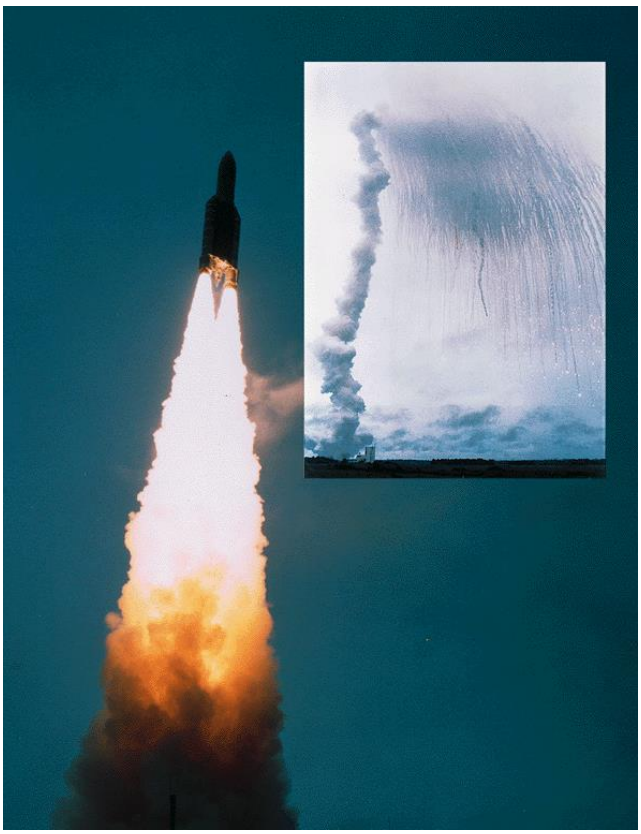
SOFTWAROVÉ INŽENÝRSTVÍ

1. Úvod do softwarového inženýrství



Chyba v software může způsobit pád aplikace v telefonu, a nebo nějaký opravdu velký průšvih. Některé reálné důsledky nepovedeného software uvádí Richta (2011) ve své přednášce jde například o:

Pád rakety Ariane 5: Díky chybě dal řídicí počítač rakety příkaz k současnému vychýlení trysek urychlovacích bloků, tak i trysky motoru. Tím se kurs rakety prudce změnil a v důsledku aerodynamických sil se horní část rakety odlomila. Byl aktivován vlastní autodestrukční systém rakety a raketa se změnila v oblak hořících úlomků. Celková cena: 100 mil. $ (včetně družic Cluster, které nesla, 500 mil. $).

Výpadek elektřiny USA (2003): Způsobena neregistrovaným výpadkem automatizovaného hlásiče poruch v elektrárně u Niagary, který se kaskádně rozšířil sítí. Výpadek vyřadil z provozu celkem 21 elektráren. Postiženo bylo až 50 mil. obyvatel, zemřeli 3 lidé, škoda 6 mld. $

Havárie přistávacího modulu na Marsu (1999) Problém komunikace mezi komponentami – uživatel rozhraní očekával hodnotu v kilometrech, poskytovatel ji udával v mílích. Namísto plánovaných 140 až 150 kilometrů tedy zamířila pouze 57 kilometrů nad povrch. V té výšce je však atmosféra Marsu na sondu příliš hustá. Climate Orbiter shořel nejspíše ve výšce kolem 80 kilometrů.
(Zdroj: Technet.cz)

1.1. Důvody vzniku softwarového inženýrství

Na začátku vývoje počítačů ve 40tých a 50tých letech byl jejich výpočetní výkon velmi nízký. Tento výkon se časem zvyšoval geometrickou řadou podle známého Moorova

zákona. Vývoj software byl v té době v plenkách. Nebyli známi žádné postupy, které vývoj systematizovali. V šedesátých letech při dalším zvýšení výkonu počítačů se pomalu začíná mluvit o softwarové krizi.

Charakteristickými znaky softwarové krize bylo neúnosné prodlužování a prodražování projektů, nízká kvalita programů, nesnadnost či nemožnost údržby a inovace, špatná produktivita práce programátorů, neefektivita vývoje, nejistota výsledku a řada dalších.

Příčin této krize bylo hned několik:

- Špatná komunikace mezi osobami tvořícími software a také mezi vývojáři a zákazníkem.
- Nesprávný přístup. Požadovaný software vyhotovoval a schvaloval vývojář a nespokojený zákazník byl označen za osobu, která tomu nerozumí.
- Špatné plánování celého projektu. Vývojáři předpokládali, že to nějak stihnou.
- Nesprávné odhady trvání vývoje, nákladů, rozsahu.
- Nízká produktivita práce. Programátoři se zabývali vším možným, jen ne dosti tím, čím měli.
- Neznalost základních pravidel. Např. Brooksův zákon z roku 1975: „Přidání řešitelské kapacity u zpožděného softwarového projektu způsobí jeho další zpoždění.“
- Podcenění hrozeb a rizik. Málo byly sledovány hrozby a místo snadného předcházení přerůstaly ve velké problémy.
- Nezvládnuté technologie. Falešná představa, že po zavedení nové technologie se potíže samy vyřeší.

Jako reakce byla uspořádána v roce 1969 konference, která zavedla základní principy softwarového inženýrství.

V té době byl definován takto:

Softwarové inženýrství je disciplína, která se zabývá zavedením a používáním řádných inženýrských principů do tvorby software tak, abychom dosáhli ekonomické tvorby software, který je spolehlivý a pracuje účinně na dostupných výpočetních prostředcích.“

Softwarové inženýrství bychom mohli popsat jako inženýrskou disciplínu, zabývající se reálnými problémy vývoje rozsáhlých softwarových systémů. Tato disciplína v sobě obsahuje několik různých myšlenkových postupů. Je to inženýrská disciplína, díky používání pragmatických postupů. Mezi které patří znalosti z oblasti specifikace požadavků na softwarový produkt, jeho analýzy, návrh, implementaci a testování a samotné zavedení systému u koncového zákazníka. Zároveň v sobě integruje manažerský přístup vedení projektu umožňující efektivní využití jednotlivých postupů, jejichž hlavním cílem je efektivně vytvořit kvalitní produkt software.

1.2. Psychologická odbočka

Kognitivní psychologie popsala obecné rozdíly v řešení problémů mezi experty na danou oblast a nováčky. Experti si dokáží vybavit částečná řešení podobných situací a použít předchozí znalosti ke klasifikaci a definici problémů (Eysenck a Keane, 2008). Experti řeší zadaný problém tak, že vezmou zadání problému, všechny vnější okolnosti a na základě těchto informací se snaží najít řešení. Nováčci se naopak nejdříve snaží vytipovat řešení, a poté se snaží najít ověření, zda lze ze známých faktů vyvodit jimi nalezené řešení. Používají tedy strategii zpětného posunu (Nolen Hoeksema at al., 2012). Dle Plháková je největší rozdíl mezi nováčky a experty v rozsahu organizaci znalostí (Plháková, 2003). Mezi znaky úspěšných jedinců (expertů) v odborných profesích patří velmi rozvinuté auto regulativní dovednosti (popisované v samostatné kapitole) umožňující efektivní uplatňování dovedností a vědomostí (Helus & Pavelková, 1992). Porozumění problémům bývá u laiků relativně povrchní. Znalci nejprve odhadnou podstatu problému a snaží se ji ujasnit a utřídit. Nepouštějí se do řešení, dokud nemají připraven efektivní plán. Odborníci věnují přípravě a plánování proporcionálně více času než laici, ale celkové řešení problému jim trvá kratší dobu (Plháková, 2003). Říčan (2016) experty popisuje: „ve své oblasti disponují nejenom větším objemem znalostí (kvantitativní stránka), ale rovněž se odlišují kvalitativně, a to ve smyslu více rozvinutého strategického chování během řešení úkolových situací (odlišují se tedy v jejich schématech v učení). Experti mají více a lépe organizované doménově-specifické znalosti a tím jsou schopni rychleji získat nové informace ze své oblasti erudice, jelikož předchozí znalosti, které jsou k dispozici, fungují jako integrační struktura pro nově příchozí informace prostřednictvím asociativního pojení pouze s minimem kognitivního úsilí.“

Obecně, tak lze říci, že softwarové inženýrství v sobě integruje strategie řešení problémů, které podle kognitivní psychologie náleží expertům.

2. Životní cyklus informačních systémů

Velmi trefné přirovnání používá Wikipedie, kdy životní cyklus přirovnává k mostu. Most je ve své podstatě prostředek k zjednodušení nějaké lidské aktivity. Podobný smysl by měly plnit i softwarové systémy. Stejně jako mosty se software musí navrhnout, používat a udržovat a na konci své životnosti pak demontovat. Jistě ve stavebnictví existuje životní cyklus staveb, který bude dosti podobný životnímu cyklu software.

Existuje velké množství klasifikací životního cyklu softwarových systémů. My jsme zvolili Systems Development Life Cycle(SDLC) od The Department of Justice USA z roku 2003.

- **Inicializace** (Initiation) – nejdříve je třeba, aby objekt mající dostatek financí (zákazník) měl potřebu nějakého zlepšení realizovaného pomocí software. Nazývejme jej záměrem.
- **Vývoj konceptu** (System Concept Development) - Tento záměr je přezkoumán z hlediska proveditelnosti a vhodnosti. Je na hrubo popsán rozsah systému na jehož základě jsou vyčísleny náklady, které musí být schváleny zákazníkem.
- **Plánování** (Planning Phase) - Tento koncept je dále rozvíjen tak, aby popsal, jak bude podnik fungovat po zavedení schváleného systému. Jsou naplánovány a schváleny konkrétní plány projektu.
- **Analýza požadavků** (Requirements Analysis Phase) - Všechny požadavky jsou definovány na úroveň detailů, která postačuje pro pokračování návrhu systému. Typicky vymezují požadavky z hlediska dat, výkonu systému, zabezpečení a požadavků na údržbu systému a podobně.
- **Design** (Design Phase) Fyzické charakteristiky systému jsou navrženy v průběhu této fáze. Jsou definovány hlavní části systému a jejich vstupy a výstupy. Vše, co vyžaduje vstup nebo schválení uživatele, musí být dokumentováno a přezkoumáno uživatelem.
- **Vývoj** (Development Phase) veškeré specifikace vytvořené v předchozích fázích jsou vloženy do vnikajícího software. Následně jsou testovány.
- **Integrační a testovací fáze** (Integration and Test Phase) Jednotlivé komponenty systému jsou složeny a systematicky testovány.
- **Implementační fáze** – systéme je nainstalován a zprovozněn u zákazníka. Zákazník si musí systém vyzkoušet a odsouhlasit. Fáze končí nasazením produktu do běžného provozu.
- **Provoz a údržba** (Operations and Maintenance Phase) Provoz systému by měl být monitorován pro odpovídající službu. V případě potřeby jsou zapracovány potřebné úpravy systému. Tento proces pokračuje tak dlouho, dokud systém může být účinně přizpůsobován potřebám zákazníka. Pokud proces již nelze dále přizpůsobovat anebo je to již příliš drahé, dochází k fázi plánovaní nového software.

- **Ukončení používání** (Disposition Phase) – Ukončení používání systému. Zvláštní důraz je kladen na řádné zachování dat zpracovávaných systémem, aby se údaje mohly efektivně přenést do jiného systému nebo archivovat v souladu s platnými předpisy a politikami správy záznamů pro případný budoucí přístup.

V tomto textu si podrobně popíšeme celý životní cyklus software. Ukážeme si jednotlivé metody vývoje software, abychom se následně podrobně zaměřili na metodu RUP (Rational Unified Process). Tato metoda podrobně pokrývá z výše uvedených od fáze Inicializace až po fázi implementační. Metoda RUP hodně pracuje s UML, proto si ukážeme celkem šest různých druhů diagram UML. Seznámíme se s modelováním firemních procesů (Business Process Model and Notation), které může být užitečné na začátku tvorby nového softwarového systému. Samostatně si pak popíšeme testování software a jeho údržbu pomocí metodiky ITIL.

3. Metody vývoje softwaru

Je v obecné rovině souhrn postupů, pravidel a nástrojů vedoucích k tvorbě software. Občas je též nazývaný softwarový proces. Jednotlivé nástroje určené pro tvorbu software se nazývají Framework. Cílem těchto postupů je efektivní vývoj a udržování software.

Využívání těchto postupů by mělo snížit riziko nepředpokládaných problémů a navyšuje přehlednost harmonogramu práce na daném software. Pokud jednotlivý vývojáři využívají stejné postupy. Výrazně se jim usnadní spolupráce díky dopředu daným pravidlům vývoje programu.

Postupem času vzniklo velké množství metod softwarového vývoje. Dodnes však neexistuje detailně a přesně definovaná podoba metody vývoje software, který by mohl přijat jako referenční. Patří mezi ně například:

3.1. Vodopádný přístup

je postupný vývojový proces, ve kterém je na vývoj pohlíženo, jako postupnou proces, kdy se postupně prochází jednotlivé jeho části: návrh, implementace, testování, integrace a údržba, případně předávání. Tento model vznikl již v sedmdesátých letech. A dá se označit, jako základní model, od kterého se odráží ostatní modely.

Přechod z jedné fáze do druhé se realizuje až v momentě částečného, nebo úplného dokončení předchozí fáze. V reálném životě ale bohužel často tento model není možné využít. Například specifikace programu často zákazník mění až v průběhu tvorby tohoto programu anebo až při jeho využívání. Jako problém tohoto modelu se dají označit:

- Není možné v průběhu vývoje software odhadnout výslednou kvalitu produktu, na základě splnění předem daných kritérií na specifikaci software
- Výsledný produkt záleží na kvalitě zadání na výsledný software
- Čas potřebný k vývoji software je příliš dlouhý

Snaha o opravení nedostatků v tomto přístupu vedla ke vzniku dalších přístupů. Tyto procesy popisuje prof. Vondrák (2002):

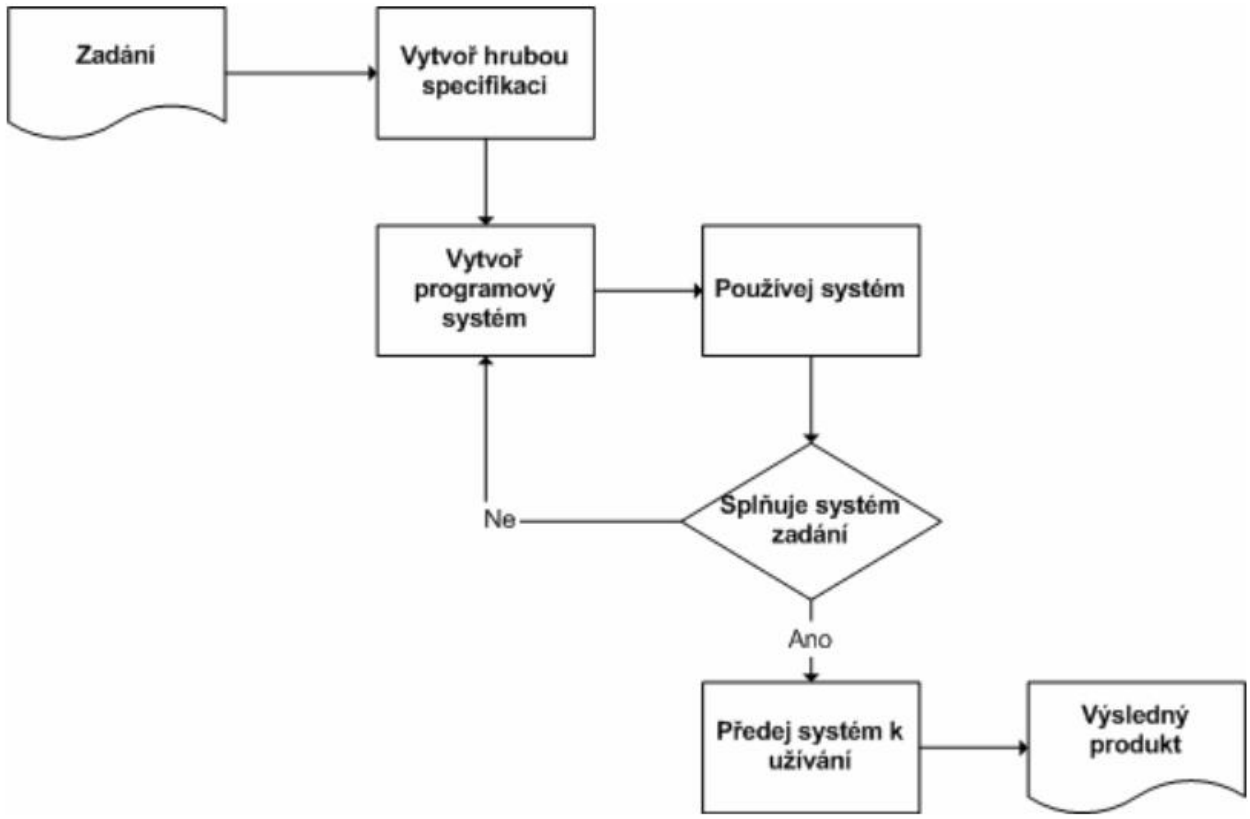
3.2. Iterativní přístup

Jednotlivé aktivity vývoje se opakovaně vykonávají v „iteracích“. Kdy se v každé iteraci přidá do vznikajícího software vybraná malá skupina funkcí vedoucích k cílovému stavu. Pomalu tak vyrůstá požadovaný program a zpřesňuje se jeho funkcionalita. Zákazník tak má možnost vyzkoušet si vyvíjený software již v rané fázi vývoje. Může tak rychleji upřesňovat své požadavky na vyvíjený software. Případné chyby, nedodělky, či chybně zadané požadavky na program se tak odhalí výrazně dříve. Díky tomu, že zákazník vidí

software již od raného vývoje, může více ovlivňovat k obrazu svému výslednou podobu produktu. V tomto přístupu se poněkud smazávají rozdíly mezi jednotlivými fázemi, tak jak o nich mluví vodopádný přístup. Spíše se nabízí paralela, kdy každá iterace je jeden malý vodopád. Do této kategorie spadá i proces RUP, kterému se budeme věnovat v samostatné kapitole.

„průzkumné“ programování

prof. Vondrák (2002) ještě zmiňuje, jako odstrašující model, který je občas využíván v praxi. Tento model nazývá průzkumné programování.



3.3. Agilní techniky

Většina softwarových procesů historicky vznikla ve velkých korporacích. Tyto postupy vyžadovali striktní dodržování naplánovaného postupu a povinné vytváření celé řady dokumentů. Což je v pořádku v případě velkých firem, kdy na projektu pracuje velká spousta lidí, nebo je třeba vysoký stupeň spolehlivosti (například řízení rozvodné elektrické sítě). V případě menších projektů ale čas strávený dodržováním všech postupů a jiných formalit byl mnohdy delší než čas určení pro programování a testování.

Jako reakce tak vzniká na začátku nového milénia proces Agilní metody, či techniky. Základní principy těchto technik byly v roce 2001 formulovány v manifestu agilního vývoje software.

- Lidé a jejich interakce jsou důležitější než procesy a nástroje

- Fungující software je důležitější než podrobná dokumentace
- Spolupráce se zákazníkem je důležitější než uzavřené smlouvy
- Reagování na změny je důležitější než dodržování plánu

Hlavní idea těchto metod je umožnit vývojářům flexibilní nesvazující přístup k tvorbě software. Vývojář se zaměřuje na vývoj software, jeho navrhování a dokumentování již tolik neřeší. Tyto metody se řadí mezi iterativní metody. Je založen na aktivní spolupráci se zákazníkem. Rozpis práce se většinou řeší v horizontu další iterace. Ze své trošku trochu „anarchistické“ povahy jsou tyto techniky vhodné pro menší kvalitní vývojářské týmy. Na členy týmu jsou kladeny zvýšené nároky v oblasti komunikace a spolupráce.

V současné době existuje několik agilních metod. Mezi nejznámější patří extrémní programování a Scrum (Sklenář, 2007)

4. RUP

Proces RUP (Rational Unified Process) vnikal v druhé polovině devadesátých let ve spolupráci několika firem pod vedením firmy Rational. Tato firma se pak v roce 2003 stala divizí IBM. RUP není jediným konkrétním předepsaným procesem, nýbrž adaptivním iteračním procesním rámcem, který má být přizpůsoben potřebám vývojových organizací a týmů vyvíjejících software. Toto přizpůsobení je realizováno tak, že vývojové týmy si vybírají prvky procesu, které odpovídají jejich potřebám. RUP vychází z UP (inified Process). RUP patří v současné době mezi výrazně rozšířené modely softwarového procesu. Podporuje jej mnoho nástrojů.

V klasickém vodopádném přístupu jsou běžně rozděleny jednotlivé role. Stává se tak, že analytik od zákazníka přebíral představu fungování systému. Tuto představu následně přepíše do dokumentu, který předá programátorovi. Pokud dochází k absenci pravidelné spolupráce mezi analytikem a vývojářem. Vývojář si musí „dovozovat" požadavky na systém z požadavků sepsaných v dokumentu. Velmi jednoduše si pak může tyto požadavky vyhodnotit jinak, než zamýšlel zákazník.

Tento proces je postaven na několika zásadách. Tento proces je postaven na ideji **iteračního vývoje** software. Díky tomu by měl být každý program vyvíjen ve verzích (iteracích) Každá tato iterace by měla být vytvořena spustitelným kódem.

Žáček (2017) uvádí základní principy RUP:

- Snahu minimalizovat rizika projektu co nejdříve a neustále.
- Ujištění se, že dodáváme zákazníkovi přidanou hodnotu.
- Zaměření na spustitelný software.
- Zapracovat změny v časných fázích projektu.
- Brzké nastínění spustitelné architektury.
- Znovupoužití existujících komponent.
- Úzká spolupráce, všichni jsou jeden tým.
- Kvalita je způsob provádění celého projektu, nejen část (testování).

Prvotní specifikace na software je zásadním předpokladem pro kvalitní vývoj produktu. RUP vytváří **Systém správy požadavků.** Od jejich získání, dokumentaci, po monitorování změn v požadavcích. Tento systém následně tvůrci software v komunikaci se zákazníkem. Stejně tak jsou řízeny změny ve způsobu vývoje software.

Mnohé části softwarových produktů jsou si podobné. Proto je ztráta času je znovu vyvíjet. RUP zavádí **vývoj na základě komponent**. V momentě, kdy máme k dispozici potřebné komponenty, vývoj produktu se stává skládáním jednotlivých komponent dohromady. Toto skládání můžeme připodobnit například k legu, anebo ke skládání stolního počítače.

Díky komplexnosti vyvíjených programů je pro lepší představení důležitá **vizualizace softwarového systému**. RUP pracuje s jazykem UML.

Pro zajištění spokojenosti zákazníka díky funkčnímu software a pro potřeby zpětné vazby pro vývojáře je důležité **ověření kvality vytvářeného software**. Toto ověření by se mělo týkat všech vývojářů produktů. RUP specifikuje specifické postupy popisující kvalitu vytvářeného software.

Tvorba software je v RUP rozdělena celkem do čtyř fází. Fáze v RUP bychom mohli pojmenovat jako jednotlivé statusy projektu, jeho změn v čase. Každá fáze často obsahuje několik iterací.

4.1. Schématický model

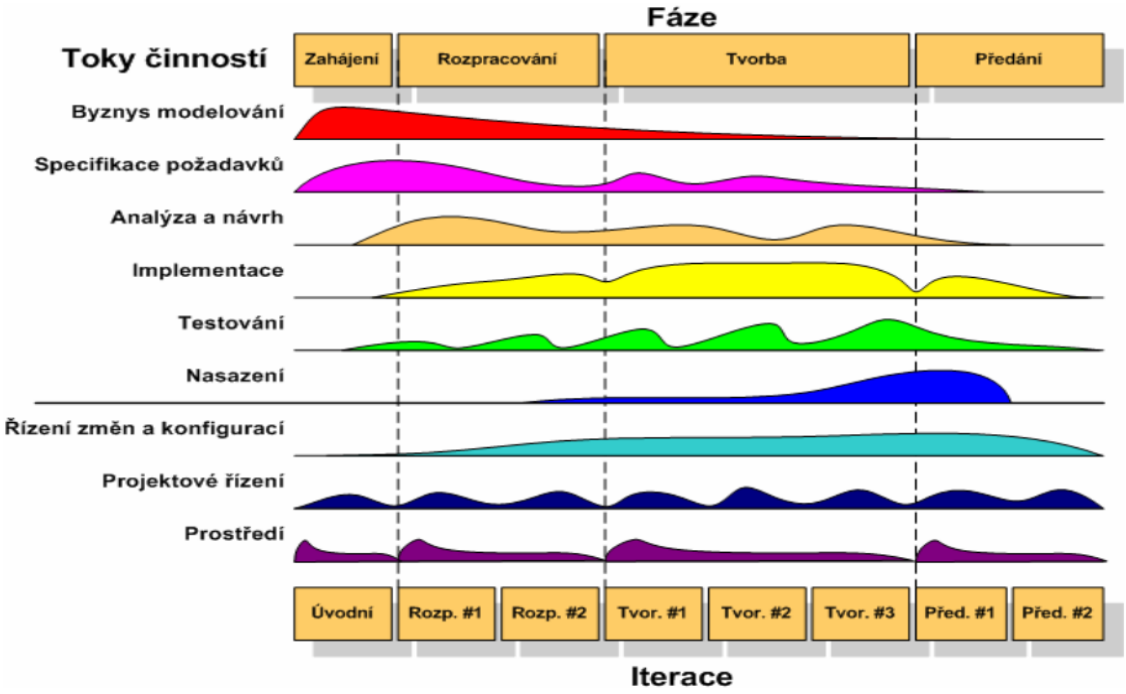
Samotný schématický model se sestává u celkem šesti inženýrských disciplín a tří. Mezi inženýrské disciplíny patří:

- Byznys modelování
- Specifikování požadavků
- Analýza a návrh
- Implementace
- Testování
- Nasazení

Do pomocných disciplín spadá:

- Správa konfigurací a změn
- Řízení projektu
- Příprava prostředí pro projekt

Přibližnou četnost aktivit u jednotlivých disciplín v závislosti na jednotlivých fázích a iteracích projektu vypracoval prof. Vondrák (2002).



Přibližné časové a zdrojové rozdělení je zobrazeno vypadá asi takto:

	Zahájení	Rozpracování	Tvorba	Předání
Zdroje	Cca 5%	20%	65%	10%
Čas	10%	30%	50%	10%

Žáček (2017) uvádí, jaké jsou výstupy jednotlivých fází:

- Výstupem Inception (zahájení) je pochopení problematiky, vize projektu, identifikovaná rizika.
- Výstupem Elaboration (rozpracování) je spustitelná, otestovaná architektura (=
- fungující část aplikace).
- Výstupem Construction (tvorba) je beta-release aplikace, relativně stabilní, opět spustitelná, téměř kompletní aplikace.
- Výstupem Transition (předání) je pak již produkt připravený k finálnímu nasazení včetně veškeré dokumentace a hardware. Zásadní rozdíl je také v tom, že každá fáze může

Na jednotlivých fázích se běžně podílí různé počty zaměstnanců a různých pozic. Je ale důležité, aby tito zaměstnanci stále pracovali spolu v jednom či více týmech. Pokud je to potřeba jsou si navzájem k dispozici k dovysvětlení nejasností.

4.2. Zahájení (Inception)

Na začátků je třeba stanovit rozsah projektu, posoudit podmínky, rizika, časové a finanční náklady realizace celého projektu. Žáček (2017) uvádí celkem 5 cílů této fáze:

- Porozumění tomu, co vytvořit – vytvoření vize, definice rozsahu systému, jeho hranic; definice toho, kdo chce vytvářený systém a co mu to přinese.
- Identifikace klíčových funkcionalit systému – identifikace nejkritičtějších Use Casů.
- Návrh alespoň jednoho možného řešení (architektury), (tedy zda je vůbec možné pokračovat ve vývoji pozn. autor)
- Srozumění s náklady, plánem projektu, riziky.
- Definice/úprava procesu, výběr a nastavení nástrojů.

Definice a význam požadavků

V rámci zjišťování požadavků se snažíme o získání maximálního množství informací o tom, co uživatelé od nového systému očekávají a jaké činnosti bude systém primárně vykonávat. Mezi možné techniky zběru informací například patří:

- Studium dokumentů
- Rozhovory s budoucími uživateli
- Dotazníky
- Pozorování
- Nahrávání aktivit uživatele

Tyto požadavky se dají rozdělit na:

- **Funkční požadavky**. Tedy jaké jsou požadavky na funkčnost systému ze strany systému, případně se specifikuje i co by systém neměl umět. Základním způsobem pro popis funkčních požadavků je graf způsobů užití.
- **Nefunkční požadavky.** Jsou to požadavky na vlastnosti systému jako celku. Mohou se týkat výkonosti, spolehlivosti, bezpečnosti, kapacity systému, případně se zde mohou být organizační nároky, jako dodržované standardy a postupy práce. Případně zde mohou být zahrnuty požadavky týkající se platné legislativy.

Cíle

Hlavním cílem je pak vytvoření vize projektu. Tato vize by měla být definována z pohledu uživatele. Zároveň by ale měla obsahovat základní technický náhled na využité technologie, či komponenty architektury. V této vizi by měly být hlavně popsány požadavky na budoucí software. Tato fáze běžně proběhne jenom jednou. Ve složitějších případech ale může proběhnout vícekrát.

Zároveň by zde měla být zhodnocena míra rizik v oblastech:

- Rizika technického směru (technologií, kompatibilitu, komunikaci s ostatními systémy)
- Finanční rizika – náklady na vývoj a následnou údržbu

- Časová rizika – přičemž čím více budeme mít zdrojů, tím pravděpodobně kratší bude samostatný vývoj

Na konci této fáze je třeba posoudit, zda není vhodné skončit vývoj. Čím dříve ukončíme vývoj problematického projektu, tím jsou náklady na vývoj tohoto projektu nižší. K tomuto účelu se používá Lifecycle Objective Milestone (LOM) jeho základní kritéria jsou:

- Shoda zúčastněných stran na rozsahu, nákladech a časovém harmonogramu projektu
- Shoda na tom popsání správných požadavků na systém a chápání těchto požadavků je shodné
- Shoda správnosti finančních a časových odhadů, prioritách, rizicích a odpovídajícím vývojovém procesu
- Shoda v odhadu všech rizik a strategiích jejich snižování

4.3. Rozpracování (Elaboration)

Ve fázi zpracování se projekt začíná formovat. Cílem této fáze je popsat architekturu systému, abychom na jejím základě mohli v následující fázi navrhnout většinu funkcionalit. Tato architektura se vyvíjí na základě poznatků z předchozí fáze. Řeší se zde hlavní problémy týkající se návrhu architektury vznikajícího software. Funkčnost systému může být pouze přibližná. Většina kritických funkcionalit by již ale měla být do systému integrována. Řešitelský tým by si měl v průběhu této fáze klást otázky týkající se stability návrhu.

V této fázi se provádí analýza problému a architektura projektu získává základní podobu. Vhodným návrhem architektury se snažíme minimalizovat rizika spojená se vznikem software. Jde především o rizika:

- Finanční a časová
- Architektonická
- Technická a procesní
- Zaměření vyvíjeného software

Pomocí několika iterací v této části projektu můžeme snížit míru výše uvedených rizik. Hlavně pak správnost zaměření vývoje. Pokud tvoříme systém s využitím podobných technologií, jaké jsme již tvořili, existuje logicky menší množství potencionálních rizik. Můžeme si proto i dovolit naplnit cíle této fáze pouze v jedné iteraci. Naopak pokud technologii neznáme nebo je projekt z nějakého důvodu složitější (např. vyšší bezpečnostní požadavky). Pracujeme ve dvou, či více iteracích. V případě zásadnější změny architektury v průběhu této fáze je rozumné přidat další iteraci, která ověří dostatečnou funkčnost a stabilitu pozměněné architektury. Podle zásady, čím později budu zásadně měnit základy celé stavby, tím nákladnější bude celá oprava.

Pro ověření správného zaměření vyvíjeného software může být přínosné vytvořit beta

verzi uživatelského rozhraní, na kterém se otestují nejdůležitější funkcionality systému.

Na konci této fáze musí být architektura ve stabilním tvaru. Pro její ověření je třeba spustitelná architektura, na které ověříme kritické funkcionality systému. Tyto kritické funkcionality je také vhodné vymodelovat třeba například pomocí sekvenčního diagramu v UML. Snažíme se nalézt potencionální problémy a rizika. Následně bychom měli seskupit vybrané třídy do balíčků s ohledem na pravidla viditelnosti a budoucí konfiguraci produktu. Měli bychom si vytvořit představu, jak bude nakládáno s daty v databázi.

Pravidelně bychom pak měli provádět integraci komponent. Tedy určovat v jakém pořadí a jak budou vybrané komponenty integrovány.

Na závěr velmi důležitou částí je testování kritických scénářů. Testování kritických scénářů může být ukazatelem našeho pokroku ve fázi rozpracování. Mimo jiné je vhodné v této fázi testovat v kritických scénářích takové aspekty jako je velká zátěž systému, dostatečná výkonost architektury, či spolupráci s externími systémy.

Na konci této fáze dochází opět k otestování pomocí Lifecycle Architecture milestone (LCA) základními evaluačními kritérii LCA jsou:

- Jsou vize výroby a požadavky na výrobek stabilní?
- Je architektura stabilní?
- Jsou otestovány klíčové postupy a přístupy? A tím je ukázána jejich použitelnost.
- Pomocí testování spustitelného prototypu byly popsány hlavní rizikové části projektu. Byly tyto části následně vyřešeny
- Jsou připraveny iterační plány pro následující fázi v takové podobě, že podle nich půjde postupovat?
- Jsou tyto plány podpořeny důvěryhodnými odhady?
- Všichni zúčastnění se shodují v tom, že současná vize může být naplněna, pokud bude současný plán dotažen v kontextu současné architektury?
- Jsou akceptovatelné současné náklady oproti plánovaným?

Projekt může být přerušený nebo výrazně přehodnocen, pokud nedosáhne tohoto milníku.

4.4. Konstrukce (Construction)

Hlavním cílem této fáze je vybudovat reálně fungující softwarový systém a minimalizace nákladů na jeho tvorbu.

V této fázi se hlavní pozornost soustředí na vývoj komponent a dalších prvků systému. Toto je fáze, kdy dochází k větší části kódování, díky čemuž je lidsky (vyžaduje hodně programátorů a testerů) i časově náročná. Dochází zde k návrhu zbylých funkcionalit tedy zbytku hlavních (požadovaných zákazníkem) a většiny ostatních. Pokud jsou zde

zásahy do architektury systému, pravděpodobně byla špatně provedena předchozí fáze.

Hlavními předpoklady pro úspěch této fáze je: kompaktnost architektury, paralelní vývoj jednotlivých týmů, management konfigurací a změn a automatizované testování.

U větších projektů je možné projít několik iterací (běžně 2 – 4). Většinou v této fázi projektu bývá nejvíce iterací. Plánování iterací odpovídá počtu a typu ještě neimplementovaných funkcionalit. V každé iteraci se řeší individuální segment projektu. V ideálním případě je z předchozích fází vytvořena architektura, která se dále rozšiřuje, využívá, či znovu využívá. Probíhá integrace a testování systému. Organizačně je ideální jeden tým zodpovědný za architekturu a několik dalších týmů, které paralelně pracují na vývoji subsystému. Tyto „paralelní týmy“ hlavně komunikují s týmem zodpovědným za architekturu software.

Díky iteračnímu vývoji vzniká velké množství soborů, jejich verzí, které jsou následně testovány a integrovány. Z tohoto důvodu musí být zaveden management konfigurací a změn. Který jednotlivé změny a konfigurace zaznamenává. Pokud takový systém funguje, mohou se vývojáři věnovat vývoji, což jim ušetří čas, tedy zvýší efektivitu jejich práce.

Na konci této fáze musí být hotova beta verze programu včetně nápovědy v aplikaci, instalačních instrukcí, uživatelských manuálů a tutoriálů. Tak aby vybraní budoucí uživatelé mohli vyzkoušet software a dát nám relevantní zpětnou vazbu.

Na konci této fáze bychom si měli položit několik otázek podle milníku IOC (Initial Operational Capability)

- Je beta verze vhodná k uvolnění prvním uživatelů (testerům)?
- Jsou uživatelé připraveni a schopni používat naši beta verzi?
- Jsou akceptovatelné současné náklady oproti plánovaným?

4.5. Nasazení (Transition)

- Cílem této fáze je přesun systému z jeho vývoje do reálného využití a vychytání jeho posledních problémů (nejčastěji z pohledu výkonosti, uživatelské přívětivosti a funkcionality). Činnosti této fáze zahrnují:
- školení koncových uživatelů a správců
- testování systému, aby se ověřilo naplnění očekávání s koncových uživatelů.
- Příprava marketingových materiálů
- Příprava prostředí a dat, například migrace dat ze starého do nového systému
- Produkt je dále kontrolován v kontextu kvalitativních rámců vytvořených na začátku projektu.
- Vnitřní zhodnocení projektu, ponaučení se z chyb a problémů vzniklých při práci na projektu

Anglická wikipedie ze dne 24. 6. 18 dále uvádí: „Systém také prochází evaluační fází, každý vývojář, který nevytváří potřebnou práci, je nahrazen nebo odstraněn.“

I tato fáze by měl být zakončena milníkem. Tentokrát PRM (Product Release Milestone) Kritéria tohoto milníku jsou:

- Jsou uživatelé spokojeni
- Jsou akceptovatelné konečné náklady oproti plánovaným? Pokud co by se mělo změnit, aby se tomuto problému předešlo?

5. Business Process Model and Notation

Primárním cílem bylo poskytnout grafický nástroj, který je jednoduše pochopitelný všemi firemními uživateli (od analytiků, přes vývojáře až po vlastníky podnikových procesů). Díky čemuž se výrazně zjednoduší komunikace mezi vývojářem a zákazníkem. Business Process Model and Notation (Dále již jen BPMN) se skládá z množiny jednoduchých grafických prvků, ze kterých se dá vymodelovat firemní procesní model. BPMN je vyvíjen od roku 2005. Někdy v roce 2011 nastupuje verze 2.0. Procesní diagram je založen na vývojovém diagramu, který je velmi podobná diagramu aktivit z UML, o kterém budeme také mluvit.

Tokové objekty (Flow Objects)

Bezprostředně souvisí s tokem informací v procesu. Jsou to nejdůležitější grafické prvky. Spadají do nich tři skupiny prvků. Události, aktivity a brány.

Událost (Event)

Značíme je kolečkem, do kterého může být zobrazena různá ikona. Značí děj, který přímo ovlivňuje tok procesu. Události dále dělíme na **počáteční** (Start event), které iniciují vznik procesu a značí se kruhem s jednoduchým okrajem, někdy také zelenou barvou.

Dále máme **konečné události** (End event), které značí ukončení procesu, anebo výsledek aktivity. Je znázorněn tučným okrajem kolečka anebo tučnou ikonou uvnitř kolečka, občas také červenou barvou. V některých zdrojích se dále uvádí střední událost, která se pak značí dvojitým kruhem

Aktivita (Activity)

Značíme ji obdélník s kulatými rohy. Tento prvek znázorňuje činnost či práci. Aktivita může být buď **atomická** (tzv. Task), pak na ni pohlížíme jako na dále nedělitelný celek. Který se vždy musí vykonat celý.

Případně v sobě může aktivita obsahovat samostatný proces, pak ji nazýváme Subproces. Tento druh aktivity využijeme u procesů, u kterých nechceme, aby byly v dané úrovni znázorněny. Aktivitu Subproces značíme malým plus dole v zaokrouhleném obdélníku.

Brána (Gateway)

Značíme je čtvercem či kosočtvercem, stojícím na špičce. Označují větvení či souběh toků procesu, např. rozhodování či paralelní zpracování.

Spojovací objekty (Connecting Objects)

slouží ke spojení tokových objektů, či s artefakty. Díky spojení vzniká struktura (kostra) podnikového procesu. Spojovací objekty se dělí na 3 základní typy:

- **Sekvenční tok** (Sequence Flow) - plná čára s plnou šipkou. Určuje pořadí, v jakém

se mají jednotlivé řásti grafu vykonat

- **Tok zpráv** (Message Flow) - přerušovaná čára s prázdnou šipkou. Zobrazuje tok zpráv mezi jednotlivými účastníky procesu
- **Asociace** (Association) – tečkovaná čára. Spojuje objekt s nějakou dodatečnou informací. Využívá se také pro zobrazení vstupů a výstupů.

Plavecké dráhy (Swimlanes)

Zobrazuje účastníky procesu, nebo uspořádání činnosti v procesu typicky podle rolí. Používají se dva typy plaveckých drah.

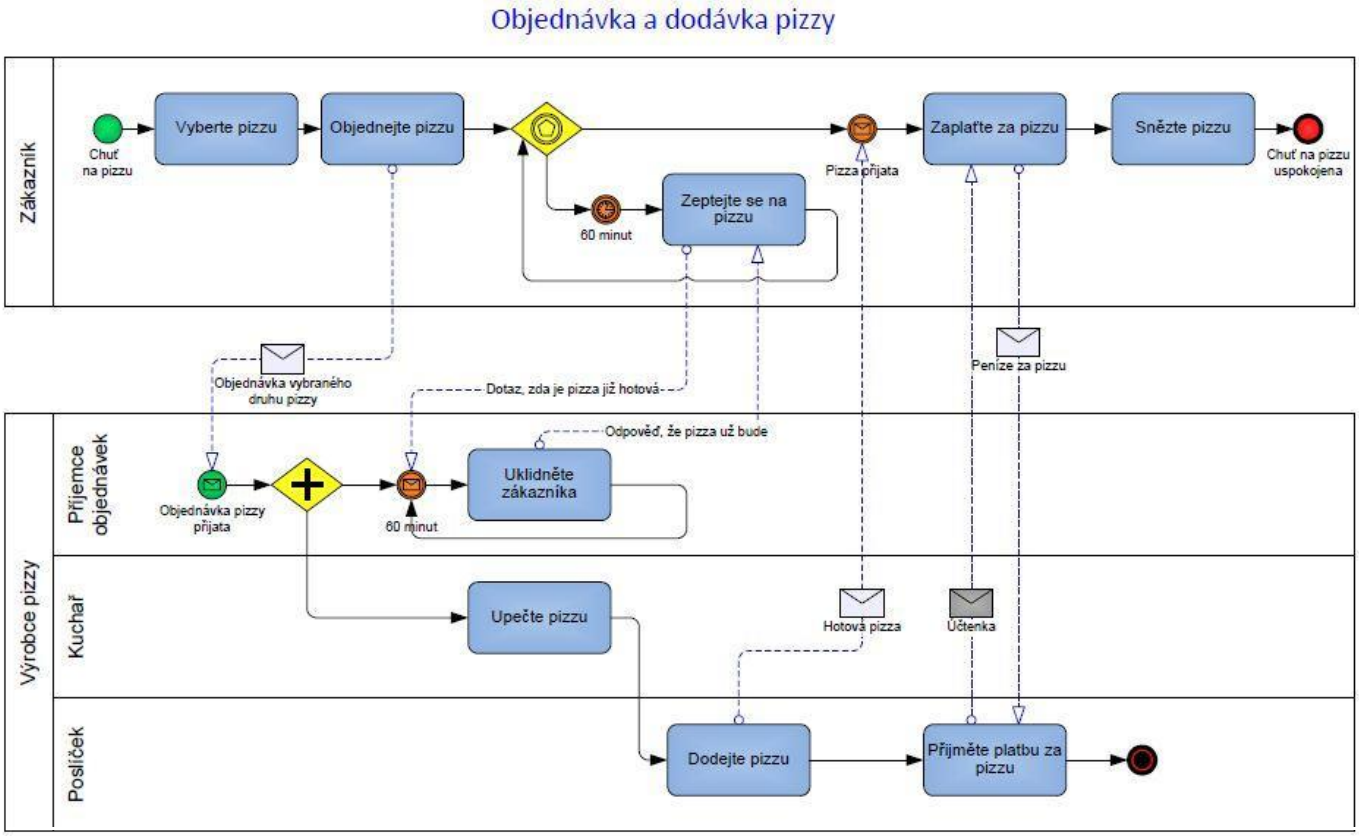
- **Bazén** (Pool) - Zobrazuje účastníky procesu, případně odděluje různé části organizace. Může mít více drah. V jednom bazénu se nachází jeden samostatný proces.
- **Dráha** (Lane) – pod částí bazénu. Používá se pro organizaci a kategorizaci aktivit

Artefakty (Artifacts)

Nějaké další upřesňující informace pro proces. Bez vlivu tok.

- **Datový objekt** (Data Object) – značíme obdélníkem se zahnutým rohem (list papíru). Značí data, se kterými pracují aktivity. Případně jsou nezbytná pro vykonání činnosti.
- **Seskupení** (Group) – značíme obdélníkem kresleným přerušovanou čárou. Seskupují různé objekty.
- **Poznámka** (Annotation) – Poznámky celému diagramu dodávají srozumitelnost a přehlednost

Jako vzor přikládáme model objednávky a dodávky pizzy.



6. UML

Mnohé softwarové systémy jsou velmi složité. Jejich vývoj tak klade velké požadavky na vývojářovu představivost. V momentě kdy na projektu pracuje více lidí, je třeba budoucí softwarový systém nějak exaktně popsat – modelovat. Existuje velká spousta přístupů k modelování. Každý z nich má své klady i zápory. Pro spolupráci více vývojářů je ale důležité, aby všichni znali, používali a hlavně stejně chápali modelovací systém a terminologii. Z tohoto důvodu se rozšířil UML (Unified Modeling Language, unifikovaný modelovací jazyk). Standard UML definuje standardizační skupina Object Management Group (OMG).

UML je jednotný jazyk pro vytváření určený k diagramů. UML jazyk umožňuje specifikaci (struktura a model), vizualizaci (diagramy), konstrukci (Software development methods) a dokumentaci artefaktů softwarového systému. Existuje velká řada diagramů. Každý z těchto diagramů se používá za rozdílným cílem a v rozdílných částech projektu. UML celkem poskytuje 13 druhů diagramů. V tomto textu budeme diagramy dělit na Strukturální diagramy a Diagramy tříd.

Strukturární diagramy

(Structure diagrams) - popisující strukturu projektu. Často se používání při popisu architektury vyvíjeného software. Ukážeme si několik vybraných diagramů spadajících do této kategorie

Diagram tříd (class diagram)

Diagram ukazuje členění tříd v projektu, vztahy mezi nimi, operace, nebo metody a jejich metody. Tyto části se často zapisují do obdélní vertikálně rozděleného na 4 části. Tento diagram je důležitým stavebním kamenem v objektově orientovaném modelování. Může se také sloužit k datovému modelování. Diagram zobrazuje statickou strukturu modelovaného systému. A je specificky závislý na konkrétním programovacím jazyku. Cílem této aktivity je ukázat jakým způsobem bude produkt realizován v implementační fázi.

Níže příklad diagramu tříd.

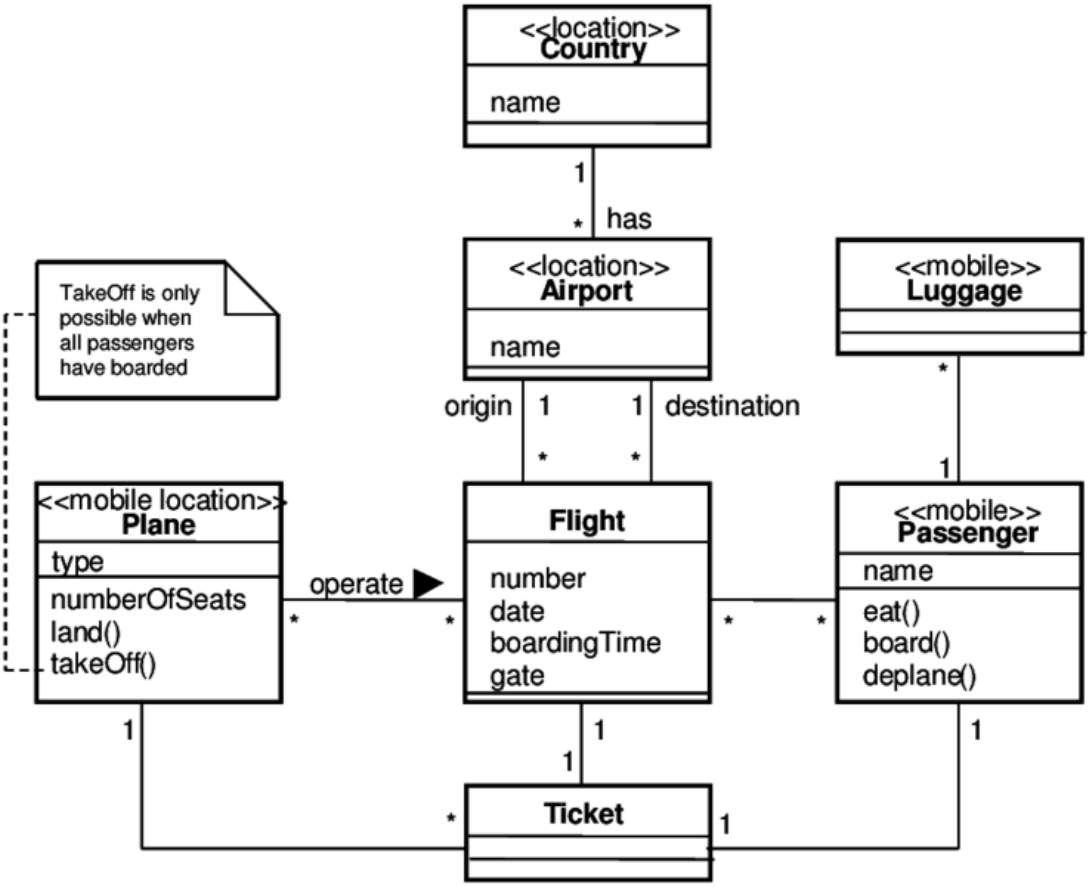


Figure 3 Zdroj: Andrade et al, Recent Trends in Algebraic Develeopment Techniques—16th International Workshop, WADT 2002, Frauenchiemsee, Germany. Vol. 2755 of LNCS.

Diagram komponent

Diagram pracuje s komponentami využitými v systému a znázorňuje, jak jsou komponenty propojeny k vytvoření větších komponent nebo softwarových systémů. Jsou používány k popisu struktury různě složitých systémů. Tento diagram se využívá hlavně při tvorbě software pomocí komponent. Tento diagram má vyšší úroveň abstrakce, než diagram tříd. Většinou komponenta v sobě implementuje jednu anebo více tříd.

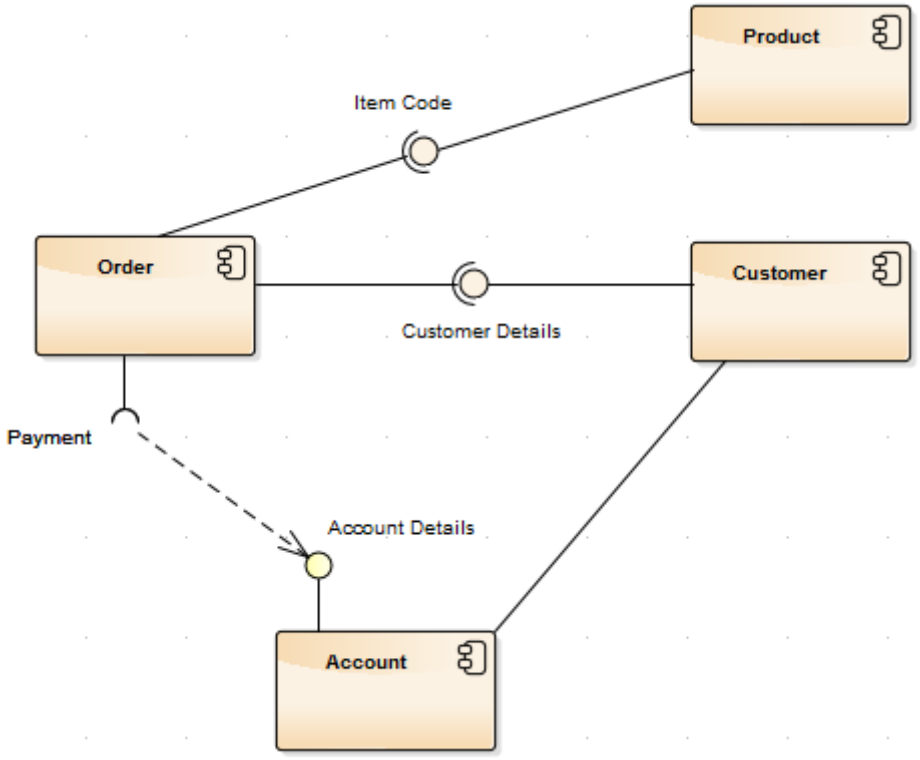
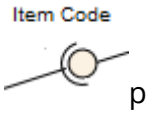


Figure 4 Zdroj:

http://sparxsystems.com/enterprise_architect_user_guide/13.0/model_domains/componentdiagram.html

V našem obrázku máme celkem 4 komponenty.



Montážní konektory (Assembly connectors) propojují rozhraní objednávky s produktem a zákazníkem. V našem případě komponenta vyžaduje informace ID produktu, a detaily zákazníka.

Prostá čára mezi komponentami Zákazník a učet zákazníka znázorňuje vztah mezi komponentami.

Objektový diagram

Objektový diagram znázorňuje objekty (instance tříd) a vztahy mezi nimi (links – instance asociací) právě v jednom okamžiku. Tento diagram vypadá podobně jako diagram tříd a vlastně může být jeho speciálním případem. Hlavně se využívá, pokud chceme ukázat příklady datových struktur v jeden konkrétní moment.

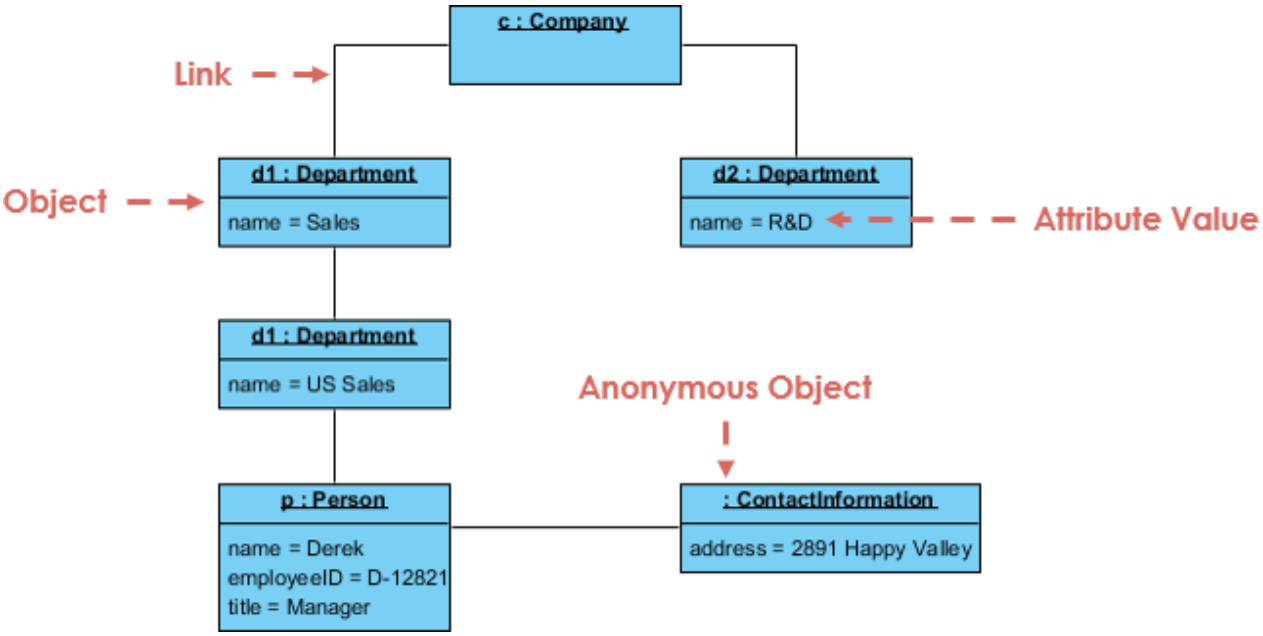


Figure 5 Zdroj: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-object-diagram/

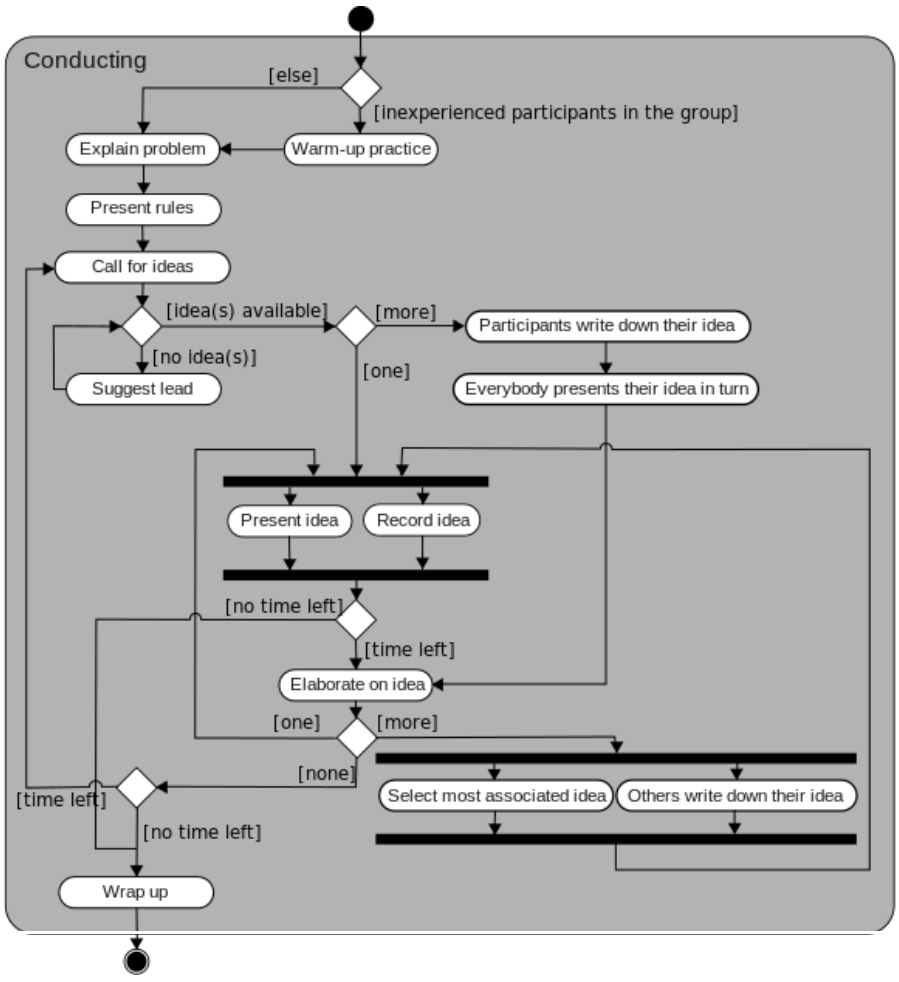
7. Diagramy chování (Behavior diagrams)

Popisující jednotlivé procesy pomocí aktivit reprezentujících jeho stavy a přechody mezi nimi. Vzhledem k tomu, že diagramy aktivit popisují chování systému, používají se k popisu funkčnosti softwarových systémů.

Diagram aktivit (Activity diagram)

Diagram je určen k modelování výpočetních či organizačních procesů (tj. Pracovních postupů) lze jej použít i k modelování datových toků. Diagram modeluje procesy pomocí aktivit reprezentujících jeho stavy a přechody mezi jednotlivými stavy. Díky tomu je jedním z diagramů, které umožňují zachytit chování systému.

- zaoblené obdélníky představují akce
- kosočtverce představují rozhodnutí
- čáry se šipkami spojují jednotlivé akce
- černý kruh představuje počátek (počáteční uzel)
- zakroužkované černý kruh představuje konec (konečná uzel)



Sekvenční diagramy

Diagram sekvence zobrazuje interakce objektů, které se podílejí na konkrétní funkcionalitě systému. Krom objektů a tříd zobrazuje posloupnost zpráv nutných k dosažení dané funkcionality, mezi jednotlivými objekty a to v podobě, která řeší hlavně jejich časovou návaznost. Tyto diagramy se často objevují v souvislosti s realizací případů použití v logickém pohledu na vývojový systém. Cílem této aktivity je ukázat jakým způsobem bude produkt realizován v implementační fázi.

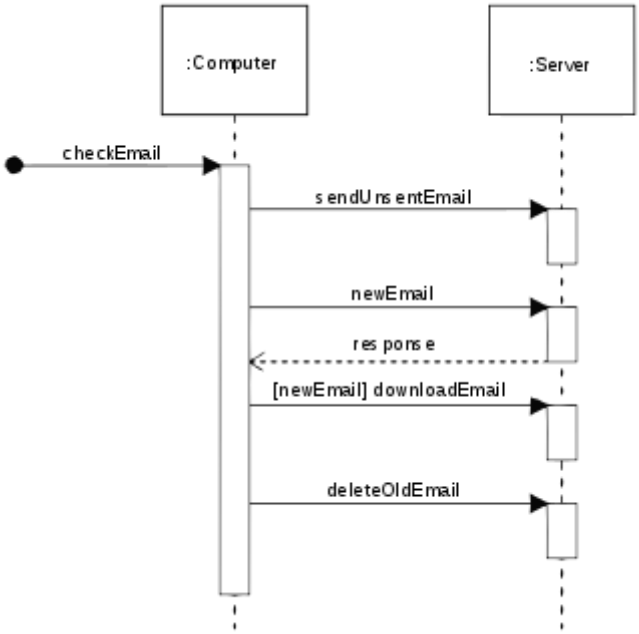


Figure 7 Zdroj: https://en.wikipedia.org/wiki/Sequence_diagram#/media/File:CheckEmail.svg

Časová osa v diagram je svislá (čas běží zhora dolů). Objekty jsou rozmístěny vodorovně. V našem obrázku máme 2 objekty (Server a počítač) čárkovaná svislá čára také nazývá čára života. V našem případě objekty existovali a budou existovat před i po diagramu. Obdélníky na čáře života znázorňují aktivitu. Vodorovné čáry se šipkou znázorňují zprávy. Šipka od koho komu zpráva půjde. Plná čára značí povinou zprávu. Čárkovaná nepovinou.

Diagram případů užití (Use case diagram)

Nejčastěji znázorňuje vztah mezi zákazníkem a systémem. Kdy znázorňuje různé případy využití systému zákazníkem. Což je často slovně popsáno pomocí scénářů. Pro převod scénářů do diagramů případů užití se využívají v čase realizace. Jde o algoritmus přepisu bodů scénáře do diagramu.

Vstupem pro vznik tohoto diagramu může být například BPMN. Výsledkem je seznam aktivit, které může například místo lidí vykonávat právě vznikající software. Postavičky v diagramu se nazývají aktér (actor). Aktéři jsou tedy propojeni s těmi případy užití, které se jich týkají. Jednoduchá čára se nazývá asosiace. Horizontální směr dále naznačuje strukturu aktérů. Spodní aktér dědí vlastnosti aktérů nad ním. Tento typ vazby je nazýván generalizace. Vazba include se realizuje vždy, když je realizován případ užití, ze

kterého tato vazba vychází.

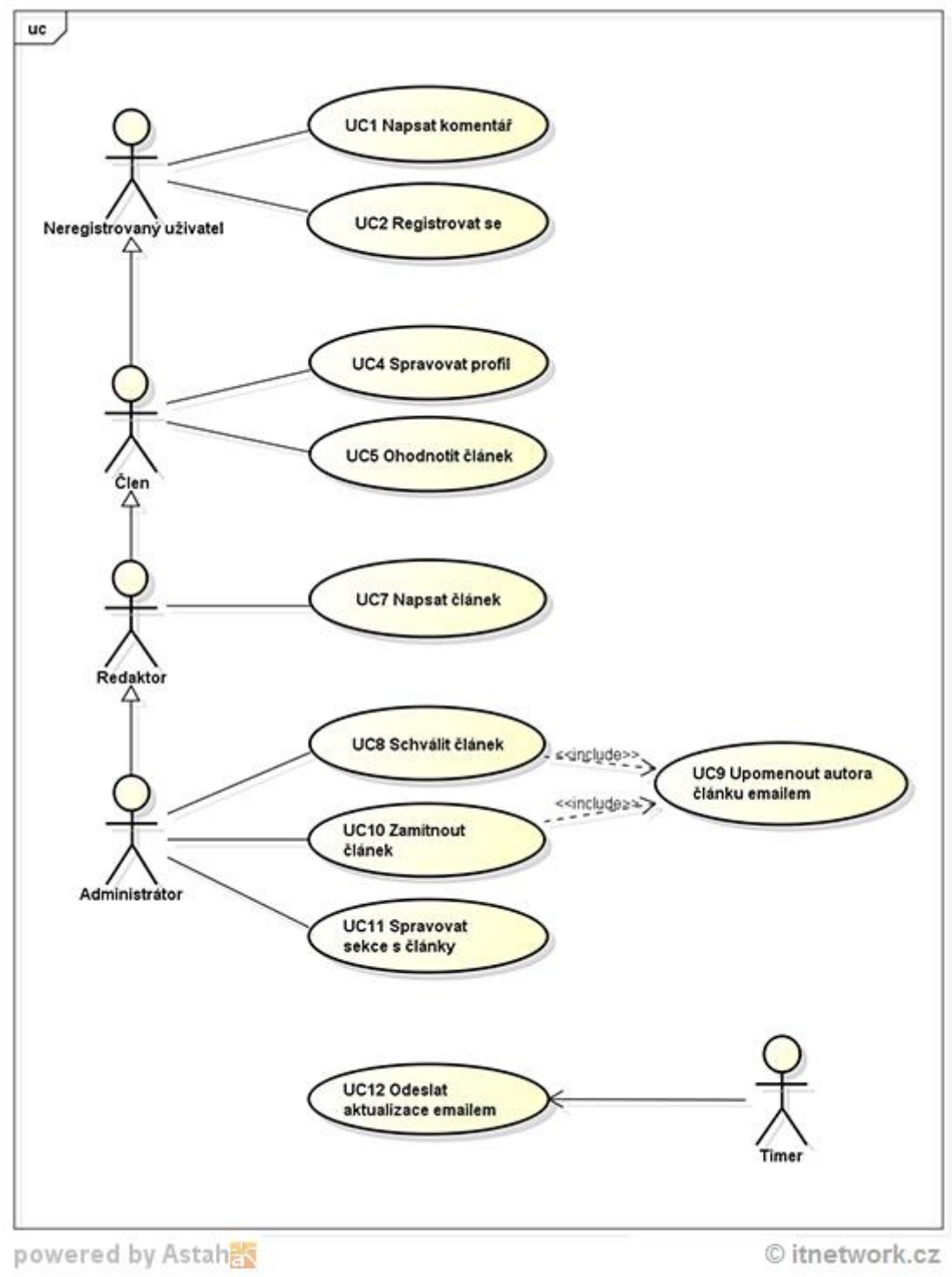


Figure 8 Zdroj: https://www.itnetwork.cz/navrh/uml/uml-use-case-diagram

7.1. Testování a nasazení vytvářeného software

Existuje velké množství testů, přičemž každý test je většinou zaměřen na jinou oblast testování. Jednotlivé testy lze rozdělit na funkční a nefunkční. Funkční testy jsou zaměřeny na přímé požadavky uživatele. Cíle těchto testů můžeme například definovat z pohledu standardu ISO/IEC 12207 nebo z výsledků zjištěných pomocí FURPS+.K příslušnému požadavku následně vytvoříme test.

Nefunkční testy netestují přímo funkčnost systému, testují jeho vlastnosti, které uživatel běžně automaticky očekává. Jde například o:

- Bezpečnostní testy - testují bezpečnost systému
- Zátěžové testy testující chování systému při zátěži.
- Testy použitelnosti – budou použitelné za různých omezujících podmínek. Třeba rychlost připojení k internetu
- Testy dokumentace – u uživatelské dokumentace testujeme srozumitelnost a konzistetntnost. U vývojářské pak množství pokrytí

Softwarové systémy se také testují z pohledu verifikace a validace.

- Pokud **Verifikujeme,** snažíme se zodpovědět otázku. Zda-li je produkt vytvářen a nebo již vytvořen správně po technické stránce.
- Pokud **Validujeme,** odpovídáme na otázku: Slouží software k tomu, k čemu by měl sloužit?

Například pokud zákazník chce systém e-shop Námi vytvořený systém má mnoho zábavných funkcí bohužel možnost koupit je schována někde hluboko, takže většina zákazníku ji nenajde, tudíž nenakoupí.

Někdy se také mluví o Black box a White box testování.

- U **Black box** testování přestupujeme k vyvíjenému softwarovému systému jako k black boxu. Neřešíme tedy co je uvnitř. Kontrolujeme pouze funkcionalitu z pohledu uživatele. Nejčastěji porovnáváme, zda na zadané vstupy vychází odpovídající výstupy. Snažíme se při tom najít i nezvyklé, či neočekávané vstupy, na které by systém nemusel reagovat korektně
- U **White box** testování známe strukturu programu. Ověřujeme zda, že jednotlivé větve programu fungují bez problémů. Snažíme se otestovat každou část programu.

Pro samotné nasazení softwarového systému u uživatele zmiňuje Vondrák (2002) následující aktivity:

- vytvoření výsledného produktu či jeho verzí
- kompletace softwarového systému
- distribuce softwarového systému
- instalace softwarového systému u uživatele
- poskytnutí asistenční služby uživatelům
- plánování a řízení beta testování
- migrace již existující dat a softwarových produktů

8. Měření kvality softwarových systémů

Pokud na konci procesu vývoje je spokojený uživatel/zákazník, můžeme předpokládat, že jsme vytvořili kvalitní software. Tento ukazatel je ale velmi subjektivní. Navíc každý uživatel/zákazník má jiné a jinak náročné očekávání. Proto je třeba najít nějaké objektivnější kritéria hodnocení kvality a přínosů softwarových systémů.

Tato kritéria se nazývají metrikami. (Je to poněkud nešťastný název, protože v matematice má jiný význam – zobecnění vzdálenosti)

Žáček (2017) definuje metriku takto:

Metriku můžeme definovat následovně: „Metrika je přesně vymezený finanční či nefinanční ukazatel nebo hodnotící kritérium, které je používáno k hodnocení úrovně efektivnosti konkrétní oblasti řízení podnikového výkonu a jeho efektivní podpory prostředky IS/ICT. Skupinu metrik sdružených za určitým cílem (tzn. vztahujících se ke konkrétní oblasti, procesu či projektu) nazýváme „portfolio metrik“

Metriky se většinou rozdělují na tvrdé a měkké.

- Tvrdé metriky – jsou objektivně měřitelné. Mezi příklady může být průměrná doba odezvy, počet chyb za časovou jednotku či délka záruky
- Měkké metriky – Jsou pak více subjektivní, těžko objektivně měřitelné. Mohou se vyhodnocovat například ve stupnicích typicky 0 – 100, v pořadí porovnávaných produktů případně i slovní hodnocení produktu.

Dále pak existují celé systémy pro řízení kvality vývoje softwarových systémů. Zákazník je pak většinou velmi rád, když jeho produkt má certifikát dokládající kvalitu. Mnohdy si za něj i rádi připlatí. Tyto systémy nám mohou pomoci k minimalizaci chyb a zvýšení efektivity vývojového procesu. My zde krátce pohovoříme o některých z nich.

8.1. CMMI

Capability Maturity Model (Integration). Tento standard je rozšířen hojně v USA a Japonsku. Je určen pro vývojové týmy. Je to poměrně podrobný model, díky čemuž může mít návodný charakter ke zlepšení úrovně vývojového týmu. Je to soubor pravidel a doporučení, která by měly vývojové týmy dodržovat pro efektivní vývoj a plánování nových produktů. Zaměřuje se na organizaci, plánování a sledování vývojových procesů. Jeho specifikem je rozdělení vývojových týmů do pěti úrovní schopností a zralostí. Jednotlivé týmy se pak mohou v tomto žebříčku posouvat. Posouzení úrovně schopností posuzuje vyškolený posuzovatel přesně daným postupem.

Stupně zralosti jsou (podle wikipedie)

- Počáteční (Initial): Týmy na této úrovni definované procesy nevykonávají nebo pouze částečně

- Řízená (Managed): Je stanoveno řízení projektů a činnosti jsou plánovány
- Definovaná (Defined): Postupy jsou definovány, dokumentovány a řízeny
- Kvantitativně řízená (Quantitatively Managed): Produkty i procesy jsou řízené kvantitativně
- Optimalizující (Optimizing): Tým soustavně optimalizuje své činnosti

Úrovně schopnosti

- Neúplné (Incomplete): Některé činnosti nejsou vykonávány
- Vykonávané (Performed): Činnosti jsou vykonávány
- Řízené (Managed): Je řízeno, jak jsou činnosti vykonávány
- Definované (Defined): Je definováno, jako jsou činnosti vykonávány

8.2. ISO 9000:2001

Je psán obecněji než CMMI. Definuje systém managementu kvality. Tato norma umožňuje prokázat daným organizacím schopnost výroby či distribuci produktů. Je tedy psána velmi obecně. Proces vývoje je dosti specifický. Z tohoto důvodu vznikla například Anglicko – Švédská interpretace TickIT a nebo ISO/IEC 90003

8.3. Six Sigma

Je souhrn postupů řízení. Jejíž cílem je pochopení a následné neustálé průběžné jejich zlepšování. Poznání zákazníka a jeho očekávání. Pomocí porozumění potřeb zákazníků, analýzy procesů, standardizace metod měření a jejich analýzy pomocí statistických metod.

Mezi základní filozofické předpoklady této metodologie patří:

- Neustálé úsilí o dosažení stabilních a předvídatelných výsledků procesu má zásadní význam pro obchodní úspěch.
- Výrobní a obchodní procesy mají vlastnosti, které lze definovat, měřit, analyzovat, zdokonalovat a kontrolovat.
- Dosažení trvalého zlepšování kvality vyžaduje angažovanost celé organizace, od nejvyšší úrovně řízení.

Mezi základní techniky patří například cyklus DMAIC – určeného k pochopení a řízení procesů.

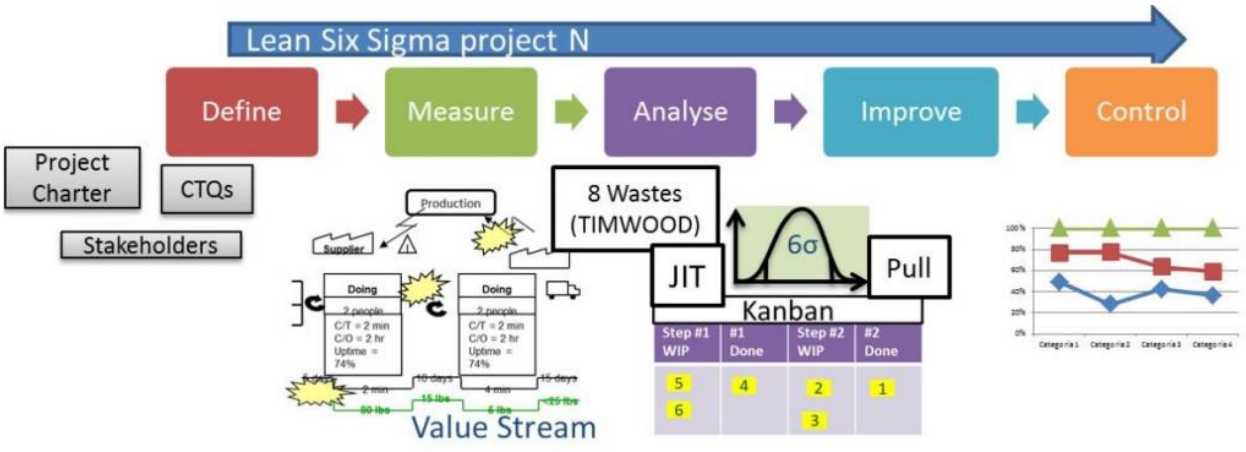


Figure 9 Zdroj: http://www1.osu.cz/~zacek/infs2/02.pdf

DMAIC obsahuje celkem 5 fází.

- **Define (definovat)** – popsání reálného procesu a ideálního procesu. Navrhuje se plán cesty od reálného k ideálnímu procesu
- **Measure (měřit)** – měření současného procesu. (Třeba průměrná doba čekání), tak abychom do rukou dostali „tvrdá data" snažící se popsat „objektivní realitu".
- **Analyze (analyzovat)** – Výsledky měření jsou statisticky analyzovány. Abychom mohli vytvořit fungující model konkrétního projektu. Případně zjistit, které části procesu je vhodné zlepšit.
- **Improve (zlepšovat)** – Realizace myšlenek vzniklých při předchozí fázi. Běžné cíle jsou zvýšení spokojenosti zákazníka, zvýšení efektivity, snížení nákladů.
- **Control (řídit)** – podařilo-li se změnu úspěšně zavést. Je třeba ji standardizovat. A kontrolovat, že přinesla nějaké zlepšení

Původně byla vytvořena ve společnosti Motorola. V současnosti je využívána například ve firmách Honeywell, HP, Texas Instruments, NASA a mnoha dalších. Je založena na aplikovaných statistických postupech, doplněných o kvalitativní nástroje.

9. Údržba a provoz softwarových systémů

Do této chvíle jsme se zabývali vývoje software. Vývojem ale životní cyklus software teprve začíná. Velmi důležitou částí je jeho samotný provoz. Pokud bychom vytvořili báječný software, který po měsíci provozu začne pravidelně kolabovat kvůli jeho špatné údržbě, zákazník by jistě měl pocit, že vývoj software nebyl dobře odveden. Je tedy důležité se náležitě věnovat také údržbě a provozu software.

Způsobům údržby a provozu obecně informačních technologií se věnuje ITIL. Je to souhrn konceptů a postupů, které umožňují obecně lepší využití IT služeb. IT službou rozumíme souhrn IT systémů, jejichž cílem je podpora podnikových procesů (tedy aby podnik dělal to co má).

ITIL je fyzicky sada knižních publikací, které obsahují souhrn nejlepších zkušeností z oboru řízení služeb informačních technologií. Je tedy frameworkem sloužícím ke správě IT služeb, který nám říká, kdy a co máme dělat. Myšlenka ITIL je celkem jednoduchá, proč navrhovat a vymýšlet celý proces znovu od začátku, když už ho spousta jiných firem má zavedený a průběžně ho i vylepšuje.

ITIL je zástupce proaktivního přístupu. Který se snaží problémy řešit dopředu, ne až zpětně. Pokud již nějaké problémy nastanou, snaží se je vyhledávat aktivní detekcí. K čemuž se snaží nastavit odpovídající procesy. Celý proces je běžně řízen, monitorován, měřen, vyhodnocován a neustále vylepšován. Všechny tyto procesy by měly být navrženy za účelem přidané hodnoty zákazníkovy. V některých případech může být problém v terminologiích, kdy rozdílné společnosti používají rozdílné názvy. ITIL se proto snaží sjednotit terminologii. Tyto postupy jsou navrženy jako platformě neutrální.

Pravděpodobně hlavním kladem těchto přístupů je zefektivnění. Díky kterému například daří zkracovat výpadky IT systémů.

Z tohoto důvodu byly zavedeny dva základní procesy Service Support a Service delivery. V rámci těchto procesů je zaveden Service desk – zákazníkovo kontaktní místo. Toto místo shromažďuje požadavky zákazníků či uživatelů. Má evidenci IT procesů. Díky čemuž může na požadavky reagovat a řešit je. Plní roli základní podpory.

V případě problémů nastupuje proces Incident management, který se snaží minimalizovat pro zákazníka nepříjemné důsledky problémů IT systémů. Jedno z nejdůležitějších kritérií je zde rychlost vyřešení problému.

Incident management má dále podporu od **Problem Management** který zpravuje evidenci řešení problémů. Zároveň analyzuje nastalé problémy a jejich trendy, a pokud je to vhodné navrhuje strukturární změny IT systémů.

Jednotlivé změny eviduje **Change managememement**. **Release management** pak plánuje a řídí jednotlivé změny IT služeb (release)

ITIL verze 3 se skládá z 5ti plus jedna úvodní kniha (Žáček, 2017) je charakterizuje takto:

- Service Strategy – zabývá se sladěním byznysu a IT, strategií správy IT služeb,

plánováním.

- Service Design – řešení IT služeb, návrh procesů (tvorba a údržba IT architektury, postupů).
- Service Transition – předání IT služby do byznys prostředí.
- Service Operation – doručení a řídící aktivity procesu, správa aplikací, změn, provozu, metrik.
- Continual Service Improvement – hnací body zlepšení IT služeb, oprávněnost vylepšení, metody, praktiky, metriky.

10. Literatura

Eysenck, M. W., & Keane, M. T. (2008). Kognitivní psychologie. Praha: Academia.

Nolen-Hoeksema, S. (2012). Psychologie Atkinsonové a Hilgarda (Vyd. 3., přeprac.). Praha: Portál.

Sklenář, V. (2007). SOFTWAROVÉ INŽENÝRSTVÍ [Online]. Retrieved June 29, 2018, from https://phoenix.inf.upol.cz/esf/ucebni/syspro.pdf

Softwarové inženýrství [Online]. (2001-). In Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation. Retrieved from https://cs.wikipedia.org/wiki/Softwarov%C3%A9_in%C5%BEen%C3%BDrstv%C3%AD#Softwarov%C3%A1_krize

Plháková, A. (2004). Učebnice obecné psychologie. Praha: Academia.

Rational Unified Process [Online]. (2001-). In Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Rational_Unified_Process

Rychta, A. (2011). Softwarové inženýrství [Online]. Retrieved June 29, 2018, from http://www.ksi.mff.cuni.cz/~richta/NSWI026/NSWI026-1-Uvod.pdf

Říčan, J. (2016). Používané metakognitivní strategie žáků pátých tříd ve specifické doméně čtení (Disertační práce). Praha.

US Department of Justice (2003). INFORMATION RESOURCES MANAGEMENT Chapter 1. Introduction.

Vondrák, I. (2002). Úvod do softwarového inženýrství [Online]. Retrieved June 29, 2018, from http://vondrak.cs.vsb.cz/download/Uvod_do_softwaroveho_inzenyrstvi.pdf

WHITE, Stephen A. Business Process Modeling Notation [online]. [cit. 2018-06-26]. Dostupné z: https://is.muni.cz/el/1433/jaro2014/PV165/um/46771256/pr_06_bpmn.pdf

Žáček, J. (2017). SOFTWAROVÉ INŽENÝRSTVÍ [Online]. Retrieved June 29, 2018, from http://www1.osu.cz/~zacek/sweng/skripta_sweng.pdf