhají při inženýrství požadavků vysvětlit navržené požadavky jiným osobám zainteresovaným na systému. Technici pomocí těchto modelů rozebírají designové návrhy

a dokumentují systém pro implementaci. V procesu inženýrství řízeného modely lze z modelu systému generovat úplnou nebo částečnou implementaci systému.

Nejdůležitějším aspektem modelu systému je to, že vynechává podrobnosti. Model je abstrakce studovaného systému, nikoli jeho alternativní reprezentace. Reprezentace systému by měla v ideálním případě zachovat veškeré informace o reprezentované entitě. Abstrakce záměrně zjednodušuje a vybírá pouze nejdůležitější vlastnosti. Například ve velmi nepravděpodobném případě, že by tato kniha vycházela na pokračování v novinách, tato prezentace by byla abstrakcí hlavních témat knihy. Český překlad této knihy představuje její alternativní reprezentaci. Záměrem překladatele je zachovat veškeré informace, které obsahuje anglický originál.

Můžeme vyvinout různé modely, které budou systém reprezentovat z odlišných hledisek. Například:

1. Externí perspektiva, která modeluje kontext nebo prostředí systému.
2. Interakční perspektiva, která modeluje interakce mezi systémem a jeho prostředím nebo mezi komponentami systému.
3. Strukturní perspektiva, která modeluje uspořádání systému nebo strukturu dat, která systém zpracovává.
4. Behaviorální perspektiva, která modeluje dynamické chování systému a to, jak systém reaguje na události.

Tyto perspektivy mají hodně společných prvků s Krutchenovým pohledem 4 + 1 na systémovou architekturu (Kruchten, 1995), v němž navrhuje dokumentovat architekturu a organizaci systému z různých perspektiv. Uvedeným přístupem 4 + 1 se budeme zabývat v kapitole 6.

V této kapitole používáme diagramy definované v UML (Booch et al., 2005; Rumbaugh et al., 2004), který se etabloval jako standardní modelovací jazyk pro objektově orientované modelování. Jazyk UML obsahuje mnoho typů diagramů, a umožňuje proto vytvářet mnoho různých typů systémových modelů. Studie z roku 2007 (Erickson a Siau, 2007) však ukázala, že podle názoru většiny uživatelů jazyka UML lze základní aspekty systému reprezentovat pomocí pěti typů diagramů:

1. Diagramy aktivity, které znázorňují aktivity související s procesem nebo zpracováním dat.
2. Diagramy případů použití, které zobrazují interakce mezi systémem a jeho prostředím.
3. Sekvenční diagramy, které představují interakce mezi aktéry a systémem a mezi komponentami systému.
4. Diagramy tříd, které ukazují objektové třídy v systému a asociace těchto tříd.
5. Stavové diagramy, které popisují, jak systém reaguje na interní a externí události.

Vzhledem k tomu, že zde nemáme místo na rozbor všech typů diagramů UML, zaměříme se na těchto pět klíčových typů diagramů, které se používají v systémovém modelování.

Při vývoji systémových modelů lze často způsob použití grafických značek pružně přizpůsobovat. Pokaždé není nutné se přísně držet detailů dané notace. Úroveň podrobností a striktnost modelu závisí na tom, jak chceme s modelem pracovat. Grafické modely se běžně používají třemi způsoby:

1. Jako základ pro diskuse o existujícím nebo navrženém systému.
2. Jako způsob dokumentace stávajícího systému.
3. Jako podrobný popis systému, pomocí něhož lze generovat implementaci systému.

V prvním případě je účelem modelu stimulovat diskusi mezi softwarovými inženýry, kteří se podílejí na vývoji systému. Modely nemusí být úplné (za předpokladu, že zahrnují klíčová témata diskuse) a modelovací notaci mohou používat neformálně. Tímto způsobem se modely obvykle uplatňují v takzvaném „agilním modelování" (Ambler a Jeffries, 2002). Když modely slouží k dokumentaci, nemusí být úplné, protože účelem může být vývoj modelu pouze pro některé části systému. Tyto modely však musí být správné – měly by správným způsobem používat notaci a přesně popisovat systém.

JAZYK UML

Jazyk UML (Unified Modeling Language) je sada 13 různých typů diagramů, které umožňují modelovat softwarové systémy. Vznikl v 90. letech na základě práce na objektově orientovaném modelování, kdy došlo k integraci podobných objektově orientovaných notací. Hlavní revize (UML 2) byla dokončena roku 2004. Jazyk UML je univerzálně uznáván jako standardní přístup k vývoji modelů softwarových systémů. Objevily se také návrhy pro obecnější systémové modelování.

http://www.SoftwareEngineering-9.com/Web/UML/

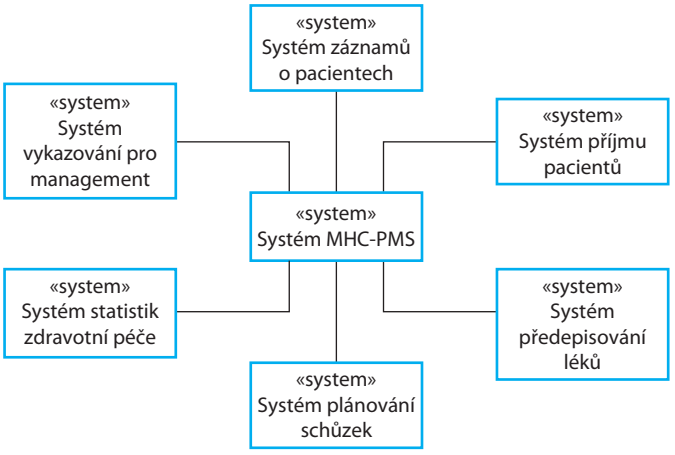
Ve třetím případě, kdy se modely používají v rámci vývojového procesu založeného na modelech, musí být systémové modely úplné i správné. Důvod spočívá v tom, že se podle těchto modelů generuje zdrojový kód systému. Je proto nutné věnovat mimořádnou pozornost tomu, aby nedošlo k záměně podobných symbolů, jako jsou šipky s čárovými a plnými hroty, které mají odlišný význam.

5.1 Kontextové modely

V rané fázi specifikace systému se musíme rozhodnout, kde budou hranice systému. Přitom musíme spolupracovat s osobami zainteresovanými na systému a určit, které funkce bude systém zahrnovat a co bude poskytovat prostředí systému. Můžeme například požadovat, že pro některé podnikové procesy bude implementována automatizovaná podpora, ale další procesy budou fungovat manuálně nebo jejich podporu zajistí jiné systémy. Měli bychom zjistit, kde se funkčnost systému překrývá se stávajícími systémy, a zvolit, zda se budou nové funkce implementovat. Tato rozhodnutí je potřeba přijmout v rané fázi celého procesu, aby se omezily náklady a zkrátil čas na analýzu systémových požadavků a návrhu.

V některých případech vede mezi systémem a jeho prostředím relativně jasná hranice. Pokud například automatizovaný systém nahrazuje stávající ruční nebo počítačový systém, prostředí nového systému zpravidla odpovídá prostředí existujícího systému. V jiných případech je k dispozici větší flexibilita a přesnou hranici mezi systémem a jeho prostředím lze určit během procesu inženýrství požadavků.

Předpokládejme například, že vyvíjíme specifikaci informačního systému o psychiatrických pacientech. Tento systém má uchovávat informace o pacientech navštěvujících psychiatrické kliniky a o léčbě, která jim byla předepsána. Při vývoji specifikace tohoto systému je potřeba se rozhodnout, zda se má systém zaměřit výhradně na shromažďování informací o konzultacích (kdy se osobní informace o pacientech budou získávat z jiných systémů), nebo zda bude rovněž shromažďovat osobní data pacientů. Pokud se informace o pacientech získávají z jiných systémů, je to výhodné z toho hlediska, že nedochází k duplikování dat. Hlavní nevýhoda však spočívá v tom, že použití jiných systémů může zpomalit přístup k informacím. Jestliže tyto systémy nejsou dostupné, nelze systém MHC-PMS použít.



Obrázek 5.1 – Kontext systému MHC-PMS

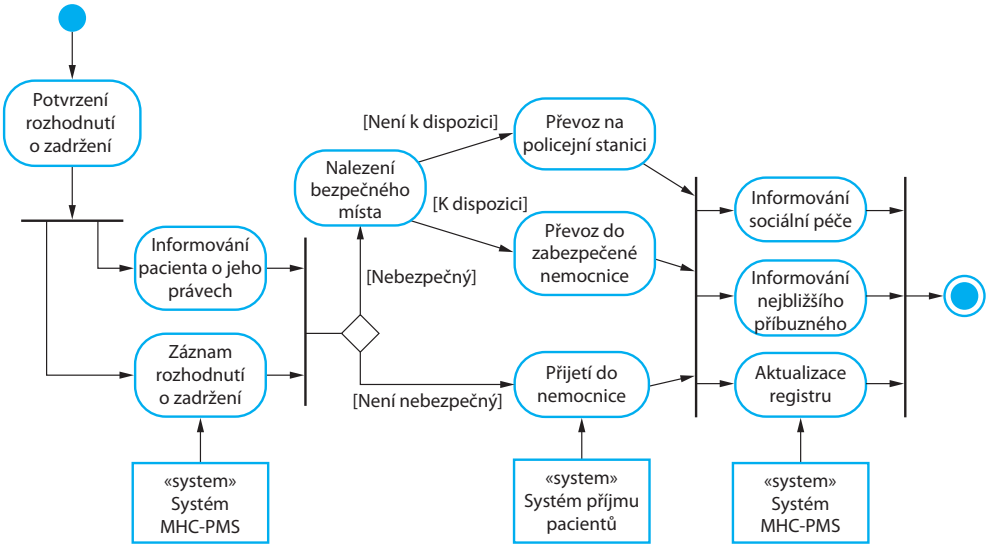
Systémovou hranici nelze definovat nezávisle na hodnotách. Vzhledem ke společenským a organizačním hlediskům může být pozice systémové hranice určena mimotechnickými faktory. Systémová hranice může být například záměrně umístěna tak, aby bylo možné analytický proces kompletně zajistit v jedné lokalitě, může být zvolena tak, aby nebylo nutné konzultovat konkrétního manažera, se kterým se obtížně spolupracuje, může být stanovena tak, aby se zvýšily náklady na systém, a bylo tedy nutné rozšířit divizi systémového vývoje, která systém navrhuje a implementuje.

Jakmile jsou přijata rozhodnutí ohledně hranic systému, je potřeba v rámci analytické aktivity definovat kontext a závislosti systému na jeho prostředí. Prvním krokem této aktivity zpravidla bývá vytvoření jednoduchého modelu architektury.

Obrázek 5.1 znázorňuje jednoduchý kontextový model, který představuje informační systém o pacientech a jiné systémy v jeho prostředí. Z obrázku 5.1 je patrné, že systém MHC-PMS je připojen na systém plánování schůzek a obecnější systém záznamů o pacientech, se kterým sdílí data. Systém je také připojen k systémům na správu vykazování a přidělování nemocničních lůžek a statistickému systému, který shromažďuje informace pro výzkum. Nakonec používá i systém předepisování léků, který pacientům generuje recepty na jejich léky.

Kontextové modely obvykle ukazují, že prostředí obsahuje několik dalších automatizovaných systémů. Neposkytují však informace o typech vztahů mezi systémy v prostředí a o specifikovaném systému. Externí systémy mohou poskytovat data pro daný systém nebo přijímat data z tohoto systému. Mohou se systémem sdílet data nebo je možné připojit je přímo či prostřednictvím sítě a také nemusí být připojeny vůbec. Někdy se nacházejí na stejném místě, případně mohou být v různých budovách. Všechny tyto vztahy mohou mít vliv na požadavky a návrh definovaného systému a je potřeba je zohlednit.

Jednoduché kontextové modely se proto používají spolu s jinými modely, jako jsou například modely obchodních procesů. Tyto modely popisují ruční a automatizované procesy, v nichž se určité softwarové systémy používají.



Obrázek 5.2 – Procesní model nedobrovolného zadržení

Obrázek 5.2 modeluje důležitý systémový proces, který ukazuje způsob, jakým se systém MHC-PMS používá. Pacienti, kteří trpí duševními problémy, mohou být někdy nebezpeční sobě i jiným. Může být proto nutné zadržet je v nemocnici proti jejich vůli, aby je bylo možné léčit. Toto zadržení podléhá přísným zákonným omezením. Rozhodnutí o zadržení pacienta je například nutné pravidelně revidovat, aby lidé nebyli bez dostatečného důvodu dlouhodobě omezováni na svobodě. Jednou z funkcí systému MHC-PMS je kontrolovat, zda jsou implementovány takové zákonné pojistky.

Na obrázku 5.2 vidíme diagram aktivity UML. Diagramy aktivity mají znázornit aktivity, ze kterých sestává systémový proces, a postup, kterým si jednotlivé aktivity vzájemně předávají řízení. Začátek procesu je symbolizován vyplněným kruhem a konec je znázorněn vyplněným kruhem uvnitř jiného kruhu. Obdélníky se zaoblenými rohy reprezentují aktivity, tzn. konkrétní dílčí procesy, které je potřeba provést. Do grafů aktivity lze umístit i objekty. Na obrázku 5.2 jsou znázorněny systémy, které se používají na podporu různých procesů. Z funkce stereotypu UML je patrné, že se jedná o samostatné systémy.

Šipky v diagramu aktivity UML představují tok činností od jedné aktivity k jiné. Plná čára znamená koordinaci aktivit. Když tok z více aktivit vede k plné čáře, pak další postup vyžaduje, aby byly dokončeny všechny tyto aktivity. Když tok od plné čáry směřuje k více aktivitám, lze tyto aktivity provádět paralelně. Na obrázku 5.2 je tedy patrné, že aktivity informování sociální péče a nejbližšího příbuzného pacienta a aktualizace registru zadržení mohou probíhat zároveň.

K šipkám je možné doplnit popisky informující o podmínce, za které daný tok probíhá. Na obrázku 5.2 se nacházejí popisky, které označují toky pro pacienty podle toho, zda jsou nebo nejsou nebezpeční pro společnost. Pacienty nebezpečné pro společnost je potřeba zadržet v zabezpečeném zařízení. Pacienty se sebevražednými sklony, kteří jsou tedy nebezpeční sobě samým, lze však zadržet na vhodném nemocničním oddělení.

5.2 Modely interakcí

Ve všech systémech dochází k nějakému druhu interakcí. Může se jednat o uživatelskou interakci, která spočívá v uživatelském vstupu a výstupu, interakci vyvíjeného systému s jinými systémy nebo interakci mezi komponentami systému. Modelování uživatelské interakce je důležité, protože pomáhá identifikovat uživatelské požadavky. Modelování interakce mezi systémy ukazuje, jaké problémy mohou nastat při komunikaci. Modelování interakce komponent pomáhá porozumět tomu, zda navržená struktura systému pravděpodobně zajistí požadovanou úroveň výkonu a spolehlivosti systému.

V této části se budeme zabývat dvěma souvisejícími přístupy k modelování interakcí:

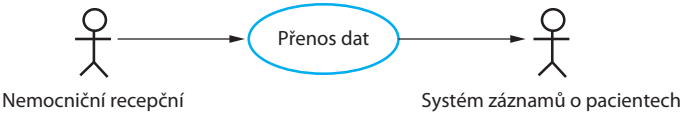
1. Modelování případů použití, které se používá zejména při modelování interakcí mezi systémem a externími aktéry (uživateli nebo jinými systémy).
2. Sekvenční diagramy, které slouží k modelování interakcí mezi systémovými komponentami, ačkoli mohou zahrnovat i externí agenty.

Modely případů použití a sekvenční diagramy předkládají interakce na různé úrovni podrobností, a lze je proto používat společně. Sekvenční diagram může dokumentovat podrobnosti interakcí, které se účastní na vysokoúrovňovém případu použití. Jazyk UML zahrnuje také komunikační diagramy, které dovolují modelovat interakce. Těmto diagramům se zde nebudeme věnovat, protože představují alternativní reprezentaci sekvenčních grafů. Některé nástroje dokážou v praxi generovat komunikační diagram ze sekvenčního diagramu.

5.2.1 Modelování případů použití

Modelování případů použití původně vyvinul Jacobson et al. (1993) v 90. letech a toto modelování se dostalo do první verze jazyka UML (Rumbaugh et al., 1999). Jak jsme diskutovali v kapitole 4, modelování případů použití se široce používá na podporu zjišťování požadavků. Případ použití lze považovat za jednoduchý scénář, který popisuje, co uživatel očekává od systému.

Každý případ použití reprezentuje diskrétní úkol, který zahrnuje externí interakci se systémem. Ve své nejjednodušší formě je případ použití znázorněn formou elipsy. Aktéři, kteří se na případu použití podílejí, jsou pak reprezentováni schematickými figurkami. Obrázek 5.3 představuje případ použití systému MHC-PMS. Jedná se o úkol odeslání dat ze systému MHC-PMS do obecnějšího systému záznamů o pacientech. Tento obecnější systém udržuje souhrnná data o pacientech a nikoli data o jednotlivých konzultacích, která jsou ukládána do systému MHC-PMS.



Obrázek 5.3 – Případ použití přenosu dat

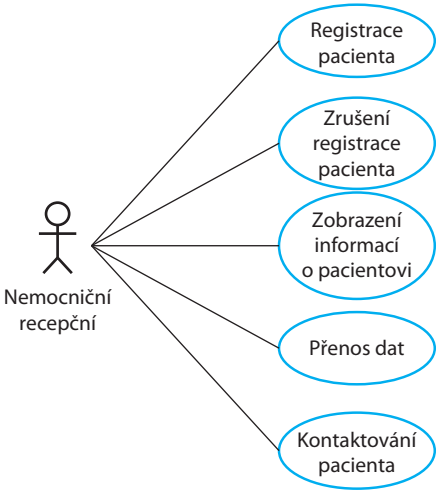
Všimněme si, že v tomto případu použití vystupují dva aktéři: operátor, který přenáší data, a systém záznamů o pacientech. Notace se schematickými panáčky měla sice původně znázorňovat interakci osob, ale nyní umožňuje reprezentovat i jiné externí systémy a hardware. Diagramy případů použití by formálně měly používat čáry bez šipek, protože šipky v jazyce UML označují směr toku zpráv. V případech použití jsou zprávy samozřejmě přenášeny oběma směry. Šipky na obrázku 5.3 však neformálně umožňují označit, že transakci inicializuje nemocniční recepční a data se přenášejí do systému záznamů o pacientech.

Diagramy případů použití poskytují poměrně jednoduchý přehled o interakci. Chceme-li tedy rozumět souvislostem, potřebujeme více podrobností. Tyto detaily mohou mít formu jednoduchého textového popisu, strukturovaného popisu v tabulce nebo sekvenčního diagramu (viz následující rozbor). Nejvhodnější formát můžeme zvolit v závislosti na případu použití a na úrovni podrobností, kterou podle našeho názoru model vyžaduje. Podle názoru autora je nejužitečnější standardní tabulková forma. Obrázek 5.4 představuje tabulkový popis případu použití „Přenos dat".

Jak jsme zmínili v kapitole 4, složené diagramy případů použití znázorňují více různých případů použití. Někdy lze do jediného složeného diagramu případů použití zahrnout všechny možné interakce v systému. Jindy to však vzhledem k počtu případů použití nemusí být možné. V takových scénářích můžeme vytvořit několik diagramů, z nichž každý představuje související případy použití. Obrázek 5.5 například znázorňuje všechny případy použití v systému MHC-PMS, na kterých se podílí aktér „Nemocniční recepční".

MHC-PMS: PŘENOS DAT	
Aktéři	Nemocniční recepční, systém záznamů o pacientech (PRS)
Popis	Recepční může přenést data ze systému MHC-PMS do databáze univerzálního systému záznamů o pacientech, který provozuje ministerstvo zdravotnictví. Přenášet lze buď aktualizované osobní údaje (adresa, telefonní číslo atd.) nebo souhrnné informace o diagnóze a léčení pacienta.
Data	Osobní údaje o pacientovi, souhrn léčby
Impuls	Uživatelský příkaz zadaný nemocničním recepčním
Odpověď	Potvrzení o aktualizaci systému PRS
Poznámky	Recepční musí mít odpovídající bezpečnostní oprávnění pro přístup k informacím o pacientovi a do systému PRS.

Obrázek 5.4 – Tabulkový popis případu použití „Přenos dat"



Obrázek 5.5 – Případy použití související s rolí „nemocniční recepční"

5.2.2 Sekvenční diagramy

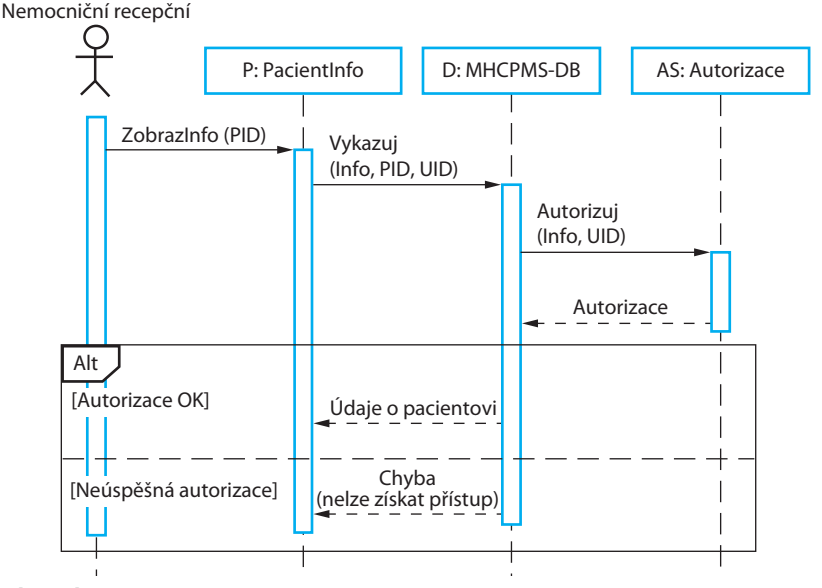
Sekvenční diagramy v jazyce UML primárně slouží k modelování interakcí mezi aktéry a objekty v systému a mezi samotnými objekty. Jazyk UML poskytuje bohatou syntaxi sekvenčních diagramů, díky níž lze modelovat mnoho různých typů interakcí. Nemáme zde místo, abychom se zabývali všemi možnostmi, takže zůstaneme u základů tohoto typu diagramu.

Jak vyplývá z názvu, sekvenční diagram znázorňuje pořadí interakcí, k nimž dochází během určitého případu použití nebo instance případu použití. Obrázek 5.6 představuje příklad sekvenčního diagramu, z něhož lze vysledovat základy příslušné notace. Tento diagram modeluje interakce, které se uplatňují v případu použití Zobrazení pacienta, kdy může nemocniční recepční zobrazit některé údaje o pacientovi.

Zúčastněné objekty a aktéři jsou uvedeni v horní části diagramu a svisle od nich vede tečkovaná čára. Interakci mezi objekty symbolizují šipky s popisky. Obdélník na tečkované čáře znamená životní čáru dotčeného objektu (tj. čas, kdy se instance objektu účastní na výpočtu). Sekvenci interakcí lze číst shora dolů. Poznámky na šipkách popisují volání objektů, jejich parametry a návratové hodnoty. V tomto příkladu je také znázorněn zápis označující alternativy. Používá se rámeček s názvem „alt" s podmínkami, které jsou uzavřeny v hranatých závorkách.

Obrázek 5.6 lze číst takto:

1. Nemocniční recepční aktivuje metodu ZobrazInfo v instanci P třídy objektů PacientInfo a poskytne identifikátor pacienta PID. P je objekt uživatelského rozhraní, který se zobrazuje jako formulář s údaji o pacientovi.
2. Instance P zavolá databázi, která vrátí požadované informace. Přitom poskytne identifikátor recepčního, který umožní bezpečnostní kontrolu (v této fázi se nezajímáme o to, odkud tento identifikátor UID pochází).



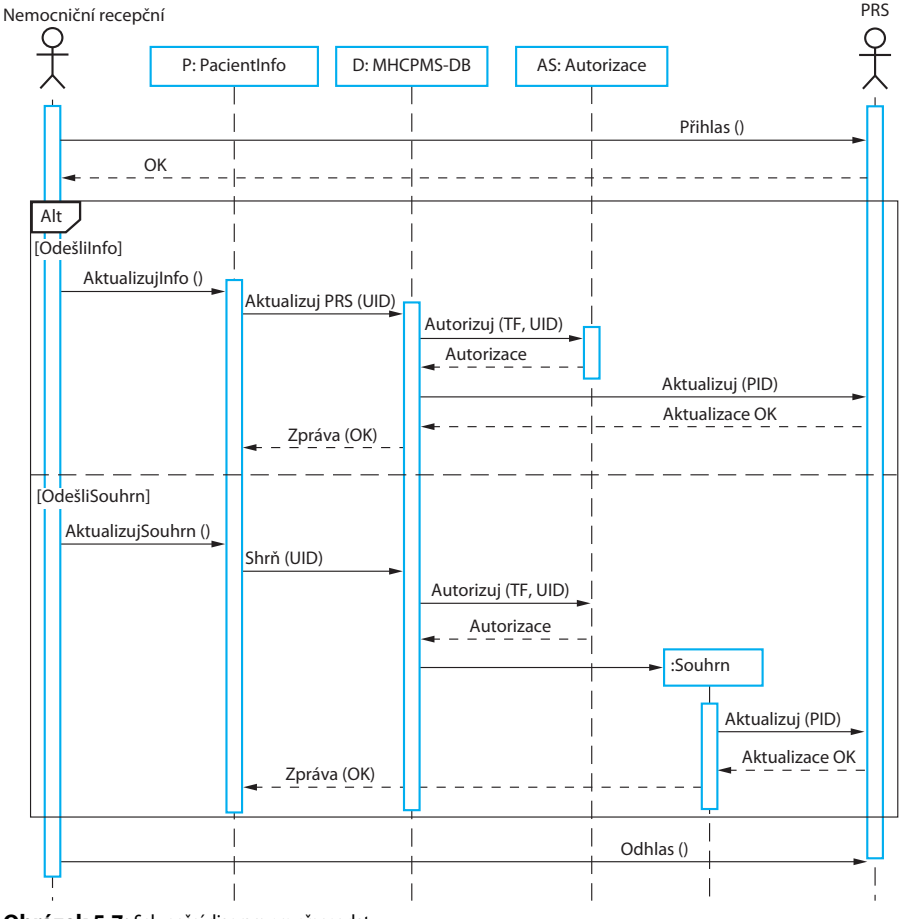
Obrázek 5.6 – Sekvenční diagram pro Zobrazení informací o pacientovi

3. Databáze pomocí autorizačního systému zkontroluje, zda je uživatel oprávněn provést danou akci.

4. Pokud ano, jsou vráceny informace o pacientovi, které vyplní formulář na obrazovce uživatele. Jestliže autorizace není úspěšná, zobrazí se chybová zpráva.

Obrázek 5.7 je dalším příkladem sekvenčního diagramu ze stejného systému, který ilustruje dvě další funkce. Jedná se o přímou komunikaci mezi aktéry v systému a vytvoření objektů jako součást sekvence operací. V tomto příkladu je vytvořen objekt typu Souhrn, který uchovává souhrnná data pro odeslání do systému PRS (systému záznamů o pacientech). Tento diagram lze číst následovně:

1. Recepční se přihlásí do systému PRS.
2. K dispozici jsou dvě možnosti. Dovolují přímý přenos aktualizovaných údajů o pacientovi do systému PRS a přenos souhrnných zdravotních dat ze systému MHC-PMS do systému PRS.
3. V obou případech se pomocí autorizačního systému kontrolují oprávnění recepčního.
4. Osobní informace lze přenášet přímo z objektu uživatelského rozhraní do systému PRS. Případně je možné vytvořit souhrnný záznam z databáze a následně přenést tento záznam.
5. Při dokončení přenosu vydá systém PRS stavovou zprávu a uživatel se odhlásí.



Obrázek 5.7: Sekvenční diagram pro přenos dat

Pokud sekvenční diagramy neslouží ke generování kódu nebo k podrobnému dokumentování, nemusí zahrnovat všechny interakce. Vyvíjíme-li na začátku vývojového procesu systémové modely, které budou podporovat inženýrství požadavků a vysokoúrovňový návrh, budou obsahovat mnoho interakcí, které závisí na volbách při implementaci. Například na obrázku 5.7 lze odložit rozhodnutí o tom, jak se bude získávat identifikátor uživatele při kontrole autorizace. Při implementaci se může uplatnit interakce s objektem Uživatel, ale v této fázi to není důležité, a proto to do sekvenčního diagramu není nutné umístit.

OBJEKTOVĚ ORIENTOVANÁ ANALÝZA POŽADAVKŮ

Při objektově orientované analýze požadavků se entity reálného světa modelují pomocí objektových tříd. Můžeme vytvářet různé typy objektových modelů, které ukazují, jak vzájemně souvisejí objektové třídy, jak lze agregací objektů vytvářet jiné objekty, jak objekty interagují s jinými objekty atd. Každý z těchto modelů předkládá jedinečné informace o specifikovaném systému.

http://www.SoftwareEngineering-9.com/Web/OORA/

5.3 Strukturní modely

Strukturní modely softwaru znázorňují organizaci systému z hlediska komponent, které systém tvoří, a jejich vztahů. Strukturní modely mohou být statické, které představují strukturu návrhu systému, nebo dynamické, které ukazují organizaci systému při jeho činnosti. Jedná se o různé pohledy – dynamická organizace systému jako sady interagujících vláken se může značně lišit od statického modelu systémových komponent.

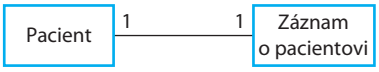
Strukturní modely systému lze vytvořit během diskuse a návrhu systémové architektury. Návrh architektury představuje zvláště důležité téma softwarového inženýrství. Při prezentaci modelů architektury lze použít diagramy komponent, balíčků a nasazení jazyka UML. Jednotlivými aspekty softwarové architektury a modelování architektury se budeme zabývat v kapitolách 6, 18 a 19. V této části se zaměříme na použití diagramů tříd při modelování statické struktury objektových tříd v softwarovém systému.

5.3.1 Diagramy tříd

Diagramy tříd se používají při vývoji objektově orientovaného modelu systému. Ukazují třídy v systému a jejich asociace. Objektovou třídu si můžeme volně představit jako obecnou definici jednoho druhu systémového objektu. Asociace (přidružení) je vazba mezi třídami, která informuje o tom, že tyto třídy nějak souvisejí. Každá třída proto může mít určité informace o svých přidružených třídách.

Při vývoji modelů během raných fází procesu softwarového inženýrství reprezentují objekty určité prvky reálného světa, jako je pacient, recept, lékař atd. Jak postupuje vývoj implementace, obvykle je nutné definovat její další objekty, které poskytují požadované funkce systému. Zde se soustředíme na modelování objektů reálného světa v rámci zpracování požadavků nebo procesů počátečních návrhů softwaru.

Diagramy tříd v jazyce UML lze vyjádřit na různé úrovni podrobností. Když vyvíjíme model, v první fázi se obvykle podíváme do reálného světa, identifikujeme klíčové objekty a reprezentujeme je formou tříd. Nejjednodušší způsob, jak třídy znázornit, je zapsat jejich název do rámečku. Můžeme také jednoduše zaznamenat existenci přidružení tak, že mezi třídami nakreslíme čáru. Obrázek 5.8 například představuje jednoduchý diagram tříd, který znázorňuje dvě třídy: Pacient a Záznam o pacientovi, mezi nimiž existuje asociace.



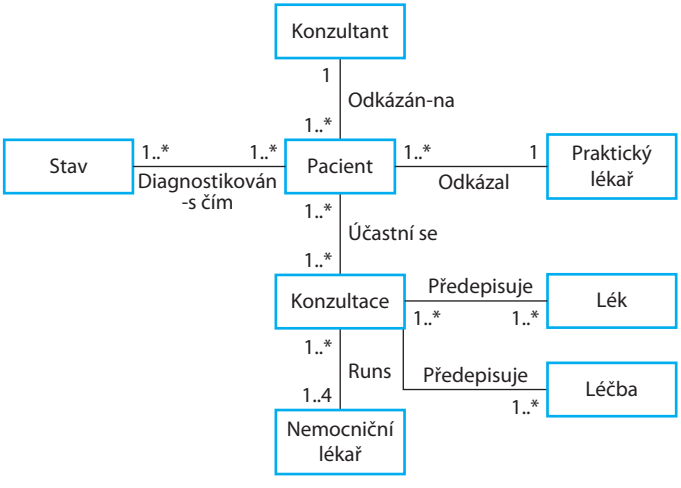
Obrázek 5.8 – Třídy UML a jejich asociace

Obrázek 5.8 ilustruje další funkci diagramů tříd – schopnost znázornit, kolik objektů se na přidružení podílí. V tomto příkladu je každý konec přidružení označen číslem 1, což znamená, že mezi objekty těchto tříd existuje vztah typu 1:1. To znamená, že každý pacient má přesně jeden záznam a každý záznam uchovává informace právě o jednom pacientovi. Jak je patrné z dalších příkladů, mohou existovat i jiné četnosti. Můžeme definovat, že se na vztahu podílí přesný počet objektů, nebo můžeme symbolem * (viz obrázek 5.9) označit, že se na asociaci účastní neomezený počet objektů.

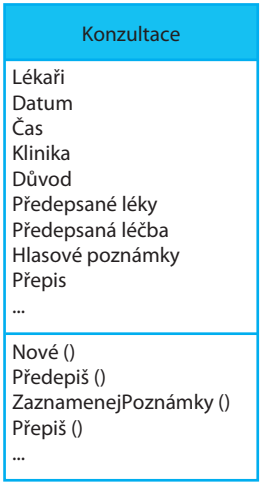
Obrázek 5.9 rozvíjí tento typ diagramu tříd a ukazuje, že objekty třídy Pacient se také podílejí na vztazích s mnoha jinými třídami. Z tohoto příkladu je zřejmé, že přidružení lze pojmenovat, aby čtenáři grafu získali představu o typech existujících vztahů. Jazyk UML rovněž dovoluje specifikovat roli objektů, které se na asociaci podílejí.

Na této úrovni podrobností vypadají diagramy tříd jako sémantické datové modely. Sémantické datové modely se používají při návrhu databází. Znázorňují datové entity, jejich přidružené atributy a vztahy mezi těmito entitami. Tento přístup k modelování poprvé navrhl v polovině 70. let Chen (1976). Od té doby vzniklo několik variant (Codd, 1979; Hammer a McLeod, 1981; Hull a King, 1987), které sdílejí stejnou základní formu.

Jazyk UML nezahrnuje konkrétní notaci pro toto databázové modelování, protože předpokládá objektově orientovaný vývojový proces a modeluje data pomocí objektů a jejich vztahů. Pomocí jazyka UML lze však reprezentovat i sémantický datový model. Entity sémantického datového modelu si můžeme představit jako zjednodušené objektové třídy (nemají žádné operace), atributy jako atributy objektové třídy a vztahy jako pojmenovaná přidružení mezi objektovými třídami.



Obrázek 5.9 – Třídy a asociace v systému MHC-PMS



Obrázek 5.10 – Třída konzultace

Při znázorňování asociací mezi třídami je výhodné reprezentovat tyto třídy co nejjednodušším způsobem. Chceme-li třídy definovat podrobněji, doplníme informace o jejich atributech (vlastnosti objektu) a operacích (činnostech, které od objektu požadujeme). Například objekt Pacient bude mít atribut Adresa a můžeme k němu přidat i operaci označenou ZměňAdresu. Tato operace se bude volat, když pacient oznámí, že se přestěhoval na jinou adresu. V jazyce UML se atributy a operace znázorňují rozšířením jednoduchého obdélníku, který označuje třídu. To je patrné na obrázku 5.10, kde:

1. Název objektové třídy je umístěn v horní části.
2. Atributy třídy jsou ve střední části. Nesmí zde chybět názvy atributů a volitelně mohou být uvedeny jejich typy.
3. Operace (které se v jazyce Java a jiných objektově orientovaných programovacích jazycích nazývají metody) přidružené k objektové třídě jsou v dolní části obdélníka.

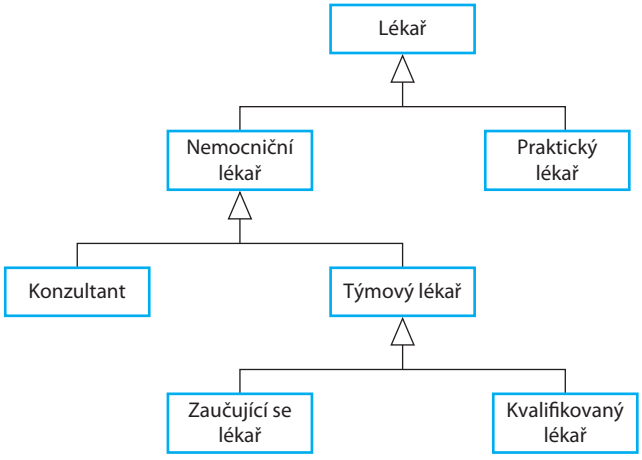
Obrázek 5.10 znázorňuje možné atributy a operace třídy Konzultace. V tomto příkladu předpokládáme, že lékaři nahrávají hlasové poznámky, které jsou později přepisovány za účelem zaznamenání podrobností o konzultaci. Při předepsání léku musí příslušný lékař použít metodu Předepiš, která generuje elektronický recept.

5.3.2 Generalizace

Generalizace je běžná metoda, díky níž zvládáme složitost světa. Místo toho, abychom se seznamovali s podrobnými vlastnostmi každé entity, se kterou se setkáváme, řadíme tyto entity do obecnějších tříd (zvířata, auta, domy atd.) a učíme se vlastnosti těchto tříd. Díky tomu můžeme odvodit, že různí členové těchto tříd mají některé společné vlastnosti (například veverky a krysy jsou hlodavci). Můžeme učinit obecné závěry, které se vztahují na všechny členy třídy (například všichni hlodavci mají zuby, kterými dokážou hlodat).

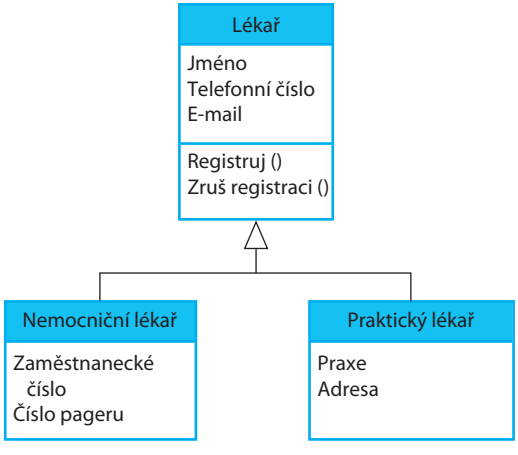
V modelovacích systémech je často užitečné prozkoumat třídy v systému a vyhodnotit, zda neexistuje prostor pro jejich generalizaci. To znamená, že společné informace lze uchovávat jen na jednom místě. Jedná se o vhodný návrhový postup, protože pokud jsou navrženy změny, není nutné kontrolovat

všechny třídy v systému, zda na ně příslušná změna bude mít vliv. V objektově orientovaných jazycích, jako je Java, se generalizace implementuje pomocí mechanismů dědění tříd, které jsou součástí jazyka.



Obrázek 5.11 – Hierarchie generalizace

Jazyk UML má konkrétní typ asociace, který označuje generalizaci (viz obrázek 5.11). Generalizace se znázorňuje pomocí šipky, která směřuje k obecnější třídě. Z toho je zřejmé, že praktické lékaře a nemocniční lékaře lze zobecnit jako lékaře a že existují tři typy nemocničních lékařů – ti, kteří nedávno absolvovali lékařskou fakultu a je nutné na ně dohlížet (zaučující se lékař), ti, kteří mohou pracovat bez dozoru v rámci konzultačního týmu (registrovaný lékař), a konzultanti, což jsou zkušení lékaři s plnými rozhodovacími pravomocemi.



Obrázek 5.12 – Hierarchie generalizace s doplněnými podrobnostmi

Při generalizaci se atributy a operace přidružené ke třídám vyšší úrovně přidružují také ke třídám nižší úrovně. Třídy nižší úrovně jsou v zásadě podtřídami, které dědí atributy a operace od svých nadtříd. Tyto třídy nižší úrovně pak doplňují konkrétnější atributy a operace. Například všichni lékaři mají jméno a telefonní číslo. Všichni nemocniční lékaři mají zaměstnanecké číslo a oddělení, kde jsou zaměstnáni,

ale praktičtí lékaři tyto atributy nemají, protože pracují samostatně. Místo toho však používají název a adresu své praxe. To je patrné na obrázku 5.12, který znázorňuje část hierarchie generalizace, kterou jsme doplnili o atributy tříd. Operace přidružené ke třídě Lékař mají zajistit registraci a zrušení registrace daného lékaře v systému MHC-PMS.

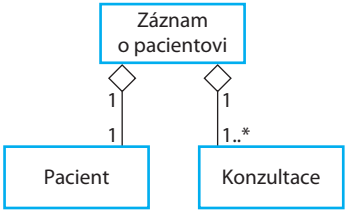
5.3.3 Agregace

Objekty reálného světa se často skládají z různých částí. Například studijní materiály ke kurzu mohou zahrnovat knihu, prezentace PowerPoint, testy a doporučení další literatury. Někdy je potřeba zajistit, aby model systému toto složení dokumentoval. Jazyk UML poskytuje speciální typ přidružení tříd označovaný jako agregace. Znamená, že jeden objekt (celek) se skládá z jiných objektů (částí). Při znázornění používáme kosočtverec vedle třídy, která reprezentuje celek. To je patrné na obrázku 5.13, který znázorňuje, že záznam o pacientovi obsahuje položku Pacient a neomezený počet Konzultací.

5.4 Behaviorální modely

Behaviorální modely znázorňují dynamické chování systému při jeho činnosti. Ukazují, co se děje nebo má dít, když systém reaguje na impulsy ze svého prostředí. Tyto stimuly můžeme zařadit do dvou typů:

1. *Data* – doručována jsou určitá data, která musí systém zpracovat.
2. *Události* – nastávají jisté události, které aktivují systémové zpracování. Události mohou mít přidružena data, ale neplatí to vždy.



Obrázek 5.13 – Přidružení typu agregace

DIAGRAMY DATOVÉHO TOKU

Diagramy datového toku (DFD – data-flow diagram) jsou systémové modely, které znázorňují funkční perspektivu, kde každá transformace reprezentuje jedinou funkci nebo proces. Tyto diagramy umožňují znázornit, jak data postupují sekvencí kroků svého zpracování. Krokem zpracování může být například odfiltrování duplicitních záznamů z databáze zákazníků. Data se v každém kroku transformují a následně postupují do další fáze. Tyto kroky či transformace zpracování reprezentují softwarové procesy či funkce, kdy lze pomocí diagramů datového toku dokumentovat návrh softwaru.

http://www.SoftwareEngineering-9.com/Web/DFDs

Mnoho podnikových systémů patří mezi systémy zpracování dat, které primárně slouží k manipulaci s daty. Jsou řízeny vstupem dat do systému a poměrně v malé míře jsou řízeny zpracováním externích událostí. Zpracování v tomto případě zahrnuje sekvenci akcí s daty a generování výstupu. Například systém účtování telefonních služeb přijímá informace o voláních, která zákazník uskutečnil, počítá cenu těchto volání a generuje fakturu, která bude příslušnému zákazníkovi odeslána. Oproti tomu systémy

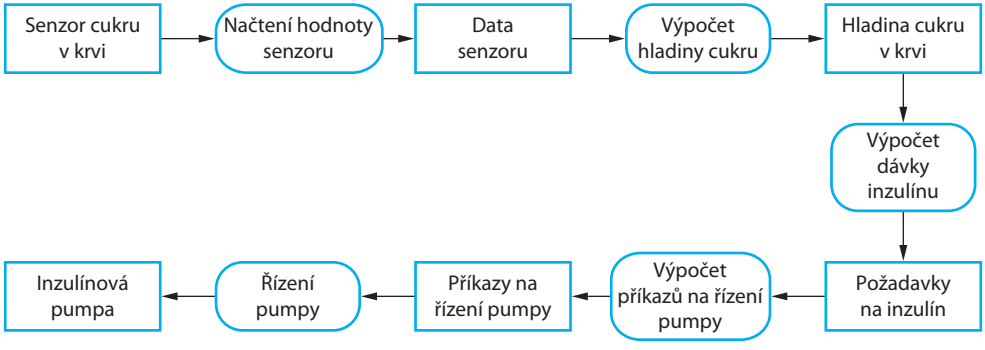
fungující v reálném čase jsou často řízeny událostmi a pouze minimálně zpracovávají data. Například systém ústředny pro pevnou telefonní síť reaguje na různé události – například na zvednuté sluchátko generováním oznamovacího tónu nebo na stisknutí tlačítek telefonu zaznamenáním telefonního čísla.

5.4.1 Modelování řízené daty

Modely řízené daty znázorňují sekvenci akcí, které se podílejí na zpracování vstupních dat a generování příslušného výstupu. Hodí se zejména k analýze požadavků, protože umožňují vizualizovat kompletní postup zpracování v systému. To znamená, že ukazují celou sekvenci akcí, která probíhá od příjmu vstupu po generování odpovídajícího výstupu, který představuje odpověď systému.

Modely řízené daty patřily mezi první grafické softwarové modely. V 70. letech byly v rámci strukturovaných metod, jako je DeMarcova strukturovaná analýza (DeMarco, 1978), zavedeny diagramy datového toku (DFD), které umožňovaly ilustrovat kroky zpracování v systému. Modely datového toku jsou užitečné, protože sledování a dokumentování toho, jak data související s určitým procesem procházejí systémem, pomáhá analytikům a návrhářům porozumět fungování systému. Diagramy datového toku jsou jednoduché a intuitivní a obvykle je lze vysvětlit potenciálním uživatelům systému, kteří se pak mohou účastnit na validaci modelu.

Jazyk UML neposkytuje podporu diagramů datového toku, protože byly původně navrženy a používaly se k modelování zpracování dat. Důvod spočívá v tom, že diagramy datového toku se zaměřují na systémové funkce a nerozpoznávají systémové objekty. Vzhledem k tomu, že systémy řízené daty jsou v podnicích velmi běžné, však jazyk UML 2.0 zavedl diagramy aktivity, které se podobají diagramům datového toku. Například obrázek 5.14 znázorňuje řetězec akcí zpracování, který se uplatňuje u softwaru inzulínové pumpy. Na tomto diagramu jsou patrné kroky zpracování (reprezentované jako aktivity) a data směřující mezi těmito kroky (reprezentovaná jako objekty).



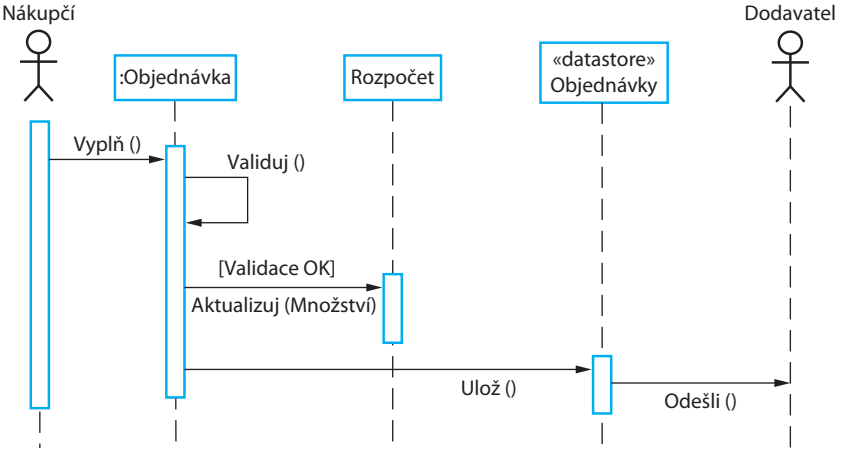
Obrázek 5.14 – Model aktivity fungování inzulínové pumpy

Alternativní způsob znázornění sekvence zpracování v systému využívá sekvenční diagramy UML. Již jsme viděli, jak lze tyto diagramy uplatnit při modelování interakcí. Pokud je však nakreslíme tak, aby zprávy proudily pouze zleva doprava, pak znázorňují sekvenční zpracování dat v systému. To je patrné na obrázku 5.15, kde se používá sekvenční model zpracování objednávky a jejího odeslání dodavateli. Sekvenční modely se zaměřují na objekty v systému, zatímco diagramy datového toku zvýrazňují funkce. Odpovídající diagram datového toku pro zpracování objednávky je k dispozici na webových stránkách knihy.

5.4.2 Modelování řízené událostmi

Modelování řízené událostmi znázorňuje, jak systém reaguje na externí a interní události. Vychází z předpokladu, že systém má omezený počet stavů a událostí (stimuly) mohou způsobit přechod z jednoho stavu do jiného. Když například systém řídící ventil přijme příkaz od operátora (stimul), může přejít ze stavu „Otevřený ventil" do stavu „Zavřený ventil". Tento pohled na systém je vhodný zejména pro systémy fungující v reálném čase. Modelování založené na událostech se objevilo v metodách návrhu pro systémy v reálném čase, které navrhli například Ward a Mellor (1985) a Harel (1987, 1988).

Jazyk UML podporuje modelování řízené událostmi díky stavovým diagramům, které jsou založeny na stavových grafech (Harel, 1987, 1988). Stavové diagramy znázorňují stavy a události systému, které způsobují přechod mezi jednotlivými stavy. Nedokumentují tok dat v rámci systému, ale mohou zahrnovat další informace o výpočtech, které probíhají v jednotlivých stavech.



Obrázek 5.15 – Zpracování objednávky

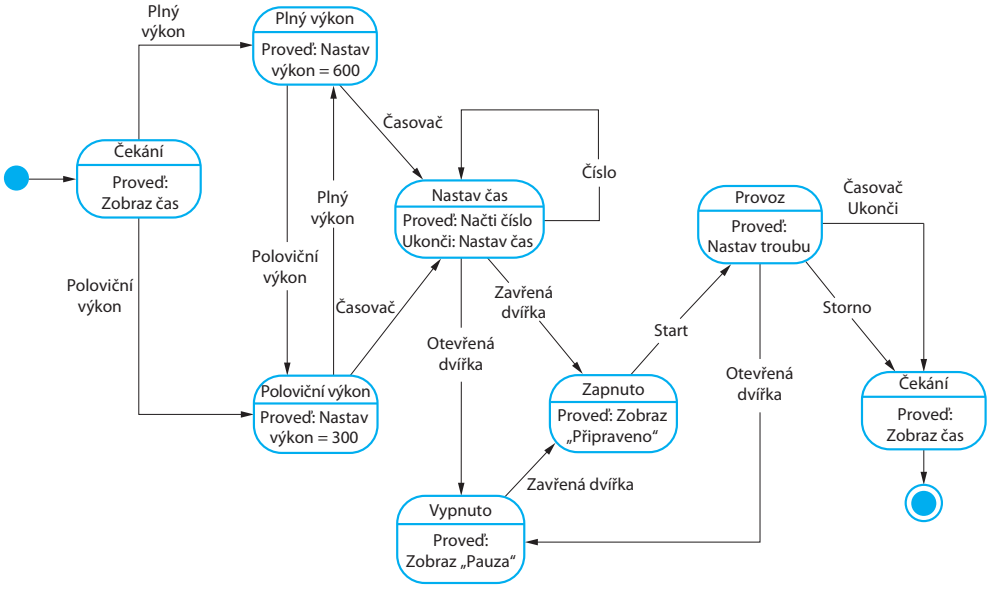
K ilustraci modelování řízeného událostmi nám poslouží příklad řídicího softwaru velmi jednoduché mikrovlnné trouby. Skutečné mikrovlnné trouby jsou sice mnohem složitější než tento systém, ale uvedený příklad je lépe srozumitelný. Toto jednoduché zařízení má vypínač, který umožní zvolit poloviční nebo plný výkon, číselnou klávesnici, která slouží k zadání času vaření, a alfanumerický displej.

Předpokládejme, že sekvence akcí při používání této trouby vypadá takto:

1. Vyberte úroveň výkonu (buď poloviční, nebo plný výkon).
2. Zadejte čas vaření na numerické klávesnici.
3. Stiskněte tlačítko Start a jídlo se bude připravovat po zadanou dobu.

Z bezpečnostních důvodů by mikrovlnná trouba neměla fungovat, když jsou otevřená dvířka. Po dokončení vaření se pak ozve zvukový signál. Trouba je vybavena velmi prostým alfanumerickým displejem, který dokáže zobrazit různá upozornění a varovné zprávy.

Ve stavových diagramech UML jsou stavy systému reprezentovány obdélníky se zaoblenými rohy. Tyto obdélníky mohou obsahovat stručný popis (po klíčovém slově „Proveď") akcí, které jsou v daném stavu realizovány. Popsané šipky reprezentují stimuly, které vynucují přechod z jednoho stavu do jiného. Počáteční a koncové stavy lze znázornit pomocí vyplněných kruhů stejně jako v diagramech aktivity.



Obrázek 5.16 – Stavový diagram mikrovlnné trouby

Na obrázku 5.16 je patrné, že systém začíná svou činnost ve stavu čekání a zpočátku reaguje na přepínač plného či poloviční výkonu. Uživatelé se mohou po výběru jedné z těchto možností rozmyslet a zvolit druhou možnost. Poté nastaví čas; jsou-li zavřena dvířka, lze použít tlačítko Start. Stisknutí tohoto tlačítka zahájí činnost trouby a po nastavenou dobu probíhá vaření. Tím končí cyklus vaření a systém přejde zpět do stavu čekání.

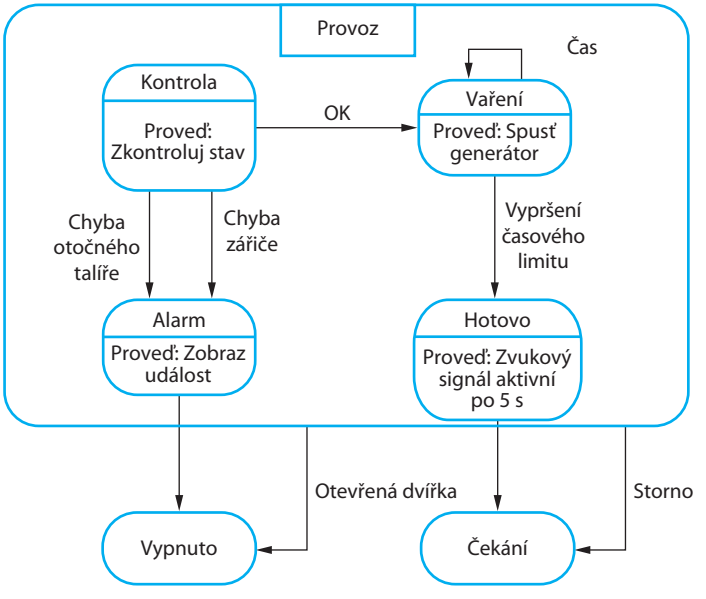
Stav	Popis
Čekání	Trouba čeká na vstup. Displej zobrazuje aktuální čas.
Poloviční výkon	Výkon trouby je nastaven na 300 wattů. Displej zobrazuje zprávu „Poloviční výkon“.
Plný výkon	Výkon trouby je nastaven na 600 wattů. Displej zobrazuje zprávu „Plný výkon“.
Nastavení času	Čas vaření je nastaven podle hodnoty uživatelského vstupu. Displej zobrazí vybraný čas vaření a při nastavení času se aktualizuje.
Vypnuto	Činnost trouby je z bezpečnostních důvodů vypnuta. Vnitřní osvětlení trouby je zapnuto. Displej zobrazuje zprávu „Nepřipraveno“.
Zapnuto	Činnost trouby je zapnuta. Vnitřní osvětlení trouby je vypnuto. Displej zobrazuje zprávu „Připraveno k vaření“.
Provoz	Trouba je v činnosti. Vnitřní osvětlení trouby je zapnuto. Displej zobrazuje odpočítávání časovače. Při dokončení vaření se na pět sekund ozve zvukový signál. Osvětlení trouby je zapnuto. Spolu se zvukem se na displeji zobrazuje „Vaření je dokončeno“.

Impuls	Popis
Poloviční výkon	Uživatel stiskl tlačítko poloviční výkonu.
Plný výkon	Uživatel stiskl tlačítko plného výkonu.
Časovač	Uživatel stiskl jedno z tlačítek časovače.
Číslo	Uživatel stiskl číselnou klávesu.
Otevřená dvířka	Spínač dvířek trouby není sepnutý.
Zavřená dvířka	Spínač dvířek trouby je sepnutý.
Start	Uživatel stiskl tlačítko Start.
Storno	Uživatel stiskl tlačítko Storno.

Obrázek 5.17 – Stavy a stimuly mikrovlnné trouby

Notace UML umožňuje popsat aktivitu, která se odehrává v příslušném stavu. V podrobné specifikaci systému je nutné uvést více podrobností o stimulu i stavech systému. To je patrné na obrázku 5.17, který obsahuje tabulkový popis všech stavů a způsob generování stimulů, které vedou ke změnám stavů.

Problém s modelováním řízeným stavy spočívá v tom, že počet možných stavů rychle roste. V případě modelování velkých systémů je tedy nutné v modelu skrýt některé detaily. Jedna metoda je založena na koncepci superstavu, který zapouzdřuje několik samostatných stavů. Tento superstav ve vysokoúrovňovém modelu vypadá jako jediný stav, ale následně jej lze rozbalit a zobrazit více podrobností v samostatném diagramu. Tento princip si můžeme předvést na stavu Provoz z obrázku 5.16. Jedná se o superstav, který je možné rozbalit, jak je zřejmé na obrázku 5.18.



Obrázek 5.18 – Provoz mikrovlnné trouby

Stav Provoz zahrnuje několik dílčích stavů. Graf ukazuje, že provoz začíná kontrolou stavu. V případě zjištění nějakých problémů zazní zvukový signál a provoz je zastaven. Při vaření se na určenou dobu spouští generátor mikrovln. Po dokončení přípravy jídla se ozve zvukový signál. Jestliže uživatel při provozu trouby otevře dvířka, systém přejde do vypnutého stavu, jak je patrné na obrázku 5.16.

5.5 Inženýrství řízené modely

Inženýrství řízené modely (model-driven engineering – MDE) je přístup k vývoji softwaru, kde základním výstupem vývojového procesu jsou namísto programů modely (Kent, 2002; Schmidt, 2006). Programy, které lze spustit na hardwarové či softwarové platformě, jsou pak automaticky generovány z modelů. Zastánci inženýrství řízeného modely argumentují, že zvyšuje úroveň abstrakce v softwarovém inženýrství, takže se inženýři nadále nemusí zabývat podrobnostmi programovacích jazyků nebo specifiky provozních platforem.

Inženýrství řízené modely vychází z architektury řízené modely (model-driven architecture – MDA), kterou navrhla skupina OMG (Object Management Group) roku 2001 jako nové paradigma vývoje softwaru. Inženýrství a architektura řízené modely se často považují za synonyma. Podle názoru autora však MDE má širší záběr než MDA. Jak rozebereme dále v této části, MDA se v rámci vývoje softwaru zaměřuje na fázi návrhu a implementace, zatímco MDE se zabývá všemi aspekty procesu softwarového inženýrství. Témata jako inženýrství požadavků založené na modelech, softwarové procesy pro vývoj založený na modelech a testování založené na modelech jsou tedy částí MDE, ale v současnosti nepatří do oblasti MDA.

MDA se sice používá od roku 2001, ale inženýrství založené na modelech je stále v počáteční fázi svého rozvoje a není jasné, zda bude mít na praxi softwarového inženýrství zásadní dopad. Hlavní argumenty pro a proti MDE zní takto:

1. *Pro MDE* – inženýrství založené na modelech inženýrům umožňuje, aby o systémech uvažovali na vysoké úrovni abstrakce a mohli přitom ignorovat podrobnosti jejich implementace. Tím se snižuje pravděpodobnost chyb, urychluje se proces návrhu a implementace a lze vytvářet opakovaně použitelné aplikační modely nezávislé na platformě. Díky použití výkonných nástrojů lze ze stejného modelu generovat implementace systému pro různé platformy. Chceme-li tedy systém adaptovat pro některou novou technologickou platformu, stačí pouze napsat překladač pro danou platformu. Když je tento překladač dostupný, je možné modely nezávislé na platformě rychle implementovat pro tuto novou platformu.
2. *Proti MDE* – jak jsme zmínili dříve v této kapitole, modely představují vhodný základ pro diskuse o návrhu softwaru. Z toho však ale vždy nevyplývá, že abstrakce podporované modelem mohou být zároveň vhodnými abstrakcemi pro implementaci. Můžeme tedy vytvořit neformální návrhové modely, ale pak systém implementovat pomocí komerčně dostupného a konfigurovatelného balíčku. Argumenty nezávislosti na platformě navíc platí pouze pro velké systémy s dlouhou životností, u kterých platforma v průběhu životnosti systému zastarává. U této třídy systému však víme, že implementace nepředstavuje hlavní problém – významnější jsou hlediska inženýrství požadavků, bezpečnosti a spolehlivosti, integrace se staršími systémy a testování.

Skupina OMG na svých webových stránkách (http://www.omg.org/mda/products_success.htm) zmiňuje významné úspěchy MDE a tento přístup používají i velké společnosti jako IBM a Siemens. Techniky se úspěšně uplatňují při vývoji velkých softwarových systémů s dlouhou životností, jako jsou systémy řízení letového provozu. V době psaní této knihy se však přístupy řízené modely v softwarovém inženýrství široce nepoužívají. Podobně jako u formálních metod softwarového inženýrství, kterými se budeme