PB007 – SWING výpisky z přednášek

Vypisoval jsem to, co mi přišlo důležité, co by se mohlo objevit na zkoušce. Tyhle výpisky obsahují pouze teoretickou část, bez UML diagramů.

## Přednáška 1 – Proces tvorby

**Generické produkty**

* Stand aolne systémy udělané pro libovolné zákazníky
* CAD, grafické programy

**Customized products**

* Software určený pro specifického zákazníka
* Vestavěné systémy

**Typy aplikací**

* Vestavěné (embeded) systémy – zabudované v počítači
* Dávkové zpracování (batch processing) – vykonávání dávek příkazů určených předem bez zásahu uživatele, maximalizace využití počítače
* Data collection, monitoring
* Standalone – nenačítá žádný externí modul, knihovnu nebo funkci

**Softwarový proces ( tvorba softwarového systému)**

* Specifikace (zákazník se dohodne s vývojářem na podmínkách)
* Analýza a design (požadavky jsou implementovány v designu)
* Validace (kontrola, jestli je vše OK podle zákazníka)
* Evoluce (Software je průběžně měněn podle změny požadavků zákazníka a marketingu)

**Modely vývoje software**

**Waterfall model**

* **Rozdělení projektu do odlišných fází. Je těžké reagovat na změny v požadavcích zákazníka**
* **Většinou se používá pro velké systémy a projekty, kde se systém vyvíjí na několika stránkách a pro generické produkty**

**Inkrementální vývoj**

* **+ Cena za přápadné změny v zadání je značně menší**
* **+ Je jednodušší dostávat feedback od uživatele**
* **+ Rychleji se uživateli dodává užitečný software (uživatel jej může používat už před dokončením)**
* **- Proce není viditelný, není produktivní dělat dokumentaci pro každou verzi**
* **- Systémová struktura degraduje s každou změnou (ledaže je na to věnováno víc peněz a času, stává se struktura s každou změnou složitá a drahá)**

**Reuse-Oriented Swing**

* **Založeno na systematickém znovuužití, kde jsou systémy integrovány z existujících komponent (commercial-off-the-shelf)**
* **Jedná se o nyní standardní budování bussiness systémů.**

**Prakticky se využívá kombinace všech modelů.**

## Přednáška 2 – Funkční požadavky

**Uživatelské požadavky -**Výroky v přirozeném jazyce plus diagramy služeb, které bude systém poskytovat. Napsáno pro uživatele

**Systémové požadavky –** Strukturovaný dokument obsahující detailní popis funkcí systému. Definující, co by mělo by být implementováno. Kontrakt mezi klientem a vývojářem.

**Funkční požadavky** – Výroky, co by systém měl obsahovat, jak by se měl chovat v učitých situacích. Např. Uživatel požaduje, aby se každý den generovala nová databáze, aby každý zaměstnanec používal při loginu své osmimístné ID, uživatel má možnost prohlédnout si historii půjček

**Nefunkční požadavky** – Vlastnosti a omezení služeb, jenž jsou systémem nabízeny. Jako časová náročnost, spolehlivost a bezpečnostní omezení. Často se vztahují na systém jako celek než na individuální části.

* Vytvoření jednoho nefukčního požadavku může zapříčinit vznik více funkčních požadavků
* Nefunkční požadavky mohou změnit celou strukturu systému nežli jednotlivé komponenty
* Typické nároky kladené na nefukční požadavky :
  + Rychlost (počet transakcí za sekundu)
  + Velikost (počet MB na soubor)
  + Jednoduchost na použití (doba výcviku zaměstnanců)
  + Spolehlivost (pravděpodobnost nespolohelivosti)
  + Robustnost (čas, po kterém se restartuje)
  + Portabilita (Počet cílových systémů)

**Dokument softwarových požadavků –** Měl by ukazovat, co má systém dělat, ne jak by to měl dělat. Měl by obsahovat definici všech typů požadavků.

* Předmluva (popisek dokumentu, jeho verze a historie)
* Úvod (popis potřeby pro systém)
* Slovníček
* Definice uživatelských požadavků (zde se popisují požadavky, poskytnuté zákazníkem a také nefunkční požadavky)
* Architektura systému
* Specifikace systémových požadavků (popis funkčních a nefunkčních požadavkl do větších detailů)
* Systémové modely (většinou grafický systém znázorňující vztahy mezi modely)
* Evoluce systému (možné budoucí změny podle zákazníka či marketingu)
* Dodatky (například popisy databáze a harwaru)
* Rejstřík

**Způsoby zápisu dokumentace**

* Přirozený jazyk (většinou chybí jasnost textu, funkční a nefunkční požadavky bývají pomíchané, požadavky mohou být špatně sloučené)
* Strukturovaný přirozený jazyk (jednotlivé body jsou od sebe odděleny a popisují každý svou sekci, hodí se pro embeded systémy, často moc hrubý pro business systémy)
* Jazyk pro popis designu (podobně jako programovací jazyk, ale více abstraktní)
* Grafická notace (Use case diagramy)
* Matematické specifikace (Množinové zápisy, mohou snížit nejednoznačnost, ale těžko se v nich orientuje)

**Proces inženýrství požadavků**

* Vyvolání a analýza požadavků
  + Nalezení požadavků ->
  + Klasifikace a organizace požadavků ->
  + Prioritizace a organizace požadavků ->
    - MoSCoW kritéria
      * Must Have
      * Should have
      * Could have
      * Want to have
    - RUP atributy
      * Status
      * Benefit
      * Effort
      * Risk
      * Stability
      * Target release
  + Specifikace požadavků ->
* Validace požadavků
  + Zjišťujeme, zdali požadavek opravdu definuje to, co uživatel chce. Opravování chyby požadavků je 100x horší než opravování chyby v implementaci.
  + Kontrolujeme:
    - Konzistenci (jsou zde nějaké konfliktní požadavky?)
    - Ucelenost (jsou v požadavku zabudovány všechny fce?)
    - Realizaci (dá se požadavek aplikovat při daném rozpočtu?)
    - Ověřovatelnost (Můžeme požadavek zkontrolovat?)
* Management požadavků
  + Správa požadavků, které se mění během tvorby požadavků

**Stakeholder –** Zainteresovaná osoba (Např. Ve zdravotnictví Sestra, používajíce databázi, pacient jehož informace jsou založeny do systému, Doktoři, kteří jsou zodpovědní za pacienty)

Problémy – Stakeholdeři neví, co chtějí. Jejich požadavky se často mění během specifikace. Mohou mít konfliktní požadavky.

* Důležitým způsobem řešení problémů jsou rozhovory.
  + Uzavřené (předem připravené otázky na stakeholdery)
  + Otevřené (společně ve skupině se stakeholdery se naleznou nové požadavky)
* Demonstrujeme-li požadavky, pomáháme si tak ujasnit, co vlastně uživatelé chtějí

## Přednáška 3 – Nefunkční požadavky

Klasifikace nefunkčních požadavků:

**Požadavky produktu –** požadavky určující kvalitu s jakou bude produkt fungovat, jako rychlost, spolehlivost etc.

**Organizační požadavky** – organoizační procedury a omezení – standardy procesu, implementační požadavky eg.

**Externí požadavky** – faktory, které jsou systému externí, například interoperabilita, požadavky legislativy eg.

**Dostupnost –** Pravděpodobnost, že systém bude v dané časové době funkční a schopen podat pořadované služby.

* Jak dlouho by měl systém fungovat bez selhání
* Jak nejdéle by měl být mimo provoz

**Spolehlivost –** Pravděpodobnost, že systém bude po určitou časovou dobu fungovat bez pádu, za určitou časovou dobu za určitých podmínek

* Jak je detekována systémová chyba
* Jak často může chyba nastat
* Co se bude dít, až chyba nastane
* Jednalo- li se o systémovou chybu, chybu uživatele, selhhání systému

**Bezpečí –** vlastnost, že systém může fungovat bez ohrožení, které by se odráželo na jeho schopnosti fungovat nebo aby nezpůsobil žádnou škodu lidskému životu, či prostředí systému.

* Nehoda, hazard, poškození, risk

**Bezpečnost** – vlastnost odrážející schopnost systému bránit se proti externím útokům

**Výkon –** je o časování, odpověď na určité události (přerušení, zprávy, požadavky od uživatelů) Citlivé na hardware, počet připojených uživatelů, operační systém, implementaci.

* Pro finanční systém například (kolik transakcí se udělá za vteřinu)

**Možnost modifikace –** cena za změnu

* Co se může změnit? Platforma, na které systém běží, prostředí, ve kterém běží. Tato možnost by měla být obsáhnuta při instalaci

**Možnost testování** - Možnost, v jakých podmínkách je schopen systém demonstrovat své vlastnosti v testování. (Až 40% období vývoje systému zahrnuje testování)

* Ukazuje se, jak efektivní jsou testy na určité úrovni
* Aby byl systém správně testovatelný, musí být možné kontrolovat vstup a výstup každé komponenty

**Použitelnost –** se zabývá tím, jak jednoduchý je software k použití pro uživatele

* Nauka systémových vlastností
* Efektivně používat systém
* Minimalnizovat dopad chyb
* Adaptovat systém uživatelským potřebám
* Zvýšení důvěry a spokojenosti

**Ogranizační požadavky**

* Vývojové požadavky (programovací jazyk, vývojové prostředí)
* Operační požadavky (Platforma, na které systém poběží, životnost)
* Požadavky prostředí (integrace s legacy systems)

**Time to Market –** Pokud je vývoj či vydání systému ohroženo konkurencí, je doba vývoje důležitá

* Často se používají starší již použité elementy nebo COTS

## Přednáška 4 – Analýza a Design

**Softwarová analýza** je kreativní aktivita, ve které se identifikují softwarové procesy, entity a jejichz vztahy

**Softwarový design** vylepšuje analytické modely implementačními detaily

**Implementace** je proces realizace designu jako programu

**Build or buy - Aneb proč programovat něco, co už existuje. Chcete-li například programovat systém dávkování léků pacientům, dá se zkoupit COTS verze. S psaním vlastní by bylo zbytečně mnoho komplikací.**

**Běžné aktivity při procesu tovrby**

* **Definovat kontext a způsob užití systému**
* **Načrtnout systémovou architekturu**
* **Indentifikovat hlavní systémové procesy a entity**
* **Vytvořit model designu**
* **Specifikovat rozhraní komponent a objektů**
* **Finalizovat systémovou architekturu**

**Architektura – abstrakce**

* **Architektura v malém – se zabývá architekturou jednotlivých individuálních programů. Rozdělení programu do komponent**
* **Architektura ve velkém - Architektura komplexních systémů, jenž zahrnují systémy a programy.**

**Výhody explicitní architektury**

* **Komunikace se stakeholderama**
* **Systémová analýza (zdali je možné implementovat určité nefunkční požadavky)**
* **Vysokoškálové znovupoužití**

**Modely designu - vylepšují analytické modely o informace potřebné ke komunikaci a záměrnou implementaci v systému**

**Pohledy na softwarové systémy**

* **Funkčně orientovaný pohled (systém jako množina fukncí, které spolu komunikují)**
* **Datové orientovaný pohled (Hledá základní struktury dat v systému, Funkční aspekt systému je méně významný)**
* **Objektově orientovaný model (systém jako možina objektů, které spolu komunikují zahrnující jak data, tak operace daty prováděné)**

**Strukturované metody**

* **DeMarco: Structured Analysis and System Specification (SASS)**
* **Gane-Sarson: Logical Modelling (LM)**
* **Yourdon: Modern Structured Analysis (YMSA)**
* **Structured Systems Analysis and Design Method(SSADM)**

**Ukázková metoda (Gane-Sarson)**

**1. Definujme kontext systému a vytvořme původní systém DFD.**

**2. Načrtnout původní datový model (ERD).**

**3. Analyzujme datové entity a vztahy do finálního ERD.**

**4. Vylepšeme DFD podle ERD datového modelu .**

**5. Rozdělíme logický model procesu do proceduálních elementů**

**6. Specifikujeme detaily každého individuálnígo procedruálního elementu**

**Objektově orientované metody**

* **Jim Rumbaugh: Object Modelling Technique (OMT)**
* **Coad-Yourdon: Method for Object-Oriented Analysis (OOA)**
* **Jacobson: Object-Oriented Software Engineering (OOSE)**
* **Kruchten et al.: Rational Unified Process (RUP)**
* **Booch-Jacobson-Rumbaugh: Unified Process (UP)**

**Objekty se skládají z dat a funkcí, jenž jsou spolu svázány do balíčku. Každý objekt je instance nějaké třídy, která deklaruje jeho základní vlastnosti (atributy a operace) sdílené se všemi svými instancemi**

* **Identita (každý objekt je unikátní)**
* **Status (data uložená v objektu)**
* **Chování (Množina operací, které může objekt dělat)**

**Třídy nám pomáhají modelovat množiny objektů se stejnými vlastnostmi**

## Přednáška 5 – Objektově orientovaná analýza

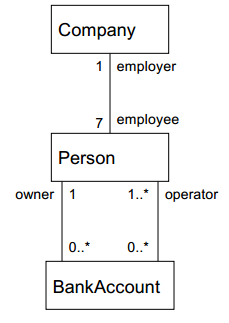
**Vztah** je spojení mezi modelovanými elementy

**Link** je spojení mezi objekty (způsob, kterým objekty komunikují)

**Asociace** je vztah mezi třídami (Asociace mezi třídami indikují, že zde existují linky mezi objektami těchto tříd.

* Syntaxe mezi třídami (například zaměstnanec ->zaměstanvatel nebo zaměstnává nikdy ne oboje)

**Multiplicita**

0..1 zero or 1

1 exactly 1

0..\* zero or more

\* zero or more

1..\* 1 or more

1..6 1 to 6

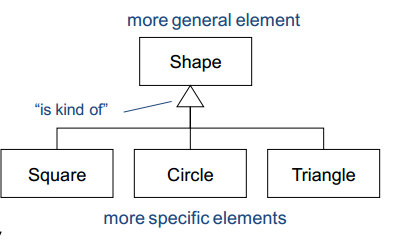
**Kvalifikátory**

**Například 1 klub může mít \* členů. Přídám však před klub kvalifikátor ClenID, zredukuji si tak vztah na 1 k 0..1**

**Závislosti**

* **Využití** 
  + **<<Use>> – Klient využije poskytovatele aby implementoval jeho chování**
  + **Call – operace klienta vyvolá operaci poskytovatele**
  + **Send – klient(operace) pošle poskytovateli (signál )na nějaký nespecifikovaný cíl**
  + **Klient je instancí poskytovatele**
* **Abstrakce (<<trace,substitute,refine,derive>>)**
* **Přístup (<access,import,permit>>)**

**Generalizace je Vztah mezi více obecným elementem a více specifickým elementem**



**Dědičnost – podtřídy dědí všechny vlastnosti své rodičovské třídy**

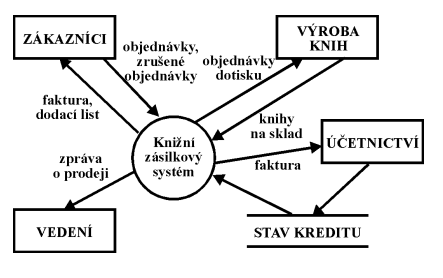
**Překrytí - někdy může podtřída překrýt vlastnosti rodičovské třídy**

**Polymorfismus definuje, že dvě metody se stejným názvem můžou mít různé implementace a ta implementace se může dynamicky vybírat během běhu systému podle toho, který dílčí typ se dostal do proměnné.**

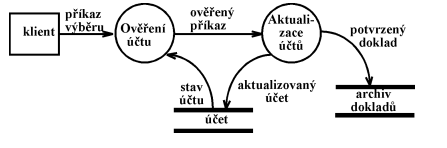
## Přednáška 6 – Strukturovaná analýza

**Yourdon Modern Structured Analysis (YMSA)**

* **Enviromentální model**
  + **Kontextový diagram je speciální případ diagramu toku dat, obsahující jeden proces, který představuje celý systém**
  + **Seznam událostí je textový seznam impulzů pocházejících z prostředí, na které musí systém reagovat**

****

* **Model chování specifikuje tok dat skrze systém, modelujíce aspekty jeho procesů**
  + **Ukazuje, jaké informace budou na vstupu a výstupu ze systému, odkud a kam přijdou data a kde budou uloženy**
  + **Neukazuje informace o časování procesoru**
  + **Data flow Diagram (DFD)**
    - **Je grafická reprezentace systému jako sítě procesů, které plní funkce systému a komunikují skrze systémová data. Skládá se z :**
      * **Procesů**
      * **Datových toků**
      * **Úložitští dat**
      * **Terminátorů (externí entita, která komunikuje se systémem)**



**Zbytek přednášky se zaměřuje na ERD diagram**

**ERD vs. UML Class diagram**

**Class diagram**

* **Modeluje strukturové vlastnosti i vlastnosti chování systému**
* **Obsahuje více druhů spojení (agregace, asociace, kompozice)**
* **Lépe se s ním mapují obejkty z reálného světa**

**ERD**

* **Modeluje pouze strukturové vlastnosti a má malý počet druhů spojení**
* **Lépe se snimi mapují databázové tabulky**

**I když může existovat jen jedno mapování mezi ERD a Class, je velmi běžné, že**

* **Jedna třída je mapována na více než jednu entitu**
* **Víc tříd se mapuje na jednu entitu**

**ERD je datově orientované a vytrvalostně specifické**

**Class diagram cílí také na operace a je nezávislý na vytrvalosti**

## Přednáška 7 – Design

Model designu je vylepšení analytického modelu na takový stupeň, že může být implementován

* V MDD (Model driven development) obsahují modely designu všechny implementační detaily a mohou být automaticky přeloženy v kódu
* V OO (object oriented) jsou všechny atributy specifikovány včetně typu, viditelnosti, a původních vlastností

**Analytický model vs. Model designu**

Designový model může obsahovat až 100 krát více tříd než analytický model

* Analytický model nám pomáhá vidět „obrázek“ modelu bez toho, abychom se v něm ztratili
* Pokud tvoříme velký systém, musíme implementovat oba modely, pokud malý nebo se jedná o systém s nízkou životností či významem, stačí nám model analytický

**Designové schéma** je způsob použití abstraktních vědomostí o problému a jeho řešení

* Měl by mít vlastní jméno, popis problému a popis řešení a následků
* **Pozorovací schéma** odděluje zobrazení objektu od objektu samotného

**Různorodost a redundance(nadbytečný) spolehlivosti**

* Redundance – Ponechat si jednu verzi komponenty jako zálohu, pokud jedna spadne, nahodí se druhá náhradní
* Různorodost – poskytnutí stejné funkcionality více způsoby, tak aby nedošlo k jejich současnému selhání

**Voting** procesy běžící na redundatních procesorech vezmou ekvivaltení vstup a jejich výstupy se následně porovnají (poslány „voliči“)

**Aktivní redundance – všechny redundantní komponenty reagují na události paralérně**

**Pasivní redundance – Jedna primární komponenta reaguje na události a informuje ostatní komponenty  
Spare – náhradní platforma je odstavena a je konfigurována k nahrazení komponent, které selhaly**

**Shadow operation – Komponenta, která nedávno selhala, běží chvíli ve stínovém módu, aby se zajistilo, že již nedojde k selhání**

**Checkpoint/rollback – bod obnovy**

**Strategie přežití**

* **Odolnost – Vyhýbání se úkotokům tak, že se do systému zabudují komponenty, jenž zvýší odolnost vůči útokům**
* **Rozpoznání – detekování problémů**
* **Obnova**

**Taktiky výkonu – prostředkový management**

* **Zajistit více kopií (dat nebo výpočtů)**
* **Zvýšit počet prostředků, které jsou k dispozici (rychlejší procesory, více paměti)**

**Taktiky měnitelnosti**

* **Runtime registration (podporuje plug and play operaci)**
* **Konfigurační soubory (nastaví parametry při startu)**
* **Polymorfismus**
* **Nahrazování komponent**
* **Record/Playback (Informace, jenž jde z interfacu je uložena do externího repozitáře)**
* **Oddělit rozhraní od implementace**
* **Sepcializace přístupu cest/rozhraní**

**Agregace – Některé obejkty jsou slabě společné, asi jako počítač a tiskárna (tiskárna může existovat bez počítače)**

**Kompozice – Některé objekty jsou spojené silně, jako list a strom (list bez stromu existovat nemůže)**

## Přednáška 8 – Design Architektury a Implementace

**The model view controller** (MVC) – Odděluje prezetnaci a interakci od systémových dat. Systém je rozdělen do tří komponent, které spolu navzájem komunikují.

* Komponenta Model spravuje systémová data a operace asociované těmto datům
* Komponenta View definuje, jak jsou data ukázána uživateli
* Komponenta Controller spravuje uživatelskou interakci
* Výhodou je, že se data mohou měnit nezávisle na své reprezentaci.
* Nevýhodou je komplexní kód

**Schéma vrstvené architektury** (layered architecture pattern) – Rozděluje systém do vrstev. Vrstva poskytuje.

* Uživatelské rozhraní
* Management uživatleského rozhraní (autentifikace, autorizace)
* Core business logic/ funkcionalita aplikací, systémové utility
* Systémová podpora (OS, databáze...)
* Výhodou je, že celé vrstvy mohou být nahrazeny, dokud se systém spravuje
* Nevýhodou je, že v praxi je těžké zavést čistou separaci mezi vrstvami.

**Schéma architektury repozitáře**

* **Buď jsou data uložená v centrální databázi (repozitáři)**
* **Každý subsystém si spravuje vlastní databázi**
* **Výhodou je, že komponenty jsou nezávislé a nemusí vědět o ostatních komponentách**
* **Nevýhodou je, že problémy ve sdíleném repozitáři můžou ovlivnit celý systém**

**Schéma Klient – Server**

* **Funkcionalita systému je rozdělena do jednotlivých služeb a každý server poskytuje jinou službu**
* **Klienti jsou uživateli ěchto služeb a přistupují na server, aby s nimi mohli pracovat**
* **Hlavní výhodou je, že servery mohou být distribuovány přes síť. Jedna fukncionalita může být poskytována všem klientům přes jeden server**
* **Nevýhodou je, že každá služba je náchylná k selhání serveru. Výkon je limitován rychlostí sítě.**

**Schéma pipe and filter**

* **Zpracování dat v systému je organizováno tak, že každá komponenta (filtr) je odlišná a nese vlastní transformace dat, které pak tečou (jako trubkou) k dalšímu zpracování**
* **Výhodou je jedoduchost na pochopení a podpora znovu využití transformace**
* **Nevýhodou je, že formát transformovaných dat musí souhlasit mezi transformacemi, které spolu komunikují**

**Implementační záležitosti**

* **Znovupoužití(Reuse)**
* **Konfigurační management –Během výrobního procesu je důležité sledovat vývoj každé komponenty**
* **Host-target development – systém se většinou spouští na jiném počítači, než na kterém je tvořen, proto je důležité brát ohledy na jeho prostředí, ve kterém pak poběží**

## Přednáška 9 – Uživatelské rozhraní

**Human-computer interaction (HCI) – je studie o tom, jak lidé komunikují s počítačem. Zahrnuje v sobě jak umění, tak vědu**

* **Design uživatelského rozhraní se zaměřuje na systémový design se zaměřením na uživatelské zkušenosti a interakci**

**Proč by vývojáři neměli navrhovat uživatelské rozhraní?**

* **Měli by se soustředit více na niterní záležitosti systému, než na jeho využívání**
* **Vývojáři uvažují jinak než uživatelé**

**WIMP Paradigm –Okna, ikony, menu a ukazovátko (myš)**

**Užitečnost – jednoduché, aby byl uživatel spokojen**

**Uživateslká zkušenost**

**Look and feel – uvádí uživatelské zkušenosti a identifikaci produktu**

**Human interface guidelines –uživatelé si mohou přenášet znalosti s operací na jednom systému na systém další**

**Fittův zákon – Model lidského pohybu předpovídá čas, který je potřeba k trefení cíle**

* **Fyzicky s rukou nebo prstem**
* **Virutálně s ukazovátkem**

**Hickův zákon – předpovídá čas, vynaložený na výběru jedné položky ze seznamu**

**Human-Interface Guidelines (HIG) – popisuje především správné používání komponentů (vzdálenost mezi čudlíky)**

**Zbytek deváte přednášky tu nebudu vypisovat, fakt jsou to základní věci, pro které si stačí projet slidy.**

## Přednáška 10 – Testování

**Validační testování** má ukázat vývojáři a zákazníkovi, že software splňuje své požadavky

**Defektní testování** má ukázat, ve kterých situacích se program chová špatně nebo nesplňuje očekávání své specifikace

* Tyto testy jsou navrhnuty tak, aby ukázaly defekty systému. Můžou být až obskutní, a neodrážet způsob, jakým je normálně software používán
* Úspěšný test ukazuje, že systém funguje špatně a tak ukazuje na defekt v systému

**Vertifikace** – Stavíme software správně

* Software funguje, tak jak bylo navrhnuto

**Validace** – Stavíme správný software

* Software by měl dělat opravdu to, co uživatel chce

**Softwarové inspekce(statická vertifikace) – zahrnuje lidi, kteří prohlédnou zdrojovou reprezentaci se zaměřením na nalezení anomálií a defektů. Inspekce nepotřebují spuštění systému a tak mohou být použity ještě před implemetnací**

* **Výhody inspekcí – Během testování mohou chyby skrýt jiné chyby, ale jelikžo je inspekce statický proces, žádná interakce mezi chybama se konat nebude**
* **Mohou být prohléduty i nekompletní verze systému**

**Softwarové testování (dynamická vertifikace) – systém je spuštěn společně s testovacími daty a jeho chování je zaznamenáváno a kontrolováno**

**Statická analýza – jsou to techniky, které vertifikují systém a nevyžadují spuštění programu.**

* **Mají svou hodnotu, kdykoliv je levnější najít a odstranit chyny než platit za selhání systému**

**Formální metody mohou detekovat implementační chybu ještě před testováním, kdy je prográm analyzován společně se svou specifikací**

* **Formální metody však vyžadují specializované notace, kterým doménoví experti nemusí rozumět. Jsou také velmi drahé. A stejnou úroveň důvěryhodnosti lze dosáhnout levněji použitím VaV technik (Vertifikace a Validace)**

**Model Checking – zahrnuje vytvoření rozšířeného modelu systému a využívání specializovaného systému (model checker) k hledání chyb v modelu.**

* **Model checking se hodí k vertifikaci souběžných systémů a i když je velmi drahý, je dobré ho použít v malých a středně velkých systémech**

**Stupně statické analýzy**

* **Charakteristická kontrola chyb – kontrola chyb, které například dělají programátoři, jenž používají specifický programovací jazyk (chybějějící středníky)**
* **Uživatelsky definovaná kontrola chyb – chyby, které se mají hledat jsou nadeklarovány samotnými uživately**
* **Kontrola tvrzení – vývojáři zabudují do programu formální tvrzení. Statický analyzér pak symbolicky spustí kód a zvýrazní potencionální problémy.**
* **Vhodné využití, například, když se používá jazyk C, spousta chyb typických se dá detekovat**

**Test-driven development (TDD) – je přístup k vývoji programu, ve kterém jsou mezi sebou testování a vývoj kódu mezilehlé**

* **Testy jsou psané ještě před kódem a „projití“ testem je kritické**
* **Kód se vyvjí inkrementálně společně s testy.**

**Výhody TDD**

* **Informace o kódu – každý kousek kódmu má asociován alespoň jeden test**
* **Regresní testování**
* **Zjednodušené debugování – když selže test, mělo by být jasné, kde je chyba**
* **Systémová dokumentace – Samotné testy jsou formou dokumentace, která popisuje, co by kod měl dělat**

**Regresní testování – zkouší, jestli změny v systému neporušily předochí kód**

**Release testing - důležitým rozdílem je, že na testování se podílí skupina lidí, kteří nebyli přítomni u vývoje. Měli by se zaměřit na hledání bugů (defective testing)**

## Přednáška 11 – Evoluce

**Změna softwaru je nevyhnutelná –** nové požadavky se objevují jak je software používán, mění se také prostředí a musí být opraveny chyby. Klíčem ke všem organizacím je implementace a správa změn do již existujících softwarových systémů.

**Proces evoluce záleží na:**

* Typu softwaru, který se spravuje
* Typu vývojových procesů
* Umění a zkušenostech lidí, jenž na ní pracují

**Implementace změny**

* Iterace vývojového procesu opatřena o vylepšení
* Kritický rozdíl je, že první fáze implementace změny vyžaduje znalost softwaru

Rychlé metody jsou založeny na inkrementálním vývoji, tak, že přechod z vývoje do evoluce je souvislý

**Program evolution dynamics** je studie procesů systémových změn

**Lehmanovy zákony**

* Průběžná změna – program, který se používá v reálném světě se musí nutně měnit jinak se stane zbytečným
* Zvýšená komplexita – jak se evolvující program mění, jeho struktura má tendenci stát se více komplexnější. Struktura se tedy musí zjednodušovat
* Velká programová evoluce – evoluce programu je seberegulační proces. Systémové atributy jako velikost, čas mezi vypuštěním verzí a podobně je větišnou invariantní
* Organizační stabilita – během života programu je těžké zachovat konstatní aktualizace
* **Zachování podobnosti – každá změna v každé verzi by měla být konstantní**
* **Kontinuální růst**
* **Klesající kvalita - kvalita systémů bude klesat jestliže jsou modifikovány tak, aby odrážely změny v jejich operativním prostředí**
* **Feedback system**

**Typy údržby**

* **Oprava sowftwarových chyb**
* **Adaptace softwaru na jiné operační prostředí**
* **Přidat nebo měnit systémovou funkcionalitu**

**Faktory ceny údržby**

* **Stabilita týmu**
* **Smluvní zodpovědnost**
* **Um zaměstanců**
* **Věk a struktura programu**

**Systémový re-engineering**

* **Znovu napsání celé části systému bez účelu změnit její funkcionalitu**
* **Aplikovatelné, když některé ale ne všechny subsystémy vyššího systému vyžadují častou údržbu**
* **Výhodou je snížený risk a cena**
* **Překlad zdrojového kódu**
* **Reverzní engineering**
* **Vylepšení struktury programu**
* **Modularizace programu**
* **Re-engineering dat**

**Reengineering se praktikuje až potom, co byl systém nějakou dobu udržován a jeho cena za údržbu stoupá. K reingeneeringu se používají automatitozovaná zařízení, aby se vytvořil nový systém, který by se lépe udržoval.**

**Refactoring je kontinuální proces vylepšení skrze vývojový a evoluční proces. Jeho záměrem je vyhnout se degradaci struktury a kódu, která zvyšuje cenu za údržbu.**

**„Bad smells“**

* **Duplikovaný kód**
* **Dlouhé metody**
* **Switch (case) statements**
* **Data clumping (duplikace dat)**
* **Spekulativní obecnost (kód, který je v programu obsažen, aby sloužil někdy do budoucna)**

# Přednáška 12 – Process management

**Vývoj prototypů**

* Může zahrnovat vynechání funkcionality a celkovou nízkou kvalitu
* Prototypy se soustředí na části produktu, které nejsou dobře pochopeny

**Inkremenrální doručení** – než aby byl systém doručen jako celek, je vývoj a doručení rozděleno do inkrementů. S každým inkrementem se doručí část funkcionality. Výhody:

* Systémová funkcionalita může být doručena dříve
* Brzké inkrementy se chovají jako prototypy aby vyvolaly požadavky pro pozdější inkrementy
* Menší risk celkového selhaní projektu

**Nevýhody:**

* Většina systémů vyžaduje množinu základních zařízení, které jsou využívány jinými částmi systému
* Princip iterativních procesů je, specifikace je vyvíjena společně se softwarem

**Boehmův spirálový model**

* Proces je reprezentován jako spirála, ne jako sekvence aktivit
* Každá smyčka spirály reprezentuje fázi procesu
* Spirálový model pomáhá lidem zamyslet se na iterací v softwarovém procesu
* V praxi nicméně se model zřídka kdy používá pro praktický vývoj softwaru

**Fáze RUP**

* Počátek
* Rozpracování
* Konstrukce
* Přechod (Transition)

**Agilní metody**

* zaměřují více na kód než design
* Jsou založeny na iterativním přístupu k vývoji softwaru
* Jejich záměrem je poskytnout fungující software rychle a také jej rychle vyvinout, aby splňoval své požadavky

**Plan-driven development**

* Je založeno na oddělených fázích vývoje. Jedna fáze poskytuje produkt pro vývoj další fáze.
* Nejedná se nutně o waterfall model
* Iterace se odehrává v aktivitách

**Extrémní programování**

* Asi nejznámější agilní metoda
* XP je extrémní přístup k iterativnímu vývoji
* Nové verze vznikají i několikrát denně
* Inkrementy jsou zákazníkovi posílány každé dva týdny
* Všechny testy musí proběhnout a další progres je povolen pouze, když testy projdou

Project management jsem vynechal, DUH

# Přednáška 13 – pokročilé techniky

**Object constraint language (OCL**) – je deklarativní jazyk pro popis pravidel vzahujících se na UML modely. Jeho výroky se vztahují na 4 části:

* Context, který definuje limitované situace, ve kterých je výrok validní
* Property (vlastnost) která reprezentuje charakteristiku kontextu
* Operation, která manipuluje nebo kvalifikuje vlastnosti
* Keywords kterými se specifikují kondiční výrazy

**UML profily**

* Uml profily poskytují mechanismus generických nástaveb k upravení UML modelů na určité domény nebo platformy.
* Profily jsou definovány použitím stereotypů, definic tagů a omezení, které se aplikují na specifické moely elementů jako třídy, atribury a aktivity
* Profil je kolekce takovýchto nástaveb, která kolektivně upravuje UM“ pro určitou doménu nebo platformu

Zbytek třinácté přednášky tvoří opakování. Doporučuji si ji stáhnout z video.muni.cz a všechno si ještě jednou projet. Good luck mateys!