České vysoké učení technické v Praze
Fakulta elektrotechnická

Tématické okruhy ke státní závěrečné zkoušce pro magisterský studijní program Otevřená Informatika (OI)

http://www.fel.cvut.cz/cz/education/master/topicsOI.html

OI Mgr - SI

Vygenerováno: 31. května 2015 20:40

Obsah

1 TPJ - sémantiky: operační, denotační **1**

1.1 Sémantika malého kroku (SOS) . . . 1
1.2 Sémantika velkého kroku (BOS) . . . 2
1.3 Denotační sémantika . . . 3
1.3.1 Významová funkce (Meaning function) . . . 3
1.4 Pevný bod funkce . . . 4
1.5 Vázání jmen . . . 4
1.6 Stav programu . . . 5
1.7 Data programu . . . 6

2 TPJ - Statická sémantika **7**

2.1 Typy . . . 7
2.2 Polymorfní typy . . . 8
2.3 Typy vyššího řádu . . . 8
2.4 Rekonstrukce typů . . . 9
2.5 Abstraktní typy . . . 9

3 NUR - Teorie HCI **10**

3.1 HCI - Human-Computer Interaction . . . 10
3.2 Kognitivní aspekty . . . 10
3.3 Způsoby interakce . . . 11
3.4 Speciální UI . . . 11

4 NUR - Metody návrhu, uživatelské a konceptuální modely **13**

4.1 Specifikace požadavků . . . 14
4.2 Uživatelský průzkum . . . 14
4.3 Modely pro návrh UI . . . 15

5 NUR - Formální popis uživatelských rozhraní **16**

6 NUR - Prototypování uživatelských rozhraní **19**

6.1 Low-Fidelity . . . 19
6.2 High-Fidelity . . . 20

7 OSP - Open source,git,lincence **21**
7.1 Nástroje pro správu verzí kódu . . . 21
7.2 Nástroje pro sledování chyb (bug trackers) . . . 21
7.3 Nástroje pro automatické generování dokumentace . . . 21
7.4 Systémy pro spolupráci mezi vývojáři . . . 21
7.5 Licence . . . 22
7.5.1 BSD (Berkeley Software Distribution) . . . 22
7.5.2 MIT License . . . 22
7.5.3 Apache License . . . 23
7.5.4 GNU GPL (GNU General Public License) . . . 23
7.5.5 GNU LGPL (GNU Lesser General Public License) . . . 23
7.5.6 MPL (Mozilla Public License) . . . 23
7.5.7 CreativeCommons . . . 23
7.6 Kontribuce do projektu . . . 24

8 OSP - Přenositelnost, multiplatformnost **25**
8.1 Požadavky a pravidla pro tvorbu přenositelného kódu . . . 25
8.2 Organizace projektů a struktura operačních systémů pro zajištění přenositelnosti mezi různými platformami (OS, CPU) . . . 25
8.3 Kompilace GNU balíků . . . 26
8.4 Portace kódu a křížový překlad . . . 26
8.5 Konverze vnitřních a vnějších/síťových formátů dat . . . 26

9 AOS - SOA **28**
9.1 Architektura zaměřená na služby (SOA) . . . 28
9.1.1 Webová služba . . . 28
9.2 Konceptuální model . . . 29

10 AOS - WS **31**
10.1 Webové služby . . . 31
10.2 Vyhledávání služeb - UDDI registr . . . 31
10.3 Technologie a protokoly . . . 32
10.4 Kódování obsahu . . . 35
10.5 Bottom-up design . . . 35
10.6 Top-down design . . . 35

11 AOS - Kompozice služeb **36**
11.1 Orchestrace . . . 36
11.2 Choreografie . . . 36
11.3 Mashup . . . 36
11.4 Modelovací jazyky BPMN, BPEL . . . 37

12 AOS - Kvalita, bezpečnost **38**
12.1 Bezpečnost . . . 39

13 TVS - Chyby,optimalizace testů,test. automatů **40**
13.1 Kategorie SW chyb . . . 40
13.2 Optimalizace návrhů testů . . . 45
13.2.1 Princip párového testování . . . 45
13.2.2 Optimalizace metodou ortogonálních polí . . . 45
13.2.3 Optimalizace metodou latinských čtverců . . . 46
13.3 Testování automatů . . . 47

14 TVS **48**
14.1 White/black box testování . . . 48
14.2 Strukturální analýza . . . 48
14.3 Statická a dynamická analýza . . . 49
14.4 Analýza toků řízení (metoda hlavních cest) . . . 49
14.5 Analýza datových toků (metoda du-cest) . . . 49
14.6 Testování OO softwaru . . . 50

15 TVS - Formální specifikace programu, model checking **52**
15.1 Model-checking . . . 52
15.1.1 UPPAAL . . . 53
15.1.2 Temporální logika . . . 53

16 WA2 - Java EE **55**
16.1 JAVA EE . . . 55
16.2 Servlet . . . 56
16.3 Java Bean . . . 57
16.3.1 Inversion of Control (IoC) . . . 57

17 WA2 - Web. architektury, perzistence, WS, messaging **59**
17.1 Sdílení dat . . . 59
17.2 persistence . . . 59
17.3 WS a REST . . . 60
17.4 Messaging . . . 61

19 WA2 - Cloud **62**
19.1 Infrastructure as a Service (IaaS) 62
19.2 Platform as a Service (PaaS) 62
19.3 Software as a Service (SaaS) 63
19.4 Omezení cloudu 63
19.5 Náklady na provoz 63
19.6 Virtualizace 63
19.7 Vhodné aplikace pro cloud 64

1 Sémantika: operační sémantika, denotační sémantika, pevný bod funkce, vázání jmen, stav programu a data.

Sémantika programovacích jazyků je v teorii programovacích jazyků **obor** zabývající se **důsledným matematickým popisem významu** programovacího jazyka [2].
3 hlavní charakteristiky jazyka jsou:

- **syntaxe** se zabývá formou (znaky a jejich vztahy),
- **sémantika** významem znaků,
- **pragmatika** závislostí konstrukcí na nositeli - implementace.

Operační sémantika je **přístup**, který **definuje sémantiku** programovacího jazyka tak, že určí jak je libovolný program vykonán na počítači, jehož činnost je známa. Může to být nějaký abstraktní stroj, který je dostatečně jednoduchý pro snadné pochopení jeho činnosti (například Turingův stroj). Operační sémantika potom specifikuje, jak tento stroj zpracovává program v definovaném jazyku.

Operační exekuce

- Program + vstupy → Vstupní funkce
- Iniciální konfigurace
- (Mezikonfigurace ...)
- Finalní konfigurace
- Výstupní funkce → Odpověď

1.1 Sémantika malého kroku (SOS)

- Definujeme přepisovací relaci $\Rightarrow \in Expr \times Expr$.
- Výraz $e \Rightarrow e'$ znamená, že přepis e na e' je vykonán v jednom kroku.

Formální definice $S = \langle CF, \Rightarrow, FC, IF, OF \rangle$

- CF – doména konfigurací (obor hodnot)
- $\Rightarrow$ – přepisovací relace (transformuje konfigurace) ($\Rightarrow \subseteq CF \times CF$)
- FC – množina finálních konfigurací (nezjednodušitelné konfigurace) ($FC \subseteq CF$)
- IF – vstupní funkce (Prog × Inputs) → CF.
- OF – výstupní funkce FC → Answer

Pravidla $e, e_1, e_2, e' \in Expr \quad n, n' \in Num$

$$\frac{}{\triangle n \Rightarrow -n}$$

$$\frac{}{n \odot n' \Rightarrow n + n'}$$

$$\frac{e \Rightarrow e'}{e\triangle \Rightarrow \triangle e'}$$

$$\frac{e_1 \Rightarrow e'}{e_1 \odot e_2 \Rightarrow e' \odot e_2}$$

$$\frac{e_2 \Rightarrow e'}{e_1 \odot e_2 \Rightarrow e_1 \odot e'}$$

1.2 Sémantika velkého kroku (BOS)

Odlišný přístup než SOS. Program je vyhodnocen v jednom kroku. Zde je definována přepisovací relace

$$\Longrightarrow \in Expr \times Num$$

Pravidla $e, e_1, e_2 \in Expr \quad n, n_1, n_2 \in Num$

$$\frac{}{n \Longrightarrow n}$$

$$\frac{e \Longrightarrow n}{\triangle e \Longrightarrow -n}$$

$$\frac{e_1 \Longrightarrow n_1 \quad e_2 \Longrightarrow n_2}{e_1 \odot e_2 \Longrightarrow n_1 + n_2}$$

► **Příklad SOS (jedno z možných vyhodnocení):**

$$\frac{\dfrac{\dfrac{}{\triangle 15 \Rightarrow -15}}{(\triangle 15) \odot (\triangle 24) \Rightarrow -15 \odot (\triangle 24)}}{\triangle((\triangle 15) \odot (\triangle 24)) \Rightarrow \triangle(-15 \odot (\triangle 24))}$$

► **Příklad BOS:**

$$\frac{\dfrac{\dfrac{\dfrac{}{15 \Longrightarrow 15 \quad 24 \Longrightarrow 24}}{\triangle 15 \Longrightarrow -15 \quad \triangle 24 \Longrightarrow -24}}{(\triangle 15) \odot (\triangle 24) \Longrightarrow -15 + -24}}{\triangle((\triangle 15) \odot (\triangle 24)) \Longrightarrow -(-15 + -24)}$$

1.3 Denotační sémantika

Denotační sémantika používá k popisu sémantiky programovacího jazyka funkce. Tyto funkce přiřazují sémantické hodnoty správným syntaktickým zápisům. Příkladem může být funkce Val, která přiřadí výrazu jeho hodnotu:

$$\text{Val} : \text{Výraz} \rightarrow \text{Integer}$$

Doménou syntaktických funkcí se nazývá syntaktická doména - u funkce Val je to množina všech správných aritmetických výrazů. Obor hodnot je sémantická doména - zde množina celých čísel. Denotační definice jazyka má tedy 3 části - syntaktickou doménu, sémantickou doménu a definici jednotlivých funkcí.

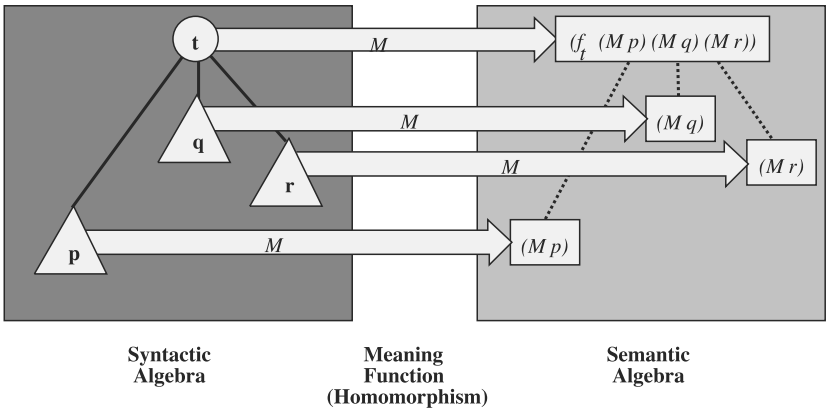
- **Syntaktická algebra** - popisuje abstraktní syntaxy jazyka, může být specifikována gramatikou.
- **Sémantická algebra** - modeluje význam frází, skládá se z kolekce sémantických domén.
- **Významová funkce** - mapuje elementy syntaktické algebry k jejich významu v sémantické algebře. Funkce musí být homomorfismus mezi syntaktickou a sémantickou algebrou.

1.3.1 Významová funkce (Meaning function)

Uvažujte M jako významovou funkci a t je prvek v abstraktním syntaktickém stromu s potomky $t_1, \ldots, t_k$. Pak

$$(Mt) = (f_t(Mt_1) \ldots (Mt_k))$$

kde f_t je funkce určená syntaktickou třídou.



Denotační sémantika $Expr ::= Num \quad | \quad \triangle\, Expr \quad | \quad Expr \odot Expr$
Sémantická doména N

$$[\![n]\!] = n$$

$$[\![\triangle e]\!] = [\![\triangle]\!]([\![e]\!])$$

$$[\![\triangle]\!] = \lambda x. -x \quad \text{(např. unární mínus)}$$

$$[\![e_1 \odot e_2]\!] = [\![\odot]\!]([\![e_1]\!], [\![e_2]\!])$$

$$[\![\odot]\!] = \lambda x, y. x + y \quad \text{(např. plus)}$$

1.4 Pevný bod funkce

Jako pevný bod označujeme bod, který se v daném zobrazení zobrazí sám na sebe. Označuje se také jako samodružný bod. Například pevnými body funkce $f(x) = x^2 - 4x + 6$, jsou čísla 2 a 3.

- bod, ve kterém platí $f(x) = x$
- využívá se pro rekurzivní funkce
- Y kombinátor v lambda kalkulu: `Y = λy(λx.y(xx))(λx.y(xx))`

Př: | Generující funkce faktoriálu fact = `λF.λX.if x==0 then 1 else F(decrement(X))`. Generující funkci dám do Y kombinátoru:

```
Y fact =
      = λf (λx.y(xx))(λx.y(xx)) fact =
      = (λx.fact(xx))(λx.fact(xx)) =
      = fact( (λx.fact(xx) (λx.fact(xx) ) =
      = fact(Y fact)
```

`Y fact` je pevný bod funkce, která počítá faktoriál.

1.5 Vázání jmen

Vázání jmen se vyskytuje v lambda kalkulu.

λ-kalkul Je formální popis, který slouží jako základ pro funkcionální jazyky, takže všechny konstrukce v těchto jazycích jdou přepsat právě na λ-kalkul. Základní prvky λ-kalkulu jsou tři následující:

- **Proměnné** Obyčejné proměnné, tak jak je znáte z jiných jazyků, většinou se značí `x,y,z`.
- **Abstrakce** Definice funkce – představte si např. funkci `f(x) = x+2`, tak přesně taková funkce se v λ-kalkulu zapíše takto: `λx.x+2`. Část mezi λ a tečkou jsou parametry funkce (zde máme pouze jeden parametr) a za tečkou se nachází tělo funkce.
- **Aplikace** Volání funkce – když si vezmu naši funkci `f(x) = x+2`, tak ta se zavolá např. s argumentem 3 takto: `f(3)`. Funkce v λ-kalkulu se volají podobně, funkce se volá takto: `(f 3)`, tzn. nejdříve je uvedena funkce a poté její argumenty. Funkce f se v λ-kalkulu tedy zavolá takto: `(λx.x+2) 3`. Vysvětlení by mělo být už jasnější. Číslo 3 se dosadí za parametr `x` a přejde se do těla funkce, tam se ke 3 přičte 2 a výsledkem je 5.

Proměnná je v λ-výrazu *vázaná*, pokud se jedná o parametr nějaké funkce, takže např. ve výrazu (`λx.yx`) je `x` vázaná proměnná. Ostatní proměnné (v minulém příkladu `y`) jsou *volné*. Proměnná se vždycky váže na nejbližší lambdu vlevo, takže ve výrazu `(λx((λx.x) w))` se proměnná `x` uprostřed výrazu váže na lambdu co je hned vlevo od ní a ne na tu úplně vlevo! Proměnná `w` je samozřejmě volná [3].

- Funkce, co mají jen vázané proměnné, vrátí při každém zavolání stejný výsledek. Funkce s volnými proměnnými jsou závislé na globálním kontextu.
- Funkce, co mají jen vázané proměnné, se v λ-kalkulu nazývají kombinátory.

`(λx.xy)`

`(λx.x)(λy.yx)`

1.6 Stav programu

Stav = proměnné v prostředí, proměnné mají typ, prostředí = množina všech proměnných, kontext (v handoutech Γ (Gamma)) Čistě funkční jazyky (a matematika) jsou bezestavové, stavové výpočty mohou být reprezentovány jako iterace skrz stavy.

Čistě funkční jazyky (a matematika) jsou bezestavové, stav může být modelován jako iterace skrz stavy.

► Funkce na nalezení maxima z pole:

$$max : N* \to N$$

$$max(\langle a_1, \ldots, a_n \rangle) = loop(\langle a_1, \ldots, a_n \rangle, 1, 0)$$

$$loop : N * \times N \times N \to N$$

$$loop(\langle a_1, \ldots, a_n \rangle, c, m) = m \qquad \text{if} \quad c > n$$

$$loop(\langle a_1, \ldots, a_n \rangle, c, m) = loop(\langle a_1, \ldots, a_n \rangle, c + 1, m) \qquad \text{if} \quad c \leq n \wedge a_c \leq m$$

$$loop(\langle a_1, \ldots, a_n \rangle, c, m) = loop(\langle a_1, \ldots, a_n \rangle, c + 1, a_c) \qquad \text{otherwise}$$

Monády - struktury (typy), co reprezentují výpočet jako sekvenci kroků.

1.7 Data programu

Dělí se na **součiny**, **sumy** a **sumy součinů**.

- Součiny
  - Positional data = N-tice, každý prvek může mít jiný typ
  - Sequence, List = pole
  - Named = třída
  - Nonstrict, stream = data, která se získají/vypočítají v okamžiku, kdy je potřeba (např. InputStream, odněkud se to vezme)
- Sumy - Union v C, nadtypy v Javě.
- Sumy součinů - Binární a ternální operátory, double dispatch.

2 Statická sémantika: typy, polymorfní typy, typy vyššího řádu, rekonstrukce (inference) typů, abstraktní typy.

Statická sémantika je řešena při překladu programu, zde jsou definovány a deklarovány jednotlivá pravidla a prvky programovacího jazyka. V těchto prvcích je zahrnuta jazyková konstrukce, její typy parametrů, význam příkazů a další prvky. Statická sémantika dále kontroluje statické typy a práci s tabulkou definovaných programových symbolů.

- Staticky typované jazyky požadují uvedení datového typu u každé deklarace. Zde nelze deklarovat proměnnou, či funkci nebo objekt bez zadání datového typu.
- Všechny typové kontroly jsou prováděny staticky při překladu. Už při překladu má být každé proměnné přiřazen datový typ.
- Je možné daný datový typ přímo přetypovat. Přetypování především slouží k obcházení typových kontrol.
- **Výhodou statického typování je lepší možnost odhalení typových chyb.**
- Hlavní nevýhodou této metody je větší složitost programových konstrukcí, délka zdrojového kódu a tím i menší pružnost programovacího jazyka.
- K nejčastěji vyskytovaným běhovým chybám patří přetečení datového typu.
- Mezi neznámější zástupce staticky typovaných jazyků patří Java, Ada a jazyk C.

2.1 Typy

$$\Gamma \vdash e : A \qquad \text{„}e \text{ je well-formed term typu } A \text{ v prostředí } \Gamma\text{“}$$

- Typování
  - **statické** - formálně specifikováno typovým systémem (judgements, typová pravidla, prostředí). Typová pravidla rozhodují platnost rozhodnutí (judgements) na základě jiných rozhodnutí o kterých je známo, že jsou platné.
  - **dynamické**
- Typ je množina přípustných hodnot a operací s nimi.
- **TopType**
- **Type preservation** - zachování typu během přepisovací relace $\forall e, e' \in Expr : (\vdash e : t) \wedge e \rightsquigarrow e' \Rightarrow (\vdash e' : t)$
- **Progress** - když mám nějaký term v konfiguraci, tak pak to je buď finální konfigurace, nebo lze ještě nejméně jednou přepsat. Tzn. dobře otypovaný term neskončí ve *stuck* stavu (stav, pro který není definován výstup). (důkaz lze udělat analýzou pravidel). $\forall e \in Expr : (\vdash e : t) \Rightarrow e \in (Num \cup Bool) \vee \exists e' \in Expr : e \rightsquigarrow e'$.
- Type preservation + progress = **Soundness**. An argument is sound if and only if: *The argument is valid* and *All of its premises are true.* (i.e. All men are mortal. Socrates is a man. Therefore, Socrates is mortal.)

- **Terminace** - důkaz např pomocí *energie*, nadefinuji „energii“, nadefinuji, že při každém přepsání se musí snížit.
- **Determinismus** - programovací jazyk je deterministický právě tehdy když existuje právě jeden výstup pro každý pár *programu* a *vstupů*. $\forall v, v' \in (Num \cup Bool) : \forall e \in Expr : e \rightsquigarrow^* v \wedge e \rightsquigarrow^* v' \Rightarrow v = v'$. Platí to, protože platí konfluence relace.
- **Konfluence** - „strom vyhodnocování relace se rozdělí a pak se zase spojí“.

2.2 Polymorfní typy

Polymorfní typ je typ, jehož operace mohou být aplikovány na hodnoty jiného typu.

- **Rekurzivní** - používají se na zakodování seznamů a stromů, obsahují více hodnot stejného typu http://en.wikipedia.org/wiki/Recursive_data_type
- **Univerzální** - generický typ, např. List<X>
- **parametrický polymorfismus** - pomocí něj lze zapsat funkce jedním stylem a zároveň zachovat bezpečné typování. Tzn. funkce je zapsána genericky a umí zpracovávat vstupy bez ohledu na jejich typ. Typický příklad je třeba funkce append která se může zavolat s jakýmkoliv typem a správně se vyhodnotí. Není potřeba deklarovat `append :Integer` nebo `append:Bool`.
- **ad-hoc polymorfismus** - jedna funkce může mít mnoho implementací na základě zpracování jednotlivých typů - přetěžování funkcí v Javě.

2.3 Typy vyššího řádu

- Product Type - např. tuple, 2 a více typů, 2 operandy $(A \times B)$ http://en.wikipedia.org/wiki/Product_type
- Union Type - datová struktura, která může udržovat více fixních typů http://en.wikipedia.org/wiki/Tagged_union
- Function Type
- Record Type - klasický objekt, který má fieldy

Jde o takzvané monády a debuggable functions (tohle je spíš možné využití).

Monády - umožňují řetězit procedury za pomocí čistě funkcionálního programování. Skládají se ze dvou operací a to bind a return(unit).

„In functional programming, a monad is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together. This allows the programmer to build pipelines that process data in steps, in which each action is decorated with additional processing rules provided by the monad.“

2.4 Rekonstrukce typů

Schopnost z proměnné vyvodit její typ. Je to vlastnost některých silně staticky typovaných jazyků.

„Type inference refers to the automatic deduction of the data type of an expression in a programming language. If some, but not all, type annotations are already present it is referred to as type reconstruction.“

Například v csharpu, stačí místo typu psát klíčové slovo `var`.

```
var results = new Car();
// results je typu Car
```

2.5 Abstraktní typy

Příkladem jsou Abstraktní třídy v Javě. Vyskytují se hlavně ve staticky typovaných jazycích.

Fronta, HashMapa, Množina (Set), Seznam (List), Zásobník (Stack), ...

3 Teorie HCI, kognitivní aspekty, způsoby interakce, speciální uživatelská rozhraní.

3.1 HCI - Human-Computer Interaction

Human-Computer Interaction (česky interakce člověk - počítač) je průnikový **obor, který se zabývá fenoménem tvorby UI**. Zabývá se analýzou návrhu, vyhodnocování a zavádění interaktivních výpočetních systémů používaných lidmi a jevů, které interakci doprovázejí. Skládá se ze tří částí: **jedinec, počítač a způsob, jakým dohromady spolupracují**.

Cílem je návrh a vývoj prostředků či systémů, které jsou použitelné, efektivní, bezpečné a intuitivní. Dále se snaží přizpůsobit výměnu dat mezi lidmi a stroji tak, aby byla méně stresující a náchylná k nedorozumění.

3.2 Kognitivní aspekty

- **kognitivní psychologie** zkoumá proces myšlení, učení a rozhodování
- mentální model:
  - kognitivní struktura
  - vnitřní reprezentace okolního světa, kterou si vytváříme v hlavě
  - jak objekty určité třídy reagují s objekty jiné třídy, jak objekty v průběhu interakce mění své vlastnosti
  - založeny na zkušenosti, mohou být nepřesné, neodpovídat zákonům fyziky
  - lze je použít k predikci (kam dopadne hozený míč)
- kognitivní model uživatele - model, jak uživatel pracuje, na jehož základě se předpoví jeho chování (interakce s UI), výhody: nemusí se vytvářet prototypy, není nutné testování se skutečnými uživateli, vědecký základ pro návrh
- estetika a efektivita kognitivních funkcí - důležitost vizuální podoby, atraktivní věci jsou použitelnější

Kognitivní teorie v HCI Modelování úkolů - metody KLM, GOMS

- GOMS (Goals, Operators, Methods, Selectors) - popis struktury úloh, task rozpadlý do menších subtasků v hierarchicky přehledné síti,expert provádí UI operace
- KLM (Keystroke-Level Model) - low-level verze GOMS, uživatelské operátory (K-keystroke, P-point, D-drawing, M-mental think), tabulkové časy pro každý operátor, každé operaci se přiřadí čas vykonání
- Hick's Law - čas potřebný k rozhodnutí se, n stejně pravděpodobných možností, průměrný čas výběru jedné z nich: $T = b \log_2(n+1)$
- Fitt's Law - předpovídá jak dlouho trvá uživateli vybrat cíl, vyhodnocení vstupních zařízení, pohyb k cíli o velikosti S ve vzdálenosti D: $T = a + b \log(\frac{D}{S} + 1)$, a, b - konstanty závislé na zařízení

- Model Human Processor / Human Information Processor Model - model lidského poznání vytvořený za použití teorií uvedených výše, modeluje, jak uživatel zachází s informacemi, perceptuální, kognitivní a motorický subsystém

3.3 Způsoby interakce

- přímá manipulace - hry (značné implementační požadavky)
- navigace (menu, odkazy) - web - není třeba si pamatovat příkazy jako CL, ale zabírá příliš místa na obrazovce
- formuláře - web
- příkazy - unix, první metoda komunikace člověk - PC, BNF, konečný automat
- přirozený jazyk
- nové typy interakce - mluva, gesta, eye-tracking, haptické (hmatová odezva) displeje

3.4 Speciální UI

- mluvící systémy AI (Eliza, Cortana, Siri atd.)
- multimodální - kombinování více vstupů
- pokročilý vstup z klávesnice - pubtran se „učí“
- magnetický prsten - klikání, scrollování
- haptická odezva
- papírový mobil - ohýbací gesta
- spolupráce telefonů např při sharování
- multi-touch

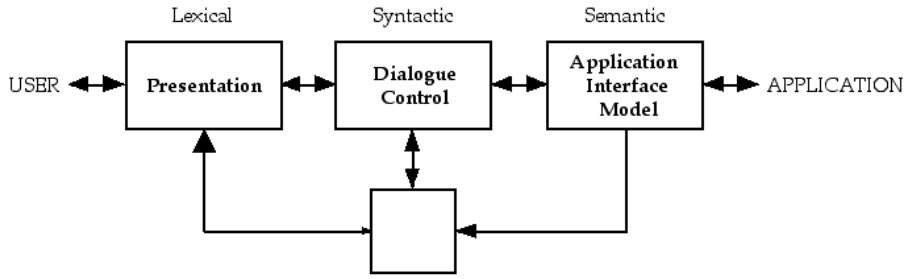
Architektura UI Cíl: oddělení UI a aplikace, výběr možností prezentace informace uživateli, koordinace interakce, modifikovatelnost a přenositelnost. Interaktivní systém poskytuje tři funkce (vrstvy):

- prezentační (UI)
- dialogovou (komunikace s uživatelem)
- aplikační (vlastní účel SW systému)

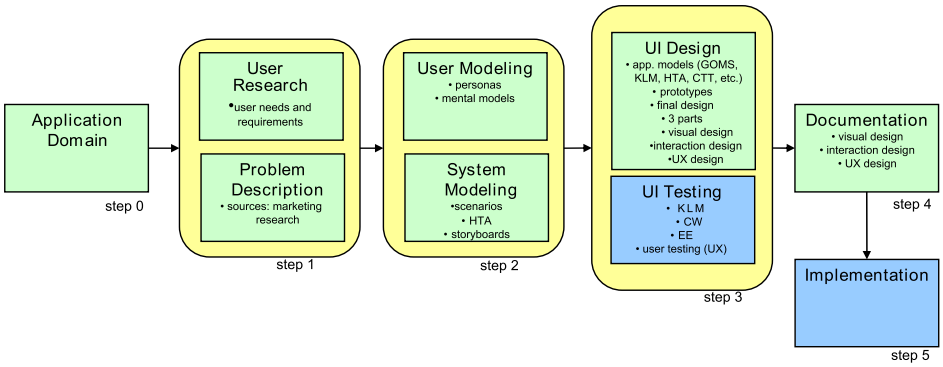
Monolická architektura = všechny části v jednom

Seeheim model URL sémantická vazba je často pomalejší, přímá vazba mezi aplikační a prezentační vrstvou regulovaná dialogem umožní okamžitou odezvu (switch).

Výhody:oddělená prezentační vrstva podporuje přenositelnost a modifikovatelnost, oddělená aplikační vrstva dovoluje modifikace aplikace beze změny UI, oddělená dialogová část umožňuje změnit uživatelskou interakci beze změny prezentační části. Nevýhody: řada modifikací se promítá do všech částí, komplikované sémantické vazby.


Lexical
Syntactic
Semantic
USER
Presentation
Dialogue Control
Application Interface Model
APPLICATION

4 Metody návrhu, uživatelské a konceptuální modely



Cyklus tvorby UI

- Analýza & Návrh - porozumění uživateli (cílová skupina) a jeho potřebám
  - **analýza úlohy** - výkonnost software, hardware, uživatele při provádění úlohy, co uživatelé dělají, co k tomu potřebují za nástroje, co potřebují vědět, metoda: HTA
  - popis průběhu dialogu, slouží k následující implementaci UI
- Implementace - prototypování
- Vyhodnocení - hodnocení prototypu ve spolupráci s uživateli (kvalitativní & kvantitativní)
- další iterace...

Analýza

- specifikace aktivit, které systém bude dělat
- specifikace uživatelů - ti, kteří aktivity budou dělat
- volba formy řešení - forma UI, SW podporující UI, OS, systémové požadavky, HW

Uživatel

- Uživatelské požadavky - Obecné požadavky: fyzické, kognitivní, sociální. Mohou být také specifické požadavky, které se vztahují přímo k problémmu.
- Modely uživatele - KLM (Keystroke-level model), persony

Principy použitelného návrhu (usable design)

- jednoduché a přirozené dialogy v jazyku uživatele
- konzistentnost akcí, příkazu, layoutu, terminologie
- minimalizovat paměťovou zátěž uživatele - rozpoznávání je snazší než vzpomínání
- zpětná vazba
- kontrola vstupu
- snadné vrácení akcí - podpoří chuť experimentovat
- výrazně značená ukončení - uživatel se nesmí ocitnout v pasti
- zkratky - rychlé provedení časté akce pro zkušené uživatele
- robustní systém poskytující snadno ovladatelné prostředky k nápravě chyb
- užitečná nápověda a dobrá dokumentace (hledána v kritických situacích)

Terminologie

- Goal - to čeho chceme dosáhnout
- Task - posloupnost aktivit, která dá *goal*
- Action - krok nebo akce - část *tasku*

4.1 Specifikace požadavků

- HTA (Hierarchical task analysis) - dekompoziční strom, kde je *goal* zakreslen do postupně se rozpadajících menších *tasků*. (CTT[1] - strom s operátory a symboly)
- Storyboard - série snímků a skeč§ („komiks“ popisující *goal*)
- Scénáře - jednoduché výpravné příběhy průběhu úkoli *„Uživatel napíše všechny účastníky akce, poté systém zkontroluje zda je vše vyplněno v pořádku a vytvoří událost...“.*
- Případy užití - popis interakce člověka se systémem.

4.2 Uživatelský průzkum

Získávání informací o budoucích uživatelech systému, jejich **potřeby, zvyky, zkušenosti a dovednosti**.

- **kvalitativní** - menší vzorek lidí, více informací, rozhovor, etnografické (pozorování chování) pozorování
- **kvantitativní** - více lidí, méně informací, průzkumy, testy, pozorování
- kombinovaný

[1] Concurrent Task Tree

Persona Detailně popsaný hypotetický uživatel reprezentující nějakou uživatelskou skupinu. Je založen na nasbíraných informacích. Nevýhody mohou být: nekonzistentní uživatel, představování si sama sebe jako uživatele.

Metody sběru dat

- pozorování - introspekce (sebepozorování) - zahrnuje kognitivní průchod, extrospekce
- rozhovor - strukturovaný vs. volný, být neutrální a zvědavý (30-90 min), přímé otázky, žádná anonimita, vliv tazatele
- dotazování (průzkum) - jednoduché otázky, používat rozsahy, neopakovat otázky, lidé lžou (chtějí vše, levně a ihned)
- experiment

4.3 Modely pro návrh UI

Konceptuální model (design model) znamená to, jak to je navrženo. **Uživatelský** model (user model) značí co uživatel očekává. Nesoulad modelů vede k pomalému provádění úloh, chybám a frustraci

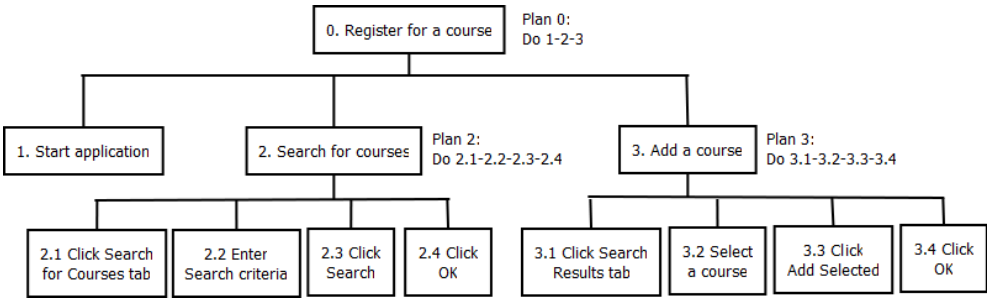
Mentální model Uživatelovo porozumění jak se objekty chovají a jak akce prováděné přes UI ovlivňují jejich chování získané na základě zkušeností. Očekávané struktury a chování (menu, ukládání souborů, zpětná vazba, interpretace akcí). Vědomé i podvědomé procesy, které obsahují aktivaci obrazů a analogií. Hluboké a mělké modely (řízení auta vs. fungování auta).

UI musí prezentovat model vizuálně, mapování reálných prvků na rozhraní. Dobrý konceptuální model zahrnuje:

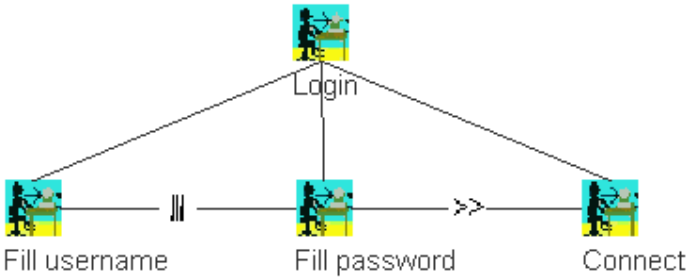
- dostupnost funkcí (affordances)
- návaznost (kauzalita)
- omezení (constraints)
- mapování jednotlivých kroků na akce
- vzory chování cílových uživatelů

5 Formální popis uživatelských rozhraní

HTA (Hierarchical task analysis) Dekompoziční strom, kde je nějaký cíl (úloha) zakreslen do postupně se rozpadajících menších podúloh. Používá se ve fázi návrhu UI pro popis vzájemného uspořádání podúloh.



CTT (Concurrent Task Tree) Podobný strom jako HTA, ale s operátory a symboly. Úloha přihlášení: vyplním uživatelské jméno a zároveň heslo, pak je mi umožněno se připojit.



- Enabling T1 >> T2
- Disabling T1 [> T2
- Interruption T1 |> T2
- Choice T1 [] T2
- Iteration T1* nebor $T1_n$
- Concurrency T1 ||| T2
- Optionality [T]

Storyboard Série snímků a skečů („komiks“ popisující nějakou úlohu).

Scénáře Jednoduché výpravné příběhy průběhu úkolu

„Uživatel napíše všechny účastníky akce, vyplní místo a datum konání. Systém poté zkontroluje zda je vše vyplněno v pořádku a vytvoří událost.“.

Případy užití Popis interakce člověka se systémem..

Keystroke-Level Model (KLM) Cílem je vypočítat čas potřebný pro provedení úlohy Operátory:

- stisk klávesy (**K**eystroke) - určený rychlostí psaní
- ukázat na cíl na displeji (**P**ointing) - určeno pomocí Fitt's Law
- položit ruku na vstupní zařízení (**H**oming) - odhad měřením
- mentální příprava akce (**M**ental preparation) - odhad měřením, heuristika pro předřazení
- čas reakce systému (**R**eaction)

Jsou časové odhady (tabulkové) pro každý operátor. Předpokládá provádění úloh bez chyby, předpovídá jen efektivitu, ignoruje paralelní zpracování, prokládání úloh, mentální zátěž, plánování a řešení úlohy („přemýšlecí" čas, uvažovány jsou jen holé akce)

Goals, Operators, Methods, Selection Rules (GOMS) Nejznámější používaná metoda. Složky:

- **Goals** - cíle z hlediska úmyslů koncového uživatele
- **Operators** - elementární perceptuální, kognitivní a motorické akce s fixním časem bez ohledu na kontext
- **Methods** - posloupnost operátorů a podcílů
- **Selection rules** - if-then pravidla určující, kterou metodu použít

Předpokládá provádění úloh bez chyby, úlohy musí mít přesně definovaný cíl, nemodeluje proces řešení problému, chování uživatele

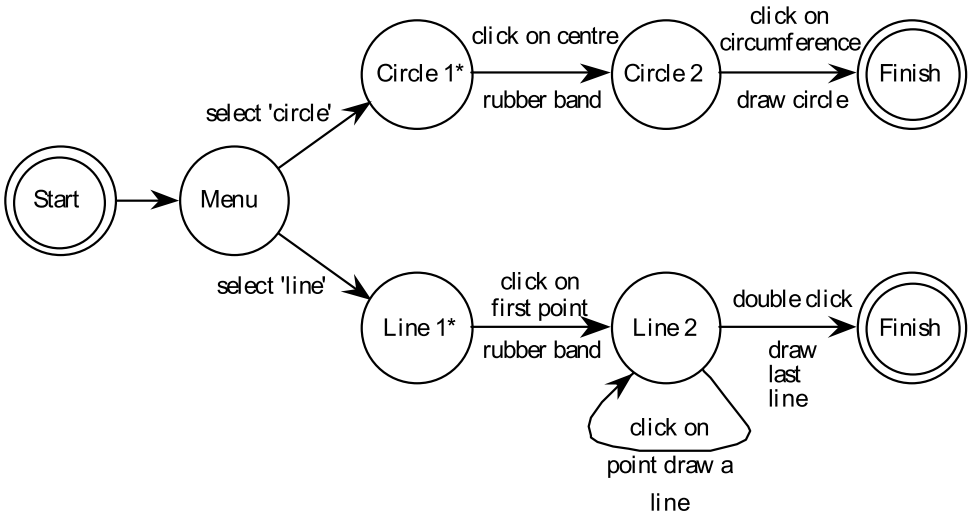
Dialog modeling Z HTA máme představu o posloupnosti kroků, potřebujeme popsat, jak při provádění kroků spolu budou komunikovat uživatel a počítač - jak bude probíhat dialog

- textové (gramatiky, produkční pravidla, událostní algebry)
- diagramy - (STN, PN, flowcharts, JSD)

State Transition Networks (STN) Varianta konečných automatů, konečný počet stavů a přechodů mezi nimi, automat se nachází v pravě jednom stavu (stavy jsou disjunktní). Reakcí na každý uživatelský vstup je přechod z daného stavu do nového stavu. Stav má přiřazenou akci, musí být odlišitelný od jiných stavů, charakterizován vstupy, které k němu vedou. Přechod mezi stavy může být vázán podmínkou, lze k nim přiřazovat popis akcí.

+ model UI, se kterým lze experimentovat
+ možnost automatického nebo poloautomatického vytváření UI
+ kontrola vlastností (úplnost, reversibilita, dostupnost, nebezpečné stavy - ukončení bez uložení)
− některá zařízení mohou mít velká množství stavů

hierarchické STN - varianta pro popis složitých dialogů, obsahuje sub-dialogy (vnořené další sítě).



Petriho sítě (PN) Oproti STN mají synchronizaci - pokračování při splněné podmínce

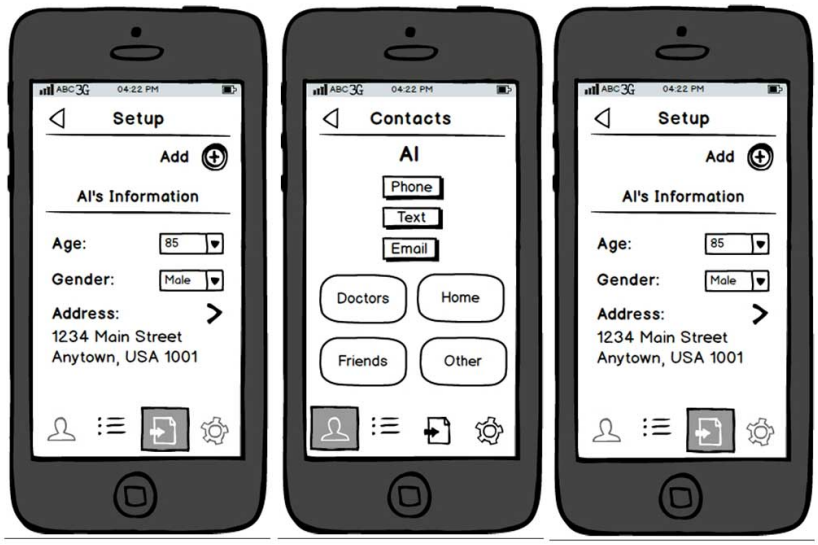
6 Prototypování uživatelských rozhraní

Prototypování se dělá v ranných fázích návrhu. Rozděluje na **Low-Fidelity** a **High-Fidelity** prototypování.

6.1 Low-Fidelity

První náčrtky rozhraní. Jsou to víceméně narychlo načmárané skicy bez detailů, spíše jde o základní rozvržení (layout) prvků, žádná interakce. Typicky skicy na papíru nebo na elektronických prototypovacích SW (není na finálním zařízení). Je to spíše o rychlém zhodnocení nápadů, které jsou převedeny na papírový prototyp.

- velké množství nápadů/alternativ
- krátká doba vývoje - hodiny/dny
- neběží na finálním zařízení
- bez interakce
- testování v laboratoři

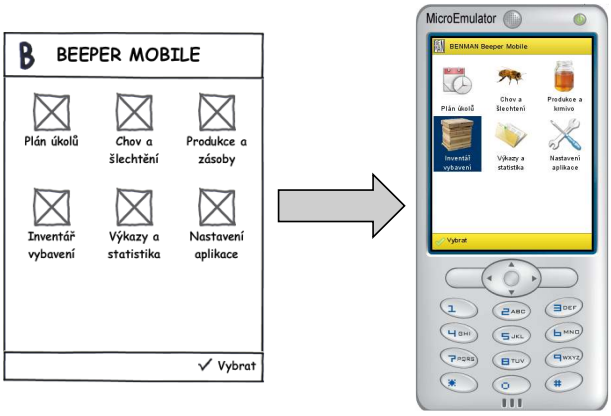


Např. prototypování mobilní app. Vytvoří se několik papírků, každý popisuje krok aplikace. Interakce v podobě fiktvního klikání na papír, dochází k výměně papírků (změna stavu) a tím dochází k fiktivní interakci.

6.2 High-Fidelity

Iluze finálního vizuálního (i interagujícího) návrhu. Vzhled by měl následovat *guidelines* cílové platformy (MS Windows, Android, iOS, ..). Prototyp již ve funkční podobě na cílovém zařízení, např. na telefonu. Interakce realizována jakoby to byla již výsledná aplikace, ovšem logika ještě nemusí být implementována (dummy data, *Wizard of Oz*, atd.,). Hlavní části

- málo alternativ (pokud vůbec nějaké jsou)
- dlouhá doba vývoje - dny/týdny
- dostupné na finálním zařízení
- podoba a interakce téměr podobná té finální
- testování v laboratoři a v „terénu“ (v reálných podmínkách využití)



Problémy s prototypováním

- U lo-fi prototypů dochází ke skipování hlubokých (detailních) uživatelských požadavků
- user confusion: hi-fi prototyp vs. finální aplikace
- drahé a žrout času (speciálně hi-fi)

7 Techniky správy a organizace rozsáhlých softwarových projektů. Nástroje pro správu verzí zdrojových kódů, sledování chyb, pro automatické generování dokumentace a podporu orientace v rozsáhlých projektech. Způsoby komunikace mezi vývojáři navzájem a i s uživateli. Systémy pro sledování a řešení chyb a uživatelskou podporu. Open-source licence a z nich vyplývající práva a licence. Postup začlenění úpravy (patche) do velkého open-source projektu (např. Linuxové jádro)

Motivace pro verzování zdrojového kódu Verzování kódu umožňuje sdílení kódu mezi vývojáři, zálohování, paralelní vývoj v několika souběžných větvích, vrácení se ke konkrétní revizi kódu, zjištění autorství nebo zobrazení statistik. Bez verzovacího systému není možná efektivní spolupráce více vývojářů na jednom projektu. Verzovací systém přináší výhody i v případě, že na projektu pracuje jediný vývojář.

7.1 Nástroje pro správu verzí kódu

GIT je distribuovaný SCM[2] od Linuse Torvaldse. Každá working copy je zároveň repozitář, nezávislé na centrálním serveru. Spoustu operací jako merge, branch,... lokálně. Commity jsou hash celé historie vedoucí ke commitu. Nevýhoda: častější konflikty Subversion (SVN) je centrální SCM, ale rozšířený = dobré nástroje, GUI atd. Míň konfliktů, ale závislost na serveru. CVS podobné SVN ale starší, nevýhody: drahé branchování, problémy s Unicode, netrackuje přejmenovávání a mazání souborů.

7.2 Nástroje pro sledování chyb (bug trackers)

Jsou nástroje (např. bugzilla), které sledují a soustřeďují nalezené chyby. Každá chyba má vypsaný svůj *ticket*, ve kterém jsou veškeré informace k chybě: popis, priorita, reportující osoba, přiřazená osoba, návrh úpravy (např. v podobě patche).

7.3 Nástroje pro automatické generování dokumentace

Javadoc generuje HTML dokumentaci z komentářů v Java kódu, Doxygen je multijazykový generátor dokumentace z kódu, Enterprise Architect umí generovat UML diagramy z kódu.

7.4 Systémy pro spolupráci mezi vývojáři

GitHub je populární sociální platforma pro vývojáře na hosting a spolupráci open-source projektů založený na použití Gitu, Trac je project management systém v Pythonu, který si

[2]Source Code Management

můžete nasadit na vlastní server, obsahuje wiki, bug tracking, time management, etc. Spíše pro menší projekty. JIRA je SaaS systém podobný Tracu se spoustou pluginů, hodí se pro větší projekty.

- Google Groups a podobné mailing listy mohou také sloužit k podpoře.
- wiki stránky
- fóra

7.5 Licence

Svobodný software je software, který respektuje svobodu svých uživatelů a poskytuje jim čtyři základní svobody, které svobodný software definují (publikace FSF 1986):

1. svoboda používat program za jakýmkoliv účelem
2. svoboda zkoumat a upravovat program (předpokladem je přístup ke zdrojovému kódu)
3. svoboda šířit původní verzi programu
4. svoboda šířit upravenou verzi programu

- **Komerční software** – licence daná smluvními podmínkami jež uživatel potvrzuje při nákupu SW
- **Freeware** – zdarma, většinou bez zdrojových kódů, podmínky mohou omezovat další šíření, (komerční) použití, zkoumání
- **Shareware** – jako freeware, ale specifikuje pro které druhy použití je nutné pořídit placenou verzi
- **Permisivní** (akademické) licence (BSD, MIT) – povolují použití/integraci do komerčního SW, vyžadují jen uvádění autora/ů (to je i instituce)
- **Copyleftové** (reciproční) licence (GPL, LGPL, MPL) vyžadují zahrnutí uživatelů do okruhu oprávněných osob k právu nakládat s dílem (modifikovat ho a šířit za stejných podmínek)

Upozornění: Definice open-source nevyžaduje *copyleft*.

7.5.1 BSD (Berkeley Software Distribution)

BSD licence je licence pro svobodný software, mezi kterými je jednou z nejsvobodnějších. Umožňuje volné šíření licencovaného obsahu, přičemž vyžaduje pouze uvedení autora a informace o licenci, spolu s upozorněním na zřeknutí se odpovědnosti za dílo.

7.5.2 MIT License

Licence podobná BSD licenci umožňuje se software nakládat téměř libovolně (používat, kopírovat, modifikovat, slučovat, publikovat, distribuovat či prodávat), jedinou podmínkou je zahrnutí textu licence do všech kopií a odvozenin software.

7.5.3 Apache License

Stejné myšlenkové základy jako licence BSD a MIT. Výslovná zmínka možnosti šířit odvozená díla pod jinou kompatibilní licencí.

7.5.4 GNU GPL (GNU General Public License)

GPL je nejpopulárnějším a dobře známým příkladem silně copyleftové licence, která vyžaduje, aby byla odvozená díla dostupná pod toutéž licencí.

GNU General Public License. Software šířený pod licencí GPL je možno volně používat, modifikovat i šířit, ale za předpokladu, že tento software bude šířen bezplatně (případně za distribuční náklady) s možností získat bezplatně zdrojové kódy. Toto opatření se týká nejen samotného softwaru, ale i softwaru, který je od něj odvozen. Na produkty šířené pod GPL se nevztahuje žádná záruka. Licence je schválená sdružením OSI a plně odpovídá Debian Free Software Guidelines.

V rámci této filosofie je řečeno, že poskytuje uživatelům počítačového programu práva svobodného softwaru a používá copyleft k zajištění, aby byly tyto svobody ochráněny, i když je dílo změněno nebo k něčemu přidáno. Toto je rozdíl oproti permisivním licencím svobodného softwaru, jejímž typickým případem jsou BSD licence

7.5.5 GNU LGPL (GNU Lesser General Public License)

Lesser/Library GPL. Licence je kompatibilní s licencí GPL. Pod touto licencí se šíří zejména knihovny, protože narozdíl od licence GPL umožňuje nalinkování LGPL knihovny i do programu, který není šířen pod GPL.

7.5.6 MPL (Mozilla Public License)

Mozilla Public License. Základním elementem pokrytým licencí je každý jednotlivý zdrojový soubor. Autor takového souboru umožňuje komukoliv používat, měnit a distribuovat jeho zdrojový kód (i jako součást většího díla). Každá změna původních souborů je krytá licencí, tzn. musí se tedy zveřejnit. To samé platí pokud přenesete část původního souboru do nového souboru, tj. celý nový soubor je pak nezbytné zveřejnit. Pokud vytváříte nový produkt přidáním nových souborů, můžete pro tyto nové soubory použít libovolnou licenci. Binární verze lze licencovat libovolně, pokud to není výslovně v rozporu s MPL (zákaz distribuce zdrojů). Produkty pod touto licencí jsou distribuované jak jsou ("as is"), tj. bez záruk libovolného druhu.

7.5.7 CreativeCommons

Je license vhodná pro jakýkoli obsah, nejenom pro software. Do license si člověk může dát 1-4 z těchto atributů:

- Attribution - nutno uvést autora originálního projektu
- Noncommercial - možno upravovat jenom pro nekomerční účely

- NoDerivatives - možno pouze kopírovat dané dílo, ale nikoliv ho upravovat
- Share-alike - znamená virální copyleft

7.6 Kontribuce do projektu

- Vyberu si projekt do kterého chci přispět.
- Naleznu jejich nástroj pro sledování chyb, a vyberu si některý z bugů.
- „Přivlastním“ si bug, přiřadím ho k sobě (změní se stav ticketu) a tím dam ostatním vývojářům vědět, že tento bug již řešit nemusí.
- Vytvořím si vlastní fork repozitáře (vlastní kopii), do kterého budu vyvíjet
- V lokálním repozitáři si vytvořím novou větev z aktuální vývojové větve.
- Opravím bug, commitnu a pushnu změny do repozitáře (svého forku).
- Pak buď vytvořím pull request, nebo vytvořím patch soubor, který připojím k ticketu bugu. Změním stav ticketu na „fixnuto“, tím dám echo, že bug je z mé strany splněn.
- Čekám na zrevidování mého kódu správcem a případné zařazení do vývojového repozitáře.

8 Požadavky a pravidla pro tvorbu přenositelného kódu. Organizace projektů a struktura operačních systémů pro zajištění přenositelnosti mezi různými platformami (OS, CPU). Vnitřní a vnější reprezentace dat, převody mezi nimi, vztah k síťovým protokolům (endianing, serializace atd.)

8.1 Požadavky a pravidla pro tvorbu přenositelného kódu

Při realizaci softwarových systémů často nemůžeme zanedbat cílovou hardwarovou architekturu stroje a musíme psát kód tak, aby jej bylo možné snadno upravit (ideálně pouze rekompilovat) pro běh na jiných platformách. V okamžiku kdy je nutné přímo interagovat s některou hardwarovou komponentou počítače (řídící jednotky, SCADA systémy, ...), se dostáváme až na úroveň, kdy musíme řešit věci, jako uspořádání bytů v proměnné apod.

Nejjednodušší je použít virtual machine jako Java nebo .NET. Pokud píšeme v C, tak to chce: používat standardizovaná API (POSIX, IEEE Std. 1003.1,), nepředpokládat pořadí byte ve slovech (little endian, big endian), nepředpokládat počet bitů pro adresaci (32 vs 64), používat multiplatformní knihovny a nevolat přímo metody systému.

8.2 Organizace projektů a struktura operačních systémů pro zajištění přenositelnosti mezi různými platformami (OS, CPU)

Aby bylo možné snadno přenášet projekty mezi různými operačními systémy a jejich verzemi a hardwarovými architekturami, bylo definováno několik standardů a vyvinuto několik nástrojů, které mají tuto přenositelnost usnadnit:

POSIX - Portable Operating System Interface - standardy IEEE 1003 a ISO/IEC 9945 - definuje, co by měl poskytovat operační systém, jaké příkazy, jaké vlastnosti - na základě tohoto standardu je potom poměrně snadné převádět projekty mezi různými distribucemi Linuxu, UNIXY, atd. (uvádí se, že to je mnohdy jednodušší než převod projektů mezi různými verzemi Windows). Definuje příkazy OS nezávislé na platformě.

Filesystem hierarchy standard (FHS) - dokument popisující, co má být ve kterém adresáři v linuxové distribuci, mezi typické adresáře patří

- `/usr/bin` - binární programy (platformně závislé)
- `/usr/local` - prefix do kterého je instalován software lokální pro tento stroj
- `/usr/share` - sdílené (na hw platformě nezávislé) zdroje - často může být tento systém sdílen mezi různými stroji (např. pomocí NFS)
- `/usr/lib` - knihovny (platformně závislé)
- FHS popisuje, co by měl/neměl který adresář správně obsahovat

8.3 Kompilace GNU balíků

Nejrozšířenějším nástrojem pro kompilaci GNU balíků je systém Autotools. Tento systém se skládá z více nástrojů. Mezi hlavní a nejdůležitější patří:

- m4, aclocal, autoheader
- autoheader
- autoconf
- automake

Tyto nástroje zpracovávají vstupní soubory, které obsahují informace o projektu, závislostech, požadavcích na cílovou platformu atd. Jedná se zejména o soubory: `Makefile.am`, `configure.ac`, `config.h.in`.

Proces sestavování (build) softwarového balíku se dělí na část, která probíhá u vývojáře (vyžaduje ke svému běhu větší množinu nástrojů) a část, kterou spouští klient.

Kompilace GNU balíků autoheader vygeneruje hlavičky na dané platformě, které se používají v kódu, autoconf vygeneruje soubor configure, který klient používá k vygenerování Makefilu, automake přechroustá vstupní Makefile do správného formátu pro configure.

8.4 Portace kódu a křížový překlad

Portace kódu znamená generování binárek pro jiné platformy: Native build to vygeneruje pro vlastní platformu. Cross build vygeneruje přímo binárky pro danou platformu. Cross-native build vygeneruje tool-chain, který vygeneruje binárky na dané platformě.

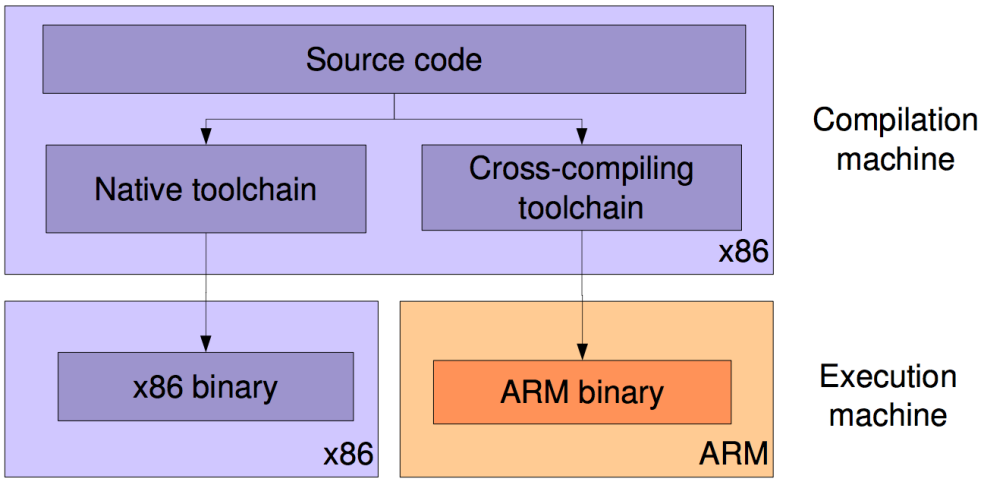
Většinou není nutné, aby každý uživatel kompiloval software ze zdrojových kódů. Je možné dodávat konkrétní verzi zkompilovanou pro cílovou platformu. Pro sestavení programu se používá build toolchain (v případě použití GCC se označuje jako GNU Tool Chain). Ten zahrnuje jednak vlastní kompiler (např. GCC - GNU Compiler Collection) a dále nástroje pro vytváření knihoven, linker atd.

Tento toolchain může kompilovat kód pro různé platformy. Na základě toho rozeznáváme native toolchain a cross-compiling toolchain (výsledné binární soubory běží na platformě toolchainu nebo na jiné platformě).

8.5 Konverze vnitřních a vnějších/síťových formátů dat

Vnitřní: závisí na architektuře

- **little endian** má bity zprava doleva, což je dobré pro binární operace - Intel, ARM
- **big endian** - bity zleva doprava, jako když čteme, PowerPC



Vnější: Platformě nezávislé, třeba int a double - konverze v programovacím jazyku, Síťové: nutná (de)serializace např. JSON, XML, HTML,

- little endian/big endian
- textové formáty: XML, HTML, SOAP, JSON
- XDR (external data representation) - nezávislý na arch. systému, od roku 1995 IETF standard
- RPC, CORBA
- padding - zarovnávání packetů

9 Co je to architektura zaměřená na služby (SOA)? Základní pojmy, vztah k objektově orientované architektuře. Konceptuální model a formalismy pro modelování SOA.

9.1 Architektura zaměřená na služby (SOA)

SOA je model softwarové architektury, kde byznys funkcionalita je logicky oddělena (grupována).

- **sada návrhových principů** používaných během vývoje systému a integrace.
- SOA je styl návrhu takových aplikací, které jsou složeny z vícero odlišných částí (poskytující funkcionalitu jako služby jiným aplikacím), které mají definované jednoduché rozhraní.
- SOA nabízí množinu služeb, které mohou být použity v různých business doménách
- způsob, jak spolu mohou dvě **distribuované aplikace** komunikovat, i když běží na různých **platformách** a **technologiích**
- definuje **rozhraní** ve smyslu protokolů (WSDL, XML, HTTP, UDDI, SOAP) a funkcionalit
- dodržuje zásady loose coupling jak mezi jednotlivými službami, tak mezi službou a vrstvou, která leží pod ní
- odděluje funkčnost do nezávislých menších jednotek (nebo services), které jsou dostupné na síti

„SOA is a style of architecting applications in such a way that they are composed of discrete software agents that have simple, well defined interfaces and are orchestrated through a loose coupling to perform a required function.“

9.1.1 Webová služba

Služby zahrnují neasociované, *loosely coupled* jednotky funkcionality, kter0 jsou přístupn0 přes internet dalším programům. Služby (a jejich konzumeři) komunikují mezi sebou předáváním dat ve *well-defined* sdíleném formátu (např. XML, JSON).

- **loose coupling** - services udržují vztah, který minimalizuje závislosti a udržuje jen minimální povědomí o ostatních službách.
- **abstrakce** - služby skrývají logiku s vnějškem (my se jen dotážeme, a dostaneme naše data)
- **znovupoužitelnost** - logika je rozdělena do dalších služeb za účelem dalšího znovupoužití
- **kompozice** - více služeb může spolupracovat (kompozice) a tvořit tak plnohodnotnou aplikaci (*mashup*)

Stateless vs. Stateful service

Stateful:

- každý dotaz musí obsahovat veškerá data (kontext, stav) - stavová služba si **drží session** mezi klientem a serverem.
- **hůře se škáluje** (pokud máme hodně klientů musíme si někde držet kontext pro každého klienta zvlášť).

Stateless:

- bezestavový znamená, že server si **nedrží** žádnou **session**
- každý **request** je izolovaná transakce (request musí obsahovat všechny potřebná data) a nemá vztah k žádnému předchozímu requestu
- **lépe škálovatelné** (není nutnost držet kontext pro každého klienta)
- lépší zotavení z chyb (díky izolovaným requestům)
- **spolehlivost**

Idempotentní requesty

Služba by měla zajišťovat omezení na idempotentní requesty. To znamená, že **duplicitní požadavky** mají **stejný efekt** jako unikátní požadavek.Toto omezení umožní poskytovateli a klientovi zlepšit celkovou spolehlivost služby pouze tím, že se v případě výskytu chyby požadavek zopakuje.

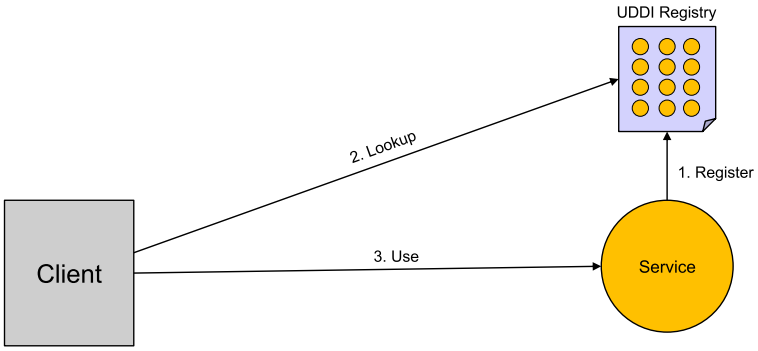
Metody PUT a DELETE v HTTP protokolu jsou definovány jako idempotentní. Metody GET, HEAD, OPTIONS a TRACE, jsou předepsány jako bezpečné, a měly by být rovněž idempotentní. Oproti tomu POST metoda nemusí být nutně idempotentní, a proto se zasláním shodného POST požadavku vícekrát, může navíc ovlivňovat stav nebo způsobit další nežádoucí účinky.

Vztah k objektově orientované architektuře

- používání komponent
- dodržování zásad *low coupling* a *high cohesion*
- komunikace přes rozhraní
- abstrakce

9.2 Konceptuální model

Koncept SOA je založen na interakci mezi dvěma klíčovými entitami: poskytovatelem a spotřebitelem služeb. Každá služba je specifikovaná svým popisem, na základě kterého spotřebitel vyhledá odpovídající službu v registru služeb a naváže s ní komunikaci.


UDDI Registry
2. Lookup
1. Register
Client
3. Use
Service

10 Webové služby. K čemu slouží? Popis a vyhledávání služeb. Technologie pro implementaci a nasazení služeb a klientů. Protokoly, kódování obsahu. Top-down a bottom-up design

10.1 Webové služby

W3C definice - SW systém designovaný k vzájemné *machine-to-machine* spolupráci přes internet, který má API přístupné přes **HTTP**, rozhraní je popsáno v strojové čitelném formátu (**WSDL**) a interakce s WS je pomocí **SOAP** zpráv (HTTP + XML).

RPC Web Service RPC webové služby představují volání vzdálené funkce. Metoda je popsána jako operace ve WSDL. Parametry metody i odpověď jsou posílány jako XML zabalené v SOAP. Vetšinou je implementováno jako mapování služby přímo na jazykově specifické volání funkce. Není tedy loosely coupled.

SOA Web Service Webové služby se také používají k implementaci SOA, kde je základní jednotkou komunikace zpráva, nikoli operace. Tato skutečnost je často označována jako „message-oriented“ služba. Na rozdíl od RPC webových služeb jsou SOA webové služby loose coupled, jelikož jsou zaměřeny na kontrakt poskytnutý WSDL, ne na implementační detaily.

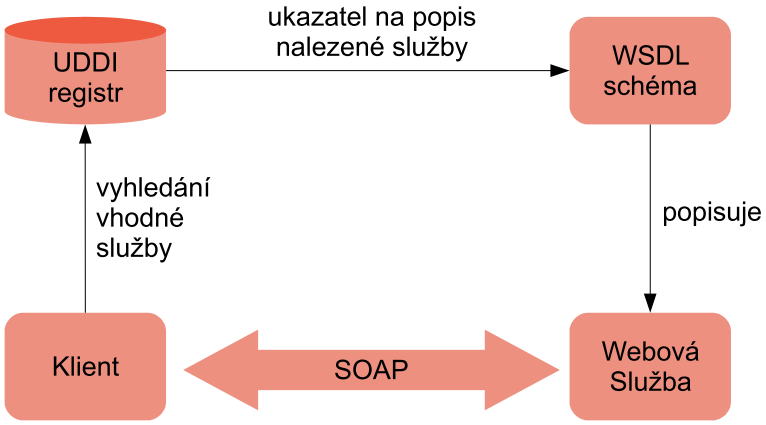
RESTfull Web services RESTfull webové služby jsou speciální podmnožinou webových služeb, které mají sadu předem definovaných operací. Tyto předem definované operace jsou přímo operace převzaté z protokolu HTTP, jedná se tedy o operace GET, POST, HEAD, atd. Každá z operací má vymezené svoje postavení vůči zdroji, který je identifikován pomocí URL. Následující tabulka shrnuje použití jednotlivých HTTP funkcí z pohledu REST služeb.

Využití metod ve správném významu je zcela klíčové pro čisté REST služby. Existuje mnoho reálných implementací webových služeb, kde je právě tento princip porušen.

10.2 Vyhledávání služeb - UDDI registr

UDDI (The Universal Description, Discovery and Integration Service) poskytuje mechanismus, přes který mohou klienti dynamicky hledat požadované webové služby. Tímto způsobem by aplikace měly být schopny se kontaktovat na služby poskytované externími partnery. Registr UDDI má dva druhy klientů: ty, kteří chtějí nějakou službu poskytovat a ty, kteří chtějí službu využívat.

- platformně nezávislý XML-based registr služeb
- mechanismus pro hledání WS
- registry:
  - white pages - adresy, kontakty, identifikátory
  - yellow pages - kategorizace servis založena na standartních taxonomiích
  - green pages - technický popis servis (jak k ní přistoupit atd.)



10.3 Technologie a protokoly

RPC (Remote Procedure Call)

- technologie dovolující programu vykonat proceduru (zavolat službu), která může být uložena na jiném místě než je umístěn sám volající program
- klient/server model
- synchronní volání klienta - je blokován dokud server neodpoví
- může nastat chyba v případě chyby sítě

Postup Nejprve proběhne jednoduché zabalení parametrů a identifikátorů procedury do formy vhodné pro přenos mezi počítači (tzv. *marshalling* (serializace)). Poté se balíček odešle. Balíček se na vzdáleném místě rozbalí, zjistí se o jakou proceduru jde (*unmarshalling*). Následuje samotné zavolání a provedení procedury. Výsledek procedury se opět zabalí a odešle zpět. První počítač výsledek opětovně rozbalí. A přijatá hodnota se předá proceduře.

CORBA (Common Object Request Broker Architecture)

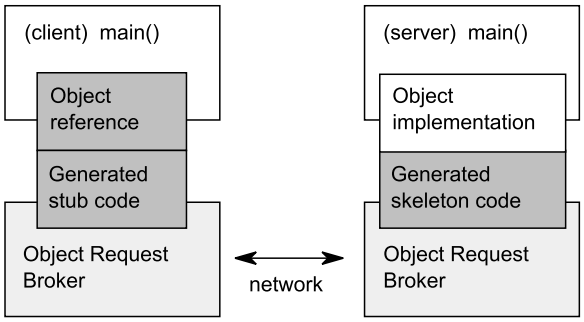
- standard, který umožňuje komunikaci mezi různými platformami a jazyky
- jazykově, technologicky a platformě nezávislý
- specifikace zahrnuje: silné typování, výjimky, síťový protokol
- CORBA IDL (Interface Definition Language)
- transakce, security, marshalling

Je to objektová sběrnice, která objektům umožňuje transparentně vysílat požadavky (requests) směrem k jiným objektům (ať již lokálním nebo vzdáleným) a následně od těchto objektů přijímat odpovědi (replies). Toto prostředí je nezávislé jak na programovacím jazyce, tak na operačním systému či hardwarové platformě.

Pro komunikaci mezi CORBA objekty jsou důležité specifikace jejich rozhraní v jazyce IDL (Interface Definition Language). Ty slouží jako jakási forma kontraktu mezi CORBA objekty a jejich potenciálními klienty jinak řečeno, klient volá pouze metody nad rozhraním

definovaným v IDL, a nestará se o to, jakým způsobem je daný objekt implementován. IDL specifikace rozhraní jsou následně, s pomocí IDL kompilátoru, mapovány do konkrétních programovacích jazyků, ve kterých jsou již pak dané CORBA objekty implementovány. Použitím tohoto mechanismu dochází k oddělení implementace CORBA objektů od jejich rozhraní, čímž se dosahuje již jednou zmíněné nezávislosti CORBA prostředí na programovacím jazyce.

Volání metod CORBA objektů Vzdálený CORBA objekt je v klientském adresovém prostoru zastupován jiným objektem, tzv. *proxy*. Proxy má obvykle stejné rozhraní jako cílový objekt, a jeho metodami jsou tzv. *stuby* (stubs). Stubem se označuje kód, který vezme všechny parametry volání, vloží je spolu s identifikací dané operace do zprávy (marshalling) a tuto zprávu předá jádru ORBu k doručení směrem k cílovému objektu. Na straně serveru předá ORB došlou zprávu skeletonu. Úkolem skeletonu je vyjmout z ní identifikaci požadované operace spolu s jejími parametry, a tuto operaci zavolat nad implementací CORBA objektu. Formát zpráv předávaných mezi klientem a serverem je specifikován protokolem IIOP (Internet Inter-ORB Protocol), který zajišťuje interoperabilitu mezi ORBy různých výrobců. Z popsaného mechanismu je patrno, že lokální volání metody nad proxy má tedy za následek pro klienta transparentní vyvolání odpovídající metody nad vzdáleným objektem (tento mechanismus již známe z RPC). Poznamenejme pouze, že kód klientského proxy i serverovského skeletonu je automaticky generován IDL kompilátorem na základě popisu rozhraní cílového CORBA objektu [1].



Obrázek 1: CORBA

DCOM (Distributed Component Object Model)

- microsoftí CORBA (konkurent)
- garbage collecting - uvolnění referencí držených klientem (např. při ztrátě spojení)

WSDL (Web Services Description Language)

- XML-base jazyk, který poskytuje popis (rozhraní) služby
- služby jsou definovány jako množina síťových endpointů (portů s adresami a bindováním)
- popisuje veřejný interface služby

REST (Representational State Transfer)

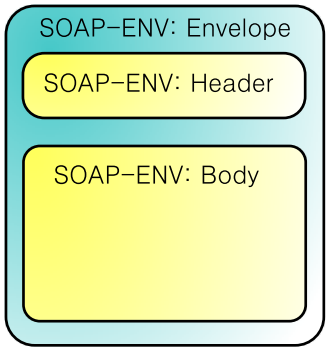
- architektonický styl nebo množina principů pro tvorbu WS
- klienti/servery
- klient se dotáže - server zpracuje požadavek a vrátí odezvu
- původně popsán pro HTTP (ale není na něj omezen)
- vlastnosti:
  - stateless
  - cacheable
  - layered system
  - code on demand (opt)
  - uniform interface

SOAP (Simple Object Access Protokol)

Jelikož veškerá komunikace mezi službami je založená na posílání zpráv, musely být zavedeny takové standardy, aby služby mezi sebou komunikovaly jednotným způsobem. Takovýmto standardem se stal SOAP. SOAP je protokol umožňující spotřebiteli služeb komunikovat s jejich poskytovatelem. Tento protokol je nezávislý na typu sítě, podporuje zprávy ve formátu XML a v současné době je ve specifikaci 1.2 od organizace W3C.

- specifikace pro výměnu strukturovaných informací pomocí webových služeb
- SOAP je nezávislý na transportních protokolech. HTTP je jen jedním z podporovaných.
- postaveno na WSDL a UDDI
- navrženo jako object-access protokol

Struktura zprávy v SOAP Každá zpráva dodržující podmínky kladené SOAP je v podstatě balíček (obálka, angl. envelope). Tento balíček obsahuje hlavičku (angl. head) a tělo (angl. body). Hlavička se skládá z několika bloků, které obsahují metainformace. Hlavička je nepovinná (tzn. může být vynechána). Metainformace v sobě ukrývají část komunikační logiky a obecně umožňují zavádět nová rozšíření. Typicky hlavička obsahuje nutné informace pro všechny služby, které mohou zprávu obdržet. Cílová služba potom na základě těchto informací rozhodne o způsobu zpracování zprávy. Tělo obsahuje samotná data (ve formátu XML). Tělo může také obsahovat sekci pro chyby (angl. faults), která obsahuje logiku pro zpracování výjimek. Většinou jsou v této části uloženy jednoduché zprávy, které slouží k odeslání informací o chybě při výskytu výjimky. SOAP zahrnuje také prostředky pro posílání dat, která jsou těžko popsatelná pomocí XML (binární data, např. obrázky). Takováto data se posílají jako přílohy (angl. attachments).



Obrázek 2: SOAP zpráva

10.4 Kódování obsahu

Posílaný obsah je potřeba nějakým způsobem reprezentovat. Požadavky jsou aby **reprezentace** (zakódování) byla lidsky **čitelná**, **neukecaná** (binary efficient - chceme posílat co nejmenší obsah), platformně **nezávislá** a **standardizovaná**.

- XML - příliš ukecané
- JSON používané
- YAML
- buffers by Google

10.5 Bottom-up design

Nejdříve je naimplementována služba v konkrétním jazyce, následně je vygenerováno WSDL. Považováno za jednodušší návrh. Riziko vzniku závislosti na programovacím jazyku či platformě.

10.6 Top-down design

Nejdřive je napsán WSDL dokument (koresponduje s SOA modelem), následně je z něj vygenerován kód. Považováno za obtížnější. výsledkem je ale čistší design.

11 Webové služby, automatická kompozice služeb. Orchestrace a choreografie, web mash-up. Modelování služeb a procesů (BPMN, BPEL).

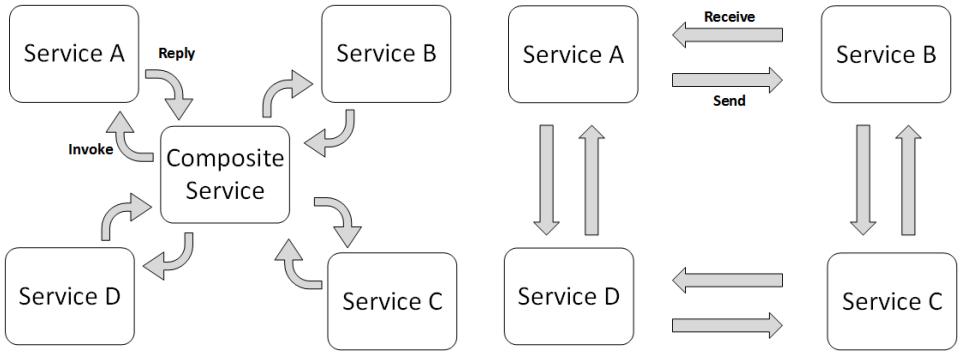
Na kompozitní službu se můžeme dívat jako na sadu služeb, které spolu spolupracují za účelem vykonání určitého procesu, který definuje interakční workflow. **Orchestrace** a **choreografie** jsou rozdílné vzory pro vytváření kompozice služeb. Jazyky pro popis vykonávaného procesu (a tedy i pro popis spolupráce služeb) jsou např. BPMN, BPEL, UML.

11.1 Orchestrace

Při tomto přístupu je centrální prvek, který koordinuje všechny zapojené služby. Tento centrální prvek může být také webová služba. Zúčastněné služby nevědí, a ani nemusí vědět, že jsou účastníky nějakého většího procesu, o tom ví jen centrální prvek. Iterakce při orchestraci nastávají na úrovni zpráv.

11.2 Choreografie

U choreografie není centrální prvek procesu, a proto všechny zapojené služby ví, kdy volat a s kým dalším spolupracovat. V praxi se více využívá **orchestrace**, protože se díky centrálnímu prvku lépe zotavuje z chyb (chyby jsou odchytávány na jednom místě).



Obrázek 3: Orchestrace vs. choreografie

11.3 Mashup

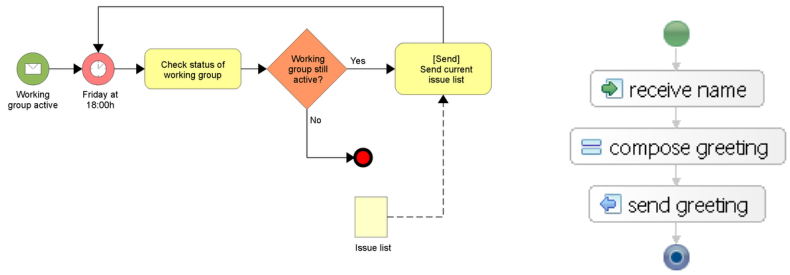
Aplikace kombinující data/funkcionalitu z 2 a více zdrojů.

- Client-based - všechny data se z více služeb volají z klientova prohlížeče (problém s javascriptem Same-Origin Security Policy - POST dotazy).
- Server-based - Prostředník v podobě serveru, uživatel se dotazuje na jeden server, a ten za něj získává data z ostatních služeb.

11.4 Modelovací jazyky BPMN, BPEL

BPMN (Business process modeling notation) Grafická reprezentace pro specifikaci byznys procesu. Podobné UML. Poskytuje mapování mezi grafickou notací a BPEL.

BPEL Definice obchodního procesu pomocí jazyka založeného na standardu XML. BPEL nedefinuje grafické znázornění procesů, ale standardizuje jeho XML definici. BPEL je jazyk pro popis WS kompozice.



Obrázek 4: BPMN diagram (pravý diagram je popisován v BPEL kódu)

```
<process name="HelloWorld" targetNamespace="http://jbpm.org/examples/hello"
  xmlns:tns="http://jbpm.org/examples/hello"
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
    <!-- establishes the relationship with the caller agent -->
    <partnerLink name="caller" partnerLinkType="tns:Greeter-Caller" myRole="Greeter" />
  </partnerLinks>

  <variables>
    <!-- holds the incoming message -->
    <variable name="request" messageType="tns:nameMessage" />
    <!-- holds the outgoing message -->
    <variable name="response" messageType="tns:greetingMessage" />
  </variables>

  <sequence name="MainSeq">
    <!-- receive the name of a person -->
    <receive name="ReceiveName" operation="sayHello" partnerLink="caller" portType="tns:
        Greeter" variable="request" createInstance="yes" />
    <!-- compose a greeting phrase -->
    <assign name="ComposeGreeting">
      <copy>
        <from expression="concat('Hello:',bpel:getVariableData('request','name'),'!')"/>
        <to variable="response" part="greeting" />
      </copy>
    </assign>
    <!-- send greeting back to caller -->
    <reply name="SendGreeting" operation="sayHello" partnerLink="caller" portType="tns:
        Greeter" variable="response" />
  </sequence>
</process>
```

Kód 1: BPEL

BPMN → BPEL → Application

12 Architektura zaměřená na služby (SOA). Kvalita, výkonnost a škálování služeb. Zabezpečení, integrita, bezpečnost, a autentifikace služeb. Point-to-point a end-to-end šifrování.

http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/

Výkonnost Výkonnost reprezentuje jak rychle služba zpracuje dotaz. To může být měřeno dobou odezvy, dobou spouštění, nebo doby jednotlivých transakcí.

Spolehlivost Schopnost poskytnout službu na požádání uživatelem a poskytovat ji po požadovanou dobu v rámci specifikovaných tolerancí a dalších daných podmínek.

Škálovatelnost Schopnost zvyšování výpočetní kapacity za účelem zvýšení výkonnosti a kapacity. Služba by měla být škálovatelná ve smyslu navýšení počtu operací a transakcí.

Kapacita Služba by měla být poskytována s požadovanou kapacitou, tzn. garantovaným výkonem při zadaném počtu (kapacitě) paralelních requestů.

Robustnost Robustnost značí schopnost správného fungovaní služby i při nevalidních (nekompetních, konfliktních) vstupech.

Zpracování výjimek Služby by měli být poskytovány s funkcionalitou zpracování chyb a výjimek.

Integrita Služba umí předcházet/bránit neopravněnému přistupu nebo modifikaci k datům. Jsou dva typy integrity: datová a transakční.

Přístupnost Přistupnost je schopnost obsluhovat klientské požadavky (requesty). Dostupnost se dá navýšit vytvářením škálovatelných služeb.

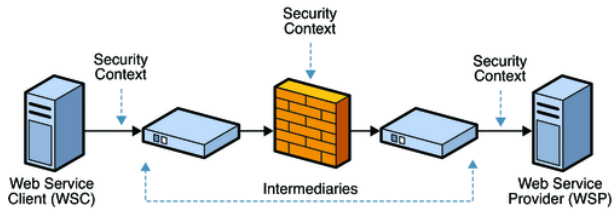
Dostupnost Chceme co nejvyšší dostupnost služby, jako parametr se udává Time-to-Repair (TTR).

Bezpečnost Service Security Performance - Odolnost proti neoprávněnému odposlechu, neoprávněnému použití, zlovolnému poškození, nesprávnému použití, lidským chybám a přírodním katastrofám.

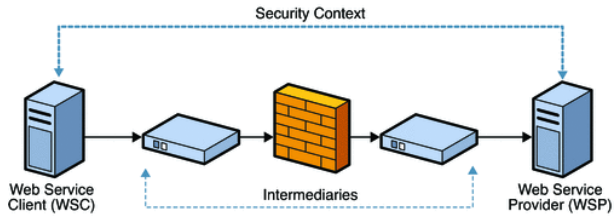
12.1 Bezpečnost

Poskytuje se bezpečnost klienta, serveru a zpráv. Bezpečnost zpráv je pokryta kryptografií (symetrické/asymetrické šifrování, digitální podpisy a certifikáty). U bezpečnosti klienta je řešeno: service discovery risk, phishing risk, autenticita a korektnost přijatých zpráv, autentikace poskytovatelů služeb. A zabezpečení serveru je řešení pomocí zabezpečení přenosového kanálu (HTTPS,SSL,SSH).

Point-to-point Communication channel level zabezpečení. Guarantied on the server basis



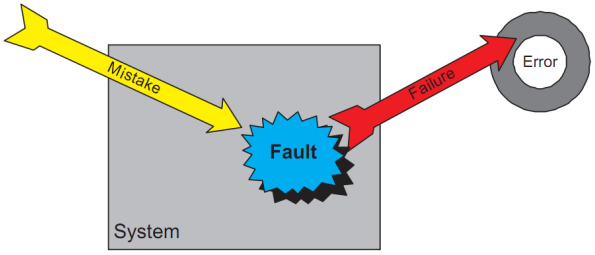
End-to-end Message level zabezpečení. Application oriented security



- **XML signatura** - Podepisuje se buď celý dokument, nebo jen část
- **XML šifrování** - end-to-end, v soap obálce je oddíl se security tokenem a signaturou.

13 Kategorizace SW chyb, optimalizace návrhu testů. Testování automatů.

SW chyba je prezentace toho, že program dělá něco nepředpokládaného. Je to míra toho, kdy program přestává být užitečný (lidem). Je to nesouhlas mezi programem a jeho specifikací (pouze tehdy, jestliže specifikace existují a jsou správné!).



- **Pochybení**: Akce člověka, která produkuje nesprávný výsledek.
- **Vada**: Nesprávný krok, proces nebo definice dat v počítačovém programu. Výsledek pochybení. Potenciálně vede k selhání.
- **Selhání**: Nesprávný výsledek. Projev vady.
- **Chyba**: Kvantitativní vyjádření toho, na kolik je výsledek nesprávný

13.1 Kategorie SW chyb

- Chyby UI - vstupy (nápověda), výstupy (rychlost odezvy)
- Chyby omezení - výjimky, výpočetní chyby
- Procesní chyby - chyba při první inicializaci, zátěž, souběh více vláken
- Chyby vedení - stará verze knihovny, slabá doc
- Chyby požadavků, vlastností a funkčnosti - neúplné, nejednoznačné
- Strukturální chyby - logika (neporozumění log. operátorům), GOTO, konverze typů
- Datové chyby - specifikace objektů
- Chyby implementace - syntaktické chyby, chyby paměti, hranice pole

Chyby uživatelského rozhraní

- Problémy s funkčností - program nedělá něco, co by měl dělat nebo to dělá nevhodně (zmatečně, neúspěšně), lze některé operace provést obtížně.
- To co se „předpokládá“ od programu, žije pouze v mysli uživatele
- Všechny programy mají problémy s funkčností vzhledem k různým uživatelům.
- Funkční problém je chybou, pokud očekávání uživatele jsou rozumná.
- **Vstupy**

  - Jak lze nalézt, jak program používat (jaká je nápověda)?
  - Jak snadné je ztratit se v programu?
  - Jaké chyby uživatel dělá a kolik ho to stojí času?
  - Co mu chybí?
  - Nutí prg. uživatele přemýšlet nepřirozeně?

- **Výstupy**

  - Rychlost je základ interaktivního sw.
  - Cokoli co budí dojem pomalého prg. je problém.
  - Získá uživatel, co potřebuje?
  - Mají výstupy smysl?
  - Může uživatel přizpůsobit výstup potřebám?
  - Zle výstup přesměrovat dle potřeby(monitor, tisk, soubor, ...)?

Chyby omezení

- Chyby zpracování vyjímek zahrnující neschopnost

  - Předvídání možnosti chyby a bránit se jim
  - Zpracování podmínek chyby
  - Zpracovaní detekované chyby různými způspby

- Chyby hraničních podmínek

  - Nejjednodušší hranice jsou numerické
  - Mezní nároky na paměť, za kterých program může pracovat

- Výpočetní chyby

  - Chyby aritmetiky jsou časté a obtížně detekovatelné
  - Program ztrácí přesnost během výpočtu vlivem zaokrouhlovacích chyb a ořezání
  - Výpočetní chyby způsobené chybnými algoritmy

Procesní chyby

- Sekvenční

  - Počáteční a jiné speciální stavy
    - ◇ Funkce mohou selhat při prvním použití, např. chybějící inicializační informace či soubory
    - ◇ Nastaví se skutečně vše do výchozího bodu, vymažou se všechna data, jestliže uživatel provede reset programu?
  - Chyby řízení nastane, pokud program provede chybný příští krok

        - ◇ Extrémní chyba nastane, pokud se program zastaví nebo vymkne kontrole řízení

- Paralelní
    - Chyby souběhu - race error
        - ◇ Jedny z nejméně testovaných
        - ◇ Nastávají v multiprocesových sys. a integračních sys.
        - ◇ Velmi obtížně se opakují
    - Zátěžové podmínky
        - ◇ Program se začne chovat chybně pokud se přetíží
            - ∗ Chyby velkého objemu - hodně práce za dlouhou dobu
            - ∗ Chyby velkého stresu - hodně práce v daném okamžiku
        - ◇ Všechny programy mají své limity, ale je důležité vědět co nastane

Chyby vedení

- Hardware - program posílá chybová data na zařízení, ignorují chybové kódy přicházející zpět a zkouší použít zařízení, která neexistují nebo jsou aktuálně vytížená
- Řízení zdrojů a vedení
    - Staré problémy se opět objevují, pokud programátor zakomponuje do programu nějakou starou verzi komponenty
    - Ujistěte se, že program má správný copyright, vstupní obrazovky a

  čísla verzí
- Dokumentace - slabá dokumentace může způsobit ztrátu víry uživatele, že sw. pracuje spráně
- Chyby testování
    - Chyby udělané testy jsou nejčastějšími chybami objevenými během testování
    - Jestliže program navádí většinu uživatelů ke způsobení chyb, pak je špatně navržen

Chyby požadavků, vlastností a funkčnosti

- Požadavky a specifikace
    - Neúplné, nejednoznačné, vzájemně si odporující
    - Hlavní zdroj drahých chyb
- Chyby vlastností - chybějící, chybné, nevyžádané vlastnosti
- Interakce vlastností - nepredikované interakce(přesměrování telefonu ve smyčce)
- Preventivní opatření proti chybám ve specifikacích a vlastnostech:
    - Problém v komunikaci člověk-člověk
    - Jazyk formálních specifikací poskytující krátkodobé řešení, avšak neřeší problém chyb v dlouhodobém horizontu.

Strukturální chyby

- Chyby v řízení sekvencí
  - Příkaz GOTO
  - Většina chyb řízení (v novém kódu) se dá snadno testovat a je chycena během testování jednotek
  - Neupravený starý kód může mít řadu chyb v řídicím toku
  - Stlačování za účelem kratšího prováděcího času nebo menšího nároku na paměť je špatná politika
- Chyby zpracování
  - Zahrnuje chyby vyhodnocení aritmetických, algebraických nebo matematických fcí. a výběr algoritmu
  - Řada problému vznikne špatnou konverzí dat na jinou reprezentaci
- Chyby logiky
  - Neporozumění jak se selekční či logické operátory chovají samostatně nebo v kombinacích
  - Neporozumění sémantice uspořádání logických výrazů a jeho vyhodnocení specifickými překladači
  - Chyby datového toku - nevztahují se k chybám řízení
  - Chyby toku řízení - část logického výrazu, která je použita pro ovládání toku řízení
- Inicializační chyby - opomenutí inicializace pracovního prostoru, registů, nebo části dat
- Chyby a anomálie toku dat
  - Nastane pokud existuje cesta, při které se udělá s daty něco neodůvodněného např. použití neinicializované proměnné, nebo neexistující proměné
  - Jsou stejně tak důležité jako anomálie toku řízení

Datové chyby

- Lze je nalézt ve specifikacích datových objektů, jejich formátů, počtu nebo jejich počátečních hodnotách
- Sw. se vyvíjí k tabulkám obsahujících řídicí a procesní fce.
- Trendy programování vedou k zvýšenému používání nedeklarovaných, interních, speciálních programovacích jazyků
- Dynamické vs. statické
  - Protože efekt poškození dynamických dat se může projevit velmi vzdáleně od příčiny, nalézají se takovéto chyby velmi obtížně

  - Základní problém zbytků ve sdílených zdrojích (např. vyčištění po použití uživatelem, sdílené čištění pomocí ovladače zdrojů, žádné čištění)

- Informace, parametr, řízení
  - Údaj plní jednu ze tří rolí: jako parametr, jako řízení, jako zdroj informace
  - Informace je obvykle dynamická s tendencí lokality pro danou transakci (nedostatek ochranného kódu validace dat)
  - Neadekvátní validace dat často vede k ukazování prstem

- Obsah, struktura, atributy
  - Obsah - aktuální bitový vzor, řetězec znaků, nebo číslo vložené do datové struktury
  - Struktura - velikost, tvar a počty popisující datové položky
  - Atributy - specifikace významu (sémantika)
  - Základem je explicitní dokumentace obsahu, struktury a atributů všech datových objektů

Chyby implementace

- Chyby kódování
  - Dobrý překladač chytne syntaktické chyby, nedeklarovaná data, procedury, kód a mnoho inicializačních problémů
  - Častou chybou kódu jsou dokumentační chyby (komentáře)
  - Úsilí programování je dominováno údržbou

- Chyby paměti
  - Charakteristiky
    - ◇ Nejobtížnější chyby z hlediska lokalizace
    - ◇ Nejdůležitější chyby z hlediska opravy
    - ◇ Projevy nesprávného obsahu paměti jsou nepredikovatelné
    - ◇ Chyby v obsahu paměti se typicky projevují vzdáleně od jejich příčiny
    - ◇ Chyby zůstávají často nedetekováné dokud nejsou náhodně spuštěny
  - Typy chyb
    - ◇ Chyby hranic polí
    - ◇ Přístup přes nedefinovaný ukazatel
    - ◇ Čtení z neinicializované paměti
    - ◇ Chyby ztráty paměti (memory leaks)
  - Slabá místa výkonnosti
    - ◇ Kolekce vyčerpávající přesné množiny dat pro výkonnostní test programu a každé jeho komponenty (profilování)
    - ◇ Zaměření se na kritická data
    - ◇ Sběr správně vybraných dat

* Řádka - kolikrát proběhla každá řádka - nejpřesnější, ale nejnáročnější na sběr dat
* Funkce - méně podrobné než předchozí
* Čas - data se sbírají z údajů časovaných běhů funkcí. Data jsou správná pro daný běh, ale závisí na stavu mikroprocesoru a paměti. Nejméně náročný sběr

13.2 Optimalizace návrhů testů

Optimalizovat testování chceme za účelem snížení jeho časové náročnosti. Např. u PLC chceme testovat instrukci, která má 5 parametrů a každý z nich může nabývat 17ti hodnot. To je 17^5 kombinací (1 419 857) tedy cca 230 dnů testování.

V praxi se ale ukázalo, že většina chyb nevzniká při všech 5ti nastavených parametrech, ale při interakci jen několika z nich. Vytvářejí se pak testy, které pokrývají jen všechny možné k-tice (ne všechny kombinace). Pomocí následujících metod se dá počet testovaných případů redukovat.

13.2.1 Princip párového testování

Jak již bylo zmíněno, při **ideálním testovacím plánu** by bylo nutné projít všechny kombinace hodnot parametrů, tj. $1419857 = 17^5$ kombinací - 230 dnů testování.

Praktický testovací plán využívá toho, že selhání jsou způsobena interakcí pouze několika parametrů, Proto se provádí testování kombinacemi pokrývající všechny možné k-tice (k-tice komprimovány do plných kombinací).

Příklad

- 5 parametrů
- 17 hodnot pro každý parametr
- **Optimalizace:**
- Nezávislé parametry: 17 kombinací,
- Počet párů: 5 nad 2 = 10 parametrových párů,
- Hodnot pro každý pár: $289 = 17^2$ kombinací hodnot pro každý pár,
- Počet hodnot pro všechny pár: $2890 = 289 * 10$ párů parametrových hodnot,
- zredukovaných 289 kombinací obsahující všechny páry,
- 30 minut testování.

13.2.2 Optimalizace metodou ortogonálních polí

Ortogonální pole je matice velikosti $m \times n$, ve které při testování parametrických párů jednotlivé sloupce odpovídají faktorům (parametrům funkce, případně jiným hodnotám, na kterých závisí testovací případ) a řádky jednotlivým testovacím případům.

Ortogonální pole jsou označována podle toho, kolik úrovní (významných hodnot) umožňují jednotlivým faktorům. Ortogonální pole $(2^1 \times 3^7)$ má celkem 8 faktorů. Jeden faktor může mít dvě významné hodnoty, a sedm dalších může mít až 3 významné hodnoty.

Pro použití této techniky je zapotřebí příklad nejprve zakódovat. Tato procedura je velmi snadná – nejprve nalezneme všechny faktory (parametry příkladu) a zjistíme, kolik mají úrovní. Poté nalezneme vhodné ortogonální pole.

Vybrané pole musí umožňovat alespoň tolik faktorů, kolik má náš případ. Dále je nutné, aby jednotlivé faktory měly dostatečný počet úrovní.

Nyní můžeme přejít k samotnému kódování. Každému parametru naší úlohy přiřadíme jeden sloupec – případné přebytečné sloupce můžeme ignorovat. Každé významné hodnotě přiřadíme jednu úroveň – pokud má sloupec více úrovní než potřebujeme, tak můžeme některé významné hodnotě přiřadit více úrovní, tím tuto hodnotu otestujeme více než ostatní (ale stále platí, že otestujeme každou parametrickou dvojici alespoň jednou).

V tomto okamžiku již můžeme pomocí našeho zakódování číst jednotlivé řádky ortogonálního pole - samotné testovací případy.

Ortogonální pole není jednoduché nalézt, testovací případy musí být balancované atd., proto existují jejich katalogy (pole nekonstruujeme, ale používáme již dostupná pole z katalogů).

	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Tabulka 1: Ortogonální pole $L_4(2^3)$

13.2.3 Optimalizace metodou latinských čtverců

Latinský čtverec je matice s n řádky a n sloupci taková, že každý element obsahuje celé číslo do 1 do n tak, že se v žádném sloupci nebo řádku nevyskytuje žádné číslo vícekrát než jednou.

Párově ortogonální latinské čtverce - každý elment v daném čtverci vystupuje v relaci s každým elementem druhého čtverce právě jedenkrát. Pro jakoukoliv mocninu prvočísla n existuje $n - 1$ párově ortogonálních latinských čtverců velikosti $n \times n$.

- ze zadání se identifikují faktory a úrovně
- určím počet párově-ortogonálních čtverců jako N = max(faktorů-2, nejvyšší_úroveň)
- najdu si čtverce v literatuře, $(N + 1)$ rozměrné
- nyní generuji testovací případy: první faktor je číslo řádku, druhý faktor číslo sloupce, další hodnoty podle čísel na souřadnicích v jednotlivých čtvercích
- vygeneruje mi to $(N + 1)^2$ testovacích případů

13.3 Testování automatů

Konečný automat je výborný model pro testování aplikací řízených pomocí menu (ovládání aplikace se provádí pomocí výběru položek z menu).

Vrcholy zobrazují stavy (stav aplikace). Hrany mezi stavy reprezentují vyběr položky menu (a tím přechod do dalšího stavu). Automat by měl být silně souvislý, pokud existují oddělené stavy (izolované), tak tyto stavy zavání chybou, jsou podezřelé (nevede k nim položka v menu).

Skryté stavy Test nelze zahájit, pokud systém není potvrzením způsobem v počátečním stavu. Dát si pozor na skryté stavy. Při testování SW můžeme předpokládat věci, které nemusí obecně platit (např. že víme, ve kterém stavu se systém nachází). Pokud existují skryté stavy tak se typicky nejedna o jeden či dva stavy, ale o celý stavový prostor.

Postup návrhu testů

- identifikuj vstupy
- definuj kódy vstupu
- idenfikuj stavy
- definuj kódovaní stavů
- identifikuj výstupní akce
- definuj kódování výstupních akcí
- specifikuj tabulku přechodů a tabulku výstupů - a zkontroluj ji
- navrhni testy
- proveď testy
- pro každý vstup ověř jak přechod tak i výstup

Každý test začíná v počátečním stavu. Z počátečního stavu se systém přivede nejkratší cestou k vybranému stavu, provede se zadný přechod a systém se nejkratší možnou cestou přivede opět do počátečního stavu (tzv. okružní cesta).

Formální konstrukce testů Nechť L je **množina vstupních sekvencí** a q, q' dva stavy. L **rozliší** stav q od q', jestliže existuje sekvence k v L taková, že **výstup** získaný apikací k na automat ve **stavu** q je **různý** od **výstupu** získaný aplikací k na **stav** q'. Chceme minimální automat (tj. takový, který neobsahuje redudantní stavy).

Množina vstupních sekvencí W se nazývá **charakterizační množina**, jesltiže může rozlišit jakékoliv dva stavy automatu.

- **Pokrytí stavů** je množina vstupních sekvencí L taková, že lze nalézt prvek množiny L, kterým se lze dostat do jakéhokoliv žádaného stavu z počátečního stavu q_0.
- **Pokrytí přechodů** minimálního automatu je množina vstupních sekvencí T, která je pokrytím stavů a uzavřená z hlediska pravé kompozice s množinou vstupů $Input$.

$$seq \in T = L \bullet (Input \cup <>)$$

14 Testovaní metodami bílé a černé skříňky. Strukturální, statická a dynamická analýza. Analýza datových toků. Testování objektově orientovaného softwaru.

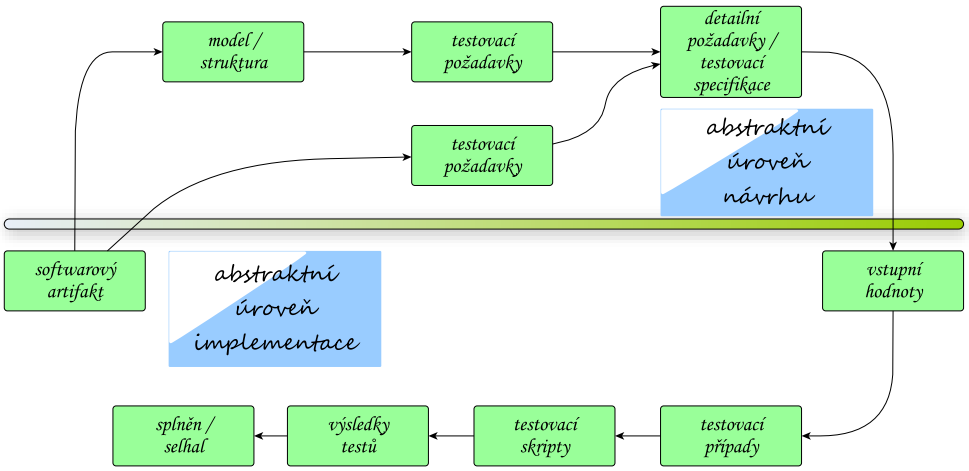
14.1 White/black box testování

Při **testování černé skříňky** se zaměřujeme na vstupy a výstupy programu bez znalosti, jak je naimplementován. Produkt je černou skříňkou, do které se nelze podívat, vidíme jen jak vypadá a jak se chová navenek. Smyslem je analyzovat chování softwaru vzhledem k očekávaným vlastnostem tak, jak ho vidí uživatel. Do této kategorie spadají skoro všechny druhy **testů uživatelských rozhraní**, akceptační testy, testování podle scénářů, které krok za krokem provádějí uživatele tím, co má zadat a jaké jsou očekávané reakce systému.

Při **testování bílé skříňky**, má tester přístup ke zdrojovému kódu a testuje produkt na základě něj. Vidí nejen co se děje na povrchu skříňky, ale i vnitřní reakce systému. Testuje se tak, aby se pokryly všechny cesty a okrajové hodnoty. Třeba *unit testing*. Počet testovacích případů se dá snížit pomocí ortogonálních polí, latinských čtverců.

14.2 Strukturální analýza

Strukturální analýza je převedení programu na model (typicky graf). V grafu se pak dají zobrazit toky řízení, toky dat, stromy závislosti volání funkcí, či grafy konečných automatů.



Návrh testování podle modelu

- definuj graf, definuj relace
- navrhni testy pro pokrytí uzlů
- navrhni testy pro pokrytí hran
- otestuj všechny atributy
- navrhni testy smyček

Kritéria pokrytí

- **Pokrytí řádek** - požaduje provedení každé řádky kódu alespoň jednou
- **Pokrytí větví** - znamená, že podmínka každého větvení musí být alespoň jednou pravdivá a jednou nepravdivá
- **Pokrytí podmínek** - zkontroluje všechny možné způsoby, za kterých daná podmínka je pravdivá či nepravdivá
- **Úplné pokrytí cest** - vyžaduje provedení všech možných různých úplných cest (v praxi neproveditelné)

14.3 Statická a dynamická analýza

Statická analýza je model checking (viz následující otázka) a testování konečného automatu.

Analýza kódu, hledájí se v něm logické chyby jako např. nekonečný cyklus, nepoužívané proměnné, přiřazení místo porovnání atd. Řeší to např. pluginy do IDE (nebo IDE samotné).

Dynamická analýza je test běžící aplikace AUT=application under test. Třeba checkování použití paměti (Rational Purify, Boundchecker)

14.4 Analýza toků řízení (metoda hlavních cest)

Jednoduchá cesta Cesta z n_i do n_j na které se žádný uzel neobjevuje více jak jedenkrát s vyjímkou, že počáteční a koncový uzel mohou být identické.

Hlavní cesta Cesta z n_i do n_j je hlavní, jestliže je to jednoduchá cesta a není žádnou vlastní podcestou jakékoliv jiné jednoduché cesty (tj. je maximální).
Cílem je nalézt hlavní cesty. **Princip algoritmu:**

- Nalezni cesty délky 0 (uzly).
- Kombinuj cesty délky 0 do cest délky 1 (hrany).
- Kombinuj cesty délky 1 do cest délky 2.
- atd.

14.5 Analýza datových toků (metoda du-cest)

Tok datových hodnot: testy zajišťující, že hodnoty vzniklé na jednom místě jsou použity správně na jiných místech.

- **Definice** *def*: místo, kde je hodnota proměnné uložena do paměti.
- **Užití** *use*: místo, kde se přistupuje k hodnotě proměnné.
- **DU páry def-use:** asociace určující přenosy hodnot.

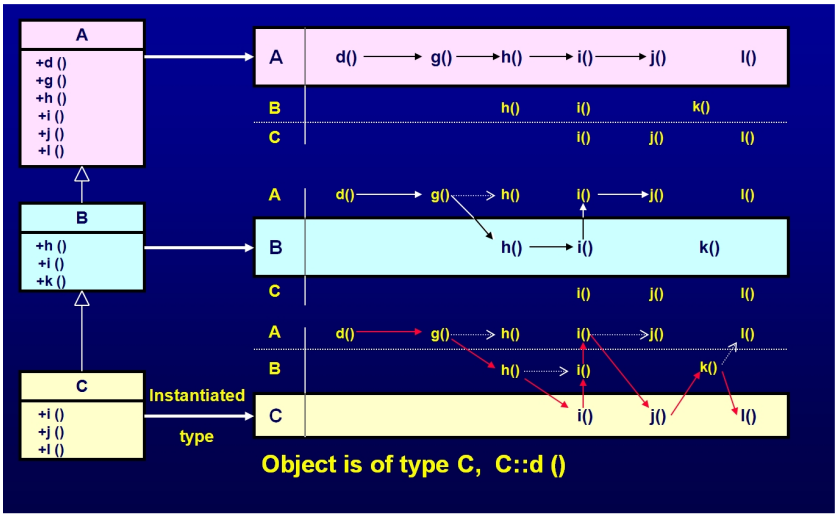
Formalizace

- V : množina proměnných asociovaná se softwarovým artefaktem.
- def : místo, kde je hodnota proměnné uložena do paměti
- use : místo, kde se přistupuje k hodnotě proměnné
- def (n) = podmnožina množiny proměnných V, které jsou definovány uzlem n
- use (n) = podmnožina množiny proměnných V, které jsou použity v uzlu n
- du(n, v) je množina všech du-cest vzhledem k proměnné v, která začíná v uzlu n

14.6 Testování OO softwaru

Anomálie DU párů Anomálie souvisí s děděním a polymorfismem.
Jsou třídy A ← B.

- **ITU** - inconsistent type use - nepřepisování ale volání parent metod, nejdřív použiju metody B, pak nějaké z A a pak opět B (A metody mohou něco nečekaně přepsat - přivézt objekt do stavu nekonzistentním s B).
- **SDA** - state definition anomaly - přepisící metoda má jinou def množinu než přepisovaná (přepisující metody v B nenadefinují některé proměnné, které jsou definovány přepsanými metodami v A).
- **SDIH** - podobné SDA ale potomek přepíše něco v prarodiči a rodič pak používá špatně definovanou proměnnou.
- **ACB1** - přepis konstruktoru, potomek pak používá něco co není definováno.
- **SVA** - přepsání metody v prarodiči, později rodič taky přepíše tu metodu a potomek začne volat rodiče, ne prarodiče



Obrázek 5: Problém polymorfismu

Testování párových sekvencí Typy testování: intra/inter metod a intra/inter tříd

Používá se k reprezentace interakce stavových prostorů, identifikace bodů integrace a testovacích požadavků. V softwarových artefaktech se hledají vazby last-def a firt-use po volání metody a uvnitř metody.

Testumeme, jak na sebe vzájemně působí metoda a instance vázaná na objekt o. Uvažují se metody které mohou být ve skutečnosti provedeny, testují se všechny vazby se všemi typy.

15 Formální specifikace programu. Verifikace pomocí metod automatického dokazování a metody model-checking.

Cíle kladené na požadavky specifikace. Musí být demonstrováno že požadavky jsou správné, úplné, přesné, konzistentní a testovatelné. Díky takto nadefinovaným požadavkům můžeme SW vefirikovat

Formální verifikace je technika založená na formálních metodách. Využívá matematicky založené jazyky, které umožňují **specifikaci** a **verifikaci** systémů:

- specifikace = zapsání požadavků na systém v matematickém jazyce
- verifikace = formální důkaz toho, že splňuje požadavky

Na vstupu dostáváme (matematický) model systému (M) a specifikaci požadavků kladených na systém v podobě formulí φ určité temporální logiky.

Verifikace je pak ověření, že systém splňuje specifikaci. Tzn. rozhodnutí, zda-li M je modelem formule φ, tj. $M \models \varphi$.

Techniky verifikace

- Statická analýza = ověření chování programu, aniž by se musel spustit
  - Abstraktní statická analýza (např. analýza ukazatelů v modern. kompilátorech)
  - Verifikace modelů = úplné procházení dosažitel. stavů programu
  - Omezená verifikace modelů = viz. předchozí, ale jen do určité hloubky
- Dokazování vět = nalezení důkazu vlastnosti, kdy systém i jeho vlastnosti jsou vyjádřeny jako formule v nějáké matematické logice

Verifikace pomocí automatického dokazování je formální důkaz, že systém splňuje formální specifikaci. Temporální verifikace - je potřeba dokázat dosažitelnost, bezpečnost, živost. Nebo např. statická analýza (např. analýza ukazatelů v moderních kompilátorech)

15.1 Model-checking

Verifikace pomocí model-checking je budování konečného modelu (automatu) systému a kontrola vlastností úplným prohledáváním stavového prostoru. Může dojít k explozi stavů. Výhody: automatizace, rychlost, produkce protipříkladů při nesplnění.

Způsoby verifikace modelů

- **Temporální verifikace modelů:** Použití temporální logiky (vyjádření času). Systémy modelovány jako přechodové systémy s konečným počtem stavů
- **Automatový přístup:** Specifikace i model vyjádřen jako automaty, Oba automaty se porovnávají.

Stavový prostor: Je formulován pomocí **atomických výroků** a **Kripkeho struktury**.

- **Atomický výrok** = základní tvrzení popisující daný systém (výrazy, konstanty, predikátové symboly). Je algoritmicky rozhodnutelný na základě daného stavu (ohodnocení všech proměnných)
- **Kripkeho strukrura** = typ nedeterministického konečného automatu. Kripkeho struktura je trojice (S, T, I), kde S = konečná množ. stavů $T \subseteq S \times S$ je přechodová relace $I : S \to 2^{AP}$ je interpretace množiny atomických propozic.
  - **Rozšířená Kripkeho struktura** je čtveřice (S, T, I, s_0), kde (S, T, I) je Krip. Struktura a s_0 je počáteční stav.
  - **Kripkeho přechodový systém** je pětice (S, T, I, s_0, L), kde (S, T, I, s_0) je Rozšiř. Krip. Struktura a $L : T \to Act$ je značkovací funkce.

15.1.1 UPPAAL

UPPAAL je nástroj pro modelování, simulaci a verifikaci systémů. Systém se modeluje jako nedeterministické automaty s hodinami.

Komponenty

- **Systémový editor** - jazyk nedeterministických podmíněných příkazů, jednoduché datové typy (ohraničená celá čísla, pole, atd.), sítě automatů s hodinami a datovými proměnnými.
- **Simulátor** - grafická vizualizace a záznam možného chování systému (vyšetřováí možných dynamických běhů systému), detekce vad modelu před jeho verikací, možnost vizualizace trasy generované verifikátorem (umožňuje analýzu záznamu běhů vedoucích k nežádaným stavům),
- **Verifikátor** - prověření všech možností dynamického chování modelu, kontrola invariantů a živosti prohledáváním stavového prostoru, dosažitelnost symbolických stavů reprezentovaných omezeními.

15.1.2 Temporální logika

Temporální **logika** je odvětví logiky, které zkoumá logickou strukturu výroků **o čase**, s nimiž se klasická výroková nebo predikátová logika nedokáže plnohodnotně vypořádat.

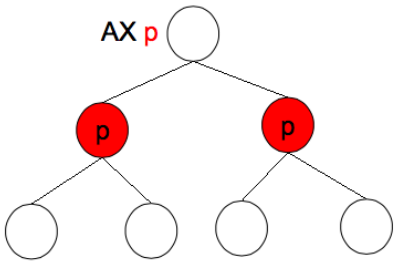
Abstrakce času Logický čas - pracuje s částečným uspořádáním stavů/událostí v chování systému. Fyzický čas - měření doby uběhlé mezi dvěma stavy/událostmi.

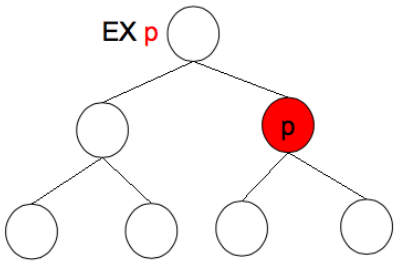
Čas ve verifikaci modelů Lineární čas - dovoluje se vyjadřovat pouze o dané lineární trase. Větvící se čas - dovoluje kvantifikovat možné budoucnosti počínaje daným stavem.

Ověřované vlastnosti v temporální logice

- **Dostažitelnost**: existuje taková cesta, kde bude podmínka splněna. $E[\ ]\ p$
- **Bezpečnost**: vlastnost, která nesmí nikdy nastat $A[\ ]\ p = E <> neg(p)$
- **Živost**: Nakonec se automat dostane do nějakého stavu $A <> p$.

- **CTL*** logika se skládá s atomických výroků, logických spojek, kvantifikátorů $(\exists, \forall)$, temporálních operátorů (ne**X**t, **F**uture, **G**lobally, **U**ntil, **R**elease).
- **CTL** logika je podmnožinou CTL*. Běhové formule jsou omezeny na $X\varphi, F\varphi, G\varphi, \varphi U\psi$ a $\varphi R\psi$ (kde φ a ψ jsou stavové formule). (kvantifikátor-temp.operátor musí být v párech)
- **LTL** logika je podmnožinou CTL*, která obsahuje jenom kvantifikátor A (pro všechny)
- **Z notace** je založena na teorii množin a predikátový kalkulus prvního řádu.
- **PVS** je specifikační jazyk implementován v Common Lisp





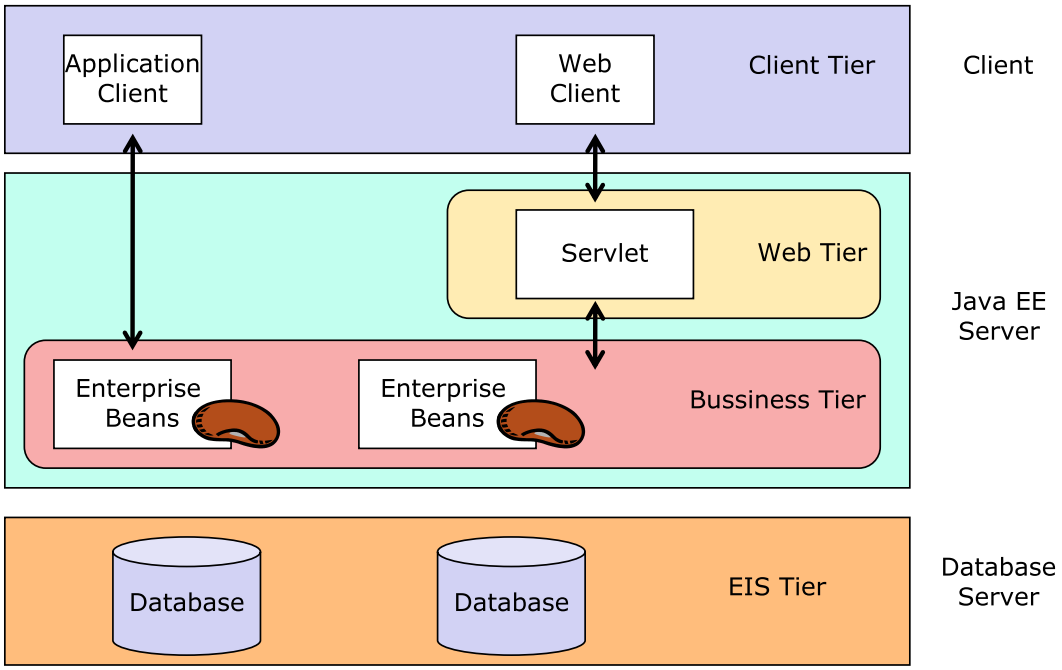
16 Architektura Java EE, funkce jednotlivých vrstev, životní cyklus standardizovaných komponent Java EE, návrhové vzory využitelné v architektuře webové aplikace.

Distribuce Javy se liší podle jejího zamýšleného použití:

- **Java ME** (*Java Micro Edition*) pro mobilní aplikace, omezený rozsah funkcí.
- **Java SE** (*Java Standard Edition*) pro desktopové aplikace.
- **Java EE** (*Java Enterprise Edition - J2EE*) pro webové aplikace.

16.1 JAVA EE

- Komponentový přístup.
- Komponenty mohou být distribuované na různých strojích (klient, server, DB).
- Aplikace rozdělena do vrstev.
- serverová část aplikace bývá nasazena na aplikačním serveru, který umožňuje zpracování více požadavků naráz (multi-threading). Nasazení znamená zabalení celého projektu do např. WAR archivu, který obsahuje vše potřebné (zdrojové třídy, statické resources, použité knihovny).
- Podpora JNDI, java beans, JAAS (autentizace), JMS (Java Messaging Service), servlety, transakce (JTA – Java Transactions API).



MVC Většinou je aplikace navržena podle vzoru MVC. Jedná se o konkrétní aplikaci vzoru oddělení zodpovědností (*separation of concerns*), který předepisuje vysokou kohezi jednotlivých komponent. Každá komponenta by měla mít vysoce soudržnou sadu zodpovědností a všechny ostatní požadavky by měla delegovat komponentám, které jsou úzce specializované zase na jinou činnost.

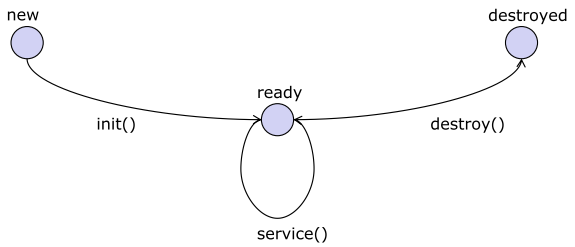
MVC tedy odděluje zodpovědnosti za data (model), pohled na data (view) a manipulaci s pohledem na data (controller). Aplikace je rozdělena na tři vrstvy: datovou, prezentační a ovladače (controllers). Tyto tři vrstvy jsou na sobě nezávislé a umožňují snadné vyjmutí jedné vrstvy a nahrazení jinou implementací.

16.2 Servlet

Servlet je třída, která rozšiřuje schopnost webserveru. **Zpracovává** a odpovídá na **požadavky** z webových klientů, typicky HTTP requestů. Třída musí implementovat rozhraní `javax.servlet.Servlet`, které definuje metody životního cyklu vyžadované servletovým kontejnerem. Při použití na webu servlet typicky rozšiřuje třídu `javax.servlet.http.HTTPServlet`, která definuje metody na zpracování jednotlivých HTTP požadavků: `doGet()`, `doPost()`, atd.

```
public class MujServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        res.getWriter().write("<html>Text ve formatu HTML</html>");
    }
}
```

Životní cyklus Aplikační server pro každý dotaz vytvoří nový servlet a zavolá nejdříve metodu `init()`. Pak následuje obsluha požadavku metodou `service()` a nakonec je po zavolání `destroy()` servlet zničen. Programátor může poskytnout vlastní implementaci metod `init()`, `service()` a `destroy()`.



JSP (Java Server Pages) JSP je textový dokument obsahující statické (X)HTML tagy a JSP tagy, které umějí generovat dynamický obsah. JSP umožňuje používat kusy Java kódu v HTML stránce (scriptlet).

Za tím vším se ale skrývá servlet - **JSP jsou servlety**. JSP se při prvním použití kompiluje a vytvoří se Servlet, který dělá to, co uměla původní JSP stránka (např. do metody `doGet()` se „vyprintí“ celý obsah toho souboru).

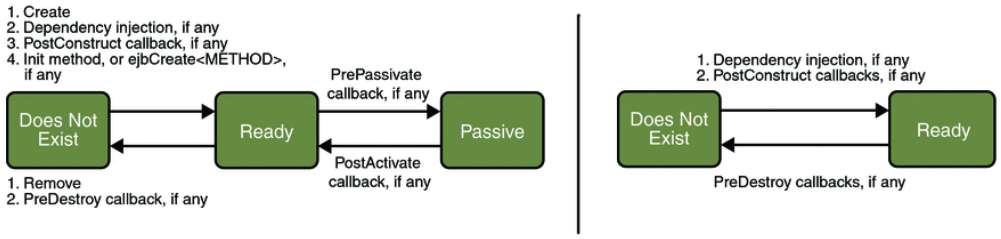
Deskriptor Deskriptor je konfigurační soubor s názvem `web.xml`, který musí být v každé webové aplikaci. Obsahuje mapování servletů a filtrů na requesty, kódování JSP stránek, parametry servletů a stanovuje, která stránka se zobrazí jako první (welcome-file).

16.3 Java Bean

Bean implementují aplikační logiku nebo vystupují v roli entit cílové domény (entita zákazník, výpůjčka...). Bean může mít lokální nebo vzdálené rozhraní podle toho, jestli se nachází na stejném stroji, nebo je na jiném stroji. Pak jsou také **EJB** (Enterprise Beans), to jsou komponenty běžící na aplikačním serveru, které jsou manažovány EJB kontejnerem. Bean je několik druhů:

- **stateful** - Uchovává kontext pro každého klienta zvlášť. Je náročnější a pomalejší, proto bychom měli používat stateless, kdekoliv je to jen možné.
- **stateless** - Jednodušší, nepamatuje si kontext volání klienta.
- **message driven** - messaging (základní účel pro distribuované systémy).

Životní cyklus bean https://docs.oracle.com/javaee/6/tutorial/doc/giplj.html



Životní cykly: stateful vs. stateless bean

Návrhové vzory využitelné v architektuře webové aplikace MVC, DI, Factory, Service Locator, cokoliv

16.3.1 Inversion of Control (IoC)

IoC, nebo také obrácené řízení, je návrhový vzor, který umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami. DI je jedna z možností implementace IoC. V klasickém modelu programování vytváříme nějakou třídu, která pak využívá další třídy a tak dále. Tím jsou potom třídy velmi pevně svázány a změna používané třídy za jinou vyžaduje zásah do kódu. IoC umožňuje uvolnit tuto vazbu, což nejjednodušeji vystihuje Hollywoodský princip „Nevolejte nám, my se ozveme.“. To je klasická odpověď, kterou dostávají amatérští herci, kteří se snaží získat v Hollywoodu nějakou roli ve filmu, seriálu apod. Proto tedy Hollywoodský princip. V podstatě to znamená, že třída nevytváří sama instance dalších tříd, které potřebuje, ale jsou jí dodány nějakým způsobem z vnějšku. Těchto způsobů existuje několik, nejznámější jsou tři:

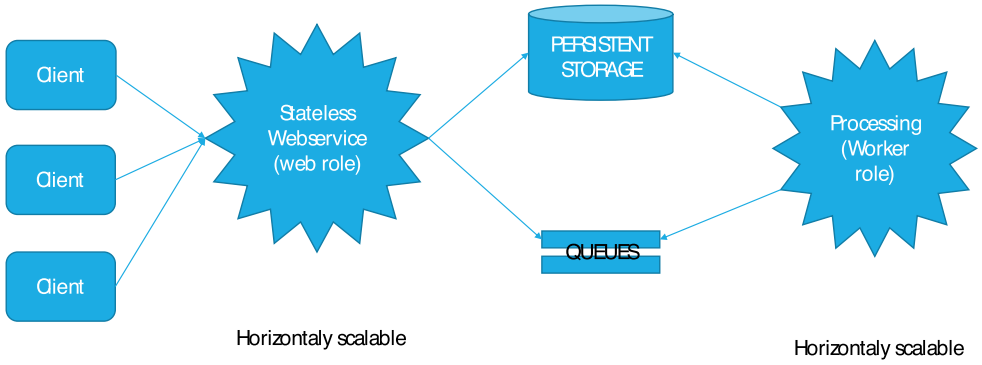
- **Constructor Injection** - třídy, do nichž je vkládána instance další třídy, kterou potřebují, musejí mít vytvořen konstruktor, který umí přijímat potřebné typy objektů.
- **Setter Injection** - třídy, do nichž jsou vkládány instance, musí mít definovány potřebné settery, setter injection používá například Spring Framework
- **Interface Injection** - nejprve je definováno rozhraní, které určuje metody, pomocí kterých budou nastavovány instance objektů, které příslušná třída potřebuje, každá třída, která chce tyto instance využívat, musí implementovat příslušné rozhraní.

17 Vhodnost nasazení jednotlivých webových architektur, sdílení dat, perzistence, webové služby a REST, asynchronnost, messaging.

Vhodnost nasazení jednotlivých webových architektur záleží na povaze projektu. Třeba kalkulačka nemusí být vůbec na webu, eshop může běžet na vlastním serveru někde na hostingu u kamaráda, bankovní aplikace by měla mít tlustého i tenkého klienta a s použitím enterprise funkcí jako J2EE, server na sdílení videa bude chtít svůj privátní cloud aby dobře škáloval.

Př: | Elektronické bankovnictví, které má 20 let starý back-end legacy system naprogramovaný v COBOLu. Klienti přistupují přes webové JEE (JSP) rozhraní. zvážit architekturu systému vzhledem k nárůstu počtu klientů nebo růstu bank, zvážit privátní cloud vs. existující provideři AWS, Google, MS Azure.

Typická architektura (cloudové) aplikace. Je mnoho klientů, kteří interagují například s webovou službou. Ta může být horizontálně škálovatelná (co do počtu serverů). Zde může být i nějaký load balancer, který klienty rovnoměrně přiřazuje méně vytíženým strojům. Služba buď zpracuje požadavek rovnou do databáze (storage), nebo pokud se jedná o nějakou (časově) náročnější akci (například převod fotek atd.), tak tu pošle do fronty ke zpracování. Z fronty si tzv. „worker role“ postupně berou jednotlivé úlohy a na pozadí je provádění (opět mohou být horizontálně škálovatelné).



17.1 Sdílení dat

Master Slave vs. High replication

17.2 persistence

Hlavně relační DB, protože objektové nejsou moc cool - nevýhody přímého použití JDBC

ORM

- Blíže OO paradigmatu

- JPA - standardizované API pro ORM: implementace např. Hibernate (konfigurace pomocí XML nebo anotacemi)
- ORM by měl být neintruzivní - pracuje přímo s POJO (implicitní konstruktor, settery a gettery, ne final atributy)
- podpora ISA (dědění) hierarchie:
  - SINGLE_TABLE - vše v jedné tabulce + rozlišující sloupec
  - TABLE_PER_CLASS - každá entita má vlastní tabulku s celou sadou atributů
  - JOINED - hlavní tab. má základní attrib., ostatní se s ní spojují (slabé entity)
- podpora vztahů (1:1, 1:N, M:N)
- dialekty SQL (snaha o sjednocení SQL syntaxe)
- perzistenci se starají třídy EntityManager nebo Hibernate Session - operace `persist, find, merge, remove, query`

17.3 WS a REST

A web service is a function that can be accessed by other programs over the web (Http). A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards (XML, SOAP, HTTP)

- architektonický styl pro distribuovaná media
- platformně nezávislé
- založeno na technologii HTTP a URI
- cíl: obecné aplikační rozhraní, možnost komunikace přes proxy - manipulace se zdroji (informace, data, soubory)
- zdroj je identifikován svým URI
- client-server (aplikační logika rozdělená mezi server a klient),
- stateless (bezestavová komunikace - jeden požadavek nese všechny informace které server potřebuje k jeho zpracování),
- pro popis metadat se používají HTTP hlavičky
- minimalizace závislostí klienta a serveru
- server může cacheovat odpovědi
- klient neví, zda komunikuje se serverem nebo prostředníkem (proxy, cache)

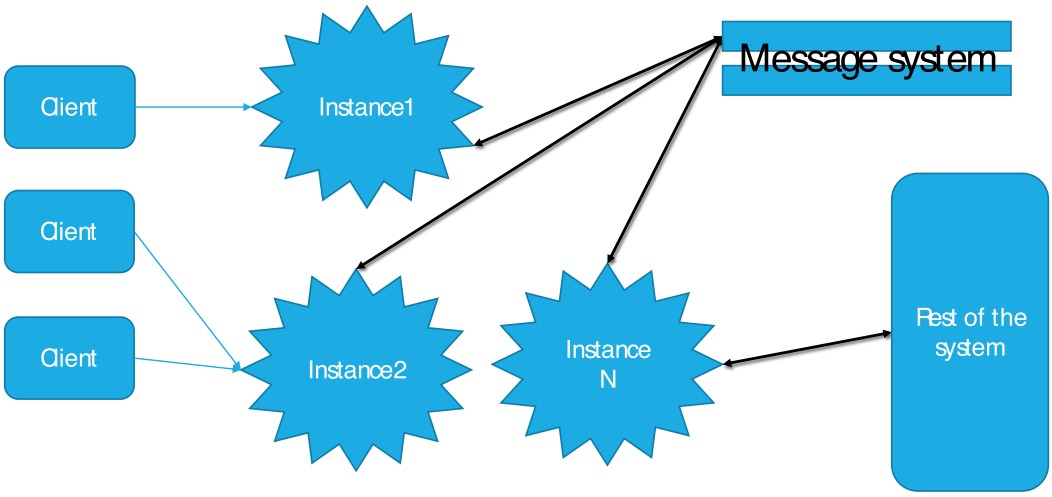
17.4 Messaging

Asynchronnost znamená neblokující volání, dá se řešit třeba AJAXem nebo messagingem.

Messaging může efektivně řešit škálování a load balancing. Na serveru je fronta zpráv, kterou si rozebírají jednotlivé servery. Dá se využít na schedulování úkolů co trvají dlouho (konverze videa).

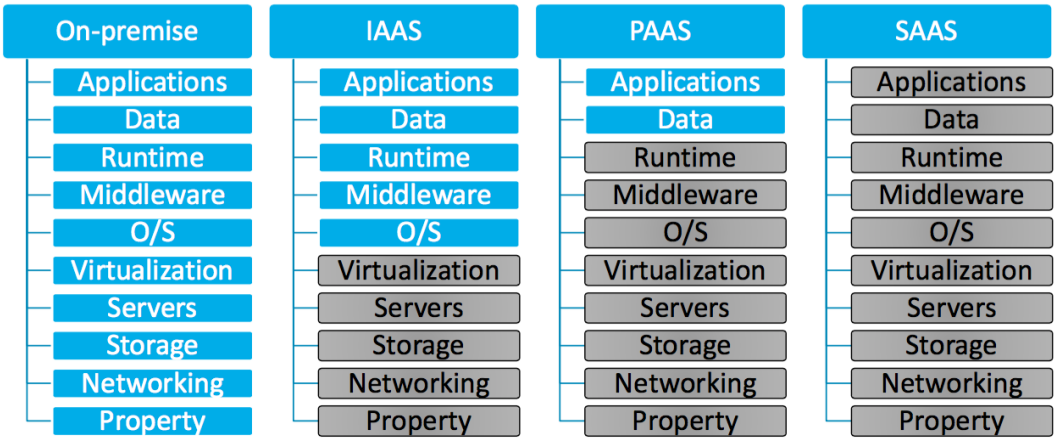
JMS (Java Messaging Service) dva módy: point-to-point (zprávy určené přímo někomu) nebo publish-subscribe (víc klientů se může subsribnout na nějaký topic)

Cloudové fronty: GAE Task Queue, AWS SImple Queue Service, MS Azure Queue.



19 Cloud architektury, virtualizace, různá pojetí cloudových řešení, omezení cloudových aplikací, náklady na provoz, vlastnosti aplikací vhodných pro nasazení v cloud architektuře.

Cloud-Computing architektury můžeme rozdělit do 3 hlavních vrstev:



19.1 Infrastructure as a Service (IaaS)

Je to vlastně outsourcing výpočetního vybavení (servery, HW, síťové komponenty, storage). Poskytovatel služby je odpovědný za housing, běh a správu těchto komponent. Klient obvykle platí v modelu *pay-as-you-go* či smluvním paušálem (předplatným), popřípadě kombinací obojího.

Příklady služeb IaaS: Amazon EC2, Amazon S3, GTS Managed Server - Virtual Server.

19.2 Platform as a Service (PaaS)

Provider poskytuje přístup ke computing platformě či computing stacku (množina softwarových subsystemů, které poskládáte do výsledné služby). Jako v případě IaaS máte k dispozici servery, storage, ale omezené vybranou platformou. Zatímco na IaaS můžete v rámci virtuálního prostředí provozovat téměř jakoukoliv aplikaci, volba platformy (operacniho systemu ci programovaciho jazyka) je plně v kompetenci klienta, u PaaS jste již vázáni kompletní platformou.

Nebezpečí Vendor Lock-In jste omezeni platformou poskytovate (.NET na MS Azzure), proprietární službou či množinou podporovaných programovacích jazyků. Flexibilita služby nemusí stačit rapidně se vyvyjejícím projektům (paměťové limity, model automatického deploymentu).

Příklady služeb PaaS Google App Engine (platforma: Java, Python, Go), MS Azure (platforma: .NET, Ruby, Java, PHP).

19.3 Software as a Service (SaaS)

Model pronájmu hotových aplikací, které jsou k dispozici klientovi skrze interface (nejčastěji kombinace REST+JSON) či uživatelské rozhraní (Facebook, YouTube, BaseCampHQ, Google Apps..).

Příklady aplikací SaaS Google Apps, E-mail, Kalendář, Microsoft Live

19.4 Omezení cloudu

- v **bezpečnosti** spoléháme na dodavatele cloudu
- **zneužití** - můžu si pronajmout cloud a udělat DDOS útok
- **legislativa** - např. některé evropské země nedovolují ukládání osobních informací mimo EU
- **copyright** - nesmíte přenášet copyrightované materiály mimo zemi licence

19.5 Náklady na provoz

Pay-As-You-Go - Platíte pouze za spotřebované zdroje (použité storage, vypočetní čas, traffic). Obvykle 0 upfront investment. Konsolidovaná fakturace (denní, týdenní, .. vyúčtování za použité služby na jednom účtu). Cena zahrnuje náklady na:

- hardware
- údržba
- utitilies
- případné SW licence

Vendor uplatnuje obvykle economies-of-scale (jenotkove mensi naklady na MB storage, traffic, hodinu procesoroveho casu...)

Paušál Můžete si **předplatit** určitý počet hodin procesorového času. Paušál za měsíční provoz virtual serveru atd..

19.6 Virtualizace

Virtualizace je technika (postup) přístupu ke zdrojům jiným způsobem, než jsou fyzicky zapojeny. Dá se virtualizovat **celý stroj** (VirtualBox, Vmware, Microsoft Virtual PC, Android emulátor), **jednotlivé komponenty** (např. RAID se navenek tváří jako jeden disk, virtuální paměť může naalokovat více paměti než je fyzicky dostupné, ...), jiný **systém** (Wine v Linuxu, NTFS driver v Linuxu), aplikace (JVM, .NET VM). Virtualizace je ekonomicky výhodná, centrálně spravovatelná, bezpečná (když uživatel něco provede naloadujeme čistý image), odděluje logiku od konkrétního hardware.

- emulace nebo simulace - VM simuluje celý hardware, dovoluje běh neupraveného OS na zcela odlišném hardware (Microsoft Virtual PC, AVD emulator pre Android)
- nativní (plná) virtualizace - VM simuluje dostatečné množství komponent, aby umožnil běh neupraveného OS (VMware, VirtualBox, Parallels)
- částečná virtualizace - VM simuluje instance mnoha prostředí, na kterém běží hostitel - Linux, MS Windows
- paravirtualizace - VM nesimuluje HW ale nabízí API (hypercall), přes které se komunikuje (Xen)
- virtualizace na úrovni operačního systému
- aplikační virtualizace

19.7 Vhodné aplikace pro cloud

Nové weby, které nejsou svázány konkrétní databází/technologií. Malé weby, které v cloudu můžou běžet i zadarmo. Globální weby, které potřebují hodně škálovat a mít nízkou latenci všude po světě. Aplikace které nejsou primárně sekvenční a využijí paralelizaci. Weby které potřebují pracovat s obrovskými daty. Eshopy, které se většinu roku flákají a pak před Vánocemi to přijde.

Reference

[1] PCWORLD. Co je to CORBA? aneb middleware dneška. Dostupné z: http://pcworld.cz/software/co-je-to-corba-aneb-middleware-dneska-15175.

[2] Přispěvovatelé wikipedie. Sémantika programovacích jazyků. Dostupné z: http://cs.wikipedia.org/wiki/Sémantika_programovacích_jazyků.

[3] ZEMEK, P. lambda kalkul rychle a pochopitelně. Dostupné z: http://publications.petrzemek.net/articles/PZ_-_IPP-Lambda-Kalkul.pdf.