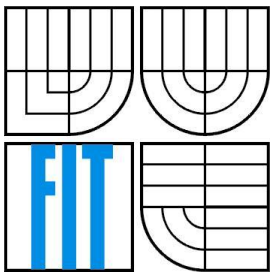
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



AIS

(TEMATICKÉ OKRUHY KE STÁTNICÍM 2009)

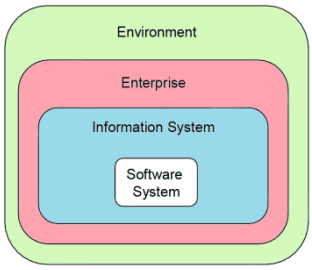
Vladimír Veselý – xvesel38

OBSAH

1 MODELY ŽIVOTNÍHO CYKLU VÝVOJE SOFTWARE

1.1 SW INŽENÝRSTVÍ

Softwareové inženýrství lze chápat jako oblast počítačové vědy (computer science), která se zabývá vytvářením softwarových systémů, které jsou tak rozsáhlé nebo tak složité, že jsou budovány týmem nebo týmy vývojářů. Software je **modelem** reality, proto je SW inženýrství většinou o modelování.



Softwarový systém je speciálním případem systému. Systémem rozumíme kolekci komponent, které jsou ve vzájemném vztahu a které pracují společně k dosažení nějakého cíle. Softwarový systém je tedy systém tvořený softwarovými komponentami (programy, binárními komponentami, agenty apod.).

Softwarovým projektem budeme rozumět plánovanou činnost, která směřuje k dodání nějakého softwarového produktu nebo služby za nějakou dobu. V našem případě je tím softwarovým produktem informační systém.

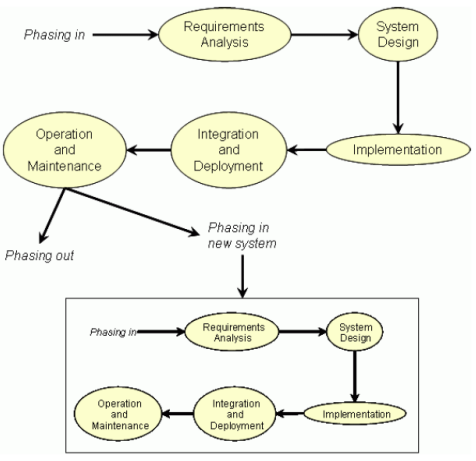
SW inženýrství je více než programování, výsledný produkt by měl splňovat kritéria podporovatelnosti (je pochopitelný + škálovatelný + udržovatelný), formální verifikovatelnosti či přijatelnosti ze strany zákazníka.

Základní kameny SW inženýrství při vývoji SW projektů tvoří:

1) **životní cyklus vývoje**;
2) **modelovací jazyk**;
3) **podpůrné nástroje**;
4) **plánování projektu**;
5) **řízení projektu**.

1.2 ŽIVOTNÍ CYKLUS

Pojem **životněného cyklu** používáme běžně v softwarovém inženýrství pro změny, kterými prochází v průběhu svého „života“ softwarový produkt. Můžeme rozlišit dvě základní období v životním cyklu, prvním je období postupného zavádění (v angličtině **phasing in**), druhým naopak postupného vyřazení z provozu (**phasing out**). V každém časovém okamžiku, snad s výjimkou instalace, se produkt nachází v jednom z těchto dvou období.



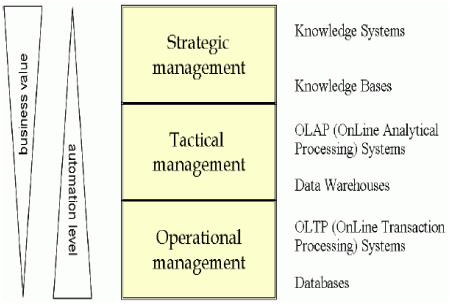
Předchozí obrázek také ukazuje **fáze životního cyklu**:

1) **analýza požadavků**;
2) **návrh systému**;
3) **implementace**;
4) **integrace a nasazení**;
5) **provoz a údržba**.

Ověřování správnosti a testování je součástí každé fáze.

Výsledkem SW procesu je SW systém, který je efektivní, výsledkem business procesu je obchod. SW systém tedy vzniká, aby podpořil business proces. Ve skutečnosti může organizace používat software na **třech úrovních rozhodování**:

- **operační** – každodenní činnost organizace (faktury, účtenky), doména OLTP;
- **taktické** – slouží při rozhodnutích krátkodobého charakteru (*„jaký produkt zdražit?“*), doména OLAP;
- **strategické** – slouží při rozhodnutích dlouhodobého charakteru (*„jaký druh zboží kupovat?“*).



Analýza požadavků znamená analýzu a specifikaci požadavků uživatele, tj. zákazníka, pro kterého systém vyvíjíme. Velmi složitá fáze, ve které chyby vznikají nedostatečnou komunikací či nepochopením ze strany zákazníka a vývojářů. Sestává se z vysoce prefikaných a vymakaných inženýrských technik – **CASE (computer assisted SW engineering)**.

Návrh systému je popis: struktury softwarového systému, který má být implementován; dat, která jsou součástí systému; rozhraní komponent, které systém tvoří; uživatelského rozhraní a případně i použitých algoritmů. Rozpracováním modelu analýzy požadavků vzniká **detailní návrh**. Architektonickým rámcem systému, který musí detailní návrh respektovat, se pak zbývá **návrh architektury**.

Implementace znamená především programování, avšak moderní programování spočívá nejen v zápisu programu v daném programovacím jazyce, nýbrž je založeno na využití existujících či dostupných komponent. CASEu se zde využívá pro generování kostry programu z existujícího modelu návrhu systému. Důležitou částí implementace je **validace** (*„děláme správný produkt?“*) **a verifikace** (*„děláme produkt správně?“*), kterou lze dělit ještě na části:

- **testování** – veškeré aktivity, jejichž cílem je odhalení chyb, mohou být typu:
  - **black-box** – funkční testování ze specifikace;
  - **white-box** – strukturální testování kódu;
- **ladění** – aktivity zaměřené na odstranění (syntaktických či sémantických) chyb.

Fáze **integrace a nasazení** znamená sestavení systému z již implementovaných a otestovaných komponent a následné nasazení u zadavatele k využívání. Klíčovou aktivitou této fáze je **integrační testování**, jehož cílem je ověření funkčnosti celého systému. Testuje se:

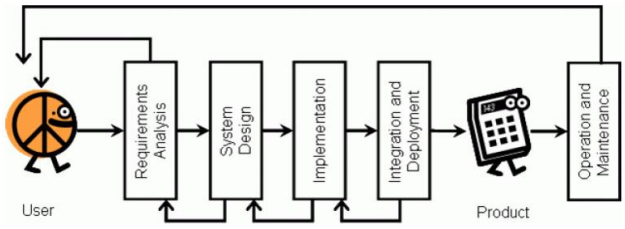
- **shora dolů** – za použití náhražek **(stubs)** dosud neimplementovaných komponent;
- **zdola nahoru** – za použití agregátků **(drivers)**.

Fáze **provozu a údržby** značí období životního cyklu, kdy je systém rutinně používán a rozvíjen. SW sytém se tak dostává do fáze phasing out. Zahájení provozu nového systému současně znamená zahájení údržby tohoto systém, kde rozlišujeme následující typy údržby:

- **opravná** – odstranění chyb a nedostatků odhalených při provozu;
- **adaptivní** - přizpůsobení systému změnám IT/podnikového/podnikatelského prostředí;
- **vylepšovací** – rozvoj systému přidáváním další funkčnosti nebo zlepšováním kvality.

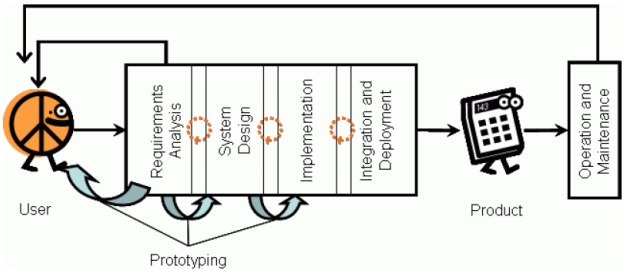
1.3 MODELY ŽIVOTNÍHO CYKLU

MODEL VODOPÁD



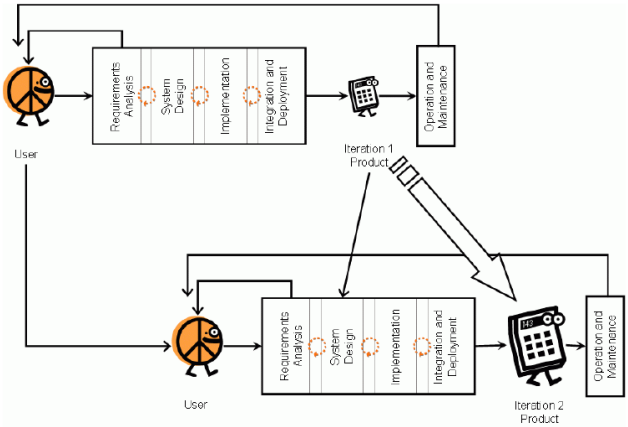
Je historicky nejstarším modelem životního cyklu. Model odráží snahu softwarového inženýrství o prosazení systematického vývoje software. Jeho charakteristickým rysem je, že jednotlivé fáze životního cyklu následují postupně za sebou. Následující fáze může začít teprve tehdy, až je dokončena fáze předchozí. Ukončení konkrétní fáze vývoje je dáno dokončením příslušného dokumentu, který je výsledkem dané fáze. Mezi jednotlivými fázemi je možná a v praxi často nezbytná zpětná vazba.

MODEL VYLEPŠENÉHO VODOPÁDU



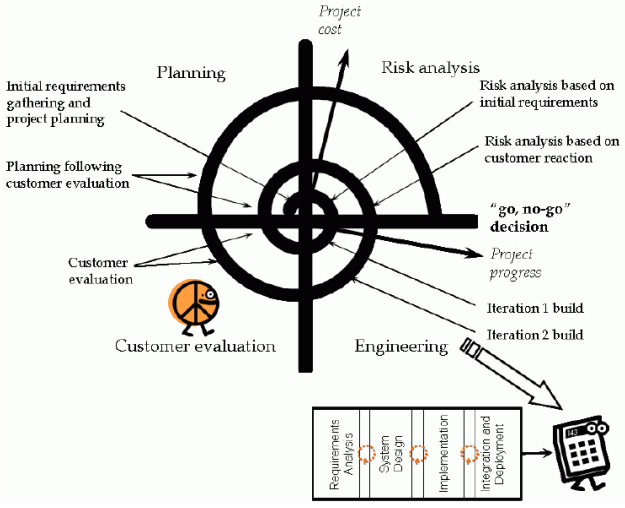
Zavedl **protoypování** a flexibilnější přechody mezi fázemi (jejich překrývání). **Prototypem** zde rozumíme program, který představuje ukázku možného řešení problému. Jde tedy o částečně funkční implementaci, která se použije v rámci analýzy požadavků, kde kvalita implementace je z tohoto pohledu druhořadá.

ITERATIVNÍ ŽIVOTNÍ CYKLUS S PŘÍRŮSTKY



Pro (označované někdy také jako evoluční nebo inkrementální) je charakteristické, že zahrnují iterace. **Iterace** znamená opakovaný průchod fázemi životního cyklu s cílem obohatit vytvářený systém v každé iteraci o nějaké rozšíření či vylepšení, které budeme nazývat **přírůstkem**. Oproti vylepšenému vodopádu iterativní životní cyklus vytváří v každé iteraci novou verzi vytvářeného systému.

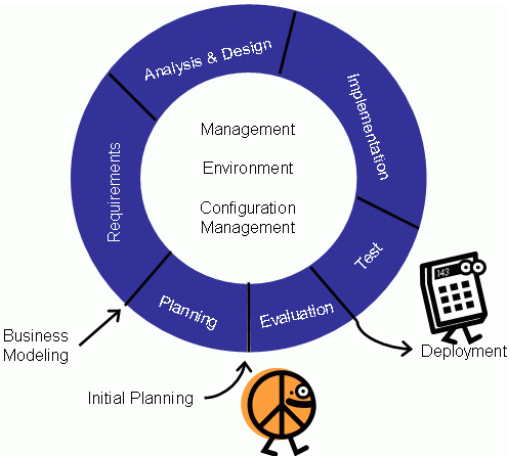
SPIRÁLOVÝ MODEL



Je ve skutečnosti rámcem nebo metamodelem, který může obsahovat jiné modely životního cyklu. Model se prezentuje v podobě spirály, která prochází čtyřmi kvadranty – plánování, analýza rizik, inženýrství a hodnocení zákazníkem. Důraz na opakované plánování projektu a vyhodnocování zákazníkem v každém cyklu dává modelu vysoce iterativní charakter. Analýza rizik jako explicitní fáze je unikátní pro spirálový model, u jiných se v této podobě nevyskytuje.

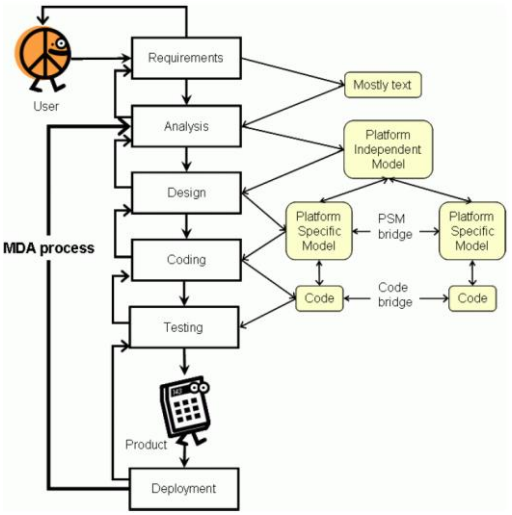
UNIFIED PROCESS

IBM **Rational Unified Process (RUP)** je více než model životního cyklu. Jde rovněž o prostředí (platforma RUP), které slouží vývojářům. To má podobu HTML a jiných dokumentů poskytujících online nápovědu, šablony dokumentace a průvodce. Podpůrné prostředí je součástí balíku CASE nástrojů dodávaných firmou IBM (dříve firmou Rational Corporation), ale RUP lze použít jako model životního cyklu pro vývoj software obecně. Nejvýraznější odlišností od iterativního modelu je explicitní uvedení fáze testování.



MODEL DRIVEN ARCHITECTURE (MDA)

Architektura řízená modelem je rámec životního cyklu, kde je snahou použití jazyka UML jako spustitelné (executable) specifikace, tj. specifikace, ze které je možné vygenerovat softwarový systém, který model v UML reprezentuje. Přestože obrázek připomíná jednoprůchodový životní cyklus vodopád, posloupnost transformací podporuje evoluci probíhající v řadě iterací.



AGILNÍ VÝVOJ SOFTWARE

Má hlavní zásady shrnuty do manifestu, který zavádí následující čtyři priority:

- Jednotlivci/ interakce před procesy a nástroji;
- Fungující software před úplnou dokumentací;
- Spolupráce se zákazníkem před vyjednáváním kontraktu.
- Reagování na změny před dodržováním plánu projektu.

Významnou roli zde pak hraje **refaktorizace kódu**, **extrémní programování** a akceptační testy.

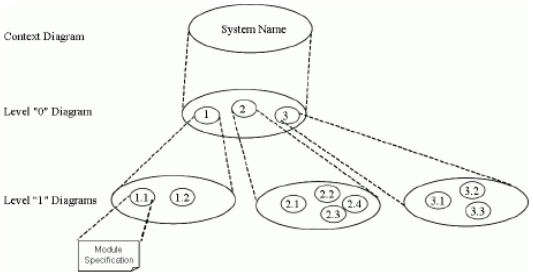
2 MODELOVÁNÍ STRUKTURY V UML 2.0

2.1 ZÁKLADY MODELOVÁNÍ

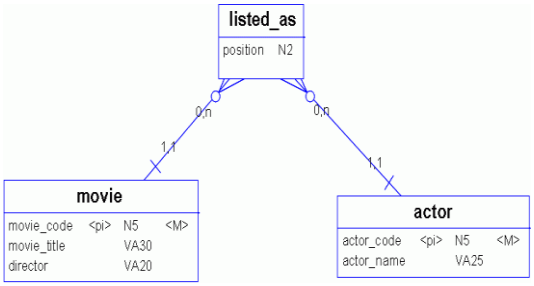
SW inženýrství je především o modelování a modelování jako takové vždy potřebuje modelovací jazyk, jako prostředek pro komunikaci se zákazníky a pro vyjádření abstrakce v průběhu návrhu.

Modelovací jazyky můžeme rozdělit na:

- **strukturované** (obvykle přístup shora dolů při návrhu, je to modelování založené na postupné dekompozici systému na množinu interagujících funkcí v procesu zvaném **funkční dekompozice**):
  - **dataflow (modelování datových toků)** – Základní myšlenka modelování datových toků spočívá v chápání modelovaného systému jako kolekce funkcí označovaných **procesy**. Systém pracuje v kontextu jistého okolí, které mu poskytuje data pro zpracování a naopak systém výsledky v podobě dat poskytuje okolí. Jinými slovy do systému vstupují určité **datové toky** a jiné z něho vystupují jako výsledek zpracování;



  - **entity-relationship** – Model má tři hlavní elementy, a to entity (konceptuální datové struktury), vztahy (představují asociace mezi instancemi entit) a atributy (představují datově-hodnotovou dvojici, která tvoří entity). Představují nejznámější techniku datového modelování, která se i v současnosti používá pro modelování dat, která budou uložena v relační databázi;



- **objektově orientované** – **jazyk UML** (class diagramy, use-case diagramy, interakční diagramy, atd).

2.2 ZÁKLADY UML

UML 2.0 je jazyk pro specifikování, vizualizaci, konstruování a dokumentování artefaktů softwarových systémů, právě tak jako modelování podniku a jiných nesoftwarových systémů. UML představuje kolekci nejlepších inženýrských praktik, které se ukázaly úspěšné při modelování rozsáhlých a složitých systémů.

Především **objekt** budeme chápat jako část software, která má stav, chování a identitu. **Stav objektu** je dán hodnotami jeho atributů. Chování objektu je definováno službami (operacemi), které může objekt provádět, když je o to požádán jinými objekty. **Identita objektu** je specifická vlastnost objektu, podle jejíž hodnoty lze libovolné dva objekty systému odlišit, a to i tehdy, když jsou hodnoty všech atributů obou objektů stejné a oba sdílí stejné operace.

Objekty klasifikujeme do tříd, **třída** tedy určuje typ objektu – definuje, jaké atributy objekt má a jaké operace může provádět. O objektu potom hovoříme jako o **instanci třídy**. Každý objekt třídy nese konkrétní hodnoty atributů definovaných třídou a poskytuje operace určené třídou. Vlastnost objektu, že ostatní objekty mohou ovlivnit stav objektu pouze prostřednictvím těchto operací, se nazývá **zapouzdření**, neboli **ukrývání dat (encapsulation, data-hiding)**.

Dalším důležitým pojmem je **dědičnost**. Jde o vztah mezi třídami, kdy jedna třída (označovaná jako **podtřída/subclass**) dědí všechny vlastnosti (atributy, operace, vztahy a omezení vztahující se na objekty) třídy jiné (označované jako nadtřída nebo také **bázová třída/superclass/base class**). Vlastnosti nadtřídy může dědit několik podtříd. Pokud podtřída dědí vlastnosti jen jedné nadtřídy, hovoříme o **dědičnosti jednoduché**, má-li nadtříd více, hovoříme o **dědičnosti vícenásobné**. S dědičností souvisí i operace bez implementace, která se nazývá **abstraktní operace** a třída, která obsahuje alespoň jednu abstraktní operaci, se nazývá **abstraktní třída**.

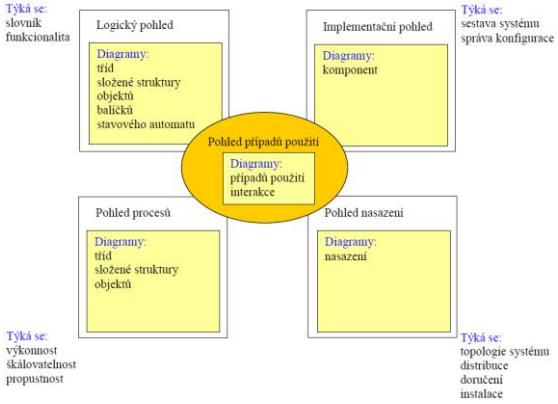
Pokud má jedna operace se stejnou signaturou (tj. hlavičkou) několik implementací, pak jde o **polymorfní operaci**.

Mezi **stavební bloky UML** patří:

- **prvky (things)** – modelovací elementy jazyka;
- **vztahy (relationships)** – modelování vazeb mezi prvky;
- **diagramy** – pohledy do modelů v UML, způsob vizualizace toho, co systém bude dělat (diagramy úrovně analýzy) nebo jak to bude dělat (diagramy úrovně návrhu).

UML chápe architekturu systému jako organizační strukturu systému, vycházející z koncepce „4 + 1 pohledu“:

- **logický** – slovník domény (množina tříd a objektů), jak implementují požadované chování;
- **procesů** – proveditelná vlákna a procesy jako „aktivní třídy“;
- **implementační** – soubory a komponenty tvořící fyzický kód systému a jejich závislosti;
- **nasazení** – fyzické nasazení artefaktů na množině výpočetních uzlů;
- **případů použití** – základní požadavky na systém v podobě množiny případů použití;



2.3 MODELOVÁNÍ STATICKÉ STRUKTURY

Modelování statické struktury se děje pomocí následujících diagramů:

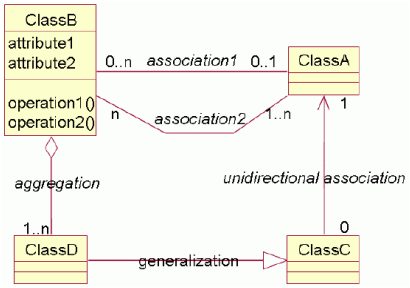
- diagram tříd;
- diagram objektů;
- diagram balíčků;

- diagram komponent;
- diagram složené struktury;
- diagram nasazení.

DIAGRAM TŘÍD (CLASS DIAGRAM)

Základním diagramem UML pro modelování logického pohledu na statickou strukturu systému je **diagram tříd**, který vizualizuje třídy (a rozhraní), jejich interní strukturu a vztahy k ostatním třídám. **Třída** je v UML definována jako popis množiny objektů, které sdílejí tytéž specifikace rysů, omezení a sémantiky. Rysy třídy jsou **atributy** a **operace**.

Vztahy v diagramu tříd jsou významná spojení tříd (případně rozhraní). Existují následující typy vztahů, které používáme v diagramu tříd:



- **asociace** – Reprezentuje vazby objektů daných tříd, podobně jako vztahy v ER diagramu. Může být *n*-ární a zpravidla má uvedenu **násobnost**, někdy i **navigovatelnost**;
- **agregace** – V UML speciální druh asociace modelující vztah celek-část, tj. kde instance jedné třídy **celek/agregát** obsahuje instance jiné třídy **části**;
- **kompozice** - Silnější forma agregace, která vyjadřuje těsnější vazbu celku a částí, kde mohou části patřit vždy jen jednomu celku. Celek je pak zodpovědný za vytvoření a zrušení částí, kde pokud je zrušený celek, musí být zrušeny všechny jeho části (nemohou existovat samostatně);
- **generalizace** – vztahem mezi obecnějším prvkem a specielnějším prvkem, kde specielnější prvek je zcela konzistentní s obecnějším prvkem, ale obsahuje více informací (což souvisí s dědičností).

DIAGRAM KOMPONENT

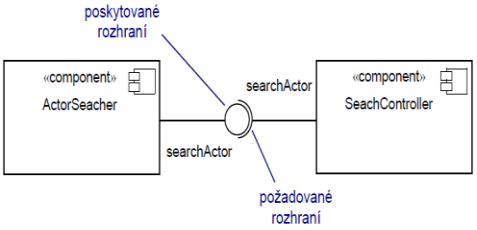
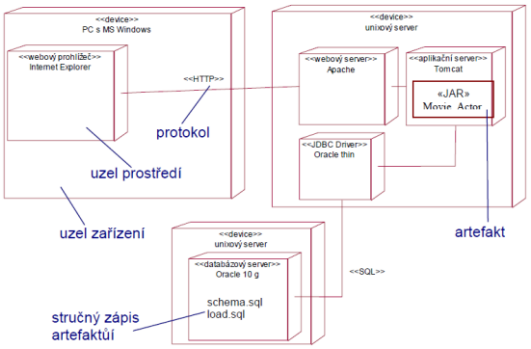


Diagram komponent ukazuje závislosti mezi softwarovými komponentami, včetně klasifikátorů, které je specifikují (např. třída), a artefaktů, které je implementují (např. soubory zdrojového kódu, spustitelné soubory, skripty). Základním modelovacím prvkem diagramu je **komponenta**. Podle UML 2.0 „komponenta reprezentuje modulární část systému, která zapouzdřuje svůj obsah a jejíž projev je nahraditelný v jejím okolí“. Můžeme si ji představit jako černou skříňku, jejíž vnější chování je plně definováno jejími rozhraními, která poskytuje a požaduje. Tím je dáno, že jedna komponenta může být nahrazena jinou, která podporuje tentýž protokol komunikace s okolím. Patří mezi implementační diagramy.

DIAGRAM NASAZENÍ

Diagram nasazení ukazuje nasazení softwarových prvků na fyzickou architekturu a komunikaci mezi fyzickými prvky. Je užitečný pro komunikaci vývojářů navzájem a se zákazníkem v záležitostech týkajících se celkové architektury systému (včetně hardware).

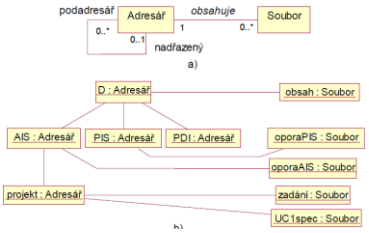
Základním prvkem diagramu je **uzel (node)**. Představuje typ výpočetního zdroje, na kterém mohou být nasazeny softwarové prvky vyvíjeného systému. Protože uzel může mít charakter hardwarového nebo softwarového zdroje, rozlišují se v UML 2.0 dva **typy uzlů**:



- **Uzel zařízení (device node)** – reprezentuje typ fyzického zařízení, jako je PC, server;
- **Uzel prostředí (execution environment node)** – reprezentuje typ prostředí (softwarového), například operační systém Linux, webový server Apache, databázový server Oracle.

DIAGRAM OBJEKTŮ

Ukazuje objekty a jejich vztahy v jistém časovém okamžiku (snímek vytvořený podle diagramu tříd). Lze použít jako pomocný diagram pro vysvětlení nějaké složitější struktury v diagramu tříd.



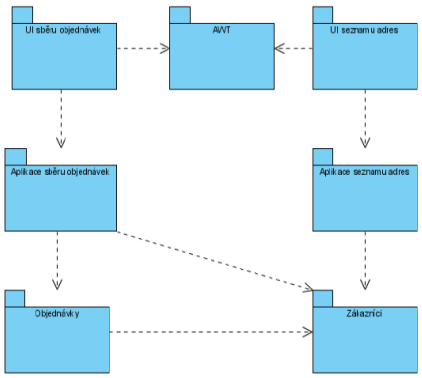
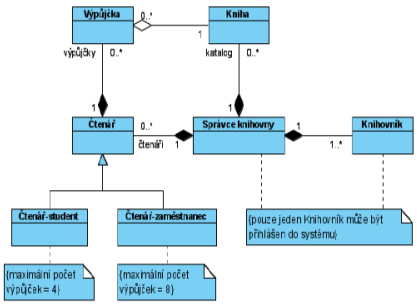


DIAGRAM BALÍČKŮ (PACKAGE DIAGRAM)

Obecně se dá říci, že **diagram balíčků** ukazuje balíčky a vztahy závislosti a generalizace mezi nimi. **Balíček (package)** je prostředek k seskupování (logickému) prvků modelů, díky kterému lze vytvářet dobře strukturované modely.

DIAGRAM SLOŽENÉ STRUKTURY

Ukazuje sestavu propojených prvků reprezentujících instance existující v době běhu programu, které spolupracují pro dosažení nějakého společného cíle. Na rozdíl od diagramů interakce se zaměřuje pouze na zachycení statické struktury (bez zpráv).



3 MODELOVÁNÍ CHOVÁNÍ V UML 2.0

3.1 ÚVOD DO MODELOVÁNÍ DYNAMICKÉ STRUKTURY

Modelování dynamické struktury (chování) se děje v UML za použití následujících diagramů:

- diagram případů použití;
- diagramy interakce;
- diagram sekvence;
- diagram komunikace;
- přehledový diagram interakce;
- diagram časování;
- diagram stavového automatu;
- diagram aktivity.

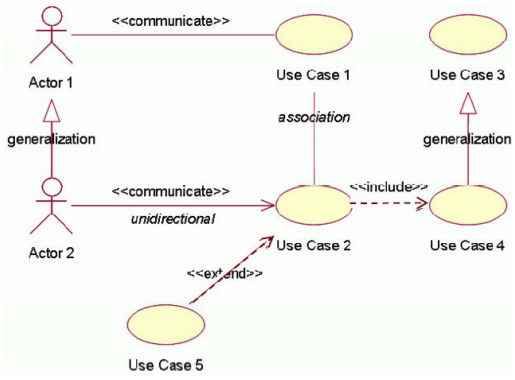
DIAGRAM PŘÍPADŮ POUŽITÍ (USE-CASE DIAGRAM)

Diagramy případů použití jsou hlavní technikou modelování chování systému na úrovni analýzy v UML. Diagram případu použití má v UML zvláštní postavení a to ze dvou důvodů. Jednak jeho síla nespočívá v samotném grafickém vyjádření, ale v podstatně větší míře v textové specifikaci případů použití, která odráží požadavky uživatelů.

Případ použití reprezentuje nějakou důležitou část celkové požadované funkčnosti vyvíjeného systému. **Aktér** reprezentuje uživatele systému (osobu nebo jiný systém) v nějaké roli, který přímo komunikuje s případem použití a očekává od něho nějakou zpětnou vazbu – nějaký pozorovatelný výsledek. Je důležité uvědomit si dvě důležité vlastnosti případů použití:

1) Každý případ použití je vždy zahájen aktérem;
2) Případy použití jsou vždy specifikovány z pohledu aktéra nikoliv systému, tj. popisujeme, co aktér potřebuje, aby systém dělal, ne jak to má systém dělat.

Mezi aktéry může existovat pouze vztah generalizace. Mezi případy použití mohou existovat vztahy generalizace, asociace a stereotypy závislosti «include» (provedení **bázového** případu použití vždy zahrnuje provedení **dodatkového** případu) a «extend» (provedení nějakého případu použití může vyžadovat provedení jiného **rozšiřujícího** případu použití a to v tzv. **bodu rozšíření**).



Specifikace případu použití textovou formou je podstatnou součástí modelu. Jedná se v praxi o strukturovaný popis, protože umožňuje lepší orientaci a snižuje riziko nejednoznačností. UML nezavádí žádný standardní formát, ale běžně se používá šablona specifikace, která obsahuje následující informace (Jméno případu použití, identifikátor, stručný popis, aktéři, předpoklad, hlavní tok, následná podmínka, alternativní tok).

3.2 DIAGRAMY INTERAKCE

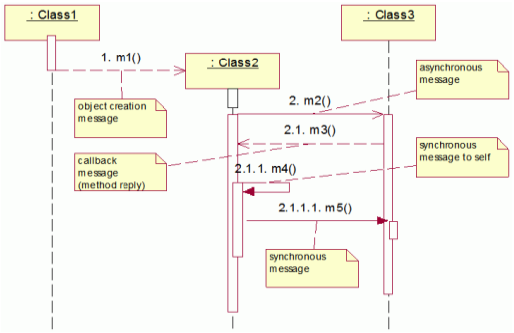
Jsou hlavní modelovací technika úrovně návrhu, ve kterém se řeší modelování interakce tříd spolupracujících k dosažení požadované funkcionality.

DIAGRAM SEKVENCE (SEQUENTIAL DIAGRAM)

Diagram sekvence vizualizuje posloupnost zpráv zasílaných mezi účastníky interakce. Můžeme v něm rozlišit dvě dimenze. Prvou (zleva doprava) je **dimenze účastníků** interakce (v terminologii UML lifelines). Druhou **dimenzí** (shora dolů) je **čas**, který není vyjádřen explicitně, ale zasílané zprávy se v diagramu znázorňují tak, jak jdou po sobě v čase.

Rozlišujeme následující **typy zpráv sekvenčního diagramu**:

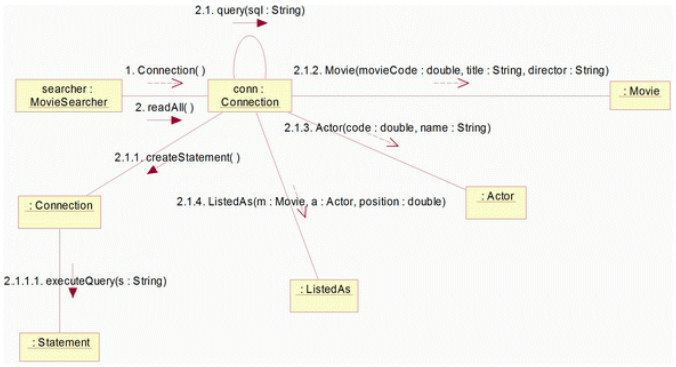
- **asynchronní** – odesílatel zůstává aktivní;
- **synchronní** – odesílatel předává řízení;
- **vytvoření objektu**;
- **zpětné volání** – reakce na zprávu, návrat zpravidla nezobrazujeme.



Do těchto diagramů lze vkládat i operátory kombinovaného fragmentu jako `opt` (podmíněné provedení), `alt` (větvení), `loop` (opakované provedení), `break` (opuštění cyklu), `par` (paralelní provedení).

DIAGRAM KOMUNIKACE

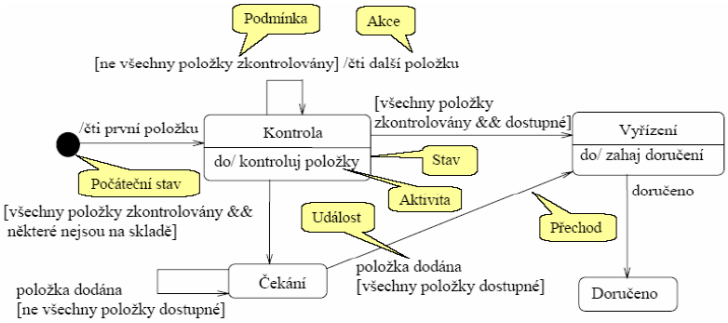
Zdůrazňuje statickou strukturu, která se využije při interakci k dosažení požadovaného chování. Syntaxe podobná diagramu sekvence, ale bez „dvoudimenzionálnosti“ – neobsahuje časovou dimenzi. V diagramu lze vyjádřit iterace i podmínky – ovšem s větším nebezpečím snížení přehlednosti.



STAVOVÝ DIAGRAM

Při objektově orientovaném modelování používáme **stavový diagram** k zachycení **stavů** objektu a **aktivit** objektu v těchto stavech, dále pak **událostí**, které vedou ke změně stavu a **akcí**, které se změnou stavu souvisí. Připomínáme, že stav objektu je určen hodnotami jeho atributů a vazbami na jiné objekty. Stav objektu je určen hodnotami jeho atributů a vazbami na jiné objekty.

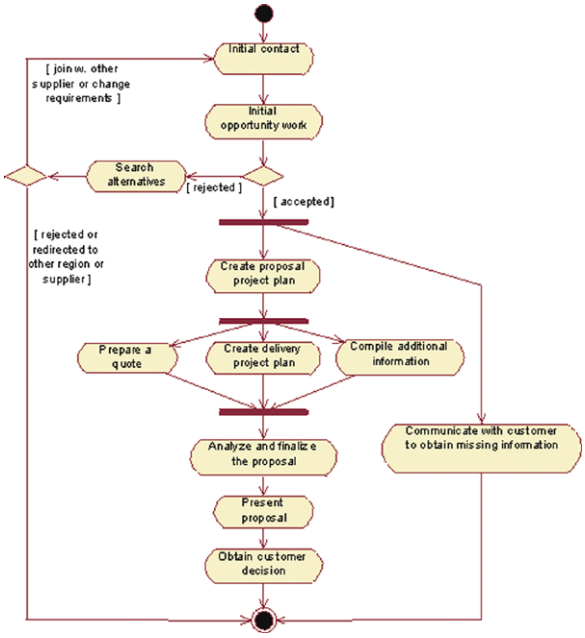
Za zmínku stojí vysvětlení rozdílu mezi pojmy akce a aktivita. **Akce** je v UML chápána jako okamžitá a nepřerušitelná, zatímco **aktivita** trvá nějakou dobu a může být přerušena nějakou událostí. **Aktivita je vždy spojena se stavem**, pozná se podle



uvozujícího `do/`. **Akce je vždy spojena s nějakou událostí**. Tou může být přechod mezi stavy nebo tzv. interní přechod. Akce můžeme dělit ještě na vstupní (`entry`) a výstupní (`exit`).

DIAGRAM AKTIVIT

Vychází z modelovacích technik vývojových diagramů, Petriho sítí a modelování workflow. Modeluje aktivitu, což může být např. business proces, workflow, tok dat, operací, chování případů použití, atd. Modeluje chování bez nutnosti specifikovat statickou strukturu tříd a objektů. Základními entitami jsou uzly a hrany.



Kategorie uzlů:

- **Uzly akcí (action nodes)** – atomické jednotky činnosti;
- **Řídicí uzly (control nodes)** – řídí toky modelovanou aktivitou;
- **Uzly objektů (object nodes)** – reprezentují objekty použité v aktivitě.

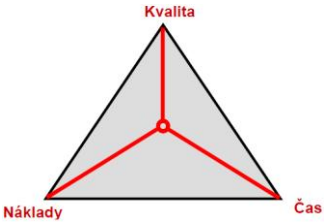
Kategorie hran:

- **Řídicí toky (control flows)** – reprezentují tok řízení aktivitou;
- **Toky objektů (object flows)** – reprezentují tok objektů aktivitou.

Sémantika diagramu aktivit vychází z Petriho sítí, kde pohyb značek (může reprezentovat řízení, objekt nebo data). **Stav modelované aktivity** je dán rozmístěním značek. Akce je spuštěna, pokud jsou značky na všech vstupních hranách uzlu a je splněn jeho lokální předpoklad (**precondition**). Po skončení akce se kontroluje následná podmínka (**postcondition**) a je-li splněna, nabízí uzel značky současně na všech svých výstupních hranách. Jejich šíření po hraně potom závisí na podmínce přiřazené hraně.

4 ÚVOD DO PLÁNOVÁNÍ A SLEDOVÁNÍ PROJEKTU, ŘÍZENÍ PROJEKTU

4.1 ÚVOD DO PLÁNOVÁNÍ PROJEKTU



Základem celého procesu plánování a sledování projektu je tzv. **„trojimperativ projektu“**. Obvykle je znázorňován jako trojúhelník, kde každá jeho hrana či vrchol reprezentuje jednu z dimenzí projektu: **časově** (*„KDY se to má udělat?“*), **věcně** (*„CO se má udělat?“*) a **nákladově** (*„ZA KOLIK se to má udělat?“*).

Plánování a sledování softwarového projektuje nepřetržitou činností plánování toho, kolik času, peněz, úsilí a zdrojů je třeba vynaložit na projekt. Plánování je opakujícím se procesem!

Projekt je časově ohraničená aktivita, která směřuje k dosažení určitého cíle. Jedná se o jedinečnou neopakovanou činnost, která je limitována kvalitou, náklady a časem. Forma projektu je často využívána při zavádění změn, nebo vývoji nových produktů a služeb. Mnoho projektů zadávaných v rámci organizace se uskutečňuje na základě dlouhodobých plánů, vizí, strategií a cílů organizace.

Proces je soubor vzájemně propojených zdrojů a činností, které přeměňují vstupy na výstupy.

Hlavníčásti projektového plánu:

1) **Inicializace** – definování projektových cílů a hlavních omezení času a nákladů;
2) **Organizace projektu** – popis organizace vývojového týmu;
3) **Analýza rizika** – identifikace rizik a způsobů jejich řešení a řízení;
4) **Požadavky na HW a SW zdroje** – specifikace HW a SW potřeb pro vývoj práce;
5) **Struktura členění prací** – definice činností, milníků a dodávek;
6) **Plán projektu** – přerozdělení času a zdrojů odpovídajícím činnostem;
7) **Monitorovací a reportovací mechanismus** – určení potřeb pro monitorovací mechanizmy a řízení reportů.

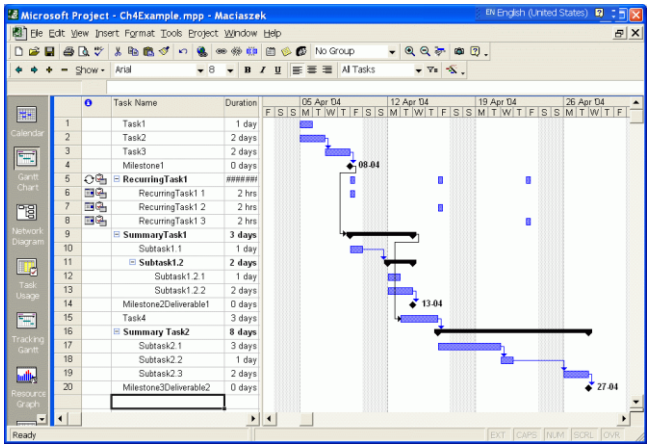
Dodávkami projektu rozumíme softwarové produkty nebo služby, které odpovídají projektovým cílům. **Milník** je kontrolním bodem projektu, který představuje významnou událost v projektu. Označuje konec nějaké aktivity procesu a používá se k monitorování průběhu projektu. Milník je používán zpravidla k reprezentaci kompletace projektové fáze (provedení je typickým výstupem uzavírací fáze). **Fáze** je časový úsek ohraničený počáteční a koncovou událostí v životním cyklu věcného postupu. Na konci každé fáze je vytvořena zpráva o fázi a plán fáze následující. **Etapa** je skupina fází. **Úkoly/Činnostmi** jsou aktivity s definovaným datem zahájení a ukončení, s ideální dobou trvání jeden až dva týdny.

Jakmile jsou dodávky jasně deklarovány a specifikovány, může být celkové úsilí/práce potřebná k jejich realizaci rozčleněna a zorganizována do milníků, fází a úkolů apod. Toto členění je zpravidla označováno jako **WBS (Work Breakdown Structure)**.

Projektové plánování znamená **pohybující se cíl** (komplexnost činností, které na sebe vzájemně navazují). **Kritická cesta** je sekvence vzájemně propojených úkolů v projektu, která musí být

splněna včas, aby mohl projekt proběhnout v rámci daného časového plánu. Zpoždění úkolu na kritické cestě se 100 % promítá do zpoždění projektu jako celku.

Zdroji rozumíme pracovníky, vybavení, materiál, dodávky, software, hardware atd., které jsou potřeba k realizaci projektových činností. Zdroje tedy můžeme rozdělit na **pracovní** a **materiálové**.



Plánování řízené úsilím je přístup k plánování, při kterém je možné měnit jak dobu trvání úkolu (+/-), tak i zdroje je možné libovolně přidávat a odebírat. Úsilí vyžadované k dokončení úkolu zůstává neměnné, ale doba trvání úkolu se změní.

Prostředkem vizualizace plánování je **Ganttův diagram**.

4.2 PLÁNOVÁNÍ ROZPOČTU

Plánování rozpočtu zahrnuje snahu nalézt kompromis mezi úsilím vynaloženým na to, abychom dospěli k přesnějšímu rozpočtu, a náklady na zjištění potřebných informací.

Mezi **hlavní metody plánování rozpočtu** patří následující:

- **Expertní odhad** – je metodou vycházející z odhadu provedeného expertem, podloženém předchozími zkušenostmi, odhad je jen tak dobrý jak dobrý je expert;
- **Odhady na základě analogie** – mohou být použity tehdy, když podobné projekty ze stejné aplikační oblasti byly realizovány a aktuální náklady jsou známy. Rozpočet projektu je poté stanoven analogicky s činnostmi, které se udály v dřívějších projektech;
- **Odhady rozpočtu řízené plánem** – jedná se o tzv.“bottom-up“ metody, berou v úvahu úkoly a zdroje z dříve sestaveného plánu a rozpočet získají sumarizací nákladů zdrojů a jiných fixních nákladů;
- **Odhady rozpočtu založené na modelu/algoritmu** – jedná se o algoritmy, které na základě nějaké měřitelné vstupní hodnoty vypočítají výstupní hodnoty (pracnost, trvání projektu).

FUNCTION POINT ANALYSIS FPA (ANALÝZA FUNKČNÍCH BODŮ)

Patří mezi metriky nepřímé, vychází z empirického vztahu mezi počitatelnými veličinami a ohodnocením složitosti. Tuto metodu lze uplatnit na konci detailního návrhu, kdy jsou již specifikovány transakce v budovaném systému a je jasná struktura a obsah databáze. Metoda samotná se skládá ze dvou základních kroků:

- specifikace složitosti systému;
- převod tzv. funkčních bodů definujících složitost systému na člověkodny pracnosti.

COCOMO

U této metodologie se vychází z ideje, že cena vývoje aplikace přímo závisí na velikosti SW. Přesnost odhadu velikosti SW závisí na etapě vývoje, ve které se projekt nachází, a může se lišit ve výsledku až 4:1 oběma směry. Základním parametrem v COCOMO je tzv. **#SLOC (Source**

Lines of Code) jako hlavní indikátor velikosti a složitosti software. V úvahu bere tři vývojové módy SW, a to **organický** (< 50 kSLOC), **bezprostřední** (< 300 kSLOC) a **vázaný** (> 300 kSLOC). Implementací COCOMO je Costar.

„PRICING TO WIN"

Cena softwaru je stejná částka, jakou může zákazník utratit za projekt ⟹ závislost nikoli na funkcionalitě dodávaného SW a z ní odvozené pracnosti, ale pouze na zákazníkově rozpočtu (skutečná pracnost doplácena zákazníkem při zapracování dalších požadavků).

„PARKINSONŮV ZÁKON"

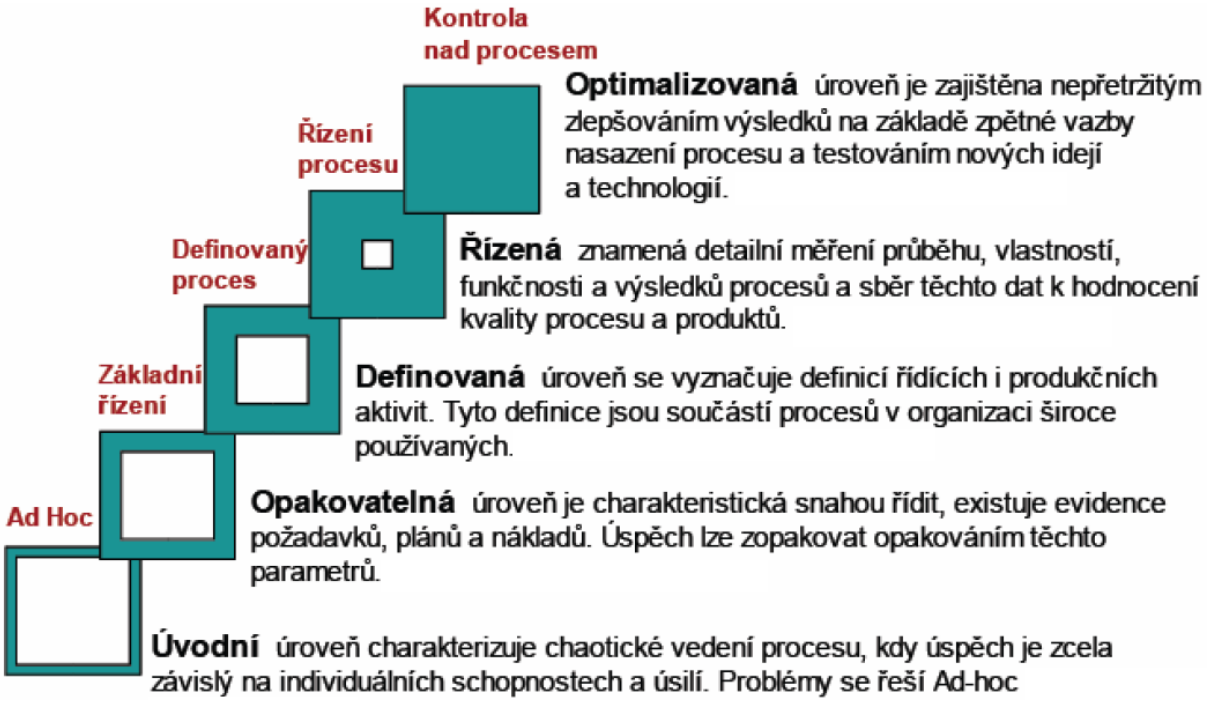
Pracnost je taková, že spotřebuje všechny dostupné zdroje a čas. Pokud má být projekt dokončen nejpozději do jednoho roku a k dispozici jsou 4 pracovníci, celková pracnost bude 48 člověkoměsíců.

KARNEROVA METODA

Metoda odhadu založená na „Use case points". Vychází z teze, že funkcionalita systému z pohledu uživatele je základem pro odhad velikosti informačního systému.

4.3 ÚVOD DO ŘÍZENÍ PROJEKTU

Koncepty způsobilosti procesu a klíčových faktorů ve zlepšování procesů jsou základem pro tzv. **Model vyspělosti řízení procesů (CMM)**. Cílem CMM je poskytnout obraz o tom, jak probíhá řízení informačních projektů z hlediska definovaných klíčových oblastí (zabezpečení, zákaznické požadavky, řízení lidí, aj.).



Řízení projektu je strategie při řízení projektu, která vyžaduje specifické postupy plánování a řízení, zejména s důrazem na čas, kvalitu a omezené zdroje. Projektové řízení je uplatnění znalostí, dovedností, nástrojů a technik v projektových činnostech s cílem splnit nebo překročit potřeby zájmových skupin a jejich očekávání od projektu.

Vedoucí projektu je osoba zodpovědná za řízení projektu. **Projektový tým** se skládá z lidí, kteří pracují na projektu a jsou přímo či nepřímo podřízeni vedoucímu projektu.

Řízení lidských zdrojů zahrnuje procesy požadované pro co nejefektivnější využití osob zapojených do projektu. Zahrnuje procesy:

- **Plánování organizačního uspořádání** – Určování, dokumentování a přiřazování úloh a odpovědností a vztahů podřízenosti a nadřízenosti v rámci projektu;
- **Nábor pracovníků** – Získávání a přiřazování potřebných lidských zdrojů pro práci v rámci projektu.
- **Rozvoj týmů** – Rozvoj dovedností jednotlivců a skupin s cílem zlepšit výkony v rámci projektu.

4.4 ŘÍZENÍ RIZIK

Řízení rizika je systematický proces identifikace, analýzy a reakce na projektová rizika. Zahrnuje maximalizaci pravděpodobnosti (naděje) na dosažení positivních výsledků a minimalizaci pravděpodobnosti důsledků nepříznivých událostí ve vztahu k projektovým cílům.

Mezi **hlavní procesy řízení rizika** patří následující:

- **plánování rizika** – rozhodování o tom, jak přistupovat a plánovat činnosti řízení rizika v projektu;
- **identifikace rizika** – určení, která rizika mohou ovlivnit projekt a dokumentace jejich charakteristik;
- **kvalitativní analýza rizika** – vykonání kvalitativní analýzy rizika a podmínek pro upřednostnění vlivu (důsledků) na projektové cíle;
- **kvantitativní analýza rizika** – měření pravděpodobnosti a míry (důsledků) rizika a odhadování jejich důsledků pro projektové cíle;
- **tvorba protirizikových opatření** – rozvoj postupů a technik ke zvýšení příležitostí a redukci hrozeb ve vztahu k projektovým cílům;
- **monitoring a kontrola rizika** – monitorování zbývajících rizik, identifikace nových rizik, provádění redukce plánů rizik a ověřování efektivnosti prostřednictvím životního cyklu projektu.

Tvorba protirizikových opatření spočívá v definování opatření k využití příležitostí a definování odezev na hrozby, které obecně spadají do jedné z kategorií:

- **předcházení** – Vyloučení konkrétní hrozby, obvykle eliminováním jejích příčin. Řídící tým projektu nemůže vyloučit všechna rizika, ale často může eliminovat konkrétní rizikové události;
- **zmírňování** – Snižování očekávané peněžní hodnoty rizikové události zmenšováním pravděpodobnosti jejího výskytu. Př.: použitím vyzkoušené technologie s cílem snížit pravděpodobnost, že se produkt nepovede;
- **přijetí** – Akceptování následků. Rozlišuje dva základní typy:
    - **aktivní** – kdy sestavíme plán ošetření nepředvídaných událostí pro případ jejich výskytu;
    - **pasivní** – při kterém jsme ochotni akceptovat nižší zisk v případě překročení nákladů některé činnosti.

4.5 ŘÍZENÍ KVALITY

Řízení kvality popisuje procesy požadované pro zajištění toho, aby projekt uspokojil potřeby, kvůli kterým je realizován. Je základní aktivitou, která prochází napříč celému životnímu cyklu. Zahrnuje procesy:

- **plánování jakosti** – vymezení toho, které normy se vztahují na projekt a určování, jak je splnit;
- **zabezpečování jakosti** – pravidelné vyhodnocování celkového plnění projektu s cílem poskytnout důvěru, že projekt bude vyhovovat příslušným normám jakosti;
- **operativní řízení jakosti** – sledování výsledků projektu s cílem určit, zda odpovídají normám jakosti a určení způsobů odstraňování příčin nevyhovujících výkonů.

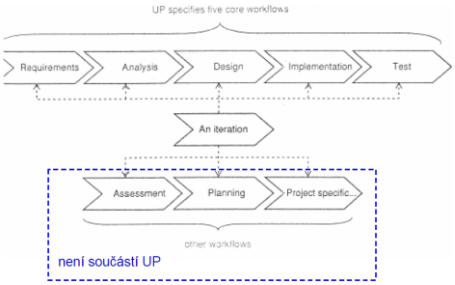
Základní zásada moderního řízení kvality je, že kvalita se plánuje, ne jenom kontroluje!

Kvalitativní charakteristiky SW produktů:

- **Funkčnost (Functionality)** – Množina atributů, která se opírá o existenci množiny funkcí a jejich specifikovaných vlastností. Jedná se o stanovené funkce nebo o funkce vyplývající z potřeb;
- **Přiměřenost (Suitability)** – Existence funkcí pro stanovené úkoly;
- **Přesnost (Accuracy)** – Zabezpečení správnosti/shody výsledků či účinků. Zahrnuje potřebný stupeň přesnosti vypočítaných hodnot;
- **Interoperabilita (Interoperability)** – Schopnost interakce se specifikovanými systémy. Vyjadřuje kompatibilitu k obraně před mnohoznačnostmi při změně prostředí;
- **Shoda (Compliance)** – Jak jsou dodrženy standardy, vztahující se k aplikační oblasti, konvence nebo právní předpisy a ustanovení;
- **Bezpečnost (Security)** – Schopnost produktu zabránit neautorizovanému přístupu k aplikaci a k datům;
- **Spolehlivost (Reliability)** – Množina atributů týkajících se způsobilosti SW udržet jeho úroveň výkonu (stanovené podmínky);
- **Snadnost užívání (Usability)** – Množina atributů souvisejících s vynaloženým úsilím k používání SW a s individuálním posouzením použitelnosti stanovenou/předpokládanou množinou uživatelů (pochopitenost, poznatelnost, ovladatelnost);
- **Výkonnost (Efficiency)** – Množina atributů, které postihují vztah mezi úrovní výkonu software a množství použitých zdrojů se specifikovanými parametry;
- **Udržovatelnost (Maintainability)** - Množina atributů, která se opírá o úsilí vynaložené k provádění specifikovaných modifikací (analyzovatelnost, modifikovatelnost, stabilita, testovatelnost);
- **Přenositelnost (Portability)** – Množina atributů, která se opírá o schopnost provozování software v různém systémovém prostředí (přizpůsobivost, instalace, slučitelnost, nahraditelnost).

5 ÚVOD DO METODIKY UNIFIED PROCESS (DÁLE JEN UP)

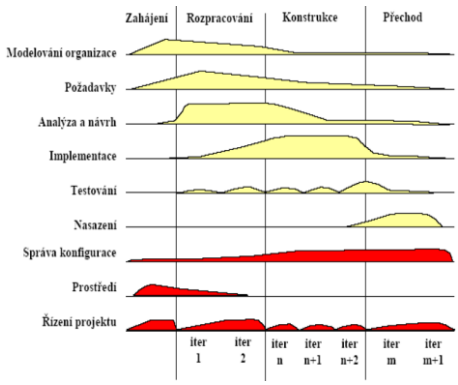
Unified process UP vzniklý v roce 1999 je generický proces vývoje software, který musí být nejprve adaptován pro organizaci a pak pro každý projekt. Musí se tak začlenit do firemních standardů, šablon dokumentů, nástrojů, DB (sledování chyb), musí se zmodifikovat životní cyklus (např. sofistikované metriky pro řízení kvality RT systémů).



Tři základní axiomy:

- **je řízený požadavky a riziky** – význam analýzy rizik;
- **staví na robustní architektuře systému (architecture centric)** – kvalitní architektura je základem kvalitního systému;
- **je iterativní a inkrementální** – každá iterace zahrnuje všechny činnosti běžného SW projektu (plánování, analýzu a návrh, konstrukci, integraci a testování, interní/externí uvedení).

Jsou **čtyři hlavní fáze**, ve kterých se projekt nachází:



- **zahájení (inception)** – Cíli v této fázi je stanovení proveditelnosti (může vyžadovat ověření technologie, ověření splnitelnosti business cílů), vytvoření business případu (projekt bude mít kvantifikovatelný přínos), zachycení podstatných požadavků a identifikace kritických rizik;
- **rozpracování (elaboration)** – Cílem je vytvoření spustitelné verze, zpřesnění ohodnocení rizik, definice atributů kvality, zachytit případy použití 80% funkčních požadavků, vytvořit detailní plán fáze konstrukce a formulovat nabídku zahrnující zdroje, čas, vybavení, lidi a náklady;
- **konstrukce (construction)** – Cílem je dokončení sběru požadavků, analýzy a návrhu, stejně jako celková údržba integrity architektury;
- **přechod (transaction)** – Cílem je opravy chyb, příprava pracoviště uživatele k nasazení nového SW, přizpůsobení SW, aby fungoval na pracovišti uživatele, úprava SW v případě nečekaných problémů, tvorba manuálů a další dokumentace, konzultace pro uživatele a provedení závěrečné revize.

Model požadavků FURPS+:

- **Funkčnost** – rysy, schopnosti, bezpečnost;
- **Použitelnost (Usability)** – lidské faktory, nápověda, dokumentace;
- **Spolehlivost (Reliability)** – frekvence výpadků, zotavitelnost;
- **Výkonnost (Performance)** – doba odezvy, propustnost dostupnost, využití zdrojů;
- **Podporovatelnost (Suportability)** – přizpůsobitelnost, udržovatelnost, konfiguro-vatelnost, internacionalizace;

6 PODSTATA A METODY URČENÍ POŽADAVKŮ

Práce s požadavky vyžaduje systematický přístup ke zjišťování, dokumentování, organizování a sledování měnících se požadavků. **UP chápe požadavky jako měnící se**, vyvíjející se v čase, což je jeho velkou předností oproti vodopádu.

Určení požadavků (u UP ve fázi zahájení a rozpracování) je prováděno obchodním analytikem ve spolupráci se zákazníkem, který poskytuje své požadavky a znalcem oboru, který poskytuje obchodní pravidla a znalosti, které se týkají daného oboru a jsou ve většině případů v čase neměnné. Výsledkem je obchodní model použití a diagram tříd.

Požadavky jako takové lze rozdělit do dvou intuitivních skupin, a to **funkční** (*„Co to má umět?“*) a **nefunkční** (*„Jak to má umět?“* – kvalitativní požadavky na produkt). Požadavky vytváří nějakou hierarchii.

Techniky získávání požadavků:

- **tradiční**:
  - **interview** – s reprezentantem zadavatele nebo se znalcem domény, lze dělit dále na strukturované a nestrukturované;
  - **dotazníky** – pasivní technika, lze dělit na dotazníky s množinou odpovědí a s odpovědí, kterou vytváří sám dotazovaný;
  - **pozorování**:
    - **pasivní** – pozorujeme bez zasahování, třeba pomocí kamery;
    - **aktivní** – analytik se účastní aktivit;
    - **vysvětlující** – uživatel vysvětluje;
  - **studium dokumentů a softwarových systémů** – např. organizační dokumenty, formuláře a sestavy, časopisy a knihy, atd.;
- **moderní**:
  - **prototypování** – rychlé neúplné řešení, většinou k získání okamžité zpětné vazby;
  - **brainstorming** – vysoce propracovaná metodologie ovlivňování mozku za pomocí šokové terapie s historií prvního použití jako mučící techniky KGB;
  - **JAD (Joint Application Devlopment)** – výtvor IBM, který má podobu schůzky a konference skupiny lidí, ve které jsou zástupci vývojářů, zákazníků a moderátora se zapisovatelem.

Požadavky získané od zákazníků se mohou překrývat nebo být v konfliktu. Některé mohou být nejasné nebo nereálné. Proto je nutné provést další **vyjednávání** a **validaci požadavků**. S tím také souvisí požadavky na **správu a sledovatelnost požadavků** (dokumentování změn v průběhu životního cyklu).

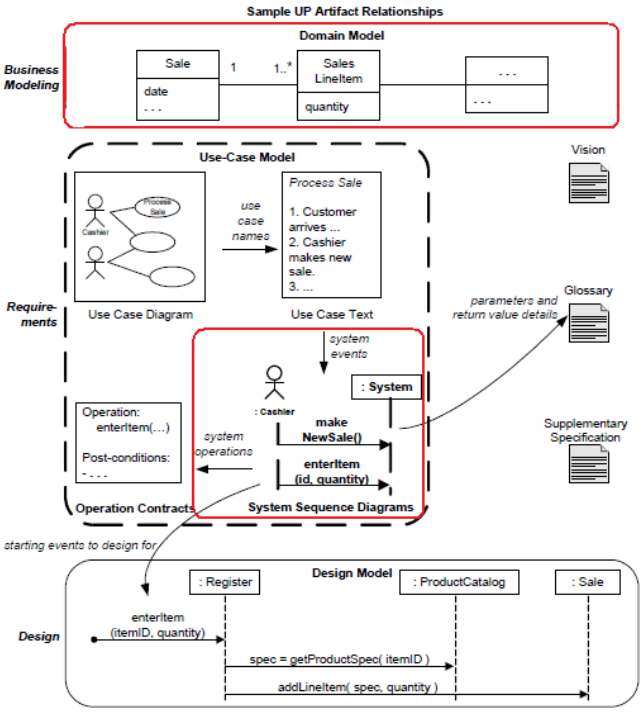
Jedním z důležitých aspektů při vývoji systému je jasná obchodní terminologie a její interpretace, za tímto účelem se vytváří **obchodní slovníček**, který bude sdílen vývojáři.

7 FÁZE ROZPRACOVÁNÍ UP: PODSTATA, MODEL DOMÉNY, SYSTÉMOVÝ DIAGRAM SEKVENCÍ

Artefakty fáze se v rámci UP rozumí obvykle dokumentační výstupy jako např. případ použití, slovníček, plán iterace, prototyp a proof-of-concept, dodatečná specifikace, aj.

Mezi **obvyklé aktivity fáze rozpracování** patří: krátký seminář k požadavkům, pojmenování většiny aktérů, cílů a případů použití (stručný popis, některé však už i detailněji), většina nejrizikovějších a nejvlivnějších nefunkčních požadavků je identifikována, vytvoření seznamu rizik a doplňující specifikace.

Mezi **obvyklé činnosti fáze rozpracování** patří: naprogramovat a otestovat základ a rizikové části architektury, určit a stabilizovat většinu požadavků, snížit či odstranit hlavní rizika, odhadnout celkový plán a zdroje.



Model domény ukazuje koncepty (proto se také někdy nazývá **konceptuální model**) dané aplikační domény, ne softwarové objekty. Je nepovinný, ale může být užitečný (má vliv na softwarové objekty vrstvy domény v modelu návrhu). **Model domény má podobu diagramu tříd** s vynechanými detaily o použitých datových typech pro atributy. Je třeba se vyvarovat nadměrného úsilí na tvorbu dokonalého modelu domény (typické pro model vodopád).

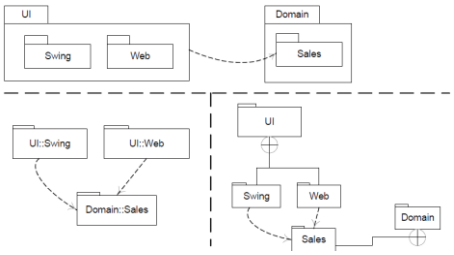
Systémový diagram sekvencí (systém sequential diagram SSD) ukazuje vstupní (generované aktérem, vyžadující nějakou systémovou operaci) a výstupní události systému. Jedná se vlastně o diagram sekvence v UML. Pro konkrétní scénář ukazuje aktéry, generované události a jejich pořadí. Kreslí se pro hlavní úspěšný scénář každého případu použití a časté nebo komplikované alternativy.

Kontrakt operace (Operation Contract) je popis chování systému při systémové operaci. Popisuje zejména stav objektů modelu domény po provedení dané systémové operace (následná podmínka).

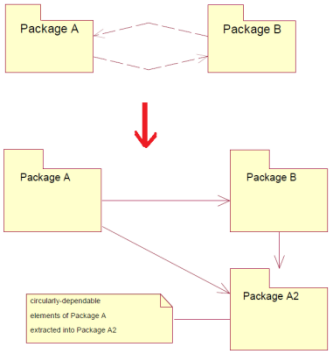
8 NÁVRH ARCHITEKTURY: POJEM LOGICKÁ ARCHITEKTURA, ZÁVISLOSTI BALÍČKŮ, MODELOVÁNÍ ARCHITEKTURY V UML

Softwareová architektura je soubor významných rozhodnutí o organizaci SW systému, výběru strukturních prvků a jejich rozhraní, pomocí nichž je systém sestaven, společně s jejich chováním specifikovaným spoluprací těchto prvků, seskupení do postupně větších podsystémů a architektonický styl, který řídí tuto organizaci.

Logická architektura je organizace tříd do balíčků (jmenných prostorů), podsystémů, vrstev. Neříká nic o nasazení do různých prostředí, na různé počítače atd., tím se zabývá **architektura nasazení (deployment architecture)**.



V UML se k modelování logické architektury používá diagram balíčků.



Balíček (třída) A **závisí** na balíčku B, jestliže si změny v balíčku B mohou vynutit změny v balíčku A. Závislost tříd → závislost balíčků → závislost vrstev. Mezi zdroje závislostí patří dědění, asociace, volání operace či parametry operace.

Problémem bývají i cyklické závislosti balíčků, které lze celkem elegantně vyřešit zavedením agregujícího balíčku nebo použitím interfaceů.

Princip oddělení modelu a pohledu nařizuje, aby se nevkládala aplikační logika do operací tříd UI (např. výpočet daně), ale delegovala požadavky na ne-UI objekty.

9 VRSTVY A ARCHITEKTONICKÉ RÁMCE (MVC, PCMEF)

9.1 VRSTVY

Vrstvou rozumíme seskupení tříd, balíčků, podsystémů, které souvisejí z hlediska zodpovědnosti za nějaký významný aspekt systému. Organizovány hierarchicky, vyšší využívá služeb nižší vrstvy. Mezi typické vrstvy patří vrstva GUI, vrstva aplikační logiky či vrstva technických služeb.

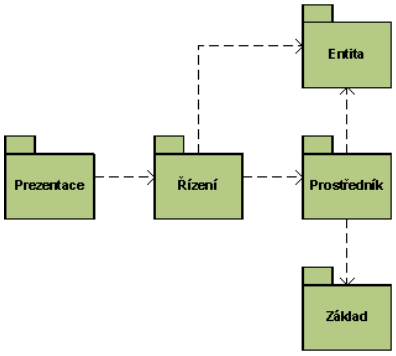
Návrh s vrstvami je organizace do vrstev s různou zodpovědností, s jasným oddělením toho, co řeší tak, že nižší vrstvy poskytují obecnější služby nižší úrovně, zatímco vyšší vrstvy jsou aplikačně specifičtější. Spolupráce a vazba (reprezentovaná závislostmi) je od vyšších vrstev k nižším, ne naopak. Nižší vrstvy by měly být vždy stabilní (jen minimální změny v čase).

Vrstvené architektury můžeme rozdělit na:

- **striktní (uzavřená)** – vyšší vrstva může používat jen bezprostředně nižší vrstvu.
- **uvolněná (otevřená)** – vyšší vrstva může používat několik nižších vrstev.

9.2 PCMEF

Jedním ze známých modelů architektury je **model PCMEF**, podle pěti úrovní hierarchie, které jsou modelovány jako podsystémy. Závislosti jsou jednosměrné, tedy např. prezentace závisí na řízení. Tyto závislosti odpovídají komunikačnímu spojení typu klient/server, kde klientem je v našem příkladu prezentace a serverem řízení. Komunikace může samozřejmě probíhat oběma směry.



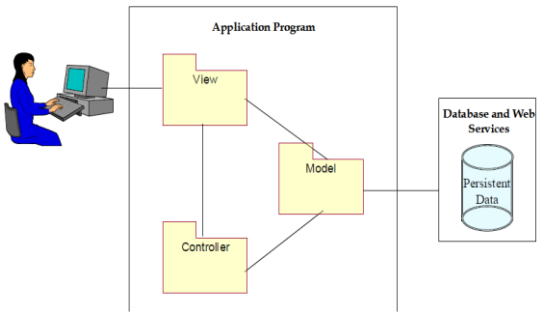
Architektura PCMEF vyžaduje po vývojáři, aby přiřadili každou třídu k příslušnému podsystému. Návrh je tím kvalitnější, protože každá třída tak přispívá k předdefinovanému účelu systému. Propojení tříd je povoleno pouze v rámci v obrázku naznačených komunikačních spojení mezi podsystémy. Podsystémy jsou následující:

- **prezentace (presentation)** – leží na hranici s klienty aplikace, skládá se z tříd, které zajišťují chod uživatelského rozhraní a interakce mezi člověkem a počítačem;
- **řídící (control)** – obsahuje třídy, které jsou zodpovědné za zpracování žádostí uživatelů, které přicházejí z podsystému prezentace a po zpracování žádosti vyhledají příslušné informace buď v paměti, nebo v databázi;
- **entitní (entity)** – spravuje objekty aktuálně uložené v paměti, některé z tříd tohoto podsystému přímo slouží k ukládání údajů, mohou mezi nimi existovat asociace;
- **„prostředník" (mediator)** – je zodpovědný za to, co se děje mezi aplikací a databází, tento podsystém je také zodpovědný za synchronizaci mezi databází a pamětí;
- **základní (foundation)** – obsahuje třídy, které komunikují s databází (např. jsou to třídy, které tvoří dotazy jazyka SQL).

V rámci rámce PCMEF je definováno **sedm základních principů architektury**:

1) **Princip závislosti směrem dolů (DDP – downward dependency principle)** – vyšší úrovně v hierarchii. Nižší vrstvy (Základ, Entita, …) by měly tedy být stabilnější, protože vyšší jsou na nich závislé.
2) **Princip sdělování směrem nahoru (UNP – upward notification principle)** – nižší vrstvy se spoléhají na komunikační rozhraní objektů vyšších úrovní.
3) **Princip komunikace sousedů (NCP – neighbor communication principle)** – objekty dané úrovně mohou přímo komunikovat pouze se svými sousedy.
4) **Princip seznamovacího balíku (APP – acquaintance package principle)** – jde o další podsystém, který obsahuje pouze rozhraní pro komunikaci mezi objekty, které spolu nesousedí, bez znalosti jejich metod.
5) **Princip explicitních asociací (EAP – explicit association principle)** – pokud chce objekt využívat služby jiného, pak mezi třídami těchto objektů musí být asociace.
6) **Princip eliminace cyklů (CEP – cycle elimination principle)** – cyklické závislosti mezi podsystémy, balíčky, třídami apod. jsou nežádoucí, a pokud je to možné, měly by být odstraněny.
7) **Princip pojmenování tříd (CNP – class-naming principle)** – každá třída v systému by měla ve svém názvu zahrnovat úroveň, do které patří v rámci PCMEF.

9.3 MVC



Model MVC (model-view-control) se snaží minimalizovat počet vrstev oproti PCMEF na 3, které jsou:

- **model** – odpovídá vrstvě domény, obsahuje objekty domény, přidává k datům aplikační logiku (např. výpočet sumy prodeje u třídy Sale);
- **view** – zodpovědný za prezentaci modelu (objekty UI, dynamické HTML stránky);
- **controler** – zpracovává a reaguje na události (typicky akce uživatele), může měnit stav objektů balíčků Model i View.

10 NÁVRH TŘÍD A INTERAKCÍ, PRINCIPY PŘIŘAZENÍ ZODPOVĚDNOSTI GRASP DLE LARMANA

Existují tři základní přístupy k návrhu:

- **Přímé kódování** – návrh prováděn při programování, užitečné s nástrojem podporujícím refaktorizaci;
- **Namodelování, pak kódování**;
- **Pouze modelování** – MDA, nikdy zcela automatické, vždy je potřeba připojit nějaký kód.

Podstata OO návrhu je nalézt třídy a interakci jejich objektů, které budou realizovat chování reprezentované případy použití při respektování omezení daných nefunkčními požadavky.

Návrh řízený odpovědností je způsob návrhu založený na uvažování o zodpovědnostech a spolupráci, jejich přiřazování třídám. Hledání množiny spolupracujících zodpovědných objektů, které zajistí splnění cílů návrhu. Existují dva **typy odpovědností**, a to **za činnost** (dělání něčeho, spuštění akce, řízení a koordinace) a **za vědomost** (věci, které lze odvodit nebo zpočítat).

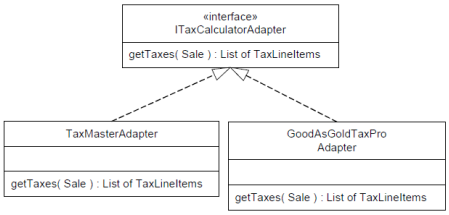
GRASP (General Responsibility Assignment Software Patterns)
GRASP je učební pomůcka pro OO návrh založený na zodpovědnostech. Jedná se o princip přiřazení zodpovědností třídám a objektům podle vzorů:

- **Information Expert** – centrální prvek, který má důležité informace a proto i zodpovědnost;
- **Creator** – třída, která zakládala objekty jiné třídy (důvod: obsahuje/agreguje/často používá danou třídu)
- **Controller** – prvek, který řídí systémové události (není v GUI);
- **Low Coupling** – přidělovat zodpovědnost tak, aby bylo co nejméně závislostí;
- **High Cohesion** – udržovat kohezi (soustředění závislostí na jednom místě);
- **Polymorphism** – je-li chování závislé na typu, zodpovědnost dát polymorfním operacím;
- **Pure Fabrication** – speciální třída, která se použije pro dodržení výše uvedených pravidel (pokud nejde aplikovat na existující třídy).
- **Indirection** – třída, která dělá prostředníka a zamezuje přímému propojení

11 NÁVRHOVÉ VZORY, PODSTATA A PŘÍKLADY

Návrhové vzory jsou osvědčená řešení často se vyskytujících problémů OO návrhu.

ADAPTER



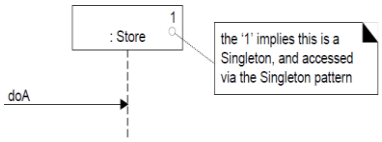
Problém: Jak vyřešit nekompatibilní rozhraní nebo poskytnout stabilní rozhraní pro podobné komponenty s různými rozhraními?
Řešení: Převést původní rozhraní komponent na jiné rozhraní prostřednictvím pomocného objektu adaptoru.

FACTORY

Problém: Kdo by měl být zodpovědný za vytváření objektů, když jde o složitou logiku vytvoření nebo je snaha oddělit zodpovědnost za vytvoření kvůli lepší kohezi?
Řešení: Vytvořit specielní objekt zodpovědný za vytvoření.

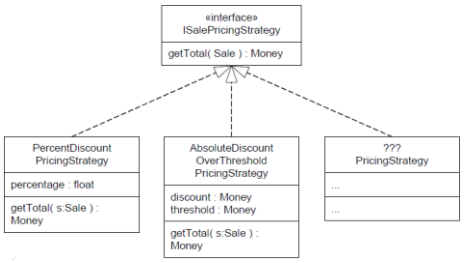
Vylepšená verze továrny, která se nazývá **Abstract Factory**, umožňuje vytvořit rodiny souvisejících tříd, které implementují společné rozhraní.

SINGLETON



Problém: Může existovat jen jedna instance, tj. tzv. „singleton“. Objekty potřebují globální a jediný bod přístupu.
Řešení: Definovat statickou metodu třídy, která vrátí singleton.

STRATEGY



Problém: Jak navrhnout řešení pro lišící se, ale související algoritmy/postupy?
Řešení: Navrhnout samostatnou třídu pro každý algoritmus/postup se společným rozhraním.

COMPOSITE

Problém: Jak zpracovat skupinu nebo složenou strukturu objektů stejně (polymorfně), jako by byly atomickým objektem?
Řešení: Vytvořit třídu pro kompozit a atomické objekty tak, aby implementovaly totéž rozhraní.

FACADE

Problém: Požaduje se společné sjednocující rozhraní množiny nesourodých implementací nebo rozhraní, např. podsystému. Může existovat nežádoucí propojení prvků podsystému, nebo když se implementace podsystému může změnit.
Řešení: Definovat bod kontaktu s podsystémem – objekt **„fasáda“**, který „zabalí“ podsystém. Tento objekt je jediným sjednocujícím rozhraním a je zodpovědný za spolupráci s komponentami podsystému.

OBSERVER (PUBLISH - SUBSCRIBE)

Problém: Různé druhy objektů odběratele se zajímají o změny stavu nebo události objektu vydavatele a chtějí reagovat svým způsobem na události generované uživatelem.
Řešení: Definovat rozhraní odběratele, které implementují konkrétní odběratelé. Vydavatel může dynamicky registrovat odběratele a uvědomuje je, když událost nastane.

12 REFAKTORIZACE, PODSTATA A PRINCIPY

12.1 VÝVOJ ŘÍZENÝ TESTEM

Vývoj řízený testem (test driven development TDD) je metodologie prosazovaná iterativními a agilními metodami, aplikovatelná na UP, která káže, že nejprve se napíše test a teprve potom se implementuje to, co bude testováno. Uspokojení programátorů vede ke psaní konzistentnějších testů, kde dochází k ujasnění si detailního rozhraní a chování, což umožňuje později prokazatelnou, opakovatelnou a automatizovanou verifikace.

12.2 REFAKTORIZACE

V současné době existuje technika, jež umožňuje v rámci postupného vytváření programu a změn prováděných v rámci údržby vylepšovat vnitřní strukturu programu a vylepšit nebo alespoň udržet tak jeho udržovatelnost. Tuto techniku nazýváme **refaktorizace** a formálněji je definována jako disciplinovaná metoda přepisu nebo restrukturalizace existujícího kódu beze změny vnějšího chování, aplikací malých transformací kombinovaných s opakovaným spouštěním testů, což vede ke snadnějšímu pochopení kódu a usnadnění dalších úprav.

Aktivity a cíle refaktorizace:

- odstranění duplicitního kódu;
- zvýšení srozumitelnosti;
- zkrácení dlouhých metod;
- odstranění „natvrdo“ kódovaných literálových konstant.
- zmenšování rozsáhlých metod;
- třídy s mnoha atributy;

Typy/Vzory refaktorizace:

Typ/Vzor	Popis
Extract method	Transformuj dlouhou metodu na kratší vyčleněním logické části do privátní pomocné metody.
Extract constant	Nahraď literálovou konstantu konstantní proměnnou.
Introduce explaining variable	Ulož výsledek výrazu nebo části výrazu do přechodné proměnné se jménem vysvětlujícím smysl
Replace constructor call with factory method	V OO jazycích se nahradí použití `new` a konstruktoru voláním pomocné metody, která skryje detaily vytvoření.

Refaktorizace může být problém při snaze refaktorizovat DB, obvykle se to dá zvládnout u OODBS, kdy měníme interface, který přistupuje k DB.

13 ZAJIŠTĚNÍ KVALITY SOFTWARE: PODSTATA A METODY

Kvalita SW je důležitá, protože na ní záleží – tak to bychom vskutku netušili!

DMAIC model kvality začíná fází *Define*, ve které se vytyčují cíle (definice procesů, plán testů a revizí), který bere v potaz danou organizaci/projekt/jednotlivce. O kvalitě je nutné uvažovat na všech úrovních od organizace až po jedince. Ve fázi *Measure* probíhá validace a verifikace projektu, které se dějí ve formě testů (**V-model**):

- **unity** – souhlasnost s návrhem a jeho implementací;
- **integračních a systémových** – vše je propojeno a funguje jak má;
- **výkonové** – jsme schopni zpracovat všechny relevantní vstupy na výstupy;
- **akceptačních** – splnění požadavků zákazníka;

Ve fázi *Analyze* se zhodnocují výsledky měření a porovnávají se s výsledky předchozího měření (dřívější verze programu). Ve fázích *Improve* a *Control* se pak implementují a kontrolují změny produktu, které mají reflexí důsledku z fáze analýzy.

QA (quality assurance) je nutné naplánovat, proces k ní vedoucí musí být pragmatický. Je nutné tento proces nějak měřit, neb bez měření se nelze zlepšovat. **Přezkoumání** je efektivní (a mnohdy jediný) způsob zajištění kvality.

14 ZAJIŠTĚNÍ BEZPEČNOSTI DAT: NEPOVINNÁ A POVINNÁ AUTORIZACE, DEKLARATIVNÍ A PROGRAMOVÁ AUTORIZACE

14.1 ZÁKLADY

Bezpečnost dat znamená, že data jsou držena v bezpečí a chráněna před korupcí, přičemž přístup k nim je kontrolován ⇒ bezpečnostní požadavky a omezení.

Integrita dat znamená, že data jsou úplná a korektní vzhledem k modelu a požadavkům reálného světa ⇒ integritní požadavky a omezení.

Tři hlavní **úkoly bezpečnosti dat**:

- **confidentiality (secrecy)** – informace by neměla být dostupná neautorizovanému uživateli;
- **integrity** – jen autorizovaní uživatelé mohou měnit data a to v souladu s pravidly na udržení integrity dat;
- **availability** – autorizovaným uživatelům by neměl být odepřen přístup k datům.

Pomocí **bezpečnostní politiky** lze dosáhnout bezpečnosti systému, je nutné ji vynucovat na různých úrovních daného systému (DBS, OS, síť, fyzická bezpečnost, lidský faktor).

Řízení přístupu se dá **kategorizovat** na:

- **nepovinné (discretionary access control DAC)** – které je zaměřené na systém **privilegií**, kdy jsou přístupová práva explicitně přidělována uživateli, případně skupině uživatelů (vytváření rolí);
- **povinné (mandatory access control MAC)** – založena na systému úrovní zabezpečení (**security class**) pro objekty a stupni pověření (**clearance**) pro subjekty v systému.

14.2 DAC

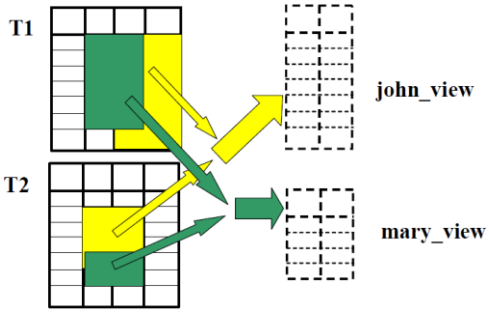
Řeší dvě základní aktivity, a to **autentizaci** (věření identity) a **autorizaci** (ověření přístupových práv).

DEKLARATIVNÍ AUTORIZACE

Je postavena na **systému privilegií**, což vyjadřuje právo vykonat určitou akci nad DB/schématem (systémové privilegium), objektem (objektové privilegium) jako celek. Zatímco objektová privilegia jsou součástí definice SQL, systémová privilegia nejsou a jejich implementace je tak v režii výrobce DBS. Sdružená množina privilegií se nazývá **role** a samotné role jsou pak přidělování skupině uživatelů. V SQL se přidělování/odebírání práv děje obvykle pomocí příkazů `GRANT` a `REVOKE`.

PROGRAMOVÁ AUTORIZACE

SQL ve svém standardu však obsahuje procedurální způsoby, kterými se dá taktéž kontrolovat DAC, a to jsou tři techniky:



- **views (pohledy)** – Umožňují omezit přístup k datům a skrýt tak logickou strukturu DB. Pohledy jako takové nejsou upravovatelné (nejdou využít pro operace `INSERT`, `DELETE` či `UPDATE`), takže chrání data;
- **synonyms (synonyma)** – Umožňují sdílet jednotné schéma mezi uživateli;
- **stored routines (uložené procedury)** – Jsou objekty DB, které obsahují kód, jež se vykoná nad DB, jejich autor tak jasně specifikuje jejich použití. Jsou dostupné přes standardizovaná API (SQL příkaz `EXECUTE`).

14.3 MAC

Uživatelé mají stupeň prověření a objekty DB pak úroveň zabezpečení, přičemž uživatel musí mít stupeň prověření větší nebo roven úrovni zabezpečení objektu, ke kterému přistupuje, aby ho mohl použít.

Prověření a zabezpečení jsou pak spravovány centrální autoritou, a to uživatelé samotní je nemohou měnit.

Bell-LaPadulův model viz BIS.

Obvyklé jsou čtyři úrovně/stupně:

- top secret;
- secret;
- confidential;
- unclassified.