ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

Katedra informačního inženýrství

Metody řízení softwarových projektů využívající moderní paradigmata

Disertační práce

Autor: Ing. Robert Pergl

Školitel: Prof. Ing. Ivan Vrana, DrSc.

Obor: Informační management

Praha 2008

Poděkování

Rád bych vyjádřil svoje nejupřímnější díky všem, kteří přispěli svojí pomocí ke vzniku této práce, jmenovitě především svému školiteli prof. Ing. Ivanu Vranovi, DrSc., prof. RNDr. Jiřímu Vaníčkovi, CSc., doc. Ing. Vojtěchu Merunkovi, Ph.D., Mgr. Františku Jirsovi a dalším odborníkům ze společnosti Deloitte Advisory, s.r.o a podobně i Ing. Petru Štěpánkovi a dalším z firmy e-Fractal, s.r.o.

Obsah

Seznam vyobrazení a tabulek **5**

Vyobrazení 5

Tabulky 6

1 Úvod a motivace **7**

2 Cíl **10**

3 Metodika **11**

4 Terminologie **13**

5 Literární rešerše **15**

5.1 Vývoj softwarového inženýrství 15

5.2 Tvrdý metodický přístup – Unified Process 23

5.3 Měkký metodický přístup – Agilní metodologie 28

5.4 Snahy o formalizaci v disciplíně softwarového inženýrství 62

6 Strukturovaný přehled procesu vývoje softwaru **67**

6.1 Úvod 67

6.2 Příprava 68

6.3 Vývoj 70

6.4 Nasazení 73

6.5 Správa a podpora 77

7 Formalizace softwarového projektu z hlediska aparátu systémového modelování **81**

7.1 Úvod 81

7.2 Cíl 81

7.3 Metodika 82

7.4 Struktura softwarového projektu 82

7.5 Formalizace řízení softwarového projektu 87

8 Tvorba metodologických pomůcek na základě modelu **97**

9 Pomůcka pro optimalizaci struktury projektu **99**

9.1 Úvod 99

9.2 Vymezení a konkretizace vazeb 99

9.3 Volba vstupů a činitelů 101

9.4 Příklady vstupů 102

9.5 Příklady činitelů 104

9.6 Postup použití nástroje 110

9.7 Implementace nástroje 111

9.8 Kompletní příklad 112

9.9 Poznámky k interpretaci hodnot ... 115
9.10 Zhodnocení ... 116

10 Znalostní metodologická pomůcka ... 117

10.1 Úvod ... 117
10.2 Datový model ... 117
10.3 Příklady znalostí ... 119
10.4 Implementace znalostními mapami ... 124
10.5 Implementace znalostní databází ... 127
10.6 Zhodnocení ... 128

11 Závěr ... 129

11.1 Shrnutí a diskuse ... 129
11.2 Rekapitulace splnění cíle práce ... 130
11.3 Další možnosti rozvoje tématu ... 131

12 Shrnutí dosud dosažených výsledků ve výzkumu ... 133

Příloha 1: Charakteristiky jakosti dle normy ISO 9126 ... 134

12.1 Funkčnost (Functionality) ... 134
12.2 Bezporuchovost (Reliability) ... 135
12.3 Použitelnost (Usability) ... 136
12.4 Účinnost (Efficiency) ... 138
12.5 Udržovatelnost (Maintainability) ... 139
12.6 Přenositelnost (Portability) ... 140

Literatura ... 142

Vlastní publikace autora a publikace s podílem ... 142
Ostatní literatura ... 144

Seznam vyobrazení a tabulek

Vyobrazení

Obr. 1 – Studie Forrester o nasazení agilních metodik 9

Obr. 2 – Model „napiš a oprav“ 19

Obr. 3 – Model posloupnosti fází 20

Obr. 4 – Vodopádový model 21

Obr. 5 – Spirálový model životního cyklu 22

Obr. 6 – Adaptivní životní cyklus 23

Obr. 7 – Struktura UP 25

Obr. 8 –Pracovní nasazení v UP 26

Obr. 9 – Proces vývoje v metodice SCRUM 38

Obr. 10 – Proces vývoje ve FDD 46

Obr. 11 – Dynamické vytváření týmu v FDD: týmy jsou tvořeny podle vlastníků tříd 48

Obr. 12 – Trojrozměrná matice metodik Crystal 51

Obr. 13 – Rozdělení metodik do barevných skupin podle rozsahu projektu 52

Obr. 14 – Proces DSDM 60

Obr. 15 – Příklad grafu softwarového projektu na úrovni tříd s vazbami toku dat 85

Obr. 16 – Příklad grafu softwarového projektu na úrovni tříd s vazbami typu závislost 86

Obr. 17 – Příklad změnové funkce a reziduální adaptace 90

Obr. 18 – Průběh adaptace systému z příkladu 94

Obr. 19 – Příklady systémů s různou schopností agilnosti 96

Obr. 20 – Postup od obecného formálního modelu projektu k praktickým metodologickým pomůckám 97

Obr. 21 – Přehled příkladů vybraných činností pro naplnění pomůcky 107

Obr. 22 – Přehled příkladů artefaktů pro naplnění pomůcky 108

Obr. 23 – Zaznamenání nároků a přehled diferencí 113

Obr. 24 – Substituce 114

Obr. 25 – Výsledné diference 115

Obr. 26 – Schematický E-R datový model znalosti řízení softwarového projektu 118

Obr. 27 – Znalostní mapa znázorňující vazby znalostí na role v systému 125

Obr. 28 – Znalostní mapa znázorňující příslušnost znalosti k činitelům systému 126

Obr. 29 – Návrh aplikace pro práci se znalostní metodologickou pomůckou 127

Tabulky

Tab. 1 – Přehled vývoje softwarového inženýrství..... 18

Tab. 2 – Příklady měr procesu vývoje pro metodiku Scrum..... 64

Tab. 3 – Složky systémového modelu softwarového projektu..... 84

Tab. 4 – Průběh adaptace systému z příkladu a statistické ukazatele celkové reziduální adaptace (CRA)..... 93

Tab. 5 – Statistiky celkové reziduální adaptace (CRA) pro systémy z Obr. 19..... 95

Tab. 6 – Přehled pojmů použitých v pomůcce pro optimalizaci struktury projektu 99

Tab. 7 – Rozsahy kvantifikací..... 116

1 Úvod a motivace

Neustálé zkracování vývojových, realizačních i produkčních cyklů je důsledkem prorůstání softwarových aplikací do všech složek lidských aktivit. V oblasti tvorby softwaru jsou zřetelné usilovné snahy po dokonalejším, spolehlivějším, efektivněji vytvořeném programovém vybavení a jeho zajištění v nebývalých rozměrech, vytvářeným s přiměřenými náklady. Existuje zde nepochybně paralela s výrobními odvětvími a nabízí se tu srovnání s průmyslovou revolucí z přelomu 18. a 19. století. Ta měla za důsledek přesun objemu výroby od drobných řemeslníků k prvním velkovýrobcům, což bylo doprovázeno novou technologií i organizací práce.

Obdobně u vývoje softwaru je třeba s novými technologiemi a požadavky na produkty revidovat, upravit a optimalizovat stávající způsob řízení, jednotlivé postupy a jejich návaznosti. V této oblasti bylo již v posledních letech mnoho učiněno, nicméně stále zde zůstává značný prostor pro další potřebný výzkum.

Softwarové inženýrství se potýká s řadou problémů [Standish 1995]:

- **Vysoké náklady na vývoj** – software je velmi drahý, protože jeho vývoj je náročný a plný rizik;
- **Zpoždění harmonogramu** – tj. neschopnost dodat funkční systém do dohodnutého termínu;
- **Nedodání systému** – různá rizika projekt zcela ochromí a zabrání jeho dokončení;
- **Nevyhovující systém** – projekt je sice dokončen, ale systém nelze dlouhodobě úspěšně používat. V lepším (bohužel však méně častém) případě lze systém používat po dodatečných (a často nákladných a zdlouhavých) úpravách. Často je však systém prakticky zcela nepoužitelný.

Procento případů, kdy je dodaný systém (byť po mírných úpravách) úspěšně používán se stále pohybuje okolo 25 % ([Standish 1995] i novější statistiky firmy Gartner), což je v porovnání s ostatními inženýrskými oblastmi neuspokojující.

Průzkumy též ukazují, že hlavní příčina selhání projektů spočívá ve způsobu jejich řízení. První moment je dán tím, že projekty jsou dnes řízeny metodikami, které vznikaly v průběhu šedesátých až osmdesátých let 20. století. Sortiment řešených projektů se však od té doby značně rozšířil. Softwarové inženýrství se dlouhá léta zabývalo jen problémy s algoritmickým charakterem – typicky z oblasti úloh hromadného zpracování dat a vědeckých výpočtů. Zákazníkem byly typicky armáda a státní instituce, kde ke změnám docházelo jen velmi pozvolna. S rozvojem hardwaru a technologií (především levných osobních počítačů) na konci minulého století se začala výpočetní technika prosazovat i ve firemním a soukromém sektoru. Tento sektor však přináší nový charakter projektů, které se vyznačují těmito typickými charakteristikami:

- Projekty jsou složité, zpracovávaná data se vyznačují spoustou vnitřních vazeb, pravidel a omezení, závislostí, výjimek, atd., které se navíc mohou v čase měnit;
- Zákazník často není schopen (či z nějakého důvodu nemůže) na počátku přesně specifikovat, co bude od systému přesně požadovat;
- Zákazník z firemního sektoru (oproti vědeckým pracovníkům a pracovníkům výpočetních center) často není technicky či vědecky orientován;

- Projekt musí být typicky dodán rychle (má-li být konkurenční výhodou);
- Softwarový systém musí být snadno udržovatelný a modifikovatelný. Říká se, že dnes služby softwarové firmy dokončením vývoje systému nekončí, ale v podstatě začínají.

Tradiční osvědčené postupy softwarového inženýrství založené na důkladném plánování a návazností kroků se tak v této oblasti potýkají s řadou problémů. DeGrace v [DeGrace 1990] shrnuje důvody, proč tradiční vodopádový přístup k vývoji softwaru dnes nefunguje:

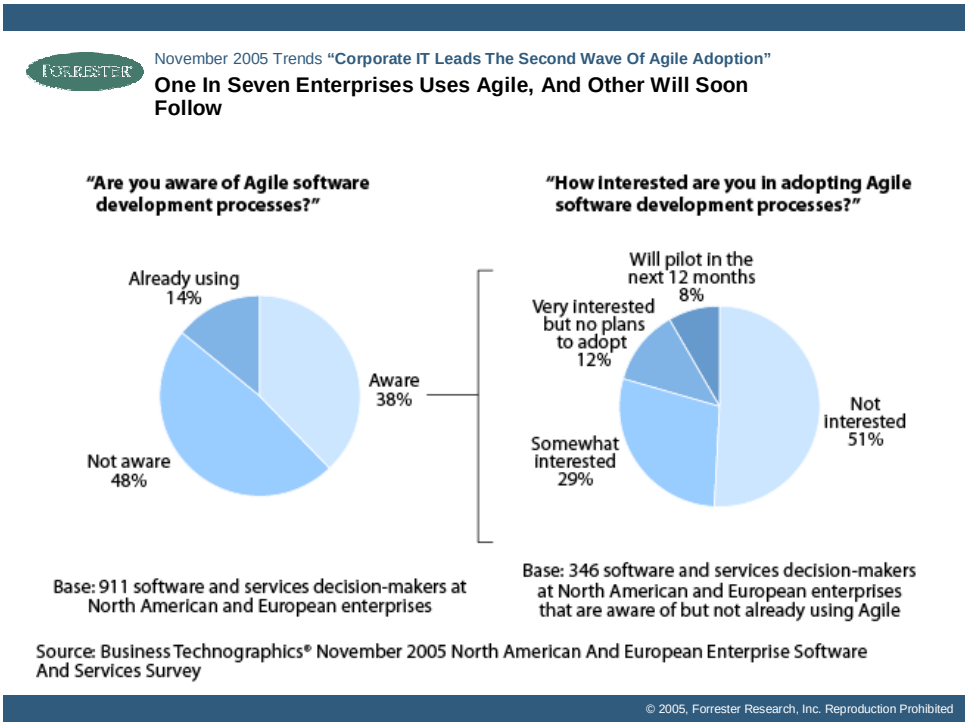
- Požadavky není možné plně pochopit před zahájením projektu;
- Uživatelé vědí, co chtějí teprve poté, co mají možnost vidět prvotní verzi aplikace;
- Požadavky se mění během vývoje;
- Nové nástroje a techniky činí implementační strategie obtížně předvídatelnými;

Wegner navíc ve [Wegner 1997] ukázal, že není možné plně specifikovat či testovat interaktivní systém navržený pro reakci na vnější vstupy (Wegnerovo Lemma).

Druhým aspektem jsou **možnosti současných vývojových prostředí**, s jejichž potenciálem tradiční metodiky nepočítají. Tradiční metodiky reflektují sekvenční schéma práce vývojových nástrojů 60.-80. let, kdy programování znamenalo dlouhé hodiny psaní kódu bez možnosti odzkoušení. Poté byl kód přeložen do strojového jazyka počítače a započala fáze testování. Pokud byla nalezena chyba, programátoři se vrátili ke psaní zdrojového textu, poté byl kód znovu přeložen, atd. Po úspěšném odladění jednotlivých částí systému byly tyto následně integrovány, celý systém testován, v případě nalezení chyby následoval znovu návrat k psaní zdrojového textu, atd.

Oproti tomu moderní překladače, vývojová prostředí a síťové technologie umožňují kooperativní a evoluční přístup k práci a k dispozici jsou též pokročilé nástroje pro automatizované provádění testů, refaktorizaci (zlepšení vlastností kódu bez změny funkčnosti), správu verzí, společného vlastnictví kódu, atd. Ve snaze využít tento potenciál se týmy často odkloňují od používání metodik a vývoj se potom sice stává z hlediska tvorby kódu efektivnějším, nicméně objevují se negativní jevy jako např. živelnost, nekoordinovanost procesu a postupná ztráta přehledu o funkčnosti a architektuře systému. Systém je poté sice naprogramován rychle, často však ani zdaleka nesplňuje očekávání zákazníka, není dlouhodobě udržovatelný a jeho kvalita je v nejlepším případě diskutabilní a neprokazatelná.

Z těchto důvodu jsem se rozhodl zabývat se řízením softwarových projektů využívajících moderní paradigmata. Ta jsou dnes nejvíce reprezentována myšlenkou agilního vývoje a agilních metodik, jež se začala formovat ve druhé polovině 90. let minulého století a v současnosti nabývá stále více na popularitě. Zaměřuje se na výše zmíněné problémy. Agilní metodiky nabízí řadu inovativních myšlenek a přístupů, jež oprošťují vývoj softwaru od zbytečné režie a umožňují více rozvinout potenciál týmu i nástrojů. Situace však není zcela jednoznačná. Z odborných kruhů zaznívá i kritika a pochybnosti o některých otázkách souvisejících s měřitelností, kvantifikovatelností a dokumentovatelností procesu vývoje vedeného agilně. To dokumentuje i jedna z posledních studií Forrester, která ukazuje, že odborná veřejnost o agilních metodikách ví, nicméně k úspěšnému masovému nasazení nejsou dosud splněny všechny předpoklady (Obr. 1).



Obr. 1 – Studie Forrester o nasazení agilních metodik

V současnosti tedy má softwarový inženýr na výběr mezi klasickými osvědčenými metodikami, které však nemusí poskytovat dostatek potřebné pružnosti a novými agilními metodikami, které však také mají svá úskalí. Nabízí se řešení ve formě kombinace těchto přístupů. Výsledky takovýchto snad v současnosti existují, nicméně jedná se pouze o dílčí řešení.

Z popsané situace současného softwarového inženýrství je vidět, že chybí určitý zastřešující pohled, což má za následek partyzánské snahy o zlepšení přístupu k tvorbě softwaru. Softwarové inženýrství dospělo do stavu, kdy nabízí velkou zásobárnu přístupů, postupů a řešení jednotlivých problémů, chybí však určitý objektivní holistický pohled, který by metodicky zastřešil softwarové inženýrství a poskytl formalizovaný výchozí bod pro další snahy.

2 Cíl

Tato disertační práce reaguje na situaci popsanou v úvodu. Jako jednu z hlavních příčin problémů softwarového inženýrství spatřuji neexistenci zastřešujícího pohledu, jež by umožňoval nahlížet na rozsáhlou praxi softwarového inženýrství integrujícím pohledem a který by se mohl stát nástrojem pro další rozvoj poznání v této oblasti. Práce je zaměřena především na metodiky reflektující moderní programovací paradigmata, nicméně neopomíjí ani klasické osvědčené metodiky.

Středem zájmu softwarového inženýrství je softwarový projekt. **Cílem práce je navrhnout vhodný formální model softwarového projektu, jež by umožnil vytvořit zastřešující pohled na řízení softwarového projektu. Model by měl sloužit pro rozvoj poznání v oboru metodologie softwarového inženýrství a podporovat řešení praktických problémů.**

Vytvořený formální model projektu by měl splňovat především tyto požadavky:

- Být založen na ověřených vědeckých poznatcích a formalismech;
- Být dostatečně obecný a umožňit konzistentní začlenění co největší části znalostí disciplíny softwarového inženýrství;
- Nebýt v rozporu s ověřenými postupy a principy softwarového inženýrství;
- Umožňit dále rozvíjet formalizaci a prohlubovat poznání zákonitostí softwarového inženýrství a vnitřních souvislostí;
- Podporovat tvorbu praktických pomůcek a aplikací.

Dalším cílem je návrh postupu využití formálního modelu pro podporu práce softwarového inženýra a demonstrování postupu na konkrétních praktických příkladech.

3 Metodika

Disertační práce navazuje na rešerši, kde jsem provedl analýzu současného stavu softwarového inženýrství se zaměřením na metodiky řízení softwarových projektů využívajících moderní programovací paradigmata. Postup řešení byl následující:

1. Specifikoval jsem terminologii, se kterou budu v práci pracovat (kap. 4);
2. Provedl jsem rešerši (kap. 5) s následujícím obsahem:
    a. Analýza historických souvislostí softwarového inženýrství;
    b. Tvrdý metodologický přístup: současná praxe rigorózních metodik;
    c. Měkký metodologický přístup: rodina agilních metodik;
    d. Formalizace v disciplíně softwarového inženýrství.
3. Na základě rešerše jsem sestavil obecný přehled procesu vývoje softwaru (kap. 6);
4. Navrhl jsem formalizaci struktury a softwarového projektu a formalizaci řízení softwarového projektu (kap. 3.v);
5. Navrhl jsem postup tvorby metodologických pomůcek na základě rozpracování obecného modelu (kap. 8);
6. Navržený postup jsem ilustroval na 2 konkrétních příkladech (kap. 9 a 10);
7. Vyhodnotil jsem splnění cílů práce.

Pro dosažení vytyčeného cíle jsem byl nucen provést rozsáhlou rešerši mapující disciplínu softwarového inženýrství především s ohledem na požadavek co největší obecnosti formalizace a též pro vytvoření kvalifikovaných příkladů zakládajících se na skutečné praxi. Současný stav disciplíny vychází z historických souvislostí, rešerše tedy začíná stručným přehledem historického vývoje disciplíny a hlavních paradigmat. Jak již bylo zmíněno v úvodu, současná praxe je ve znamení dvou paradigmat: klasického tvrdého (rigorózního) a moderního měkkého (agilního). Tyto přístupy mají své zastánce a odpůrce a je cítit i jistá rivalita těchto dvou táborů. Návrh řešení jsem koncipoval především s ohledem na moderní paradigmata, nicméně pro ucelený pohled jsem byl nucen se zaměřit na oba tyto přístupy.

V rešeršní části práce, kde představuji nejdůležitější metodiky popisuji pro každou metodiku:

- Vznik a vývoj;
- Základní charakteristiku;
- Klíčové pojmy;
- Proces vývoje pomocí metodiky.

Vazbu na praxi jsem zajistil formou posouzení výsledků a závěrů odborníky z oboru. Pro tyto účely jsem při řešení práce navázal spolupráci se dvěma firmami:

- Firmou e-Fractal, s.r.o., jež je českou softwarovou firmou úspěšně fungující řadu let na našem trhu. Firma se zabývá především vývojem softwaru na zakázku s použitím moderních agilních metodik;
- Společností Deloitte Advisory, s.r.o, jež je řazena mezi přední mezinárodní konzultační firmy. Společnost Deloitte je v oblasti IT vyhledávaná jako poradce při optimalizaci informačních systémů a informační infrastruktury, analýze a vývoji nových informačních systémů pro soukromý sektor i státní správu.

4 Terminologie

Pojem **softwarové inženýrství (SI)** začal být používán na přelomu 50. a 60. let. Oficiálně poprvé zazněl v r. 1968 na konferenci NATO o softwarovém inženýrství.

Tento pojem bývá používán ve více významech:

- Jako zastřešující pojem zahrnující široké spektrum aktivit, původně nazývaných programování a systémová analýza;
- Jako široký pojem pro všechny aspekty *praxe* počítačového programování, jakožto protiklad *teorie* nazývané teoretická informatika (computer science);
- Jako pojem ospravedlňující určitý specifický přístup k programování, který zdůrazňuje, že programování je inženýrská profese spíše než určité řemeslo či umění.

Pro potřeby této práce budu softwarové inženýrství chápat v souladu se standardem IEEE 610.12:

Softwarové inženýrství je

1. „Použití systematického, disciplinovaného, kvantifikovaného přístupu k vývoji, provozu a správě softwaru, tj. aplikací inženýrství do softwaru“;
2. Studium přístupů jako v 1.

Důležitými pojmy je trojice metodologie-metodika-metoda. V této práci budu tyto tři pojmy chápat v souladu s [Kadlec 2004], tedy

- **Metodologie** – nauka o metodách a metodikách;
- **Metodika** – soubor vzájemně provázaných metod, jež poskytuje komplexní postup a návod (pro vývoj softwarové aplikace). Metodika zahrnuje více etap řešení (fází vývojového cyklu);
- **Metoda –** označení pro konkrétní postup vedoucí k vyřešení dílčího problému. Nezahrnuje více etap vývoje.

Pojem **metodika** v práci používám ve dvou variantách:

- Pojem **rigorózní metodika, klasická metodika** či **tradiční metodika** označuje tvrdé metodiky s důrazem na formální stránku řízení. Tyto metodiky kladou důraz na přesnou specifikaci činností, jejich obsahu a návazností;
- Pojem **lehká metodika** budu v souladu s [Buchalcevová 2005] používat pro měkké agilní přístupy a metodiky, které oproti rigorózním nespecifikují detailně prováděně činnosti, ale pouze definují myšlenky a zásady softwarového procesu s různou mírou detailu.

Pokud nebude řečeno jinak, samotný pojem *metodika* zahrnuje obě tyto varianty.

Následující termíny budu chápat v souladu s normou ČSN ISO 9000 (citace [ISO 9000]):

- **Proces** – soubor vzájemně souvisejících nebo vzájemně působících činností**,** který přeměňuje vstupy na výstupy;
- **Produkt** – výsledek procesu;
- **Postup** – specifikovaný způsob provádění činnosti nebo procesu;
- **Požadavek** – potřeba nebo očekávání, které jsou stanoveny, obecně se předpokládají nebo jsou závazné;
- **Specifikovaný požadavek** je požadavek, který je stanoven například v dokumentu;

  V SI hovoříme o požadavcích především jako o *požadavcích zákazníka*. Tento termín má v klasických metodikách blíže k pojmu „specifikovaný požadavek“ a v agilních přístupech je někde mezi pojmem „požadavek“ a „specifikovaný požadavek“, jelikož ne vždy je ve fázi analýzy zanesen.
- **Projekt –** jedinečný proces sestávající z řady koordinovaných a řízených činností s daty zahájení a ukončení, prováděný k dosažený cíle, který vyhovuje specifickým požadavkům, včetně omezení daných časem, náklady a zdroji;
- **Spokojenost zákazníka** – vnímání zákazníka týkající se stupně splnění jeho požadavků;
- **Jakost** – stupeň splnění požadavků souborem inherentních charakteristik. Jakost je tedy chápána výhradně ve vztahu s mírou splnění požadavků zákazníka;

Dále budu pracovat s termínem **znalost**. Pro tento mladý termín neexistuje zatím jedna všeobecně uznávaná definice. Pro potřeby této práce budu pojem znalost užívat ve významu „schopnost (člověka či inteligentního stroje) použít informace k vyřešení problému: použít informace na správném místě, ve správném čase a správným způsobem tak, aby byl problém správně vyřešen“ [Havlíček 2006].

5 Literární rešerše

5.1 Vývoj softwarového inženýrství

Vývoj disciplíny softwarového inženýrství si lze (hrubě) rozčlenit do několika základních etap (Tab. 1), které naznačují, jak se formovala potřeba systematických postupů [Kadlec 2004].

Etapa		
	Charakteristiky	
do poloviny 60. let 20. stol.: pionýrské doby softwarových projektů		
	Charakteristika HW	sálové počítače, velmi drahý, omezený výpočetní výkon, specializovaný na hromadné zpracování dat.
	Typ uživatelů	armáda, vědecká pracoviště, státní instituce
	Typ softwaru	specializovaný software s dávkovým zpracováním, úlohy numerické a datového zpracování
	Životní cyklus softwaru	software vytvářen jednoúčelově bez ohledu na budoucí potřeby, programy většinou neudržovatelné a neměnné
	Požadavky na programátora	detailní znalost konkrétního hardwaru, provádění vlastní analýzy
	Požadavky na analytika	specializovaná funkce analytika neexistuje
	Řízení projektů	neexistuje disciplína řízení softwarových projektů
	Rozvíjející se metody	žádné
přelom 60. a 70. let: první náznaky snah, konference NATO o softwarovém inženýrství 1968		
70. léta		
	Charakteristika HW	HW se stává dostupnější
	Typ uživatelů	uživatelská obec se rozšiřuje i na instituce a úřady
	Typ softwaru	první aplikace umožňující interakci se svými uživateli, počátek „produktového“ softwaru
	Životní cyklus softwaru	software vytvářen stále většinou jednoúčelově bez ohledu na budoucí potřeby
	Požadavky na programátora	znalost hardwaru, provádění vlastní analýzy, programátoři se postupně seznamují se vznikajícími technikami, masově však využívány nejsou

	Požadavky na analytika	specializovaná funkce analytika neexistuje
	Řízení projektů	vývoj není řízen, v druhé polovině 70. let se začínají využívat dnes známé techniky SWI, např. specifikace, návrh, architektura, testování, zajištění kvality, modely životního cyklu, atd.
	Rozvíjející se metody	strukturované metodiky a dílčí metody řešící jednotlivé fáze vývojového cyklu (Yourdonova, Hatleyova)
80. léta		
	Charakteristika HW	vznik osobních počítačů, miniaturizace výkonných počítačů
	Typ uživatelů	uživatelská obec se dále rozšiřuje do průmyslu i služeb, s výpočetní technikou se setkávají i uživatelé z „nevýpočetních“ oborů
	Typ softwaru	rozvoj produktového SW, vznik a zdokonalování komponentových technologií, rozvoj softwaru pro nejrůznější účely
	Životní cyklus softwaru	snaha o interoperabilitu, ústup od jednorázových řešení
	Požadavky na programátora	programároři se diferencují na systémové a aplikační, u systémových vzrůstá důraz na abstrakci a znalost knihoven spíše než hardwaru
	Požadavky na analytika	začíná se prosazovat funkce analytika, často jde však současně i o programátora
	Řízení projektů	masivní nástup softwarově-inženýrských metodik, rozmach a popularizace objektově orientované analýzy a návrhu, později nástup standardizace
	Rozvíjející se metody	dominují strukturované metody, začínají se prosazovat objektově-orientované přístupy k analýze a návrhu
90. léta		
	Charakteristika HW	dramatický nárůst výkonnosti HW, trvalý značný pokles ceny, výpočetní technika i příslušenství se stávají běžnou záležitostí dostupnou pro všechny firmy i jednotlivce
	Typ uživatelů	instituce, školy, firmy i jednotlivci

	Typ softwaru	široká nabídka „krabicového“ SW pokrývající značnou šíři oblastí činností, např. zpracování textu, databáze, grafika, DTP, hry, aj. současně rozvoj softwaru na zakázku, značně roste komplexnost SW a rozsáhlost projektů softwarové systémy se začínají prosazovat i v zařízeních denní potřeby (mobilní telefony, domácí spotřebiče)
	Životní cyklus softwaru	snaha o rozšiřování softwaru, vznik nových verzí, snaha o uspokojování potřeb uživatelů, uživatelská podpora se stává součástí produktu
	Požadavky na programátora	orientace v dynamicky se rozvíjejících technologiích, práce s komplexními vývojovými nástroji, znalost rozsáhlých knihoven
	Požadavky na analytika	vzrůstá úloha analytika, oddělení funkce analytika a programátora
	Řízení projektů	řízení projektů se stává klíčovým pojmem, projekty řízeny hierarchicky
	Rozvíjející se metody	velký rozmach komplexních metodik, především objektově-orientovaných
1997 softwarové inženýrství uznáno jako obor s certifikátem v USA		
od r. 2000		
	Charakteristika HW	velmi levný a výkonný hardware a periferie, „hardwarem“ se postupně stává většina elektroniky a elektrických spotřebičů, rozšiřování různých infostánků
	Typ uživatelů	v různé formě téměř každý občan vyspělých zemí
	Typ softwaru	software má mnoho podob: • podnikový SW, • uživatelský SW, • SW stojící za webovými aplikacemi, • SW mobilních zařízení a elektroniky, • aj. software začíná být organicky chápán v souvislostech s klíčovými pojmy informace a znalosti, je kladen důraz na „inteligenci“ softwaru a uživatelského rozhraní

	Životní cyklus softwaru	díky sofistikovaným nástrojům je SW vyvíjen velmi rychle, nicméně stoupá jeho komplexnost a požadavky na bezpečnost, jakost, dostupnost, atd., klíčovou roli hraje podpora a služby
	Požadavky na programátora	programátoři se stále méně zabývají technickými detaily a stávají se výkonnými „dělníky“ generujícími za pomoci obsáhlých nástrojů typizovaný kód
	Požadavky na analytika	role analytika je klíčová, vzrůstají nároky na schopnost práce s informacemi a abstraktní myšlení
	Řízení projektů	projekty jsou vytvářeny v dynamicky se měnících podmínkách, řízení projektů je věnována značná pozornost teoretická i praktická, objevují se alternativní způsoby řízení
	Rozvíjející se metody	kromě klasických metodik se začínají prosazovat odlehčené metodiky umožňující pracovat efektivněji v dynamicky se měnícím prostředí

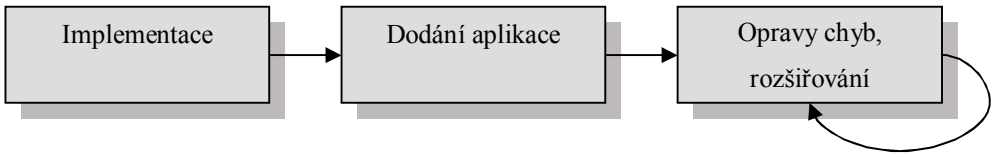
Tab. 1 – Přehled vývoje softwarového inženýrství

V průběhu vývoje informačního inženýrství vzniklo několik typů životního cyklu projektu ([Wilkie 1993]). Odráží vždy potřeby období, ve kterém vznikly, a s postupným rozvojem nároků byly vždy nahrazovány dalšími. Nicméně i v současnosti lze pro všechny životní cykly nalézt vhodné uplatnění.

5.1.1 Model napiš a oprav

Tento model se používal v začátcích vývoje moderního programování (50. léta) a spočíval v sepsání aplikace, v jejím předání do provozu (spuštění) a následném opravování chyb (Obr. 2). Hlavním rysem tohoto modelu je absence explicitních prvků analýzy a řízení. Z tohoto důvodu použití tohoto modelu začalo brzy narážet na problémy.[1]

V současné době lze tento model využít při tvorbě drobných prográmků a utilit, které jsou zvládnutelné jedním programátorem v řádu hodin až dní a použití sofistikovanějšího modelu je zbytečné.



[1] V této době se ovšem nehovořilo o metodách a řízení softwarových projektů, tyto pojmy se začaly používat až koncem 60.let. Tento model tedy vznikl zcela spontánně a byl katalogizován až dodatečně.

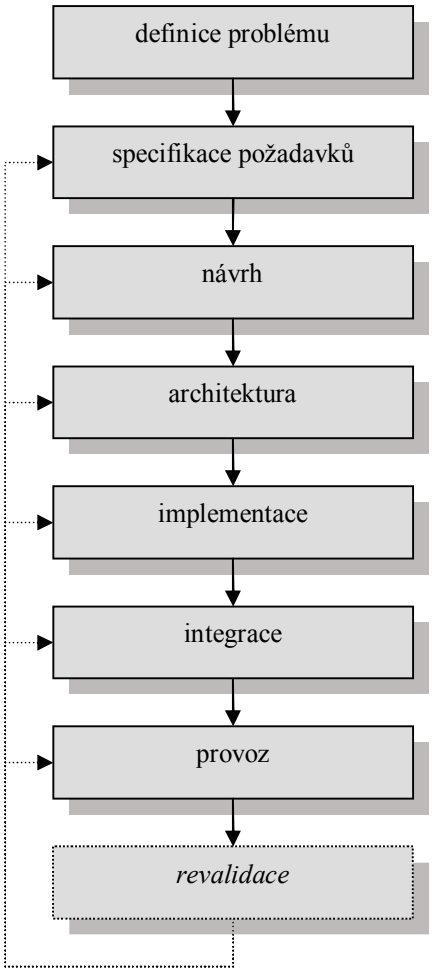
Obr. 2 – Model „napiš a oprav“

5.1.2 Striktní posloupnost fází

V roce 1957 byl vytvořen první strukturovaný model, který se skládal z přesné posloupností fází (Obr. 3):

- Definice problému;
- Specifikace požadavků;
- Návrh;
- Architektura;
- Implementace;
- Integrace;
- Provoz.

Fáze definované v této metodě se dodnes v určité podobě praktikují. V této rané fázi však chyběla jakákoliv zpětná vazba: po dokončení jedné fáze se pokračoval dál bez ověření výsledku až do konce. Po dodání aplikace mohla ještě následovat fáze revalidace, ve které bylo možné rozhodnout o navrácení zpět a opětovném spuštění procesu. Tím se do vývoje softwaru vnesl též prvek řízení.



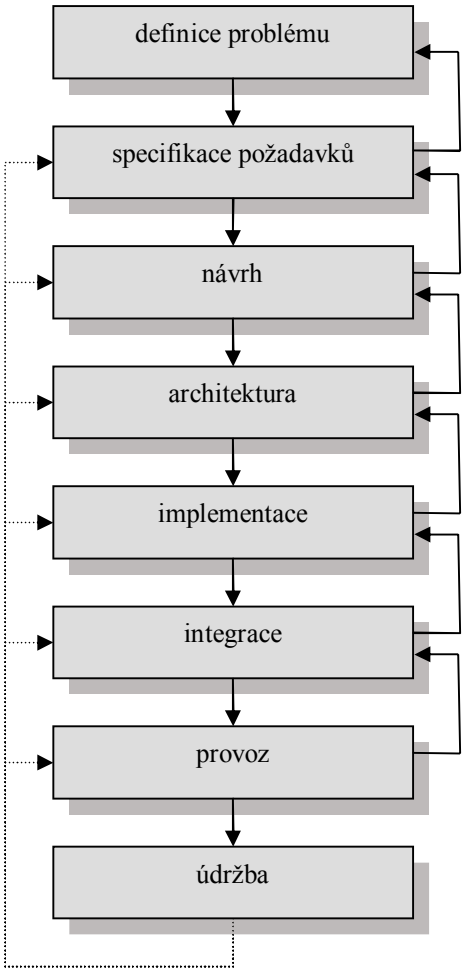
Obr. 3 – Model posloupnosti fází

5.1.3 Vodopádový model

Prvním modelem vytvořeným v rámci nově vznikající disciplíny softwarového inženýrství v 70. letech byl vodopádový model z roku 1970. Jeho autorem je Dr. Winston Royce [Royce 1970].

Z hlediska fází tento model vychází z předchozího a jeho přínosem je především zavedení zpětné vazby: po každé fázi dojde k jejímu vyhodnocení a případným opravám. Tento model tím přinesl značný pokrok, jelikož bylo možné kontrolovat dosažený pokrok a vytvořil tak půdu pro rozvoj sofwarových.

Při nárůstu rozsahu IS se však postupně začaly projevovat nevýhody tohoto modelu – nepružnost a dlouhé dodací lhůty.



Obr. 4 – Vodopádový model

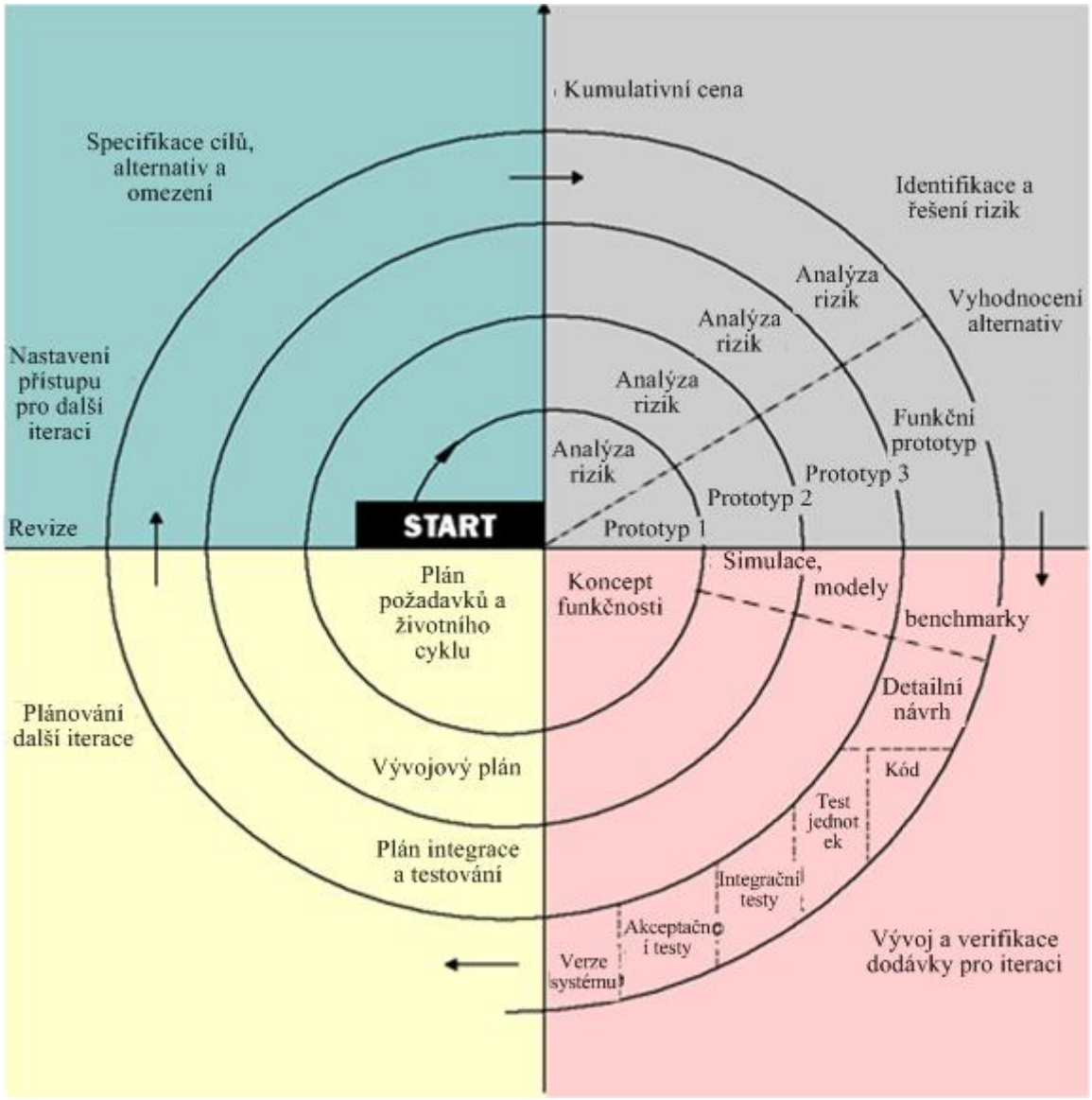
5.1.4 Spirálový model

Spirálový model (Obr. 5) byl navržen Barry Boehmem v roce 1985. Reaguje na některé nedostatky vodopádového modelu a závádí dva přelomové koncepty:

- Iterativní přístup;
- Opakovanou, důslednou analýzu rizik.

Uspořádání tohoto modelu lze prohlásit za důležitý mezník a dnes je na něm založena řada metodik (např. Rational Unified Process, kap. 5.2 a především agilní metodiky, kap. 5.3). Model vychází ze situace, kdy je v úvodu projektu velmi obtížné dopodrobna specifikovat všechny požadavky a vyjmenovat všechny funkce. V úvodu se stanoví pouze obecný, vnější rámec. Následně se vyvine část aplikace, provede se konzultace se zákazníkem a podle konkrétní situace se pokračuje analogicky dalším krokem (iterací).

Postup v jednotlivých fázích je řízen riziky, tj. je vždy provedena důsledná analýza rizika možných problémů. V závislosti na výsledcích rizikových analýz se rozhoduje o dalším směru vývoje, konkrétním postupu a úpravách postupu. Tento přístup umožňuje dobré přizpůsobení situacím, do nichž se vývoj dostává.



(upraveno z http://www.elanman.org)

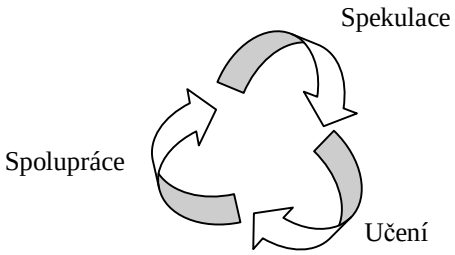
Obr. 5 – Spirálový model životního cyklu

Speciální variantou spirálového modelu je **inkrementální model,** který v jednotlivých otočkách spirály dodává rovnou hotové použitelné části systému. Inkrementální model má například Extrémní programování (kap. 5.3.4).

5.1.5 Adaptivní model

Adaptivní model je založen na teorii komplexních adaptivních systémů [Holland 1995]. Vychází z přesvědčení, že moderní komplexní projekty není možné řídit na základě vnuceného řádu, ale nejlepších výsledků je dosahováno, když tradiční model příkazy-kontrola je nahrazen modelem vedení-spolupráce. Hodnota v projektu je vytvářena na základě principu *emergence*, kdy spolupráce mnoha jedinců na společném cíli vede k vyšší hodnotě, než je součet přispění jednotlivých jedinců.

Životní cyklus sestává z několika fází iterativního cyklu znázorněného na Obr. 6.



Obr. 6 – Adaptivní životní cyklus

Fáze spekulace nahrazuje fázi plánování. I když obsah této fáze je v podstatě shodný, název je zvolen úmyslně s ohledem na to, že v adaptivním modelu se je možné se od plánu odklonit. Místo návrhu a implementace se zde hovoří o spolupráci. Snaží se tím opět evokovat naladění této fáze, které by mělo probíhat v duchu vzájemné spolupráce, a to jak na úrovní týmu, tak se zákazníkem. Náplní třetí fáze, učení, je především hodnocení. Termín „učení“ je zvolen pro zdůraznění zpětné vazby, která z hodnocení vyplývá, tedy zlepšení vývojového procesu.

5.2 Tvrdý metodický přístup – Unified Process

5.2.1 Vznik a vývoj

Unified Process (UP) je rozsáhlá, propracovaná, objektově-orientovaná iterativní metodika vývoje softwaru, která je v současné době uznávaným standardem. Podkladem této kapitoly je publikace [Merunka et al. 2005], na níž se autor podílel.

Metodika návrhu objektových informačních systémů Unified Software Development Process (USDP), známá spíše pod zkráceným názvem Unified Process (UP), je jednou z mnoha objektově-orientovaných metodik založených na jazyce UML. Tato metodika pochází přímo od autorů jazyka UML (Booch, Jacobson, Rumbought) a je (spolu se svými deriváty – např. RUP) v současnosti nejpoužívanější metodikou. Metodika UP je založena na metodikách Ericson (Ericsonapproach), Rational (Rational Objectory Process), OMT a na dalších zdrojích vycházejících z nejlepších postupů. Kořeny metodiky sahají do roku 1967, kdy vznikl Ericssonův model vycházející z faktu, že se složité systémy mají modelovat jako množiny vzájemně propojených bloků, které byly dále rozšířeny přidáváním dynamických pohledů.

V roce 1987 zakládá Ivar Jacobson firmu Objectory AB. Ta vyvinula metodiku Objectory (více [Jacobson et al. 1992, která byla vyvinuta jako systém se svými náležitosti a návaznostmi – celý průběh metodiky (požadavky, analýza, návrh, implementace, test) byl zachycen pomocí diagramů. Tato metoda umožňovala (a také vyžadovala – podobně jako UP) své přizpůsobení konkrétnímu uživateli a projektu. Další vývoj metodiky probíhal v devadesátých letech v těsné návaznosti s vývojem jazyka UML.

V roce 1998 firma Rational vytváří metodiku Rational Unified Process (RUP, [Kruchten 2000]). Ta obsahuje další zlepšení v oblastech zachycování požadavků, správy konfigurace, testování atd. V roce 1998 Jacobson publikuje *knihu*

Unified Software Development Process (viz [Jacobson et al. 1999]), v níž je popsána metodika Unified Process (UP). Zatímco RUP je komerčním produktem firmy Rational, tak UP je otevřený standard od tvůrců jazyka UML.

Na základě RUP bylo Scottem Amblerem vytvořeno rozšíření Enterprise Unified Process (EUP), který životní proces rozšiřuje o dvě fáze: Production a Retirement a zavádí několik dalších disciplín [Ambler et al. 2005].

5.2.2 Základní charakteristika

Unified Process je rigorózní metodika, která je spjata s notací UML. Je velmi rozsáhlá a podrobná a je koncipována jako obecná metodika tvorby softwaru – obsahuje mnoho různých metod pro jednotlivé fáze životního cyklu projektu, pro jednu fázi může existovat i několik konkurenčních metod práce [Jacobson et al. 1999].

5.2.3 Klíčové pojmy

Metodika UP je řízena **požadavky**. Ve všech fázích tvorby softwaru posuzuje další postup na základě analýzy rizik. Metodika UP je založena na návrhu a postupném vývoji robustní architektury systému. Architektura popisuje nejen aspekty rozkladu systému na komponenty, ale také popisuje způsob jakým se tyto komponenty ovlivňují.

Metodika UP je **iterativní a přírůstková**. Iterativní aspekt znamená, že rozklad projektu na menší podprojekty – iterace. Znamená to, že projekt tvoříme postupným upřesňováním a rozšiřováním funkcí systému.

Metodika UP je založena na třech základních pilířích. Jsou to:

- Řízení případem užití a rizikem;
- Zásada soustředění na architekturu;
- Zásada iterace a přírůstku.

Iterace se v metodice UP skládá z těchto pěti pracovních postupů (částí):

- **Požadavky** – zachycují to, co by měl systém dělat;
- **Analýza** – vybroušení požadavků a jejich strukturování;
- **Návrh** – realizace požadavků v architektuře systému;
- **Implementace** – vlastní tvorba softwaru;
- **Testování** – ověření, zda jsme implementace funguje podle zadaných požadavků.

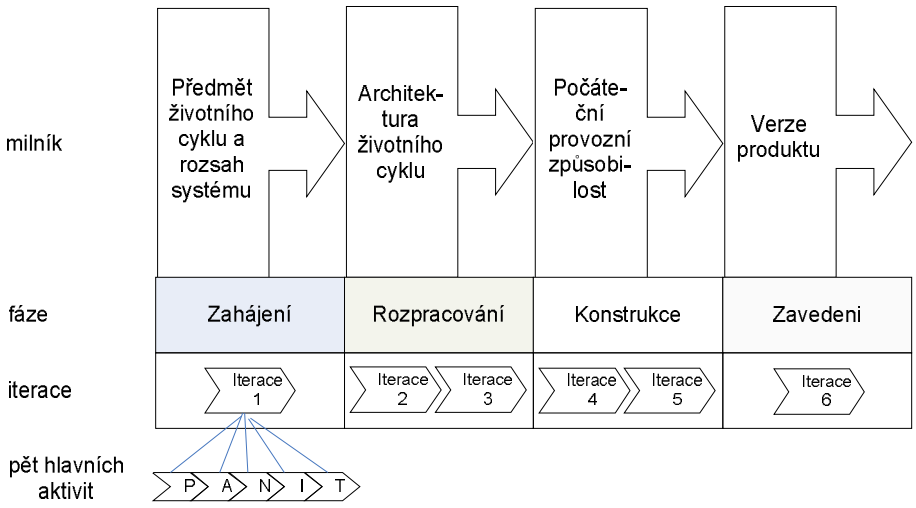
Každá iterace metodiky UP tvoří základní linii (baseline), což je soubor revidovaných a schválených výstupů dané iterace. Přírůstky (inkrementy) jsou rozdílem mezi jednou základní linií a jinou základní linií. Přírůstky jsou jednotlivými dílčími kroky směrem k finální verzi systému. Jednotlivé práce na jednotlivých iteracích mohou probíhat paralelně (pokud to závislosti mezi nimi dovolují). To umožňuje zrychlit proces návrhu systému.

5.2.4 Proces vývoje pomocí metodiky

Metodika UP se skládá ze čtyř po sobě následujících fází (viz Obr. 7). Každá tato fáze končí hlavním milníkem. Tyto fáze jsou:

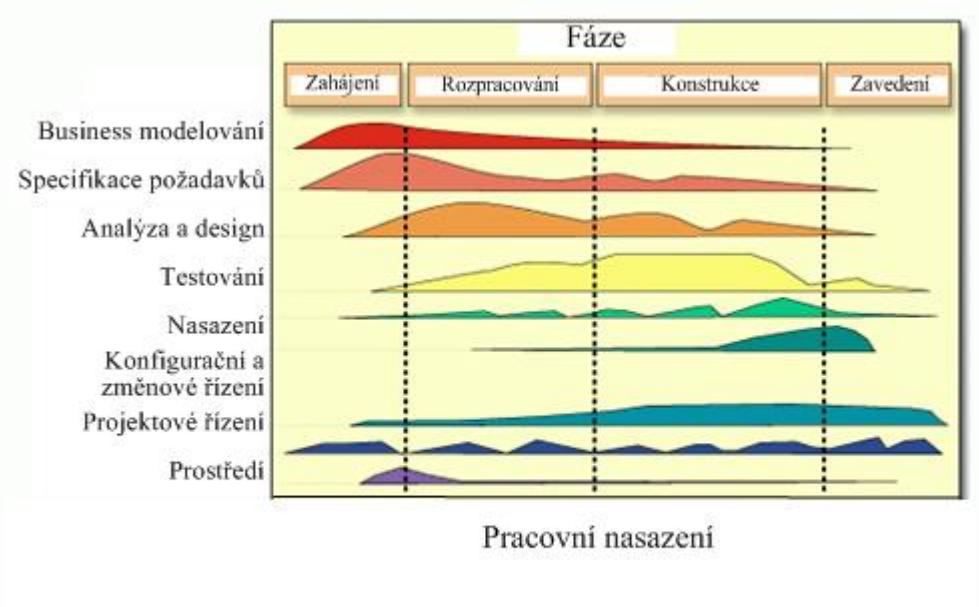
- **Začátek (inception)** – období plánování;
- **Rozpracování (elaboration)** – období vytváření (? návrhu) architektury;
- **Konstrukce (construction)** – počátky provozuschopnosti;
- **Zavedení (transition)** – nasazení produktu do uživatelského prostředí.

Každá fáze má určitý cíl, na nějž jsou soustředěny aktivity jednoho nebo více pracovních postupů a jenž je významným milníkem v životním cyklu projektu.



Obr. 7 – Struktura UP

Jednotlivé fáze kladou různé nároky jednotlivé části týmu (Obr. 8 –Pracovní nasazení v UP)



(upraveno z http://www.elewise.com)

Obr. 8 –Pracovní nasazení v UP

5.2.4.1 Fáze Začátek

Cílem fáze začátek je zahájení projektu. Tato fáze obsahuje:

- Tvorbu podmínek proveditelnosti – to může zahrnovat tvorbu prototypů;
- Prvotní návrh obchodního (podnikatelského) případu – na něm má být možno ukázat kvantitativní obchodní přínos;
- Zachycení podstatných požadavků – ty umožňují definoval rozsah systému;
- Označení kritických rizik.

Hlavními pracovníky jsou v této fázi manažer projektu a systémový projektant.

5.2.4.2 Fáze Rozpracování

Hlavními výstupy fáze rozpracování jsou:

- Tvorba spustitelného architektonického základu;
- Vylepšení odhadu rizik;
- Definice atributů jakosti;
- Zachycení případů užití pro 80% funkčních požadavků;
- Tvorba přesného plánu konstrukční fáze;

- Formulace nabídky, která obsahuje veškeré požadavky na čas, vybavení, personál a náklady.

Hlavním cílem je tvorba spustitelného architektonického základu. Je to skutečný, spustitelný systém, sestavený na základě specifikací architektury. Není to prototyp, který můžeme zahodit, ale je to první pokus o konečný systém.

5.2.4.3 Fáze Konstrukce

Cílem konstrukční fáze je splnit všechny požadavky analýzy a návrhu a vyvinout ze základu získaného z předchozí etapy konečnou verzi systému. Klíčovým zadáním konstrukční fáze je zachování integrity architektury vytvářeného systému.

5.2.4.4 Fáze Zavedení

Fáze zavedení začíná v okamžiku , kdy je dokončeno testování a konečné zavedení systému. To zahrnuje opravu všech chyb nalezených v beta verzi. Cíle této etapy jsou takovéto:

- Oprava chyb;
- Příprava uživatelského pracoviště na přijetí nového softwaru;
- Přizpůsobení softwaru v případě vzniku problémů;
- Tvorba manuálů a dokumentace;
- Konzultace s uživateli;
- Konečná revize.

5.2.4.5 Shrnutí

Metodika Unified Process je velmi rozsáhlá a propracovaná metodika. Z důvodu omezeného rozsahu textu jsem představil pouze základní strukturu, metodika zabíhá do detailů jednotlivých fází a obsahuje návody a postupy na provádění všech klíčových činností včetně specifikace milníků a checklistů. Unified Process jakožto zástupce rigorózních metodik též klade důraz na řízenost a měřitelnost celého procesu. I přesto, že je navržena jako obecná konfigurovatelná metodika, každá její instance je přesným návodem na postup prací (v porovnání s agilním přístupem popisovaným dále).

Síla metodiky spočívá v její značné propracovanosti. Metodika je vyzkoušena léty průmyslové praxe a je ve své komerční verzi Rational Unified Process firmy Rational de-facto průmyslovým standardem ve vývoji informačních systémů. Metodika vede řídící pracovníky i vývojáře celým vývojovým životním cyklem a zabraňuje sklouznutí k neorganizovanému, neměřitelnému a špatně řiditelnému vývoji.

V řadě případů se však projekt dostává do určitých problémů právě kvůli nepružnému procesu UP[2]. Jedná se především o situace, kdy na počátku vývoje nelze získat přesné zadání a provést ucelenou analýzu nebo se zadání z různých důvodů mění během vývoje. Těmto situacím nelze v dnešním dynamickém světě businessu vždy zabránit (a ani to nemusí být z ostatních hledisek žádoucí). V další kapitole představím, jakým způsobem se tento problém pokouší řešit agilní metodiky.

Dalšími zástupci rigorózních metodik jsou např. Coad-Yourdonova metoda, OOSD, OOSE, J. Martin a další [Wilkie 1993, Henderson-Sellers 1994, Yourdon 1994, Merunka et al. 2005]. Z důvodu omezení rozsahem textu těmto metodikám nebudu již věnovat další prostor, jelikož nepřináší oproti UP nic, čím by bylo třeba se zabývat pro splnění cílů práce.

5.3 Měkký metodický přístup – Agilní metodologie

Tato kapitola bude věnována tzv. agilnímu přístupu. Představím, co tento pojem znamená, jeho vznik a historii. Dále přistoupím k představení nejznámějších agilních metodik[3].

V rámci shrnutí budu u jednotlivých metodik diskutovat pouze relativní charakteristiky v rámci agilního přístupu. Společné aspekty týkající se celé rodiny agilních metodik shrnu na závěr.

5.3.1 Vznik a vývoj

Jak jsem uvedl na začátku práce, zhruba od poloviny 90. let jsme svědky pronikání výpočetní techniky a softwaru do všech oblastí lidské činnosti. Spolu s překotným vývojem společnosti a všech odvětví se mění i pohled na řízení softwarových projektů. Typické softwarové projekty z oblasti obchodu a služeb se dnes vyznačují těmito aspekty:

- Aby systém mohl být konkurenční výhodou, je třeba **rychle ho dodat;**
- V průběhu vývoje se mohou **změnit požadavky**;
- Uživatelé **nemají na počátku přesnou představu** o svých potřebách[4];
- Zákazníci žádají **jakostní řešení**.

---

[2] Zde mluvím o pružnosti procesu v průběhu vývoje – metodika UP je pružná v tom smyslu, že je možné vytvořit instanci metodiky na míru konkrétní oblasti, týmu a podmínkám, instance sama je však poté již nepružná.

[3] V této části budemu pojem **metodika** chápat ve významu **lehká metodika**. O klasických metodikách budu mluvit jako o „tradičních metodikách“ či „klasických metodikách“ (viz 4 Terminologie)

[4] Norma ISO 9000 definuje jakost jako „stupeň splnění požadavků“ [ISO 9000], přičemž z definice „požadavku“ vyplývá, že tento nemusí být nutně předem přesně specifikován v dokumentu (to platí pouze pro tzv. „specifikovaný požadavek“).

Zhruba od 2. poloviny 90. let se na poli softwarového inženýrství začínají objevovat snahy o přizpůsobení vedení projektů novým požadavkům doby. Za posledních 10 let vznikla celá řada úprav stávajících metodik či metodik zcela nových, které se snaží zaměřit na zmíněné problémy. Nejvýraznější snaha je označována jako **agilní přístup** (AP) nebo **rodina agilních metodik**.

Agilní přístup vznikl jako sjednocení snah skupiny významných osobností softwarového inženýrství: *Alistair Cockburn, Kent Beck, Ward Cunningham, Martin Fowler, Ken Schwaber, Jeff Sutherland* a další. V průběhu své mnohaleté praxe tito odborníci identifikovali v oboru SI řadu problémů a přemýšleli o možnostech jejich zlepšení. Začali nezávisle na sobě pracovat na návrhu nového přístupu k programování. Postupně zjistili, že ač se jejich snahy určitým způsobem liší, obsahují zároveň překvapivé množství společných principů. V únoru 2001 se společně sešli v americkém Utahu a cílem jejich setkání bylo:

- Podrobně analyzovat jednotlivé nové metodiky, na kterých pracují.
- Najít společné prvky a formulovat zastřešující teze.

Účastníci byli překvapeni, jak snadno našli společnou řeč a výsledkem setkání je tzv. **Manifest agilního vývoje softwaru** (The Agile Manifesto) [Beck et al. 2001], který se stal základem rodiny agilních metodik a přístupů. Tento manifest podpořilo jen do roku 2004 svým podpisem již přes 1400 zájemců z kruhů odborné veřejnosti po celém světě.

Zajímavé je, že základy řady metodik zařazených mezi agilní byly položeny mnohem dříve – např. metodika DSDM vznikla již v roce 1984 a kořeny filosofie Lean Development sahají až do 50. let minulého století. Agilní manifest je tedy třeba chápat jako jistý integrující prvek metodik s určitým společným jmenovatelem, který je popsán níže.

Česká terminologie okolo agilního přístupu a agilních metodik není dosud zcela ustálená. Používají se originální anglické termíny, jejich počeštěné verze, české překlady i slangová hantýrka. České výrazy vznikají bohužel povětšinou násilným překladem originálních publikací. Z tohoto důvodu se autor práce rozhodl používat jen obecně akceptované české výrazy kombinované s původní terminologií. Pokud je to smysluplné, je při zavedení termínu uveden český překlad i anglický originál.

5.3.2 Základní charakteristika

Agilní metodiky jsou svými autory označovány jako metodiky, které umožňují vytvořit řešení velmi rychle a efektivně a toto řešení dovolují pružně přizpůsobovat. Jedná se o několik metodik, z nich nejpopulárnější je takzvané Extrémní programování (XP).

Manifest deklaruje následující priority (citace z [Beck et al. 2001]):

Odhalili jsme lepší způsob vývoje softwaru, sami jej používáme a chceme pomoci i ostatním, aby jej používali. Dáváme přednost:

a) individualitám a komunikaci před procesy a nástroji,

b) provozuschopnému softwaru před obsažnou dokumentací,

c) spolupráci se zákazníkem před sjednáváním kontraktu a

d) reakci na změnu před plněním plánu.

Hlavním cílem je tedy **fungující software** a **splnění představ zákazníka** (tj. zajištění jakosti).

V dalším textu budou představeny tyto agilní metodiky:

- Extrémní programování;
- Metodika SCRUM Development Process;
- Lean Development;
- Feature Driven Development;
- Agile Unified Process;
- Adaptive Software Development;
- Dynamic Software Development Method.

Pro téma disertační práce jsou důležité i dvě doplňkové metody, které však z důvodu rozsahu v rešerši podrobněji neuvádím. jedná se o Agilní modelování [Ambler 2005, Pergl 2006] a techniku Test Driven Development [Beck 2002b].

5.3.3 Klíčové pojmy

Agilní metodiky vycházejí z následujících principů:

- Jedinou cestou, jak prověřit správnost navrženého systému, je co nejrychleji jej vyvinout, předložit zákazníkovi a na základě zpětné vazby upravovat. Fungující software je primárním měřítkem postupu;
- Teze, že „jediná jistota je změna“. Nemá smysl návrh přizpůsobovat budoucím požadavkům, protože ty se mohou změnit. Známé heslo SI „Implementujeme pro dnešek, navrhujeme pro zítřek“ tak AP upravuje na „Implementujeme pro dnešek, navrhujeme pro úspěšnou implementaci“. AP tedy vytváří řešení pro současnost a nezabývá se příliš budoucími potřebami. Až tyto potřeby vzniknou, budou řešeny. Současné řešení musí být pochopitelně založeno na takových paradigmatech a technikách, aby nebylo překážkou budoucího rozvoje;
- Práce týmu je odváděna na základě motivovaných jedinců a osobní zodpovědnosti;
- Nejefektivnější metodou komunikace je konverzace tváří v tvář;
- **Prinicip udržitelného vývoje** říká, že zákazník, vývojáři a tým by měli být schopni časově neomezeně udržovat konstantní tempo;
- Klíčový je důraz na **jednoduchost** ve všech aspektech;
- Důraz na samoorganizující a učící se týmy.

5.3.4 Extrémní programování

5.3.4.1 Vznik a vývoj

Extrémní programování je jedna z agilních metodik. Jejím autorem je Kent Beck a za základní publikaci je považována jeho kniha [Beck 2002a]. Kent Beck má za sebou dlouhou a bohatou kariéru vývojáře ve Smalltalku a systémového konzultanta. Extrémní programování je velmi mladá metodika (1997), principy v ní užívané jsou však všechny dlouho dobře známé.

5.3.4.2 Základní charakteristika

Její název vychází z filosofie vzít pár osvědčených věcí a ty dotáhnout do extrémů, např.: Pokud je dobré testovat, tak budeme testovat extrémně často. Pokud je dobré integrovat, tak budeme integrovat každou chvíli. Pokud tzv. „víc hlav víc ví“, tak budeme u počítače sedět vždy dva. Může se zdát, že tak nepřináší tolik potřebnou revizi zastaralých způsobů, ale opak je pravdou. Výsledkem jsou velmi revoluční, mnohdy až kontroverzní doporučení.

Ke třem proměnným zmíněným v kapitole 5.3.2 přidává XP ještě čtvrtou: *jakost.* Pracuje tedy se sadou proměnných náklady, čas, jakost, funkčnost. Důležité je, že tyto parametry nejsou nezávislé a jejich vzájemné závislosti jsou poměrně komplikovaném, jelikož každý z parametrů vyžaduje citlivé nastavení. Platí pravidlo, že vedení může týmu předepsat hodnoty maximálně *tří* ze čtyř proměnných.

Nasazení metodik extrémního programování vyžaduje především silnou motivaci a disciplínu všech zúčastněných - vedoucích pracovníků, programátorů, testerů, zákazníků, aj.. Základem XP prostupujícím všechna pravidla jsou *čtyři hodnoty*:

- Komunikace;
- Jednoduchost;
- Zpětná vazba;
- Odvaha.

Všechny postupy XP v sobě realizují minimálně jednu tuto hodnotu.

5.3.4.3 Klíčové pojmy

Aby bylo možné provádět správné korekce včas, je třeba zajistit **rychlou zpětnou vazbu.** Zpětná vazba také podstatně zlepšuje proces učení a získáváni zkušeností. Tento princip souvisí s principem přírůstkové změny .

Přírůstková změna (pravidlo drobných korekcí) říká, že velké změny by neměly být prováděny najednou, ale pomocí řady malých změn, které ještě mají vliv.

Pravidlo drobných korekcí říká [Beck 2004]: „Když chcete dobře řídit auto, je třeba provádět neustálé drobné korekce směru v závislosti na okamžité situaci. Nelze řídit tak, že vůz pouze nasměrujete na vzdálený bod na horizontu a strnule držíte volant.“ Toto pravidlo přispívá ke snížení rizika změny zadání[5] a k vnitřní stabilitě vývojového procesu.

Metodika XP řeši i psychologické a sociologické aspekty vývoje softwaru a kalkuluje **s přirozeným chováním** vývojářů a ostatních účastníků procesu.

Dalším pojmem je tzv. „**cestování nalehko“**, které říká že pro zajištění rychlého postupu je třeba, aby tým udržoval co nejmenší počet jednoduchých a hodnotných nástrojů, které jsou třeba pro úspěšné dosažení cíle.

Jádrem metodiky je tzv. 12 postupů: Plánovací hra *(The Planning Game)*, Malé verze *(Small Releases)*, Metafora *(System Metaphor)*, Jednoduchý návrh *(Simple Design)*, Testování *(Continuous Testing)*, Refaktorizace *(Refactoring)*, Párové programování *(Pair Programming)*, Společné vlastnictví *(Collective Code Ownership)*, Nepřetržitá integrace *(Continuous Integration)*, 40-ti hodinový týden *(40-Hour Work Week*, nověji *Sustainable Pace)*, Zákazník na pracovišti *(On-site Customer)*, Standardy pro psaní zdrojového textu *(Coding Standards)*

Plánovací hra

Vývoj softwaru v XP je vždy rozvíjející se dialog mezi možným a žádoucím. Zákazník a dodavatel spolupracují na zadání, aby zákazník mohl dostat co největší hodnotu v co nejkratším čase. Plánovací hra se skládá z těchto kroků:

1. Zákazník sepíše seznam požadovaných funkcí systému (obvykle funkčních celků) formou karet zadání (*User Story*). Každá karta nese určité jméno a je na ní v několika větách popsána požadovaná funkčnost;
2. Dodavatel zanalyzuje jednotlivé karty, odhadne náročnost (typicky člověkohodiny) a určí vazby mezi funkcemi (tj. kterou funkci je třeba mít hotovu pro implementaci jiné). Také odhadne, kolik je tým schopen vyprodukovat práce za jednu iteraci;
3. Zákazník poté určí pořadí, ve kterém se jednotlivé karty budou implementovat (tj. důležitost funkcí).

Pro interní potřeby týmu je běžné složitější karty dekomponovat do tzv. úkolů *(Engeneering Tasks)* a jednotlivé karty implementovat jako soubor úkolů. Vazby mezi kartami jsou potom implikovány vazbami mezi úkoly.

Malé verze

Tento postup znamená uvést minimální možný funkční systém do provozu a pak uvolňovat malé verze ve velmi krátkých cyklech (iteracích). Všechny nové verze by měly obsahovat jen ta nejcennější zadání. Verze by však měla dávat smysl jako celek, tj. je třeba vždy implementovat celou kartu. Délka iterace není nijak pevně stanovena a je třeba ji přizpůsobit objemnosti systému. Musí být nejkratší možná, ale dostatečně dlouhá na to, aby se s rezervou dala implementovat alespoň jedna karta. V praxi se délka iterace pohybuje typicky od jednoho týdne do jednoho, dvou měsíců.

Metafora

Řešitelský tým se zákazníkem (i mezi sebou) hovoří o systému pomocí jednoduše sdíleného příběhu o tom, jak má celý systém fungovat. Počítač může být třeba připodobňován k ploše stolu, výpočet důchodu k tabulce, atd. Metafora

[5] XP tvrdí, že změna je ve vývojovém procesu běžnou událostí.

pomáhá účastníkům projektu pochopit základní prvky a vztahy mezi nimi. Metafora slouží k lepšímu dorozumění a snazšímu vytvoření mentálního modelu.

Jednoduchý návrh

Systém by měl být navržen co nejjednodušeji, jak je to v daný okamžik možné. Nepotřebná komplikovanost je odstraněna hned, jakmile je zjištěna. Jelikož se požadavky mohou změnit, je implmentováno jen to, co je nutné pro aktuální chápání funkčnosti zákazníkem.

Testování

Testování je jedna z věcí, která je v XP dovedena do „extrému“. Existují dva typy testů:

- **Testy jednotek *(Unit Tests)*** - tyto testy píší programátoři *před* implementací funkce. Jedná se o nezávislé a zautomatizované testy pro *všechny* části systému, které by se mohly porouchat. Není samozřejmě možné testovat úplně každou metodu v každé situaci, ale je třeba naučit se odhadovat, které testy by se mohly vyplatit. Klíčovou roli zde hraje zkušenost. Všechny tyto testy musí probíhat na 100%;
- **Funkční testy *(Functional Tests n. Acceptance Tests)*** - tyto testy píší zákazníci za pomoci testerů, a to pro každou kartu. Nemusí být vždy zautomatizované. Jedná se o testy, kterými si zákazník ověřuje, že systém pracuje tak, jak si představoval. Zpravidla není nutné, aby tyto testy z počátku běžely na 100%, s postupem času se procento úspěchu s opravami chyb a implementací dalších funkcí zlepšuje.

Refaktorizace

Refaktorizace je změna systému, jež nemění jeho chování, ale vylepšuje některé charakteristiky přímo nesouvisející s funkčností systému, např. jednoduchost, pružnost, pochopitelnost, výkonnost. V XP se nejedná o doplňkovou činnost, ale je jedním z pilířů, na kterých XP stojí. K refaktorizaci se tým uchyluje, kdykoli je zjištěna její potřeba. Možnost refaktorizace je zvažována před i po přidání každé funkce do systému. Velké refaktorizace jsou prováděny v souladu s principem přírůstkové změny po malých krocích.

Párové programování

Veškerý produkční kód (tj. kód, který je součástí dodávaného systému) píší společně dva programátoři sedící u jednoho počítače. Ten, který v daný okamžik obsluhuje klávesnici a myš, píše kód a řeší aktuální problém. Druhý programátor kontroluje správnost kódu a přemýšlí o celkové koncepci. Důležité je, aby šlo o vyrovnaný dialog a ne jen o koukání přes rameno. Pro tvorbu párů nejsou stanovena žádná pevná pravidla. Seskupení do párů je dynamické a může se i v průběhu dne měnit.

Společné vlastnictví

Všichni mohou měnit jakoukoli část systému v jakoukoli dobu. Všichni jsou tedy obeznámeni se všemi částmi systému, ačkoli na různě úrovni. Jestliže pár pracuje a vidí příležitost ke zlepšení části zdrojového textu, začne jej zdokonalovat. Všichni tedy přijímají zodpovědnost za celý systém. Kód je třeba psát tak, aby z něj byl zřejmý záměr (tzv. *Intention-revealing*). Je proto vhodné používat jazyky co nejvyšší úrovně (z univerzálních jazyků nejlépe Smalltalk, Eiffel, atd.). Pokud je tým nucen používat jiné jazyky, je třeba o to více náležitě komentovat a strukturovat.

Nepřetržitá integrace

Kdykoli je nějaký úkol dokončen, systém je sestaven a integrován. Integraci je typicky prováděna i několikrát denně. Po integraci je každý zodpovědný za to, aby všechny testy jednotek běžely opět na 100%. Nepřetržitá integrace vyžaduje jistou režii navíc spojenou s odstraňováním kolizí, práce navíc však nebude nikdy mnoho, protože neustálá

refaktorizace způsobí rozložení systému na mnoho malých objektů a metod, za kterými nestojí dlouhá práce. Nespornou výhodou je, že nesrovnalosti v chápání části systému je odhalena hned v zárodku.

40-ti hodinový týden

Není důležité, jestli týden bude mít 35 či 45 hodin, ale tým nesmí být trvale přepracovaný. Jinak se snižuje schopnost soustředění, což se negativně projeví na výsledcích práce. Když je třeba, lze krátkodobě dělat přesčasy, ale ne více jak jeden týden v kuse. Ukazuje se, že přílišná potřeba přesčasů je příznakem vážného problému v projektu, který se takto nevyřeší.

Zákazník na pracovišti

V týmu by v ideálním případě měl sedět skutečný zákazník, který je k dispozici na plný úvazek a odpovídá na otázky týmu, řeší spory a určuje priority v menším měřítku. Tímto zákazníkem by měl být jeden z budoucích uživatelů systému nebo alespoň někdo, kdo přesně zná potřeby skutečných uživatelů *(Customer Proxy)*. Zákazník na pracovišti vytváří hodnotu v projektu tím, že specifikuje obsah testů funkčnosti (viz odstavec Testování). Značně také snižuje riziko projektu snížením potřeby plánovat příliš dopředu. Pokud není fyzická přítomnost zákazníka možná, je třeba alespoň zajistit nepřetržitou telefonickou dostupnost a časově flexibilní schůzky.

Standardy pro psaní zdrojového textu

Je třeba vytvořit standardy pro psaní zdrojového textu, které zlepší čitelnost kódu a komunikaci přes zdrojový kód. Tyto standardy musí všichni vývojáři bezpodmínečně dodržovat. V ideálním případě by nemělo být rozpoznatelné, kdo psal kterou část systému.

5.3.4.4 Proces vývoje pomocí metodiky

Proces vývoje je postaven na představených 12 postupech.

Fáze plánování

Zákazník projeví zájem o vybudování systému. Sejde se s týmem, kde společně vytvoří karty zadání. Tým zadání zanalyzuje - obvykle si jednotlivé karty ještě rozdělí na menší úkoly *(Engeneering Tasks)*, ohodnotí jednotlivé karty předpokládanou náročností a vytvoří relace prerekvizit.

Zákazník prostuduje analýzu a může některé funkce (např. z důvodu přílišné finanční náročnosti) třeba zrušit nebo i odstoupit od celého projektu. Pokud se projekt bude realizovat, zákazník ohodnotí karty důležitostí pro něj, tj. určí pořadí, ve kterém (s ohledem na prerekvizity) budou jednotlivé funkce realizovány. Domluví se podrobnosti (způsob financování, délka iterací, velikost verzí, kdo bude dělat zákazníka na pracovišti, atd.) a poté lze přistoupit k vývoji.

Fáze vývoje

V počáteční fázi vývoje dojde (ve spolupráci se zákazníkem na pracovišti) ještě k upřesnění nejasností a provede se analýza pro první iteraci. Pak začne vlastní vývoj v párech, který probíhá v krátkých iteracích:

1. Analýza úkolu;
2. Napsání testů;
3. Implementace úkolu;
4. Spuštění testů (+ev. oprava);
5. Integrace;

6. Spuštění testů (+ev. oprava).

Společné vlastnictví kódu navíc umožňuje (a vyžaduje) provést zjednodušení či refaktorizaci, kdykoliv vývojář identifikuje její potřebu. Po provedení úpravy je třeba se přesvědčit o konzistenci systému ověřením testy.

Fáze nasazení a údržby

Jakmile je hotova první minimální verze, systém je nasazen do provozu. Po dokončení každé verze provede obvykle vybraný tester ještě tradiční komplexnější testy (např. na výkonnost) a systém je předán zákazníkovi k otestování funkčními testy. Případné problémy jsou - dle závažnosti - buď ihned řešeny nebo je jejich řešení odloženo do další verze. Nasazení systému tedy není v XP speciální událost. Systém je uveden do provozu co nejdříve a paralelně jsou vyvíjeny nové verze a opravovány nedostatky zjištěné provozem.

5.3.4.5 Shrnutí

Metodika Extrémní programování je propracovaná metodika zaměřená především na programovací techniky. Její principy se navzájem doplňují a podporují [Pergl 2005a]. Její výhodou je její popularita, díky níž existuje dostatek literatury, i český překlad [Beck 2002a]). Díky většímu počtu uživatelů lze (na internetu) nalézt i obsáhlá diskusní fóra, postřehy z praxe a lze tak obdržet radu i konzultaci.

Metodika není zcela univerzální, o jejím nasazení lze uvažovat u projektů, které splňují určité předpoklady. Výhody XP se nejvíce projeví u projektů, kde se v průběhu vývoje často mění specifikace. Projekty vhodné pro XP musí především vyhovovat v těchto bodech:

- Velikost projektu odpovídá velikosti týmu (max 10-12 členů);
- Přiměřený rozsah a odborná náročnost problému (XP zavádí společné vlastnictví kódu, a tím i nutnost všeobecných znalostí pro všechny vývojáře, což znesnadňuje úzkou specializaci);
- Projekt nesmí být v rozporu s některými principy XP. U některých projektů může být třeba problém s nemožností získávání rychlé zpětné vazby, svázání určitými nutnými formálními postupy (např. ISO 9000) nebo nemožnost automaticky a rychle testovat software (např. u programů pro řízení technologických zařízení);
- XP není vhodné (přesněji řečeno je zbytečné) pro projekty, kde je nutné před započetím provést důkladnou analýzu a té se striktně držet (např. vědecké, vládní či armádní zakázky);
- Systém musí být granulovatelný, aby ho bylo možné rozdělit na karty zadání a dodávat přírůstkově;
- Systém musí mít objektové rysy. XP není vhodné například pro projekty zaměřené na datové modelování, jako jsou Data Warehouse.

Pomocí XP je možné vyvíjet nejen zakázkové systémy, ale i tzv. „krabicové produkty“. Roli zákazníka potom přebírá marketingové oddělení, které identifikuje zadání, které chce trh, kolik z každého zadání je zapotřebí, v jakém pořadí by měla být zadání implementována. atd.

Autor metodiky doporučuje zavést metodiku celou, což často nebývá možné. Od přijetí metodiky také může odrazovat její „extrémní“ terminologie, která může v managementu (a hůře v zákaznících) evokovat představu divokého, nekontrolovatelného procesu (či chaosu) s nejasnými zárukami, nicméně úspěšná praxe v zahraničí i u nás

ukazuje, že zavedení XP znamená často více organizovaný vývoj, než praktiky UP prováděné účelově z formálních důvodů a nikoliv pro zkvalitnění vývoje.

5.3.5 Metodika SCRUM Development Process

5.3.5.1 Vznik a vývoj

je agilní metodika vývoje softwaru, jejímž cílem je především zvýšení efektivity při vývoji softwaru. Podobně jako další agilní postupy i SCRUM využívá výhod iterativního a inkrementálního přístupu. SCRUM inklinuje k objektově orientovanému vidění světa dle Booche ([Booch 1994]), ale organizuje a řídí proces zcela novým způsobem.

První nástin metodiky SCRUM byl představen *Kenem Schwaberem* a *Mikem Beedlem* v roce 1995 [Schwaber 1996]. Oba autoři předtím prošli procesem vývoje softwaru v několika společnostech a týmech a stejně jako pro Kenta Becka, bylo pro ně hlavní motivací vyvinout flexibilní metodiku, která se dokáže lépe vyrovnat s měnícími se požadavky a která vylepší produktivitu vývoje softwaru.

Název metodiky vznikl z anglického výrazu Scrum, který se používá v ragby. Jeho význam je skrumáž, mlýn, situace kdy se několik hráčů (typicky mnoho) shlukne na jednom místě za účelem společného dotlačení míče na požadovanou pozici. Paralela s vývojem softwaru je taková, že vývojový tým by se měl shluknout podobným způsobem, aby „dotlačil“ projekt do zdárného konce.

5.3.5.2 Základní charakteristika

SCRUM vychází z objektově orientovaného přístupu, díky němuž odpovídá každý vývojář za *množinu objektů s jasně definovaným chováním a rozhraním*. Vývoj pomocí metodiky SCRUM probíhá v rámci *tři až osmi posloupností pevně daných časových intervalů*. Tyto intervaly se nazývají **sprinty**. Každý z těchto sprintů trvá obvykle zhruba *měsíc*. V rámci sprintů nedefinuje SCRUM žádné konkrétní procesy, předpokládá však denní schůzky – **Scrum meetings**, z nichž vzejde konkrétní určení činností. Tyto meetingy jsou nejoriginálnějším prvkem metodiky a mají řadu významů, od shrnutí dosavadního pokroku přes předvedení mezivýsledků a identifikaci nových úkolů až po zvyšování soudružnosti týmu a pěstování mezilidských vztahů. Scrum meetingy nahrazují centrální plánování a jsou odezvou této metodiky na všeobecný předpoklad agilních přístupů o dynamických změnách během vývoje.

Klíčovým pojmem v metodice je **riziko.** Metodika silně dbá na analýzu rizik. Rizika jsou revidována na konci každé iterace, ale i v průběhu každého sprintu v rámci každodenních schůzek. Rizika neformují jen náplně iterací, ale i obsah dokumentů, funkčnost verzí a další podstatné atributy.

5.3.5.3 Klíčové pojmy

Projekty vyvíjené pomocí metodiky SCRUM mají několik základních charakteristik, jejichž společným jmenovatelem je flexibilita a spolupráce:

- **Flexibilní předměty dodání** (*Flexible Deliverables*) – obsah dodávky je diktován prostředím. Pro některá prostředí je vhodný model, pro jiná prototyp či detailní specifikace dle normy IEEE;
- **Flexibilní harmonogram** (*Flexible Schedule*) – SCRUM akceptuje určitý výkyv v harmonogramu oběma směry. Je třeba zákazníkům vysvětlit, že SCRUM preferuje upřímné deklarování tohoto faktu před nalháváním pohádek o termínu dokončení, který je v klasických metodikách zpravidla výrazně překročen (či je „ošizena" jiná proměnná);
- **Malé týmy** – tým má mezi pěti a devíti členy. Na jednom projektu však může pracovat více týmů (max okolo 10). Tím je definován i rozsah projektů, který musí být v rozumném čase zvládnutelný takovýmto týmem. Důležitým pojmem je *samoorganizace* týmu;
- **Časté revize** – dosažený pokrok a případné změny jsou předmětem neustálého zkoumání, jednak v rámci Scrum Meetings a také při práci pracovníků;
- **Spolupráce** – podobně jako v XP se očekává spolupráce napříč vývojovým týmem i vzhledem k zadavateli. Oproti XP SCRUM explicitně nepředepisuje přítomnost zákazníka na pracovišti, nicméně také doporučuje úzký kontakt s přestaviteli zákazníka.

Oproti XP neobsahuje SCRUM pravidlo kolektivního vlastnictví kódu, ale za každý objekt (či množinu objektů) je odpovědná konkrétní osoba v týmu.

SCRUM také definuje některé specifické pojmy:

- **Backlog –** agenda projektu. Nese informace o tom, které části systému je nutné ještě implementovat nebo provést. Metodika nespecifikuje přesně obsah ani formu backlogu – lze použít User Stories z XP, CRC karty, tabulky, atp, důležité je, aby obsahoval úkoly a jejich priority. Metodika říká, že backlog je modifinován pouze jednou osobou (manažerem projektu), která také řídí požadavky podle priority. Backlog je veřejně přístupný všem členům týmu na viditelném místě;
- **Sprint –** jeden z fundamentálních pojmů metodiky. Sprint je základní vývojovou etapou (iterací), probíhá obvykle 30 dní. Délka jednoho sprintu je ovlivněna komplexností produktu, množstvím rizik, velikostí týmu a dalšími faktory;
- **Scrum –** jednodenní iterace v rámci sprintu. Během každého Scrumu se koná Scrum Meeting;
- **Scrum Meeting –** každodenní setkání vývojového týmu sloužící k přehledu vykonané práce, plánu příštího dne a k pojmenování všech důležitých aktuálních témat (rizik, změn, apod.). Typická délka trvání je mezi 15 a 30 minutami;
- **Scrum Master –** člen týmu, který vede a moderuje Scrum Meeting. Je to obvykle manažer projektu, ale není to pravidlo a Scrum Master nemusí být každý den stejný.

5.3.5.4 Proces vývoje pomocí metodiky

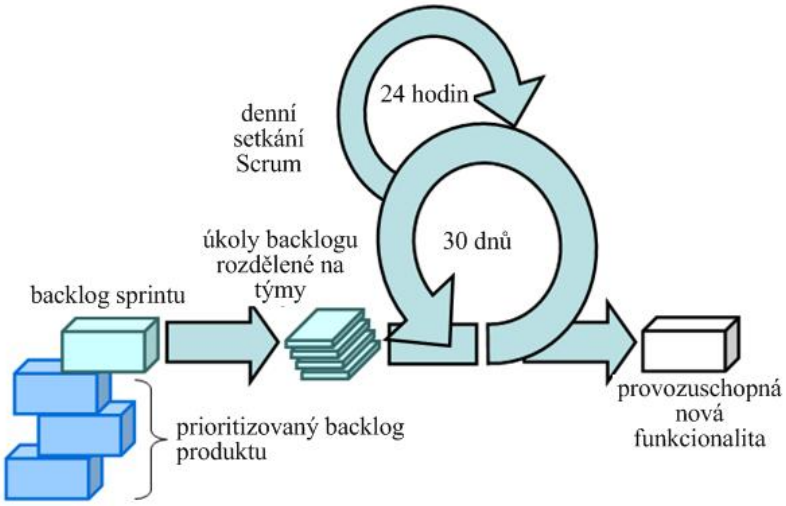
Vývoj podle metodiky SCRUM sestává ze tří fází:

1. **Předehra** (*Pregame*):

- Plánování (*Planning*);
- Systémová architektura (*System Architecture*);
- Vysokoúrovňový návrh (*High-level Design*);

2. **Hra** (*Game*):
   - Sprinty (souběžné provádění) (*Sprint*);
   - Vývoj (*Development*);
   - Zabalení (*Wrap*);
   - Revize (*Review*);
   - Přizpůsobení (*Adjust*);
3. **Dohra** (*Postgame*):
   - Uzavření (*Closure*).

Předehra a dohra jsou lineární, zatímco hra je iterativní.



(upraveno z http://accelerate.mountaingoatsoftware.com)

Obr. 9 – Proces vývoje v metodice SCRUM

Plánování

V rámci plánování se definuje rozsah aktuální verze, harmonogram, nutné zdroje, atd. V průběhu plánování jsou úkoly, které je třeba udělat zaznamenány do *backlogu* (5.3.5.3). Ty se poté mapují na objekty, provádí se analýza rizik, volí se vývojové nástroje, atd. Množství práce z backlogu pro aktuální sprint si však volí tým sám, není mu direktivně přidělován.

Architektura a Design

Při této činnosti se vytváří nebo modifikuje architektura v závislosti na nových požadavcích, poznatcích, rizicích. Je prováděna doménová analýza a související činnosti.

Hra

Hra se skládá z vývoje, zabalení, revize a přizpůsobení (Obr. 9). Vývoj je iterativní cyklus vývojových prací, který probíhá po *sprintech*. Obsah sprintu je určen *backlogem*, který je řazen podle priorit jednotlivých požadavků. O prioritách požadavků rozhoduje zákazník.

Na začátku každého sprintu je vytvořen tzv. *Sprint Backlog*, kdy manažer vybere skupinu požadavků, které budou v rámci sprintu implementovány. Volba konkrétních požadavků z backlogu probíhá na základě řady kritérií.

Vývojový tým následně provedeexpanzi a podrobné zmapování všech požadavků z daného backlogu. Zároven je nutné stanovit přibližný postup a plán vývoje v závislosti na zvolených požadavcích a na očekávaných výstupech.

Poté následuje samotná náplň sprintu, kdy se provádí návrh a implementace požadavků v backlogu. Součástí každého sprintu jsou řídicí činnosti (včetně řízení rizik), které zabraňují chaosu, ale nepotlačují flexibilitu.

V průběhu vývoje se na konci každého vývojového dne koná 15-30 minut Scrum Meeting, na kterém se jednak prezentují dosažené výsledky a řeší problémy technické, komunikační i organizační.

Každý sprint je zakončen schůzkou, které se účastní všichni členové týmu i vedení a popřípadě i zákazník a další zainteresované osoby. Na schůzce se předvede funkční prototyp (nová verze), detekují se nové položky *backlogu* a provede se hrubý nástin obsahu následujícího *sprintu*.

Projekt je stále přístupný změnám až do fáze uzavření. Po celou dobu je možné a žádoucí provádět změny.

Na konci každé vývojové iterace se vyhodnotí aktuální backlog, detekují se položky přecházející do další iterace a hledají se nové položky.

Rizika jsou revidována na koncí každé iterace, ale i v průběhu každého sprintu v rámci každodenních schůzek.

5.3.5.5 Shrnutí

SCRUM je metodika zaměřená hlavně na organizaci vývoje. Oproti XP je konzervativnější v ohledu vlastnictví kódu – místo kolektivního vlastnictví je definována zodpovědnost za každý objekt či množinu objektů. Výhodou je zodpovědnost a možnost specializace, na druhou stranu, pokud vlastník opustí tým, může nastat problém s předáním objektu kompetentnímu následovníkovi.

Výhodou metodiky SCRUM je především větší orientace na *řízení týmu* a na zásahy vedení organizace. Je tedy vhodný pro týmy, které potřebují pevnější vedení než v XP.

Další velkou výhodou je důraz na řízení rizik v průběhu celého vývoje.

5.3.6 Lean Development

5.3.6.1 Vznik a vývoj

Metodika Lean Development nevznikla původně jako metodika SI, ale má své kořeny v řízení průmyslové výroby. Myšlenky převzaté z výrobních odvětví byly pouze přizpůsobeny pro vývoj softwaru.

Kořeny Lean Development (*lean* znamená „štíhlý“, „zeštíhlený“) sahají do 80. let 20. stol., kdy Japonští výrobci automobilů začali hledat efektivnější, pružnější a levnější způsob řízení výroby aut. V této době vznikla řada zajímavých myšlenek a jednou z nich je i *Lean Manufacturing*, která byla vyvíjena Číňanem Taichi Ohno pracujícím pro automobilku Toyota již od 50. let. Metodika tedy byla vyvíjena bezmála třicet let.

Základem Ohnova návrhu bylo *absolutní odstranění všeho zbytečného*, co v průběhu vývoje vznikalo a co mohlo snížit efektivitu a zvýšit náklady. V roce 1980 bylo definováno *10 pravidel*, která mají napomoci k praktickému naplnění idejí systému Lean Manufacturing.

V rámci agilního snah na konci minulého tisíciletí byla potom pravidla Lean Manufacturing přizpůsobena pro oblast softwaru a nazvána Lean Development.

5.3.6.2 Základní charakteristika

Lean Development ([Poppendieck 2003]) je definován jako systematický přístup k identifikování a eliminaci možných zdrojů plýtvání v průběhu celého vývojového procesu, který se pokouší zákazníkovi dodat perfektní produkt a splnit tak jeho požadavky.

Metodika si klade tyto cíle:

- Vyvíjet software za třetinu obvyklého času;
- Vystačit s třetinou obvyklého rozpočtu;
- Snížit četnost chyb na třetinu obvyklého množství.

Jedním z nejdůležitějších pravidel metodiky Lean Development je *ignorování zbytečností* a provádění jen toho, co *poskytne hodnotu výsledné aplikaci*. Ačkoliv se jedná o přirozený požadavek, je až s podivem, jak málo projektů tuto samozřejmost beze zbytku naplňuje.

Věci, které Lean Manufacturing považuje za hodnotné jsou:

- Suroviny a základní materiály;
- Výsledný produkt, který je někdo ochoten koupit;
- Nástroje používané týmem při výrobě produktu;
- Dovednosti získané týmem.

Oproti tomu meziprodukty, modely a další průběžné artefakty jsou považovány za důležité pouze v případě, kdy meziprodukt pomůže týmu dokončit výsledný produkt levněji než bez něj.

5.3.6.3 Klíčové pojmy

Hodnota a zbytečnost

Metodika Lean Development je založena na pojmech **hodnota** a **zbytečnost**. Věci, které Lean Manufacturing považuje za hodnotné jsou:

- Suroviny a základní materiály;
- Výsledný produkt, který je někdo ochoten koupit;
- Nástroje používané týmem při výrobě produktu;
- Dovednosti získané týmem.

Oproti tomu meziprodukty, modely a další průběžné artefakty jsou považovány za důležité pouze v případě, kdy meziprodukt pomůže týmu dokončit výsledný produkt levněji než bez něj.

10 pravidel

Metodika Lean Development je postavena na **deseti obecných pravidlech**, která byla definována už pro systém Lean Manufacturing:

1. Odstranit vše, co je zbytečné;
2. Minimalizovat zásoby (minimalizivat meziprodukty);
3. Maximalizovat tok (zkrátit čas potřebný pro vývoj);
4. Vývoj je „tažen poptávkou“ (většina rozhodnutí probíhá i na nejnižších úrovních);
5. Pracovníci mají pravomoci rozhodovat (rozhodování probíhá i na nejnižších úrovních);
6. Hlavním cílem je uspokojovat požadavky zákazníků (a to nejen teď ale i v budoucnu);
7. Zvést zpětnou vazbu (nebát se změn v učiněných rozhodnutích);
8. Odstranit lokální optimalizaci (neustálé optimalizace stávajícího řešení postrádají smysl);
9. Vybudovat partnerství s dodavateli (využívat subdodávek a předpřipravených komponent);
10. Vybudovat kulturu pro možnost neustálého zlepšování.

5.3.6.4 Proces vývoje pomocí metodiky

Proces vývoje metodika Lean Development přesně nestanovuje, ve svých pravidlech pouze naznačuje, jakým způsobem by měl vývoj probíhat, jak by měl být prováděn, čeho se vyvarovat a kam by měl směřovat.

Odstranění zbytečností

Jako zbytečné se mohou ukázat různé formy specifikací, modelů, diagramů, návrhů a dalších meziproduktů. Dle zásad Lean Development mají opodstatnění jen ty meziprodukty, jejichž odstraněním by byl negativně ovlivněn výsledný produkt. Různé dokumenty a meziprodukty spotřebovávají cenné zdroje a negativně tak ovlivňují výsledný produkt. Zdroje jsou spotřebovány na:

- Vytvoření meziproduktu;
- Proces jeho schvalování a posuzování;
- Udržování v aktuálním stavu.

Je tedy třeba si položit otázky:

- Které meziprodukty používat?
- Jaký mají mít účel a obsah?
- Jaký mají mít rozsah?

Optimální vyladění meziproduktů je velmi náročné na zbavení se předsudků a vyžaduje hodně odvahy (odvaha je i jednou ze základních hodnot v XP), neboť vždy lze nalézt spoustu „rozumných" důvodů, proč právě ten či onen meziprodukt je „naprosto klíčový" ve své současné podobě.

Minimalizace zásob

Druhé pravidlo metodiky hovoří o tom, že je zbytečné vyvíjen a vytvářet cokoliv, co není zrovna teď potřebné. Součástí tohoto pravidla je také pokyn „minimalizovat množství dodávaných artefaktů". Z hlediska softwaru se jedná opět o otázku množství a obsahu dokumentace podobně jako v předchozím bodě.

Maximalizace toku

Při tradičních způsobech postupu vývoje (Obr. 3, Obr. 4) je rychlost provádění limitována nejpomalejší činností a v průběhu provádění jedné fáze jsou pracovníci následujících fází v nečinnosti. Tento způsob vedení projektu bývá označován *„push systém"*. Tento systém může pracovat velmi efektivně, pokud ho organizace používá dostatečně dlouhou dobu a podařilo se jí optimálně vyladit poměr jednotlivých činností a pracovníků na jednotlivé činnosti. Jakmile je však projekt zatížen nejednoznačností a neznáme přesně dobu trvání jednotlivých činností, je vyladění systému velmi obtížné.

Metodika Lean Development doporučuje jako řešení zavést **iterativní** vývoj a zkrátit délku jednotlivých fází na miminum. To umožní snížit nevyrovnanost využití zdrojů (pracovníků). Iterace též umožní průběžné předvádění prototypů a betaverzí zákazníkovi.

Vývoj je tažen poptávkou

Směr vývoje by měl v každém okamžiku přesně odrážet poptávku zákazníka. Poptávka se však přirozeně mění – základní teze manifestu agilního vývoje říká, že „jedinou jistotou je změna" [Beck et al. 2001]. Lean Development proto doporučuje provádět všechna rozhodnutí co nejpozději, neboť se požadavky mohou kdykoliv změnit. V praxi toto pravidlo vyžaduje značnou zkušenost a cit, aby rozhodnutí bylo učiněno co nejpozději, ale ne pozdě.

Pracovníci mají pravomoc rozhodovat

Toto pravidlo je v souladu s obecným trendem agilních přístupů. Lean Manufacturing má kořeny v Japonsku, Lean Development je proto v tomto ohledu opatrnější a nepočítá od počátku s naprostou demokracií. Doporučuje zkusit, kolik pravomocí je možné týmu svěřit, aby pracovníci udrželi efektivitu a zaměření na výsledek.

Hlavním cílem je uspokojovat požadavky zákazníků

Lean Development v souladu s AP počítá s tím, že

1. Zákazník na počátku nemusí mít přesnou představu o tom, co od projektu vlastně očekává;
2. Požadavky na výsledný projekt se mohou během vývoje měnit.

Jedná se o shodný přístup jako např. v XP (kap. 5.3.4).

Zavedení zpětné vazby

Pravidlo „Vývoj je tažen poptávkou“ a „Hlavním cílem je uspokojovat požadavky zákazníků“ mj. říkají, že nelze definovat všechny požadavky v úvodu projektu a požadavky se mohou měnit. Proto je třeba zavedení zpětné vazby, tj. časté odezvy od zákazníka (podobně jako v ostatních agilních metodikách).

Zpětná vazba s sebou nutně přináší potřebu změn již hotové části systému a tím také nebezpečí zanesení chyb. Tato nebezpečí jsou podobně jako v XP eliminována důsledným používáním refaktoringu a vývoje řízeného testováním.

Odstranění lokálních optimalizací

Lean Development říká, že bychom se neměli zbytečně dopouštět plýtvání energií na provádění různých dílčích optimalizací, které ve svém důsledku nepomohou zvýšení celkové efektivity. Optimalizace musí být spíše prováděny komplexně a s vědomím souvislostí, aby byl celý proces vyvážený a nevznikala úzká hrdla.

Partnerství s dodavateli

Velkým přínosem metodiky Lean Development je diskuse otázky subdodávek a nákupu komponent. Jako ideální považuje situaci, kdy je vybudován prověřený řetězec subdodavatelů. V okamžiku nákupu hotové součásti se totiž stáváme závislí na dodavateli a jakosti jeho výrobku a služeb zahrnujících potřebnou budoucí podporu. Nefungující podpora může v důsledku vést ke značnému zkomplikování či prodražení projektu či přímo k jeho pádu, i přes to, že vše jinak může fungovat excelentně. Na druhou stranu hojným používáním jakostních hotových komponent od prověřených dodavatelů lze ušetřit nemalé prostředky na vývoj a podstatným způsobem zrychlit a zefektivnit celý proces vývoje.

S touto problematikou souvisí i požadavek na vedení, které by mělo programátorům zajistit možnosti dostatečné uživatelské podpory pro nástroje a technologie používané týmem. Je vhodné mít též v případě potřeby dostupné konzultace odborníků z oblasti programovacích jazyků i aplikačních domén. Toto zajištění ovšem nesmí být v rozporu s principem „Odstranění zbytečností“.

Kultura pro neustálé zlepšování

Tento bod říká dvě věci: Za prvé by mělo docházet k neustálému zlepšování na všech úrovních: na úrovni schopností týmu, řízení projektu, vztahy v týmu i se zákazníkem, atd. Slovo „kultura“ upozorňuje, že pro takovýto všeobecný růst je třeba vytvořit vhodnou firemní kulturu, která bude pracovníky motivovat odvádět maximum.

Principy a nástroje

Někteří autoři [Poppendieck 2003] definují **sedm principů** vzešlých z uvedených pravidel, které poskytují praktické návody k realizaci těchto principů v praxi. Jsou to:

- Eliminace plýtvání;
- Rozvinout učení;
- Pozdní rozhodování;
- Časté dodávky;
- Pravomocní pracovníci;
- Integrita;
- Vidění celku.

Jelikož tyto principy nepřináší nic nového, pouze rozvádějí představených deset pravidel, nebudu je zde z důvodu rozsahu a zaměření práce dále rozvádět.

5.3.6.5 Shrnutí

Metodika Lean Development je zajímavá svým původem z průmyslového prostředí. I přes odlišnost disciplíny SI od ostatních inženýrských oborů lze nalézt mnoho analogií a inspirace. Ve svém důsledku je tato metodika naprosto koherentní s principy agilního vývoje.

Lean Development je zaměřena strategičtěji než ostatní agilní metodiky. Vyznává hodně principů společných s XP (vývoj řízený testováním, refaktoring, zpětná vazba, krátké iterace, atd.), nicméně je v některých ohledech opatrnější (např. svěřování pravomocí pracovníkům).

Hlavním přínosem metodiky je zaměření na pojmy *hodnota* a *plýtvání*, které jsou cestou k zefektivnění procesu a odstranění zbytečných činností a artefaktů. Velkým přínosem metodiky je též explicitní diskuse otázky efektivních *subdodávek a používání komponent*.

Ačkoliv metodika na první pohled ve srovnání např. s XP působí strohým, chladným a odlidštěným dojmem, zdůrazňuje důležitost příjemného motivujícího prostředí, ve kterém se všichni budou cítit dobře a odvádět maximální výkony.

5.3.7 Feature Driven Development

5.3.7.1 Vznik a vývoj

FDD ([Felsing 2002, Buchalcevová 2005]) bylo představeno *Jeffem De Lucou* a *Peterem Coadem* v devadesátých letech minulého století. Jeff De Luca měl již za sebou bohatou kariéru jako vývojář i projektový manažer. Peter Coad je jedna z nejuznávanějších legend SI a zaměřuje se především na objektově orientované koncepce.

Jeff De Luca dal za dobu své kariéry pozoroval vývoj a srovnával novinky v oblasti SI. Zaujaly ho myšlenky Millových chirurgických týmů, DeMarcovy definice projektu jako takového nebo Faganovské inspekce. Peter Coad přišel v roce 1997 s myšlenkou, že „features“, tedy vlastnosti produktu jsou při vývoji klíčovým stavebním kamenem. Luca a Coad dali dohromady všechny uvedené myšlenky a přidali k nim vlastní názory a náměty. Výsledkem byl pokus definovat jakýsi „recept“, po jehož použití by se stal vývoj softwaru jednodušší, čitelnější a efektivnější.

5.3.7.2 Základní charakteristika

FDD je založeno na vývoji, který je řízen **vlastnostmi produktu**. Je podobně jako ostatní agilní metodiky založeno na iterativním vývoji s krátkými iteracemi a klade silný důraz na efektivitu a výsledný produkt. Součástí FDD je také **modelování,** a to objektové, typicky v UML. Vývoj podle metodiky FDD začíná vytvořením celkového, globálního modelu systému. Vývoj podle FDD poté pokračuje posloupností krátkých iterací obsahujících návrh a realizace pro jednotlivé vlastnosti. Vždy po skončení každé iterace (cca 1-3 týdny) je zákazníkovi dodán fungující meziprodukt.

Metodika přesněji specifikuje své pojmy a procesy než ostatní agilní metodiky a v jejím rámci mohou spolupracovat i větší týmy (do dvou až tří desítek programátorů).

5.3.7.3 Klíčové pojmy

Feature

Klíčovým pojmem je samozřejmě **feature**, tedy **vlastnost.** Ta je z pohledu FDD definována jako malý výsledek (malá část funkčnosti) užitečný z pohledu zákazníka. Vlastnosti mají tyto charakteristiky:

- **Srozumitelnost** – musíme být schopni vlastnost popsat a pochopit;
- **Ověřitelnost –** musíme být schopni rozhodnout, zda je implementovaná vlastnost totožná s vlastností, kterou zákazník požadoval.
- **Realizovatelnost** – musíme mít jistotu, že vlastnost lze realizovat a doba její realizace nebude nepřiměřeně dlouhá. Vlastnost by měla být realizovatelná spolu s s dalšími souvisejícími vlastnostmi v rámci jedné dvoutýdenní iterace. Pokud by byla doba potřebná pro realizaci vlastnosti příliš dlouhá, je třeba ji rozčlenit do více menších vlastností.

Formát vlastnosti je daný a měl by mít podobu

<akce> <předmět> <podrobnosti>,

např. „vypočtení celkového součtu prodejů za loňský rok“, „načtení předpřipravených hodnot ze síťového serveru“, „zjištění zůstatku bankovního účtu“, atp. Nevhodné vlastnosti jsou např. „podpora zákazníků“ či „modul pro správu bankovní sítě“, jelikož jsou příliš obecné, rozsáhlé a nepřesné. Na druhou stranu, vlastnost by se neměla zúžit na pouhou přístupovou metodu, která uloží nebo načte hodnotu atributu.

Z hlediska procesního modelování lze říci, že v obchodním systému je vlastností obvykle jeden krok v nějaké aktivitě daného obchodního procesu. Z pohledu hotové aplikace může vlastnost odpovídat jedné položce nabídky, jedné možnosti nebo jedné operaci spouštěné uživatelem.

Přístup řízený modelem

FDD je založeno na přístupu řízeném modelem (*model-driven approach*), což znamená, že klade velký důraz na **modelování**. Vývoj každého systému začíná vytvořením globálního modelu a každá iterace následně disponuje svým vlastním, konkrétním modelem. Model je nutný pro udržení konzistence systému a celkového přehledu jeho vlastností.

Další prvky

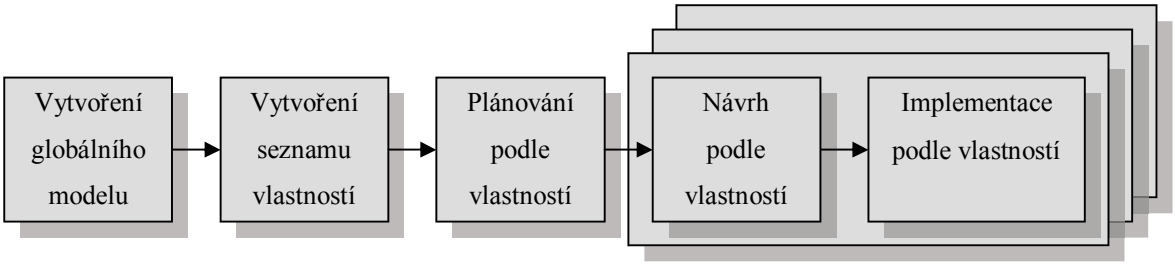
FDD používá všemožné techniky **inspekce**, tj. postupy k detekci chyb a problémů při vývoji a zavádí pravidelné vytváření **rozvrhů**, čímž vnáší do vývoje určitý časový řád a umožňuje lépe plánovat jednotlivé akce. Dále obsahuje tradiční **konfigurační řízení**, **řízení jakosti** a **testování**.

5.3.7.4 Proces vývoje pomocí metodiky

Metodika definuje pět tzv. **lehkých procesů** (*light processes*, Obr. 10). V závorkách jsou uvedeno hrubé procento času celkové doby projektu strávené těmito procesy:

1. Vytvoření celkového (globálního) modelu (10% úvodní, 4% opakované);
2. Vypracování podrobného seznamu vlastností (4% úvodní, 1% opakované);
3. Plánování podle vlastností (2% úvodní, 2% opakované);
4. Návrh podle vlastností (77 % dohromady s 5.);
5. Implementace podle vlastností.

přičemž kroky 1-3 jsou prováděny na začátku projektu a kroky 4-5 jsou iterativní. Po implementaci každé vlastnosti je možné se vrátit zpět k první fázi a provést případná přizpůsobení a modifikace. Čísla v závorkách udávají přibližné procento doby vývoje strávené v jednotlivých fázích, přičemž se počítá se 7% času strávenými modifikacemi.



Obr. 10 – Proces vývoje ve FDD

Pro každý proces metodika specifikuje:

- Obecný popis procesu;
- Vstupní kritéria pro proces;
- Úkoly v rámci daného procesu (včetně řešitelů a zodpovědností);
- Nástroje a metody verifikace;
- Výstupní kritéria procesu.

Procesy jsou (oproti např. RUP) definovány spíše jako vodítka co dělat a nikoliv jak a předpokládá se, že pracovníci již mají vyvinuty kompetence a přehled v oboru a vědí, jak provádět jednotlivé činnosti.

Fáze 1: vytvoření globálního modelu

V první fázi nejprve spolupracují *doménoví experti* a *členové vývojového týmu* společně se zkušeným *systémovým architektem* a vytváří společně **základní (skeletový) model**. Poté je tato úvodní kostra rozpracována do podrobnějšího **globálního modelu**. Pokud je projekt rozsáhlejší, může být vytvořeno více menších týmů zaměřujících se na jednotlivé části systému.

Cílem této první fáze je vytvoření úvodního modelu a prvotního seznamu požadovaných vlastností**,** které vyplynou při schůzkách. Jsou též vypracovány úvahy o alternativách a případných dalších možnostech. Model vytvořený v této fázi je později iterativně aktualizován a upravován ve fázi 4.

Fáze 2: vytvoření seznamu vlastností

V této fázi je úzce vymezen systém a je vytvořen podrobný seznam vlastností. Tento seznam ale nemusí být zcela úplný a je možné ho později ve fázi 4 doplňovat. Seznam je rozdělen do kategorií a případně podkategorií podle oblastí funkčnosti. Jednotlivé vlastnosti jsou poté klasifikovány podle důležitosti pro zákazníka. Tento proces klasifikace je prováděn společně s týmem a je též brána v úvahu složitost implementace.

Na závěr této fáze se provádí identifikace konečného výsledku, kdy zákazník určí minimální množinu vlastností, které vyžaduje.

Fáze 3: plánování

V této fázi je vytvořen plán dalšího vývoje. Jeho součástí je i datum ukončení vývoje. Poté jsou rozpracovány skupiny plánů, které se týkají jednotlivých skupin vlastností. Každé skupině se přiřadí vlastník a stanoví se datum jejího ukončení. Nakonec je přiřazen vlastník a odpovědný člen týmu jednotlivým kusům programu: třídámči skupinám tříd v objektovém přístupu či modulům v případě strukturovaného přístupu.

Po rozpracování plánu je vytvořeno předběžné pořadí, v jakém se budou vlastnosti implementovat. Toto pořadí je vytvořeno na základě priorit a také vzájemných závislostí mezi vlastnostmi. Pořadí se může později modifikovat.

Fáze 4: návrh

Tato a následující fáze jsou již prováděny iterativně. V každé fázi návrhu hlavní programátor vybere jednu nebo více vlastností pro implementaci. Při výběru se řídí stanovenými prioritami a vzájemnými závislostmi. Vybírá se množina vlastností, která bude realizovatelná přibližně v jednom až dvou týdnech.

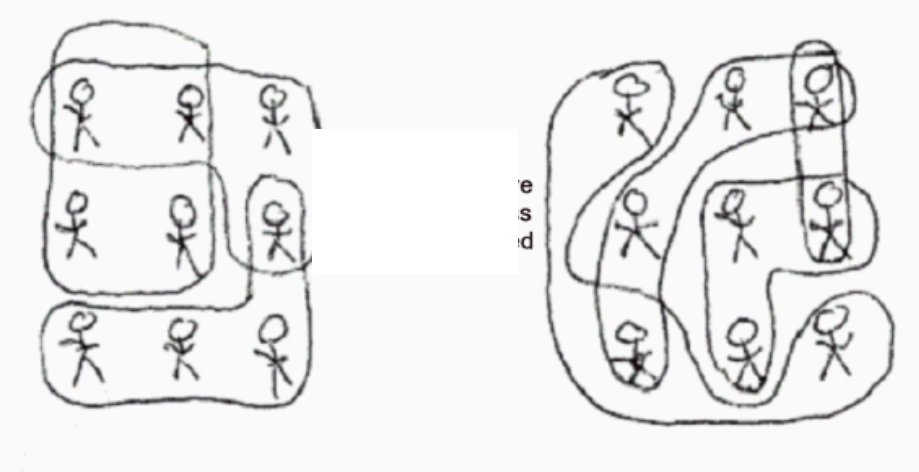
Hlavní programátor poté zahájí činnost nazvanou „**Design by Feature“** (DBF, návrh podle vlastností). Její proces spočívá v určení tříd, které danou vlastnost pokrývají a v kontaktování příslušných vlastníků těchto tříd. Vlastníci poté vytvoří dočasný tým, jehož úkolem je vypracování **detailního návrhu** pro implementaci vlastnosti a stanovení **plánu realizace**.

Fáze 5: implementace

Pátá fáze zahrnuje činnost označovanou „**Build by Feature“** (BBF, realizace podle vlastností). Vlastník každé třídy odpovídá za její návrh (který byl vytvořený v předchozí fázi) a za její implementaci. Vlastník píše metody, vytváří pro třídu testovací případy a provádí všechny potřebné testy. Po dokončení třídy je třída vložena do sdíleného **systému pro správu tříd** (class repository).

V okamžiku, kdy je hlavní programátor spokojen s dosaženým výsledkem, je vlastnost integrována do hlavní aplikace.

Vývoj organizuje hlavní programátor. Jeho náplní práce je výběr nových vlastností, integrace dokončených vlastností do aplikace a koordinace vývoje. Obvykle pracují dva až tři relativně nezávislé týmy, které souběžně pracují na nových vlastnostech. Složení týmu se dynamicky mění (Obr. 11) a každý vývojář může být členem více týmů, pokud je vlastníkem třídy, která souvisí s vlastností implementovanou týmem. Velikost týmu je obvykle mezi třemi a šesti vývojáři.



(převzato z http://bdn.borland.com)

Obr. 11 – Dynamické vytváření týmu v FDD: týmy jsou tvořeny podle vlastníků tříd

5.3.7.5 Shrnutí

Feature Driven Development je mezi agilními přístupy považována za „konzervativnější“ metodiku a má blíže ke klasickému přístupu než ostatní metodiky. Přesněji definuje své pojmy a proces a klade důraz na modelování.

Základním stavebním kamenem metodiky Feature Driven Development jsou „features“, tedy vlastnosti systému. Oproti např. metodice SCRUM a jejím pojmu backlog (viz kapitola 5.3.5.3) je feature v FDD přesně specifikována. Díky tomu je usnadněno provádění průběžných kontrol a kvantifikace vývoje. V porovnání s případy užití jsou velikosti jednotlivých vlastností na relativně stejné úrovni a nejsou tedy tak velké rozdíly v délce implementace. Metodika však počítá s menším prostorem pro změny během vývoje (v řádu procent).

FDD se klasickým metodikám přibližuje i v tom, že ve fázi plánování stanovuje pevné datum ukončení vývoje. Nepočítá tedy s flexibilním harmonogramem jako např. SCRUM.

FDD zavádí vlastnictví tříd a přiřazuje tak konkrétní zodpovědnost za určitou třídu konkrétnímu programátorovi (oproti společnému vlastnictví tříd v XP).

FDD pracuje s dynamickým vytvářením týmů, díky čemuž je možná škálovatelnost projektů a zvládá tedy větší projekty než ostatní agilní metodiky.

5.3.8 Agile Unified Process

5.3.8.1 Vznik a vývoj

Agile Unified Process pochází z dílny Scotta Amblera, významného konzultanta na poli softwarového inženýrství a objektově orientovaných metod.

5.3.8.2 Základní charakeristika

Agile Unified Process (AUP) [Ambler 2006] je zjednodušenou variantou Rational Unified Process popsané v kap 5.2). Zjednodušuje proces vývoje a zavádí do něj agilní praktiky.

AUP je v souladu s Manifestem založen na následujících filosofiích:

1. **Tým ví, co dělá** – není ho tedy potřeba nutit do detailní procesní dokumentace, striktního vedení, atd.
2. **Jednoduchost** – tj. výrazné zmenšení objemu „papírování.
3. **Agilnost** – AUP ctí zásady Manifestu agilního vývoje [Beck at al. 2001]
4. **Zaměření na aktivity s vysokou hodnotou** – zaměření na činnosti, které mají skutečně význam.
5. **Nezávislost na nástrojích** – tým může používat takové nástroje, jaké mu vyhovují (a samozřejmě jaké se hodí pro charakter projektu).
6. **Přizpůsobení AUP konkrétním potřebám**.

5.3.8.3 Klíčové pojmy

AUP má pouze 7 aktivit:

- Modelování;
- Implementaci;
- Testování;
- Dodání;
- Správu verzí;
- Projektové řízení;
- „Prostředí“, čímž označuje podpůrné procesy.

5.3.8.4 Proces vývoje pomocí metodiky

Proces se skládá ze stejných 4 částí, jako RUP:

- Začátek;
- Rozpracování;
- Konstrukce;
- Zavedení.

Krátké iterace AUP zajišťuje dodáváním tzv. Development Release (vývojových verzí) v rámci jednotlivých RUP iterací.

Ostatní principy a praktiky vycházejí ze zmíněných zakomponovaných agilních přístupů, nebudou zde tedy znovu rozebírány.

5.3.8.5 Shrnutí

Agile Unified Process se snaží upravit původní rigorózní Rational Unified Process tak, aby odpovídal filosofii Manifestu agilního vývoje. Ambler využil svých rozsáhlých znalostí a práce, kterou na tomto již vykonal a začlenil do RUP některé praktiky z agilního světa.

AUP je zajímavou kombinací rigorózního a agilního přístupu, nicméně dosud mu chybí rozsáhlejší praktiké využití.

5.3.9 Metodiky Crystal

5.3.9.1 Vznik a vývoj

Jedná se o rodinu metodik, které je možné přizpůsobit pro konkrétní projekt. Autorem je uznávaný Alistair Cockburn, který vyšel z myšlenky, že každý software je jiný, a že tedy každý vývojový proces je ve své podstatě odlišný a co funguje jednou, nemusí fungovat podruhé.

5.3.9.2 Základní charakteristika

Crystal [Coburn 2004] je rodina adaptibilních metodik, které sám autor označuje přívlastkem „ultralehké“. Metodika je především **zaměřena na lidskou stránku** projektu a klade důraz na lidský faktor a vývojářský tým. Snaží se zvýšit efektivitu vhodným složením vývojového týmu, prací s lidskými zdroji a vhodnými technikami vedení lidí.

Metodika se snaží snížit objem dokumentace a papírování na nejmenší hodnotu, která ještě dokáže zajistit úspěšné dokončení projektu.

Metodika se skládá ze souboru metodik, které se v určitých parametrech liší a pro konkrétní projekt je vždy použita jedna z nich.

5.3.9.3 Klíčové pojmy

Metodiky Crystal vychází ze tří základních tezí:

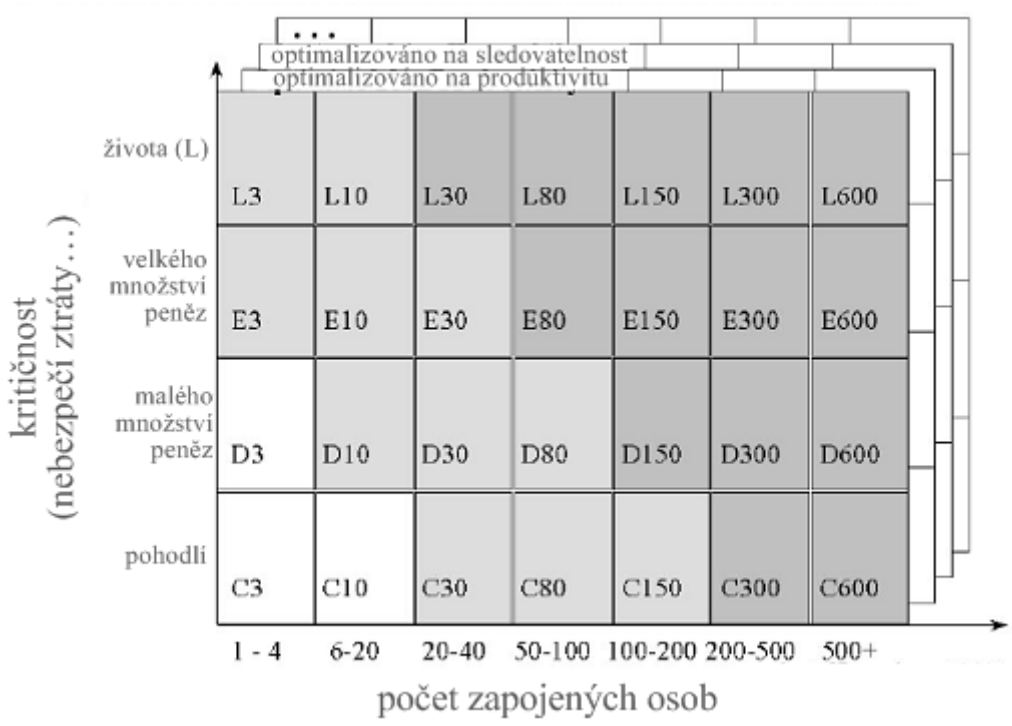
1. Každý softwarově-inženýrský projekt je vyznačuje (třebaže nepatrně, avšak neopominutelně) **odlišnou množinou parametrů**. Z toho důvodu také každý projekt vyžaduje poněkud jiný přístup a vedení;
2. Práce na projektu a jeho výsledek jsou nesmírně **závislé na lidských zdrojích** a přístupu členů týmu;
3. Základním kamenem úspěšného projektu (a také zdravého prostředí uvnitř týmu) je vzájemná, průběžná a otevřená **komunikace**.

Konkrétní metodika je z rodiny metodik Crystal vybrána na základě 3 parametrů:

- Důležitost problému;
- Počet lidí zapojených do projektu;
- Priority projektu.

a zvolená metodika je pak dále vyladěna pro potřeby konkrétního projektu.

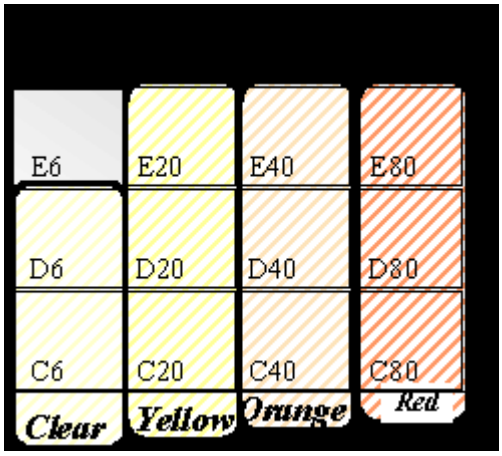
Metodika se vybírá z 3-rozměrné matice, jejíž osy tvoří zmíněné 3 parametry. Nejdůležitějšími parametry jsou *důležitost problému* a *počet lidí zapojených do projektu*. Zvětšení hodnot těchto parametrů má za následek posun k rigoróznějším variantám metodiky.



(upraveno z http://alistair.cockburn.us)

Obr. 12 – Trojrozměrná matice metodik Crystal

Metodiky pro jednotlivé rozsahy projektu jsou pro názornost pojmenovány (Obr. 13): nejjednodušší metodika je „čirá“ (*Clear*), další jsou „žlutá“ (*Yellow*), „oranžová“ (*Orange*), „červená“ (*Red*), „hnědá“ (*Brown*), „modrá“ (*Blue*), „fialová“ (*Purple*), apod. Každá „barva“ obsahuje svá vlastní pravidla a základní elementy a každá předepisuje takové množství dokumentů a dalších formálních náležitostí, kolik je pro projekty daného rozsahu přípustné.



(převzato z http://alistair.cockburn.us)

Obr. 13 – Rozdělení metodik do barevných skupin podle rozsahu projektu

Alistair Cockburn doporučuje postup, jak vytvořit metodiku „na míru“. Metodika je během projektu přizpůsobována v několika situacích:

1. **Před započetím projektu** – v této době je vhodné vést rozhovory s lidmi zainteresovanými na předchozích projektech. Zajímat se o jejich postřehy, co by bylo třeba dle jejich názoru zlepšit či dělat jinak a v čem vidí úspěchy i neúspěchy;
2. **Na začátku projektu** – v okamžiku spuštění projektu je třeba vhodně nastavit délku iterací, specifikovat požadované výstupy a mezivýstupy, organizaci práce, použité standardy, formát průběžných reportů, atd. Všechny tyto parametry metodiky (a metodika samotná) jsou zvoleny na základě úvah, zkušeností a námětů získaných v předchozím kroku;
3. **V průběhu první iterace** – přibližně uprostřed první iterace je vhodné zařadit další kolo rozhovorů s vývojáři a dalšími členy týmu. Je třeba se dovědět informace o případných problémech, se kterými se pracovníci potýkají a které by mohly mít negativní dopad na průběh projektu. Na základě těchto informací je pak možné přizpůsobit parametry procesu;
4. **Po skončení každé iterace** – v těchto pravidelných obdobích by měly být uspořádána další pravidelná setkání, na nichž by mělo být shrnuto, co se tým v uplynulé iteraci naučil, jaké problémy řešil a na co je nutné se zaměřit do příštích týdnů;
5. **V průběhu dalších iterací** – pokud jsou iterace delší než cca tři týdny, je vhodné v jejich průběhu zařazovat další pravidelná setkání, v jejichž rámci se diskutuje o průběhu projektu.

5.3.9.4 Proces vývoje pomocí metodiky

Jak bylo vysvětleno, proces vývoje pomocí konkrétní metodiky z rodiny metodik Crystal se u každé metodiky liší. Není předmětem tohoto textu představit specifika všech obsažených metodik, pro představu uvedu příklad jedné z metodik.

Příklad metodiky Crystal: oranžová

Uvedeme příklad oranžové metodiky s následujícími parametry:

- Počet členů v týmu mezi 10 až 30;
- Trvání projektu cca 1 až 2 roky;
- Důraz na dobu dodání na trh;
- Nejedná se o životně důležitý systém.

Jde tedy o metodiku pro běžnou kategorii rozsáhlejších projektů, které vzhledem k době jejich trvání vyžadují nalezení rovnováhy mezi stabilitou a měnícími se požadavky. Množství vyžadované dokumentace je spíše nižší, avšak mělo by umožňovat dostatečnou komunikaci mezi členy týmu. Metodika předpokládá určitou fluktuaci mezi zaměstnanci, proto předepisuje poměrně flexibilní přiřazování rolí.

Metodika stanovuje tyto konkrétní parametry:

- V týmu je předepsáno 14 rolí (sponzor, doménový expert, business analytik, projektový manažer, systémový architekt, poradce návrhu, návrhář, programátor, vedoucí návrhář, vedoucí programátor, tester, návrhář uživatelského prostředí, ...);
- Lidé v týmu jsou rozděleni do menších týmů odpovědných za plánování, monitoring projektu, architekturu a návrh, technologii, funkce, infrastrukturu a externí testování;
- Software je dodáván inkrementálně a pravidelně, délka iterace je stanovena na dva až čtyři měsíce;
- Pokrok je měřen prostřednictvím mezníků, která mají především podobu prototypů, méně pak dokumentací;
- Probíhá automatizované testování modulů a funkcí;
- Každou novou verzi musí posoudit alespoň dva lidé ze strany zákazníka;
- Schůzky směřující k upravení metodiky probíhají na začátku a uprostřed každé iterace;
- Mezi dodávané a vytvářené meziprodukty patří kromě jiného např. specifikace požadavků, seznam verzí, plán projektu, specifikace uživatelského rozhraní, objektový model, uživatelský manuál, zdrojový kód, testovací případy. Každý produkt je vypracován do takové úrovně podrobností, až je bez problémů pochopitelný pro ostatní členy týmu;
- Další standardy (styly psaní zdrojového kódu, detaily testování, používané šablony, apod.) jsou ponechány na vývojovém týmu; jejich stanovní proběhne v závislosti na dosavadních zvyklostech organizace;
- Konkrétní techniky a postupy používané jednotlivými programátory a dalšími členy týmu při plnění jejich povinností jsou ponechány na jejich uvážení.

Metodika říká, že všechna výše uvedená pravidla jsou povinná a závazná, avšak je povoleno jejich nahrazení pravidly z jiné metodiky.

5.3.9.5 Shrnutí

Rodina metodik Crystal se vyznačuje především svojí **konfigurovatelností** pro různé typy projektů na základě parametrů.

Metodika neobsahuje explicitní řízení rizik, ale nahrazuje je organizováním pravidelných schůzek, jejichž náplní je právě diskuse o případných problémech v projektu a jejich řešení.

Metodika Crystal je ze všech agilních metodik nejlépe škálovatelná: počítá s rozsahem projektů od několika lidí až do několika set. Z tohoto důvodu jsou některé úkony pojaty formálněji.

Metodika Crystal je oproti jiným agilním metodikám více podrobná a konkrétní ve svých doporučeních. Je tak vhodná pro méně zkušené vedoucí projektů, kteří ještě nemají vyvinut dostatečný cit pro nastavení procesu, stanovení pravidel a dokumentů.

5.3.10 Adaptive Software Development

5.3.10.1 Vznik a vývoj

Kořeny metodiky ASD sahají do roku 1992, kdy začal *Jim Highsmith* pracovat na procesu vývoje, který by splňoval základní požadavky:

- Iterativní přístup;
- Vývoj v krátkých iteracích;
- Co nejrychlejší dodání fungující aplikace.

Tyto základní myšlenky (opět v souladu s manifestem agilního vývoje) později vyústily v představení metodiky ASD [Highsmith 1999]. Na vývoji ASD pak pokračoval též Highsmithův kolega *Sam Bayer*.

Autoři v následujících letech pracovali – společně i každý zvlášť – na více než stovce softwarových projektů, při nichž používali praktiky definované v této metodice. ASD tedy bylo poměrně důkladně prakticky vyzkoušeno a ukázalo se, že při správném týmovém vedení může vést k úspěšným výsledkům i při nasazení ve velmi rozsáhlém projektu. Jako příklad bývají uváděny úspěšné projekty pro brokerské společnosti z Wall Streetu, letecké společnosti a telekomunikace.

5.3.10.2 Základní charakteristika

Autoři metodiky vyšli z předpokladu, že proces vývoje softwaru je ze svého principu **nepředvídatelný**. Neschopnost předpovědět každou událost při vývoji ovšem neznamená, že by vývoj nemohl postupovat kupředu a dosahovat pokroku. V souladu s agilním přístupem tedy změnu považují za nedílnou součást procesu a řeší, jakým způsobem se jí přizpůsobit. Nutným předpokladem je samozřejmě flexibilita členů týmu, vedení i organizace, kteří musí být schopni změnu akceptovat.

5.3.10.3 Klíčové pojmy

Změna a přizpůsobení je tedy pilířem metodiky. Tradiční statický životní cyklus je v ní nahrazen dynamickým. Mezi hlavní teze patří existence průběžného učení, opakované vyhodnocování podmínek a parametrů, intenzivní spolupráce mezi vývojáři, managementem a zákazníky.

Tradiční statický životní cyklus plánování – návrh – realizace je nahrazen iterativním cyklem **spekulace, spolupráce, učení,** (Obr. 6) které se navzájem ovlivňují. Důležité je, že v každé fázi může dojít k **odklonu** od plánu. Jednotlivé fáze se určitým způsobem překrývají a prolínají (např. nelze se učit bez spolupracování, spekulovat bez učení, atd. Vývoj v metodice ASD je totiž dynamický a nelze v jasně oddělit jednotlivé činnosti, jelikož jsou provázány a fungují jen dohromady[6].

5.3.10.4 Proces vývoje pomocí metodiky

Proces vývoje sestává z fází iterativního cyklu popsaného výše:

Fáze spekulace

Fáze spekulace nahrazuje fázi plánování. I když obsah této fáze je v podstatě shodný, název je zvolen úmyslně s ohledem na to, že od plánu v ASD je možné se odklonit (a nejspíš k tomu i dojde). Termín „plánování“ totiž evokuje cosi přesně daného a odchýlení od plánu je chápáno jako poklesek[7], zatímco „plán“ v ASD je přesným opakem.

Součástí fází spekulace jsou například následující procesy:

- Stanovení termínu ukončení projektu;
- Určení vhodného počtu iterací v závislosti na rozsahu projektu, na velikosti vývojového týmu, na požadovaných funkcích, apod.;
- Stanovení termínu končení jednotlivých iterací;
- Předběžné přiřazení komponent jednotlivým iteracím;
- Rozhodnutí o použitých technologiích, vývojových nástrojích, apod.

Fáze spolupráce

Místo návrhu a implementace hovoří ASD o spolupráci. Snaží se tím opět evokovat naladění této fáze, které by mělo probíhat v duchu vzájemné spolupráce, a to jak na úrovní týmu, tak se zákazníkem.

Metodika ASD rozděluje projekt na dvě části:

- Předvídatelnou;

[6] To je obdoba synergického působení principů v XP.

[7] Plánování resp. spekulace v ASD by se dalo nejlépe přirovnat ke známému výroku generála Eisenhovera: „Plány jsou zbytečné, nicméně plánování je nezbytné.“

- Nepředvídatelnou.

Říká, že předvídatelnou část je možné *naplánovat* a mělo by tak být učiněno *důkladně*. Nepředvídatelná část ovšem naplánovat nelze a jakékoliv snahy o to jsou kontraproduktivní. Tuto část projektu je třeba postavit na tvořivé, otevřené kreativní bázi a vzájemné důvěře za tolerance občasného nepořádku, drobných zmatků, emocionálních projevů, atd.

Zde samozřejmě vzniká nebezpečí sklouznutí projektu k řadě nešvarů známých ze špatně řízených projektů (sklony k nicnedělání, anarchie, nekoordinovanost, živelnost, atd) s velmi negativními důsledky pro projekt. Toto nebezpečí je však v případě ASD podchyceno krátkými iteracemi a včasnou a výraznou zpětnou vazbou (viz další fáze).

Náplní fáze spolupráce je vývoj jednotlivých komponent systému. V této fázi se klade silný důraz na *komunikaci* mezi členy týmu.

Fáze učení

Název třetí fáze je trochu posunut, jelikož obsahem této fáze je především **hodnocení**. Termín „učení“ je zvolen pro zdůraznění zpětné vazby, která z hodnocení vyplývá, tedy zlepšení vývojového procesu. Náplní fáze je:

1. **Hodnocení výsledků zákazníkem**. Zákazník posuzuje technické zprávy, provádí uživatelské testování betaverzí, zkoumá uživatelské rozhraní a celkově se seznamuje s hotovou prací a porovnává výsledky se svými očekáváními;
2. **Revize fungování týmu**. Fáze učení je věnována i hlubšímu rozboru fungování týmu, používaných nástrojů, úpravě plánů, standardů a pravidel, atd.

5.3.10.5 Shrnutí

Metodika ASD je nejdynamičtější metodikou z rodiny agilních metodik, přesto však zachovává procesní přístup. Jejím základním motorem je **změna** a přizpůsobení změně.

Metodika postupuje stejně jako ostatní agilní přístupy iterativně, nicméně na koncích jednotlivých iterací si neklade za cíl dodávat fungující produkt (jako např. Extrémní programování), účelem je pouze dát zákazníkovi dostatek podkladů k ověření postupu, maximálně betaverze produktu a prototypy uživatelského rozhraní. Hotový a odladěný produkt dodává až po skončení vývoje. Metodika je tedy vhodná pro zadání, které svoji podstatou není možné fragmetovat a dodávat postupně (např. bezpečnostní a řídicí systémy a systémy založené na workflow).

Metodika nedefinuje konkrétní postupy, pouze se zaměřuje na samotný adaptibilní učící proces. Konkrétní obsah procesu je třeba naplnit z jiných metodik, agilních i klasických. Je např. možné ve fázi spolupráce využívat párové programování z Extrémního programování a Scrum Meetings z metodiky SCRUM.

Metodika ASD má za sebou léta praktického používání. Oproti většině ostatních agilních metodik byla úspěšně použita i u rozsáhlých projektů.

5.3.11 Dynamic Software Development Method

5.3.11.1 Vznik a vývoj

Metodika DSDM [DSDM 2003] je dílem britského konsorcia *DSDM Consortium*, které má 16 zakládajících členů. Konsorcium bylo založeno již v roce 1984. Zakládající organizace měly společný cíl: spojit se za účelem vytvoření nezávislého prostředí pro rychlý vývoj aplikací (Rapid Application Development).

První verze byla dokončena a odsouhlasena členy konsorcia v lednu 1995 a zveřejněna o měsíc později. Verze 2 byla publikována v prosinci 1995. Následovaly další verze, které odrážely rostoucí množství zkušeností s používáním DSDM v praktických projektech. V době psaní této práce (duben 2006) je nejnovější verzí 4.2

5.3.11.2 Základní charakteristika

Proces DSDM se skládá ze sedmi iterativních fází. Základní filozofie DSDM se skládá ze čtyř dílčích pilířů (které jsou podle očekávání velmi příbuzné ostatními agilním přístupům):

- Vývoj softwaru je týmovou záležitostí. Má-li skončit úspěšně, musí v sobě kombinovat znalosti zákazníka (týkající se oboru jeho činnosti) a technické dovednosti IT profesionálů;
- Aby mohl software vykazovat vysokou jakost, musí být jednak vhodný pro zamýšlený účel a musí být technicky dokonalý;
- Vývoj může být inkrementální – není vhodné dodat všechny součásti najednou. Dodání „něčeho“ dříve je často cennější než dodání „všeho“ později;
- Zdroje musí být využívány efektivně, pouze za účelem přinést co nejvyšší hodnotu zákazníkovi.

Jak bylo zmíněno, konsorcium DSDM dodává ke své metodice **framework**, který pomáhá při použití metodiky. Framework obdrží každý, kdo se stane členem konsorcia (a zaplatí příslušný poplatek). Framework je k dispozici jednak on-line a také na CD. Framework obsahuje elektronický manuál, řadu pomocníků, průvodců, tipů a nápověd.

5.3.11.3 Klíčové pojmy

Metodika je založena na osmi principech, které dohromady tvoří výkonnou a ověřenou agilní metodiku:

1. Při vývoji je nutné aktivní zapojení zákazníka;
2. Vývojový tým musí být kompetentní přijímat důležitá rozhodnutí;
3. Důraz je kladen na časté dodávky funkčních celků;
4. Základním kritériem pro přijetí všech produktů a meziproduktů je vhodnost pro zamýšlený účel;
5. Pro vytvoření precizního, nejvhodnějšího řešení je nutný iterativní a inkrementální přístup k vývoji;
6. Veškeré změny v průběhu vývoje jsou vratné;
7. Testování je integrováno do všech fází vývojového procesu;

8. Spolupráce a součinnost všech účastníků je zcela zásadní.

Metodika definuje 11 rolí: výkonný ředitel, vizionář, uživatelův velvyslanec, uživatelský poradce, projektový manažer, technický koordinátor, týmový předák, vývojář, tester, písař a různí tradiční IT specialisté. Netradiční je možnost zvolit projektového manažera z kruhu uživatelů.

DSDM využívá řadu technik, ale počítá s tím, že každý projekt je jiný a je třeba vždy zvážit, které techniky a jakým způsobem použít. Metodika specifikuje podmínky nezbytné pro úspěšné nasazení a radí, co je důležité vzít v úvahu.

Použití technik v projektu DSDM spadá do dvou základních kategorií:

1. Projektové techniky – mohou být použity ve většině projektů bez ohledu na jejich specifickou náplň;
2. Vývojové techniky – liší se v závislosti na projektu, standardech, nástrojích, zvoleném přístupu, atd.

Metodika zavádí tzv. **Faciliated Workshops**, tedy řízená pravidelná pracovní setkání. Jedná se o obdobný princip pravidelného setkávání za účelem diskuse o projektu a zlepšování mezilidských vztahů, jako u ostatních agilních přístupů. Metodika však nestanovuje žádnou konkrétní frekvenci těchto schůzek, jejich potřebu musí posoudit členové týmu sami. Kromě týmových setkání definuje DSDM celou řadu dalších, zástupných metod, např. individuální pohovory a rozhovory, výzkum, apod.

Projektové techniky

Metodika DSDM je poměrně rozsáhlá, uveďme si zde pro příklad dvě projektové techniky, které zavádí: **Timeboxing** a **MoSCoW.**

Timeboxing je plánovací technika umožňující zachování flexibility. Vychází z *pevného data ukončení projektu.* Toto datum ohraničuje rámec (timebox) celkového objemu prací. Do této časové posloupnosti jsou vnořovány další, kratší timeboxy, jejichž trvání je typicky mezi 2 a 6 týdny.

Timebox prochází obvykle třemi fázemi:

1. **Zkoumání** (*Investigation)* **–** rychlé zhodnocení situace;
2. **Zdokonalení** (*Refinement*) – úprava řešení v závislosti na výsledcích předchozí fáze;
3. **Konsolidace** (*Consolidation*) – závěrečná fáze timeboxu sloužící k dotažení všech nepřesných, nedokončených aspektů do konce.

Každý timebox má neměnné datum ukončení a je mu přiřazena množina požadavků. Tyto požadavky jsou seřazeny podle jejich priorit, které jsou stanoveny použitím techniky **MoSCoW** popsané dále. V rámci každého fixního timeboxu jsou tedy přednostně realizovány požadavky s nejvyšší prioritou a množství zpracovaných požadavků záleží na času spotřebovaném každým z nich.

V názvu techniky techniky **MoSCoW** (angl. Moskva) mají význam velká písmena, která označují pět priorit:

- **M**ust – musí být;
- **S**hould – mělo by být;
- **C**ould – mohlo by být;
- **W**on’t – nebude.

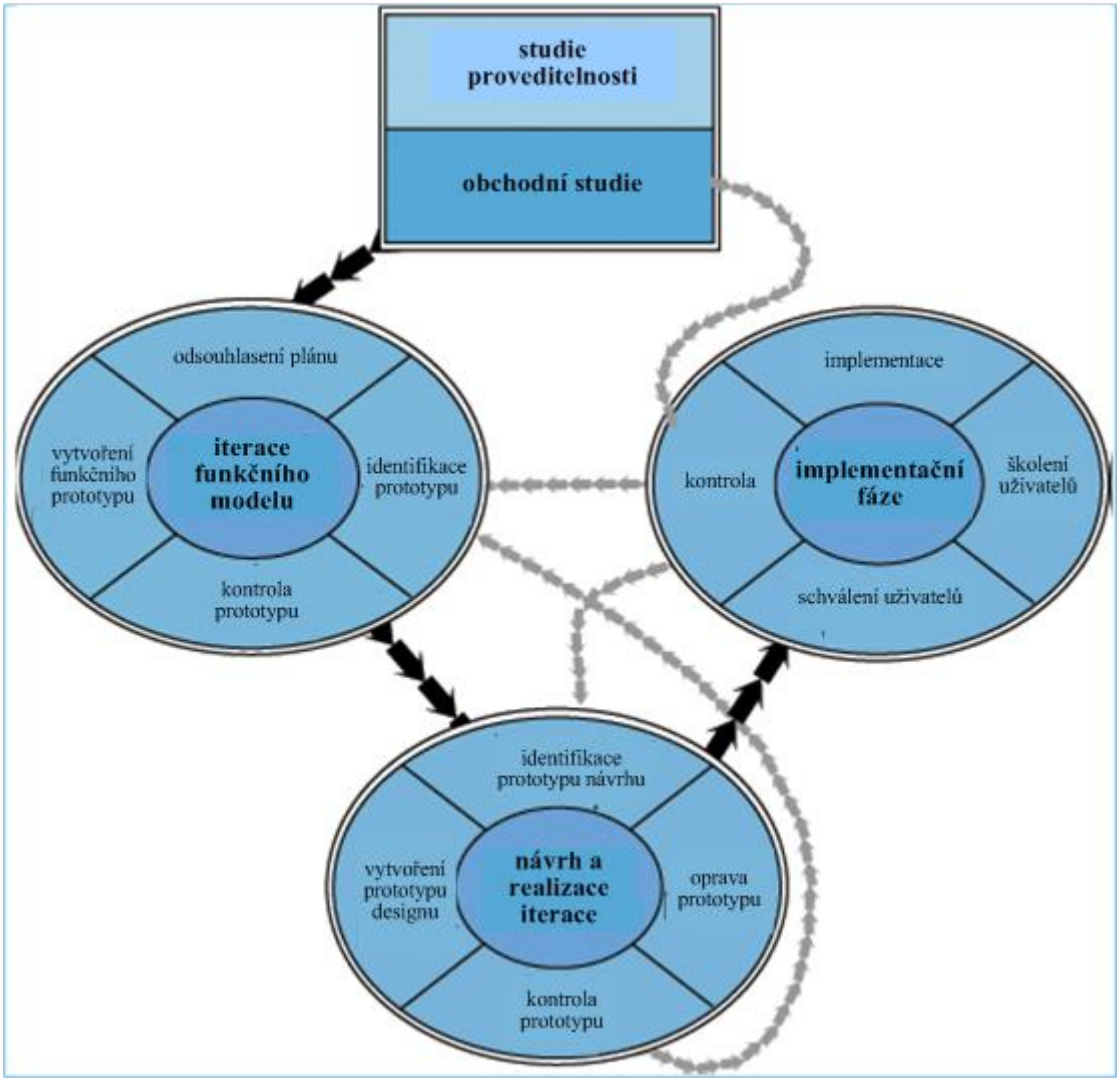
Zbylá písmena „o“ byla přidána pro lepší zapamatovatelnost označení.

Nastavení priorit je třeba udělat vyváženě, aby jednak priority odpovídaly skutečné důležitosti a kategorie Must nebyla přeplněna, jelikož pak by nezbýval prostor pro variabilní náplň timeboxů.

5.3.11.4 Proces vývoje pomocí metodiky

Metodika definuje sedm fází prováděných zajímavým iterativním postupem. Tyto fáze jsou:

1. **Úvod projektu –** slouží k nastavení správné konfigurace projektu a k ujištění, že potřebné procesy projektu jsou správně nastartovány;
2. **Studie proveditelnosti –** posouzení, zda DSDM je tou správnou metodikou pro příslušný projekt, definice problému, posouzení nákladů a technické proveditelnosti daného projektu;
3. **Obchodní studie –** hlavní důraz je kladen na „obchodní“ procesy v doménové oblasti a na analýzu jejich informačních potřeb. V rámci této fáze probíhají častá setkání, typicky formou Faciliated Workshops;
4. **Stanovení modelu funkcí –** iterativní fáze sloužící k vytvoření a zdokonalení obchodního modelu aplikace (tj. stanovení funkcí, které mají být podporovány a jak);
5. **Návrh –** iterativní fáze sloužící k navržení a vyladění jeho architektury;
6. **Implementace –** iterativní fáze sloužící k samotnému vytvoření systému;
7. **Závěr projektu –** nasazení projektu a jeho správa.



upraveno z http://www.lshift.net)

Obr. 14 – Proces DSDM

Náplň tří iterativních fází (Modelování, Návrh, Implementace) je opět prováděna iterativně a je vidět z celkového schématu vývojového cyklu na Obr. 14. Hlavními fázemi se vývojový tým nemusí pohybovat jen směrem kupředu, ale i zpět. Pokud v kterémkoliv kroku označeném jako Revize zjistí nesrovnalost, nedůslednost, případně se objeví nové požadavky na funkce či doporučení na architekturu, vývoj se přesune zpět do té fáze, v níž je možné provést potřebnou nápravu nebo rozšíření.

5.3.11.5 Shrnutí

Za vývojem metodiky DSDM stojí celé konsorcium šestnácti organizací. Metodika je zajímavá tím, že autoři k ní dodávají i **podpůrné vývojové prostředí (framework)**.

Ačkoliv bývá metodika DSDM řazena mezi agilní, svým charakterem se nalézá na pomezí mezi rigorózními a agilními metodikami, není tedy vhodná pro malé dynamické projekty. Její výhodou je značná propracovanost a fakt, že je intenzivně vyvíjena již bezmála 20 let. Během vývoje do ní byly zapracovány moderní trendy a zkušenosti z vývoje. Metodika obsahuje řadu technik spolu s doporučeními o jejich použitelnosti a zacílení, čímž se blíží detailnosti UP.

Metodika se vyznačuje propracovaným vývojovým cyklem, který probíhá iterativně jak na úrovni hlavních fází, tak uvnitř fází. Mezi hlavními fázemi je možné se i vracet, takže práce jsou vždy směrovány do místa, které je v dané chvíli klíčové.

Nevýhodou je, že metodika není zcela „zdarma“ – je třeba zaplatit členský příspěvek konsorciu. V ceně příspěvku je i podpůrný vývojový framework. Výše členských poplatků se liší podle charakteru členství (akademický, vláda, firemní, ap.) a pohybuje se od několika desítek liber do tisíců liber.

5.3.12 Shrnutí a porovnání agilních metodik

Agilní metodologie se zaměřuje na projekty z oblasti businessu a služeb, kde je nutná značná flexibilita vývojového procesu. Vychází z myšlenek tzv. Manifestu agilního vývoje softwaru, který zastřešuje metodiky s podobnou filosofií. Řada metodik zařazených mezi agilní však vznikla ještě před vznikem manifestu (2001) a byla již v praxi úspěšně používána.

Nespornou společnou výhodou agilního přístupu je, že zákazník má v průběhu vývoje mnohem větší kontrolu a přehled o skutečném postupu projektu (u některých metodik dokonce průběžně dostává kusy hotového systému k používání), což umožňuje zavést účinnou zpětnou vazbu, která je klíčem k produktu splňujícímu (měnící se) požadavky zákazníka.

Hlavním omezením většiny agilních přístupů je velikost projektů (a tím i týmů) a nutnost vhodného složení týmu z hlediska motivace a osobního přístupu. Obecně platí, že čím je projekt rozsáhlejší a čím větší jsou nároky na formální stránku, tím je použití agilních přístupů méně vhodné [Pergl 2004a]. Agilní metodiky také kladou větší nároky na role vedoucích a koordinátorů projektů, jelikož obvykle nedefinují přesné činnosti, jejich náplň a sled, nýbrž předpokládají v tomto ohledu určitou erudici a vlastní vyvinutý cit a schopnosti.

Nejznámější metodikou je Extrémní programování (kap. 5.3.4), které je propracované a existuje k němu dostatek literatury. Nicméně některé jeho praktiky nemusí být snadné přijmout (např. kolektivní vlastnictví kódu). Metodika SCRUM je v některých ohledech opatrnější (kap. 5.3.5).

Některé metodiky jsou více orientovány na různá pravidla a doporučení (Extrémní programování), jiné se více zaměřují na proces – především metodika Lean Development (kap. 5.3.6), která se snaží proces vývoje maximálně zefektivnit pro snížení nákladů a zrychlení vývoje.

Rozdíly lze mezi metodikami nalézt i z hlediska flexibility na změny požadavků v průběhu vývoje. Nejflexibilnější je Adaptive Software Development (kap. 5.3.10), která je schopna upravovat i samotný vývojový cyklus, oproti tomu Feature Driven Development (kap. 5.3.7) je zaměřena více formálněji a disponuje tak menší flexibilitou.

Jedním ze společných rysů agilních metodik je nutnost jejich přizpůsobení konkrétním podmínkám. Nejvýraznější je tento rys u metodiky Crystal (kap. 5.3.9), což je vlastně rodina metodik přizpůsobených pro různé parametry projektů.

Rozdíly jsou i ve vyzrálosti metodik – zatímco některé metodiky vznikly až v posledních letech (např. Agile Unified proces, kap. 5.3.8), jiné mají za sebou již řadu let praktického používání (Dynamic Software Development Method, kap. 5.3.11). Lean Development dokonce vychází z principů užívaných v Japonském průmyslu již několik desítek let.

Co se týká rozsahu pokrytí projektu, agilní metodiky se obvykle zaměřují nejvíce na fázi analýzy a vývoje, údržba je díky jejich flexibilitě přirozenou součástí. Největší šíři má Agile Unified Process, ale to díky tomu, že je postaven na propracovaném Rational Unified Process. Na druhou stranu, mezi agilními přístupy lze nalézt i dílčí praktiky, které je možné používat v klasickém vývojovém procesu – např. Agilní modelování a Test Driven Development.

Pro úspěšné používání agilních metodik velmi záleží na složení týmu: profesní vyzrálosti, přístupu jednotlivých členů práci, jejich osobností, vztahů mezi členy týmu, apod. Toto je jedním z klíčových aspektů, kterým je podmíněn úspěch projektu vedeného agilně.

5.4 Snahy o formalizaci v disciplíně softwarového inženýrství

5.4.1 Úvod

Při zpracování rešerše jsem se snažil nalézt vhodný základ pro zamýšlenou formalizaci softwarového projektu. Bohužel žádná z analyzovaných metodik řízení softwarových projektů není postavena na formalismu, což odráží cíl, s jakým byly metodiky vytvářeny, totiž poskytnout především praktický "návod", jak co nejúspěšněji vést softwarový projekt.

Lze však nalézt snahy formalizovat určité oblasti softwarového inženýrství, a jim bude věnována tato kapitola. Stručně představím, k čemu konkrétní formalizace slouží, na čem je založena a jakým způsobem souvisí s cílem práce.

5.4.2 Jakost v softwarovém inženýrství

Snahy o formalizaci se objevují při tvorbě *norem* zabývajících se jakostí, kde hlavním formalismem jsou **atributy jakosti** a jejich **míry** [Vaníček 2004]. Existují dvě hlavní oblasti jakosti, jež se přímo dotýkají tématu práce, a to

- Jakost procesu;
- Jakost produktu.

Jakostí procesu se zabývá dnes již široce uznávaná řada norem ISO 9000 [ISO 9000]. Z hlediska softwarového inženýrství však narážíme na problém, že tyto normy jsou zcela obecné a týkají se jak managementu jakosti procesu výroby sirek, tak i lokomotiv. Z tohoto důvodu zde nenalezneme konkrétní míry ani jiný vhodný formalismus. Neexistuje norma na management jakosti procesu vývoje softwarového produktu, existuje jen obecný předpis jak by měl vypadat [ISO 90003].

Normy týkající se jakosti softwarových produktů (např. současná ISO/IEC 14598) obsahují sice části týkající se plánování, řízení a vývoje, jedná se však o obecné postupy, které nejsou vytvořeny pro žádnou konkrétní metodiku. Některé míry jakosti produktu přináší řada ISO 9126-1 až 4 [ISO 9126-1], [ISO 9126-2], [ISO 9126-3] [ISO 9126-4]. Tato řada norem bude v budoucnosti nahrazena připravovanou řadou norem SQUARE. Aktuální situace v době dokončování práce je taková, že zhruba třetina těchto norem je již schválena (jedná se především o obecné zastřešovací

normy), třetina je v pokročilém stadiu přípravy a třetina detailních norem zatím není připravena (podrobněji viz [Vaníček 2007]).

Problematika jakosti je blízká tématu práce svým obecným přístupem nezávislým na metodice a charakteru projektu, nicméně jedná se o doplňující, byť velmi důležitou oblast.

5.4.3 Atributy a míry

Atributy a jejich míry jsou používány pro formální hodnocení jakosti, a to jakosti produktu i procesu zmíněných výše. Platné normy nabízejí některé míry, avšak dosud nebyl standardizován žádný ucelený soubor měr jakosti produktu ani procesu.

Míry procesu vývoje softwarového produktu se pokoušejí kvantifikovat proces vývoje a poskytnout tak exaktnější informaci o aktuálním stavu projektu a jeho částí včetně vývoje v čase.

Nejčastější oblasti sběru a využití měr jsou:

- Celkové informace o postupu projektu jako informace pro vedení projektu;
- Údaje týkající se jakosti jako podklad pro řízení jakosti;
- Údaje o výkonnosti členů týmu jako podklad pro hodnocení;
- Údaje týkající se organizace práce jako podklad pro zvýšení efektivity;

Množina měr vybraných a použitých v konkrétním projektu je determinována potřebami projektového vedení. Z hlediska maximální kontroly nad průběhem projektu a jeho stavem je samozřejmě ideální zavést co nejvíce měr, nicméně je třeba si uvědomit, že sledování a vyhodnocování každé míry přináší určitou režii.

Řadu měr týkajících se všech vývojových fází projektu lze nalézt např. v [Ambler 1998] a [Ambler 1999]. Často uváděnou kritikou agilních metodik je absence měr. Tato absence však není dána charakterem metodik, ale spíše nedostatkem pozornosti této oblasti. Nicméně i zde výzkum pokračuje, jako příklad můžeme uvést [Vrana, Mahnič 2007], jež přináší míry pro projekty řízené metodikou Scrum (kap. 5.3.5). V Tab. 2 je několik příkladů těchto měr a indikátorů, pro které jsou potřeba. Pro vysvětlení připomínám, že metodika Scrum jednotlivé vývojové iterace nazývá sprinty a úkoly příslušející dané iteraci jsou udržovány ve formě tzv. sprint backlogu.

Indikátor	Míry
Poměr hotové a zbývající práce	Práce strávená dne *i* pro každý úkol ze sprint backlogu - WS_{ij} Práce zbývající dne *i* pro každý úkol *j* ze sprint backlogu - WR_{ij}
Index plnění plánu	Práce zbývající dne *i* pro každý úkol *j* ze sprint backlogu - WR_{ij} Práce strávená dne *i* pro každý úkol ze sprint backlogu - WS_{ij} Délka sprintu (počet dní) - *SL* Počet dní uběhlých od začátku sprintu *DE*
Index plnění nákladů	Práce zbývající dne *i* pro každý úkol *j* ze sprint backlogu - WR_{ij} Práce strávená dne *i* pro každý úkol ze sprint backlogu - WS_{ij} Počáteční odhad nákladů na sprint - *SLC* Cena hodiny práce člena týmu (pro každý úkol *j* ze sprint backlogu) - CEH_j Délka sprintu (počet dní) - *SL* Počet dní uběhlých od začátku sprintu *DE*
Hustota chyb (počet chyb na tisíc řádků zdrojového kódu)	Počet chyb nalezených během meetingu hodnotícím sprint Počet chyb hlášených uživateli v pevném časovém období po nasazení Velikost kódu
Průměrný počet projektů, na kterých pracovníci pracují paralelně.	Velikost týmu *t* (počet členů) - NTM_t Celkový počet vývojářů – *ND*

Tab. 2 – Příklady měr procesu vývoje pro metodiku Scrum

Míry procesu vývoje softwarového produktu představují důležitý faktor kontrolovatelného a dokumentovatelného vedení softwarového projektu, nicméně nejedná se o formalizaci struktury projektu a řízení.

5.4.4 Odhady složitosti softwarového projektu

Další oblastí, jež je třeba zmínit v souvislosti se snahami o formalizaci je odhad složitosti softwarového projektu a potažmo časových a finančních nároků. Vzhledem ke složitosti problematiky není však dosud znám postup, jak na počátku projektu přesně odhadnout jeho složitost, a máme k dispozici pouze empirické vzorce zohledňující nejdůležitější faktory. Odhady jsou tak většinou značně nepřesné a záleží do značné míry na zkušenosti a intuici při nastavování různých koeficientů.

Nejznámějším odhadem je tak zvaný **odhad COCOMO** (COnstructive COst MOdel), jež je schopen poskytnout předpokládanou pracnost projektu v člověkoměsících na základě počtu zdrojových řádek kódu. Původní model byl dále rozšířen o faktory ovlivňující vývoj, jako např. požadovaná spolehlivost, stabilita vývojového prostředí, apod.

Největším problémem při praktickém využití COCOMO je nutnost znát počet řádků budoucího zdrojového kódu, což je údaj, jež není na počátku snadné odhadnout. Z tohoto hlediska jsou výhodnější metody odhad pracnosti založené na modelu. Zde můžeme jmenovat metodu **funkčních jednic (Function Points)**, jež umožňuje vypočítat odhad pracnosti na základě odhadu:

- Počtů vstupů do systému;
- Výstupů;
- Počtu dotazů;
- Interních logických souborů;
- Externích logických souborů.

Tato metoda však nebere v potaz algoritmy. Pokud ve složitosti budoucího softwarového produktu hrají algoritmy klíčovou roli, je vhodnější použít metodu **jednic vlastností (Feature Points)**, jež rozšiřuje metodu funkčních jednic právě o parametr algoritmu.

Největší úrovně abstrakce poté dosahuje metoda **jednic případů užití (Use Case Points).** Je orientována na objektový přístup k návrhu softwaru a základním vstupem je počet aktorů a počet případů užití (use case). Z nich dostaneme tzv. nevyrovnanou část odhadu, jež je násobena technickým faktorem a faktorem prostředí.

Nejprve zjistíme počet aktorů a počet use case.

V oblasti odhadů složitosti softwarových projektů bylo učiněno velké množství práce a existují i vztahy pro vliv napjatých termínů na náklady vývoje: Putnamův konzervační zákon a odhad SLIM. Dalším experimentálním zákonem tohoto typu je určení hranice, pod kterou již není možné zkrácení termínu požadovat ani v případě, že bychom se smířili se zvýšenými náklady na vývoj.

Důležitou oblastí je též trojice faktorů:

- Pracnost;
- Doba potřebná na realizaci;
- Velikost řešitelského kolektivu.

Tyto faktory spolu vzájemně souvisejí způsobem, který není právě jednoduchý. Skutečnost je navíc komplikována dynamickým charakterem pracnosti, kterým se zabývá Nordenův – Rayleighův model a Boehmova rovnice pro dynamiku vývoje.

Podrobnosti ke zmíněným modelům lze nalézt v [Král, 1996].

Odhady složitosti softwarového projektu jsou založeny na stejné myšlence, jako deklarovaný cíl práce, totiž vytvořit formální model softwarového projektu (v tomto případě matematický) a pomocí něj zvýšit poznání (v tomto případě odhad složitosti) a poskytnout praktickou pomůcku (odhad finanční a časové náročnosti). Model je však velmi

zjednodušený a omezuje se na vymezení faktorů, jež autoři považují za klíčové a matematické vztahy, jež dávají do souvislosti tyto faktory vzhledem ke složitosti vývoje. Různé metody se poté liší pouze množinou faktorů a podobou, množstvím a charakterem matematických vztahů mezi nimi. Takovýto model je plně dostačující pro účely odhadu složitosti, nicméně nenahlíží na projekt celistvým pohledem a nelze jej tedy považovat za zastřešující model.

5.4.5 Evaluace a výběr metodiky

Práce Davida Hecksela [Hecksel 2007] přináší zajímavou metodu evaluace a výběru vhodné metodiky podle určitých parametrů. Metoda je založena na objektovém přístupu [Vaníček et al. 2007a]. Hlavním objektem je projekt a dále definuje tzv. kontext projektu, který se skládá ze tří komponent:

- Lidí;
- Procesu;
- Technologií.

Každý objekt (tj. projekt a všechny 3 komponenty jeho kontextu) má množinu atributů, jejichž hodnoty spadají do nějaké ordinální stupnice (např. míra flexibility procesu je ordinální stupnice 1-7) či nominální stupnice (nejvíce flexibilní parametr procesu je šíře zadání/rozpočet/čas).

Pro metodiky jsou poté definována tzv. pravidla, jež stanovují, jaká jsou omezení hodnot atributů projektů, jež je možné danou metodikou řídit. Příkladem může být pravidlo, které říká, že minimální počet vývojářů je 2 a maximální 10.

Vhodnost metodiky pro daný projekt je vyhodnocována na základě tzv. matice kompatibility, kde jsou porovnány hodnoty atributů konkrétního projektu a pravidel metodiky. Na základě vah pro jednotlivé atributy je poté vyhodnoceno skóre vhodnosti metodiky.

Práce Davida Hecksela je zajímavá svým objektovým modelem, jež přináší již určitý formální model softwarového projektu. Model umožňuje detailní kvantifikaci softwarového projektu a jeho kontextu za účelem vyhodnocení vhodnosti metodiky. Z hlediska obecného formálního přístupu k softwarovému projektu však model není schopen postihnout vazby atributů.

6 Strukturovaný přehled procesu vývoje softwaru

6.1 Úvod

Po rešeršní části nyní navazuje popis vlastního řešení, jemuž je věnován zbytek textu.

Metodiky představené v předchozích kapitolách se vždy zaměřují na určitou fázi vývoje a často některé fáze zcela opomíjejí (typicky správu a podporu). Bylo tedy třeba začít práci poskládáním procesu vývoje softwaru v celé šíři. Jako jádro jsem použil proces definovaný Amblerem v [Ambler 1998], do něhož jsem zařadil osvědčené publikované postupy a metody a praktiky z metodik představených v rešeršní části.

Proces v celé šíři bude přítomen pouze u složitých a rozsáhlých projektů, běžné projekty budou některé fáze mít zjednodušené či budou zcela chybět. Hlavní fáze jsou:

1. Příprava (Initiate);
2. Vývoj (Construct);
3. Nasazení (Deliver);
4. Správa a podpora (Maintain and Support).

Obsah fází bude upřesněn v následujících podkapitolách.

Dle charakteru projektu mají jednotlivé fáze různou míru důležitosti a vyžadují rozdílné časové i finanční zdroje:

- **Příprava:** důkladná příprava je nutná především u rozsáhlých projektů se složitou infrastrukturou. Rovněž tak u projektů, kde jsou očekávána zvýšená rizika. Investice do přípravné fáze se však musí vyplatit, jinak nemá cenu se do projektu pouštět;
- **Vývoj:** fáze vývoje relativně převažuje ostatní fáze u malých až drobných projektů, kde ostatní fáze je možné minimalizovat;
- **Nasazení:** nasazení je třeba věnovat zvýšenou pozornost v případě, kdy podmínky provozování jsou nestandardní či komplikované, ať už z hlediska technického (např. extrémní podmínky provozu), procesního (začlenění do procesu firmy) či uživatelského (např. uživatelé mají minimální praxi s výpočetní technikou);
- **Správa a podpora:** míra důrazu na správu a podporu záleží především na:

  Míře využívání aplikace: příležitostná utilita nebude vyžadovat intenzivní správu a podporu;

  - *Kritičnosti aplikace*: kritické aplikace, např. typu řízení leteckého provozu, vyžadují perfektně nastavené mechanismy správy a podpory;

- *Předpokládané délce života aplikace*: správa a podpora vyžaduje zdroje. Pokud očekáváme, že aplikace bude mít omezenou dobu používání, je neekonomické investovat prostředky do rozsáhlé správy a nových verzí.

Náročnost této fáze také hodně závisí na kvalitě fází předchozích.

Ve složitějších případech dochází k tomu, že jednotlivé fáze se navzájem ovlivňují a prolínají.

6.2 Příprava

Ve fázi přípravy jsou prováděny tyto činnosti:

- **Vymezení a ověření počátečních požadavků**. Různé metodiky mají svůj specifický přístup ke shromáždění, dokumentaci a ověření uživatelských požadavků. Přístupy se liší především důkladností, mírou podrobnosti a formalismu. Některé příklady struktury uživatelských požadavků (detaily lze nalézt v literatuře):
  - Kontextový diagram (context diagram) [Rumbaugh et al. 1998];
  - Případy užití (use-cases) [Rumbaugh et al. 1998];
  - CRC modelování [Ambler 1998];
  - User stories (metodika Extrémní programování) [Beck 2002a];
  - Product backlog (metodika SCRUM) [Schwaber et al. 2001];
  - Feature (metodika Feature Driven Development) [Buchalcevová 2005];

  Pro získání uživatelských požadavků jsou používány různé techniky, např.
  - Interview;
  - JAD Sessions: zkratka Joint Application Development. Označuje koordinovaný meeting, kdy si uživatelé a vývojáři navzájem ujasňují požadavky pomocí společného modelování;
  - prototypování uživatelského rozhraní.

  Podrobněji o problematice uživatelských požadavků viz [Kano 1984].
- **Klasifikace počátečních požadavků do tříd podle různých kritérií a stanovení jejich priorit.** Požadavky je třeba klasifikovat (viz např. [FURPS 2007]) a pokud není možné z důvodu omezených časových a finančních zdrojů uspokojit naráz všechny požadavky, je třeba, aby byly přiřazeny k požadavkům priority. Priority určuje primárně zadavatel ve spolupráci s řešitelem, jelikož mezi požadavky mohou být technické a organizační závislosti. Některé metodiky mají systémy prioritizace požadavků, které jsou obecně založeny na ordinálních měřicích stupnicích:
  - Extrémní programování: očíslování User stories (přidělení priorit kartičkám se zadáním);
  - Dynamic Software Development Method: technika MoSCoW (Must-Could-Should-Won't).

  Existují i samostatné techniky pro prioritizování požadavků, např. Kanova analýza [Kano, 1984].

- **Posouzení projektu**. Ne každý projekt je rozumné realizovat. Některé projekty mohou být ekonomicky ztrátové a mohou přinést značné problémy řešiteli i zákazníkovi. Statistiky úspěšnosti softwarových projektů (např. Standish Group) ukazují, že takové projekty nejsou výjimečné. Proto je zvláště u rozsáhlejších projektů vždy třeba posoudit, jestli je vhodné projekt realizovat. K tomu slouží především **studie proveditelnosti** (Feasibility Study) [Keyes 2002]. Skládá se z:
    - Identifikace alternativ realizace;
    - Studie operační proveditelnosti;
    - Studie ekonomické proveditelnosti;
    - Studie technické proveditelnosti;
    - Výběru alternativy či zamítnutí projektu.
- **Vymezení typů a obsahu dokumentace projektu**. Je třeba vymezit obsah projektové dokumentace a zvolit vhodnou formu. Jedná se především o:
    - Plán projektu: rozsah, úkoly, odhady, atp.;
    - Harmonogram projektu;
    - Dokumentace řízení rizik;
    - Plán zajištění kvality (Quality Assurance);
    - Dokumentace vývojového procesu;
    - Manažerské statistiky a reporty.

  Dokumentace se obecně dělí na dvě kategorie:
    - *Výstupní* (deliverables): je předávána v průběhu projektu (např. průběžné zprávy) či na jeho konci (např. uživatelská dokumentace);
    - *Interní*: slouží jako „paměť projektu“ pro potřeby řízení i pozdějšího vyhodnocování projektu.

  Těžší metodiky typicky předepisují více dokumentace a kladou důraz na formální stránku. Lehké metodiky jdou cestou účelnosti a ponechávají volnost ve výběru množství a struktury dokumentace [Poppendieck 2003].
- **Vymezení infrastruktury projektu**. Před zahájením projektu je třeba vytvořit potřebnou infrastrukturu. Ta se bude velmi lišit především v závislosti na rozsahu projektu. Obecně do infrastruktury patří**:**
    - Ustanovení projektového týmu;
    - Subcontracting;
    - Vymezení a přizpůsobení procesu řízení a metod provádění činností;
    - Komunikace v projektovém týmu a se zákazníkem: na komunikaci kladou důraz agilní metodiky [Buchalcevová 2002];
    - Specifikace norem, standardů, směrnic a metodických pokynů pro tým týkajících se modelování, konvencí pojmenovávání a štábní kultury, přístupu k testování, designu uživatelského rozhraní, atd.;

- Výběr nástrojů: nástroje pro psaní kódu, sdílení kódu, tvorbu dokumentace, vzájemnou komunikaci, atd..

6.3 Vývoj

Do fáze vývoje se vstupuje po kladném rozhodnutí o realizaci, ve chvíli, kdy je jasné:

- Co se bude vyvíjet;
- Plán vývoje;
- Co bude k vývoji třeba a jak bude organizován;

Ve fázi vývoje vzniká vlastní produkt. V nejjednodušších případech vývoj sestává z programování, obecně však tato fáze může být značně komplexní a programování v ní může hrát i méně kritickou úlohu, jelikož se do značné míry jedná o mechanický postup.

Vývoj může dle typu vývojového cyklu (software development life cycle) probíhat jako sekvenční či iterativní proces [Ambler 1998].

Vývoj sestává z těchto navzájem úzce provázaných základních činností:

- Analýza;
- Návrh;
- Implementace;
- Testování.

K tomu navíc probíhá řada podpůrných procesů, jako

- Správa verzí;
- Řízení týmu;
- Dokumentace;
- Řízení znovupoužitelnosti.

Podle charakteru metodiky jsou jednotlivé činnosti prováděny s různou granularitou. Tradiční, těžké metodiky provádějí důkladnou komplexní analýzu, návrh, implementaci a poté hotový produkt testují. Agilní přístupy oproti tomu nastavují co nejkratší trvání těchto činností a co nejvíce iterací. Důvodem jsou kalkulace se změnami zadání, minimalizace nepochopení zadání a eliminace dalších rizik.

6.3.1 Analýza

Komplexní **analýza** je prováděna tímto způsobem:

1. Modelování;

2. Identifikace funkčních celků;
3. Rozhodnutí o znovupoužití/nákupu/vývoji celku.

Z hlediska efektivity vývoje a ušetření nákladů je ideální:

1. Použít co nejvíce modulů, které jsou hotovy (znovupoužitelnost);
2. Získat či nakoupit moduly, které jsou k dispozici, a to na co nejvyšší úrovni, tedy postupně:
    - Hotová aplikace k upravení (která je k dispozici např. ve formě open-source);
    - Framework;
    - Moduly;
    - Rutiny.
3. zbytek vyvinout vlastními silami.

Vždy je ale samozřejmě třeba zvážit, zda integrace a úprava určitého hotového prvku nebude představovat větší rizika než vlastní vývoj. Rizika je třeba hodnotit komplexně, tedy

- Pracnost úpravy a integrace;
- Technická rizika (kompatibilita);
- Spolehlivost dodavatele hotového řešení a podpora.

Modelování dělíme na:

- *Podnikové modelování*, kdy modelujeme širší kontext v podniku. Příkladem metodiky pro podnikové modelování je BORM [Merunka et al. 2003] či ARIS;
- *Doménové modelování* modelující problémovou doménu, pro kterou je produkt používán;
- *Modelování architektury* produktu jako celku;
- *Detailní implementační modelování* částí produktu.

Modelování je komplexní problematika, které je věnována řada literatury a diskusí. Existuje řada metodik, které zahrnují jeden či více typů modelování zmíněných výše. Z hlediska notace je v současné době nepsaným průmyslovým standardem objektově orientované UML [Rumbaugh et al. 1998], které je používáno v rámci metodiky Unified Process i agilních metodik.

Jednotlivé metodiky poté doporučují různý přístup k modelování s tím, že obecně opět platí, že těžší metodiky kladou větší důraz na modelování a jeho formální stránku, zatímco lehčí metodiky ponechávají modelování pouze jako doporučený nástroj pro analýzu [Pergl, Struska 2006]. Výjimkou je metodika Feature Driven Development [Buchalcevová 2005], která je řízena modelováním, i přes to, že ji řadíme mezi agilní metodiky.

Za zmínku ještě stojí metoda agilního modelování vyvinutá Scottem Amblerem [Ambler 2005], která uvádí pravidla efektivního modelování použitelná v rámci obecné metodiky.

6.3.2 Implementace

Další činností ve fázi vývoje je **implementace**, která se v širším chápání skládá z následujících činností:

- Návrh grafického designu pro program, dokumentaci, krabici, atd.;
- Návrhu funkčnosti uživatelského rozhraní;
- Tvorba plánu migrace starých dat do nové aplikace a vývoj potřebných utilit;
- Psaní kódu realizujícího vlastní algoritmus;
- Psaní dokumentace;
- Integrace hotových komponent, a to:
    - Interně vytvořených v rámci předchozích projektů (znovupoužitelnost);
    - Koupených;
    - Interní dokumentace postupu prací;
    - Kontextová nápověda aplikace;
    - Příručka uživatele a další dokumenty pro uživatele (FAQ, řešení problémů, atp.).

Tyto činnosti nemusí nutně po sobě navazovat v tomto pořadí, typické je iterativní provádění.

6.3.3 Testování

Testování bývá v tradičních metodikách prováděno po programování, v současné době se však prosazuje tzv. *test-driven development* přístup, který se vyznačuje následujícími charakteristikami:

- Testovány jsou co nejmenší celky (jednotlivé metody);
- Testy jsou psány před implementací, což umožňuje ujasnit si, co bude testovaná část přesně dělat;
- Testy jsou psány, spouštěny a vyhodnocovány s pomocí podpůrných nástrojů;

Test-driven development je přímo součástí některých metodik, např. Extrémního programování.

Metodika **Full Life Cycle Object-Oriented Testing (FLOOT)** zavádí testování do všech fází životního cyklu (kategorizováno dle činností):

- *Testování požadavků* (Requirements testing):
    - Testování scénářů užití (Use-Case scenario testing);
    - Procházení prototypů (Prototype walkthroughs);
    - Revize uživatelských požadavků (User-requirement reviews);
- *Testování analýzy* (Analysis testing):
    - Procházení modelu (Model walkthroughs);

    - Testování scénářů užití (Use-Case scenario testing);
    - Vzájemné revize mezi členy týmu (Peer reviews);
    - Procházení prototypů (Prototype walkthroughs);
- *Testování návrhu* (Design testing):
    - Procházení modelu (Model walkthroughs);
    - Vzájemné revize mezi členy týmu (Peer reviews);
    - Procházení prototypů (Prototype walkthroughs);
- *Testování kódu* (Code testing):
    - „Black-box“ testování (Black-box testing): přístup, kdy testujeme pouze, jestli výstup odpovídá zadanému vstupu;
    - Testování okrajových hodnot (Boundary value testing): testování neobvyklých nebo extrémních situací, které by prvek měl zvládnout;
    - Testování integrity tříd (Class-integration testing);
    - Testování tříd (Class testing);
    - Revize kódu (Code reviews);
    - Testování pokrytí (Coverage testing): ověření, že všechny řádky kódu jsou pokryty testy;
    - Regresní testování dědičnosti (Inheritance-regression testing): aplikace testů nadtřídy na podtřídy;
    - Testování metod (Method testing);
    - Testování cest (Path testing): ověření, že důležité logické cesty v kódu jsou pokryty testy;
    - „White-box“ testování (White-box testing): ověření, že určité řádky kódu pracují tak, jak je očekáváno;

6.4 Nasazení

K fázi nasazení se přistupuje v okamžiku, kdy je celý systém k dodání (resp. část systému k dodání, pokud bude systém dodáván inkrementálně) připraven k nasazení. V nejjednodušším případě je systém pouze předán k používání, obecně však tato fáze sestává z řady navazujících i paralelně probíhajících činností.

6.4.1 Testování ve velkém

Před nasazením je třeba hotový systém podrobit komplexním systémovým testům a uživatelskému testování. Metodika FLOOT dělí testy na dvě kategorie: testování systému a uživatelské testování. Jednotlivé testy jsou:

- *Testování systému* (System testing):

    - Funkční testování (Function testing);
    - Testování instalace (Installation testing);
    - Zátěžové testování (Stress testing): testování průchodnosti a výkonu systému;
    - Operační testování (Operations testing): testování zajištění údržby a provozu systému
    - Testování podpory (Support testing): testování zajištění uživatelské podpory;
- *Uživatelské testování* (User testing):
    - Alfa testování (Alpha testing): uživatelé dostanou možnost testovat rané (nestabilní) verze produktu a vyjdádřit se k nim;
    - Beta testování (Beta testing): podobné alfa testování, ale produkt je již ve stabilnější fázi;
    - Pilotní testování (Pilot testing): plný provoz pro omezenou skupinu uživatelů (pilotních testerů);
    - Akceptační testování (User acceptance testing): testování shody s požadavky stanovenými ve smlouvě.

6.4.2 Přepracování (Rework)

Vady objevené při testování jsou evidovány a předávány týmu, kde jsou ověřeny a posuzeny. Ze seznamu jsou vyškrtnuty ty, které mají charakter nedorozumění či neznalosti testerů. Vady lze obecně rozdělit do dvou skupin:

- *Drobné vady*: lze je jednoduše odstranit v řádu minut či hodin a neovlivňují zbytek systému;
- *Vážné vady*: jejich odstranění si vyžádá větší zásahy do systému či jeho přepracování;

Dále následují tyto kroky:

- Prioritizace vad;
- Odstranění vad. V případě rozsáhlejších úprav je třeba postupovat podobně jako ve fázi vývoje:
    - Modelování;
    - Programování;
    - Testování v malém;
    - Aktualizace dokumentace.

V případě agilního cyklu s krátkými iteracemi je fáze přepracování částečně integrální součástí vývojového cyklu. Vady charakteru vylepšení funkčnosti, které není v danou chvíli klíčové jsou odsunuty jako „feature“ do další iterace.

6.4.3 Nasazení (Release)

Jádrem nasazení je instalace systému, v obecném případě však v této fázi je třeba zajistit řadu úkonů, které se provádí i před nasazením. V závislosti na agilitě vývojového cyklu se bude měnit objemnost úkonů – v případě dodání celého systému najednou bude objem největší, při inkrementálním release se bude jednat jen o rozšíření a aktualizace.

6.4.3.1 Příprava k předání

- *Přijetí plánů školení a tréninků uživatelů***.** Plány školení by měly být z většiny hotovy ve fázi konstrukce. Před nasazením by měly být revidovány a schváleny;
- *Akceptace uživatelské, technické a provozní dokumentace*. Dokumentace by měla být opět vytvořena v průběhu fáze konstrukce, nyní by měla být zkontrolována a schválena. Weiss v [Weiss 1991] ukazuje potřebu různých typů manuálů pro různé typy participantů:
    - Provoz:
        - Procedury zálohování;
        - Požadavky na pravidelné dávky a reporty;
        - Požadavky na extrakci dat / sdílení pro související systémy a datové sklady;
        - Instalační procedury pro případ nutnosti přeinstalace;
        - Požadavky na hardwarové zdroje vyžadované systémem;
        - Dokument popisu verzí popisující všechny komponenty systému (version description document (VDD). Obsahuje záznam všech změn od poslední verze a seznam zapracovaných SPR a SCR (viz níže).
    - Podpora:
        - Kontaktní místa vývoje a provozu;
        - Pravidla evidence činností pro hlášení chyb (software problem report, SPR) a návrhů na změny (software change request, SCR) a eskalační procedury;
        - Kontakty pro eskalační procedury;
        - Kompletní uživatelská dokumentace (viz níže);
    - Uživatelé:
        - „Snadný start“;
        - Uživatelský manuál;
        - Manuál podpory;
        - Referenční manuál.
- Finalizace migrace starých dat.

6.4.3.2 Nasazení pro provoz a podporu

- *Nastavení provozního procesu.* Jsou učiněny potřebné kroky a změny k nastavení připraveného provozního procesu. Provoz bude zajišťovat například aktualizaci číselníků, zasílání systémových zpráv, apod.;
- *Školení pracovníků provozu.* Pracovníci provozu by měli být podrobněji zaškoleni do technických detailů než běžní uživatelé;
- *Nastavení procesu podpory.* Podpora bude sloužit uživatelům i provozu k řešení problémů a otázek při používání. Podpora též bude sbírat návrhy na změny (change request, change management);
- *Školení pracovníků podpory*. Pracovníci podpory musí dostat kompletní uživatelské školení, aby rozuměli, jakých informací se dostalo uživatelům. Navíc musí dostat školení eskalačních procedur a kontakty na příslušná místa oddělení vývoje a podpory programu;

6.4.3.3 Předání uživatelům

- *Oznámení nové verze a informace o způsobu získání aplikace*;
- *Školení uživatelů*;
- *Instalace aplikace*. Pro instalaci je třeba zvolit vhodnou strategii podle konkrétní situace:
  - Pracovník podpory nainstaluje software na každý počítač;
  - Uživatelům je fyzicky či síťově distribuována instalační sada a instalaci provedou uživatelé;
  - Automatická instalace (aktualizace) softwaru;

6.4.4 Vyhodnocení projektu (Assessment)

Vyhodnocení projektu má dva hlavní cíle:

- Pro projektový tým znamená možnost poučit se ze zkušeností s vývojem aplikace;
- Poskytuje možnost objektivního ohodnocení členů projektového týmu a podporu osobního růstu.

Kromě toho je možné ve fázi vyhodnocení si uvědomit některá fakta důležitá pro budoucí projekty, např.

- Odchylky odhadů;
- Spolupráci se zákazníkem.

Vyhodnocení má dva kroky:

1. Provedení vyhodnocení;
2. Vyvození závěrů.

Provedení vyhodnocení je třeba provést na základě pohledu týmu i zákazníka a spočívá především v těchto činnostech:

- Posouzení projektových výstupů;
- Posouzení zapojení zákazníka;
- Analýza metrik;
- Vyhodnocení výkonu členů týmu.

Závěry vyhodnocení (výstupy) jsou především tyto:

- Ohodnocení členů týmu;
- Plán zlepšení softwarového procesu.

6.5 Správa a podpora

Fáze správy a podpory nastává v okamžiku začátku používání aplikace.

V této fázi jsou prováděny zejména tyto činnosti:

- Uživatelská podpora;
- Identifikace chyb a rozšíření;
- Řízení a dokumentace.

6.5.1 Uživatelská podpora

Vstupem procesu uživatelské podpory je vznesení **požadavku podpory (support request)** uživatelem. Požadavek prochází tokem podpory (support flow), což je proces, kdy problémy jsou přijaty pracovníky podpory, jsou nalezena řešení a jsou vráceny odpovědi uživateli. Tourniaire a Farrell [Tourniare, Farrell 1997] považují tok podpory za klíčový prvek uživatelské podpory a uvádí dvě základní možnosti nastavení:

1. *Front-line/back-line model*. Tento model je nazývaný též *vrstvený model*, protože pracovníci podpory jsou organizováni do dvou skupin/vrstev:
   - Rozsáhlá skupina méně zkušených pracovníků, kteří přijímají příchozí požadavky podpory a zkouší je vyřešit v krátkém časovém úseku;
   - Malá skupina zkušených pracovníků, kteří řeší složité požadavky předávané jim pracovníky první vrstvy.

   Výhodou tohoto modelu je, že umožňuje efektivní způsob využití pracovníků. Poskytuje šikovný způsob školení nových pracovníků a obsahuje přirozený příslib kariérního vzestupu. Hlavní nevýhodou je, že řešení problému může vyžadovat řetězec více pracovníků.
2. *Touch-and-hold model*. Pracovník podpory, který odpovídá na prvotní požadavek je odpovědný za jeho přímé zpracování, někdy za spolupráce s experty na daný problém. Klíčem je předání požadavku správnému pracovníkovi, typicky přes automatickou distribuci hovorů (automatic call distribution, ACD). Výhodou je

méně předávání, takže uživatel nemusí vícekrát popisovat požadavek. Další výhodou je větší možnost zvyšování kvalifikace pro pracovníky podpory. Model však má i své nevýhody: pracovníci podpory musí mít rozumnou úroveň technického vzdělání a úroveň poskytnuté služby může kolísat v závislosti na odborné úrovni konkrétního pracovníka, který požadavek vyřizuje;

Pro oba modely je možné nastavit následující proces zpracování požadavků:

- Odpověď na požadavek;
- Rozhodnutí o způsobu řešení:
  - Shromáždění informací pro rozhodnutí o prioritě: vysokou prioritu mají problémy, kdy aplikaci není možno obsluhovat, menší prioritu mají problémy typu „jak udělat“ a nejmenší prioritu mají podněty na nové funkce a rozšíření aplikace;
  - Simulace problému: pokud je třeba, pracovník podpory ve spolupráci s uživatelem se snaží zreprodukovat průběh a výsledek hlášené situace;
  - Eskalace problému: pokud si pracovník podpory neví s problémem rady, žádá o pomoc odborníka (vývojáře, manažera). K eskalacím by nemělo docházet příliš často a po každé eskalaci je vhodné zkoumat, proč k ní došlo a jakým způsobem jí v budoucnu zabránit.
- Řešení požadavku:
  - Poskytnutí informace: pokud je odpověď obsažena v uživatelské dokumentaci či jiném zdroji přístupném uživateli, odkážeme při odpovědi uživatele na příslušné místo. To vede uživatele k povědomí o těchto zdrojích za účelem odlehčení zátěže uživatelské podpory;
  - Identifikace potřeby školení: Pokud je identifikována rozsáhlejší neznalost obsluhy aplikace či přímo výpočetní techniky, může být uživateli nabídnuto či doporučeno zařazení do školení;
  - Identifikace potřeby HW/SW upgrade;
  - **Záznam problému (software problem report, SPR)** nebo **požadavku na změnu (software change request, SCR)**. Požadavky na změnu jsou shromažďovány a o jejich schválení rozhoduje výbor softwarové konfigurace (configuration control board, CCB);
- Výsledné řešení.

Uživatelé by měli být zaškoleni ve způsobu hlášení problémů a seznámeni se způsobem jejich vyřizování. Tourniaire a Farrell v [Tourniare, Farrell 1997] doporučují vytvořit dokument „Průvodce uživatelskou podporou“ (Support User’s Guide), jež popisuje proces podpory aplikace z uživatelského pohledu. Dokument má obsahovat:

- Druh poskytovaných služeb a hodiny, ve kterých jsou dostupné;
- Způsob obdržení podpory;
- Kdo je oprávněn obdržet podporu a jak často;
- Úroveň podpory (doba obrátky dle priority);
- Popis, jak jsou požadavky podpory řešeny;

- Tipy pro získání nejlepších služeb včetně postupů řešení běžných problémů, informace, které je třeba mít po ruce při volání podpory a nejlepší časy pro volání.

6.5.2 Údržba

Fáze jde ruku v ruce s fází správy a podpory a zabývá se problematikou **řízení změn (change management)**. Podkladem jsou záznamy problémů (software problem report, SPR) a záznamy požadavků na změnu (software change request, SCR) shromažďované během fáze podpory. Cílem je identifikovat požadavky na budoucí verze systému. Fáze je prováděna pravidelně s periodou závislou na délce iterace. Za provádění je zodpovědný **výbor softwarové konfigurace (configuration control board, CCB)**.

Change management sestává z těchto kroků:

1. *Analýza požadavků na změnu*. Spočívá především v rozhodnutí, jestli se jedná o opravu vady, nebo o rozšíření. Pigoski v [Pigoski 1997] popisuje 4 hlavní kategorie údržby:
   - *Náprava.* Oprava vady v aplikaci (např. nesprávný výsledek výpočtu);
   - *Prevence.* Změny, které je třeba provést, aby nedošlo k vadě (např. úpravy, které bylo třeba provést v některých programech před příchodem přelomu tisíciletí);
   - *Adaptace.* Úpravy vyžádané změnami v technickém prostředí (např. přechod na nový operační systém);
   - *Vylepšení.* Rozšíření vyžádané novými potřebami (např. nový report).

   Body „náprava“ a „prevence“ lze považovat za opravu vad a body „adaptace“ a „vylepšení“ za rozšíření. Pro efektivní řízení změn je důležité správně rozlišovat mezi těmito dvěma kategoriemi. Během analýzy se tedy může ukázat potřeba překlasifikovat změnu ze Software Change Request na Software Problem Report a naopak.
2. *Prioritizace změn dle urgentosti*. Prioritizace změn by měla prováděna v úzké spolupráci s doménovými odborníky;
3. *Alokace změn na konfigurační položky*. **Konfigurační položka (configuration item, CI)** je jakákoliv část aplikace, která má svoji verzi a je předmětem **řízení konfigurací (software configuration manament, SCM)**. Alokace sestává z těchto kroků:
   i. *Identifikace konfiguračních položek potenciálně dotčených změnou.* Výbor softwarové konfigurace musí identifikovat konfigurační položky, které budou pravděpodobně dotčeny změnou. Z tohoto důvodu musí být vlastníci konfiguračních položek členy výboru;
   ii. Analýza vlivu změny. Jakmile jsou identifikovány konfigurační položky, které budou ovlivněny, je třeba analyzovat, jakým způsobem a vytvořit odhad pracnosti realizace změny. K tomu slouží především **alokační matice požadavků (requirements allocation matrix, RAM)**, která mapuje jednotlivé požadavky na části modelu realizované kódem. Posouzení vlivu změny je důležité pro zajištění integrity systému;
   iii. Ověření proveditelnosti změny (feasibility). Po analýze vlivu změny je třeba posoudit proveditelnost změny. Podklady pro rozhodnutí jsou priorita a obtížnost změny;

iv. *Naplánování změny.* Jedná se o komplexní proces, který bere v úvahu prioritu změny, existující projektový plán a závislosti mezi konfiguračními položkami a závislostmi mezi změnami navzájem. Na základě těchto úvah jsou poté naplánováno začlenění změn do nových verzí jednotlivých konfiguračních položek;

v. *Informování iniciátora požadavku.* V tuto chvíli je požadavek podpory stále považován za otevřený. Uzavřen bude v okamžiku implementace změny a uvedení do provozu. Uživatel, který podal původní požadavek na podporu by měl být informován o stavu požadavku odpovědným pracovníkem podpory.

7 Formalizace softwarového projektu z hlediska aparátu systémového modelování

7.1 Úvod

Nyní přistoupím k vytvoření formalizace softwarového projektu. Pro úvod shrnu, že počátečním impulsem, v konečném důsledku ústícím do vytvoření informačního systému, je situace, kdy **zákazníkovi** (firmě, instituci, apod.) vznikne určitá potřeba, kterou se rozhodne řešit formou využití zakoupeného nebo vyvinutého **softwarového řešení** (dále jen řešení). Ucelené softwarové řešení se typicky skládá

- ze softwarového produktu (vlastního software),
- z technického prostředí pro běh produktu (hardware a platforma),
- dokumentace software a dalších artefaktů (např. popisu potřebných úprav business procesu),
- školení uživatelů (viz kapitola 6.4).

V současnosti jsou běžné ještě širší služby, jež bývají nazývány informatické řešení a jehož součástí je navíc správa hardwaru, organizační opatření, konzultační činnost a další služby. Toto širší chápání však již přesahuje rámec zaměření práce a nebude tedy uvažováno.

Zákazník poptává **řešitele**, který by řešení dodal. Zákazník je typicky firma či instituce, která poptává jinou firmu či instituci jakožto dodavatele. Z obecného hlediska je ale možná i situace, kdy zákazníkem a řešitelem je softwarová firma v jedné osobě. To je případ tzv. krabicového softwaru, jemuž se v angličtině i normách (ISO/IEC 25051) říká **Commercial Off-The-Shelf software** – COTS. Roli zákazníka poté hraje určitá část firmy (typicky obchodní a marketingové oddělení) a roli řešitele oddělení vývoje.

Zákazník tedy poptáváním softwarového řešení řeší svoji potřebu (**potřeba zákazníka)** a má určitá **očekávání** o řešení.

7.2 Cíl

Cílem formalizace softwarového projektu je nalézt analogii mezi doménou softwarového projektu a aparátem systémového modelování, jež umožní formálně pracovat s pojmy softwarového inženýrství, jak bylo stanoveno v cíli práce.

7.3 Metodika

Řešení je vytvářeno v rámci **softwarového projektu**. Na softwarový projekt lze nahlížet jako na dynamický systém[8], jež transformuje vstupy na výstupy. Z cílů i rešerše vyplývá potřeba nahlížet na softwarový projekt v co nejširších souvislostech a různých úrovních detailu. Z tohoto důvodu jsem jako zastřešující model zvolil obecný systémový model disciplíny systémového modelování [Získal 2003]. Jako vhodný se jeví *dynamický deskriptivní* model, tedy model sloužící k formálnímu popisu reálného systému, který se v čase mění, podobně jako se může měnit struktura projektu. Pro jednoduchost budu softwarový projekt modelovat *deterministickým* systémovým modelem.

7.4 Struktura softwarového projektu

Prvním krokem formalizace softwarového projektu je formalizace jeho struktury z hlediska systémového modelování. Model systému se obecně skládá ze vstupů, výstupů, vnitřních prvků a vazeb. Vstupy dělíme na endogenní, což jsou ty, které nás z hlediska systému zajímají a exogenní, jež jsou ostatní vstupy, jež je třeba při modelování brát v potaz.

Součástí vymezení systému v systémovém modelování je tzv. *systémovým řez*, tj. ohraničení modelu podle nějakého úhlu pohledu na realitu. Při tvorbě systémového řezu je z hlediska systémového modelování klíčovým pojmem *izolovatelnost prvku*. Izolovatelnost nám určuje, nakolik je prvek nezávislý na okolí, které zůstane mimo systémový řez. Tato závislost by měla být minimální. Z tohoto hlediska lze zákazníka chápat dvojím způsobem:

1. Zákazník je vně systému a se systémem komunikuje. Tato situace odpovídá historickému přístupu k řízení projektů, kdy zákazník byl chápán jako vnější hráč. Vůči systému poté vystupuje v několika rolích:
   i. Je specifikátorem endogenních vstupů, tedy vstupů, jež nás při modelování zajímají.
   ii. V průběhu projektu interaguje se systémem pomocí vazeb zevnitř ven, zvnějšku dovnitř, zevnitř ven a dovnitř a zvnějšku dovnitř a ven.
   iii. Je validátorem výstupu: zákazník (obvykle na základě funkčních testů) posuzuje jakost výsledného produktu a jeho korelaci s endogenními vstupy.
2. Zákazník je interní prvek systému. Externí entita, jež poté komunikuje se systémem je *trh* a jeho požadavky. Zákazník je součástí systému a je mluvčím požadavků trhu. Tato varianta odpovídá moderním pohledům na projektové řízení, kdy zákazník a řešitel otevřeně a kooperativně spolupracují na řešení problému [Bělohlávek et al. 2001] a je v souladu s agilními přístupy k řízení softwarových projektů (kap. 5.3.2).

[8] Pojmem „systém“ je zde i dále označován softwarový projekt, tedy nikoliv softwarový systém, jež je projektem vytvářen. Ten bude i nadále nazýván „softwarový produkt“.

Podobně je tomu i se subdodavateli, které je možné chápat buď jako externí prvky komunikující se systémem, nebo jako vnitřní prvky systému. Jelikož však účast dodavatele na projektu vzniká jako součást řešení projektu, přiklání se autor spíše k modelu, kdy dodavatelé jsou vnitřní prvky.

Tab. 3 přináší přehled struktury systémového modelu softwarového projektu pro obě varianty. Položky týkající se situace, kdy zákazník je interní prvek systému jsou uvedeny kurzívou.

Pojem syst. modelování	Význam v SW projektu	Označení množiny
• Endogenní vstupy.	• Explicitně specifikované požadavky na softwarový produkt.	I_S
• Exogenní vstupy.	• Vnější podmínky specifikované (prostředí) i nespecifikované (krizové situace).	I_E
• Vstupy (sjednocení endogenních a exogenních vstupů)	• Všechny vnější vstupy a faktory ovlivňující projekt.	$I = I_S \cup I_E$
• Výstupy.	• Softwarový produkt a jeho parametry; • Technické prostředí pro běh produktu; • Dokumentace a další artefakty; • Školení.	O
• Vnitřní prvky.	• Tým (role v projektu); • Nástroje (vývojové a pomocné); • Artefakty (kód a dokumentace); • Dodavatelé; • *Zákazník*.	P
• Vnitřní vazby.	• Proces; • Vnitřní řízení projektu; • Komunikace uvnitř týmu; • Komunikace s dodavateli; • *Komunikace se zákazníkem.*	R_I
• Vazby zevnitř ven.	• Informace zákazníkovi o průběhu projektu. • *Marketingové aktivity*.	R_{IO}
• Vazby zvnějšku dovnitř.	• Zadání a požadavky na projekt od zákazníka. • *Vliv trhu.*	R_{OI}
• Vazby zevnitř ven a dovnitř.	• Požadavky spolupráce se zákazníkem ze strany týmu. • *Uvedení na trh a monitoring trhu.*	R_{IOI}
• Vazby zvnějšku dovnitř a ven	• Přímé poskytnutí požadované informace či dokumentu zákazníkovi.	R_{OIO}

Pojem syst. modelování	Význam v SW projektu	Označení množiny
	• *Přímá reakce na tržní posun.*	
• Vazby (sjednocení)	• Všechny vazby týkající se projektu	$R = R_{OIO} \cup R_{IOI} \cup R_{IOI} \cup R_{OI} \cup R_{OI} \cup R_{IO} \cup R_{IO} \cup R_I \cup R_I$

Tab. 3 – Složky systémového modelu softwarového projektu

7.4.1 Struktura prvků

Jednotlivé prvky modelu mohou mít různou strukturu podle konkrétní potřeby modelu. Obecně půjde o různá rozhodnutí klasického problému složitost modelu versus praktické stránky použití modelu.

Vnitřní prvky mohou být dle konkrétního účelu a potřeby modelu

- objekty,
- třídy.

Objekty volíme v případě, že je třeba detailně modelovat systém například na úrovni jednotlivých osob. V obecných modelech budou prvky typicky třídy, tedy např. prvek „programátor“ bude reprezentovat třídu programátorů. Třídu tedy použijeme v případě, že nás nezajímá struktura a dynamika mezi objekty jedné třídy. Podobně můžeme učinit i rozhodnutí například mezi prvkem typu objekt „testovací protokol“ nebo třídou „dokumentace“. Dále v celém textu však budeme pracovat jen na úrovni tříd.

Z hlediska vazeb vstupů na výstupy můžeme na projekt pohlížet jako na výpočetní problém [Vaníček et al. 2007]. Můžeme poté prohlásit, že softwarový projekt je též zobecněný výpočetní problém, tedy uspořádaná trojice (I_S, O, Q), kde I_S je množina endogenních vstupů, tedy vstupních dat, O je množina výstupů, tedy výstupních dat. Relace $Q \subset I_S \times O$ je podmínka, jež vstupům přiřazuje „správné“ výstupy.

V tomto případě musíme prvky množiny výstupů příslušným způsobem zjemnit oproti Tab. 3, např. softwarový produkt rozložit na jednotlivé moduly, objekty, procedury a další logické celky, jež bude možné dát do relace s prvky množiny vstupů (tedy požadavky). Pokud softwarový projekt „pracuje“ jako výpočetní systém správně, je každý vstup (požadavek) v relaci s množinou prvků výstupu (logických celků softwarového produktu). Podobně můžeme rozdělit další typy výstupů na vhodné celky pro vytváření relace. Jednotlivé dokumenty můžeme např. rozdělit na oddíly či kapitoly, jež jsou v relaci s požadavky na vstupu.

Oproti klasickému výpočetnímu problému je zde však posun v tom, že výstupy nejsou předem známy. V okamžiku, kdy je známo zadání (tedy vstup) je sice známo, CO má softwarový produkt dělat, není však známo JAK (viz kap. 6). Relaci je tedy možné vytvořit nejdříve ve fázi návrhu.

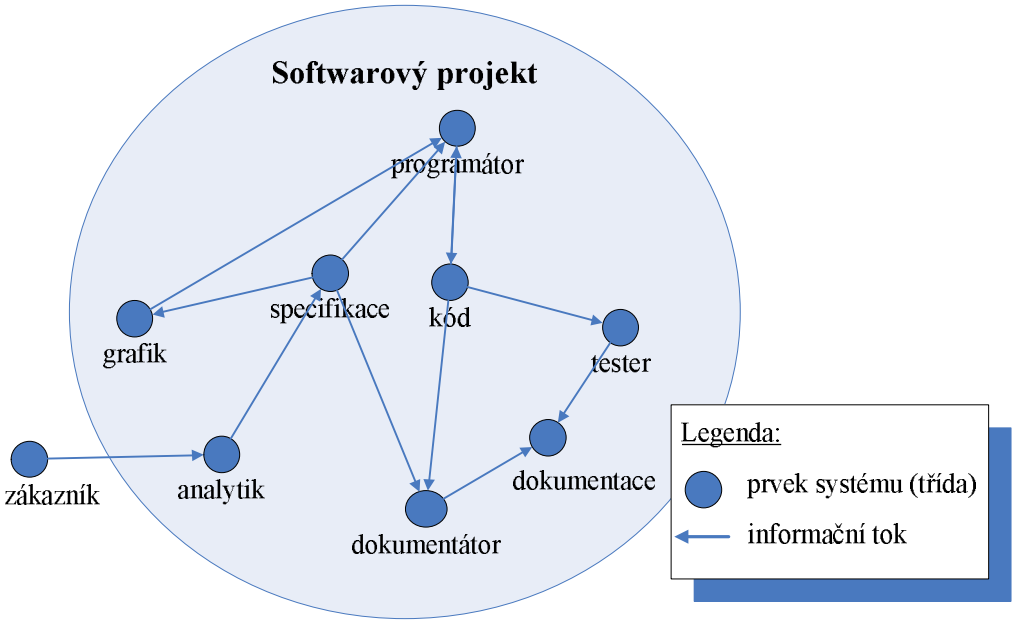
Další posun spočívá v tom, že v klasickém výpočetním problému nám splnění relace zaručuje, že výpočet dospěl k správnému výsledku. V případě navrženého systémového modelu nám splnění relace nezaručuje, že softwarový

produkt se chová správně podle zadání, existence relace však prokazuje, že konkrétní vstup má svůj odraz v softwarovém produktu, byl tedy řešen a v případě neuspokojivé funkce je snadné dohledat, které všechny výstupy s ním souvisejí.

7.4.2 Práce se systémovým modelem

Vhodným formálním aparátem pro práci se systémovým modelem je aparát teorie grafů, např. [Vaníček et al. 2007]. **Orientovaný graf** je zadán dvěma konečnými množinami. Množinou V všech vrcholů (či uzlů) a množinou H všech orientovaných hran (či spojnic uzlů). Prvek množiny V všech vrcholů grafu se nazývá **vrchol** či **uzel**. Prvek množiny H všech orientovaných hran grafu se nazývá **orientovaná hrana** či **šipka**. Dále pak je orientovaný graf pravidlem, které stanoví odkud kam, tedy ze kterého vrcholu do kterého daná orientovaná hrana směřuje. Tím je určen tak zvaný **počáteční vrchol orientované hrany** a **koncový vrchol orientované hrany.** Matematicky je tato konstrukce vyjádřena jako uspořádaná trojice (V, H, φ), kde φ je zobrazení H do $V \times V$. Takové zobrazení φ se nazývá **incidenční zobrazení** v orientovaném grafu.

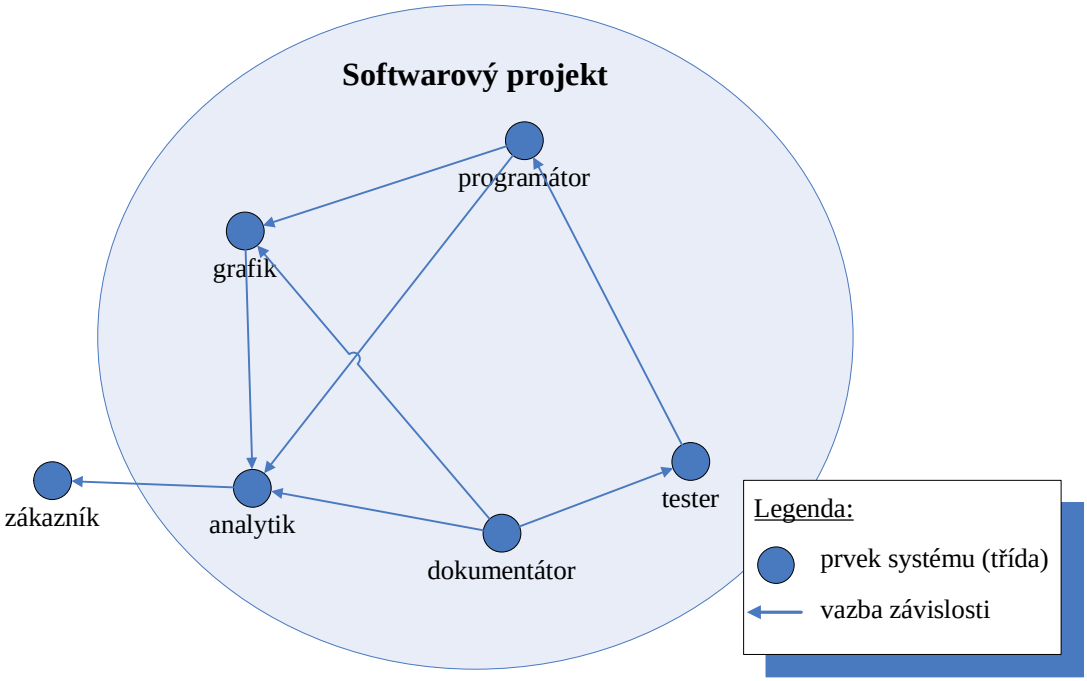
Grafem softwarového projektu budeme nazývat orientovaný graf, jehož množina vrcholů $V = I \cup O \cup P$, množina hran $H = R$. Jde o *multigraf*, jelikož mezi dvěma prvky může být i více různých hran. Graf neobsahuje smyčky. Pokud je třeba rozlišovat třídy a objekty, lze vrcholům přiřadit rozlišující atribut.



Obr. 15 – Příklad grafu softwarového projektu na úrovni tříd s vazbami toku dat

Obr. 15 a Obr. 16 obsahují příklady grafů projektu. Na Obr. 15 jsou uzly grafu příklady rolí a artefaktů a hrany reprezentují informační toky, jimiž mohou být různé podklady, zprávy a komunikace. Na Obr. 16 jsou prvky pouze role v systému a orientované hrany mezi znázorňují relaci závislosti, tj. že jedna role je při své práci závislá na práci role jiné, ke které ukazuje šipka.

Grafů projektů a jejich typů existuje velké množství, především v závislosti na použité metodice, nicméně i projekty vedené stejnou metodikou se budou v některých grafech typicky lišit, například v grafu komunikace. V optimálním případě by graf komunikace měl být úplným grafem, tj. všichni by měli komunikovat se všemi. To je model doporučovaný agilními metodikami (viz kap. 5.3.3), nicméně v praxi, zvláště u větších projektů, by režie takto husté komunikace přesáhla její přínosy.



Obr. 16 – Příklad grafu softwarového projektu na úrovni tříd s vazbami typu závislost

7.4.3 Vlastnosti softwarového projektu

Nyní, když máme k dispozici formální model softwarového projektu, můžeme s jeho pomocí definovat řadu pojmů softwarového inženýrství a řízení projektů, například i pojem úspěšnosti projektu.

Jak bylo uvedeno v kap. 4 „Terminologie“, jakost je v normě ISO 9000 definována jako míra splnění požadavků zákazníka. Za **jakostní softwarové řešení** je potom z hlediska systémového modelování považován výstup (řešení), jehož parametry odpovídají endogenním vstupům (požadavkům). Pokud nás z řešení bude zajímat především softwarový produkt, budeme hovořit o charakteristikách jakosti softwarového produktu dle ISO/IEC 9126-1 až 4 [ISO 9126-1, ISO 9126-2, ISO 9126-3, ISO 9126-4]

Za **úspěšný softwarový projekt** z hlediska zákazníka je považována situace, kdy je řešení jakostní při dodržení omezujících podmínek, jimiž jsou především čas a peníze. Omezující podmínky jsou ovšem v praxi obvykle měkké a do určité míry je možné od nich slevit, pokud to přispěje ke zvýšení jakosti produktu.

Chápání úspěšnosti projektu z hlediska řešitele se může buď ztotožňovat s chápáním úspěšnosti softwarového projektu z hlediska zákazníka, pro řešitele však mohou být důležité i jiné ukazatele, např. čistý zisk. Pro jednoduchost však lze jako **cíl projektu** chápat dosažení úspěšnosti projektu z hlediska zákazníka.

7.5 Formalizace řízení softwarového projektu

Jak bylo vysvětleno, z hlediska systémového modelování se projekt skládá ze

- vstupů,
- vnitřních prvků,
- výstupů,
- vazeb.

Vstupy a výstupy jsou dané, mohou ale být nastaveny

- vnitřní prvky,
- vazby mezi vnitřními prvky,
- vazby zevnitř ven.

Řízení projektu se tedy týká těchto prvků, ale i jejich struktur. Zavedeme tedy pojem činitel projektu:

Činitel projektu je

- vnitřní prvek,
- vazba mezi vnitřními prvky,
- vazba zevnitř ven,
- libovolný podgraf grafu, jehož množina vrcholů je P a množina hran je sjednocení vnitřních vazeb a vazeb zevnitř ven: $R_I \cup R_{IO}$.

Množinu všech činitelů projektu budeme označovat C.

Pokud budeme předpokládat, že smyslem řízení projektu je dosažení cíle projektu, potom obsahem řízení projektu je nastavování činitelů projektu. To se děje na základě vstupů, jedná se tedy o **adaptaci systému** na základě vstupů tak, aby optimálním způsobem (tj. s minimem zdrojů a času) dosahoval cíle projektu. Takový proces nazveme **optimalizace struktury projektu**.

7.5.1 Změna struktury projektu jakožto adaptace systému

Metodika řízení softwarového projektu je z tohoto hlediska tedy **znalost adaptace systému**, jež má tyto klíčové vlastnosti:

1. Na základě této znalosti je upravována struktura systému a nastavovány vazby.
2. Motivací pro její použití je víra, že adaptace systému s použitím znalosti povede ke splnění cíle projektu.

Tvrdé metodiky (kap. 5.2) se snaží adaptaci co nejvíce řídit, zatímco měkké metodiky (kap. 5.3) spíše spoléhají na schopnosti samoorganizace systému a zaměřují se spíše na vedení než řízení. Z hlediska prvků systému pak toto

znamená, že tvrdé metodiky pracují na úrovni vyšší granularity, jelikož detailně specifikují prvky, vazby i dynamiku, zatímco měkké metodiky pracují na nižší úrovni granularity, jelikož se zaměřují pouze na praktiky, principy a hodnoty.

Jak bylo řečeno, měkké metodiky využívají schopnosti samoorganizace systému. Pro jejich úspěšné použití tedy systém musí vykazovat rysy komplexního adaptivního systému [Holland 1995, Wheatley 1992]. Čím bude metodický přístup měkčí, tím zřetelnější musí být schopnost samoorganizace. Samoorganizujícími prvky jsou členové týmu, jež tedy pro splnění předpokladů úspěšné samoorganizace musí splňovat tato kritéria:

- Členové týmu musí být zaměřeni na cíl projektu, nikoliv na své osobní cíle;
- Každý člen týmu musí být schopným odborníkem;
- Členové týmu musí být schopni efektivní kooperace;
- Členové týmu se specializují a využívají svých předností.

Důležitým problémem též je, jakým způsobem je *detekována* potřeba adaptace systému. Z tohoto hlediska rozlišujeme dvě situace:

Adaptace ex post, tedy na základě minulosti. V tomto případě je podnětem k adaptaci nesoulad výstupů se vstupy. Tento typ adaptace je typický u spirálního vývojového cyklu, nelze jej však použít u vodopádového cyklu, kde výstup získáme až na konci projektu a nesoulad by tedy znamenal nesplnění cílů projektu.

Adaptace ex ante, tedy s uvážením budoucnosti. V tomto případě je určitým způsobem predikováno, že za stávající struktury systému nebude projekt úspěšný a je podstoupena adaptace.

Adaptace ex ante může být kombinována s adaptací ex post a naopak. O adaptaci ex ante hovoříme u již běžícího projektu, ale i u projektu začínajícího, jelikož počáteční vytváření infrastruktury je de-facto adaptace ex ante.

U již běžícího systému jsou zdrojem informací pro vyhodnocení potřeby adaptace metriky týkající se systému a výstupu. U začínajícího projektu je zdrojem informací analýza. Analýza může poskytnout informaci, že stávající struktura systému je vyhovující, či avizovat nutnost adaptace.

Znalost strategického řízení softwarového projektu se tak skládá ze dvou oblastí:

- Znalost, na jejímž základě je identifikována potřeba adaptace.
- Znalost, jakým způsobem adaptovat systém.

7.5.2 Proces řízení softwarového projektu

Zajímavé je, zamyslet se, co znamená **proces řízení softwarového projektu** v kontextu formalizace. Proces jsem v kap. 4 vymezil v souladu s ISO 9000 jako „soubor vzájemně souvisejících nebo vzájemně působících činností, který přeměňuje vstupy na výstupy“. Nyní můžeme rozlišit dva typy procesů:

1. Proces jako postup adaptace systému: vstupem je softwarový projekt ve stavu S_1 a výstupem softwarový projekt ve stavu S_2. Tento proces tedy transformuje softwarový projekt, tj. jeho náplní je adaptace systému popsaná výše.

2. Proces jako popis dynamiky systému: vstupem je zadání a výstupem softwarový produkt. Jedná se tedy o proces tvorby softwarového produktu.

První typ zahrnuje činnosti, jež se týkají adaptace systému, tedy např. zvyšování kvalifikace pracovníků, zatímco druhý typ reprezentuje rutinní pravidelné či ad-hoc procesy, při nichž se sice struktura systému může měnit, nicméně adaptace není primárním cílem procesu. Příkladem takového procesu jsou pravidelné integrace systému a řešení případných konfliktů.

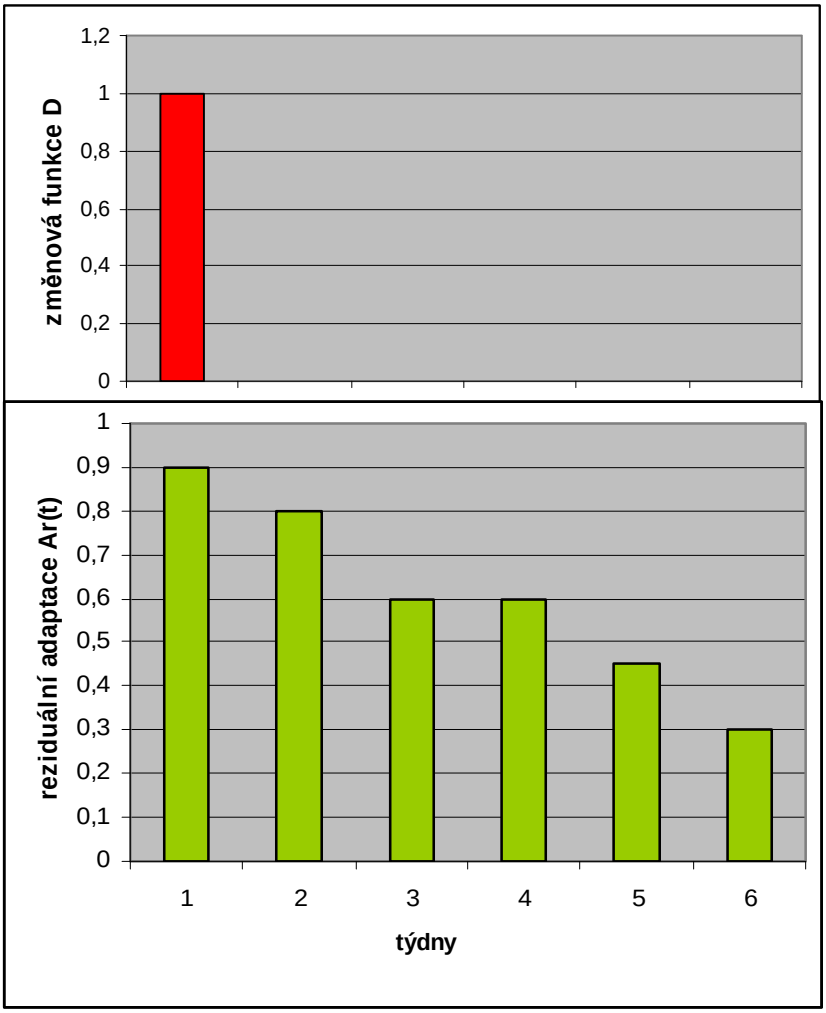
7.5.3 Agilnost metodiky

Dalším zajímavým problémem je značně diskutovaná **agilnost metodiky** a potažmo agilnost řízení projektu. David Hecksel v [Hecksel 2007] uvádí tyto klíčové charakteristiky agilnosti pro posuzení míry agilnosti procesu:

- jednoduchost,
- přijetí změny,
- inkrementální změna,
- rychlá zpětná vazba,
- nízký počet artefaktů,
- otevřená komunikace,
- rychlá integrace,
- zaměření na lidi a komunikaci spíše než na proces a nástroje,
- zaměření na funkční software spíše než na rozsáhlou dokumentaci,
- zaměření na spolupráci se zákazníkem, spíše než na dodržování plánu.

I když jde o nejpřesnější definici, jakou jsem při zpracování rešerše našel, při pohledu na problém agilnosti z hlediska provedené formalizace vidíme, že jejím problémem je, že směšuje příčiny a následky agilnosti, tedy z hlediska systémového modelování vstupy s důsledky vstupů, tj. adaptací. Z hlediska systémového modelování agilnost souvisí se schopností adaptace systému. Mějme systém, jež je ideálně adaptovaný na své vstupy, tj. má strukturu, jež dosahuje cíle projektu s minimálními zdroji. Při změně vstupu by mělo dojít k opětovné adaptaci vedoucí opět ke stavu optimální struktury projektu. Změna na vstupu je relativně velmi rychlá událost, kdy se řešitel dozví změnu požadavků. Je tedy možné tuto změnu modelovat impulsem určité velikosti. Systém na tento impuls reaguje zahájením procesu adaptace, tj. změnou struktury. Tato změna struktury se může týkat pouze určitých prvků (kód, dokumentace), či celých činitelů (řízení kvality; pojem činitel viz výše). Velikost impulsu odpovídá velikosti změny struktury systému. Impuls tedy vyvolává v systému *přechodový jev* o určité délce. Charakter průběhu přechodového jevu je ukazatelem **agilnosti systému**. Pro přechodové jevy, jež jsou běžně užívané např. v teorii obvodů je dispozici i matematický aparát, nicméně pro potřeby naplnění cílů této práce se spokojíme pouze s tímto konstatováním.

Otázkou nyní je, jakou zvolit jednotku pro velikost impulsu. Velikost impulsu by měla odrážet velikost nutné změny struktury. Ta však v okamžiku vzniku impulsu není známa. Z tohoto důvodu je velikost impulsu nutno aproximovat vhodnou veličinou. Touto veličinou může být v softwarovém inženýrství procento změny zadání. Výpočet procenta může být založen např. na počtu use-case dotčených změnou u metodiky RUP, počtu feature v případě metodiky FDD, počtu User-stories u XP, apod. Hodnotě 1 by tedy odpovídala změna celého zadání. Příklad je uveden na Obr. 17.



Obr. 17 – Příklad změnové funkce a reziduální adaptace

V horní části grafu jsou červeně zakresleny změnové impulsy. **Míru proměnlivosti zadání** v daném časovém období lze definovat jako sumu velikostí všech impulsů I_i proběhlých v časovém intervalu Δt děleno velikostí tohoto intervalu, tedy:

$$V \triangleq \frac{\sum_{i=1}^{n} I_i}{\Delta t}.$$

Funkce vynesená zeleně ve spodní části grafu na Obr. 17 znázorňuje časový průběh adaptace pomocí funkce reziduální adaptace**,** jež znázorňuje míru proběhlé adaptace v určitém časovém okamžiku. Otázkou je, jak tuto míru kvantifikovat. V praxi lze použít například princip založený na kvantifikaci činností. Proces adaptace bude vyžadovat

provedení určitých činností. Každá tato činnost má určitou důležitost. V určitém časovém okamžiku *t* máme množinu činností v určitém stádiu rozpracovanosti. Každá činnost má ohodnocení důležitosti $w(a)$. Jako stupnici ohodnocení důležitosti je možné zvolit libovolnou ordinální stupnici tak, aby platilo, že $w(a_1) > w(a_2)$ pokud činnost a_1 má větší důležitost než činnost a_2. Hodnota **reziduální adaptace** ***Ar*** pro adaptaci *d* okamžiku *t* je poté rovna

$$A_r(d,t) \triangleq 1 - \frac{\sum_i^n (w(a_i) \cdot c(a_i,t))}{\sum_i^n w(a_i)},$$

kde $c(a_i,t)$ je poměrná část splnění činnosti a_i v čase *t*, $w(a_i)$ je důležitost činnosti a_i. A_r se s postupem adaptace v průběhu času blíží k nule.

Činnosti musí mít dostatečně jemnou granularitu, aby se neprolínaly. Na druhou stranu příliš jemné členění zvyšuje pracnost zpracování reziduální adaptace.

Jelikož doba pro dokončení adaptace může být větší než interval mezi dvěma změnovými impulsy, může v určitém okamžiku probíhat více adaptací zároveň. Definujme proto **celkovou reziduální adaptaci** v čase *t*

$$A_{rc}(t) \triangleq \sum_i A_r(d_i,t)$$

Míru agilnosti systému můžeme chápat dvojím způsobem:

1. Jako *časovou veličinu* v čase *t*, jež bude ukazatelem okamžité agilnosti, jež reprezentuje rychlost adaptace. Takovouto agilnost nazveme **okamžitá agilnost** a budeme ji definovat:

$$AGC(t) \triangleq A_{rc}(t) - A_{rc}(t-1)$$

Vyšší hodnota funkce *AGC* znamená vyšší okamžitou agilnost.

Užitečnějším údajem je však agilnost jako *statistická veličina* charakterizující agilnost celého systému. Pro tyto účely lze vypočítat **průměrnou okamžitou agilnost** či **střední okamžitou agilnost** a další statistické ukazatele.

7.5.3.1 Příklad

Celou problematiku bude nejlépe ilustrovat na konkrétním příkladu sledování ukazatelů adaptace softwarového projektu.

Jako první krok je třeba se rozhodnout o časové jednotce. U projektu malého rozsahu zvolíme pravděpodobně dny, u projektu většího rozsahu týdny. Možné jsou samozřejmě i jiné volby. Mějme tedy pro ilustraci projekt většího rozsahu, kde zvolíme časovou jednotku týdny. Projekt je zahájen v týdnu 0. Projekt jakožto systém začíná v určitém stavu připravenosti. Projekt začíná tím, že je provedena příprava (viz kap. 6.2), v rámci které je ujasněno, jaké prvotní požadavky na strukturu systému projekt bude mít. Budeme předpokládat, že prvotní analýza bude trvat dva týdny. V týdnu 3 tedy bude bude znám výsledek prvotní analýzy, jež přinese specifikaci zadání. Jelikož před tímto okamžikem žádné zadání neexistovalo, hodnota změnové funkce $D(1) = 1$. Po analýze zadání se vedení projektu rozhodne

realizovat některé změny struktury. Tato adaptce bude označena I_1. Jako stupnice ohodnocení pro důležitost změn byla zvolena ordinální stupnice <1, 10>.

1. Přijetí praktiky párového programování (5.3.4.3) (důležitost 4).
2. Další školení odbornosti (důležitost 2).
3. Restrukturalizace týmu: přijetí dalšího testera: (důležitost 6) a nového grafika (důležitost 8)
4. Zakoupení nových nástrojů a jejich zavedení (důležitost 3).

Ve třetím týdnu tedy hodnota reziduální adaptace (a jelikož jde zatím o jedinou probíhající adaptaci, tak i celkové reziduální adaptace) dosahuje hodnoty 1, jelikož nebyly zatím provedeny žádné činnosti. První týden jsou zakoupeny a zavedeny potřebné nástroje, na konci týdne tedy reziduální adaptace klesne na

$$A_r(I_1, 3) = 1 - \frac{3}{3 + 4 + 2 + 6 + 8} = 1 - \frac{3}{23} \doteq 0,87.$$

Druhý týden je přijat tester a provedeno 30% školení:

$$A_r(I_1, 4) = 1 - \frac{3 + 0,5 \cdot 2 + 6}{3 + 4 + 2 + 6 + 8} = 1 - \frac{10}{23} \doteq 0,57.$$

Třetí týden je dokončeno školení, přijat nový grafik a z 20% provedeno školení párového programování.

$$A_r(I_1, 5) = 1 - \frac{3 + 2 + 6 + 8 + 0,2 * 4}{3 + 4 + 2 + 6 + 8} = 1 - \frac{19,8}{23} \doteq 0,14$$

Čtvrtý týden je dokončeno školení párového programování:

$$A_r(I_1, 6) = 1 - \frac{3 + 2 + 6 + 8 + 4}{3 + 4 + 2 + 6 + 8} = 1 - \frac{23}{23} \doteq 0.$$

Po čtvrtém týdnu je tedy adaptace I_1 dokončena.

V průběhu adaptace však v druhém týdnu adaptace (tj. ve čtvrtém týdnu projektu) došlo k upřesnění zadání, které k původním 42 use-case přidalo 3 nové a 8 původních ovlivnilo z 20%. Hodnota změnové funkce D v pátém týdnu tedy nabyla hodnoty

$$D(4) = \frac{3 + 8 * 0,20}{42} \doteq 0,11$$

To si vyžádalo přijetí vyšší úrovně zajištění kvality (důležitost 3), přijetí manažera bezpečnosti (důležitost 2) a zakoupení nových knihoven (důležitost 6). Tím byla nastartována nová adaptace I_2 s následujícím průběhem:

- Týden 1: 40% nového systému zajištění kvality, zakoupení 20% knihoven:

$$A_r(I_2, 4) = 1 - \frac{0,4 \cdot 3 + 0,2 \cdot 6}{3 + 2 + 6} = 1 - \frac{2,4}{11} \doteq 0,78.$$

- Týden 2: 80% nového systému zajištění kvality:

$$A_r(I_2, 5) = 1 - \frac{0,8 \cdot 3 + 0,2 \cdot 6}{3 + 2 + 6} = 1 - \frac{3,6}{11} \doteq 0,67.$$

- Týden 3: nový systém zajištění kvality implementován, knihovny dokoupeny:

$$A_r(I_2, 6) = 1 - \frac{3 + 6}{3 + 2 + 6} = 1 - \frac{9}{11} \doteq 0,18$$

- Týden 4: přijat nový manažer bezpečnosti:

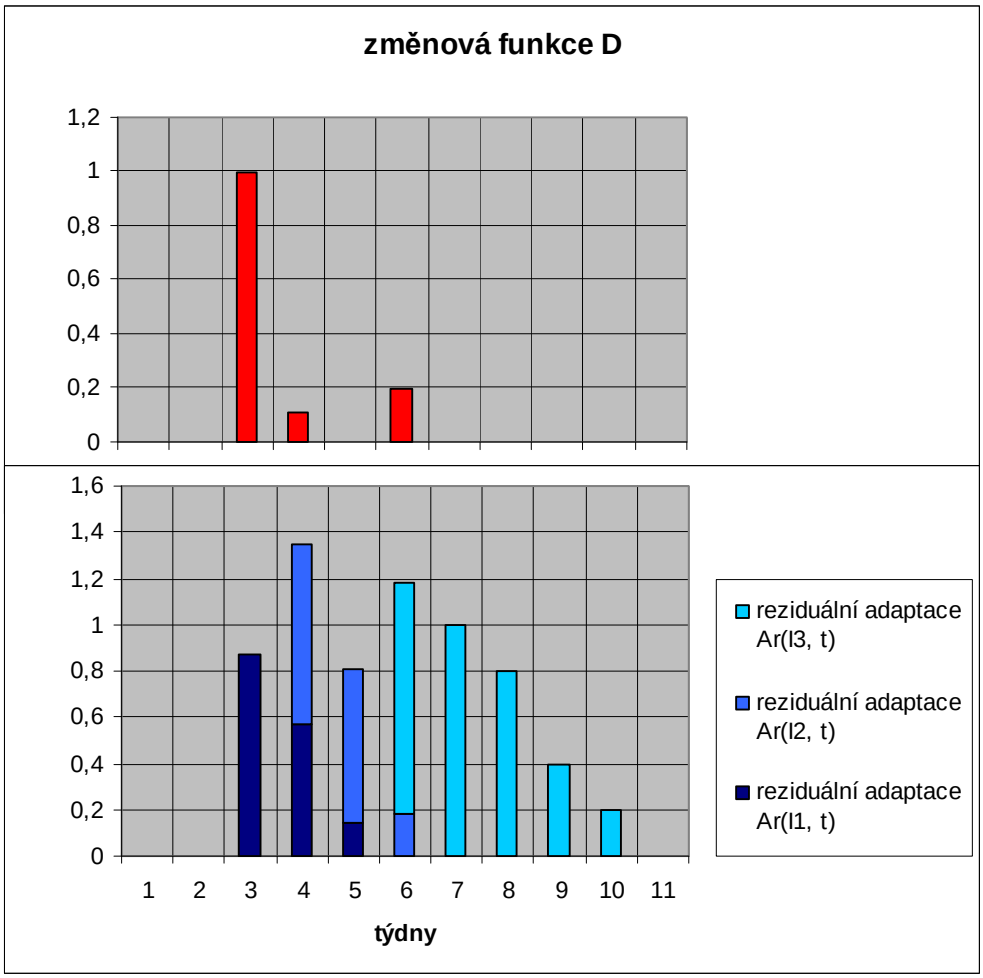
$$A_r(I_2, 7) = 1 - \frac{3 + 6 + 2}{3 + 2 + 6} = 1 - \frac{11}{11} \doteq 0.$$

Hodnoty průběhu adaptace jsou uvedeny v Tab. 4 a vyneseny v grafu na **Obr. 18**, kde je zanesen ještě průběh třetí podobné adaptace, jež byla spuštěna změnou v 6. týdnu.

týdny *t*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
změnová funkce *D*	0	0	1	0,11	0	0,2	0	0	0	0	0
reziduální adaptace $A_r(I_1, t)$	0	0	0,87	0,57	0,14	0	0	0	0	0	0
reziduální adaptace $A_r(I_2, t)$	0	0	0	0,78	0,67	0,18	0	0	0	0	0
reziduální adaptace $A_r(I_3, t)$	0	0	0	0	0	1	1	0,8	0,4	0,2	0
celková reziduální adaptace $A_{rc}(t)$	0	0	0,87	1,35	0,81	1,18	1	0,8	0,4	0,2	0

minimální CRA:	0
maximální CRA:	1,35
průměrná CRA:	0,6
medián CRA:	0,8

Tab. 4 – Průběh adaptace systému z příkladu a statistické ukazatele celkové reziduální adaptace (CRA)



Obr. 18 – Průběh adaptace systému z příkladu

7.5.3.2 Průběh adaptace systému vzhledem k agilnosti

Důležitou skutečností je, že agilnost není atributem softwarového projektu. Tím je **schopnost agilnosti**, jež představuje určité předpoklady agilnosti, např. pružnost vývojového procesu, cestování nalehko, osobní flexibilita týmu, apod. Skutečná agilnost však vždy vzniká až jako reakce systému na změnu. Bez změny tedy agilnost nemůžeme posuzovat.

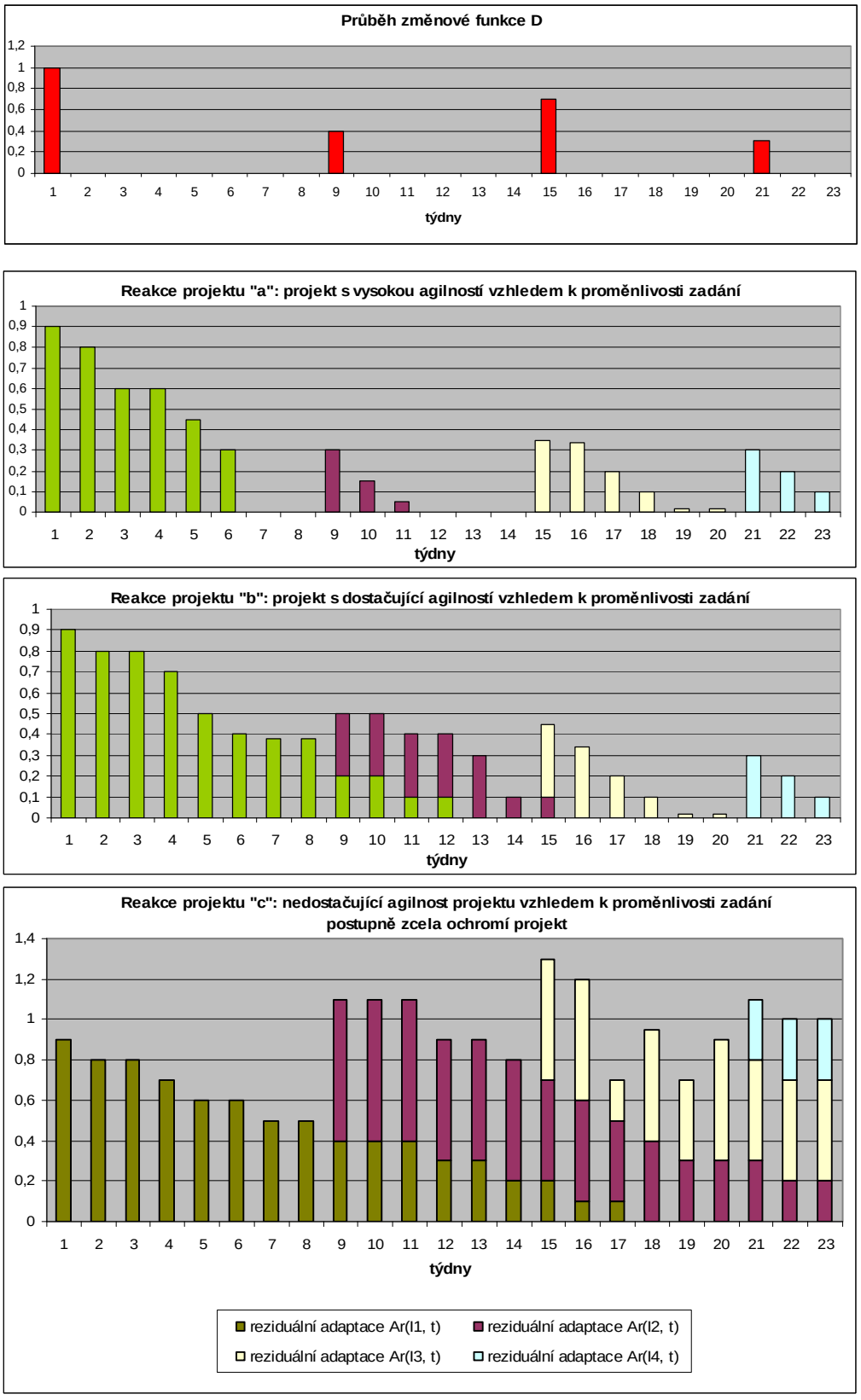
V praxi situace vypadá tak, že systém s určitou schopností agilnosti je konfrontován se zadáním s určitou mírou proměnlivosti zadání. Výsledkem potom je konkrétní průběh celkové reziduální adaptace. Příklady typických variant vidíme schematicky na Obr. 19 a statistiky celkové reziduální adaptace jsou shrnuty v Tab. 5. Schopnost agilnosti systému a) v porovnání s mírou proměnlivosti zadání je natolik velká, že systém vždy stačí (ve sledovaném období) provést adaptaci, reziduální adaptace se tedy stihne vrátit k nule. Takováto situace je ideální, neboť pokud podstatně nestoupne míra proměnlivosti zadání, tak systém bude schopen udržovat si optimální strukturu a tím je zajištěn klíčový předpoklad úspěšnosti projektu.

Oproti tomu u systému b) je agilnost nižší a dochází k tomu, že mezi dvěma impulsy nestačí adaptovat svoji strukturu. Reziduální adaptace tedy již nestačí mezi dvěma změnovými impulsy klesnout k nule, nicméně nepřeroste určitou hodnotu. U systému c) nedostatečná agilnost vede k ochromení projektu.

statistika	systém a)	systém b)	systém c)
minimální CRA	0,00	0,02	0,50
maximální CRA	0,90	0,90	1,30
průměrná CRA	0,25	0,38	0,88
medián CRA	0,20	0,38	0,90

Tab. 5 – Statistiky celkové reziduální adaptace (CRA) pro systémy z Obr. 19

Pro každý systém a projekt je možné stanovit hodnotu **krizové reziduální adaptace**. Při jejím dosažení je struktura systému již natolik nevyhovující, že dochází k pádu projektu. Velikost krizové reziduální adaptace je závislá na charakteru projektu a řadě ekonomických a psychologických aspektů na straně řešitele i zákazníka. Čím vyšší je hodnota krizové reziduální adaptace, tím více si může projekt dovolit „nezvládat“ změny.



Obr. 19 – Příklady systémů s různou schopností agilnosti

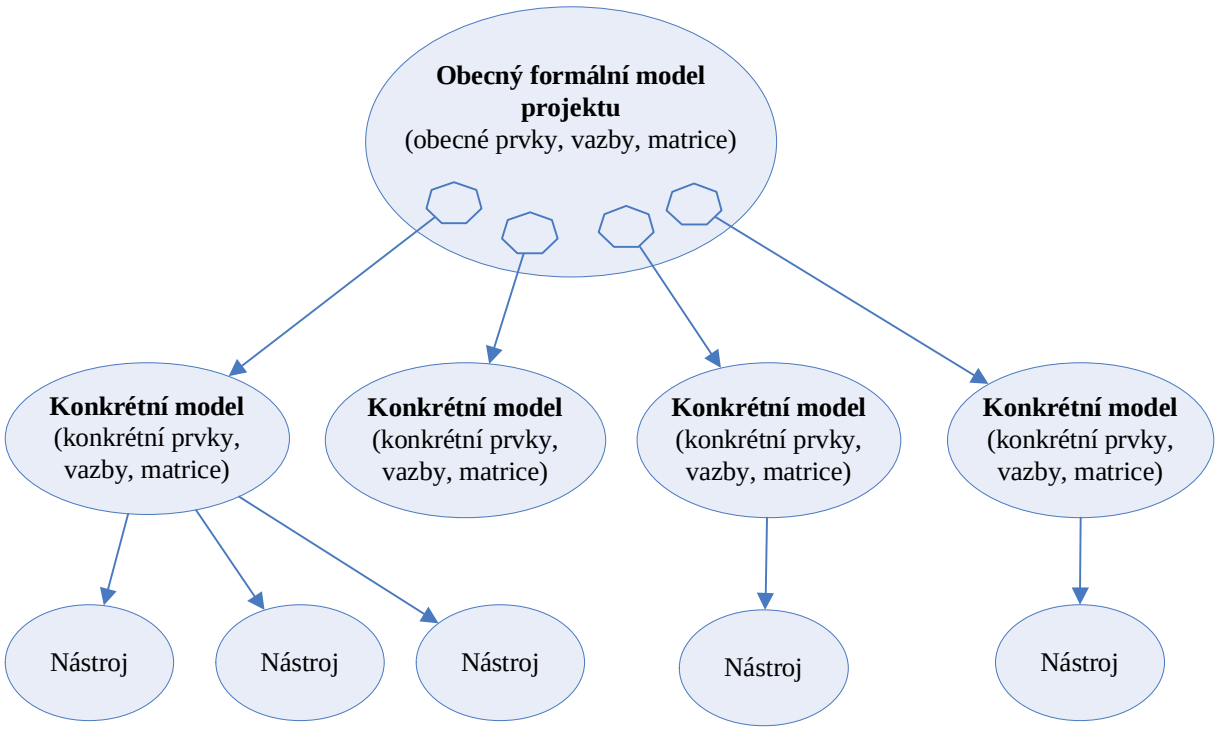
8 Tvorba metodologických pomůcek na základě modelu

Formalizace softwarového projektu popsaná v předcházející kapitole přináší nový úhel pohledu na problematiku řízení softwarových projektů. Vytvořený formální model má sloužit pro rozvoj poznání v oblasti metodologie softwarového inženýrství, a také pro podporu vývoje praktických podpůrných pomůcek, což jsou dva hlavní cíle práce.

Formální model popsaný v předchozí kapitole tvoří obecný rámec umožňující jak integraci obecných znalostí softwarového inženýrství, tak konkretizaci pro konkrétní účel, prostředí a podmínky. Na Obr. 20 je znázorněn postup, jakým způsobem je tohoto dosaženo.

Obecný formální model projektu definuje modelový rámec a pracuje s obecnými pojmy prvků a vazeb. Na základě tohoto obecného modelu je možné vytvářet konkrétní modely rozpracováním části obecného modelu na základě účelu konkrétního modelu, reality, prostředí a požadavků kladených na model. Tento konkrétní model se již skládá převážně ze specifických prvků a pojmů modelované reality, jež jsou vybírány s ohledem na cíl tvorby modelu, jímž je na konci vytvoření praktické pomůcky či pomůcek pro podporu modelovaného problému.

Poté je možné přistoupit k implementaci pomůcky. Tu je zde třeba chápat širším způsobem než je v softwarovém inženýrství obvyklé. Tedy nejen jako implementaci formou softwarového produktu, ale jako jakýkoliv aparát či nástroj přímo použitelný v praxi. Tvorba implementace modelu poté spočívá v rozhodnutí o požadovaných vstupech a výstupech a implementaci modelu v podobě vhodného softwarového či jiného podpůrného nástroje.



Obr. 20 – Postup od obecného formálního modelu projektu k praktickým metodologickým pomůckám

Celkový postup od obecného formálního modelu ke konkrétní pomůcce je tedy následující:

1. Vymezení oblasti obecného modelu k rozpracování;
2. Specifikace účelu modelu;
3. Tvorba modelu:
    a. Vymezení konkrétních prvků;
    b. Vymezení konkrétních vazeb;
4. Implementace nástroje.

V následujících dvou kapitolách ukáži dvě praktické pomůcky pro podporu řízení softwarových projektů, jež vznikly popsaným postupem.

9 Pomůcka pro optimalizaci struktury projektu

9.1 Úvod

První pomůcku jsem vytvořil v souladu s postupem uvedeným v kap. 8:

1. Vymezil jsem oblasti obecného modelu k rozpracování: vazby zvenku dovnitř a jejich vliv na prvky modelu;
2. Specifikoval jsem účel modelu: tvorba pomůcky pro podporu optimalizace struktury projektu na základě vlivu těchto vazeb;
3. Vytvořil jsem model:
    a. Vymezil jsem konkrétní vazby (podkapitola 9.2);
    b. Vymezil jsem konkrétní prvky (podkapitola 9.3);
4. Provedl jsem implementaci nástroje (podkapitola 9.9).

Dále ukáži v podkapitole 9.8 praktický příklad použití nástroje.

Jelikož je v praxi nejproblematičtějším výstupem softwarového projektu softwarový produkt, vytvořil jsem pomůcku pouze s ohledem na něj. Její rozšíření na další typy výstupů softwarového projektu je analogické.

9.2 Vymezení a konkretizace vazeb

Pomůcka je zaměřena na optimalizaci struktury projektu, tj. adaptaci systému na základě vstupů, jež mají vliv na strukturu projektu. Z hlediska systémového modelu bude tedy důležitá **vazba zvenku dovnitř**, a to

- **Vazba na prvky**, jež je vazba na jeden konkrétní vnitřní prvek, např. konkrétní roli či artefakt;
- **Vazba na strukturu,** jež je vazba na několik prvků a vazeb mezi nimi, což se týká např. organizačních otázek.

Souhrnně se tedy jedná o vazbu na činitele projektu, jež byly definovány v kap. 7.4. Z praktického hlediska pomůcky však není třeba tyto dva typy vazeb rozlišovat a budeme je souhrnně označovat pojmem **vazba**. Pojmy použité pro sestavení pomůcky jsou uvedeny v Tab. 6.

název	symbol	definice
množina vstupů	*I*	viz kap. 7.4
množina činitelů projektu	*C*	viz kap. 7.5

Tab. 6 – Přehled pojmů použitých v pomůcce pro optimalizaci struktury projektu

Jak jsem uvedl v předchozí kapitole, vazeb existuje velké množství typů. Zvažoval jsem, které vazby vyskytující se v realitě, do pomůcky začlenit. Nakonec jsem se rozhodl pouze pro dva typy vazby, který považuji za rozhodující. Jedná se o kompromis mezi přesností pomůcky a snadností jejího použití.

Definice 1:

Nárok vstupu *a* na činitel projektu ***s*,** kde $a \in I$ a $s \in C$ popisuje funkce dem(*a*, *s*). Oborem hodnot funkce dem je ordinální stupnice <0, 10>. Nulová hodnota znamená, že vstup *a* neklade nárok na adaptaci činitele projektu *s*. Nenulová hodnota nároku znamená, že vstup *a* klade nárok na adaptaci činitele projektu *s*. Čím je hodnota větší, tím větší je i kladený nárok.

- konec definice –

Příkladem je situace, kdy vedoucí projektu usoudí, že kvůli vstupu reprezentující přizpůsobitelnost produktu bude třeba adaptovat činitele projektu týkající se přenositelnosti softwarového prostředí.

V praxi lze někdy nároky substituovat, například místo přijetí nového odborníka zvýšit kvalifikaci některého člena týmu. Toto z hlediska systémového modelu představuje alternativy adaptace systému, kdy jeden činitel projektu můžeme substituovat jiným. Z formálního hlediska se jedná o jeden typ vazby mezi vnitřními prvky. Možnost substituce považuji z praktického hlediska za důležitou při optimalizaci struktury projektu, a proto jsem se rozhodl tento typ vnitřní vazby do pomůcky začlenit:

Definice 2:

Substituce činitele projektu s_1 a s_2 , kde s_1, $s_2 \in C$, je zobrazení $C \times C$ do ordinální stupnice <0, 10> popisuje funkce sub(s_1, s_2). Substituce specifikuje možnost substituovat nárok na adaptaci činitele projektu s_1 substitutem s_2. V případě, že substituce není možná, nabývá funkce sub hodnoty 0. Nenulová hodnota znamená možnost adaptace činitele projektu s_1 do určité míry substituovat činitele projektu s_2**.** Čím větší je hodnota *sub*, tím je substituce dokonalejší. V případě hodnoty 10 se jedná o dokonalé substituty.

- konec definice -

Dále uvádím další definice potřebné pro pomůcku. Jejich význam vysvětluji a ukazuji v dalších podkapitolách.

Jednotlivé nároky na adaptaci systému se sčítají. Lze tak hovořit o celkovém nároku na adaptaci činitele projektu. Tento celkový nárok budu nazývat diference činitele projektu a jeho definice je:

Definice 3:

Pojem **diference činitele projektu** s_j, kde $j = 1, \ldots, n$ označuje hodnotu funkce dif(s_j), která každému činiteli projektu $s_j \in C$ přiřazuje nezáporné celé číslo:

$$\mathrm{dif}(s_j) = \sum_{i=1}^{m} \mathrm{dem}(a_i, s_j),$$

kde a_i je vstup, *n* je počet činitelů projektu a *m* je počet vstupů.

-konec definice-

Tato definice předpokládá, že jednotlivé nároky jsou nezávislé a lze je tedy sčítat. Jedná se o zjednodušující předpoklad. Pokud tento předpoklad splněn není, je třeba provést korekci při odhadování velikosti nároku. Tj. pokud

ohodnotíme např. nárok dem(*a, s*), tj. nárok vstupu *a* na činitele *s* hodnotou 5 a nárok jiného vstupu na stejný činitel dem(*b, s*) hodnotou 8 a shledáme, že první nárok je částečně zahrnut v druhém nároku, snížíme hodnotu 8 po uvážení míry překryvu na 6.

Z hlediska substituce je zde též možnost sčítat substituční funkce, jelikož jeden činitel projektu je možné substituovat použitím více substitutů. O tom hovoří následující definice:

Definice 4:

Pojem **celková subsituce činitele projektu** s_j, kde $j = 1, \ldots, m$ označuje hodnotu funkce csub(s_j), která každému činiteli projektu $s_j \in C$ přiřazuje nezáporné reálné číslo:

$$\mathrm{csub}(s_j) = \sum_{k=1, k \neq j}^{n} \mathrm{sub}(s_j, s_k),$$

kde *n* je počet činitelů projektu.

-konec definice-

Výsledné nároky na adaptaci činitelů tedy mohou být zmírněny vnitřními substitučními vazbami. O tom hovoří následující definice:

Definice 5:

Pojmem **výsledná diference činitele** s_j budeme nazývat hodnotu funkce

$$\mathrm{vdif}(b_j) = \max(0, \mathrm{dif}(b_j) - \mathrm{csub}(b_j))$$

-konec definice-

9.3 Volba vstupů a činitelů

Pro sestavení pomůcky pro praktické využití je nyní třeba vymezit vhodnou podmnožinu množiny vstupů $I_2 \subset I$ a činitelů $C_2 \subset C$. Ideální vlastnosti těchto podmnožin by měly být tyto:

- Úplnost;
- Nezávislost;
- Minimalita.

Podmnožiny by tedy měly obsahovat všechny vstupy, resp. součásti systému, jež nejsou odvozeny od jiných. Z praktického hlediska by však mohutnost podmnožin měla být co nejmenší, jelikož časová složitost zpracování všech nároků je $\Theta(|I_2| \cdot |C_2|)$.

Vidíme, že tyto požadavky jdou proti sobě, je tedy třeba zvolit vhodný kompromis. Volba kompromisu může být založena na různých preferencích lišících se projekt od projektu. Jako příklad pro naplnění pomůcky jsem na základě rešerše zvolil základní vstupy a činitele, jež je možno aplikovat na všechny typy projektů.

Pokud budu dále hovořit o množině vstupů a množině činitelů, budu pod tímto chápat již takto vymezené podmnožiny všech vstupů a činitelů I_2 a C_2.

9.4 Příklady vstupů

Jako ukázku možných vstupů jsem zvolil charakteristiky jakosti dle norem ISO/IEC 9126-1 až 4 (viz kap. 5.4.2), jež stanovují obecné požadavky na softwarový produkt. Charakteristiky jakosti sice plně nevyhovují požadavku vzájemné nezávislosti, nicméně jsou uznávaným standardem a pokrývají všechny potřebné oblasti. Marketingová výhoda začlenění charakteristik jakosti do pomůcky spočívá též v tom, že při jejím použití je možné deklarovat, že byla vzata v úvahu ISO norma.

Charakteristiky jakosti hovoří pouze o vlastnostech produktu samotného, množina vstupů však musí obsahovat i podmínky vzniku produktu, jež též mohou významným způsobem ovlivňovat projekt. Nepodařilo se mě nalézt žádnou normu či uznávanou publikaci, jež by přehledně shrnovala podmínky vzniku produktu. Ukázku podmínek vzniku produktu jsem tedy vybral na základě rešerše a vlastní zkušenosti.

9.4.1 Charakteristiky jakosti

Jak jsem zmínil dříve, charakteristikami jakosti se zabývají normy ISO/IEC 9126-1 až 4 [ISO 9126-1], [ISO 9126-2], [ISO 9126-3], [ISO 9126-4] jež stanovují obecné požadavky na softwarový produkt.

Norma ISO/IEC 9126-1 stanovuje 6 charakteristik jakosti, kde každá charakteristika má několik podcharakteristik:

1. Funkčnost (Functionality):
    i. Funkční přiměřenost (Suitability);
    ii. Přesnost (Accuracy);
    iii. Schopnost spolupráce (Interoperability);
    iv. Bezpečnost (Security);
    v. Shoda ve funkčnosti (Functionality Compliance);
2. Bezporuchovost (Reliability):
    i. Zralost (Maturity);
    ii. Odolnost vůči vadám (Fault Tolerance);
    iii. Schopnost zotavení (Recoverability);
    iv. Shoda v bezporuchovosti (Reliability Compliance);
3. Použitelnost (Usability):
    i. Srozumitelnost (Understandability);
    ii. Naučitelnost (Learnability);

   iii. Provozovatelnost (Operability);
   iv. Atraktivnost (Attractivness);
   v. Shoda v použitelnosti (Usability Compliance);

4. Účinnost (Efficiency):
   i. Časové chování (Time Behaviour);
   ii. Využití zdrojů (Resource Utilisation);
   iii. Shoda v účinnosti (Efficiency Compliance);
5. Udržovatelnost (Maintainability):
   i. Analyzovatelnost (Analysability);
   ii. Měnitelnost (Changeability);
   iii. Stabilnost (Stability);
   iv. Testovatelnost (Testability);
   v. Shoda v udržovatelnosti (Maintainability Compliance);
6. Přenositelnost (Portability):
   i. Přizpůsobitelnost (Adaptability);
   ii. Instalovatelnost (Installability);
   iii. Slučitelnost (Co-existence);
   iv. Nahraditelnost (Replaceability);
   v. Shoda v přenositelnosti (Portability Compliance).

Vysvětlení jednotlivých charakteristik jakosti může čtenář nalézt např. v [Vaníček 2004], pro větší pohodlí je stručný popis připojen v Příloze 1.

9.4.2 Podmínky vzniku produktu

9.4.2.1 Stabilnost zadání

Stabilnost zadání určuje, do jaké míry lze předpokládat, že se zadání nezmění, oproti tomu, že se zadání bude měnit. Z hlediska stabilnosti je možné se zaměřit na různé atributy, např. předpokládané procento změn zadání, frekvence vzniku změn, apod. Z hlediska metodického rámce je ale důležitý faktický vliv na výstupní parametry zadání. Nízká stabilnost zadání bude typicky zvyšovat potřebu zpětné vazby, pružnost procesu a implementačních nástrojů.

9.4.2.2 Přesnost zadání

Zákazník není v některých případech schopen poskytnout dostatečně přesné zadání. Nepřesnost zadání zvyšuje nároky na projekt podobně jako stabilita zadání, se kterou souvisí.

9.4.2.3 Rychlost nasazení produktu

Je třeba zvážit požadovanou rychlost nasazení produktu, tj. čas na dokončení projektu v kontextu požadované funkčnosti a ostatních charakteristik jakosti. Pokud analýza a projektový plán není v souladu s očekávanou rychlostí nasazení, tj. termíny jsou napjaté, je třeba tuto situaci zvážit a ošetřit. Jednou možností je omezení rozsahu projektu, nicméně v některých případech je možné rozsah zachovat za cenu zvýšených nároků na některé výstupní parametry, např. počet členů týmu, nastavení procesu (inkrementální charakter) či posílení softwarové a hardwarové platformy.

9.4.2.4 Potřeba formální dokumentace projektu

Míra dokumentace projektu by dle zásad agilního přístupu (viz. např. [Buchalcevová 2002]) měla být nastavena jako nejmenší možná taková, aby přinášela užitek. V určitých projektech však může být zákazníkem vyžadováno větší množství formální dokumentace výstupů i průběhu projektu. V takovém případě je třeba analyzovat vliv této skutečnosti na výstupní parametry zadání.

9.4.2.5 Potřeby image projektu

Projekt jakožto cesta k vytvoření produktu nemusí být vždy chápán čistě účelově, ale řešitel se může setkat s potřebami na určitý image a reprezentaci projektu, který může být vyžadován marketingem vnějším i vnitřním. V průběhu projektu může být např. třeba produkt inzerovat a připravovat jeho vstup na trh. V praxi též dochází často k tomu, že z určitých důvodů je třeba postup projektu pravidelně prezentovat zákazníkovi nad rámec účelnosti, což opět přináší nároky.

9.5 Příklady činitelů

Z hlediska praktického využití jsem zvolené činitele rozčlenil do kategorií dle zvyklostí v softwarovém inženýrství, tedy na:

- Personální otázky (tým);
- Nastavení procesu;
- Artefakty vznikající v průběhu projektu;
- Softwarové a hardwarové nástroje, jak na straně platformy budoucího produktu, tak vývojových a podpůrných nástrojů;
- Komunikace a spolupráce.

9.5.1 Personální otázky

Personální otázky se týkají činitelů souvisejících s rolemi v týmu. Může být vyžadována adaptace charakteristik elementu systému odpovídající určité roli v týmu či širší adaptace týkající se i vazeb mezi elementy a jejich charakteristik a dynamiky.

9.5.1.1 Potřeby zvýšení kvalifikace

Potřeby zvýšení kvalifikace znamenají, že stávající úroveň kvalifikace týmu v některé oblasti je pro potřeby splnění zadání nedostatečná. Může se jednat o tyto kategorie:

- Potřeba osvojení aplikační domény;
- Potřeba prohloubení znalostí implementačních nástrojů a technik;
- Potřeba osvojení podpůrných technik a nástrojů.

9.5.1.2 Potřeba personální stabilnosti

Personální stabilnost znamená nízkou míru fluktuace členů týmu. Zde se jedná o celkovou personální stabilnost, nikoliv o rizika odchodu konkrétních rolí, která jsou předmětem řízení rizik.

9.5.1.3 Potřeba osobního nasazení

Některé projekty mohou vyžadovat zvýšenou míru osobního nasazení členů týmu, pokud očekáváme přesčasy, nudný charakter práce, apod. Tento parametr se těžko ovlivňuje, proto je třeba opatrnosti. Možností zvýšení osobního nasazení může být atraktivnější nastavení systémů odměn a benefitů, apod.

9.5.1.4 Potřeba role

Potřeba adaptace týkající se jednotlivé role může být vyvolána tím, že tato role v týmu buď chybí úplně, nebo je pokryta nedostatečně z hlediska kvalifikace či počtu pracovníků. Vybral jsem tyto nejčastější role v projektu:

- **Vývojář (Developer)**: vývojář píše kód a vytváří uživatelské rozhraní;
- **Analytik (Analyst)**: analytik provádí správu uživatelských požadavků a analýzu;
- **Systémový architekt (System Architect)**: systémový architekt vytváří návrh architektury systému, návrh softwarové a hardwarové platformy;
- **Dokumentátor (Technical Writer)**: dokumentátor má na starost psaní a správu uživatelské i projektové dokumentace;

Výše uvedené role bývají souhrnně nazývány **tým**. Další role spadají do oblasti vedení.

- **Vedoucí týmu (Team Lead):** vedoucí týmu (v některých metodikách nazývaný „Coach“) je v těsném kontaktu s týmem;
- **Vedoucí projektu (Project Manager)**: vedoucí projektu je zodpovědný za projekt jako celek. Odpovídá za plánování projektu, využívání zdrojů a odpovídající postup prací.

Kromě toho existují ještě speciální role:

- **Doménový odborník (Subject Matter Expert)**: doménový odborník disponuje hlubokou znalostí aplikační domény a poskytuje týmu podporu v této oblasti;
- **Technologický konzultant (Technology Consultant)**: technologický konzultant má typicky specializaci na určitý programovací jazyk, platformu či nástroj.

Jeden pracovník může zastávat i více rolí a na každou roli může být přiděleno nula a více pracovníků.

9.5.2 Nastavení procesu

Proces se vždy týká jak prvků, tak vazeb mezi nimi a zabývá se jejich dynamickým charakterem. Vlastnosti procesu jsem rozdělil pro přehlednost do dvou oblastí:

1. Proces jako celek a jeho charakteristiky;
2. Specifické činnosti a jejich charakteristiky.

9.5.2.1 Proces jako celek

Jako typické jsem zvolil tyto potřeby adaptace procesu:

- **Potřeba krátkosti iterací**: u některých typů projektu příliš dlouhé iterace zhoršují zpětnou vazbu a vytváří nežádoucí zpoždění mezi začátkem projektu a uvedení do provozu;
- **Potřeba inkrementálního vývoje**: inkrementální vývoj znamená, že aplikace bude dodávána postupně po malých funkčních celcích;
- **Potřeba zpětné vazby**: zpětná vazba je třeba pro možnost včasných korekcí a souvisí mj. s délkou iterací;
- **Potřeba adaptibilnosti procesu**: často nestačí pouze nastavit proces a podle původního nastavení realizovat celý projekt. Často je třeba proces vývoje upravovat v průběhu projektu na základě okamžité situace. K tomu dochází především v situacích, kdy není možné přesně predikovat podmínky vývoje.

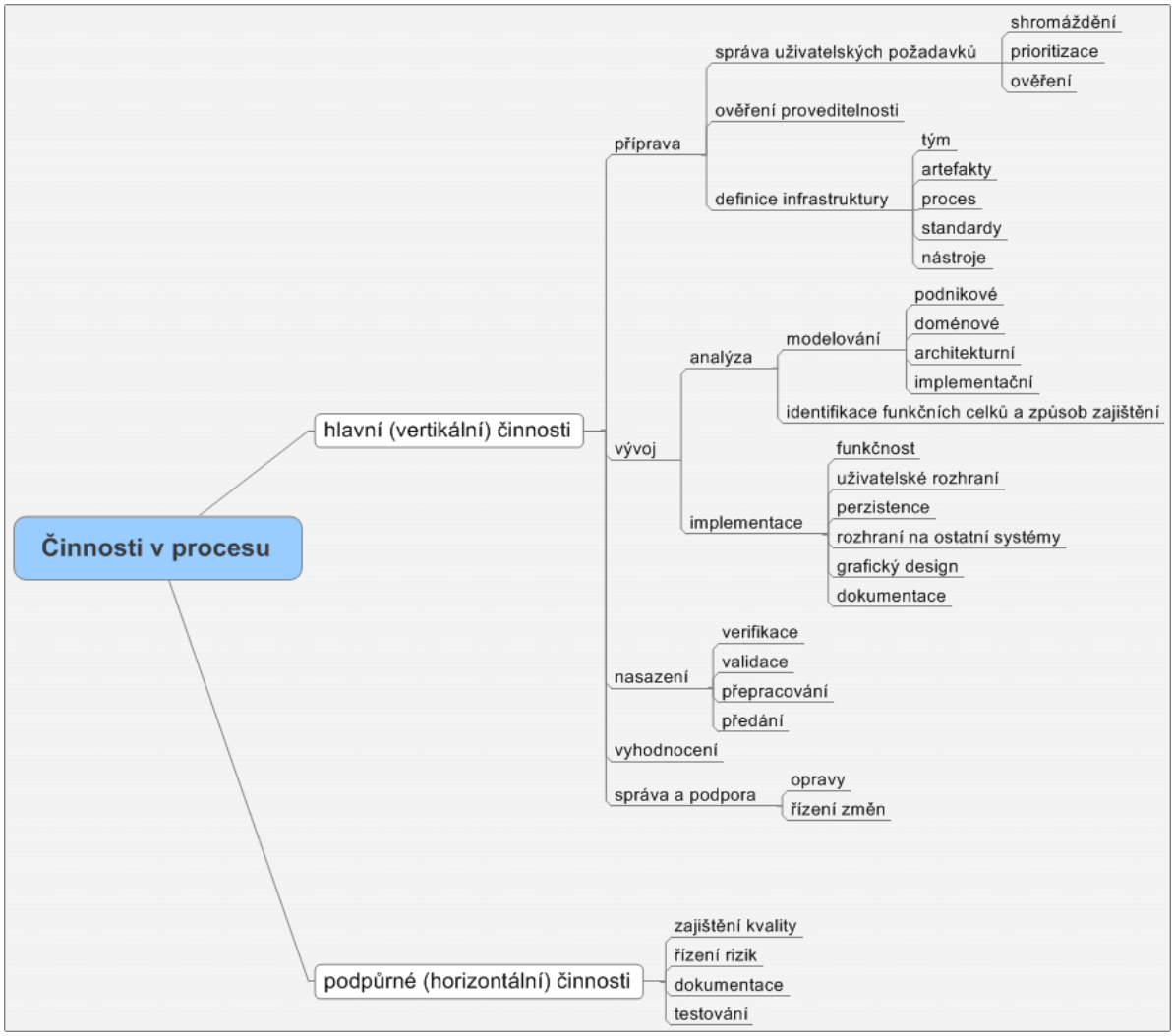
9.5.2.2 Specifické činnosti procesu

Specifické činnosti jsem rozdělil na dvě hlavní kategorie:

- Hlavní (vertikální), což jsou činnosti hlavního produkčního procesu;

- Vedlejší, podpůrné (horizontální) činnosti jdoucí napříč celým projektem.

Všechny činnosti jsem již uvedl v rešerši. Obr. 21 přehledně zobrazuje přehled činností, jež jsem zvolil jako příklad nejčastějších činností, kterých se adaptace může týkat.



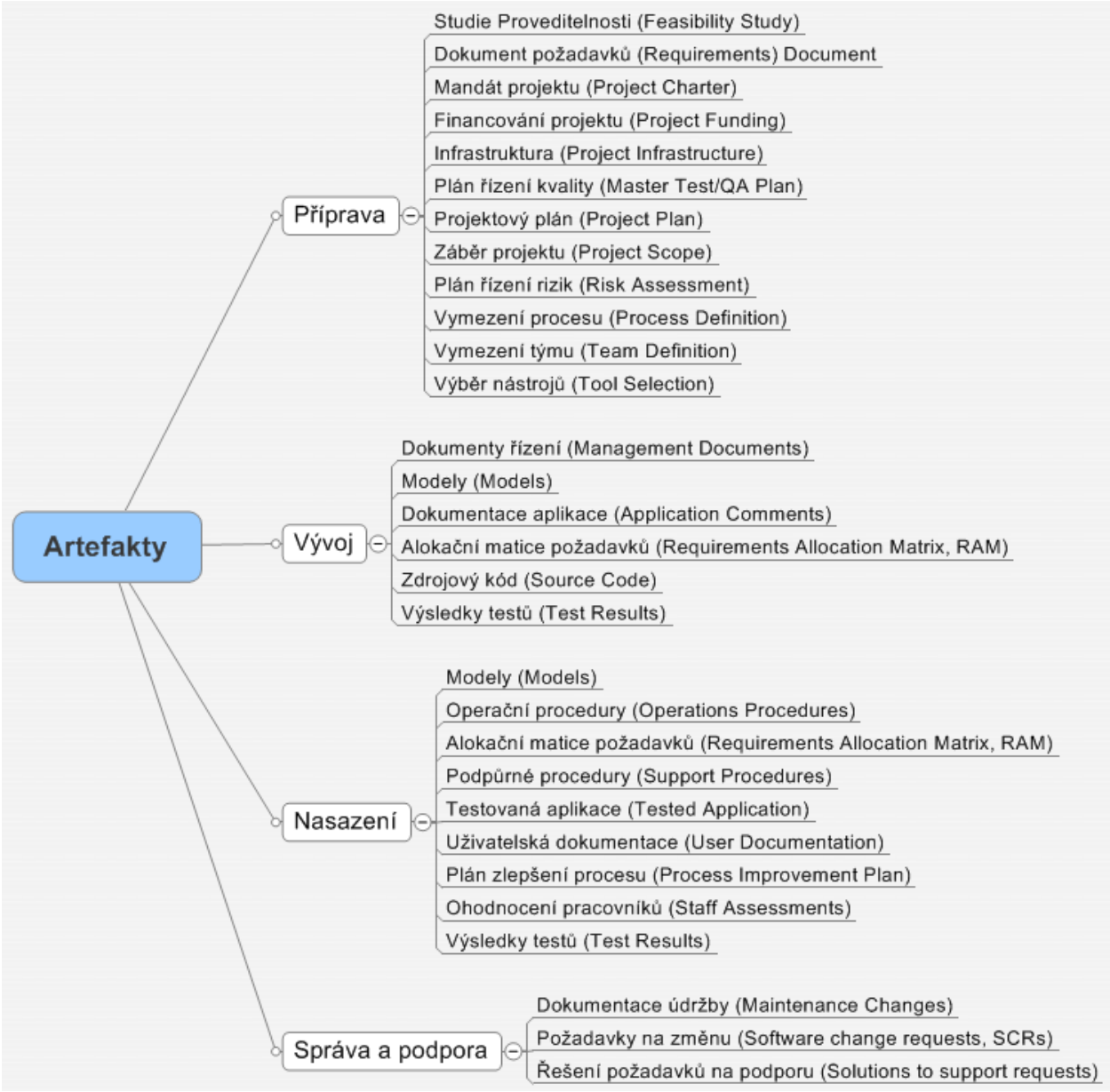
Obr. 21 – Přehled příkladů vybraných činností pro naplnění pomůcky

9.5.3 Vymezení artefaktů

Pod pojmem artefakty máme na mysli veškeré hmotné výstupy a mezivýstupy projektu. Základním artefaktem je systém sám. V průběhu projektu však vznikají i další artefakty charakteru písemného (zprávy, dokumenty), obrazového (náčty, modely), hlasového (záznamy schůzek) i fyzického (fyzický model). Nejvýznamnější jsou dokumenty. Následuje přehled dokumentů, které mohou vznikat v průběhu projektu (viz kap. 6).

Na této úrovni uvažujeme především o artefaktech jakožto prvcích systému, nikoliv o procesu vytvářejícím artefakt, i když tyto činitelé jsou samozřejmě provázány.

Artefakty, jež jsem vybral jako příklad shrnuje Obr. 22.



Obr. 22 – Přehled příkladů artefaktů pro naplnění pomůcky

9.5.4 Nároky na nástroje

Proces tvorby softwaru je možné podpořit celou řadou moderních nástrojů. Adaptace činitele týkajícího se nástroje může být provedena buď úpravou stávajícího nástroje, nebo nákupem nového nástroje. Mechanismus rozhodování o způsobu řešení potřeby je však mimo rámec práce.

Nástroje používané v projektu se dělí na dvě hlavní kategorie:

- Implementační nástroje a platforma;
- Podpůrné nástroje, jež automatizují, usnadňují a podporují provádění činností v průběhu projektu.

Potřeba adaptace může postihovat:

- Množství potřebných funkcí;
- Potřebu ergonomie nástroje;
- Potřebu vyšší úrovně abstrakce (soulad s doménou).

Hlavní nástroje, které mohou být potřebné v projektu jsou:

- Vývojové prostředí;
- Nástroj pro modelování;
- Nástroj generující kód;
- Nástroj pro podporu řízení;
- Nástroj pro správu verzí a sdílení kódu;
- Nástroj pro automatické testován;
- Nástroj pro refaktorizaci;
- Nástroj pro správu dokumentů;
- Nástroje pro podporu týmové komunikace;
- Nástroje pro podporu komunikace se zákazníkem.

9.5.5 Potřeby komunikace a součinnosti

Komunikace a součinnost se týká především vazeb mezi prvky. Potřeba vysoké míry komunikace je zdůrazňována především u agilních metodik, bude tedy důležitým tématem u projektů se zvýšenými nároky na oblasti, na které jsou agilní metodiky zaměřeny, tedy nestabilita a nejasnost zadání (kap. 5.3).

9.5.5.1 Potřeba komunikace uvnitř týmu

Zvýšená potřeba komunikace uvnitř týmu je třeba např. u systémů s nízkou granularitou funkcionality, tj. velkým množstvím malých, bohatě provázaných funkcí.

Míra komunikace má obecně souvislosti se vztahy uvnitř týmu, nastavením procesu, ale i uspořádáním místností a geografickým rozmístěním týmu.

9.5.5.2 Potřeba součinnosti s dodavateli

V současné době nabývají na významu softwarová řešení, jež jsou sestavována z hotových komponent řešících určitou oblast, například SMS brána či autentizační modul. Používání hotových komponent je doporučováno, nicméně může přinést potřebu zvýšené součinnosti s dodavatelem, pokud komponentu není možné použít standardním způsobem.

9.5.5.3 Potřeba součinnosti zákazníka

Potřeba součinnosti zákazníka nabývá na důležitosti především:

- U projektů s nepřesným zadáním;
- U projektů se zadáním, jež se může v průběhu projektu měnit.

9.5.5.4 Potřeba součinnosti s dodavateli zákazníka

V případě, kdy je systém budován jako rozšíření stávajícího systému či je třeba, aby systém spolupracoval s množstvím stávajících systémů, nabývá na důležitosti součinnost s dodavateli zákazníka.

9.5.5.5 Potřeba součinnosti s třetími stranami

Určitý projekt může vyžadovat potřebu součinnosti s třetími stranami pro řešení oborově specifických otázek , jako např. technických, právních, bezpečnostních, apod.

9.6 Postup použití nástroje

Vstupy a činitelé jsou tedy nyní zvoleny. V této podkapitole ukáži, jak pracovat s výslednými diferencemi z definice 5 v kap. 9.2, a jak je vyhodnocovat.

Výsledné diference reprezentují celkové nároky na činitele projektu, jež jsou zmírněny možnostmi substituce. Výsledné diference bude typicky vyhodnocovat vedoucí projektu, podněty však mohou přicházet i od analytiků a od všech ostatních členů týmu. Postup, jak dojít k výsledným diferencím je následující:

1. *Shromáždění potřeb zákazníka*. Shromáždění potřeb běžnými metodami (interview, dotazníky, aj, viz podkapitola 6.2);
2. *Strukturování potřeb zákazníka jakožto vstupů projektu*. V tomto kroku jsou potřeby shromážděné v předchozím kroku analyzovány a strukturovány tak, aby odrážely vstupy vymezené v kap. 9.4;
3. *Analýza nároků*. Bod znamená identifikaci a kvantifikaci funkcí nároků vstupů. Pro všechny kombinace vstupů a činitelů si klademe otázku, jestli určitý vstup vyžaduje adaptaci určitého činitele. Pro činitele, jež v projektu dosud nebyly přitomny či minimálně bude adaptace znamenat zavedení tohoto činitele. V takovém případě je tedy potřeba odhadnout náročnost zavedení činitele v takové úrovni, aby pokrýval nároky vyplývající ze vstupu.

   Jednotlivé výstupní parametry jsou mezi sebou často úzce provázány, zvláště kategorie tým-proces-nástroje navzájem, kdy např. úprava procesu si vyžádá nové role a podpůrné nástroje. Při analýze nároků je tedy vhodné postupovat iterativně;
4. *Výpočet diference činitelů*, jakožto sumy nároků na činitele dle definice 3;

5. *Stanovení substitučních funkcí a celkové substituce.* U činitelů, kde shledáme hodnotu diference vysokou se pokusíme najít substituční vazby od ostatních činitelů, které by umožnily diferenci snížit na základě toho, že místo adaptace činitele využijeme jeho substitut s podobným účinkem. Substituční funkce nakonec pro každého činitele sečteme a obdržíme celkové substituce činitelů dle definice 4;
6. *Výpočet výsledných diferencí činitelů* dle definice 5;
7. *Interpretace výsledků.* Ukazatelem potřeby adaptace činitele jsou nenulové výsledné diference činitelů, jež indikují konečné nároky na adaptaci. Čím je vyšší hodnota diference,, tím je potřeba adaptace vyšší. Nárok má však kvalitativní charakter a nelze ho v pomůcce postihnout. Může dojít i k situaci, kdy dva vstupy budou mít protichůdné nároky na adaptaci stejného činitele. Vysoká hodnota diference indikuje pouze potřebu adaptace a je na vedení projektu, jakým způsobem na tuto potřebu zareaguje, přičemž může nastat i k situace, že žádná akce bude tzv. „nejmenším zlem“.

Výše vyjmenované činnosti se provádí v průběhu celé fáze „Příprava“, ale mohou se upřesňovat i v dalších fázích. Prvotní vyhodnocení potřeb adaptace je prováděno po shromáždění prvních požadavků zákazníka a postupně doplňováno a zpřesňováno.

9.7 Implementace nástroje

Implementaci nástroje jsem provedl v prostředí Microsoft Excel a pracovně jej nazval RIAT: „Requirements Impact Analysis Tool“. Nástroj má podobu tabulky, kde je možné vyplňovat hodnoty nároků a substitucí. Nástroj automaticky počítá hodnoty diferencí, celkových substitucí a výsledných diferencí.

Vytvořený nástroj má podobu excelovského pracovního sešitu s několika listy:

1. Personální otázky (viz podkapitola 9.5.1);
2. Proces (viz podkapitola 9.5.2);
3. Artefakty (viz podkapitola 9.5.3);
4. Komunikace a součinnost (viz podkapitola 9.5.5);
5. Substituce (viz podkapitola 9.6, bod 5).

Listy 1-4 pokrývají matici všech kombinací vstupů a činitelů pro zaznamenání diferencí a list 5 obsahuje matici všech kombinací činitelů pro zaznamenání substitučních vazeb.

V souladu s definicemi a postupem uvedeným v předchozí podkapitole je nástroj používán tímto způsobem:

1. Pro listy 1-4: pro každou kombinaci vstupu a_i a činitele s_j vyplníme hodnotu nároku dem(a_i, s_j). Stačí vyplnit nenulové nároky, tj. číslo 1-10, které udává, jak velký nárok klade vstup a_i na činitele s_j. Je třeba zvážit konkrétní obsah a význam vstupu pro daný projekt a na jeho základě provést expertní odhad. Hodnota 1 znamená drobnou adaptaci, zatímco hodnota 10 znamená potřebu velmi výrazné adaptace;
2. Vyplnění substitučních vazeb na listu 5. Pokud víme, že čitele s_j je možné substituovat činitelem s_k, vyplníme do příslušného políčka sílu subsitutce. 1 znamená velmi slabou substituci, 10 indikuje možnost dokonalé substituce;

3. Na listech 1-4 se u každého výstupního parametru automaticky počítá diference dif(s_j), celková substituce csub(s_j) a výsledná diference vdif(s_j).

Ke každé hodnotě vyplňované do buňky je vhodné přiložit komentář s odůvodněním zvolené hodnoty.

9.8 Kompletní příklad

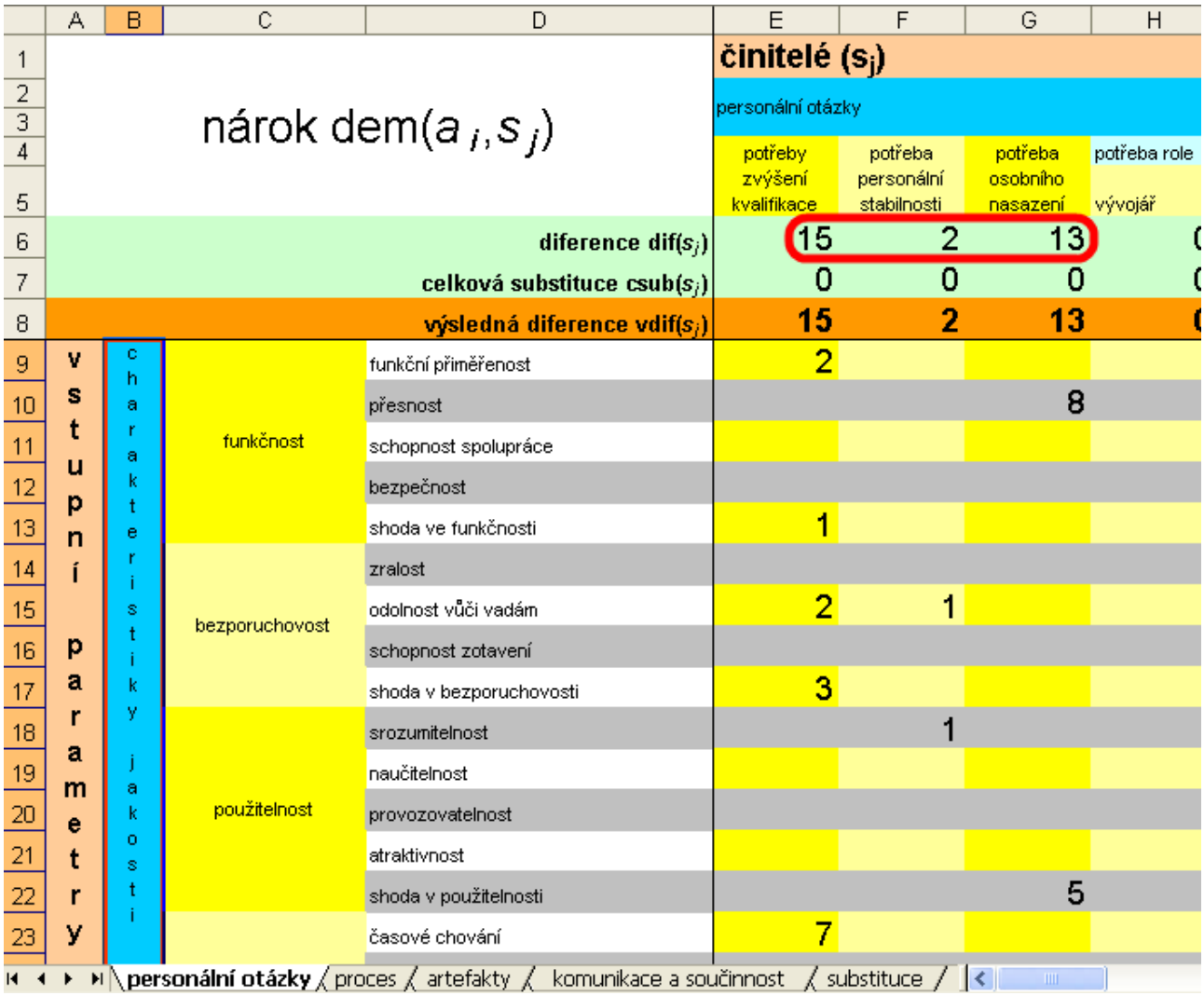
Na následujícím příkladu fiktivního projektu ukáži význam a postup využití pomůcky a nástroje RIAT.

Mějme zákazníka, jímž je fiktivní firma Medicare s.r.o. (dále „zákazník“), jež je významným poskytovatelem nadstandardní lékařské péče. Firma poptává komplexní informační systém pro vedení své agendy. Softwarová firma Megaprogrammers, s.r.o. (dále „řešitel“) byla vybrána ve výběrovém řízení jako řešitel.

Vedoucí týmu řešitele naplánoval několik schůzek s vedoucím projektu zákazníka a zástupci uživatele. Cílem schůzek pro vedoucího projektu je, ujasnit si, jaké bude mít projekt nároky na složení týmu, proces vývoje a součinnost se zákazníkem. Též si musí ujasnit, jak bude třeba nastavit proces vývoje.

Vedoucí týmu využívá při své práci popsanou pomůcku a nástroj RIAT. Z interview (kroky 1, 2 v kap. 9.6) mj. vyplyne, že charakter zadání bude vyžadovat značnou potřebu osobního nasazení týmu pro naplnění požadavku subcharakteristiky jakosti „přesnost“ u charakteristiky „funkčnost“, jelikož tým byl doposud zvyklý na jiný styl práce. Dodržení shody v použitelnosti bude též pro tým náročnější, jelikož zákazník má přesnou představu o použitelnosti budoucího systému. Vedle některých drobných nároků dále vedoucí projektu identifikoval nutnost zvýšení kvalifikace týmu, především z důvodu nutnosti použití sofistikovaných algoritmů pro práci s daty, kde bude třeba vyvinout časově velmi efektivní algoritmy.

Vedoucí projektu tyto skutečnosti kvantifikuje ordinální stupnicí 1-10 a zaznamená do nástroje RIAT na listu „personální otázky“ (krok 3, viz Obr. 23). Nástroj mu ihned ukáže výsledné diference, tj. celkové nároky na jednotlivé vstupní parametry (krok 4).



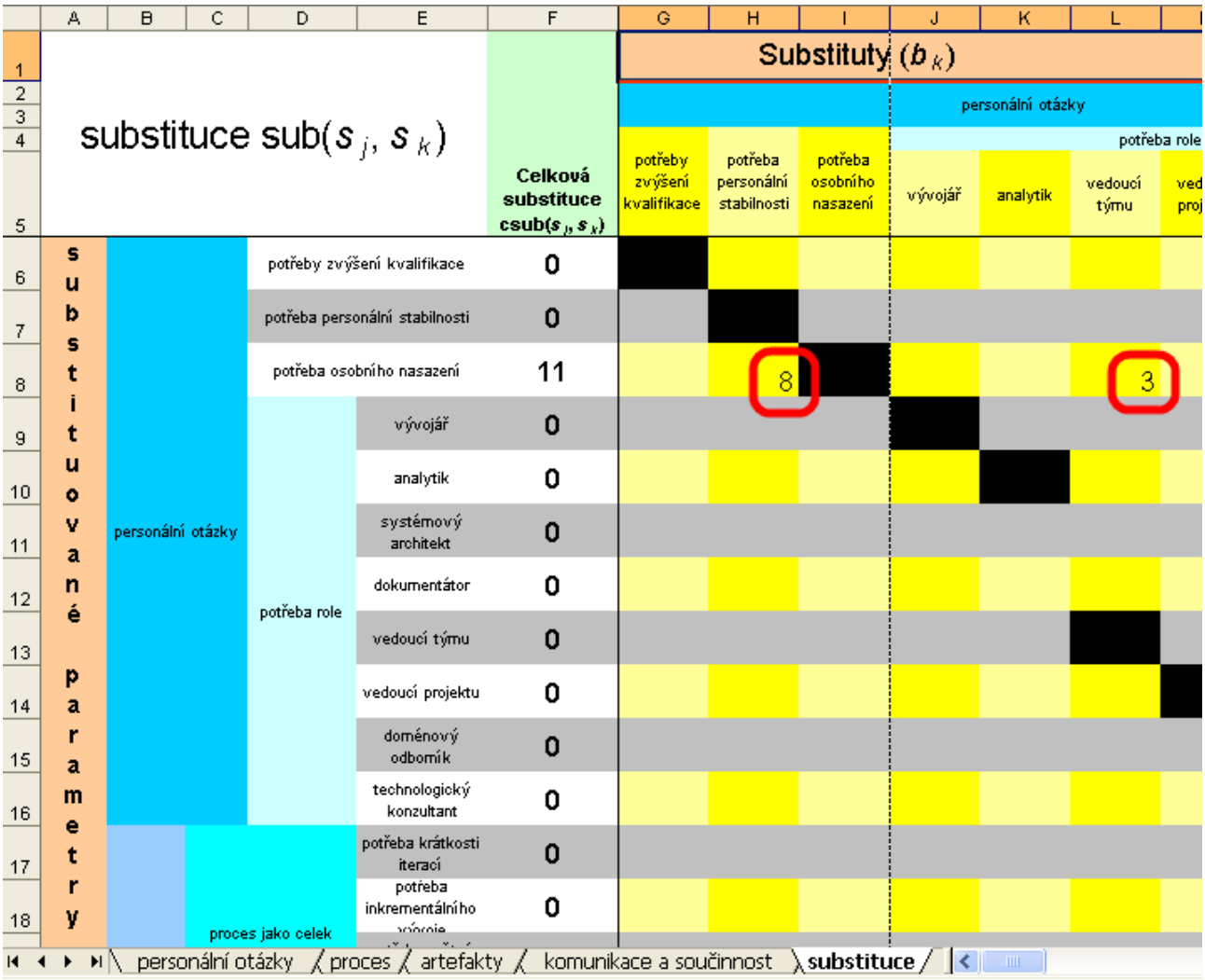
				činitelé (s_j)			
		nárok dem(a_i,s_j)		personální otázky			
				potřeby zvýšení kvalifikace	potřeba personální stabilnosti	potřeba osobního nasazení	potřeba role vývojář
			diference dif(s_j)	15	2	13	0
			celková substituce csub(s_j)	0	0	0	0
			výsledná diference vdif(s_j)	15	2	13	0
vstupní parametry	charakteristiky jakosti	funkčnost	funkční přiměřenost	2			
			přesnost			8	
			schopnost spolupráce				
			bezpečnost				
			shoda ve funkčnosti	1			
		bezporuchovost	zralost				
			odolnost vůči vadám	2	1		
			schopnost zotavení				
			shoda v bezporuchovosti	3			
		použitelnost	srozumitelnost		1		
			naučitelnost				
			provozovatelnost				
			atraktivnost				
			shoda v použitelnosti			5	
			časové chování	7			

personální otázky | proces | artefakty | komunikace a součinnost | substituce

Obr. 23 – Zaznamenání nároků a přehled diferencí

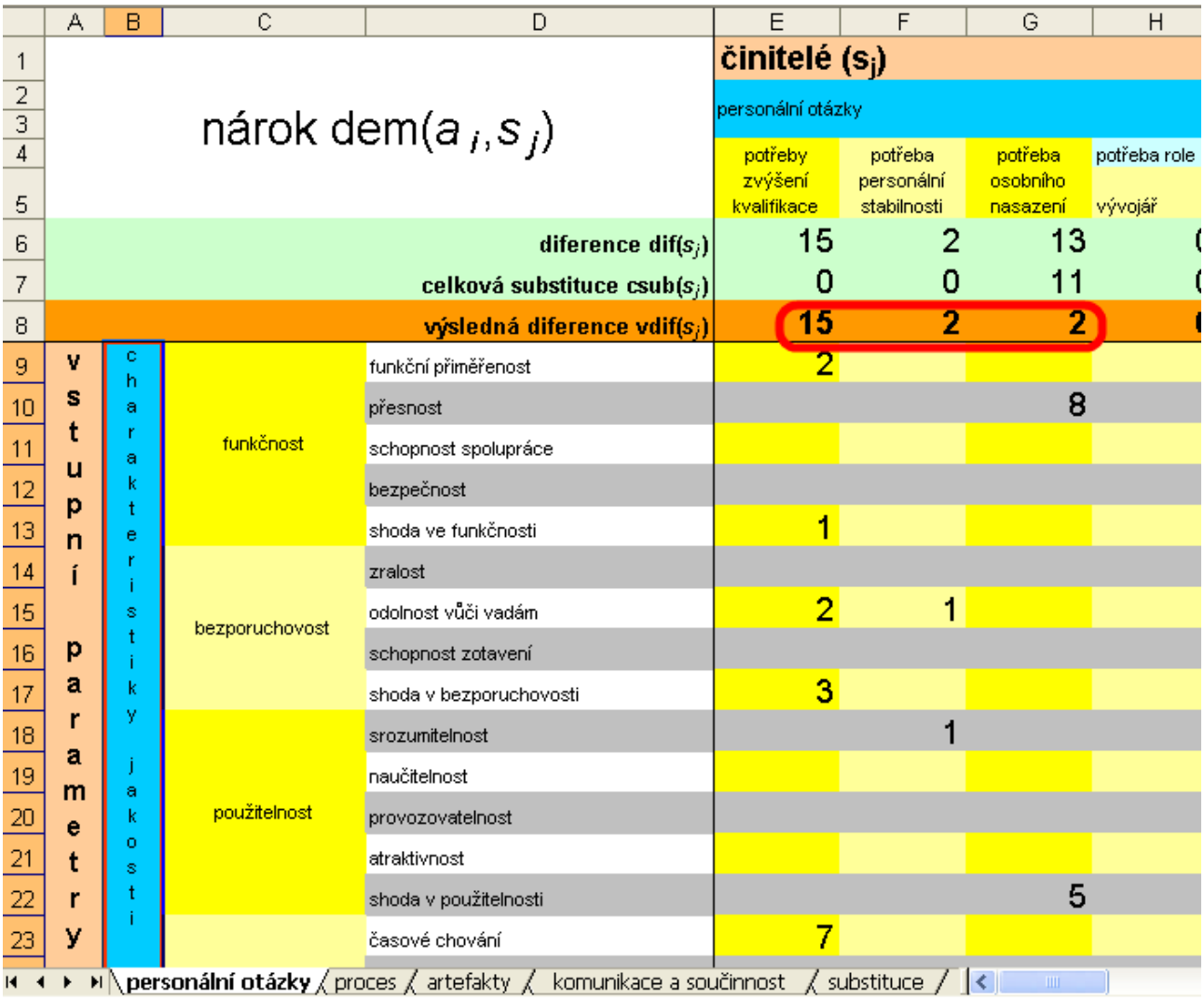
Stejným způsobem bude vedoucí projektu (či analytik) pokračovat ve svých zjištěních a postupně zanášet zjištěné nároky do tabulek.

Poté, co je hotovo prvotní vyplnění listů 1-4, vedoucí se zamýšlí nad diferencemi. Tam, kde jsou diference velké, zkouší najít možné subsituce (krok 5). Substituce zaznamená do listu „substituce“ (Obr. 24). Vedoucí projektu usoudil, že potřeba vysokého osobního nasazení je v tomto případě substituována vysokou personální stabilností týmu a schopností vedoucího týmu tým vydatně motivovat. Přidělí tedy příslušné ohodnocení ze škály 1-10.



Obr. 24 – Substituce

Podobným způsobem vedoucí označí i další možné substituce. Po návratu na list „personální otázky“ vedoucí projektu kontroluje situaci po zaznamenání substitucí (krok 6, Obr. 25).



Obr. 25 – Výsledné diference

Nyní vedoucí přistoupí k interpretaci výsledků (krok 7) a shledá, že celková substituce u parametru „potřeba osobního nasazení“ dosáhla hodnoty 11 a výsledná diference se tím snížila na hodnotu 2. Vyšší výslednou diferenci má „potřeba zvýšení kvalifikace“. Bude tedy třeba zajistit dostatečné školení. V tomto okamžiku by měl vedoucí tuto skutečnost detailnější analyzovat. Možné jsou tyto situace:

1. Vedoucí usoudí, že zajištění dostatečného školení nebude problém;
2. Vedoucí shledá, že požadavky na školení není možné naplnit a pokusí se zákazníkem vyjednat menší nároky. Zde by se nejspíš zaměřil na časové chování, jež se nejvíce podílí na diferenci.

9.9 Poznámky k interpretaci hodnot

Kvantifikace hodnot nároků a substitucí, popsané v předchozích podkapitolách, jsou prováděny na základě expertních odhadů uživatele pomůcky a nástroje RIAT. Přesnost hodnot tedy záleží na schopnosti uživatele posoudit nároky na adaptaci.

Z hlediska stupnice jsem použil ordinální stupnice s rozsahem 0-10. Pro pomůcku jsem jako příklad vybral 32 vstupů a 57 činitelů, získal jsem tak tyto možné rozsahy výsledných hodnot:

diference:	0-320 pro každého činitele (= 32 × 10)
celková substituce:	0-570 pro každého činitele (= 57 × 10)
výsledná diference:	0-320 pro každého činitele

Tab. 7 – Rozsahy kvantifikací

Hodnota výsledné diference se tedy může teoreticky pohybovat v rozmezí 0-320, průměr se však nebude blížit horní hranici. Výsledná hodnota záleží především na přístupu při ohodnocování nároků a může se lišit podle chápání pojmu „zanedbatelný“, který reprezentuje číslo 1 a „velmi vysoký“ reprezentovaný číslem 10. Každý uživatel pomůcky si tak vždy musí vytvořit vlastní pásma pro posuzování hodnoty výsledných diferencí, tj. kdy chápe hodnotu jako „nízkou“ a kdy jako „velmi vysokou“.

Pro jednoduché použití lze pracovat s určitým omezeným počtem pásem, např.

1. Pásmo 1 v rozsahu hodnot výsledných diferencí 0-50 znamená, že není třeba činitele adaptovat;
2. Pásmo 2 v rozsahu hodnot výsledných diferencí 51-250 znamená potřebu adaptovat činitele;
3. Pásmo 3 v rozsahu hodnot výsledných diferencí 251-320 znamená, že nároky na adaptaci jsou příliš vysoké.

9.10 Zhodnocení

Na příkladu pomůcky pro optimalizaci struktury projektu jsem ukázal postup konkrétního využití formálního modelu softwarového projektu. Pomůckujsem vytvořil pro obecné použití, podobným způsobem by bylo možné vytvořit i další pomůcky specializované na určitý druh projektů či metodik.

10 Znalostní metodologická pomůcka

10.1 Úvod

Další příklad, jež jsem vytvořil jako příklad pomůcky vytvořené na základě rozpracování formálního modelu softwarového projektu, je zaměřen na oblast řízení softwarového projektu. Jak bylo vysvětleno v kap. 7.5, metodika řízení softwarového projektu je z hlediska formalizace znalosti sloužící pro řízenou adaptaci projektu. Tato znalost je hierarchická a má řadu úrovní od úrovně celého projektu až po jednotlivé prvky systému. Metodika zachycuje určitou znalost, která se osvědčila a je doporučována. Vytvořením konkrétního modelu této části formálního modelu jsem vytvořil aparát pro zavedení jednoduché znalostní metodologické pomůcky, jež je nástrojem umožňujícím uchovávat znalosti strategického řízení softwarového projektu z různých metodik. Stanovil jsem si tyto hlavní vlastnosti vyvinuté metodologické pomůcky:

5. Možnost záznamu a uchování znalostí, především s ohledem na moderní paradigmata;
6. Využití uchovaných znalostí v praxi;
7. Možnosti rozšiřitelnosti pomůcky;

Z hlediska postupu uvedeného v kap. 8 jsem učinil tyto kroky:

1. Vymezil jsem oblasti obecného modelu k rozpracování: znalost adaptace systému;
2. Specifikoval jsem účelu modelu: tvorba pomůcky pro uchování znalostí řízení softwarových projektů;
3. Vytvořil jsem model:
    a. Vymezil jsem konkrétní vazby a prvky formou datového modelu (podkapitola 10.2);
    b. Vytvořil jsem příklady znalostí na základě rešerše (10.3);
4. Ukázal jsem dvě možné implementace: znalostní mapy (10.4) a databázovou aplikaci (10.5).

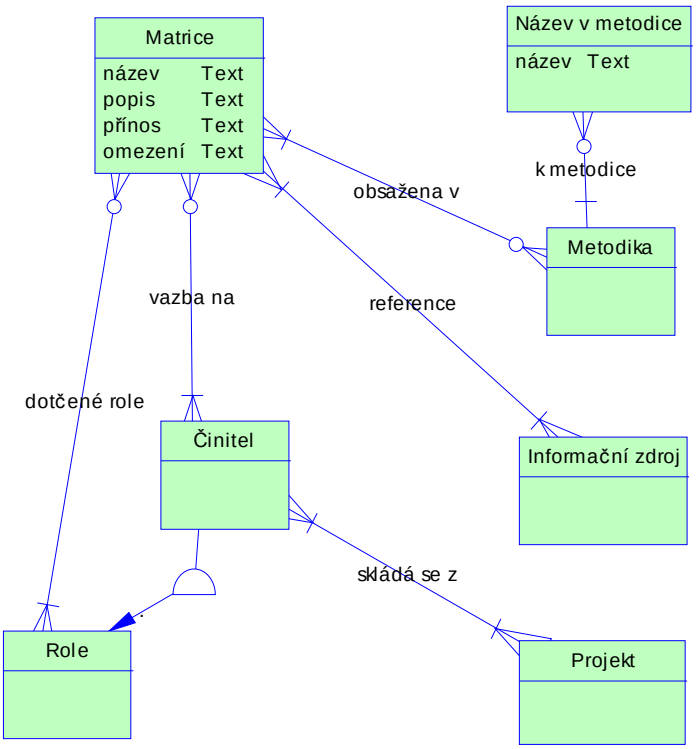
10.2 Datový model

Je třeba formalizovat a definovat znalost řízení softwarového projektu. Tato znalost má dvě roviny:

- Rovinu formální, jež zasazuje znalost do formalizovaného aparátu softwarového projektu;
- Rovinu praktickou, na jejíž úrovni je znalost metodik běžně uchovávána a prezentována. Tato rovina je obecně srozumitelná odborné veřejnosti, formálnost je však nedefinovaná.

Vhodný model znalosti by měl obsáhnout obě tyto roviny. Formalizace znalosti v rovině formální by pracovala pouze s pojmy systémového modelu, tedy s prvky, jejich atributy, vazbami, atributy vazeb a dynamikou systému. Takováto formalizace je sice asi teoreticky možná a zajímavá, nicméně těžko vybudovatelná a pro praxi obtížně použitelná. Z tohoto důvodu jsem se na úrovni znalostní pomůcky spokojil s *poloformální* reprezentací struktury

znalosti, jež specifikuje formální strukturu znalosti, obsah znalosti je však již reprezentován pouze lingvisticky. Schematický entitně-relační datový model znalosti je uveden na Obr. 26



Obr. 26 – Schematický E-R datový model znalosti řízení softwarového projektu

Jedná se o konceptuální datový model, ve kterém nejsou zachyceny implementační detaily. Entity bez atributů nejsou pro tento příklad podstatné a jsou přítomny kvůli vazbám. V nejjednodušším případě budou v implementaci obsahovat pouze název, ale mohou být implementovány i jako složitější množina dalších entit. Rozhodnutí opět záleží na konkrétních potřebách.

Hlavní prvky struktury znalosti jsou tyto:

- **Název:** název znalosti;
- **Metodika:** metodika či metodiky, ze které je znalost vytvořena;
- **Název v metodice:** název, pod kterým je znalost v metodice známa;
- **Činitelé**: činitelé, kterých se znalost týká;
- **Popis**: obsah znalosti, tj. neformalizovaný lingvistický útvar v podobě textu;
- **Přínos**: výhody a zlepšení, která lze očekávat použitím znalosti;
- **Omezení**: předpoklady a omezení při použití znalosti;
- **Reference**: odkazy na literaturu, ze které je možné doplnit obsah znalosti.

10.3 Příklady znalostí

Dále uvedu několik příkladů znalostí sestavených na základě rešerše v kap. 5.3. Název znalosti v metodice bude vždy uveden v závorce za metodikou.

Název: **Častá integrace**

Metodika: SCRUM (Daily builds), XP (Nepřetržitá integrace)

Činitelé: řízení kvality, řízení rizik

Popis: V krátkých intervalech sestavit a odzkoušet aktuální verzi systému. XP doporučuje integraci provádět několikrát denně, SCRUM jednou denně. Konkrétní interval je třeba nastavit především podle technologických omezení.

Přínos: Při denních sestaveních se včas ukáží problémy a konflikty v rozhraní. Rovněž tak se projeví nejednotné chápání aplikace. Tato praktika minimalizuje riziko zahození velkého objemu práce.

Omezení: Častou integraci je možné provádět jen pokud její režie nepřesáhne únosnou mez. Hlavním předpokladem jsou vhodné technologie. Dále je nutné, aby tým byl schopen efektivně a okamžitě komunikovat o konfliktech vznikajících při integraci.

Vazby: programátoři, testeři

Reference: [Schwaber et al. 2001], [Beck 2002a]

Název: **Sprint**

Metodika: SCRUM (Sprint), XP (Malé verze)

Činitelé: řízení projektu

Popis: Sprint je název pro jednu iteraci, která v metodice SCRUM trvá jeden měsíc. Během Sprintu tým pracuje na zadání, které má za cíl během měsíce přinést co nejvíce nejžádanějších funkcí. Řízení Sprintu je založeno na empirickém řízení a samoorganizaci týmu. Organizaci práce během sprintu je možné nastavit dle metodiky XP

Přínos: Krátké měsíční iterace poskytují dobrou zpětnou vazbu.

Omezení: Tým musí být motivován a složen ze samostatných odborníků.

Vazby: tým, vedení projektu, zákazník

Reference: [Schwaber et al. 2001], [Beck 2002a]

Název: **Denní schůzka**

Metodika: XP, SCRUM (Scrum Meeting)

Činitelé: řízení projektu

Popis: Týmový vedoucí (Coach, SCRUM Master) pořádá každý den krátké schůzky s přesně vymezenou dobou trvání (10 minut až půl hodiny), kde každý člen týmu dostane prostor ostatním sdělit:

- Co za poslední den udělal;
- Na čem bude další den pracovat;
- Jaké problémy a nejasnosti mu vznikly.

Doba schůzky nesmí být překročena, pro řešení otázek slouží workshopy, nikoliv denní schůzka.

Přínos: Zpětná vazba pro vedoucího týmu. Podpora informovanosti členů týmu. Možnost eskalace problémů.

Omezení: Realizovatelné jen u menších týmů. V případě geografického rozmístění týmu je nutné zajistit vhodnými technickými prostředky virtuální účast (konferenční hovor, videokonference).

Vazby: tým, vedení týmu

Reference: [Schwaber et al. 2001], [Beck 2002a]

Název: **Empirické řízení**

Metodika: XP, SCRUM

Činitelé: řízení projektu

Popis: Plánování je dynamické a založené na postupném ujasňování potřeb. Podmínkou jsou krátké iterace.

Přínos: Vedoucí pracovníci mohou dynamicky stanovovat priority. Rychlé viditelné výsledky po první iteraci. Po každé iteraci je možné se rozhodnout o pokračování či ukončení.

Omezení: Není možné dopředu stanovit pevný časový harmonogram a rozpočet.

Vazby: vedení týmu, vedení projektu, tým

Reference: [Schwaber et al. 2001], [Beck 2002a]

Název: **Mentální model systému pro tým**

Metodika: XP (Metafora)

Činitelé: komunikace, řízení rizik, řízení kvality

Popis: Tým si utvoří mentální model, pomocí kterého bude komunikovat o systému. Vhodnou kostrou programátorského mentálního modelu jsou návrhové vzory, nicméně je možné o systému komunikovat libovolnou metaforou, která je blízká všem členům týmu.

Přínos: Jednotný mentální model vede k těmto pozitivním rysům:

1. Usnadnění vzájemného pochopení;
2. Zrychlení komunikace;

3. Omezení rizika nedorozumění;

Omezení: Mentální model musí být sdílen všemi členy týmu.

Vazby: tým, vedení týmu

Reference: [Beck 2002a]

Název: **Mentální model systému pro zákazníka**

Metodika: XP (Metafora)

Činitelé: komunikace, řízení rizik, řízení kvality

Popis: Řešitel nabídne zákazníkovi metaforu, pomocí které budou navzájem komunikovat o systému. Metafora musí být blízká oběma stranám. Nabízí se tyto možnosti:

1. Metafora z oboru činnosti zákazníka, pokud se řešitel v oboru pohybuje. V tomto případě se může metafora až přímo překrývat s doménou;
2. Metafora z obecně známého oboru (např. faktury, objednávky, ...);
3. Metafora přímo z oboru informačních systémů, pokud se zákazník v tomto orientuje. Metafora pak může být podobná či přímo stejná jako metafora týmu;

Přínos: Jednotný mentální model vede k těmto pozitivním rysům:

4. Usnadnění vzájemného pochopení;
5. Zrychlení komunikace;
6. Omezení rizika nedorozumění.

Omezení: Mentální model musí být sdílen všemi členy týmu.

Vazby: zákazník, vedení projektu

Reference:

Název: **Zaměření na současnou situaci**

Metodika: XP (Jednoduchý návrh)

Činitelé: řízení projektu

Popis: Návrh a implementace jsou prováděny s ohledem na současné potřeby. Snaha vytvořit co nejjednodušší návrh realizující současné potřeby bez ohledu na představy o budoucích rozšířeních.

Přínos: V případě dynamicky se měnícího zadání se minimalizuje zbytečná práce v případě změn budoucích potřeb.

Omezení: Při nedostatečné flexibilnosti vývojové platformy hrozí problémy při realizaci budoucích potřeb.

Vazby: vedení projektu

Reference: [Beck 2002a]

Název: **Vývoj řízený testováním**

Metodika: XP (Testování), Test Driven Development

Činitelé: vývoj, řízení kvality

Popis: Vývoj řízený testováním určuje, že programátoři nejdříve píší testy a poté až kód. Testy by takto měly být předem napsány pro všechny důležité (a testovatelné) funkce systému. Testy jsou poté pravidelně spouštěny při potřebě prověření konzistence a funkčnosti celého systému (např. po refaktoringu). Pro podporu vývoje řízeném testováním jsou používány automatizované nástroje pro spouštění a vyhodnocování testů.

Přínos: Psaní testů před vlastním kódem nutí programátora řádně si ujasnit vstupy, výstupy a význam funkce. Test objektivně určuje, že funkce funguje (z hlediska požadavků zanesených v testu). Spuštěním všech testů je možné ověřit zachování funkčnosti systému po citlivém zásahu do něj.

Omezení: Disciplína programátorů psát testy před realizací kódu. Potřeba dovednosti v psaní testů. Potřeba nástroje pro automatizované spouštění a vyhodnocování testů.

Vazby: programátoři

Reference: [Beck 2002a], [Beck 2002b]

Název: **Refaktorizace**

Metodika: praktika nezávislá na metodice; integrální součástí XP

Činitelé: vývoj, řízení rizik, řízení kvality

Popis: Refaktorizace je změna systému, jež nemění jeho chování, ale vylepšuje některé charakteristiky přímo nesouvisející s funkčností systému, např. jednoduchost, pružnost, pochopitelnost, výkonnost. K refaktorizaci se tým uchyluje, kdykoli je zjištěna její potřeba. Potřeba refaktorizace je zvažována před i po přidání každé funkce do systému.

Přínos: Refaktorizace zvyšuje udržovatelnost a rozšiřitelnost systému.

Omezení: Refaktorizace vyžaduje dovednost. Bez nástroje pro podporu může být rozsáhlá refaktorizace časově neúnosná.

Vazby: tým

Reference: [Fowler et al. 1999]

Název: **Párové programování**

Metodika: XP

Činitelé: vývoj, komunikace, řízení rizik, řízení kvality

Popis: Veškerý produkční kód (tj. kód, který je součástí dodávaného systému) píší společně dva programátoři sedící u jednoho počítače. Ten, který v daný okamžik třímá klávesnici a myš, píše kód a řeší aktuální problém. Druhý programátor kontroluje správnost kódu a přemýšlí o celkové koncepci. Důležité je, aby šlo o vyrovnaný dialog a ne jen o koukání přes rameno. Pro tvorbu párů nejsou stanovena žádná pevná pravidla. Seskupení do párů je dynamické a může se i v průběhu dne měnit.

Přínos: Párové programování podstatným způsobem snižuje chybovost programátorské práce a zvyšuje produktivitu (dle některých studií i více než dvojnásobně). Programátoři v páru se navíc hlídají, aby nesklouzávali k programátorským zlozvykům (např. nedodržování štábní kultury).

Omezení: Párové programování vyžaduje dovednosti ve spolupráci. Členové páru musí být sehraní.

Vazby: programátoři

Reference: [Beck 2002a]

Název. **Společné vlastnictví kódu**

Metodika: XP

Činitelé: vývoj, komunikace, řízení rizik

Popis: Všichni mohou měnit jakoukoli část systému v jakoukoli dobu. Všichni jsou tedy obeznámeni se všemi částmi systému, ačkoli na různě úrovni. Jestliže programátor vidí příležitost ke zlepšení části zdrojového textu, začne jej zdokonalovat, pokud mu to usnadní život. Všichni tedy přijímají zodpovědnost za celý systém.

Přínos: Systém je stále udržován v konzistentním stavu a konflikty a nejednotné chápání aplikace se ihned projeví. Minimalizuje se riziko spojené s personálními fluktuacemi týmu.

Omezení: Společné vlastnictví kódu je považováno za nebezpečnou praktiku a přináší řadu rizik, které je třeba řádně ošetřit vhodnými opatřeními a praktikami. Toto se týká především nebezpečí konfliktů. Z tohoto důvodu je o této praktice vhodné uvažovat, pokud je možné systém rozdělit na dostatečně disjunktní části (více viz reference). Kód je třeba psát tak, aby z něj byl zřejmý záměr (tzv. *Intention-revealing*). Je proto vhodné používat jazyky co nejvyšší úrovně (z univerzálních jazyků nejlépe Smalltalk, Eiffel, atd.). Společné vlastnictví navíc vyžaduje zkušený tým se širokými znalostmi.

Vazby: programátoři

Reference: [Beck 2002a]

Název: **Zákazník na pracovišti**

Metodika: XP

Činitelé: komunikace, řízení rizik, řízení kvality

Popis: V týmu je přítomen skutečný zákazník, který je k dispozici na plný úvazek a odpovídá na otázky týmu, řeší spory a určuje priority v menším měřítku. Tímto zákazníkem by měl být jeden z budoucích uživatelů systému nebo

alespoň někdo, kdo přesně zná potřeby skutečných uživatelů *(Customer Proxy)*. Pokud není fyzická přítomnost zákazníka možná, je možné alespoň domluvit nepřetržitou telefonickou dostupnost a časově flexibilní meetingy.

Přínos: Přítomnost zákazníka eliminuje riziko nepochopení zadání a ulehčuje plánování a analýzu.

Omezení: Personální / finanční zátěž pro zákazníka.

Vazby: tým, zákazník

Reference: [Beck 2002a]

Název: **Mise projektu**

Metodika: ASD

Činitelé: řízení projektu, komunikace

Popis: Mise projektu spočívá ve vypracování projektové vize (charter). Úkolem projektové vize je specifikovat cíle projektu a hlavní pilíře, na kterých stojí jeho úspěšná realizace. Mise projektu nespecifikuje cesty, kterými se projekt bude ubírat, ale vytváří určité hranice. Jeho hlavním smyslem je vytvořit vizi, kterou bude tým sdílet a ke které se zaváže (commit) a která se stane určitou vodicí linií.

Přínos: Misi projektu je vhodné vypracovat u náročných komplexních projektů, kde navíc očekáváme komplikace při realizaci a působení řady negativních a rušivých vlivů. V takových případech je vhodně vytvořená a artikulovaná mise projektu silou, která může pomoci projekt zdárně vyvádět ze slepých uliček a udržovat výkon.

Omezení: Mise projektu se nesmí stát formalitou. Mise projektu je velmi specifická pro každý projekt a vytvořit a zavést efektivní misi projektu je velmi náročné a vyžaduje mezioborové znalosti (psychologie, sociologie) a cit.

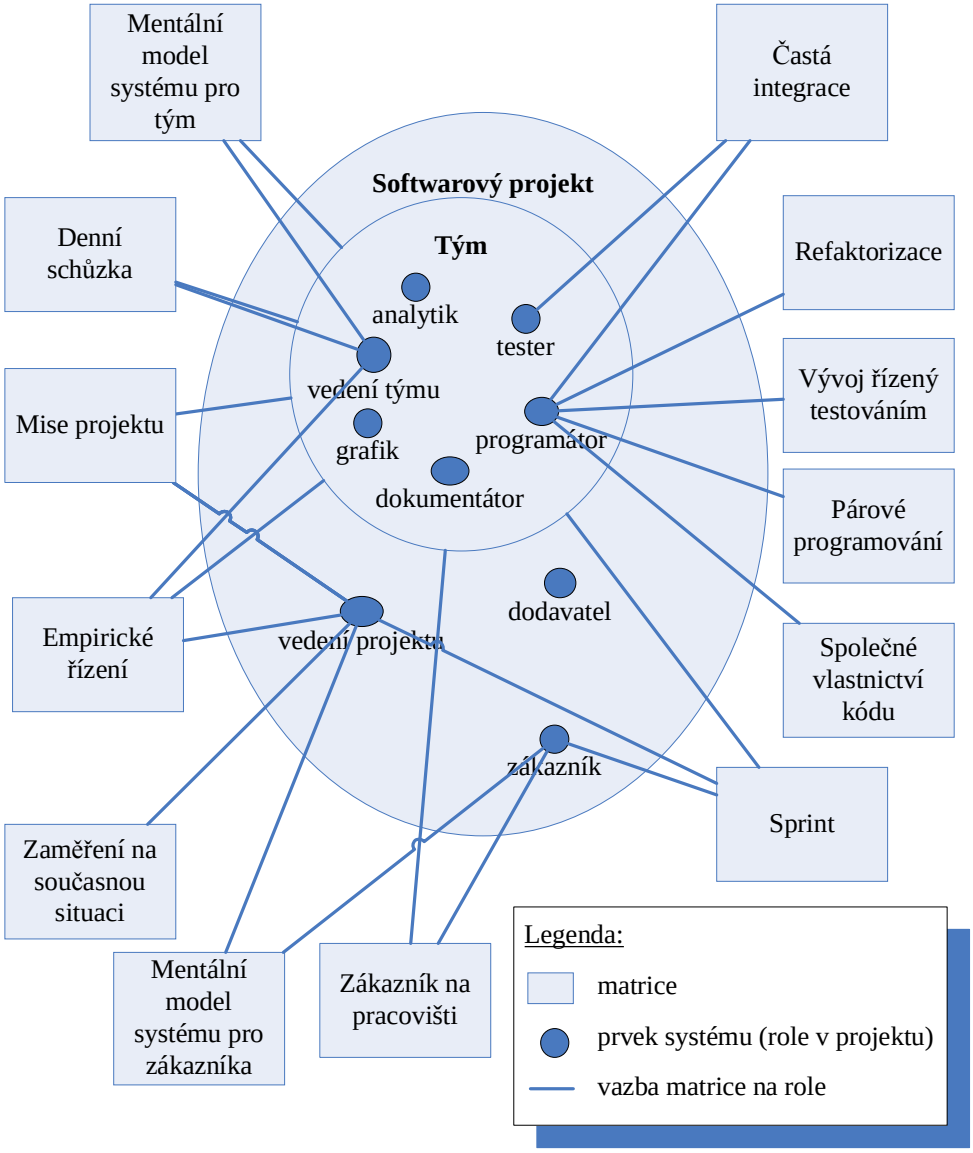
Vazby: vedení projektu, tým

Reference: [Highsmith 1999]

10.4 Implementace znalostními mapami

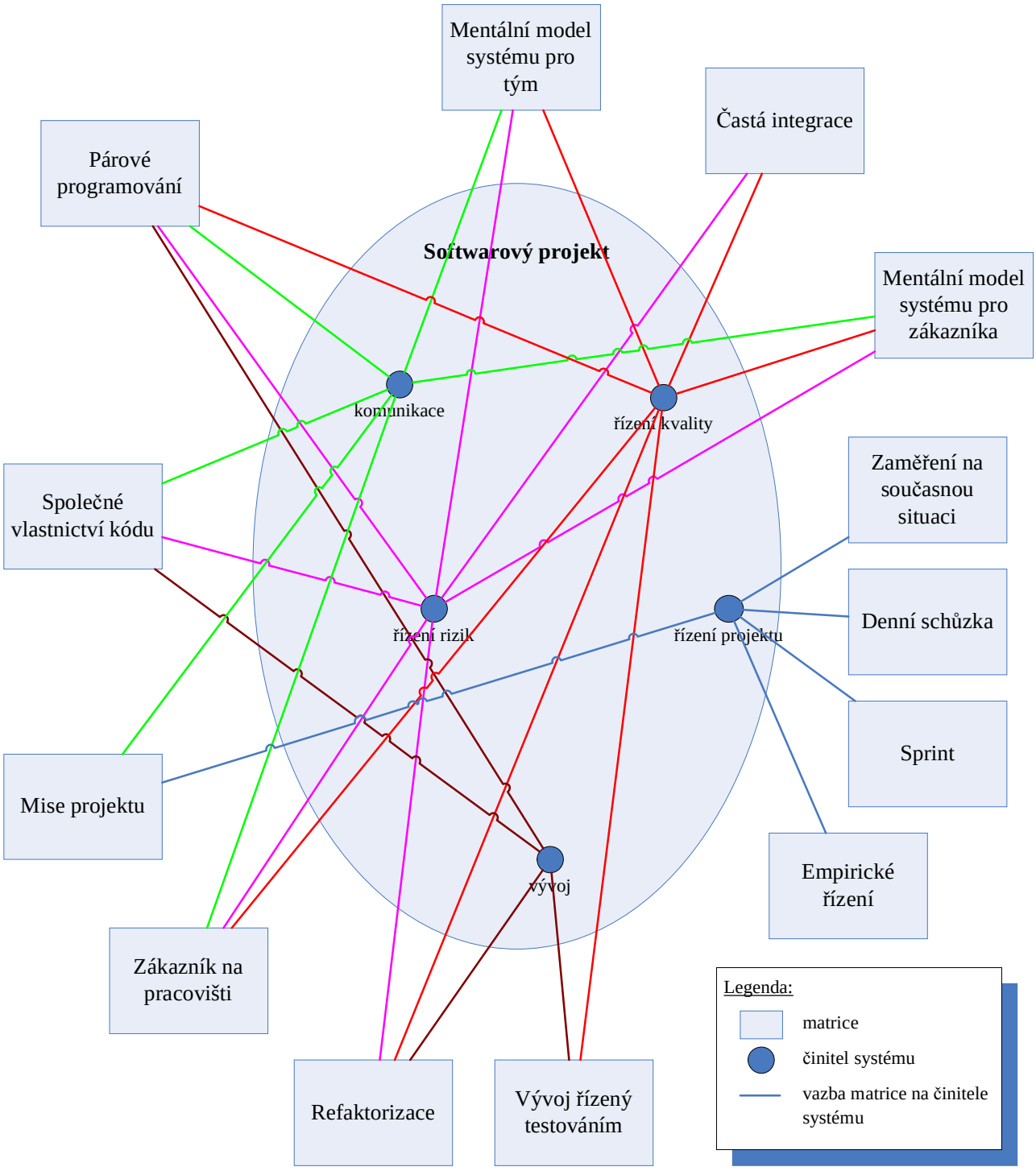
Znalostní mapy jsou poměrně mladým a rozvíjejícím se tématem. Pro potřeby této práce chápu znalostní mapu v souladu s [Havlíček, Pelikán 2007] jako sjednocení tří objektů: nosiče *S*, topologie *G* a textu *T*. Nosič znalostní mapy může být rovinný útvar (obdélník, čtverec, atd.) i celá rovina. Topologie specifikuje vlastnosti tvaru a polohy geometrických útvarů obsažených v mapě. Textové řetězce mohou, ale nemusí být přiřazeny k některému geometrickému objektu z topologie (řetězec incidentní s topologií, resp. řetězce volné).

Ve znalostní mapě lze vizualizovat především vazby mezi znalostmi a systémem.



Obr. 27 – Znalostní mapa znázorňující vazby znalostí na role v systému

Obr. 27 znázorňuje ukázku znalostní mapy, jíž jsem navrhl pro vizualizaci vazeb znalostí na role (prvky) v systému. Pomocí této znalostní mapy je možné určovat v jakých znalostech participuje která role v systému a opačně, které role jsou konkrétní znalostí zasaženy. T akovéto informace lze využít například při přípravě plánů školení.



Obr. 28 – Znalostní mapa znázorňující příslušnost znalosti k činitelům systému

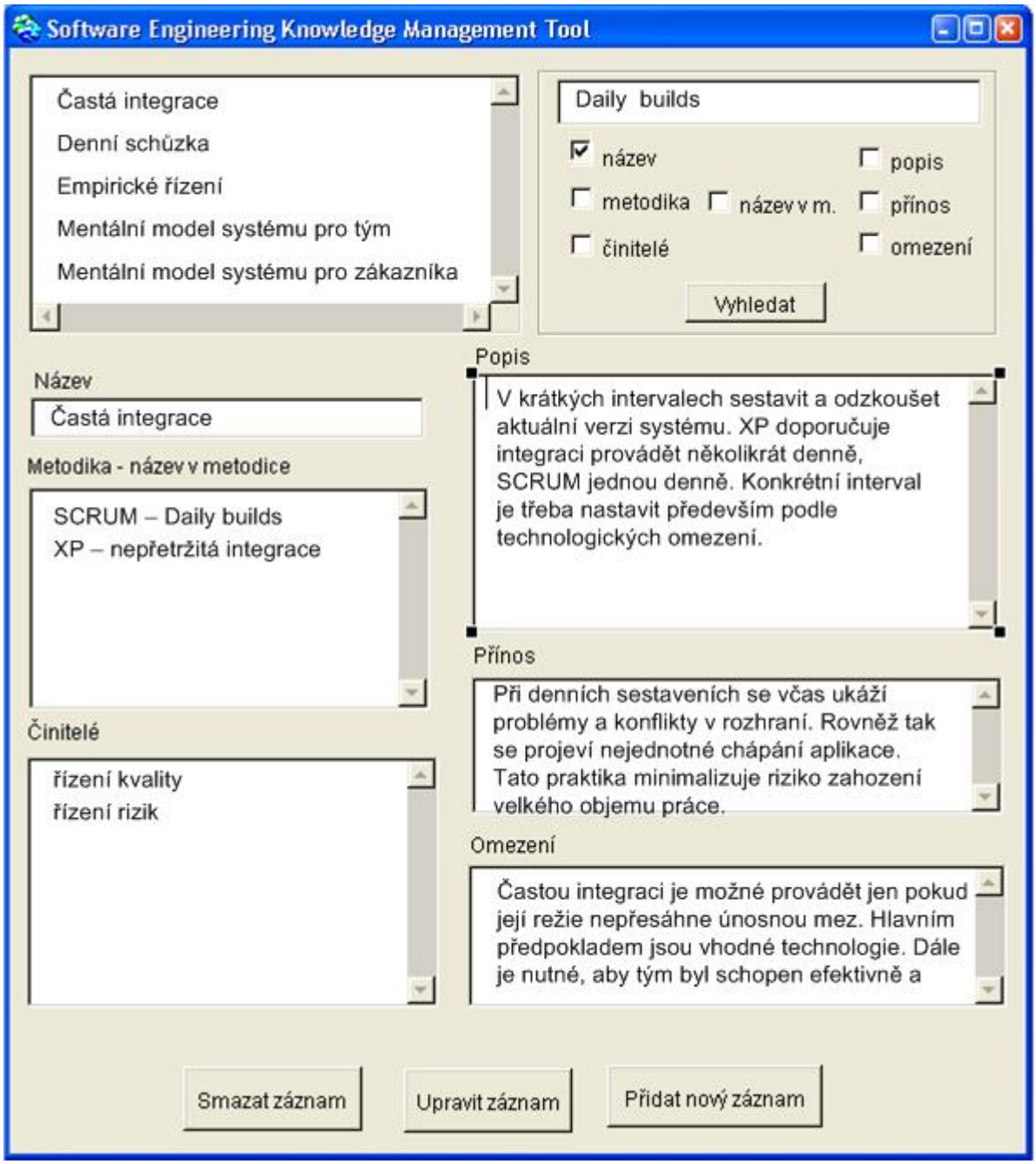
Další příklad mapy jsem navrhl pro znázornění vazeb mezi znalostí řízení softwarového projektu a vnitřkem systému: vazby na činitele v systému[9]. Tuto znalostní mapu lze využít například při řešení otázky „Které znalosti by mohly podpořit řízení kvality?“ a podobných otázek.

[9] Různých barev vazeb je použito jen pro lepší grafickou přehlednost.

10.5 Implementace znalostní databází

Znalosti je možné uložit do znalostní databáze a pomocí této databáze využívat znalosti v nich obsažené. Výhodou databáze je především snadná navigace a možnosti vyhledávání podle různých kritérií (dotazů). Struktura databáze není složitá, je možné ji implementovat relační i objektovou technologií.

Vytvořil jsem jeden ukázkový návrh aplikace, jež může sloužit pro práci s databází znalostí. Je uveden na Obr. 29. Navržená aplikace umožňuje prohlížení databáze znalostí a vyhledání v ní. Možné je též znalosti upravovat, přidávat a mazat. Informace v ní obsažené lze použít např. k úpravě (obohacení) stávající metodiky či k výběru nové metodiky.



Obr. 29 – Návrh aplikace pro práci se znalostní metodologickou pomůckou

10.6 Zhodnocení

Druhý příklad využití formalizace softwarového projektu jsem zaměřil na konkretizaci znalostí, tedy struktur modelujících znalosti řízení projektu. Na příkladu této pomůcky jsem též ukázal, že implementace nástroje nemusí mít vždy jednu podobu a nemusí ani nutně být ve formě softwaru.

11 Závěr

11.1 Shrnutí a diskuse

V rešerši jsem představil současnou praxi řízení softwarových projektů v kontextu historických kořenů. Věnoval jsem se klasickým (tvrdým) i moderním (měkkým) paradigmatům a vysvětlil jejich přednosti i omezení. Zmapoval jsem též nejvýznamnější snahy o formalizaci v disciplíně softwarového inženýrství.

Na základě těchto znalostí jsem posléze sestavil strukturovaný přehled procesu vývoje softwaru, jež je nutný pro uchopení celého rozsahu problematiky.

Poté jsem přistoupil k formalizaci softwarového projektu. Kladl jsem důraz na obecnost modelu, jelikož existující snahy o formalizaci shrnuté v rešerši mají společné to, že pomocí formalismu řeší určitou úzkou oblast problematiky v oboru (například odhad složitosti softwarového projektu) a jsou tudíž pouze omezeně využitelné pro jiné oblasti.

Jako formální model softwarového projektu jsem zvolil velmi obecnou formalizaci softwarového projektu pomocí systémového modelování. Výsledek ukázal, že existuje celkem výstižná analogie mezi pojmy systémového modelování a softwarovým projektem. Ačkoliv vytvořený formální model je poměrně jednoduchý, podařilo se mě pomocí něj formalizovat některé těžko uchopitelné pojmy, jako je například agilnost softwarového projektu a různé přístupy k projektovému řízení. Díky začlenění některých pojmů a principů z oblasti komplexních adaptivních systémů se mě podařilo dotknout i těžko formalizovatelné oblasti psychologicko-sociální, jež je přítomna v každém projektu.

V práci jsem též ukázal, že na softwarový projekt je možné nahlížet jako na výpočetní systém. To vede k vytvoření relace mezi vstupy (požadavky) a výstupy (části softwarového produktu, dokumentace, technického vybavení a další součásti řešení). Toto přináší formální vazbu mezi zadáním a řešením, která souvisí s mnoha teoretickými i praktickými problémy, např.:

- Ověření, že všechny požadavky byly implementovány, jsou zdokumentovány a technicky zrealizovány;
- Identifikace kódu, částí dokumentace, apod., které nerealizují žádný požadavek;
- Identifikace částí zasažených změnami při změnovém řízení.

Důležitým mezivýsledkem je kap. 7.5.3, jež se zabývá agilností softwarového projektu. V podmínkách neurčitosti a změny zadání typických pro většinu současných projektů je agilnost důležitým pojmem. Tento pojem je však v literatuře specifikován značně vágně a intuitivně, což umožňuje vytvořit si pouze určitý pocit o tom, co je to „míra agilnosti“. V kap. 7.5.3 ukazuji vlastní možný formální model agilnosti, jež umožňuje agilnost softwarového projektu měřit, studovat, vyhodnocovat a činit podložené závěry a predikce.

Bylo by zajímavé se pojmem agilnosti dále zabývat a kvantifikovat agilnost způsobem ukázaným v příkladu v kap. 7.5.3.1 u řady různých projektů a studovat její průběh. Domnívám se, že by bylo takto možno objevit řadu jevů, jež by pomohly lépe pochopit zákonitosti řízení softwarových projektů. Jedním z konkrétních přínosů je např. možnost sledovat na základě průběhu celkové reziduální adaptace, jak je systém „zdravý“, a včas zjišťovat varovné signály blížícího se zhroucení.

Jelikož agilnost projektu je možné posuzovat pouze jako reakci na změny, zavedl jsem navíc pojem schopnost agilnosti, jež charakterizuje schopnost projektu býti agilní. Zajímavé by v tomto kontextu bylo, pokusit se vypracovat metodu výpočtu schopnosti agilnosti na základě určitých ukazatelů, jako např. použitá metodika, struktura týmu, používané nástroje, apod. Z toho by bylo možné poté vypracovat analýzu agilnosti ex-ante při uvážení předpokládané míry proměnlivosti zadání a posoudit tak proveditelnost projektu.

Práce je zaměřena především na vedení projektů využívajících moderní paradigmata, tedy především agilní metodiky. Zde existuje diskuse o vhodnosti či nevhodnosti využití agilního přístupu pro různé typy projektů a týmů. Práce přináší do této diskuse nový příspěvek: na základě vytvořené formalizace ukazuji v kap. 7.5.2, že důležitou podmínkou úspěšného použití agilních metodik se jeví schopnost samoorganizace softwarového projektu v souladu s teorií komplexních adaptivních systémů. Tento poznatek by bylo zajímavé hlouběji ověřit a rozpracovat. V konečném výsledku by mohl takto vzniknout příspěvek k posuzování vhodnosti využití metodiky pro konkrétní projekt.

Vytvořený formální model slouží též k tvorbě metodologických pomůcek. Velká výhoda vytvořených pomůcek spočívá v zajištění vzájemné konzistence. Tím, že pomůcky vychází ze společného formálního modelu, zachovávají společnou obecnou terminologii a je možné pomůcky kombinovat. Pomůcka pro optimalizaci struktury projektu umožňuje na základě požadavků kladených na softwarový produkt a souvisejících podmínek určit projektové oblasti, na které je třeba se zaměřit. Vytvořená znalostní metodologická pomůcka má dvě podoby. První jsou znalostní mapy, jež umožňují vizuálně znázorňovat vazby mezi znalostmi softwarového inženýrství a metodikou. Druhou podobou je znalostní databáze, jež umožňuje práci s databází znalostí softwarového inženýrství a vyhledání v ní. Možné je též znalosti upravovat, přidávat a mazat. Informace v ní obsažené lze použít v projektu např. k úpravě (obohacení) použité metodiky či k výběru nové metodiky.

11.2 Rekapitulace splnění cíle práce

V disertační práci se mi podařilo splnit všechny vytčené cíle. V této závěrečné kapitole ukáži, jakým způsobem jsem naplnil vytyčené cíle.

Cílem práce bylo navrhnout vhodný formální model softwarového projektu, jež by umožnil vytvořit zastřešující pohled na řízení softwarového projektu. Model by měl sloužit jednak pro rozvoj poznání v oboru metodologie softwarového inženýrství a podporovat řešení praktických problémů.

Konečnými výsledky práce jsou:

1. Formální model softwarového projektu vytvořený dle stanovených požadavků;
2. Formální model řízení softwarového projektu;
3. Postup využití formálního modelu k tvorbě praktických nástrojů;
4. Dvě ukázky vytvoření praktických nástrojů a jejich použití.

Vytvořený formální model projektu splňuje tyto požadavky stanovené v cíli práce:

- *Model je založen na ověřených vědeckých poznatcích a formalismech*: Model je založen na formalismu systémového modelování, komplexních adaptivních systémech, teorii množin, relací, grafů a dalším matematickém aparátu;

- *Je dostatečně obecný a umožňuje konzistentní začlenění co největší části znalostí disciplíny softwarového inženýrství*: V kap. 7.5.1 jsem ukázal, že oba hlavní metodologické přístupy, tedy tvrdý i měkký se z hlediska modelu liší jen v přístupu k adaptaci projektu.
- *Není v rozporu s ověřenými postupy a principy softwarového inženýrství*: Není možné samozřejmě prokázat, že vytvořený model není v rozporu s žádným existujícím postupem či principem, nicméně porovnání vůči rozsáhlé rešerši v kap. 5 žádný rozpor neukázalo, tj. v modelu nebyl identifikován žádný prvek, jenž by byl v rozporu. Některé existující formalismy jsou vytvořenému modelu velmi blízké, například metoda funkčních jednic pro odhad složitosti softwarového projektu popsaná v kap. 5.4.4 je do modelu integrovatelná pouze s malými úpravami terminologie;
- *Umožňuje dále rozvíjet formalizaci a prohlubovat poznání zákonitostí softwarového inženýrství a vnitřních souvislostí*: V kap. 7.5.3 jsem ukázal formalizaci pojmu agilnost řízení softwarového projektu na základě formálního modelu řízení softwarového projektu, v kap. 8 jsem popsal postup vytváření konkrétních modelů, v kap. 9 jsem ukázal možnost konkretizace některých vazeb a v kap. 10 jsem popsal možný způsob formalizace znalostí z metodik softwarového inženýrství;
- *Podporuje tvorbu praktických pomůcek a aplikací*: V kap. 8 jsem popsal způsob, jakým lze od obecného formálního modelu dospět ke konkrétním pomůckám. V kap. 9 a 10 ukazuji konkrétní podrobné ukázky a výsledkem jsou dva podpůrné nástroje, jež jsou přímo použitelné v praxi.

V průběhu řešení byly s odborníky z firmy e-Fractal, s.r.o. ověřovány především praktické aspekty týkající se agilních metodik a řízení projektů v podmínkách neurčitosti.

Zpětná vazba od konzultantů společnosti Deloitte se zaměřovala především na souvislosti s řízením velkých softwarových projektů, poznatky související s vedením různých typů týmu a podobných oblastí.

Velký význam měla zpětná vazba od obou firem především především při vytváření obou praktických pomůcek. Cílem bylo, aby obě pomůcky byly nejen ukázkou praktického využití vytvořeného aparátu, ale měly potenciál skutečně přinést přidanou hodnotu do běžné praxe. Bylo doporučeno několik úprav, rozšíření a podrobnějšího rozpracování, nicméně nebyly shledány žádné závažné nedostatky bránící praktickému využití.

11.3 Další možnosti rozvoje tématu

Téma práce je velice širokou problematikou, která v práci nebyla zdaleka vyčerpána. Práce tak otevírá rozsáhlý prostor pro další výzkum v oblasti formalizace i praktických aplikací. Jedná se o předpoklady dalšího výzkumu zejména v následujících oblastech:

Rozpracování dalších částí obecného formálního modelu. V kap. 9 jsem ukázal rozpracování části formálního modelu týkajícího se vstupů, v kap. 10 jsem se zaměřil na vnitřní vazby a řízení softwarového projektu. Bylo by zajímavé podobným způsobem se podívat například na vztah vnitřních prvků a výstupů, na základě čehož by mohla vzniknout formalizace řízení výstupů projektu, změnového řízení, správy verzí, apod., a opět by bylo zajímavé na základě detailních modelů navrhovat rozličné praktické pomůcky. Další prostor pro rozvoj je samozřejmě i v oblastech postižených zmíněnými kapitolami 9 a 10.

Rozšíření modelu o kvantitativní charakteristiky. Model umožňuje pracovat s kvantitativními charakteristikami (jak jsem ukázal na příkladu v kap. 9), nicméně obecné kvantitativní charakteristiky nejsou v současné době jeho součástí. Zde je velmi inspirativní objektový model Davida Hecksela (kap. 5.4.5) použitý pro evaluaci metodiky.

Rozšíření modelu o dynamickou složku. Vytvořený formální model má statický charakter. Rozšíření o dynamickou složku by otevřelo další možnosti modelování reality. Zavedení časové veličiny by umožnilo modelovat děje v systému projektu a provádět simulace. Toto rozšíření považuji za velmi perspektivní a vidím v něm široké možnosti.

Vývoj metod odhadu pracnosti na základě formálního modelu. V kap. 5.4.4 jsem představil nejpoužívanější metody odhadu pracnosti softwarového projektu. Tyto metody jsou založeny na velmi zjednodušených a neúplných modelech, jež například nezohledňují metodiku řízení projektu. Předložený formální model otevírá možnost návrhu přesnějších metod odhadu.

Další rozvoj vytvořených pomůcek. Ačkoliv praktické pomůcky z kap. 9 a 10 jsem vytvořil pouze jako ilustrace použití formálního aparátu, při jejich tvorbě jsem kladl důraz na využitelnost v praxi. Zpětná vazba od obou firem z praxe i odezvy dalších odborníků naznačuje poptávku po rozvoji těchto a podobných nástrojů.

Výsledky této disertační práce poskytují rozsáhlé možnosti dalšího teoretického i praktického rozvoje, což byl i jeden z cílů práce.

12 Shrnutí dosud dosažených výsledků ve výzkumu

Problematikou softwarového inženýrství jsem se začal zabývat v průběhu studia na Fakultě elektrotechnické Českého vysokého učení technického, kde jsem obhájil diplomovou práci [Pergl 2003a] zabývající se agilními metodikami. V rámci diplomové práce jsem vytvořil softwarový nástroj pro podporu řízení projektů vedených metodikou Extrémního programování, který jsem dále rozvíjel pro praktické nasazení [Pergl 2003b]. Problematikou Extrémního programování se zabývají i články [Pergl 2004a] a [Pergl 2005a]. Dále jsem se zaměřil na další agilní metodiky a problematiku agilního přístupu obecně ([Pergl, Struska 2006a], [Pergl, Struska 2006b]).

Ve svém výzkumu se zaměřuji především na fázi analýzy a modelování. Ve svých publikacích se zabývám mj. otázkou hledání hranic systému v průběhu analýzy ([Pergl 2004b]), modelováním v podmínkách neurčitosti ([Pergl, Volráb 2007a]), ale též využitím modelování v praxi ([Pergl, Pícka 2005a], [Pergl, Pícka 2005b]. Věnuji se procesnímu modelování, především metodice BORM [Pergl 2004c] a možnostmi jejího použití pro různé druhy projektů a úkolů [Merunka et. al 2005b], [Pergl et al. 2005a], [Pergl, Struska 2006a], [Pergl, Kříž 2007]. Podílel jsem se metodicky na vývoji CASE nástroje pro metodiku BORM [Pergl, Pícka 2005c]. Ve svém výzkumu kladu důraz i na problematiku zajištění jakosti při vývoji informačních systémů [Pergl 2005b].

V předložené disertační práci jsem využil i výsledky svého výzkumu v oblasti metodologie ([Pergl 2007b], [Pergl 2007c] a projektového řízení ([Pergl 2006a], [Pergl, Kříž 2007]).

Výsledky výzkumu obsažené v této práci jsem publikoval též na zahraniční konferenci ([Pergl 2007a]) a je přijat k publikování článek v recenzovaném časopise Scientia Agriculturae Bohemica.

Své zkušenosti jsem využil i při přípravě několika skript pro PEF ČZU a knižní publikace (viz seznam literatury).

Nyní jsem na Katedře informačního inženýrství členem řešitelského týmu projektu „Inteligentní nástroje pro hodnocení relevance a strukturování obsahu obecných i specializovaných zdrojů dat, informací a znalostí“ Národního programu výzkumu II vyhlášeného Ministerstvem školství, mládeže a tělovýchovy. V průběhu doktorandského studia jsem se stal úspěšným řešitelem grantu Interní grantové agentury (IGA) PEF „Metodický rámec pro řízení moderních ICT projektů“ a spoluřešitelem grantu IGA PEF „Metodologie pro měření složitosti vývoje softwaru“. V současné době jsem též členem spoluřešitelského týmu grantu GAČR „AMIMADES - Inteligence prostředí pro podporu manažerského rozhodování“.

Příloha 1: Charakteristiky jakosti dle normy ISO 9126

Pro vysvětlení charakteristik a podcharakteristik jsem použil pramen [Vaníček 2004].

12.1 Funkčnost (Functionality)

Funkčnost je vymezena jako schopnost informačního systému či softwarového produktu obsahovat funkce, které zabezpečují předpokládané nebo stanovené potřeby uživatele při používání systému za stanovených podmínek.

Tato charakteristika tedy zjišťuje, **zda** jsou funkce vůbec zabezpečeny, **nikoliv jak** jsou zabezpečeny.

12.1.1 Funkční přiměřenost (Suitability)

Funkční přiměřenost je vymezena jako schopnost poskytovat funkce pro zajištění specifikovaných úloh a cílů uživatele.

Drobný rozdíl mezi funkční přiměřeností jako podcharakteristikou a funkčností spočívá v tom, že mít k dispozici funkci pro danou úlohu ještě neznamená automaticky mít k dispozici funkci, která naplňuje naše potřeby. Tato funkce nemusí být dostatečně bezpečná nebo dostatečně přesná nebo mohou vznikat potíže při následném využití výstupů z této funkce. Opět jde pouze o to, zda takové funkce k dispozici jsou či nikoliv. Podmínky, za kterých mohou být použity, a jejich vlastnosti se hodnotí v rámci dalších charakteristik a podcharakteristik.

12.1.2 Přesnost (Accuracy)

Přesnost je vymezena jako schopnost poskytnout správné a požadované výsledky s potřebnou úrovní přesnosti.

Přesnost je vlastností počítače (délka registrů), softwaru (způsob kódování čísel), ale především použitého algoritmu. Nikdy se totiž nelze zcela vyhnout nutnosti výsledky zaokrouhlovat a nutně dochází k zaokrouhlovacím chybám. K nim se může algoritmus chovat neutrálně, může je vzájemně neutralizovat, ale může také způsobit jejich skládání tak, že výsledek znehodnotí.

12.1.3 Schopnost spolupráce (Interoperability)

Schopnost spolupráce je vymezena jako schopnost spolupracovat s jedním nebo několika jinými specifikovanými systémy.

Této vlastnosti se také říká „slučitelnost“ (kompatibilita). Týká se především možnosti datové komunikace jednotlivých systémů (stejné formáty dat), ale i možnosti volání služeb jednoho systému ze systému druhého (například formou podprogramu) nebo spojení dvou systémů korutinou („rourou“) v jediný.

12.1.4 Bezpečnost (Security)

Bezpečnost je vymezena jako schopnost chránit informace a data tak, aby neautorizovaná osoba nebo systém neměla možnost je číst či modifikovat a přitom autorizovaným subjektům nebyla odepřena stanovená úroveň přístupu k datům.

Jde tedy výhradně o zabezpečení dat, nikoliv ostatních složek systému. Jich se týká charakteristika „zabezpečení“, zkoumaná v rámci jakosti používání. Někdy se této důležité podcharakteristice říká také „*bezpečnost dat*“.

12.1.5 Shoda ve funkčnosti (Functionality compliance)

Shoda ve funkčnosti je vymezena jako schopnost pracovat ve shodě s normami, standardy, zákony, konvencemi a zvyklostmi obvyklými v prostředí, v kterém je systém či produkt využíván.

Tato podcharakteristika je doplňována ke všem charakteristikám jakosti, aby bylo možné hodnotit, do jaké míry je hodnocená entita přizpůsobena lokálním požadavkům. Jde o určitý kompromis mezi potřebou uznávat „lokální specifičnosti“ a požadavkem jednotné mezinárodní normalizace.

Protože tato podcharakteristika se vyskytuje ve stejném významu i u dalších pěti charakteristik jakosti, budeme ji nadále uvádět již bez komentáře.

12.2 Bezporuchovost (Reliability)

Bezporuchovost je vymezena jako schopnost informačního systému či softwarového produktu zachovat specifikovanou úroveň výkonu při používání systému za stanovených podmínek.

Hovoří se zde o „zachování specifikované úrovně výkonu“, nikoliv o plné funkci. Objeví-li se problém, bude hodnocení bezporuchovosti jiné, znemožní-li tento problém práci se systémem zcela nebo pouze bude-li systém k dispozici s omezeným rozsahem funkcí nebo bude-li například pracovat pomaleji.

Pro tuto charakteristiku se nabízí české slovo „spolehlivost“. Bylo však rozhodnuto jej nepoužít z toho důvodu, že v obecné teorii spolehlivosti a v normách z této oblasti se tohoto slova používá jako ekvivalentu anglického termínu „*dependability*“, který má širší význam. Kromě hodnocení poruch a reakce systému na něj zahrnuje ještě další charakteristiky důležité pro uživatele, jako je například dostupnost systému a míra péče o něj. Tyto vlastnosti jsou zahrnuty do dalších charakteristik jakosti informačních systémů a softwaru. Jako ekvivalent anglického termínu „*reliability*“ je proto používáno slova bezporuchovost.

12.2.1 Zralost (Maturity)

Zralost je vymezena jako schopnost systému vyvarovat se poruchám (selháním) v důsledku závad systému nebo důsledky takovýchto selhání minimalizovat.

Poruchou (failure) rozumíme situaci, kdy systém neplní své funkce nebo se chová jinak, než bylo požadováno. Podle důsledků, které porucha má, jej obvykle nazýváme **anomálie** (funkce je zajištěna, ale na jiné úrovni než bylo

požadováno), **defekt** (funkce zajištěna není nebo jen nedostatečně) a **havárie** (funkce není zajištěna a byla způsobena další škoda, než pouhé odepření funkce).

Porucha je způsobena odchylkou hardwaru, softwaru nebo obsluhy od požadavků na systém. Takovéto odchylce stavu systému od požadovaného stavu říkáme **vada** nebo též **závada** (fault). Závadě softwaru pak obvykle **chyba** (error).

Je zřejmé, že vztah mezi vadou a poruchou není vzájemně jednoznačný. Některé závady nemusí způsobit poruchu (dané větve programu, v kterém je chyba, se v daném případě nemusí vůbec použít). Jedna vada se na druhé straně může projevit několika různými poruchami, někdy i poruchami různého typu. Často bývá obtížnější objevit vadu, která poruchu způsobila, než tuto poruchu odstranit.

Praxe ukazuje, že snaha odstranit všechny chyby softwaru, nebývá u složitějších systémů reálná. Minimalizovat důsledky takovýchto chyb bývá často rozumnou volbou.

12.2.2 Odolnost vůči vadám (Fault Torelance)

Odolnost vůči vadám je vymezena jako schopnost systému zachovat si při selhání systému nebo při nedodržení požadovaného rozhraní ze strany uživatele určitou úroveň výkonu (poskytovaných služeb).

Za nejdůležitější pro takto zachovanou úroveň výkonu lze považovat zabezpečení dat systému. Jde tedy ve své podstatě o to, aby závady způsobily pouze anomálie, v horším případě pak defekt, nikdy však ne havárii.

12.2.3 Schopnost zotavení (Recoverability)

Schopnost zotavení je vymezena jako vlastnost systému obnovit úroveň výkonu a zachovat data po odstranění poruchy.

Do této podcharakteristiky je třeba zahrnout i zhodnocení, po jakou dobu nebylo možno v důsledku poruchy se systémem pracovat a jaké byly náklady na odstranění škody, kterou porucha způsobila. Kombinací této podcharakteristiky a zralosti, která zahrnuje frekvenci poruch, je tak zvaná **dostupnost systému** (availability), popisovaná obvykle jako poměr času, v kterém systém úspěšně pracoval, k celkovému času jeho nasazení.

12.2.4 Shoda v bezporuchovosti (Reliability Compliance)

Má obdobný význam jako shoda u ostatních charakteristik.

12.3 Použitelnost (Usability)

Použitelnost je vymezena jako schopnost informačního systému či softwarového produktu být srozumitelný, se snadno naučitelnou obsluhou a atraktivní při používání za stanovených podmínek.

Je zřejmé, že některé aspekty funkčnosti i bezporuchovosti jsou též aspektem použitelnosti.

Je třeba si uvědomit, že termínem „uživatel“ je míněn nejen přímý uživatel, kterým je obsluha informačního systému, ale i uživatel nepřímý (zainteresovaná strana). Tím je míněn každý, jehož činnost závisí nebo může být ovlivněna tímto systémem. Do hodnocení použitelnosti je tedy třeba promítnout i úsilí věnované na přípravu použití systému a na vyhodnocení výsledků tohoto použití a to v různých prostředích, ve kterých k takovému použití dochází.

12.3.1 Srozumitelnost (Understandability)

Srozumitelnost je vymezena jako vlastnost systému, která umožňuje uživateli rozhodnout, zda se systém hodí pro řešení jeho problémů, jak je možné jej užít pro řešení jednotlivých úloh a za jakých podmínek. Je charakterizována mírou úsilí, které je potřeba pro to, aby uživatel porozuměl tomu, co může od systému očekávat.

Tato podcharakteristika je zaměřena na úsilí spojené s použitím systému, které musí vynaložit jeho uživatel v širším smyslu slova, tedy ten, jemuž mají výsledky sloužit, a to včetně počátečních etap využívání a etap před rozhodnutím o zakoupení softwarového balíku nebo rozhodnutí o vývoji informačního systému.

V této souvislosti je třeba analyzovat též informatickou vzdělanost a zkušenost budoucích uživatelů a tomu přizpůsobit srozumitelnost.

12.3.2 Naučitelnost (Learnability)

Naučitelnost je vymezena jako vlastnost systému charakterizovaná mírou úsilí, které je třeba vynaložit pro rutinní využívání jeho možností.

Tato podcharakteristika zahrnuje i úsilí, které je třeba vynaložit na školení personálu a dozor nad správným používáním systému.

12.3.3 Provozovatelnost (Operability)

Provozovatelnost je vymezena jako vlastnost systému usnadňující jeho obsluhu a řízení rutinní práce se systémem.

Jde o to, jak jsou snadné obslužné činnosti, které nutně nevyžadují porozumění všem funkcím systému.

12.3.4 Atraktivnost (Attractiveness)

Atraktivnost je vymezena jako schopnost systému umožnit příjemnou obsluhu a učinit užití systému přitažlivým.

Sem patří například užití barev, grafiky, zvuků a multimediálních prvků, které mohou učinit práci se systémem příjemnější, a další ergonomické parametry systému.

Atraktivnost bude důležitým parametrem zvláště v případech, kdy uživatelé jsou vůči budoucímu systému skeptičtí, či ho přímo odmítají. Vhodně nastavená atraktivnost pomůže získat sympatie uživatelů.

12.3.5 Shoda v použitelnosti (Usability Compliance)

Má obdobný význam jako shoda u ostatních charakteristik.

12.4 Účinnost (Efficiency)

Účinnost je vymezena jako schopnost informačního systému či softwarového produktu poskytovat potřebný výkon vzhledem k množství použitých zdrojů při používání za stanovených podmínek.

Mezi zdroje systému je třeba zahrnout ostatní software, konfiguraci výpočetních prostředků včetně technického a programového vybavení i potřebný materiál (média). Velmi důležitým zdrojem je i čas, potřebný pro provedení výpočtu.

12.4.1 Časové chování (Time Behaviour)

Časové chování je vymezeno jako schopnost systému zajistit požadovanou propustnost úloh za dané časové období, dobu výpočtu úlohy nebo odezvu systému při používání systému za daných podmínek.

Při zkoumání toho, jaký čas je potřeba k výpočtu úlohy v dávkovém režimu nebo jaký čas je požadován pro tak zvanou odezvu systému v interakčním režimu, je potřeba zvážit mimo jiné následující důležité skutečnosti:

Dobu výpočtu nebo dobu odezvy nelze stanovit absolutně, ale vždy relativně vzhledem ke zpracovávaným datům. Jako charakteristika těchto dat je nejčastěji vhodné vzít **rozsah dat na vstupu,** případně **rozsah datové základny,** s kterou probíhá komunikace. Podrobnější rozbor časové složitosti je však mimo rámec této práce.

12.4.2 Využití zdrojů (Resource Utilisation)

Využití zdrojů je vymezeno jako schopnost systému zajistit požadované funkce přiměřeným počtem typů a množstvím a rozsahem užitých zdrojů, které jsou potřeba k zabezpečení práce systému.

Do této podcharaskteristiky je třeba zahrnout jak potřebný rozsah paměti procesoru i vnějších pamětí, tak i další hardwarové požadavky na procesor (koprocesory, síťové a grafické karty, …), i periferní zařízení (vstupy a výstupy) a média. Tedy nároky na všechny ostatní zdroje kromě času, který je hodnocen v předchozí podcharakteristice. V širším slova smyslu sem patří i nároky na obsluhu. Nároky na lidské zdroje se však zpravidla uvažují až při hodnocení jakosti užití.

Pokud jde o paměť, je situace obdobná jako u časového chování. Potřeba paměti pro výpočet je dána rozsahem zpracovávaných dat. Stejně jako u časové složitosti lze tak zvanou prostorovou složitost algoritmu měřit pomocí funkce, jejíž nezávisle proměnná je rozsah vstupních dat a závisle proměnná je počet paměťových míst, potřebných pro realizaci výpočtu. V praktických aplikacích bývají zpravidla problémy vyplývající z časové složitosti významnějším omezením než problémy vyplývající ze složitosti prostorové.

12.4.3 Shoda v účinnosti (Efficiency Compliance)

Má obdobný význam jako shoda u ostatních charakteristik.

12.5 Udržovatelnost (Maintainability)

Udržovatelnost je vymezena jako schopnost informačního systému či softwarového produktu být modifikován. Modifikace zahrnují opravy nedostatků, vylepšování, adaptaci vzhledem ke změnám prostředí, změnám požadavků a změnám funkční specifikace.

Udržovatelnost odráží úsilí, které je potřeba vynaložit pro provedení takovýchto změn. Nezahrnuje úsilí, které se vynakládá pro běžné používání systému bez jeho změny. To je zahrnuto do charakteristiky použitelnost.

12.5.1 Analyzovatelnost (Analysability)

Analyzovatelnost je vymezena jako schopnost systému usnadnit nalezení vady v případě výskytu poruch a schopnost určit, co má být změněno, aby byla vada odstraněna.

Jde tedy o míru úsilí potřebné k určení, kde je chyba způsobující poruchu, a jak ji odstranit, nikoliv o vlastní úsilí na provedení této opravy.

12.5.2 Měnitelnost (Changeability)

Měnitelnost je vymezena jako schopnost systému usnadňující provedení modifikace.

Jde o úsilí, které je třeba vynaložit na návrh, implementaci (včetně kódování programu) a odzkoušení systému po jeho změně. Zahrnuje i případné potřebné změny v dokumentaci.

Požadavek na změny může být vyvolán potřebou opravit zjištěnou vadu, ale též potřebou změnit systém na základě změny prostředí (hardwaru, na kterém systém pracuje, softwaru, který využívá, systémů, s kterými má spolupracovat, nebo legislativních pravidel, na které musí systém reagovat). Změny mohou vyplývat i ze změny zadání, například z potřeby doplnit či zlepšit některé funkce systému.

12.5.3 Stabilnost (Stability)

Stabilnost je vymezena jako schopnost systému zabránit nežádoucím důsledkům provedených modifikací.

Je všeobecně známo, že oprava jednoho nedostatku může často způsobit problém jinde. Tomuto jevu se říká obvykle „vedlejší efekt“. U složitých systémů bývá toto nebezpečí tak velké, že rozsah a závažnost problémů vyvolaných modifikací bývá srovnatelná s výhodami, které byly provedenou modifikací získány. Někdy bývá i „ztráta“ vyšší než „zisk“. Jde o vlastnost systému tento závažný problém vyloučit, či aspoň omezit.

12.5.4 Testovatelnost (Testability)

Testovatelnost je vymezena jako schopnost systému zabezpečit snadnou validaci po provedení modifikace.

Validací se zde rozumí prověření, zda systém v zadaných podmínkách při provozu plní své funkce s dodržením požadované úrovně jakosti. Nikoliv pouhé separátní ověření jednotlivých funkcí. Tato možnost je důležitá zvlášť v případě, kdy uživatel chce nadále pracovat s daty, která byla získána v předcházejících verzích systému. V tom případě je zcela legitimní požadavek, aby si mohl sám prověřit, že modifikovaná verze může se stávajícími daty pracovat a splňuje jeho požadavky.

12.5.5 Shoda v udržovatelnosti (Maintainability compliance)

Má obdobný význam jako shoda u ostatních charakteristik.

12.6 Přenositelnost (Portability)

Přenositelnost je vymezena jako schopnost informačního systému softwarového produktu být přenesen z jednoho prostředí do jiného.

Prostředím je třeba rozumět organizační uspořádání, v kterém je systém využíván, hardwarové prostředí (typ a verze počítače a periferií) a softwarové prostředí (typ operačního systému a jeho verze). Opět je zřejmé, že v případě udržovatelnosti a přenositelnosti dochází ke značnému překrytí.

12.6.1 Přizpůsobitelnost (Adaptability)

Přizpůsobitelnost je vymezena jako schopnost systému být vlastními prostředky, které jsou jeho součástí, v průběhu používání přizpůsoben různým prostředím, v kterých má být využíván.

Jde například o možnost volby rozsahu zpracovávaných polí, tabulek, formátu zpráv, rozsahu transakcí, či velikosti oken na obrazovce podle konkrétních požadavků uživatele. Přizpůsobení musí být možné kdykoliv v průběhu práce se systémem, bez nutnosti znova systém zavádět.

U této podcharakteristiky jde opět o významné překrytí s přiměřeností (funkčnost) i s provozovatelností (použitelnost). Překrytí se týká i následujících podcharakteristik.

12.6.2 Instalovatelnost (Installability)

Instalovatelnost je vymezena jako vlastnost systému být zaveden tak, aby vyhovoval použití a práci v konkrétním prostředí.

Tato podcharakteristika zahrnuje jak možnost přizpůsobení při zavádění systému, tak i míru úsilí, kterou je potřeba pro provedení správné instalace vynaložit a při ní přizpůsobit systém konkrétním požadavkům.

12.6.3 Slučitelnost (Co-existence)

Slučitelnost je vymezena jako schopnost systému pracovat společně s jinými systémy ve společném prostředí a využívat společné zdroje.

Tato podcharakteristika se někdy nazývá „kompatibilitou“. Řešitel by se v tomto kontextu měl zamyslet, jaké klade stávající informační infrastruktura zákazníka požadavky na nastavení projektu. Tyto požadavky mohou být velmi různé. S neexistencí stávající infrastruktury se dnes setkáme jen zřídka, nicméně pokud produkt má mít charakter samostatné utility, jedná se o podobnou situaci.

12.6.4 Nahraditelnost (Replaceability)

Nahraditelnost je vymezena jako schopnost systému nahradit funkci jiných systémů, určených pro stejný účel a pracujících ve shodném nebo podobném prostředí.

Velmi významná je tato podcharakteristika při nahrazování staré verze softwaru verzí novou. V tom případě skutečnost, že nová verze nezajišťuje nějaké funkce nebo je zajišťuje jinak, může uživateli způsobit vážné problémy, které převáží nad výhodami, které nová verze slibuje. Je třeba dát pozor na to, že i této vlastnosti se často říká „kompatibilita“.

I nahraditelnost samozřejmě úzce souvisí s přizpůsobitelností i s instalovatelností.

12.6.5 Shoda ve přenositelnosti (Portability Compliance)

Má obdobný význam jako shoda u ostatních charakteristik.

Literatura

Vlastní publikace autora a publikace s podílem

Knižní publikace

[Vaníček et al. 2007a] Vaníček J., Papík M., Pergl R., Vaníček T.: *Teoretické základy informatiky*, Kernberg Publishing 2007, ISBN 978-80-903962-4-1

Skripta

[Merunka et al. 2004] Merunka V., Pergl R., Pícka M.: *Objektově orientovaná tvorba software*, ČZU Praha, 2004, ISBN 80-213-1159-2

[Merunka et al. 2005] Merunka V., Pergl R., Pícka M.: *Objektově orientovaný přístup v projektování informačních systémů*, skriptum ČZU PEF Praha 2005, ISBN 80-213-1352-8

[Vaníček, Pergl 2005a] Pergl R., Vaníček J.: *Mathematical Theory of Programs Part 2: Computational Complexity*, skriptum ČZU PEF Praha 2005

[Vaníček, Pergl 2005b] Pergl R., Vaníček J.: *Mathematical Theory of Programs Part 3: Logical and Functional Programming Paradigms*, skriptum ČZU PEF Praha 2005

Příspěvky ve sbornících zahraničních konferencí

[Pergl 2007a] Pergl R.: *Distinct Features in Agile Methodologies*, In: Software Engineering Theory and Practice, International Society for Research in Science and Technology, Tallahasse 2007, ISBN 978-0-9727412-6-2

Příspěvky ve sbornících tuzemských konferencí

[Merunka et. al 2005b] Merunka V., Pergl R., Pícka M.: *Spojení business a informačního inženýrství pomocí metody BORM*, In: Modelování a optimalizace podnikových procesů, Plzeň 2005, ISBN 80-7043-352-3

[Pergl 2003a] Pergl R.: *Programové nástroje pro podporu projektového řízení tvorby softwaru metodou extrémního programování*, diplomová práce na ČVUT FEL, 2003

[Pergl 2003b] Pergl R.: *XPMT – Extreme Programming Management Toolkit, XPMT -- Extreme*

Programming Management Toolkit, In: Objekty 2003, ČSSI, Ostrava 2003, ISBN 80-248-0274-0

[Pergl 2004a] Pergl R.: *Pozice a možnosti extrémního programování při řízení softwarových projektů*, In: Sborník příspěvků z doktorandského semináře, ČZU, Praha 2004, ISBN 80-213-1150-9

[Pergl 2004c] Pergl, R.: *Pár poznámek k metodice BORM z hlediska systémového modelování*, In: konference Objekty 2004, ČSSI, Ostrava 2004, ISBN 80-248-0672-X

[Pergl 2004b] Pergl R.: *Souvislosti problematiky systémového modelování a tvorby informačních systémů*, In: Agrární perspektivy XIII, ČZU, Praha 2004, ISBN 80-213-1190-8

[Pergl 2005a] Pergl R.: *Analýza vnitřních vazeb principů metody extrémního programování*, In: Sborník příspěvků z doktorandského semináře, ČZU, Praha 2005, ISBN 80-213-1314-5

[Pergl 2005b] Pergl R.: *Zajištění jakosti při business modelování*, In: sborník konference Tvorba Softwaru 2005, Vysoká škola báňská, Ostrava 2005, ISBN 80-86840-14-X

[Pergl 2006a] Pergl R.: *Význam objektového přístupu v softwarovém inženýrství v souvislosti s moderními přístupy řízení*, In: Objekty 2006, ČZU, Praha 2006, ISBN 80-213-1568-7

[Pergl 2007b] Pergl R.: *Strukturování prvků metodik tvorby softwaru*, In: Tvorba softwaru 2007, Vysoká škola báňská, Ostrava 2007, ISBN 978-80-248-1427-8

[Pergl 2007c] Pergl R.: *Strukturování prvků metodik tvorby softwaru: fáze nasazení, správa a podpora*, In: Agrární perspektivy XVI, ČZU, Praha 2007, ISBN 978-80-213-1675-1

[Pergl, Kříž 2007] Pergl R., Kříž J.: *Modelování procesů v organizaci*, In: Agrární perspektivy XVI, ČZU, Praha 2007, ISBN 978-80-213-1675-1

[Pergl, Pícka 2005a] Pergl R., Pícka M.: *Metamodelling Utilisation for Biometric Data Applications*, In: sborník konference Agrární perspektivy XIII, ČZU, Praha 2005, ISBN 80-213-1372-2

[Pergl, Pícka 2005b] Pergl R., Pícka M.: *Temporální datový sklad jako integrační prvek znalostního systému organizace*, In: Firma a konkurenční prostředí, Konvoj, Brno 2005, ISBN 80-7302-097-1

[Pergl, Struska 2006a] Pergl R., Struska Z.: *Čím mohou přispět nejznámější agilní metodiky ke zlepšení vývojového procesu?*, In: Agrární perspektivy XV., ČZU, Praha 2006, ISBN 80-213-1531-8

[Pergl, Struska 2006b] Pergl R., Struska Z.: *Agilní modelování a metoda BORM*, In: Tvorba softwaru 2006, ČSSI, Ostrava 2006, ISBN 80-248-1082-4

[Pergl, Volráb 2007a] Pergl R., Volráb O.: *Objektově-orientované modelování znalostních informačních systémů v podmínkách neurčitosti*, In: Objekty 2007, Vysoká škola báňská, Ostrava

2007, ISBN 978-80-248-1635-7

[Pícka, Pergl 2005c] Pergl R., Pícka M.: *Použití transformací v metodě BORM v nástroji Craft.CASE 2.x*, In: Objekty 2005, ČSSI, Ostrava 2005, ISBN 80-248-0595-2

[Struska, Pergl 2006a] Struska Z., Pergl R.: *Srovnání metod COCOMO a analýzy function points*, In: Agrární perspektivy XV., ČZU, Praha 2006, ISBN 80-213-1531-8

[Struska, Pergl 2006b] Struska Z., Pergl R.: *Metody odhadu pracnosti založené na modelu*, In: Tvorba softwaru 2006, ČSSI, Ostrava 2006, ISBN 80-248-1082-4

[Vaníček et al. 2005c] Merunka, V., Klimešová, D., Pergl, R., Toman, P., Vaníček, J., Veselý, A., Vrana, I.: *Prvé zkušenosti KII s navazujícím magisterským oborem informatika v angličtině*, In: Informatika XVI, Konvoj, Brno 2005, ISBN 80-7302-083-1

[Vaníček et al. 2007b] Vaníček J., Pergl R., Papík M.: *Obsah a principy výuky teoretických základů informatiky*, In: Informatika 2007, Konvoj, Brno 2007, ISBN 978-807302-134-X

[Vaníček et al. 2007c] Vaníček J., Pergl R., Papík M.: *Atestace softwaru, její význam a rizika*, In: Tvorba softwaru 2007, Vysoká škola báňská, Ostrava 2007, ISBN 978-80-248-1427-8

Ostatní literatura

[Ambler 1998] Ambler S.,W.: *Process Patterns : Building Large-Scale Systems Using Object Technology*, Cambridge University Press 1998, ISBN 0-521-64568-9

[Ambler 1999] Ambler S.,W.: *More Process Patterns : Delivering Large-Scale Systems Using Object Technology*, Cambridge University Press 1999, ISBN 0-521-65262-6

[Ambler 2005] Ambler, S. W.: *The Object Primer – Agile Model-driven Development with UML 2.0*, Cambridge University Press 2005, ISBN 978-0-521-54918-6

[Ambler et al. 2005] Ambler S. W., Nalbone J., Vizdos M. J.: *The Enterprise Unified Process*, Prentice Hall 2005, ISBN 0131914510

[Ambler 2006] Ambler S.: *Agile Unified Process*, dostupný na http://www.ambysoft.com/unifiedprocess/agileUP.html

[Beck 2002a] Beck K.: *Extrémní programování* (český překlad) Grada 2002, ISBN 80-247-0300-9

[Beck 2002b] Beck K.: *Test Driven Development: By Example***,** Addison-Wesley 2002, ISBN 0321146530

[Beck 2004] Andres C., Beck K.: *Extreme programming Explained -- Embrace Change*, Addison-Wesley 2004, ISBN 0321278658

[Beck at al. 2001] Beck K. at al.: *Manifest agilního vývoje*, http://agilealliance.org

[Bělohlávek et al. 2001] Bělohlávek F., Košťan P., Šuleř O.: *Management*, Rubico 2001, ISBN 80-85839-45-8

[Blaha et al. 1995] Blaha M., Premerlani M.: *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall 1995, ISBN 0-13-123829-9

[Booch 1994] Booch G.: *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin Cummings 1994, ISBN 0-8053-5340-2

[Buchalcevová 2002] Buchalcevová A.: *Agilní metodiky*, sborník konference OBJEKTY 2002, ČZU Praha 2002, ISBN 80-213-0947-4

[Buchalcevová 2005] Buchalcevová A.: *Metodiky vývoje a údržby informačních systémů*, Grada 2005, ISBN 80-247-1075-7

[Buchalcevová 2005] Buchalcevová, A.: *Metodika feature-driven development neopouští modelování a procesy, a přesto přináší výhody agilního vývoje*, In: Tvorba softwaru 2005, ČSSI Ostrava 2005

[Coburn 2004] Coburn A.: *Crystal Clear*, Addison-Wesley 2004, ISBN 0201699478

[DeGrace 1990] DeGrace P., Stahl L.H.: *Wicked Prolems, Righteous Solutions: Catalogue of Modern Software Engineering paradigms*, Yourdon Press 1990, ISBN 013590126X

[Dome 2006] stránky autorů projektu DOME týkající se modelování (duben 2006): http://www.htc.honeywell.com/dome/index.htm

[Drbal 1996] Drbal P.: *Proudy v objektově orientovaných metodikách*, sborník konference Tvorba softwaru 1996, Ostrava

[DSDM 2003] DSDM Consortium: *DSDM: Business Focused Development*, Pearson Education 2003, ISBN 0321112245

[Felsing 2002] Felsing J. M., Palmer S. R.: *A Practical Guide to Feature-Driven Development*, Prentice Hall 2002, ISBN 0130676152

[Fowler et al. 1999] Fowler M. et al.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999, ISBN 0201485672

[FURPS 2007] Metoda klasifikace uživatelských požadavků FURPS+: http://www-128.ibm.com/developerworks/rational/library/4706.html

[Gamma et al. 1995] Gamma, E.; Helm, R.; Johnson, E.R; Vlissides, J.: *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley, New York, USA, 1995. ISBN 0201633612

[Havlíček 2006] Havlíček J.: Preludes to Knowledge, In: *Scientia Agriculturae Bohemica 15, 4*, Praha 2007

[Havlíček, Pelikán 2007] Havlíček J., Pelikán M.: Metrika ve znalostních mapách, In: *Agrární perspektivy XVI*, ČZU, Praha 2007, ISBN 978-80-213-1675-1

[Hecksel 2007] Hecksel D.: *Methodology Evaluation and Selection*, dostupné na

http://www.davidhecksel.com/projectcontext/whitepaper.html

[Henderson-Sellers 1994] Henderson-Sellers B., Edwards J.M.: *MOSES - A Second Generation Object-Oriented Methodology*, pp 68-73, Object Magazine 4(3) 1994

[Highsmith 1999] Highsmith J.: *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing Company, Inc. 1999, ISBN 0932633404

[Holland 1996] Holland J. H., *Hidden Order: How Adaptation Builds Complexity*, Addison-Wesley Publishing Co., 1996, ISBN 0201442302

[Chen 1975] Chen P.: *The Enity-Relationship Model: Toward a Unified View of Data,* Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA. ACM

[ISO 9000] ČSN EN ISO 9000: Systémy managementu jakosti – základy, zásady a slovník, Český normalizační institut, Praha, 2002

[ISO 90003] ISO/IEC 90003: Software Engineering – Guidelines for the application of ISO 9001:2000 to computer software, First edition 2004

[ISO 9126-1] ISO/IEC IS 9126-1 Information Technology - Software product Quality – Part 1: Quality model

[ISO 9126-2] ISO/IEC TR 9126-2 Information Technology - Software product Quality – Part 2: External metrics

[ISO 9126-3] ISO/IEC TR 9126-3 Information Technology - Software product Quality – Part 3: Internal metrics

[ISO 9126-4] ISO/IEC TR 9126-4 Information Technology - Software product Quality – Part 4: Quality in use metrics

[Jacobson et al. 1992] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering - A Use Case Driven Appro*ach, 1992, Addison Wesley ISBN 0-201-54435-0

[Jacobson et al. 1999] Jacobson I., Booch G., Rumbaugh J.: *The Unified Software Developement Process*, Addison Wesley, 1999, ISBN 0-2015-7169-2

[Jeffries 2000] Jeffries R.: *Extreme Programming Installed (The XP Series)*, Addison-Wesley 2000, ISBN 0201708426

[Kadlec 2004] Kadlec V.: *Agilní programování. Metodiky efektivního vývoje softwaru*, Computer Press 2004, ISBN 80-251-0342-0

[Kano, 1984] Kano, N.: Attractive quality and must-be quality, *The Journal of the Japanese Society for Quality Control*, 1984, pp. 39-48

[Keyes 2002] Keyes J.: *Software Engineering Handbook*, AUERBACH, 2002, ISBN 0849314798

[Král 1996] Král, J.: *Informační systémy*: Specifikace, realizace, provoz. Science, Veletiny 1996

[Kruchten 2000] Kruchten P.: *The Rational Unified Process – An Introduction*, Addison-Wesley, 2000, ISBN 0201707101

[Martin et al. 1995] Martin J., Odell J.: *Object-Oriented Methods - A Foundation*, Prentice Hall 1995, ISBN 0-13-63856-2

[Merunka et al. 2003] Carda A., Merunka V., Polák J.: *Umění systémového návrhu*, Grada Publishing 2003, ISBN 80-247-0424-2

[Meyer 1997] Meyer B.: *Object Oriented Software Construction*, Prentice Hall, 1997, ISBN 013629155

[Page-Jones 2001] Page-Jones M.: Základy *objektově orientovaného návrhu v UML*.Grada Publishing. Praha 2001. ISBN 80-247-0210-X

[Pigoski 1997] Pigoski, T.M.. *Practical Software Maintenance: Best Practices for Managing Your Software Investment* . New York: John Wiley and Sons, Inc, 1997

[Poppendieck 2003] Poppendieck M., Poppendieck T.: *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley 2003, ISBN 0321150783

[Royce 1970] Royce W.: *Managing the Development of Large Software Systems*, IEEE Proceedings 1970

[Rumbaugh et al. 1991] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: *Object-Oriented Modelling and Design*, Prentice Hall 1991, ISBN 0-13-630054-5

[Rumbaugh et al. 1998] Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual,Addison*-Wesley, 1998, ISBN 0-2013-0998-X

[Scott et al. 2004] Scott K. et al.: *MDA Distilled*, Addison-Wesley 2004, ISBN 0201788918

[Schmuller 2001] Schmuller J.: Myslíme *v jazyku UML. Grada Publishing*. Praha 2001. ISBN 80-247-0029-8

[Schwaber 1996] Schwaber K.: *SCRUM Development Process*, http://jeffsutherland.com/oopsla/schwapub.pdf

[Schwaber et al. 2001] Schwaber K., Beedle M.: *Agile Software Development with SCRUM*, Prentice Hall 2001, ISBN 0130676349

[Standish 1995] Standish Group International: *The Chaos Report*, dostupné na http://www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf

[Tourniare, Farrell 1997] Tourniaire, F., Farrell, R.. *The Art of Software Support: Design & Operation of Support Centers and Help Desks* . Upper Saddle River, New Jersey: Prentice Hall, 1997

[UML 2006] *The Unified Modeling Language Documents*, http://www.uml.org , květen 2006

[Vaníček 2004] Vaníček, J.: *Měření a hodnocení jakosti informačních systémů*. ČZU, PEF 2004, druhé přepracované vydání vysokoškolských skript, 2004.

[Vaníček 2007] Vaníček J.: *Kvalita informačních systémů z pohledu mezinárodní normalizace (aktuální informace)*, ČZU, 1997 ISBN 978-80-213-1720-8

[Wegner 1997] Wegner, P.: Why Interaction is More Powerful Than Algorithms, *Communications of the ACM 40(5)*, p. 80-91, 1997

[Vrana, Mahnič 2007] V. Mahnič, I. Vrana: "Using stakeholder-driven process performance measurement for monitoring the performance of a Scrum-based software development process", Electrotechnical Review, Vol. 74, No. 5, 2007, pp. 241-247.

[Weiss 1991] Weiss, E.H., *How To Write Usable User Documentation.* Phoenix, Arizona: The Oryx Press, 1991

[Wheatley 1992] Wheatley, M.J.: Leadership and the New Science: Learning About Organizations from an Orderly Universe, Berrett-Koehler, 1992

[Wilkie 1993] Wilkie G.: *Object-Oriented Software Engineering*, Addison Wesley 1993, ISBN 0-201-6276-1

[Wilkie 1993] Wilkie G.: *Object-Oriented Software Engineering*, Addison Wesley 1993, ISBN 0-201-6276-1

[Yourdon 1994] Yourdon E.: *Object-Oriented System Design - An Integrated Approach*, Prentice Hall 1994, ISBN 0-13-176892-1

[Získal 2003] Získal J., Havlíček I.: *Ekonomicko matematické metody I*, ČZU PEF 2003, ISBN 80-213