Západočeská univerzita
FAKULTA APLIKOVANÝCH VĚD



Okruhy otázek ke státní závěrečné zkoušce z předmětu
Návrh informačních systémů (NIS)

Strukturální analýza informačních systémů (SAI)
Softwarové inženýrství (ASWI)
Databázové systémy (DB2)
Podnikové informační systémy (PIS)

Studijní program:	3902	Inženýrská informatika
Obor:	2612T025	Informatika a výpočetní technika – Softwarové inženýrství
	3902T031	Softwarové inženýrství

Akademický rok: 2005/2006

Obsah:

1 Základní modely životního cyklu software 4

1.1 VODOPÁDOVÝ MODEL 4
1.2 PRŮZKUMNÝ VÝVOJ (EXPLORATORY DEVELOPMENT) 5
1.3 PROTOTYP (THROW-AWAY PROTOTYPING) 6
1.4 SPIRÁLA (BOEHM, 1998) 6

2 Vývoj software pomocí iterativního a inkrementálního přístupu, agilní metodiky 8

2.1 EXTRÉMNÍ PROGRAMOVÁNÍ (XP) 9
2.2 SCRUM 9

3 Specifikace požadavků na software 11

3.1 ROZDĚLENÍ POŽADAVKŮ 11
3.2 DOKUMENT SPECIFIKACE POŽADAVKŮ (DSP) 11
3.3 METODY SHROMAŽĎOVÁNÍ POŽADAVKŮ 12
3.4 KONTROLA POŽADAVKŮ 12
3.5 MANAGEMENT POŽADAVKŮ 13
3.6 DOMÉNOVÝ MODEL 13

4 Objektová analýza a návrh systému na příkladu metodiky Rational Unified Process 14

4.1 ANALÝZA 14
4.2 NÁVRH 15

5 Postup vytváření objektového modelu a architektury, použití UML diagramů a CASE systémů 16

5.1 UML DIAGRAMY PRO VYTVÁŘENÍ OO MODELU 16
5.1.1 Diagram tříd (Class diagram) *16*
5.1.2 Diagram případů užití (Use case diagram) *17*
5.1.3 Diagram sekvencí, Sekvenční diagram (Sequence diagram) *18*
5.1.4 Diagram spolupráce (collaboration diagram) *18*
5.1.5 Stavový diagram (State transition diagram) *18*
5.2 CASE SYSTÉMY (COMPUTER AIDED SOFTWARE ENGINEERING) 19

6 Návrh a implementace s použitím návrhových vzorů a komponent 20

6.1 ZÁKLADNÍ PRVKY NÁVRHOVÝCH VZORŮ 20
6.2 ZÁKLADNÍ ROZDĚLENÍ 21
6.3 SINGLETON (JEDINÁČEK) 21
6.4 FACTORY METHOD (TOVÁRNÍ METODA) 21
6.5 COMPOSITE PATTERN 22
6.6 OBSERVER 22

7 Konfigurační management, systémy pro správu verzí, zpracování požadavků na změny a údržba software 23

7.1 KONFIGURAČNÍ MANAGEMENT 23
7.2 SCM A SPRÁVA VERZÍ 23
7.3 PROSTŘEDÍ PRO VERZOVÁNÍ: ÚLOŽIŠTĚ (=REPOSITORY) 23
7.4 NÁSTROJE PRO VERZOVÁNÍ 24
7.4.1 Revision Control System (rcs) *24*
7.4.2 Realizace variant (cpp) *24*
7.4.3 Concurrent Versioning System (cvs) *24*
7.4.4 Subversion (svn) *24*
7.5 SPRÁVA ZMĚN 25
7.5.1 Change Control Board (CCB) *25*
7.5.2 Systémy pro zprávu změn *25*

8 Způsoby prevence a detekce chyb v software – testování, oponentury 26

8.1 PREVENTIVNÍ TECHNIKY 26
8.2 TECHNICKÁ OPONENTURA 26
8.3 TECHNICKÁ OPONENTURA – POSTUP 26
8.4 WHITEBOX 26
8.4.1 Pokrytí kódu *26*
8.4.2 Jednotkové testy *27*
8.4.3 Integrační testy *27*

8.5 BLACKBOX 27
8.5.1 Smoke test 27
8.5.2 Zátěžové testy 28
8.5.3 Testování systému 28
8.5.4 Testování hraničních případů 28
8.5.5 Návrh testovacích případů (data) 28
8.5.6 Shrnutí 28

9 Systémy zabezpečení kvality softwarového procesu, přehled metodiky CMM a norem ISO 9000 29

- SLOŽKY SYSTÉMŮ ŘÍZENÍ KVALITY 29
- ZAVÁDĚNÍ SYSTÉMU ŘÍZENÍ JAKOSTI 29
- ZÁKLADNÍ NORMY 29
  - *CMM 30*
  - *CMMI 32*
  - *ISO 9000-2000 ..... 32*

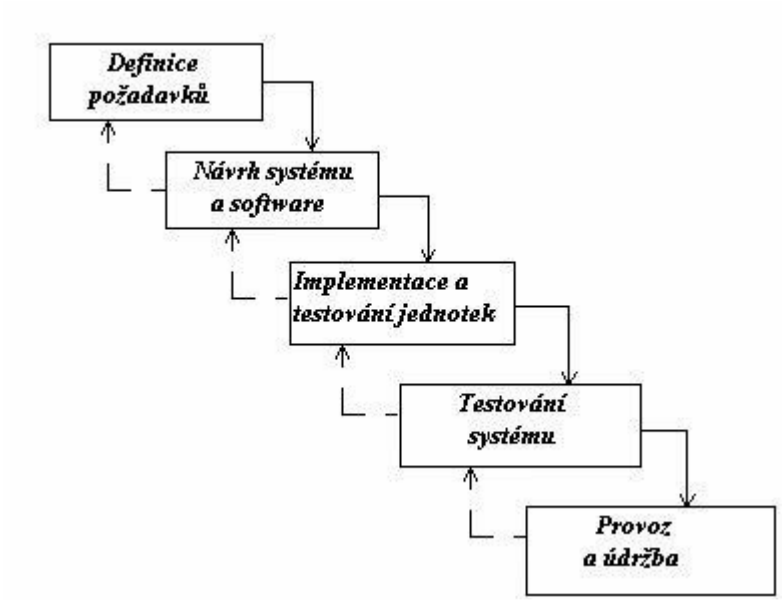
1 Základní modely životního cyklu software

1.1 Vodopádový model

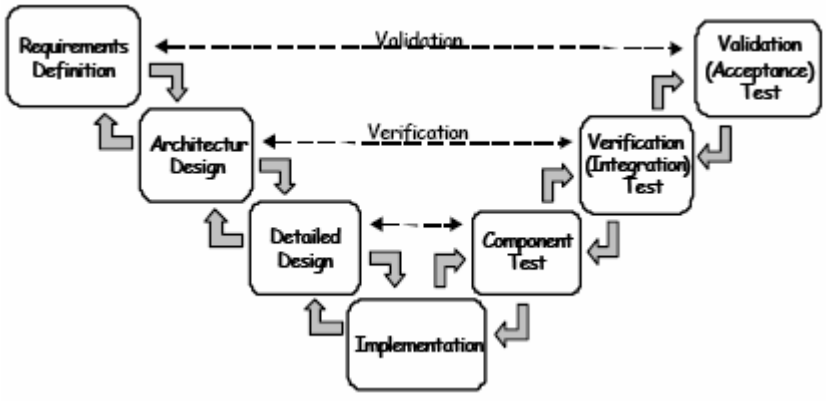
- Jeden z nejstarších modelů (dokonce asi první ☺)
- Hlavní myšlenky:
  - před návrhem řešení se musí kompletně rozumět zadání
  - výstup jedné aktivity je vstupem následující (tj.)
- Základní aktivity jsou
  - *Analýza a definice požadavků (Requirements definition)*
    - Konzultací s uživateli sstému se zjistí cíle, požadované služby a omezení kladená na systém
    - To se podrobně definuje v dokumentu, který slouží jako specifikace systému
  - *Design systému a sw (System and software design)*
    - Provede se návrh systému, rozdělí se celkové požadavky na požadavky na HW a požadavky na SW, určí se celková architektura systému
  - *Implementace a testování modulů (Implementation and unit testing)*
    - Je realizován design jako množina modulů
    - Testování na úrovni modulů, ivěření, že každý modul odpovídá specifikaci modulu
  - *Integrace a testování systému (Integration and system testing)*
    - Jednotlivé moduly jsou sestaveny do výsledného systému
    - Úplný systém je otestován na shodu se specifikací
    - Po otestování je systém předán zákazníkovi
  - *Provoz a údržba (Maintenance)*
    - Nejdelší fáze životního cyklu – systém se prakticky používá
    - Údržba = oprava chyb programu a designu, které nebyly odhaleny v předchozích fázích + rozšiřování systému podle nových požadavků + zlepšování implementace
- Výhody
  - snadné k pochopení
  - dobrá možnost řízení a sledování postupu řešení
- Nevýhody
  - vyžaduje mít na počátku přesně a úplně definované požadavky (uživatel často nedokáže stanovit předem)
  - provozuschopnost verze vidí zákazník až v závěrečných fázích řešení, případné závažné nedostatky jsou odhaleny velmi pozdě.
  - během vývoje se mohou měnit požadavky a výsledkem je, že dodaný produkt není to, co zákazník chtěl

V-model

- jednou z nejznámějších variant vodopádového modelu
- důraz na testování sw, každá aktivita musí být verifikována



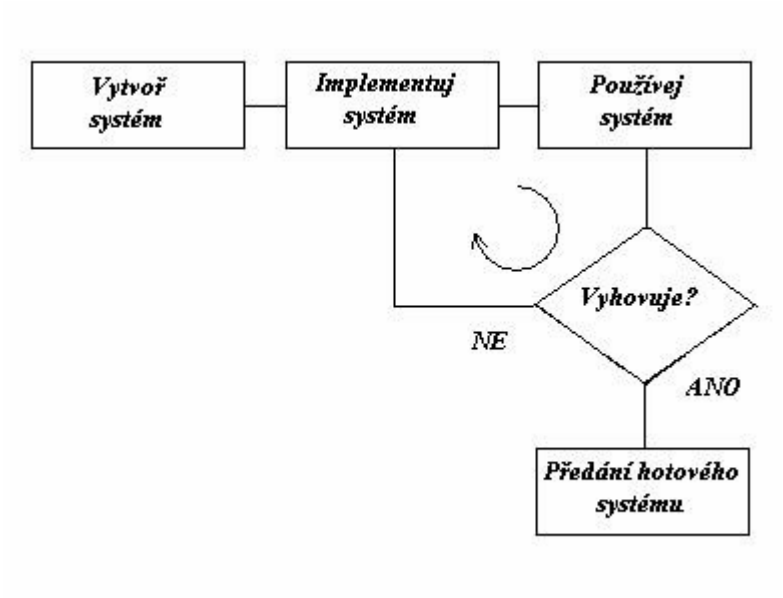
Obrázek 1: Vodopádový model



Obrázek 2: V-model

1.2 Průzkumný vývoj (exploratory development)

- vytvoří se počáteční implementace, ta se vystaví komentářům uživatele a postupně se vylepšuje přes meziverze, dokud se nedostaneme k adekvátnímu výsledku
- cílem je spolupracovat se zákazníkem na zjištění jeho požadavků, abychom mu nakonec dodali požadovaný systém
- vývoj obvykle začíná dobře srozumitelnými částmi systému
- systém se vyvíjí přidáváním nových vlastností navrhovaných zákazníkem
- VÝHODY
    - systém je velice dobře přizpůsobitelný i dodatečným požadavkům zákazníka.
- NEVÝHODY
    - manažersky velmi náročné – etapy lze stěží plánovat časově, finančně i personálně.
    - dokumentace – pokud nevzniká průběžně, je odrazem hotového díla.
    - není jasné, které požadavky byly stanoveny při zadání a které jsou nyní nové.



Obrázek 3: Model výzkumník

1.3 Prototyp (throw-away prototyping)

- tvorba prototypu, který bude zahozen
- ověřuje funkcionalitu výsledného systému
- zaměřuje se na ty části požadavků zákazníka, kterým se moc nerozumí
- cílem je lepší pochopení zákazníkových požadavků a v důsledku vytvoření lepší definice požadavků na systém
- VÝHODY
    - budoucí uživatelé testují a ověřují prototyp a je upřesňována specifikace požadavků.
- NEVÝHODY
    - náročný na vedení v okamžiku, kdy existuje třeba i několik nezávislých vývojových skupin. Počet pracovníků a způsob vývoje se pak samozřejmě nepříznivě projeví i na nákladech projektu.

Smíšený přístup

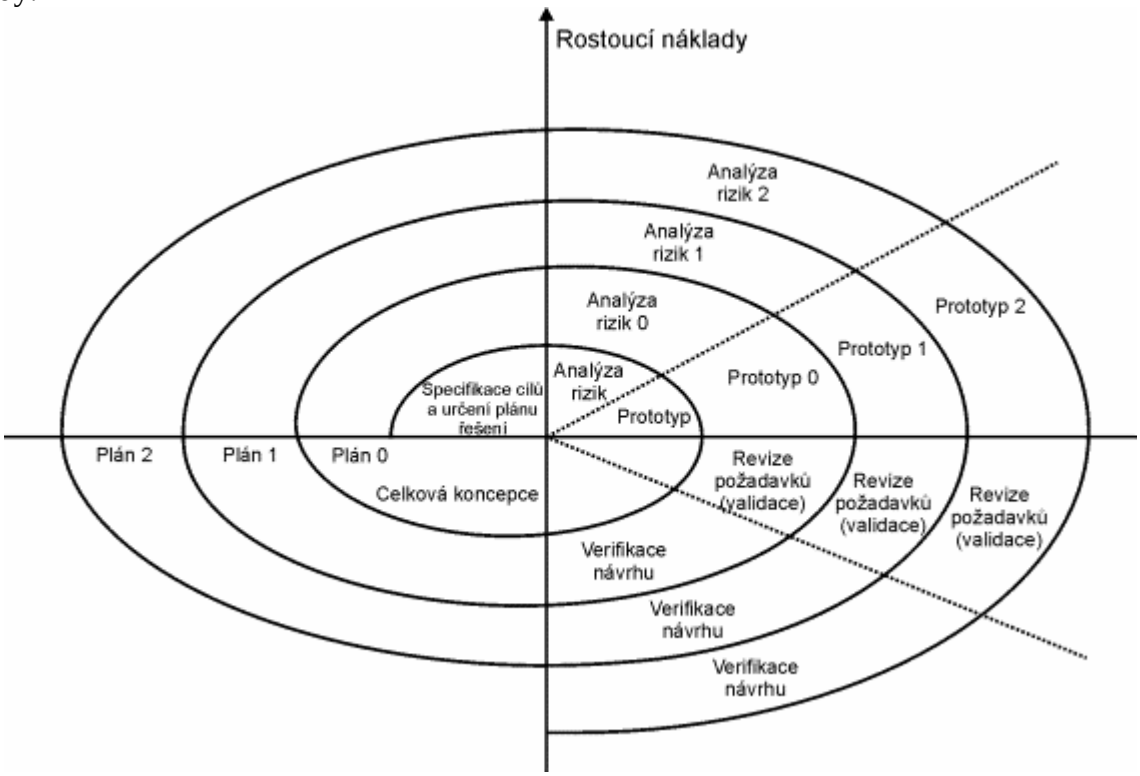
- špatně srozumitelné části – throw-away prototyping → zpřesnění DSP (dokument specifikace požadavků)
- dobře srozumitelné části – vodopádový model
- uživatelské rozhraní - exploratory development

1.4 Spirála (Boehm, 1998)

- iterativní přístup
- základem celého modelu je neustálé opakování vývojových kroků tak, že v každém dalším kroku se na již ověřenou část systému přibalují části na vyšší úrovni.
- každá otočka spirály je rozdělena do 4 sektorů
    - specifikace cílů a určení plánu řešení
        - určíme alternativní možnosti realizace
        - identifikujeme omezení daných alternativ (cena, časový plán, přenositelnost)
    - vyhodnocení a snížení rizik
        - pro každé z identifikovaných rizik je provedena detailní analýza a kroky ke snížení rizika

- vývoj a validace
- plánování
    - naplánujeme další fázi projektu (organizace, požadavky na zdroje, rozdělení práce, časový plán, výstupy)
    - každá otočka je zakončena posouzením všech produktů vytvořených v předešlém cyklu

Náklady a čas nutný na realizaci jednotlivých částí projektu, či na řešení celého projektu jsou patrné z modelu, neboť úhlová dimenze udává časovou náročnost a radiální úroveň udává rostoucí náklady.



Obrázek 4: Spirálový model

Výhody

- explicitně uvažuje rizika projektu
- umožňuje konzultovat požadavky zákazníků v jednotlivých krocích a modifikovat systém podle upřesněných požadavků
- první verze systému je možné sledovat a hodnotit při jejich postupném vzniku

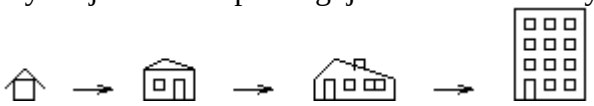
Nevýhody

- řešení systému pomocí tohoto modelu vyžaduje neustálou spolupráci zákazníků, proto není vhodný zejména pro systémy vyvíjené na zakázku bez účasti budoucích uživatelů.
- neumožňuje přesné naplánování termínů, cen a jednotlivých výstupů a tím i jejich plnění.
- je nutné provést bezchybnou analýzu rizik a vybrat aspekty u nichž budeme rizika prověřovat, neboť na této analýze jsou založeny další fáze projektu. Pozdní zjištění komponent s vysokou mírou rizika může mít zásadní vliv na celý projekt.

2 Vývoj software pomocí iterativního a inkrementálního přístupu, agilní metodiky

Iterativní

- hlavní myšlenkou je rozbít velký projekt do řady malých a na ně použít sekvenční model (vodopád).
- výsledkem každého cyklu je větší a lépe fungující část cílového systému.



- modely: Spirála, RUP (Rational Unified Process)

Inkrementální

- způsob dodání, celková funkcionalita je dodávána po částech
- zákazník tak má možnost průběžně sledovat vývoj, může se k němu vyjádřit a oponovat změny
- zákazník má na konci jistotu, že nedostane něco, co neočekával

Agilní metodiky (agile = hbitý, mrštný)



Obrázek 5: tradiční vs. agilní přístup

- nejčastěji používané: XP (extrémní programování), SCRUM
- mnohé další: FDD (Feature Driven Development), ASD (Adaptive sw development), DSDM (Dynamic Solution Delivery Model), Lean Development, …

- mýty sw projektů
  - zákazník ví co chce (přesná specifikace požadavků)
  - dodavatel ví jak na to (postup, náklady, kvalita)

- realita
  - tyto předpoklady jsou ale platné pro sériovou výrobu, což výroba sw není, naopak jde o vývoj nového produktu (ten je jedinečný a bez vzoru a modelu)
  - zákazník neví, co chce, neumí to říct, ale chce to
  - změny jsou spíše pravidlem než vyjímkou
    - změny požadavků – syndrom IKIWISI (I Will Know It When I See It = budu to vědět, až to uvidím). Typicky se mění zhruba 25% specifikovaných požadavků. ...
    - změny prostředí – legislativa, nové technologie, ...
    - změny postupu – fluktuace v týmu, zmeny nástrojů, chybná architektonická rozhodnutí, ...
  - zjednodušené modely nefungují (vodopád)

- charakteristika agilních metodik
  - iterativní vývoj s velmi krátkými iteracemi

- tolerantní ke změnám (přivítání změny – embrace change) – je to výhodou na trhu, usnadní to nasazení v praxi
- lidé jsou prvořadým faktorem (ne proces/technika/prostředek) – důraz na spolupráci a komunikaci (člověk je středem dění) – postaveno na důvěře
- jednoduchost technik (YAGNI - You Aren`t Going to Need It – nebudeš to potřebovat) – pomáhá se zaměřit na primární cíl
- automatizované testování (test-driven, test-first development)
  - pořadí: zadání → test(y) → implementace
  - testy prověří návh před implementací – včasná zpětná vazba
  - testy slouží i jako technická dokumentace
- komunikace
  - schází se kompletní tým (vývojáři, reprezentant zákazníka, QA – oddělení kvality)
  - raději face-to-face než email a podepsané specifikace. Pravidlo „kam můžeš zatelefonovat, tam nepiš, kam můžeš dojít, tam entelefonuj“ → vede k lepšímu vztahu a vyšší zodpovědnosti
- odvaha
  - zahodit kód, na kterém jsem dělal týden, když nefunguje
  - důvěřovat, že jednoduché řešení je lepší
  - kompletně změnit plán v další iteraci
  - požádat mladšího kolegu o pomoc
  - oželet bez zbytečných křečí důležitého člena, který se rozhodne odejít
- cílem není znovupoužitelnost

2.1 Extrémní programování (XP)

Proč extrémní? Protože používá osvědčené a známé postupy vývoje sw, dotahuje však jejich použití do extrémů.

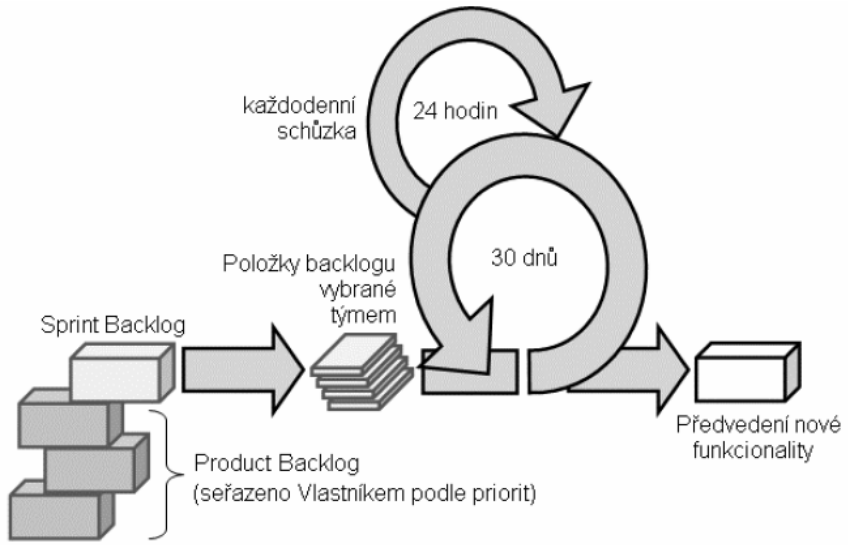
- neustálá revize zdrojového textu programů (protože kontroly nezaujatým čtenářem jsou dobrá věc, budeme kontrolovat kód nepřetržitě) – ***párové programování***
- protože testování je dobrá věc, všichni budou testovat neustále, dokonce i zákazník (testovací kód někdy přesahuje svým rozsahem vlastní výkonný kód) – ***test-first***
- ověřování, zda návrh programu je správný (protože návrh je dobrá věc, uděláme z navrhování software každodenní chléb všech programátorů) – ***refaktoring***
- program se udržuje na co nejmenší úrovni složitosti – vždy se programuje jen to, co je v danou chvíli nezbyté – ***jednoduchost***
- protože krátké iterace jsou dobrá věc, budeme iterace mít opravdu krátké – vteřiny, minuty a hodiny, ne týdny, měsíce a roky – ***krátké iterace***
- dokumentace minimální, jen pokud je opravdu potřebná

Hodí se pro menší projekty a malé týmy vyvýjející sw podle zadání, které je nejasné nebo se rychle mění.

2.2 SCRUM

Proces se skládá ze tří částí:

1. Předehra (Pre-game, Pre-sprint)
   - plánování, seznam užitečných vlastností – features, úkoly, chyby, … (=backlog), definování cíle sprintu
2. Hra, vývoj (Game, Sprint)
   - přírůstková implementace, scrum meetings
3. Dohra (Post-game, Post-sprint)
   - dodávka



Obrázek 6: SCRUM

Každodenní schůzka (SCRUM meetings)

- ve stejný čas na stejném místě
- max 30min (cílem je 15 minut) vestoje
- vede SCRUM master (člen týmu, 50% úvazek na implementaci)
- účastníci všichni členové týmu (vývojáři, testeři, uživatelé, …)
- navštěvují je manažeři, aby věděli o stavu, ale aktivně se neúčastní
- slouží ke zjištění problémů, ale ne k jejich řešení
- každý učastník odpovídá na otazky:
    - co jsi udělal od poslední scrum porady?
    - co budeš dělat do příští scrum porady?
    - jaké překážky Ti stojí v cestě?

3 Specifikace požadavků na software

Co je to požadavek?

- **požadavek** = schopnost nebo vlastnost, kterou má sw mít, aby jej uživatel mohl použít k vyřešení problému nebo dosažení cíle, který vedl k zadání, nebo aby splnil podmínky stanovené smlouvou, standardem nebo jinou specifikací.
- **vlastnosti požadavku:** úplný, bezesporný
- požadavkem není to, co uživatel nepotřebuje

3.1 Rozdělení požadavků

Typy požadavků

- *funkční požadavky (functional requirements)*
  - popisují funkce nebo služby, které jsou od systému očekávány
  - př.: požadavky na univerzitní knihovní systém
    - systém by měl poskytovat uživatelům vhodné prohlížení pro čtení dokumentů v úložišti dokumentů
- *mimofunkční požadavky (non-functional requirements)*
  - netýkají se funkcí systému, ale vlastností jako je spolehlilvost, čas odpovědi, obsazené místo na disku nebo v paměti, aj.
  - často kritičtější než jednotlivé funkční požadavky (např. pokud je řídící systém letadla nespolehlivý, je nepoužitelný)
  - někdy dané vnějšími faktory, tj. legislativní požadavky (př. zákon na ochranu osobních informací, apod.)
  - př. veškerá komunikace mezi uživatelem a systémem by měla být vyjádřitelná ve znakové sadě ISO 8859-2
    - definuje omezení návrhu uživatelského rozhraní, tj. není funkce, ale omezení → mimofunkční požadavek

Další rozdělení vyplývá z rozdílné úrovně popisu – podle čtenáře:

- *uživatelská specifikace požadavků (user requirements specification)*
  - vysokoúrovňový popis funkčních a mimofunkčních požadavků zákazníka
  - musí být srozumitelné pro uživatele, kteří nemají technické znalosti
- *systémová specifikace požadavků (system requirements specification)*
  - podrobnější specifikace uživatelských požadavků pro vývojáře
  - slouží jako výchozí bod pro design systému

3.2 Dokument specifikace požadavků (DSP)

- angl. Software Requirements Specification (SRS)
- konečný výsledek analýzy požadavků
- technický dokument, oficiální vyjádření o tom, co se od vyvíjeného systému očekává (dohoda mezi zákazníkem a dodavatelem, co má zadaný sw dělat a jak to má vypadat)
- základ pro pozdější ověření správnosti - důraz na jednoznačnost, ověřitelnost, reálnost, srozumitelnost, úplnost, přesnost a správnost, modifikovatelnost, konzistence
- měl by specifikovat pouze extérní chování systému, tj. snaha ***vyloučit z DSP design***
- strukturován tak, aby v něm vylo ***snadné provádět změny (modifikovatelnost)***
- měl by specifikovat omezení implementace
- měl by charakterizovat přijatelné odpovědi na nežádoucí události

Různé velké organizace definovaly vlastní standardy pro strukturu DSP (např IEEE/ANSI 830). Ve skutečnosti bude informace v DSP záviset na vyvíjeném produktu a na modelu sw procesu, takže je nutné si obecný model přizpůsobit

Doporučení: pro praxi si navrhnout vlastní formát a používat ho pro všechny DSP – sníží se tím ppst, že se na něco zapomene.

3.3 Metody shromažďování požadavků

Způsoby získávání:

- *interview*
  - předem připravený rozhovor, který vedemoderátor (klade otázky, dává slovo)
  - nedoporučuje se více než 2 hodiny
  - předem si připravit scénář, které okruhy se budou probírat, v jakém pořadí, scénář se snažit nenásilně dodržovat
- *monitorování, komentování postupů*
  - pozorování prací u zákazníka (účast analytiků)
  - analýza existujícího se systému
- *sběr požadavků na základě případů užití (use-case)*
  - srozumitelné pro většinu lidí (jak pro analytiky, tak pro zákazníka)
  - součástí notace UML, podpora v CASE nástrojích

Způsoby vyjádření

- *přirozený jazyk*
  - výhodou je srozumitelnost pro uživatele
  - nevýhodou – spoléhá se na to, že autoři používají stejná slova pro stejný koncept (stejná věc se dá říci mnoha různými způsoby). Obtížná modularizace - kterých všech dalších požadavků se změna dotkne.
- *formuláře*
  - pro vyjádření požadavku se nadefinuje jeden nebo více typů formulářů
  - měl by obsahovat:
    - popis specifikované funkce nebo entity
    - popis vstupů, odkud se berou
    - popis výstupů, kam putují
    - jaké další entity specifikovaná funkce nebo entita používá
    - případné pre/post conditions (co platí při vstupu do funkce a co při výstupu z ní)
    - pokud vznikají postranní efekty, pak jejich popis
- *pseudokódy*
  - v přirozeném jazyce težko vyjádřitelné vnořené podmínky nebo smyčky
  - jazyk s abstraktními konstrukcemi, které právě potřebujeme
  - vnoření konstrukcí je vyjádřeno odsazením
  - vyhýbáme se syntaktickým konstrukcím cílového programovacího jazyka (popisujeme požadovaný záměr, nikoli jak to bude v cílovém jazyce)
  - na druhou stranu musí umožňovat téměř automatickou konverzi do kódu

formuláře vs. pseudokód

- formuláře – celková specifikace systému
- pseudokód – řídící sekvence, rozhraní

3.4 Kontrola požadavků

- musíme zjistit, zda jsou požadavky úplné, konzistentní a zda odpovídají tomu, co zadavatel chce
- vstupem je úplný DSP

- metody:
    - *přezkoumání (reviews)* – požadavky jsou systematicky kontrolovány týmem, manuální proces
    - *prototypování* – zákazníkovi předvedeme spustitelný model systému
    - *generování testovacích případů* – vytvoříme testy požadavků, pokud je obtížné vytvořit test, bude požadavek obtížně implementovatelný
    - *automatická analýza konzistence* – pokud byly požadavky specifikovány jako model ve formální nebo strukturované notaci

3.5 Management požadavků

- požadavky na systém se stále mění
- měl by začít plánováním, ve kterém se rozhodne:
    - způsob identifikace požadavků – každý požadavek by měl mít jednoznačné ID
    - proces změny požadavků – definujeme proces, abychom se ke změnám požadavků chovali konzistentním způsobem
    - sledovatelnost
        - zdroj požadavku – kdo požadavek navrhnul, důvod; abychom se mohli zdroje zeptat na podrobnosti
        - vztahy mezi požadavky – kolik požadavků se změna dotkne
    - nástroje – co se použije pro uchování informací o požadavcích (malé projekty – obvyklé prostředky(textové nástroje, EXCEL, databáze, aj), velké projekty – CASE nástroje)

3.6 Doménový model

- jen základní obrysy, terminologie uživatele a pojmy → názvy tříd, vztahy mezi třídy (asociace, kardinality), nezavislé na implementaci
- sledujeme podstatná jména v definici problému, věci a místa v aplikační doméně, pro každé vytvoříme předběžnou třídu (třída je zatím popsána pouze jménem); slovesa zaznamenáváme, aby časem mohla být operacemi
- eliminujeme nepotřebné a chybné třídy třídy (třídy, které nejsou pro aplikaci relevantní, vágně definované třídy, aj.)

4 Objektová analýza a návrh systému na příkladu metodiky Rational Unified Process

Mezi 1989 a 1994 navrženo cca 40 nových OO metodik (např.: OMT, OOSE, OOD, aj.). metody měli tolik společného, že se tři klíčoví autroři (Booch, Jacobson a Rumbaugh) rozhodli své návrhy integrovat do jedné metodiky → RUP (Rational Unified Process).

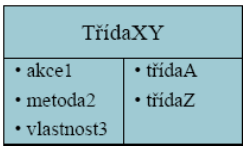
Přechod mezi analýzou a designem poznáme podle toho, že se model začne týkat implementačního prostředí (vlastností cílového jazyka apod.).

4.1 Analýza

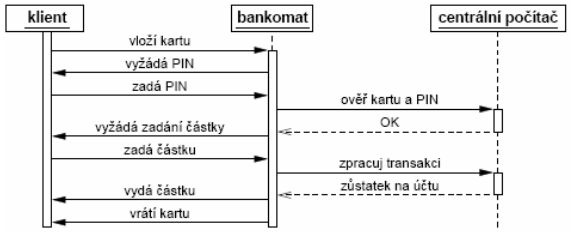
- co se chce (objektový model reálného/projektovaného světa), předpokládáme ideální technologii.
- vstupem je popis případů použití a doménový model
- případy použití zjemníme směrem k implementaci pomocí dynamických modelů (nejčastěji sekvenčních diagramů (synchronní chování) nebo pomocí stavových diagramů a diagramů spolupráce (asynchroní chování))
- přidá zodpovědnosti, detaily, vlastnosti a chování do doménového modelu
- výsledkem je analytický model = abstrakce popisující co má systém dělat, nikoli jak

Nástroje

- **CRC (Class-Responsibility-Collaborators) karty**
  - mechanická technika (papírové karty, tužka, guma, stůl)
  - pro třídy vytvoříme kartičky, nahoru napíšeme jméno třídy, vlevo zodpovědnost třídy, vpravo spolupracující třídy
  - nevýhoda – vazby mezi třídami nejsou znázorněny graficky



- **Diagram sekvencí (Sequence diagram)**
  - popis výměny zpráv
  - objekty reprezentují třídy, zprávy (výstižné fráze pro zachycení zodpovědnosti nebo jména metod při dostatku informací)
  - detaily (tj. třídy, metody, větvení, cykly) řešit, až když je jasno



- **Diagram tříd (Class diagram)**
  - na počátku je třída popsána jen názvem, zkontrolujeme, zda již není název použit, ponecháme ten název, který daný objekt popisuje nejlépe
  - ke třídě najdeme logické atributy = informace, která se nebude používat samostatně, ale je silně vázána s objektem (př. jméno, věk, adresa budou atributy nějaké osoby)
  - třídy lze rozdělit podle následujících streotypů:
    - *hraniční třídy (boundary class)* – všechno, s čím aktéři přímo komuinikují, můžeme je zjistit např z popisu užití
    - *entitní třídy (entity class)* – informace, kterou szstém udržuje delší dobu, často odpovídají objektům reálného světa (př. klient, účet)
    - *řídící třídy (control class)* – chování v systému, třídy obsahující řídící logiku, používající nebo nastavující obsah entitních tříd

4.2 Návrh

- jak to realizovat (třídy a mechanismy pro efektivní realizaci)
- adaptace (=přizpůsobení) analytického modelu pro realizaci ve skutečném světě
- implementace by měla být již přímočará a téměř mechanická
- vstupem jsou analytické třídy představující role, které mohou být pokryty jednou nebo více třídami návrhu (analytická třída se může stát v návrhu jednou třídou, částí třídy, agregovanou třídou, …)
- ve výsledku známe (skoro) všechny detaily implementace
    - vazba na konkrétní technologie , okolní prostředí
    - způsob uložení perzistentních dat
    - struktura aplikace
    - používané konvence a návrhové vzory
- architektura systému
    - logické členění do balíků
        - balík – skupina souvisejících tříd, tvořící organizační celek, mapování do jazyka (balík vytváří jmenný prostor), hierarchické vnořování
        - třídy v balíku funkčně příbuzné
        - vhodné protože bude přehled o systému a snadné rozdělení implementace mezi členy týmu
    - funkční členění do subsystémů
        - subsystém = skupina souvisejících balíků a/nebo tříd tvořící funkční celek
        - je to vhodné, protože monolitická aplikace není praktická
        - jak najít subsystémy?
            - buď je to dopředu zřejmé (jednoduché, architektonické styly)
            - na základě objektového modelu (nutno vidět všechny třídy a vazby, pak shluk těsně vázaných tříd je kandidátem)
            - na základě případů užití
    - základní vzory architektury
        - klient-server architektura
        - 3 vrstvé – oddělení prezentace, business logiky a datové části
- Návrh řešení standardních situací
    - Základní prostředky – návrhářská/programátorská zkušenost, návrhové vzory (Singleton, Composite, Factory, Fasáda), jazykové idiomy, jmenné konvence

5 Postup vytváření objektového modelu a architektury, použití UML diagramů a CASE systémů

Objektový model vytvářím z případů užití a analýzy. TODO je to výsledkem návrhu ???

- *hraniční třídy* – slouží k prezentaci, čtení/předávání dat k/od interních objektů
  - objekty zřejmé z popisu rozhraní případů užití (PU)
  - alespoň jeden hraniční objekt pro každého aktéra
- *datové (entitní) třídy* – zapouzdřují informace uchovávané delší dobu
  - jsou zřejmé z výsledků analýzy (často doménové objekty, data vstupující do systému)
  - je dobré separovat společné vlastnosti → dědičnost
- *řídící třídy* – řízení aplikace, funkčnost, která se mže měnit nezávisle na datech
  - typicky v rozborech případů užití (PU), pro začátek jeden případ užití → jeden řídící objekt
  - zajištění komunikace mezi objekty
- *systémové třídy* – využití již hotového (znalost knihoven – soubory, aj.)

Konsolidace objektového modelu

- konsolidace = upevnění, ustálení, úprava
- cílem návrhu je kompletní, konzistetní, kvalitní struktura každé třídy, jenže výsledkem z analýzy scénářů je nekonzistence, duplikáty, rozpory v sémantice (příčinou jsou různí autoři, ruzné scénáře)
- snaha o minimalitu, reuse, viditelnost vlastností (public, private)
- vyhledání opakujících se prvků →rodičovské třídy (dědičnost)
- vyjasnění vztahů (asociace → agregace)

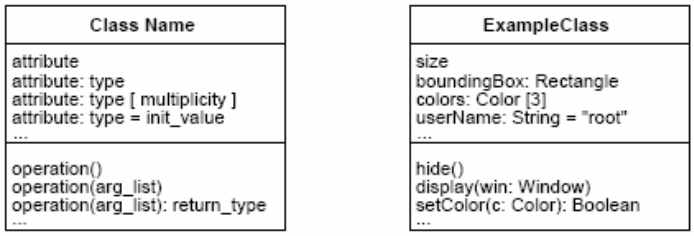
5.1 UML diagramy pro vytváření OO modelu

5.1.1 Diagram tříd (Class diagram)

- ukazuje třídy a vztahy mezi nimi (zobecnění, asociace, agregace, …)

Třídy

- obdélník rozdělen na tři části
  1. název třídy
  2. atributy
  3. metody



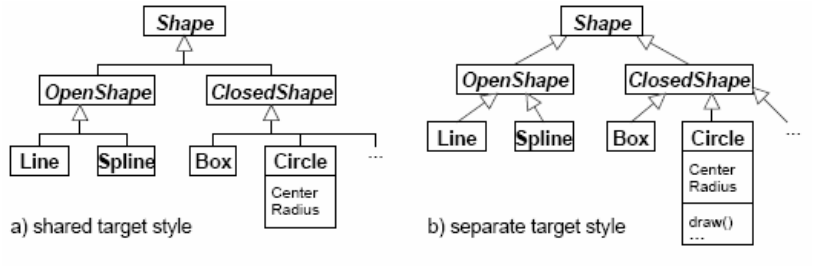
Rozhraní

- stereotyp «*interface*»
- jinak se kreslí stejně jako třída (část s atributy se vynechává)
- třídy, které implementují rozhraní se spojují s rozhraním přerušovanou šipkou
- rozhraní se může kreslit také jako kroužek spojený s plnou čarou s třídou, která ho implementuje, pod kroužkem název rozhraní



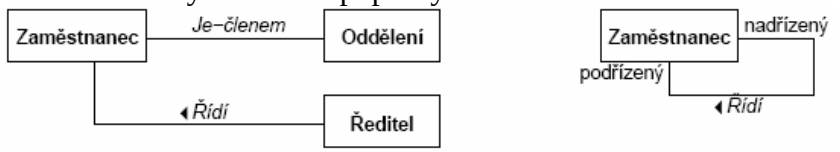
Zobecnění

- vyznačení dědičnosti

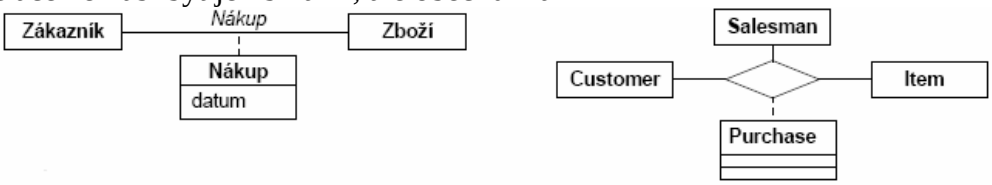


Asociace

- objekty, které jsou instancemi jedné třídy, mohou mít vztah s objekty jiné nebo stejné třídy (př. objekt třídy zaměstnanec bude mít asociaci k objektům třídy oddělení)
- v UML se znázorňuje plnou čarou, u čáry volitelně název vztahu, u názvu volitelně malý černý rojúhelníček – kterým směrem se má asociace číst
- konce asociace mohou být volitelně popsány rolemi ve vztahu



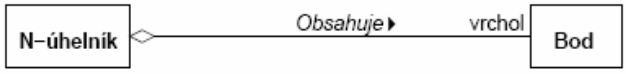
- asociační třída – pokud má asociace vlastnosti jako atributy, operace aj.
- asociace nemusí být jen binární, ale obecně n-ární



- asociace je velmi obecný typ vztahu

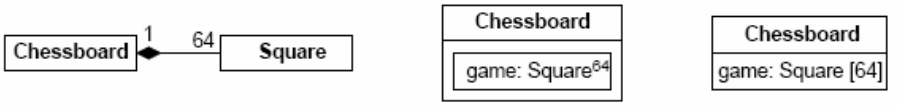
Agregace

- jednou z nejčastějších binárních asociací je agregace (znázorňuje se prázdným kosočtvercem na straně agregátu)
- znamená to, že objekt je vytvořen z dalších objektů = je agregátorem množiny objektů (př. stádo je agregátem ovcí, les je agregátem stromů, rodina je agregátem objektu typu muž a objektu typu žena a množiny objektů typu dítě, …)



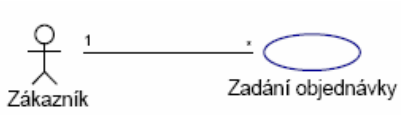
Kompozice

- je silná asociace – součást nemůže existovat samostatně (př. políčko nemůže existovat bez šachovnice).
- Znázorňuje se plným kosočtvercem nebo grafickým vnořením.



5.1.2 Diagram případů užití (Use case diagram)

- popisuje, co systém dělá, nikoli, jak to dělá
- obsahuje
  - *aktéry* (znázorněny jako panáčci) – cokoliv co potřebuje komunikovat se systémem; představuje to roli
  - *případy užití* (znázorněny elipsou) – představuje dialog nebo transakci vykonávanou systémem
    - spojeny se scénáři – posloupnost kroků
  - *komunikace* (čáry mezi akterý a PU) – spojuje aktéry s PU



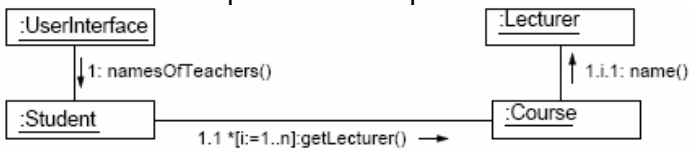
5.1.3 Diagram sekvencí, Sekvenční diagram (Sequence diagram)

- popisují stejnou informaci jako digramy spolupráce, ale soustředí se na časové závislosti
- znázorňují aktéry, objekty v systému, se kterými interagují a posloupnost výměny zpráv
- čas plyne shora dolů – pro každý objekt svislá přerušovaná čára (lifeline)
- horizontální uspořádání není podstatné a má být zvoleno s ohledem na přehlednost
- šipky podle typu komunikace
    - volání procedury nebo jiný vnořený tok řízení (tj. nější sekvence může pokračovat, až po dokončení vnitřní sekvence) – plná šipka, viz a)
    - asynchroní komunikace – neplná šipka, viz b)
    - návrat z procedury – přerušovaná šipka, viz c)



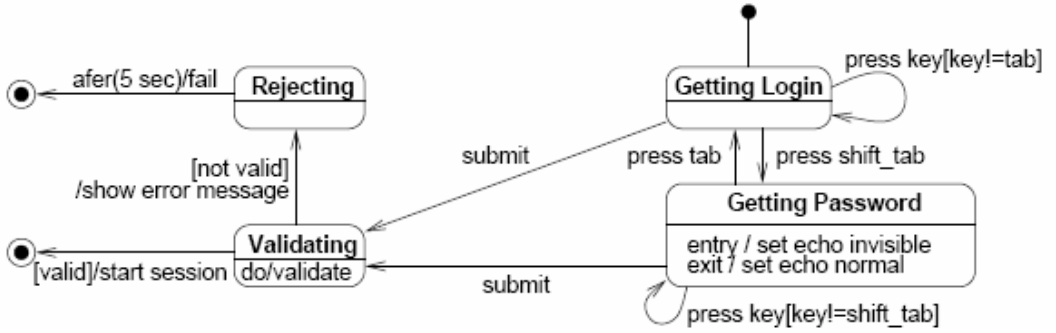
5.1.4 Diagram spolupráce (collaboration diagram)

- popisují spoluprácí mezi objekty nebo jejich rolemi
- znázorňuje strukturu objektů i dynamické chování
- vyskytují se zde:
    - spojení pro přenos zprávy – kreslí se jako plná čára
    - zprávy (stimuly) – kreslí se jako šipečka blízko čáry; šipečka s plnou hlavou znamená volání procedury, volající pokračuje až po návratu z procedury
    - každá zprává má pořadové číslo, zpráva na nejvyšší úrovni má číslo 1
    - obdélníčky jsou buď objekty (název je podtržený) nebo role (není podtržený)
    - lze znázornit iteraci nebo podmíněnou zprávu



5.1.5 Stavový diagram (State transition diagram)

- typicky pro popis chování instance třídy
- stav – během života objektu – provádí nějakou akci, čeká na událost
- přechod – změna stavu způsobená událostí
- uzly grafu = stavy, znázorněny jako obdélníky s kyulatými rohy
- orientované hrany grafu = přechody mezi stavy
- počáteční stav – znázorněn jako černé kolečko
- koncový stav – znázorněn jako černé kolečko v kroužku („býčí oko“)



5.2 CASE systémy (Computer Aided Software Engineering)

- pomáhají zlepšit produktivitu práce tím, že podporují a automatizují činnosti a potřeby
- výhody: kontrola konzistence, generování reportů, dokumentace, kódu, aj…
- nevýhody: drahé, na začátku potřebné zaškolení, doba návratnosti investice, nutnost podřídit se CASE

Rozdělení podle stupně integrace

1. *nižší CASE* – podpora tvorby sw, návrhy obrazovek, formulářů, menu, jazyková podpora
2. *vyšší CASE* – podpora analýzy a návrhu, tvorba diagramů a modelů, kontrola konzistence modelu
3. *integrované CASE* – podpora živ. Cyklu, od analýzy po generování kódu, podpora tvorby dokumentace
4. *komponentové CASE* – integrace nástrojů od různých výrobců, SCM, testování

Rozdělení podle rozsahu možností

1. Jedna fáze – podpora jedné fáze ŽC (životního cyklu) – analýza, prog.
2. Jedna metodika – podpora dané metodiky pře ŽC
3. Více fází, více metodik – transformace modelů, vlastní metodiky
4. Vývoj + management – včetně podpůrných funkcí pro řízení

6 Návrh a implementace s použitím návrhových vzorů a komponent

Návrhové vzory jsou stavebními bloky pro konkrétní řešení, většinou se jedná o takové řešení, které se často opakují. Návrhové vzory tedy řeší netriviální situace a často opakující se řešení. Avšak není cílem zachytit řešení konkrétního problému, který se pravděpodobně již nevyskytne, ale řešení obecného problému.
Vzory pro návrh architektury softwarových aplikací mají následující vlastnosti:

- Vzory se vztahují k opakujícím se problémům při návrhu a popisují jejich řešení.
- Vzory dokumentují existující a ověřené zkušenosti. Nejsou tedy vytvářeny uměle. Zpřístupňují zkušenosti expertů.
- Vzory identifikují a specifikují abstrakce, ze kterých vycházejí konkrétní softwarové komponenty.
- Vzory poskytují společný slovník a chápaní principů návrhu.
- Vzory dokumentují architekturu software.
- Vzory pomáhají zvládat složitost. Když narazíme na situaci, pro kterou existuje návrhový vzor, není žádný důvod pro vymýšlení nového řešení. Jestliže vzor správně implementujeme, pak se můžeme spolehnout na řešení, které poskytuje.

Lze vyslovit definici:
Návrhový vzor popisuje konkrétní opakující se problém spojený s návrhem spolu s příslušným kontextem a prezentuje ověřené obecné schéma pro jeho řešení. Schéma řešení je specifikováno popisem zúčastněných komponent, jejich odpovědností, vztahů a způsobů jejich spolupráce.

Dva druhy návrhových vzorů:
Implicitní jsou nikde nepopsaná řešení odvíjející se od znalostí a zkušeností.
Explicitní , kterých je mnohem méně, jsou zdokumentované zkušenosti při řešeních konkrétních problémů.

6.1 Základní prvky návrhových vzorů

- **Název** - každý návrhový vzor by měl mít název, který co možná nejvíce vystihuje jeho podstatu.
- **Problém** - každý návrhový vzor řeší nějaký problém. Podstatou jeho vzniku je nějaká konkrétní situace, která neodpovídá požadovanému stavu.
- **Podmínky** - popis všech okolností a sil, které mohou ovlivňovat použití daného vzoru. Jsou zahrnuty veškeré jevy, které se vyskytují v daném kontextu a které musí být brány v úvahu.
- **Řešení** - soubor pravidel a vztahů, které popisují jak dosáhnout požadovaného výsledku. Představuje soubor instrukcí, jak postupovat krok po kroku směrem k cíli. Řešení může obsahovat i popis jednotlivých úskalí a omezení, s kterými se musí počítat při jeho implementaci.
- **Příklady** - každý popis návrhového vzoru by měl obsahovat i ukázky jeho praktického použití, které pomůžou uživateli osvojit si novou myšlenku.
- **Výsledek** - stav nebo konfigurace systému po aplikaci vzoru, zahrnující vzájemné souvislosti, které vycházejí z implementace vzoru. Měl by obsahovat shrnutí problému, které byly vyřešeny.
- **Odůvodnění a souvislosti** - představuje vysvětlení, proč byl návrhový vzor použit a jak jeho implementace vyřešila danou situaci. Je vysvětlen způsob práce s návrhovým vzorem v praxi.
- **Související vzory** - vzor může souviset s ostatními, nastane nutnost současné aplikace s jiným vzorem. Jedná se o situaci, kdy samostatné použití popisovaného vzoru je buď úplně nebo skoro neefektivní. Pro jeho správnou aplikaci musí být použit jeden nebo i více podpůrných vzorů, které zajišťují jeho plné využití.
- **Reference** - popisují situace, kdy byl vzor již použit na existujících systémech. Použití návrhového vzoru může sloužit i jako ilustrativní příklad.

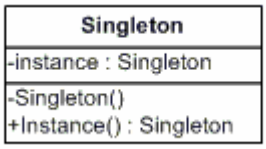
6.2 Základní rozdělení

- *Vytvářející vzory (Creational patterns)* – řeší problémy související s vytvářením objektů v systému. Snahou těchto návrhových vzorů je popsat postup výběru třídy nového objektu a zajištění správného počtu těchto objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu. Mezi tyto návrhové vzory patří:
  - Factory Method Pattern
  - Abstract Factory Method Pattern
  - Builder Pattern
  - Prototype Pattern
  - Singleton Pattern
- *Srukturální vzory (Structural Pattern)* – představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódu. Mezi tyto návrhové vzory patří:
  - Adapter Pattern
  - Bridge Pattern
  - Composite Pattern
  - Decorator Pattern
  - Facade Pattern
  - Flyweight Pattern
  - Proxy Pattern
- *Vzory chování (Behavioral Patterns)* – se zajímají o chování systému. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena spolupráce mezi objekty a skupinami objektů, která zajišťuje dosažení požadovaného výsledku. Mezi tento typ vzorů můžeme zařadit:
  - Chain Of Responsibility Pattern
  - Command Pattern
  - Interpreter Pattern
  - Iterator Pattern
  - Mediator Pattern
  - Memento Pattern
  - Observer Pattern
  - State Pattern
  - Strategy Pattern
  - Template Pattern
  - Visitor Pattern

Dále popíšu nějaké vzory (pokud byste chtěli, tak na http://objekty.vse.cz/Objekty/Vzory-prehled je kompletní popis)

6.3 Singleton (jedináček)

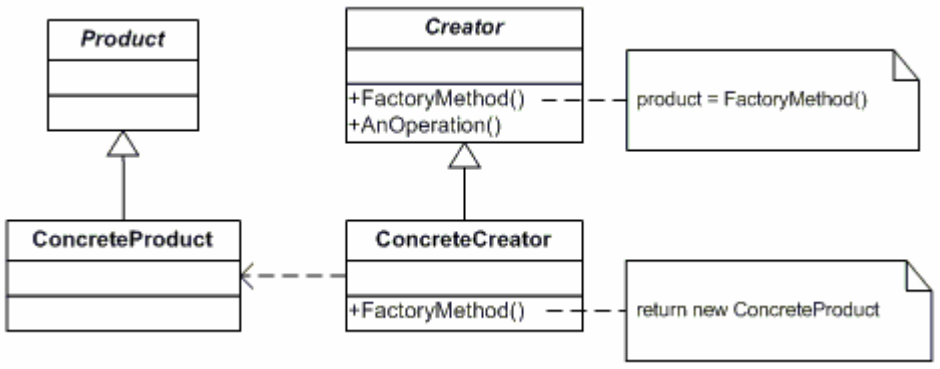
- cílem je zajištění existence pouze jedné instance dané třídy a poskytnutí globálního přístupu k ní
- vhodné např. při logování nebo použíbaní properties souborů (lokalizace)



6.4 Factory method (Tovární metoda)

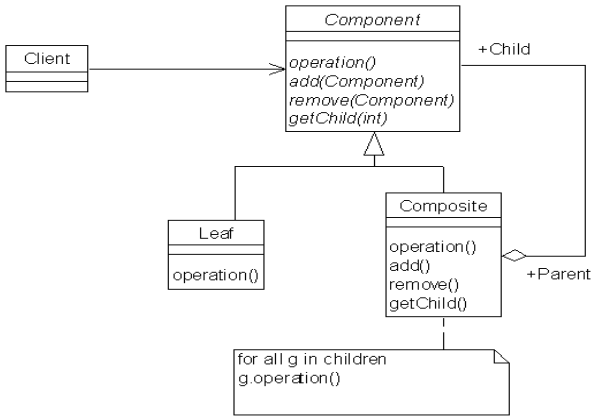
- Definuje rozhraní pro vytváření objektu. Rozhodnutí, u které třídy se má spustit její instance, ale přenechává podtřídám. Tovární metoda umožňuje třídě, aby odložila

rozhodnutí o vytvoření instance na své podtřídy.



6.5 Composite pattern

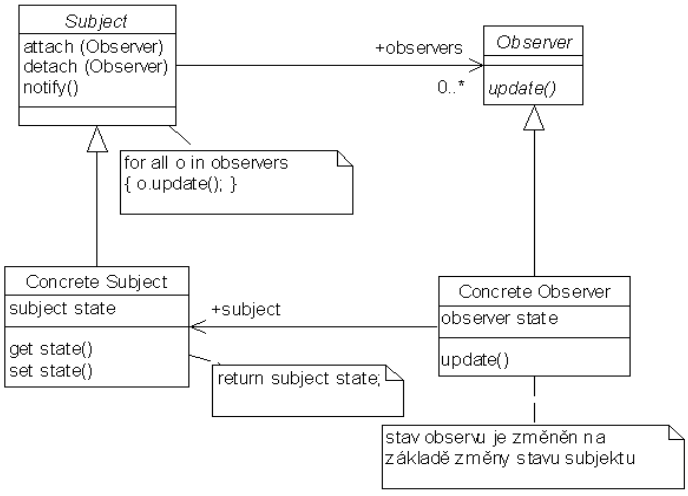
Návrhový vzor Composite představuje řešení, jak uspořádat jednoduché objekty a z nich složené (kompozitní) objekty. Snahou vzoru je, aby k oběma typům objektů (jednoduchým a složeným) bylo možné přistoupit jednotným způsobem.



Obr 20 Composite Pattern Schéma

6.6 Observer

Definování závislosti jednoho objektu k více objektům. Umožnění šíření události, která nastala v jednom objektu, ke všem závislým objektům.



Obr 45 Observer schéma

7 Konfigurační management, systémy pro správu verzí, zpracování požadavků na změny a údržba software

7.1 Konfigurační management

Konfigurační management = Software Configuration Management (SCM)

- proces identifikování a definování prvků systému, řízení změn těchto prvků během životního cyklu, zaznamenávání a oznamování stavu prvků a změn a ověřování úplnosti a správnosti prvků (=jak vytvářet, sestavovat a uvolňovat produkt, identifikovat jeho části a verze, a sledovat změny)

Prvek konfigurace = Configurable Item (CI)

- konstituující složka systému (konfigurace se sestává z prvků konfigurace)
- jsou atomické z hlediska změn a označování verzí, jednoznačně identifikovatelné
- př.: dokument, zdrojový soubor, knihovna, skript, testovací data, …

Konfigurace

- SW konfigurace = souhrn prvků konfigurace reprezentující určitou podobu daného SW systému (př.: první kompilovatelná verze programu XY pro Linux)
- V konfiguraci musí být vše, co je potřebné k jednoznačnému opakovatelnému vytvoření příškušné verze produktu (včetně překladačů, build scriptů, inicializačních dat, dokumentace)
- *Konzistetní konfigurace* = konfigurace, jejíž prvky navzájem bezrozporné (tj. zdrojové soubory lze přeložit, knihovny přilinkovat, …)

7.2 SCM a správa verzí

- správa verzí je součast úlohy SCM (aby prvek konfigurace mohl být ve správě SCM, musí být identifikovatelný, včetně všech svých podob)
- účelem je tedy udržení přehledu o podobách prvků konfigurace
    - verze popisuje jednu konkrétní podobu
    - v úložišti jsou skladovány všechny verze
- druhy verzí
    - historická podoba → revize (př. Word 6.0)
    - alternativní podoba → varianta (př. Word pro Macintosh)
- určení konkrétní verze
    - verzování podle stavu (verze prvku) – identifikují se pouze prvky
    - verzování podle změn (identifikace změny prvku) – identifikují se také změny prvků, výsledná verze prvku vznikne aplikací změn
- popis verze
    - extenzionální verzování: každá verze má jednoznačné ID
        - major.minor + build schéma- např. 6.0.2800.1106 (MSIE 6)
        - kódové jméno: One Tree Hill (= Firefox 0.9)
        - marketingový: Windows 95
    - intenzionální verzování: verze je popsána souborem atributů
        - např. OS=DOS and UmiPostscript = YES

7.3 Prostředí pro verzování: úložiště (=repository)

- = sdílený datový prostor, kde jsou všechny prvky konfigurace projektu
- udržuje si konzistenci
- základní operace
    - inicializace – vytvoření úložiště
    - check out – vytvoření kopie prvku do lokálního pracovního prostoru

    - check in (commit) – uložení změněných prvků do úložiště
    - tagging – označení konfigurace symbolickým jménem
- baseline
    - konzistentní konfigurace tvořící stabilní základ pro produkční verzi nebo další vývoj (př. milník, beta verze aplikace)
    - stabilní = vytvořená, otestovaná, a schválená
- delta
    - množina změn prvku konfigurace mezi dvěma po sobě následujícími verzemi
- kmen (trunk)
    - hlavní vývojová linie
- větev (branche)
    - paralelní vývojová linie
    - izolace hlavního vývoje od jiných změn
- spojení (merge)
    - sloučení změn z větve do kmene
    - cena za izolaci od změn jsou konflikty

7.4 Nástroje pro verzování

7.4.1 Revision Control System (rcs)

- správa verzí pro jednotlivé textové dokumenty, UNIX, Windows, DOS
- ukládá (do `foo.c`, `v` souborů)
    - historii všech změn v textu souboru
    - informace o autorovi, datumu a času změn
    - textový popis změny zadaný uživatelem
    - další informace (stav prvku, symbolická jména)
- funkce: **zamykání souborů**, poskytování R/O kopií (tipuju, že R/O je read only), symbolická jména revizí, návrat k předchozím verzím, možnost větvení a spojování změn z větví do kmene, informace o souboru a verzi lze včlenit pomocí klíčových slov (`$Author$`, `$Date$`. `$Revision$`, `$State$`, `$Log$`)
- používá `diff(1)` pro úsporu místa (rozdíly)
    - poslední revize uložená celá
    - předchozí pomocí delata vygenerované programem `diff(1)`

7.4.2 Realizace variant (cpp)

- C preprocesor umožňuje intenzionální stavové verzování
- například chceme variantu foo.c pro případ OS=DOS and UmiPostscript=YES

7.4.3 Concurrent Versioning System (cvs)

- nadstavba na rcs (používá `v` formát souborů)
- optimistický přístup ke kontrole paralelního přístupu (místo zamkni-modifikuj-odemkni (RCS přístup) pracuje systémem zkopíruj-modifikuj-sluč)
- práce s celými konfiguracemi (projekty) najednou
- zjišťování stavů prvků, rozdílů oproti repository
- free software, příkazová řádka, grafické nadstavby

7.4.4 Subversion (svn)

- následník CVS, způsob práce a příkazy velmi podobné CVS
- nové možnosti: binární diff, meta-data, abstraktní síťová vrstva (DAV), čisté API
- doporučená struktura úložiště (oddělené adresáře pro kmen, větve a značky)

    - branches, trunk, tug

7.5 Správa změn

Problém, proč se to dělá
- Jak zvládat množství požadavků na úpravy produktu (chyby, vylepšení)?
- Jak poznat, kdy už jsou vyřešeny?
- Jak dohledat, co bylo změněno?

Cyklus zprávy změn
- vytvoření/přijetí (přiděli se ID)
- vyhodnocení (možná řešení, jejich dopady a odhad pracnosti)
- rozhodnutí
    - způsob vyřízení (vyřešit/odmítnout/duplikát/odložit)
    - závažnost (kritická chyba/problém/vada na krásе/vylepšení)
    - priorita (vyřídit okamžitě/urgentní/vysoká/střední/nízka)
- zpracování
- uzavření
    - → build: ověření konzistence; → verzování: vytvoření nové baseline
    - Informovat zadavatele hlášení a další zájemce

Detaily hlášení problému
- id, autor, datum, název
- jak chyba vznikla, je možné ji znovu reprodukovat , dodat screenshot, vzorek dat
- informace o použitém sw (operační systém, knihovny (jejich verze))

7.5.1 Change Control Board (CCB)

- skupina členů projektů, která má zodpovědnost za změnové řízení
    - vyhodnocování a schvalování hlášení problémů
    - rozhodování o požadavcích na změny (může významně ovlivňovat podobu a chod projektu)
    - sledování hlášení a požadavků při jejich zpracování
    - koordinace s vedením projektu
- složení
    - jedinec – vývojář, QA osoba
    - tým – technické i manažerské role (vhodné, pokud má změna mít velký dopad)

7.5.2 Systémy pro zprávu změn

Bug tracking (BT) systémy
- evidence, archivace požadavků
- přehled, reporty, grafy, statistiky
- sledování stavu požadavku
- realizace: emailové, webové, klientské
- př. Mantis, Bugzilla, Flyspay

8 Způsoby prevence a detekce chyb v software – testování, oponentury

8.1 Preventivní techniky

- Kontroly
  - prověření meziproduktu nezávislým oponentem říve než se z něj začne vycházet v další práci
  - Technická oponentura
  - Strukturované procházení; Čtení kódu, párové prog.
- Měření
  - kvantitativní ukazatele pomáhají najít slabiny kvality
  - přesnost a dokazatelnost, možnost statistik
  - GQM přístup, FURPS

8.2 Technická oponentura

- Též Faganovská inspekce (Fagan, IBM 1976)
- Skupinová technika (využití diverzity pohledů, cca 4-7 lidí)
- Cíl: odhalit chyby v návrhu/kódu, sledování standardů, vzdělávání
- Ne: dělat potíže autorovi (neúčast vedení), hledat nápravu chyb
- Role ve skupině
  - moderátor – řídí diskusi
  - průvodce – předkládá dílo (není autor)
  - autor – vysvětluje nejasnosti
  - zapisovatel – zaznamenává nalezené problémy
  - oponenti – hledají chyby, obvykle podle seznamů otázek

8.3 Technická oponentura – postup

- Příprava
  - distribuce díla (moderátor), projití a hledání problémů (oponenti)
- Schůzka
  - sekvenční procházení díla (průvodce či moderátor)
  - vznášení připomínek
  - zapisování nálezů (chyb a otevřených otázek)
  - nejvýše 2 hodiny,nepřipouštět dlouhé diskuse, řešení chyb (moderátor), ožná následná schůzka pro vyjasnění otázek
- Závěry
  - verdikt: v pořádku / drobné chyby / nutné přepracování / nová oponentura
  - autor odstraní chyby dle nálezů, moderátor zkontroluje

8.4 Whitebox

- máme k dispozici zdrojové kódy programu, takže je to testování zaměřené na programovou logiku
- používá se k testování malých částí programů, jako jsou podprogramy nebo třídy.
- měli bychom otestovat všechny možné logické cesty(if/then/else) v programu a tím dokázat jeho bezchybnost -> v praxi je to nemužné, příliš mnoho cest

8.4.1 Pokrytí kódu

- Tato metrika nám popisuje, do jaké míry testovací případy pokrývají zdrojový kód.
- Pokrytí všech příkazů (Statement coverage)

`if(a<10 && b=5) b=b/a;`. Oba příkazy (podmínku a přiřazení) bychom mohli pokrýt vstupními daty (a=2 a b=5) a pokrytí by bylo pořádku. Bohužel při vstupu a=0 by došlo k chybě dělení.

- Pokrytí větví (branch coverage)
  se zaměřuje na větvení programu (podmínky, cykly, switch/case). Musíme tedy vytvořit takové testovací případy, které způsobí, že běh programu projde alespoň jednou všechny větve.
- Pokrytí všech kombinací podmínek v rozhodovacím výrazu (Multiple Condition Coverage)
  oproti pokrytí větví vyžaduje, aby byly provedeny všechny možné kombinace vstupních hodnot. Např. Výraz (`a && b && (c || (d && e))`) vyžaduje 6 různých vstupních hodnot. Výraz `(((a || b) && (c || d)) && e)` jich vyžaduje 11, i když mají oba stejný počet operátorů i operandů.
- Pokrytí cest **(**Path Coverage**)** – vyžaduje, aby byla provedena každá možná cesta průběhu programu alespoň jednou.

8.4.2 Jednotkové testy

- zaměřené na verifikaci malých jednotek softwarového návrhu.
- Jsou automatické
- Rozhraní (zda jednotka správně komunikuje s ostatními jednotkami)
- Lokální datové struktury (zda dočasně uložená data zachovávají svou integritu)
- Okrajové podmínky (zda modul pracuje správně na hranicích, omezujících výpočet)
- Nezávislé cesty (zaručující, že každý příkaz bude proveden alespoň jednou)
- Cesty pro zpracování chyb

8.4.3 Integrační testy

- Integrační testy se provádí hned po „složení“ systému.
- „velký třesk“ - celý systém se sestaví najednou ze všech možných součástí. (riskantní, pouze pro malé projekty)
- „inkrementální přístup“ - postupně přidáváme nové komponenty a hned systém testujeme. V praxi problematické, kvůli závislosti komponent.

8.5 Blackbox

- je metoda testování bez uvažování (resp. bez znalosti) kódu softwaru. Máme tedy k dispozici specifikaci softwaru a samotný software v podobě „černé skříňky”, tzn. že se nemůžeme a podívat dovnitř, jak funguje.
- je doplňkem k metodě testování bílé skříňky, a tudíž se zaobírá jinými chybami.
- nesprávné nebo zcela chybějící funkce
- Chyby rozhraní
- Chyby ve struktuře dat nebo externích databázích
- Neočekávané chování
- Chyby při inicializaci nebo ukončení

8.5.1 Smoke test

- vznikla při výstupní kontrole elektrických spotřebičů, kdy se prostě zapojily do zásuvky a pokud se z nich nekouřilo, prošly testem.
- Aplikace se prostě po přeložení spustí a pokud něco dělá (a tím se nemyslí dumpování jádra), tak to projde testem.

8.5.2 Zátěžové testy

- aplikací, které jsou dělané pro zpracovávání velkého množství dat, např. databáze, webové severy, distribuované výpočetní systémy.
- Princip testu: postupně zvyšujeme zátěž dokud nenarazíme na chybu aplikace nebo na určitou mez.
- Pomocí této metody se dá otestovat, jestli případná havárie nezpůsobí ztrátu dat nebo dokáže odhalit chyby, které by se normálně neprojevily.

8.5.3 Testování systému

- Účelem této metody, je otestovat systém jako celek. Testují se tedy různé charakteristiky, např. doba odezvu, kompatibilita, instalovatelnost atd.

8.5.4 Testování hraničních případů

- vstupní hodnoty velmi blízko hraničním hodnotám, způsobují aplikacím velmi velké problémy
- výhody:
    - velmi dobré rozeznání budoucích chyb
    - velmi jednoduché a velmi malé testy
- nevýhody
    - nevyzkouší všechny možnosti vstupních hodnot
    - nehledí na závislosti vstupních hodnot - tím je myšleno, že vstup složený z hodnot např. 1, 2, 3, nekontroluje jako všechny kombinace vstupních hodnot

8.5.5 Návrh testovacích případů (data)

- Dobrý test je takový test, který najde chybu.
- Dobrý test bude vyvolávat co nejvíce vstupních podmínek, a tím omezí celkový počet potřebných testovacích případů.
- Testovací případy budou typicky obsahovat:
    - příliš málo dat nebo žádná data
    - příliš mnoho dat
    - neplatná data (např. negativní počet zaměstnanců)

8.5.6 Shrnutí

- Existuje velké množství metodik, které mají své silné, ale bohužel i slabé stránky, a proto je dobré připravovat testovací metody pomocí více metod.
- Testy vytvářet souběžně s aplikací, ne až nakonec.
- Testy provádět v logické posloupnosti (nejprve smoketest a potom až zátěžový test)

9 Systémy zabezpečení kvality softwarového procesu, přehled metodiky CMM a norem ISO 9000

Systém zabezpečení kvality je soustava organizačních postupů a technických nástrojů, které mají zajistit tvorbu kvalitních produktů či poskytování kvalitních služeb (tj. to, že budou odpovídat požadavkům). Zvlášť významný pro software je proaktivní přístup, což je snaha zajistit správnost výrobků během vývoje a výroby, nikoli až odstraňováním nekvalitních při výstupní kontrole.

- Pokud bude kvalitní proces návrhu a výroby, bude kvalitní i produkt
- Základní kameny
  - techniky zajištění kvality (oponentury, testování, …)
  - rozumný proces (analýza, plánování, …)
  - řízení procesu (plánování, management)

- **Složky systémů řízení kvality**

Systém se týká celé organizace všech pracovníků

- Organizační prvky
  - podpora vedení
  - Manažer + oddělení pro otázky kvality
  - Interní kontroly – dokumentace, postupů
- Dokumentace
  - normy a záznamy
  - Standardy a definice – obecný popis (vlastností) systému
  - Politika jakosti – přístup ke kvalitě
  - Příručka jakosti – popis procedur
  - Plány pro celý vývojový cyklus
  - Záznamy – o dosažené kvalitě, průběhu vývoje, vzdělávání, …
- Audit
  - důkaz o kvalitě pro zákazníky/klienty
  - Certifikační (registrační)
  - Průběžný – periodická kontrola

- **Zavádění systému řízení jakosti**

- Základem je motivace lidí ke kvalitní práci
  - příklad vedení
  - firemní kultura
  - odměňování
- Zavádění systému řízení jakosti
  - dogmaticky
    - podle normy/příručky se vyrobí „ten správný“ systém
    - je třeba jej vnutit lidem
  - pragmaticky
    - systém je navržen na základě znalosti účelu, principů, vzorů řešení
    - … a s ohledem na lokální potřeby, zvyklosti, způsoby práce
    - lidé se jej postupně učí
    - podle jejich připomínek se doplňuje jsou zataženi do jeho tvorby

- **Základní normy**

- ISO 9000 (revize 1994, 2000)
  - norma ISO, EU, ČR
  - výrobní sféra i služby
- CMM (Capability Maturity Model),

- CMMI (CMM Integration)
  - Carnegie Mellon University, USA (1993, 2000)
  - pro softwarový průmysl
- Další „lokální“ a související standardy
  - TickIT (Velká Británie)
  - SPICE - ISO/IEC 15504 (USA)

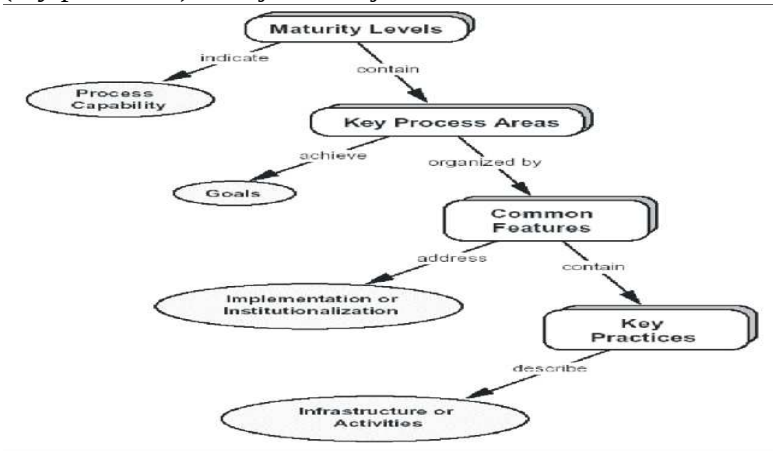
CMM

CMM (Capability Maturity Model for Software) – model procesní vyspělosti byl vyvinut v Software Engineering Institutu na Carnegie Mellon University v Pittsburgu v druhé polovině osmdesátých let.
Definuje pět úrovní procesní vyspělosti a obsahuje sadu doporučených praktik v několika klíčových procesních oblastech, pomocí kterých dochází ke zvyšování vyspělosti procesu vytváření softwaru. Slouží tvůrcům a dodavatelům softwaru jako návod, jak dostat pod kontrolu procesy vývoje a údržby softwaru a jak je rozvíjet směrem vysoké kultuře softwarového inženýrství a vynikajícímu řízení.
Model byl vytvořen, aby pomohl organizacím určit dosažený stupeň procesní vyspělosti, několik nejkritičtějších problémů v oblasti kvality softwaru a zvolit strategie zlepšení procesů, tudíž jak se posunout na vyšší úroveň procesní vyspělosti.

- Způsob hodnocení SW procesů a jejich zařazení do úrovní
  - vyzrálosti na základě klíčových prvků ovlivňujících efektivitu
  - a kvalitu
  - z úrovně plyne pravděpodobnost dosažení kvalitního produktu
- Účel
  - vodítko pro zlepšení kvality tvorby software
  - kritéria pro výběr subdodavatelů
- Vznik
  - 1991 Software Engineering Institute CMU; Humprey, Paulk
  - na základě studia procesů používaných v praxi vč. nejkvalitnějších
  - NASA, Lockheed Martin, Motorola, General Motors

Struktura CMM

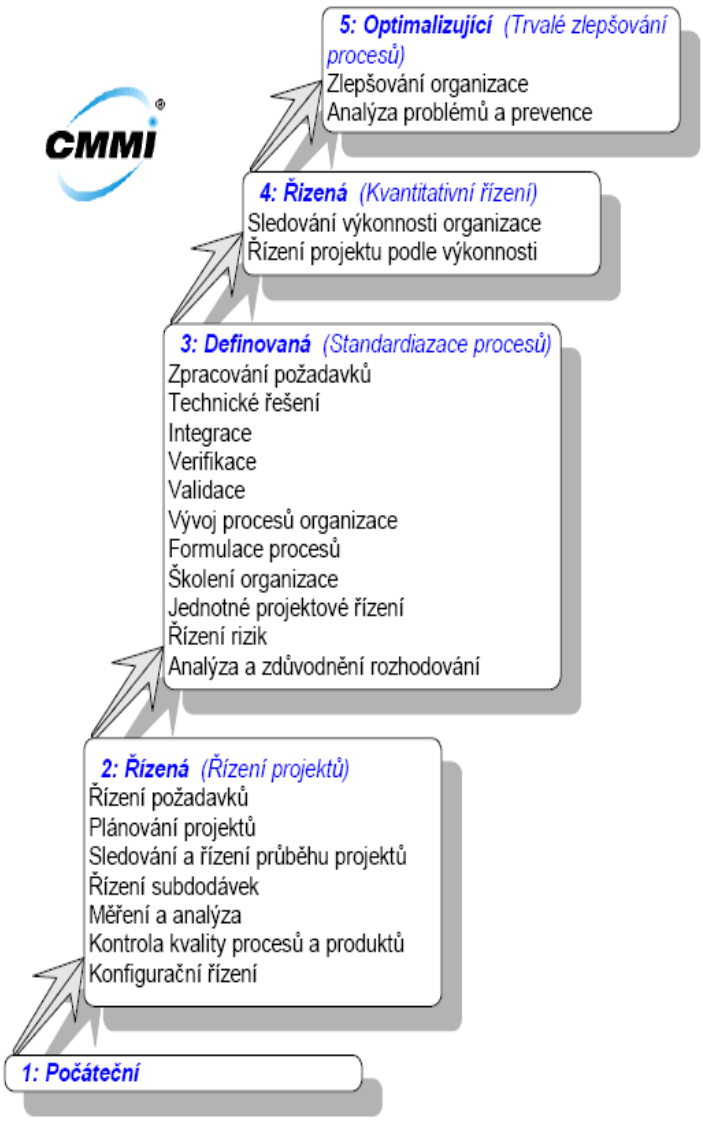
úrovně vyzrálosti procesu (*maturity levels*)
vyzrálost procesu = míra stability (v rámci projektu, mezi projekty), schopnosti detekce a opravy chyb, efektivity, predikovatelnosti výsledků
způsobilost (*capability*) = co je možné od organizace čekat v oblasti kvality
klíčové oblasti (*key process areas*) = na co je třeba se zaměřit pro další zkvalitnění procesu
klíčové techniky (*key practices*) dávají návod jak toho dosáhnout



- **Úrovně CMM**

Počáteční (initial)

- ad-hoc proces (postupy a nástroje podle momentální znalosti nebo nápadu bez celkové strategie), nestabilní až chaotický (není zřejmé, co kdo dělá a proč, kdy co bude hotovo)
- úspěch projektu závisí na silných integrujících osobnostech, chybí základní prvky managementu projektu (plánování, kontrola, kompetence, …)
- problémy zpracovávány neorganizovaně, výsledkem je „code-andfix“ přístup, zpoždění dodávky a/nebo omezení funkčnosti
- proces je nepredikovatelný



Opakovatelná (repeatable)

- stabilní manažerské postupy (plánování, sledování projektu)
- používání v minulosti osvědčených metod a zkušeností vč. dat dle potřeby (všechny oblasti – odhady, management, analýza, testování, SCM, …)
- důraz na analýzu požadavků (DSP) a spolupráci se subdodavateli
- proces je disciplinovaný: ví se, jak zopakovat předchozí úspěchy

Definovaný proces (defined)

- standardní manažerské i inženýrské postupy jsou popsány a integrovány do jednotného základního („standardního“) procesu, který je přizpůsobován pro potřeby jednotlivých projektů
- definování a organizované úpravy standardního procesu má na starosti vyčleněná skupina pro QA
- firma zajišťuje program školení jako mechanismus pro zlepšování znalostí a pro seznamování s definovanými postupy (vč. motivace pro jejich zavedení)
- proces je standardní a konzistentní: funkčnost, cena a termíny jsou pod kontrolou, existuje všeobecné srozumění s postupy, rolemi a zodpovědností

Řízený proces (managed)

- sbírají se naměřená statistická data o produktech i procesu management se může rozhodovat na základě kvantitativních (tj. jednoznačných) údajů
- lze určit odlišit náhodné fluktuace od slabých míst produktu i procesu a podle toho včas reagovat
- řízení kvality spočívá ve snaze odstranit závažné výkyvy (variace v datech)
- proces je predikovatelný: termíny, cena i kvalita jsou v rámci měřitelných a plánovatelných mantinelů

Optimalizující proces (optimizing)

- cíl firmy: produkovat systematicky kvalitní výstupy a kvalitu stále zvyšovat
- analyzují, vyhodnocují a do procesu se zapracovávají nové technologie a nejlepší postupy

tak, aby bylo dosahováno maximální možné efektivity
- proaktivně se identifikují a analyzují (na základě historických dat) slabé stránky procesu a detekují se příčiny tak, aby se předešlo vzniku chyb a problémů; poznatky jsou využívány v nových projektech
- proces je kontinuálně zlepšovaný

- ## CMMI

CMMI je zkratkou z anglického The Capability Maturity Model Integration, volně přeložitelné: integrovaný účinný a funkční model [řízení procesů vývoje], Jedná se o soubor pravidel, požadavků a doporučení, které mají splňovat firemní procesy a co je třeba dodržovat, aby procesy vývoje byly efektivní, účinné a spolehlivé, přičemž důležitou charakteristikou modelu, která jej odlišuje od norem primárně zaměřených na procesy a kvalitu výroby je právě zaměření na procesy vývoje.

- Následník CMM
- Capability Maturity Model® Integration (CMMI)
  - SEI CMU 2002; v1.1 (2006 v1.2)
  - lepší vazba na ostatní modely a standardy
  - širší sada nejlepších technik
- Rozdělení na 4 oblasti znalostí
  - business: systém, software, integrované, služby
  - aktivity: procesní, projektové, inženýrské, podpůrné
- Implementace
  - průběžná (po KPA) × postupná (po úrovních)

- ## ISO 9000-2000

- Standardy (normy) systémů zabezpečení kvality
  - co má systém obsahovat, ne jak se to dělá
  - 8 principů pro řízení jakosti
  - 5 oblastí požadavků na systém zabezpečení kvality
- Důraz na
  - procesní přístup k tvorbě produktu
  - kontrolu procesů
  - certifikaci jako indikaci pro zákazníka
- Obsahuje normy
  - ISO 9000 – základy, zásady a slovník
  - ISO 9001 – systémy řízení jakosti
  - ISO 9004 – směrnice pro zlepšování výkonnosti

- ## Základní principy

- Zaměření na zákazníka
- Efektivní vedení
- Zapojení pracovníků
- Procesní přístup
- Systémový přístup
- Podpora soustavného zlepšování
- Rozhodování na základě faktů
- Spolupráce s dodavateli

- ## Procesy související s QA

- řízení jakosti
- zjišťování externích omezení
- plánování
- kontrola dokumentace
- udržování záznamů
- stálé zlepšování

- interní audit
- přezkoumání systému
- monitoring a měření
- správa nedostatků
- řízení zdrojů
- školení a vzdělávání
- interní komunikace
- průzkum trhu
- návrh produktu
- nákup
- produkce
- poskytování služeb
- ohodnocení potřeb zákazníka
- komunikace se zákazníkem

Procesy nutno vytvořit, zdokumentovat, implementovat, monitorovat a vylepšovat.

- ## Požadavky na systém řízení jakosti
  - Systém
    - vytvořit systém řízení jakosti
    - zdokumentovat jej
  - Management má
    - podporovat kvalitu
    - uspokojovat zákazníky
    - vytvořit politiku jakosti
    - plánovat kvalitu
    - kontrolovat systém
    - provádět přezkoumání
  - Pracovat se zdroji
    - poskytovat kvalitní zdroje
    - mít kvalitní personál
    - vytvořit kvalitní infrastrukturu
  - Kontrolovat a řídit
    - plánování tvorby produktu
    - procesy komunikace se zákazníkem
    - tvorbu produktu
    - nákup; operační činnosti
    - monitorovací zařízení
  - Řešit problémy
    - vytvořit opravné procesy
    - monitorovat a měřit kvalitu
    - mít správu nevyhovujících produktů
    - analyzovat informace o kvalitě
    - zlepšovat kvalitu

- ## Zavádění a použití ISO 9000



a) určování potřeb a očekávání zákazníků a jiných zainteresovaných stran
b) stanovování politiky jakosti a cílů jakosti organizace
c) určování procesů a odpovědností nezbytných pro dosažení cílů jakosti
d) určování a poskytování zdrojů nezbytných pro dosažení cílů jakosti
e-f) zavádění a aplikování metod k měření efektivnosti a účinnosti každého procesu
g) určování prostředků pro zabránění vzniku neshod a pro odstraňování jejich příčin
h) zavádění a aplikování procesu pro neustálé zlepšování systému řízení jakosti

- ## Hodnocení systémů řízení jakosti

Klady:
- produkty podle specifikací
- zlepšená kvalita produkce
- zlepšená komunikace se
- zákazníkem i uvnitř firmy
- ... projeví se nejvíce tam, kde nebylo žádné zaměření na kvalitu

Zápory:
- někdy hlavně kontrola spíš než kvalita
- softwarový proces často zcela nepredikovatelný
- vyžadováno zvenku
- lze předstírat
- nedůvěra k dodavateli
- negativní motivace lidí

- ## Změny oproti ISO 9000-1994

- Snížení počtu norem
- 9001 zahrnuje původní 9001, 9002, 9003
- přizpůsobení struktuře ISO 14000
- Zjednodušení a zpřehlednění
  - osm základních principů managementu
  - požadavek na zlepšování, spokojenost majitelů
  - vodítka pro malé organizace
- Obsahová inovace
  - směrem k aktuálnímu vývoji managementu jakosti a TQM
  - od kontroly procesu k uspokojení zákazníka
  - od předepisující formy standardu k volnosti v rozhodnutí, co je či není pro organizaci důležité