Katedra počítačovej grafiky a spracovania obrazu

Fakulta matematiky fyziky a informatiky

Univerzita Komenského

Diplomová práca

Rostislav Šimoník

BRATISLAVA 2007

Katedra počítačovej grafiky a spracovania obrazu

Fakulta matematiky fyziky a informatiky

Univerzita Komenského

Návrh systémovo nezávislej desktop platformy

Autor: Rostislav Šimoník

Diplomový vedúci: RNDr. Damas Gruska, PhD

BRATISLAVA, Apríl 2007

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a s odbornou pomocou vedúceho práce.

....

V Bratislave 28.apríla 2007 *Rostislav Šimoník*

Abstrakt

V práci sa popisuje spôsob interakcie užívateľa s aplikáciami vrámci lokálnych, ale aj vzdialených systémov. Porovnáva súčasné technológie, kde sa snaží analyzovať ich výhody, nevýhody a nakoniec navrhuje riešenie, ktoré nielen pokrýva súčasnú architektúru, ale prináša aj nové spôsoby odstraňujúce zaznamenané nedostatky.

Navrhuje oddelenie vizuálnej interpretácie od aplikačnej logiky procesu. Snaží sa ho nahradiť virtuálnou interpretáciou pomocou základných dátových štruktúr (dáta, udalosti), ktoré nenesú žiadnu vizuálnu informáciu (rozmery, pozície, ...), ale pritom medzi komponentami zachováva vzájomnú väzbu. V tejto časti je návrh schémy, ktorý prepája dátové štruktúry a vizuálne entity. Určuje ich väzbu/spôsob interakcie využitej pri výslednej vizualizácii.

Druhú časť tvorí systém, ktorý sa pomocou komunikačných kanálov pripojí na vzdialený systém a extrahuje virtuálnu interpretáciu, ktorú sa snaží vizualizovať. Počas celého procesu ponecháva aktívne spojenie, aby zabezpečil okamžitú interakciu medzi užívateľom a procesom.

Kompletné riešenie je navrhnuté tak, aby bolo flexibilné, modulárne a systémovo nezávislé, čím umožňuje ďalšie rozšírenie a použitie v akomkoľvek prostredí.

Poďakovanie

Chcel by som sa poďakovať všetkým, ktorých som počas môjho štúdia spoznal a tiež tým, ktorí mi boli podporou nie len pri tejto práci. Predovšetkým Zuzane Hudecovej za mnoho cenných rád, podnetov a pripomienok pri písaní tejto práce.

Obsah

1 Uvedenie do problematiky **1**
1.1 Úvod 1
1.2 Reprezentácia dát 1
1.2.1 Pomocou premenných v triedach 2
1.2.2 Pomocou modelu 3
1.3 Prepojenie 4
1.3.1 Lokálne volanie funkcií 4
1.3.2 Remote procedure call 5
1.3.3 Synchronizácia dát 5
1.3.4 Klient/server 6
1.4 Vizualizácia 7
1.4.1 Pasívna - pomocou aplikačnej logiky 7
1.4.2 Aktívna - pomocou rendereru 7
1.5 Zhodnotenie 8
1.5.1 Zhrnutie z pohľadu užívateľa 8
1.5.2 Zhrnutie z pohľadu vývojára 8
1.6 Ciele 9

2 Analýza štruktúr **11**
2.1 Dátový model 11
2.1.1 Komponent - reprezentácia vizuálnej entity 12
2.1.2 Kontajner - nástroj na vytvorenie hierarchie 17
2.2 Tok udalostí 20
2.2.1 Synchrónna a asynchrónna udalosť 20
2.2.2 Odpoveď na udalosť 23

2.2.3 Reakcia komponentu na synchrónnu udalosť. 25
2.3 Zhodnotenie 26
2.3.1 Rozhranie komponentu 26

3 Návrh implementácie 27
3.1 Základné rozdelenie 27
3.2 Process Container - Kontajner procesov 29
3.2.1 Systémovo nezávislá časť aplikačného kontajneru 29
3.2.2 Systémovo závislá časť aplikačného kontajneru 30
3.2.3 Nezávislosť na programovacích jazykoch 30
3.3 Desktop Container - Kontajner plochy 31
3.3.1 Centrálna databáza dynamických objektov 32
3.4 Komunikácia 32

4 Návrh modulov 34
4.1 Moduly kontajneru procesov 34
4.1.1 Connection Controller 34
4.1.2 Session Manager 36
4.1.3 Virtual Scheme Manager 37
4.1.4 Binary Scheme Encoder 38
4.1.5 Scheme Processor 38
4.1.6 Process manager 38
4.2 Moduly kontajneru plochy 39
4.2.1 Connection Controller 39
4.2.2 Session Manager 41
4.2.3 VPO Manager 42
4.2.4 Binary Scheme Processor 45
4.2.5 Scheme Processor 45
4.2.6 VPO Organizer 45
4.2.7 Collision Detector 46
4.2.8 Input Controller 47
4.2.9 VE Controller 48
4.2.10 VE Optimizer 48
4.2.11 VE Renderer 49

4.2.12 Dynamic Object Manager 49
4.2.13 Dynamic Object Loader 50
4.2.14 Core Controller 50

5 Záver a budúca práca 51
5.1 Budúca práca 52

Zoznam použitej literatúry 53

Zoznam obrázkov

1.1 Stromová reprezentácia 2

2.1 Reprezentácia schémy 12
2.2 Reprezentácia výpisu v strome 17

3.1 Základné rozdelenie pomocou aplikačných kontajnerov 28
3.2 Vnútorná organizácia implementácie kontajneru procesov 30
3.3 Zoznam balíkov v rámci J2SE 32

4.1 Vnútorná organizácia kontajneru procesov pomocou modulov 35
4.2 Vnútorná organizácia kontajneru plochy pomocou modulov 40

Zoznam tabuliek

1.1 Model reprezentovaný pomocou XML 4

2.1 Reprezentácia komponentu v XML 13
2.2 Reprezentácia hierarchie v XML 18
2.3 Reprezentácia udalosti v XML 20
2.4 Reprezentácia odpovede na udalosť v XML 23

Slovník pojmov

Controller časť aplikácie, ktorá sleduje interakciu s užívateľom a na základe toho vytvára udalosti

Dispečer objekt, ktorý zabezpečuje volanie požadovaných príkazov

Framework súbor knižníc

Interface jedinečné uzavreté rozhranie k určitému objektu. Väčšinou po ukončení analýzy zostáva nemenné

Renderer aplikácia, alebo časť, ktorá obsluhuje vykreslenie vizuálnych entít

Kapitola 1

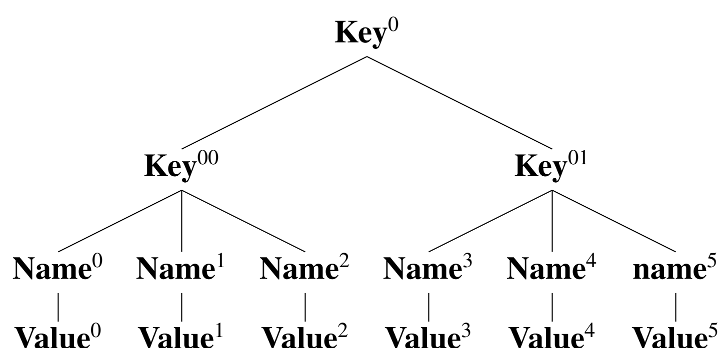
Uvedenie do problematiky

1.1 Úvod

V dnešnej dobe máme dostupnú určitú množinu operačných systémov, ktoré sa líšia svojou architektúrou a spôsobom interakcie s používateľom. U každého z nich je snaha vyvinúť prostredie, ktoré sa snaží pokryť požiadavky užívateľa. Najčastejšia reprezentácia je pomocou vizuálnych prvkov, ktoré sú určitým spôsobom naviazané na aplikáciu. Je to spôsob ktorým aplikácia zobrazuje aktuálne informácie, prípadne informácie o výsledkoch buď nami, alebo aplikáciou spustených procesov. Všeobecne teda môžeme tieto informácie nazvať dátami. Pohľad, ktorý nás zaujíma, a ktorý rieši táto práca, je spôsob akým sú tieto dáta reprezentované, prepojené s vizuálnymi prvkami a ako sú následne vizualizované.

1.2 Reprezentácia dát

Pod reprezentáciou dát si môžeme predstaviť istú skupinu dvojíc (premenných), ktoré pozostávajú z konštantnej zložky, teda identifikátora (názvu premennej) a dynamickej zložky - hodnoty (hodnoty premennej), ktorá nesie konkrétnu informáciu.



obrázok 1.1: Stromová reprezentácia

S výstupom veľkého počtu dát vznikla potreba tieto dvojice v rámci skupiny kategorizovať pomocou vhodne zvolených kľúčov. Kľúče sú volené tak, aby abstrahovali istú vlastnosť spoločnú pre konkrétnu skupinu dát. Tým je docielené, že po zadaní kľúčov pracujeme iba s malou špecifickou skupinou. Tento proces nazývame obaľovanie a je základom všetkých objektovo orientovaných návrhov.

Aktuálne sa najčastejšie používa stromová reprezentácia (obr. 1.1). Dáta sa nachádzajú na pozícii listov a kľúče na pozíciách otcov. K dátam pristupujeme tak, že postupne od koreňa volíme kľúče, až kým sa nedostaneme k listom, teda premenným. Čím vyššie sa v strome nachádzame, tým všeobecnejšiu informáciu nám kľúč ponúka a zároveň uchováva väčšiu množinu dát. Naopak, čím nižšie prechádzame stromom, tým užšiu množinu obaľuje.

1.2.1 Pomocou premenných v triedach

Táto reprezentácia je úzko spätá s programovacím jazykom. Dáta sú uchované v premenných, ktoré obaľujeme v triedach. Je to najjednoduchší spôsob ako uchovávať dáta. Pretože nemusíme využívať žiadnu ďalšiu technológiu, stačí nám samotný jazyk. Hierarchiu teda realizujeme vhodným návrhom tried. Tento spôsob je z dôvodu jednoduchosti využívaný u menších aplikácií, prípadne aplikácií, ktoré produkujú malé množstvo

výstupných dát.

Výhody:

- Nízke nároky na strojový čas. Dáta sú priamo prístupné v premenných, nemusíme vytvárať žiadne mechanizmy na kontrolu správnosti a existencie dát.

Nevýhody:

- Pevne naviazané na aplikačnú logiku, čím nám chýba možnosť externe spracovávať, prípadne manipulovať s dátami.
- Problematické hromadné spracovanie dát, ako je vyhľadávanie, triedenie, uchovávanie, synchronizácia.

1.2.2 Pomocou modelu

Model pochádza z návrhového vzoru Model-View-Controller[*]. Reprezentácia je väčšinou zabezpečená databázou alebo dátovým súborom/tokom, ktorý spĺňa preddefinované kódovanie - dohodnuté pravidlá. Toto nám umožňuje veľmi ľahko zadefinovať a vykonávať úkony pre hromadné spracovanie dát.

Vo výpise (tab. 1.1) môžeme vidieť model reprezentovaný pomocou technológie XML[†], ktorá je najvhodnejšia pre použitie dátových tokov.

Výhody:

- Ochrana aplikačnej logiky. Vizualizácia pristupuje iba k čistým dátam. To nám umožňuje tieto dáta prenášať na vzdialený systém a následne vizualizovať.

[*]prvý raz popísal v roku 1979 Trygve Reenskaug, pre Smalltalk vyvíjaný u Xerox research labs

[†]eXtensive Markup Language v preklade rozšíriteľný značkovací jazyk. Bol vyvinutý konzorciom W3C (World Wide Web Consortium) ako pokračovanie jazyka SGML a HTML. [msx, , wik, c]

```
<key index="0">
    <key index="00">
        <name index="0">value 0</name>
        <name index="1">value 1</name>
        <name index="2">value 2</name>
    </key>
    <key index="01">
        <name index="3">value 3</name>
        <name index="4">value 4</name>
        <name index="5">value 5</name>
    </key>
</key>
```

tabuľka 1.1: Model reprezentovaný pomocou XML

- Jednoduché hromadné spracovanie dát.

Nevýhody:

- Operácie s dátami majú vyššie časové nároky.
- Pre obojstrannú komunikáciu vzniká potreba zadefinovania správcu udalostí.

1.3 Prepojenie

1.3.1 Lokálne volanie funkcií

Prepojenie je väčšinou zabezpečené pomocou funkcií frameworku , ktoré vytvárajú vizuálne prvky a umožňujú nám s nimi rôzne operácie. Terajšie grafické frameworky sa z veľkej miery pevne viažu na operačné systémy, teda aj aplikácie, ktoré tento framework používajú, sú systémovo závislé. Taktiež niektoré vlastnosti u frameworku chýbajú,

jednak z dôvodu zachovania kompatibility s predchádzajúcimi verziami, ako aj z existenčných dôvodov nových technológií. Tieto chýbajúce vlastnosti sa väčšinou kompenzujú vývojármi priamo u aplikácií, čo zvyšuje jednak vývojový čas, ako aj veľa rôznych negatívnych aspektov. Príkladom je rôznorodý spôsob ovládania alebo obsadzovanie systémových zdrojov.

1.3.2 Remote procedure call[rpc,]

Je technológia, ktorá umožňuje prenos volania funkcií na vzdialený systém. Zabezpečuje to dispečer, ktorý zachytí volanie požadovanej funkcie, prenesie požiadavku na vzdialený systém a tam ju zase vykoná. Výsledok je následne zaslaný späť na pôvodný systém.

Výhody:

- Vzdialený prístup docielený jednoduchou úpravou aplikácie, ktorá pred tým obsahovala prepojenie pomocou volania lokálnych funkcií.

Nevýhody:

- Nejednotnosť technológie. Každý operačný systém si definuje vlastný protokol.
- Nevyhnutnosť existencie dispečerov na požadovaných systémoch.

1.3.3 Synchronizácia dát

Princíp je rovnaký ako u Remote procedure call, s tým rozdielom, že namiesto volania funkcií sa vytvorí na oboch stranách rovnaký model, na ktorom vykonávame synchronizáciu. Teda pri zaznamenanej zmene sa prenáša zmenená hodnota na vzdialený systém. Zmena je zaznamenaná pozorovateľmi, ktorí vykonajú následnú funkcionalitu.

Výhody:

- Oproti remote process call nieje potrebná existencia dispečerov, postačuje nám štandardný komunikačný kanál.
- Netreba udržovať spojenie.

Nevýhody:

- Potreba generovať udalosti pomocou zmeny hodnoty na vyhradených premenných.

1.3.4 Klient/server

Klient/Server[wik, a] je sieťová architektúra, ktorá nám oddeľuje aplikačnú logiku od vizualizácie. Vizualizácia je v pozícii klienta, ktorý nám posiela požiadavky na dáta, a potom čaká, pokým aplikačná logika - server spracuje požiadavku. Po spracovaní požiadavky zasiela požadované dáta späť klientovi. Medzi klientom a serverom existuje iba jednosmerná komunikácia, teda klient dokáže reagovať na zmeny na serveri iba pomocou pravidelných, kontrolných požiadaviek. Ako príklad môžeme uviesť web server a webový prehliadač.

Výhody:

- Iba jedna inštancia dát.
- Netreba udržovať spojenie.
- Podpora viacerých klientov.

Nevýhody:

- Jednosmerná komunikácia.

1.4 Vizualizácia

Určuje spôsob, akým sú dáta predstavené užívateľovi. Vytvára základné entity, ktoré zodpovedajú konkrétnej, prípadne viacerým premenným. Entity môžeme rozdeliť na dve skupiny: statické a dynamické. Statické entity sú informatívne, užívateľovi predstavujú konkrétnu informáciu bez možnosti ju modifikovať. Pritom dynamické entity nám ponúkajú možnosť interakcie s užívateľom. Dovoľujú nám modifikovať dáta alebo spúšťať udalosti, na ktoré môže aplikačná logika reagovať.

1.4.1 Pasívna - pomocou aplikačnej logiky

Vizualizácia je vykonaná pasívne, čo znamená, že tvorbu vizuálnych entít obsluhuje aplikačná logika. Návrh je závislý na grafickom frameworku a vo väčšine prípadov, po ukončení vývoja aplikácie, je nemenný.

Výhody:

- Nízke časové nároky počas vývoja aplikácie.

Nevýhody:

- Neoddeliteľnosť vizualizácie od aplikačnej logiky. Vzdialený prístup je riešený prenosom rastrov vykreslených komponentov, čo má za následok prenos veľkého množstva dát.

1.4.2 Aktívna - pomocou rendereru

K dátam sa pristupuje logicky oddelenou rutinou, prípadne samostatnou aplikáciou, ktorá vytvára vizuálne entity. Je zodpovedná za sledovanie zmien na dátach. Väčšinou býva sprevádzaná controllerom. Ten zaznamenáva interakciu s užívateľom, ktorú potom predáva aplikačnej logike pomocou udalostí.

Výhody:

- Oddeliteľnosť aplikačnej logiky od vizualizácie.
- Možnosť vzdialeného prístupu.

Nevýhody:

- Zvýšene časové nároky na vývoj aplikácie.

1.5 Zhodnotenie

Každá metóda má svoje pre a proti, ktoré zasahujú jednak užívateľa, ale aj vývojára, ktorý tvorí aplikáciu. Preto treba nájsť taký model, ktorý by všetky nevýhody minimalizoval. Buď voľbou vhodnej technológie, alebo ponúknutím konkrétnej funkcionality.

1.5.1 Zhrnutie z pohľadu užívateľa

Medzi požiadavky súčasného užívateľa patrí záujem mať jednotné ucelené prostredie, z ktorého dokáže pristupovať k rôznym službám. Buď dostupným z lokálneho, alebo aj zo vzdialeného systému. Mal by mu byť umožnený k nim prístup bez obmedzenia systému, na ktorom sú služby dostupné. Každý užívateľ má vlastný subjektívny názor na interakciu s aplikáciou. Preto vzniká potreba aplikáciu ľahko adoptovať, bez nutnosti ďalšieho vývoja.

1.5.2 Zhrnutie z pohľadu vývojára

Cieľom každého vývojára, ktorý sa zaoberá vývojom softvéru, je minimalizovať čas na vyhotovenie diela. Pritom vytvoriť aplikáciu, ktorá spĺňa všetky požiadavky užívateľa. Celý proces môžeme rozdeliť na analýzu a implementáciu vizualizácie a aplikačnej

logiky. Analýza a implementácia vizualizácie u niektorých aplikácii trvá až 50% času potrebného na celkové vyhotovenie. Zavedením frameworku, ktorý dokáže výstupné dáta aplikačnej logiky efektívne spracovať a vizualizovať, bez potreby zásahu vývojára, by proces vývoja vizualizácie minimalizoval.

1.6 Ciele

ZVÝŠENÁ STABILITA A BEZPEČNOSŤ PROCESOV

Tým že u procesov umožníme oddeliť vizuálnu časť, zároveň odstránime závislosť na vizuálnom module systému (VMS) a teda pri kritickej havárii procesu obsluhujúceho VMS predídeme aj pádu ostatných procesov.

EFEKTÍVNA INTERACKIA SO VZDIALENÝM SYSTÉMOM

Možnosť interakcie s procesmi na vzdialených systémoch bez nutnosti prenosu aplikačného kódu, prípadne akýchkoľvek dát určených aplikačnej logike procesu. Množstvo prenesených dát je minimalizované, čím je dosiahnutá plynulá interakcia.

JEDNOTNOSŤ V RÁMCI OPERAČNÝCH SYSTÉMOV

Zavedením pružného protokolu interakcie dokážeme implementovať framework prostredia na väčšinu dostupných operačných systémov.

VIZUÁLNA ČASŤ NEZÁVISLÁ NA OPERAČNOM SYSTÉME

Uzavretosť vizuálneho prostredia umožní implementáciu na platformovo nezávislej technológii.

RÔZNA INTERPRETÁCIA VIZUÁLNYCH ENTÍT

Abstraktné popisovanie vizuálnych entít umožní ich rôznu interpretáciu v rámci aplikácie.

MODULÁRNOSŤ

Základné funkčné body budu realizované pomocou modulov. Každý z nich bude uzavretý a navonok bude ponúkať nemenný transparentný interface, čím zabezpečíme prehľadnosť modelu a efektívnu opravu chýb.

DYNAMICKÝ PRÍSTUP

Celý návrh bude ponúkať silnú podporu dynamických objektov v rámci modulov, podmodulov apod. Zavedením tejto technológie vytvoríme možnosť rozšírenia existujúceho modelu o novú funkcionalitu.

TRANSPARENTNÝ A FLEXIBILNÝ DÁTOVÝ INTERFACE

Zavedením transparentného a flexibilného dátového interfacu umožníme vývojárom jednoduché a rýchle použitie v rámci aplikačnej logiky.

ZNÍŽENIE ČASOVEJ NÁROČNOSTI PRE VÝVOJ APLIKÁCIE Oddelením vizualizácie zabezpečíme rýchly vývoj aplikácií, bez potreby implementácie a návrhu vizuálnych častí.

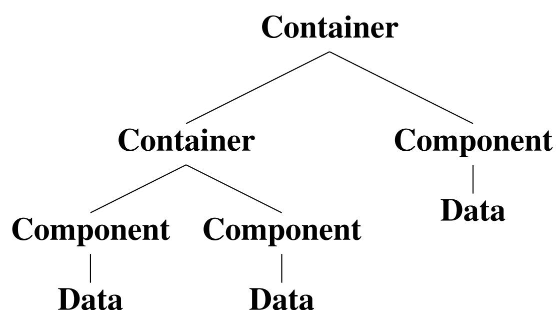
Kapitola 2

Analýza štruktúr

Pre splnenie cieľov, ktoré sme uviedli v predchádzajúcej kapitole, potrebujeme oddeliť aplikačnú logiku od vizualizácie. Aby sme toto mohli docieliť, zabezpečíme všetko potrebné čo medzi týmito dvoma časťami prebieha. Jednak je to výmena dát a za druhé je to tok udalostí. Nato si však potrebujeme zadefinovať ich reprezentáciu - schému. Schéma je pre vývojára spôsob, akým bude v rámci aplikačnej logiky dátový model popisovať a implementovať.

2.1 Dátový model

Vo vizualizácii považujeme za elementárny prvok vizuálnu entitu (napríklad editbox, label, ...). Pri dátovej reprezentácii máme istú dátovú časť, ktorá jej zodpovedá a keď ju obalíme, môžeme ju v schéme považovať za elementárny prvok. Nakoniec, aby sme mohli komponenty organizovať, musíme definovať, akým spôsobom bude zaradená do hierarchie. Nato si zadefinujeme objekt kontajner, ktorý nám dokáže obaľovať nielen komponenty, ale aj už existujúce inštancie kontajnerov. Tento objekt bude reprezentovať pozície vrcholov nad komponentami(obr. 2.1).



obrázok 2.1: Reprezentácia schémy

2.1.1 Komponent - reprezentácia vizuálnej entity

Vizuálna entita je špecifická hodnotou, kategóriou a množinou možných udalostí. Preto pre komponent v schéme definujeme nasledovné XML* elementy a atribúty (tab. 2.1). Tie tvoria definíciu komponentu, zoznam príznakov, dátovú časť a registráciu udalostí.

Definícia komponentu

---

component: *Element ktorý obaľuje komponent.*
- **name:** *Je jedinečné meno inštancie komponentu.*
- **ucid:** *Unique Category IDentifier - Jedinečné číslo kategórie komponentu.*
- **stamp:** *Jedinečná známka vytvorenia komponentu.*

Každý komponent je definovaný svojím menom inštancie a jedinečným číslom kategórie - známkou. Kategória hovorí o tom, akým spôsobom sa komponent správa. Pre každú kategóriu komponentu existuje preddefinovaný formát dát a zoznam udalostí, ktoré môže spúšťať. Keďže je nemenný, môžeme *"ucid"* považovať za identifikátor rozhrania komponentu. V ďalšej kapitole uvidíme ako každé rozhranie pokrýva viacej imple-

*Na reprezentáciu dátového modelu použijeme XML technológiu, ktorá nám najlepšie zastrešuje požiadavky návrhu.

```
<component name="listbox1" ucid="1000-0000-0000-0001" stamp="5">
    <flags default="4000-0000-0000-0002">
        <flag ufid="3000-0000-0000-0001"
              flow="4000-0000-0000-0003" set="yes"/ >
        <flag ufid="3000-0000-0000-0010"
              flow="4000-0000-0000-0001" set="no"/ >
    </flags
    <data conc="1">
        <value conc="1" key="count">30</value>
        <value conc="1" key="offset">10</value>
        <value conc="1" key="item_count">3</value>
        <value conc="1" key="item0">value 10</value>
        <value conc="1" key="item1">value 11</value>
        <value conc="1" key="item2">value 12</value>
    </data>
    <events>
        <event ueid="2000-0000-0000-0001"/ >
    </events>
</component>
```

tabuľka 2.1: Reprezentácia komponentu v XML

mentácií, čo nám zabezpečuje rôznu interpretáciu vizuálnych entít.

Meno inštancie nám slúži na rozpoznanie inštancií komponentu. Jednak nám zaručuje možnosť vytvorenia viacerých komponentov, ale aj jedinečnú identifikáciu v rámci čiastočne lokálneho priestoru[†].Známka nám zabezpečí totožnosť inštancie komponentu. Jej význam je dôležitý pre párovanie udalostí s komponentom.

[†]Túto vlastnosť vysvetlíme neskôr pri definícii obalenia potomkov

príklad: Vo výpise máme (tab. 2.1) názov komponentu *"listbox1"* a známku s hodnotou 5. Čo znamená, že ako náhle sa hodnota známky zmení, komponent s rovnakým menom je už úplne iná inštancia.

Definícia príznakov

---

component
- **flags:** *Element, ktorý obaľuje zoznam príznakov.*
  - **default:** *Implicitná hodnota pre všetky zdedené príznaky.*
- **flag:** *Element, ktorý obaľuje príznak.*
  - **ufid:** *Unique Flow IDentifier - jedinečné číslo príznaku.*
  - **flow:** *Hodnota príznaku. Určuje, akým spôsobom komponent reaguje na príznak.*
  - **set:** *Určuje či je príznak nastavený alebo sa dedí podľa hodnoty príznaku.*

Príznaky majú za úlohu podávať informáciu, akým spôsobom môže vizualizácia s komponentom manipulovať. Ich význam je pre aplikačnú logiku aj pre vizuálne entity známy. Príznaky môžeme dediť od otca v rámci schémy, alebo sú nastavené u samotného komponentu. V prípade, že príznaky dedíme, sa spôsob ako na ne bude komponent reagovať určuje atribútom *"flow"*. Ak nastane situácia, že dedíme príznak, ktorý sa nachádza aj medzi nastavenými príznakmi, tak situáciu riešime logickým operátorom OR[‡]. Atribút *"default"* nám umožňuje implicitné nastavenie *"flow"* hodnoty pre všetky ostatné príznaky, ktoré náš komponent dedí a nebola pre nich *"flow"* hodnota definovaná.

[‡]boolovký operátor $\vee$

príklad: Vo výpise (tab. 2.1) máme príznak *"3000-0000-0000-0010"*, ktorý predstavuje zakázanie zmeny hodnoty a jeho *"flow"* hodnotu *"4000-0000-0000-0001"*, ktorá znamená povolenie dedenia príznakov. Ak na jednom z otcov je nastavený tento príznak, a každý objekt na ceste medzi nastaveným príznakom a naším komponentom dovoľuje preposlanie príznaku svojím potomkom, tak i náš komponent bude mať zakázanú zmenu hodnoty.

Definícia dát

---

component
- **data:** *Element, ktorý obaľuje dátovú časť.*
  - **conc:** *Register zmeny celej dátovej časti.*
- **value:** *Element, ktorý obaľuje konkrétnu hodnotu.*
  - **conc:** *Register zmeny konkrétnej premennej.*
  - **key:** *Identifikátor hodnoty - názov premennej.*

Ako sme si zadefinovali v úvode do problematiky, dáta budú pozostávať zo skupiny premenných a budú obalené kľúčmi. Prvú úroveň obalenia nám poskytuje komponent, pre ktorý sú tieto dáta špecifické. V našej schéme to zabezpečuje množina elementov *"value"*, ktoré hodnotu obaľujú. Prvý atribút *"key"* udáva identifikátor hodnoty - názov premennej. Atribút *"conc"* zabezpečuje ochranu pred vzájomným konfliktom užívateľa a aplikačnej logiky pri paralelnej zmene hodnoty. To, ako sa má vyriešiť táto situácia, záleží na vhodnom návrhu komponentu. Atribúty *"conc"* by mali byť pri vytvorení nového komponentu nastavené na počiatočnú hodnotu.

Aby bol prenos dát minimalizovaný, treba pri návrhu komponentu korektne voliť zoznam a obsah premenných. To znamená, že by nemali byť preplnené dátami, ktoré užívateľ nemá potrebu vidieť, čo môže mať za následok zníženej kvality interakcie s užívateľom. V prípade potreby ďalších dát by mal komponent o ne požiadať formou udalosti.

príklad: Vo výpise máme (tab. 2.1) komponent typu listbox[§]. Počet všetkých hodnôt udáva premenná *"count"*. Ďalšia premenná *"item_count"* nám udáva počet pre užívateľa viditeľných dát. Je sprevádzaná premennou *"offset"*, ktorá udáva začiatočný index viditeľných dát. V prípade, že užívateľ potrebuje vidieť ďalšie dáta komponent vyšle udalosť o zmene indexu. Ak užívateľ spôsobí zmenu veľkosti komponentu, vizualizácia by mala informovať aplikačnú logiku pomocou udalosti, ktorá vyžiada adekvátnu zmenu premennej *"item_count"*.

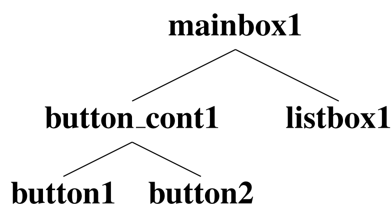
Definícia registrácie udalostí

---

component
└─ **events:** *Element, ktorý obaľuje registráciu udalostí.*
  └─ **event:** *Element, ktorý obaľuje konkrétne prihlásenie udalosti.*
    └─ **ueid:** *Unique Event IDentifier - jedinečné číslo udalosti.*

Každý komponent, s ktorým môže užívateľ prípadne vizualizácia manipulavoť, ponúka diskrétnu množinu udalostí, na ktoré môže aplikačná logika reagovať. Avšak, aby dokázala udalosť zachytiť, musí si ju zaregistrovať v definícii komponentu pomocou elementu *"event"*, kde atribút *"ueid"* určuje jej jedinečne číslo. Ako náhle nastane uda-

[§]Listbox je vizuálna entita, ktorá zobrazuje indexovaný zoznam hodnôt a umožňuje označenie jednej alebo viacerých z nich



obrázok 2.2: Reprezentácia výpisu (tab 2.2) v strome

losť a existuje na ňu registrácia, je zaslaná spolu s príslušnými parametrami aplikačnej logike.

2.1.2 Kontajner - nástroj na vytvorenie hierarchie

Aby sme mohli komponenty efektívne zaradiť do stromovej hierarchie, potrebujeme ich určitých spôsobom obaliť. Na to si zadefinujeme kontajner. Kontajner je v podstate podobný objekt ako komponent s tým rozdielom, že dátovú časť nahradíme zoznamom obalených objektov - komponentov a iných kontajnerov. Taktiež potrebujeme definíciu príznakov, ktorej zápis bude taký istý, avšak pribudne nám význam preposielania príznakov do potomkov. Registrácia udalostí u kontajnera stráca význam, pretože sú určene iba pre vizualizáciu, ktorá ich vie spracovať bez aplikačnej logiky.

Definícia kontajnera

---

container *Element ktorý obaľuje kontajner.*

- **name:** *Je jedinečné meno inštancie kontajnera.*
- **ucid:** *Unique Category IDentifier - Jedinečné číslo kategórie kontajnera..*

```
<container name="mainbox1" ucid="5000-0000-0000-0001">
    <flags default="4000-0000-0000-0002">
        <flag ufid="3000-0000-0000-0001"
              flow="4000-0000-0000-0003" set="yes"/ >
    </flags>
    <childs>
        <container name="button_cont1" ucid="5000-0000-0000-0001">
            <flags default="4000-0000-0000-0002">
                <flag ufid="3000-0000-0000-0001"
                      flow="4000-0000-0000-0004" set="no"/ >
            </flags>
            <childs>
                <component name="button1"
                           ucid="1000-0000-0000-0005" stamp="3">
                  ...
                </component>
                <component name="button2"
                           ucid="1000-0000-0000-0005" stamp="4">
                  ...
                </component>
            </childs>
        </container>
        <component name="listbox1" ucid="1000-0000-0000-0001"
                   stamp="5">
          ...
        </component>
    </childs>
</component>
```

tabuľka 2.2: Reprezentácia hierarchie v XML

Definícia obalenia potomkov

---

container
└ **childs:** *Element, ktorý obaľuje potomkov.*

Kontajner obaľuje potomkov do tzv. čiastočne lokálneho priestoru. Je to priestor v ktorom sa nachádzajú priami potomkovia. Je špecifický tým, že v jeho rámci musí byť meno každého komponentu alebo kontajneru jedinečné, aby sme vedeli tieto objekty adresovať. Na určenie adresy používame cestu tvorenú z mien objektov, ktoré ležia v strome na ceste medzi koreňom a objektom, ktorý chceme adresovať.

Napríklad pre *"button1"* z výpisu (tab. 2.2) dostávame cestu *"/mainbox1/button_cont1/button1"*.

Preposielanie príznakov

Definícia príznakov je totožná s definíciou u komponentu. Nová vlastnosť, ktorú v tejto sekcii získavame, je preposielanie príznakov na potomkov. Teda *"flow"* hodnota je rozšírená o spôsob, akým budeme príznak preposlielať.

príklad: Povedzme, že vo výpise (tab. 2.2) komponent *"mainbox1"* ma nastavený príznak na povolenie modifikácie hodnoty a *"flow"* hodnotu na povolene preposielanie príznaku na potomkov. Týmto je zaručené, že komponent *"listbox1"* aj kontajner *"button_cont1"* tento príznak zdedia. Keďže kontajner *"button_cont1"* obaľuje komponenty, ktoré tento príznak neovplyvňuje, môže vrámci vlastnej definície patričnou *"flow"* hodnotou preposielanie tohto príznaku potlačiť.

2.2 Tok udalostí

Náš návrh sa operia o istú kombináciu medzi technológiou klient/server a synchronizáciou dát. Synchronizácia dát nám prebieha vždy, keď aplikačná logika modifikuje dáta. Opačný smer komunikácie zabezpečuje tok udalostí, ktorý vzniká v komponente. Po interakcii užívateľa, a nie len vtedy, sa vytvorí udalosť, ktorá je zaslaná aplikačnej logike. Komponent môže, ale aj nemusí očakávať reakciu aplikačnej logiky. Záleží na tom, aký typ udalosti bol zaslaný.

2.2.1 Synchrónna a asynchrónna udalosť

```
<event ueid="2000-0000-0000-0001" sync="yes">
    <ref ucid="1000-0000-0000-0001" stamp="5"
         path="/mainbox1/listbox1"/ >
    <data conc="1">
        <value conc="1" key="offset">5</value>
    </data>
    <arg>
        <value key="selcount">2<value>
        <value key="selidx_0">10</value>
        <value key="selidx_1">12</value>
    </arg>
</event>
```

tabuľka 2.3: Reprezentácia udalosti v XML

Asynchrónnou udalosťou si definujme takú udalosť, ktorá neočakáva potvrdenie od aplikačnej logiky. Pre komponent to znamená, že nemusí meniť svoj stav.

Naopak synchrónna udalosť je taká, ktorá očakáva od aplikačnej logiky potvrdenie a následnú (nepovinnú) aktualizáciu dát.

Udalosť pozostáva z referencie na komponent, dátovej časti a argumentov.

Definícia udalosti

---

event *Element, ktorý obaľuje udalosť.*
- **ueid:** *Unique Event IDentifier - Jedinečné číslo udalosti.*
- **sync:** *Príznak synchrónnej/asynchrónnej udalosti.*

Definícia referencie na komponent

---

event
- **ref:** *Element, ktorý obaľuje referenciu.*
  - **ucid:** *Unique Category IDentifier - Jedinečné číslo kategórie komponentu.*
  - **stamp:** *Jedinečná známka komponentu.*
  - **path:** *Cesta udávajúca polohu komponentu v schéme*

Referencia nám udáva, ktorý komponent udalosť vyvolal. Zabezpečuje to pomocou cesty v schéme. Známka nám potvrdzuje inštanciu komponentu. Ak sa pri spracovaní udalosti nezhoduje, znamená to, že udalosť sa odvoláva na neexistujúci komponent.

Definícia dátovej časti

---

event
- **data:** *Element, ktorý obaľuje dátovú časť.*
  - **conc:** *Register zmeny celej dátovej časti.*
  - **value:** *Element, ktorý obaľuje konkrétnu hodnotu.*
    - **conc:** *Register zmeny konkrétnej premennej*
    - **key:** *Identifikátor hodnoty - názov premennej.*

Definícia dátovej časti je zhodná s definíciou u komponentu. Avšak teraz slúži, v prípade úspešného spracovania udalosti, na obalenie budúcich hodnôt premenných komponentu.

Definícia argumentov

---

event
- **arg:** *Element, ktorý obaľuje argumenty.*
  - **value:** *Element, ktorý obaľuje konkrétny argument.*
    - **key:** *Identifikátor argumentu.*

Úlohou argumentov je umožniť zasielanie dodatočných informácií aplikačnej logike. Definujeme ich podobne ako u dát, avšak register zmeny je nepotrebný.

príklad: Vo výpise máme (tab. 2.1) komponent *"listbox1"*. Užívateľ si vrámci neho označí hodnoty, čím sa vyvolá udalosť selekcie *"1000-0000-0000-0001"* (tab.

2.3). Komponent potom pomocou argumentov *"selcount"* (počet označených hodnôt), *"selidx_0"*, *"selidx_1"* (indexy označených hodnôt) odošle udalosť o selekcii aplikačnej logike.

2.2.2 Odpoveď na udalosť

```
<reply ueid="2000-0000-0000-0001">
    <ref ucid="1000-0000-0000-0001" stamp="5"
         path="/mainbox1/listbox1"/ >
    <state usid="2000-0000"/ >
    <data conc="2">
        <value conc="2" key="offset">5</value>
        <value conc="1" key="item0">value 5</value>
        <value conc="1" key="item1">value 6</value>
        <value conc="1" key="item2">value 7</value>
    </data>
</reply>
```

tabuľka 2.4: Reprezentácia odpovede na udalosť v XML

Odpoveď na udalosť je zasielaná aplikačnou logikou po spracovaní synchrónnej udalosti. O úspešnosti spracovania informuje pomocou hodnoty stavu. Ak udalosť požadovala modifikáciu dát, potom súčasťou odpovede sú aj nové nadobudnuté hodnoty.

Definícia odpovede

---

reply *Element, ktorý obaľuje odpoveď.*
- **ueid:** *Unique Event IDentifier - Jedinečné číslo udalosti.*

Definícia hodnoty stavu

---

reply
- **state:** *Unique Event IDentifier - Jedinečné číslo udalosti.*
  - **usid:** *Unique State IDentifier - Jedinečné číslo stavu.*

Zavedením stavu informujeme komponent o výsledku spracovania udalosti. Každá synchrónna udalosť má definovanú množinu možných hodnôt, avšak povinne minimálne dve. Jednu pre pozitívny, druhú pre negatívny výsledok. Zoznam možných hodnôt je uzavretý a vopred určený pri návrhu komponentu.

Definícia referencie na komponent

---

reply
- **ref:** *Element, ktorý obaľuje referenciu.*
  - **ucid:** *Unique Category IDentifier - Jedinečné číslo kategórie komponentu.*
  - **stamp:** *Jedinečná známka komponentu.*
  - **path:** *Cesta udávajúca polohu komponentu v schéme*

Definícia nových hodnôt

reply
- **data:** *Element, ktorý obaľuje dátovú časť.*
  - **conc:** *Register zmeny celej dátovej časti.*
- **value:** *Element, ktorý obaľuje konkrétnu hodnotu.*
  - **conc:** *Register zmeny konkrétnej premennej*
  - **key:** *Identifikátor hodnoty - názov premennej.*

V prípade, že udalosť požadovala zmenu dát a aplikačná logika úspešne spracovala udalosť, potom odpoveď obsahuje zoznam nových hodnôt. Zvýšené hodnoty *"conc"* atribútov taktiež informujú o akceptovaných hodnotách.

2.2.3 Reakcia komponentu na synchrónnu udalosť.

Nato, aby sme zabezpečili plynulú interakciu s užívateľom, je treba zadefinovať korektné správanie komponentov. Teda čo má komponent robiť v prípade, keď čaká na odpoveď.

Chránený a nechránený stav komponentu.

Stav medzi tým, keď komponent odoslal udalosť a tým, keď príjme odpoveď, budeme nazývať nechránený. Naopak v ostatných prípadoch bude v chránenom stave.

Komponent by mal v nechránenom stave umožniť interakciu s užívateľom pomocou ukladania ďalších udalostí do fronty. Následne po prijatí odpovede, tieto udalosti

postupne z fronty vyberať a zasielať aplikačnej logike. Užívateľ by mal byť vizuálne informovaný o aktuálnom stave komponentu.

2.3 Zhodnotenie

Zadefinovaním základných štruktúr vrámci schémy sme umožnili vytvorenie komponentov a ich organizáciu do hierarchie pomocou kontajnerov. Ich spätnú interakciu s aplikačnou logikou nám umožňujú definície udalostí. Aby sme mohli komponenty použiť v aplikačnej logike, potrebujeme k tomu ich rozhranie.

2.3.1 Rozhranie komponentu

Rozhranie komponentu nám hovorí, akým spôsobom môžeme komponent použiť a ako reagovať na jeho udalosti. Preto by mala byť dostupná dokumentácia k rozhraniu a mala by obsahovať nasledovné veci:

- Jedinečné číslo kategórie komponentu.
- Popis formátu a významu dátových premenných.
- Popis udalostí.
  - Jedinečné číslo udalosti.
  - Popis prislúchajúcich dátových premenných.
  - Popis formátu a významu argumentov.
- Popis prislúchajúcich odpovedí.
  - Popis možných stavov.
    - Jedinečné číslo stavu.

Kapitola 3

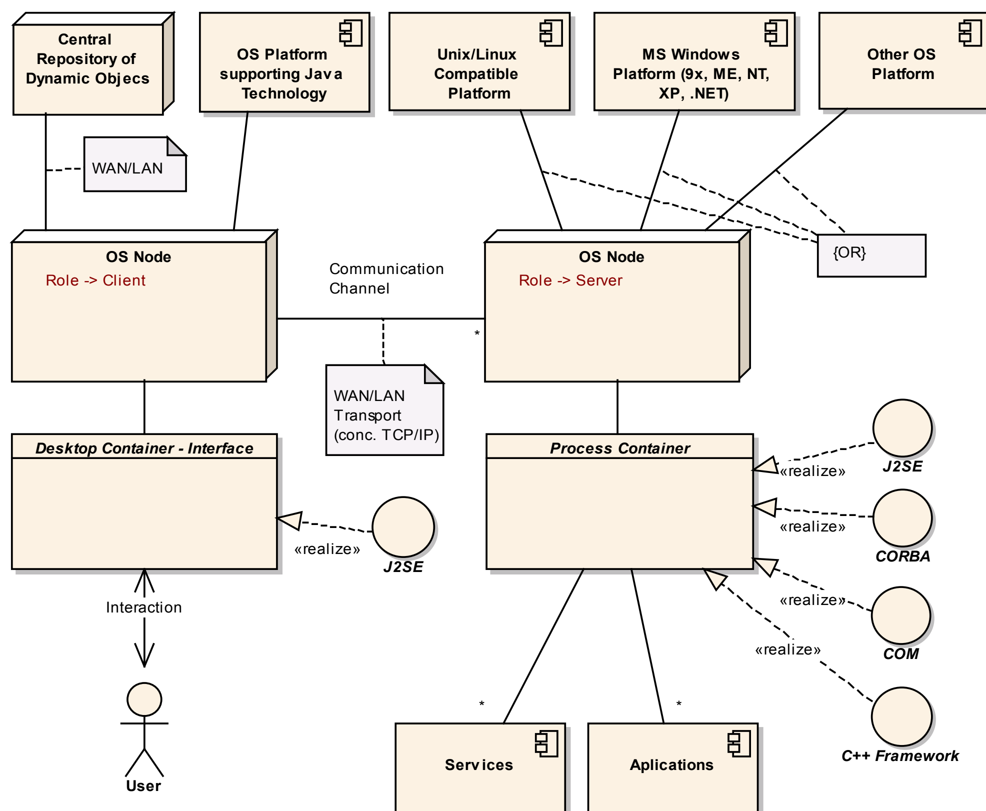
Návrh implementácie

V tejto kapitole sa bude riešiť už výsledný návrh implementácie. Budeme voliť vhodné technológie, ktoré nám zabezpečia nielen nezávislosť na operačnom systéme, ale aj použitie vo väčšine vývojových prostredí. Určíme si základné rozdelenie na aplikačné kontajnery, u ktorých si vysvetlíme ich princíp a postavenie.

3.1 Základné rozdelenie

Tým, že sme oddelili aplikačnú logiku od vizualizácie, dátovú reprezentáciu sme definovali pomocou modelu a pre prepojenie používame synchronizáciu dát kombinovanú s technológiou klient/server, vzniká nám tak rozdelenie na dve nezávislé časti - aplikačné kontajnery.

Ako vidíme na obrázku (obr. 3.1) Jedna časť bude obsluhovať prepojenie s aplikáciami a službami, teda aplikačnou logikou, a budeme ju nazývať **process container - kontajner procesov**. Jej úlohou bude registrovať aplikácie a služby, sledovať zmeny na dátovom modeli a smerovať prichádzajúce udalosti do ich aplikačnej logiky.



obrázok 3.1: Základné rozdelenie pomocou aplikačných kontajnerov

Druhá časť bude obsluhovať vizualizáciu a interakciu s užívateľom. Budeme ju nazývať **desktop container - kontajner plochy**. Kontajner plochy bude vytvárať a obsluhovať vizuálne reprezentácie procesov, aktualizovať ich dátovú časť, sledovať interakciu s užívateľom, ktorú pomocou udalostí bude posielať do aplikačnej logiky.

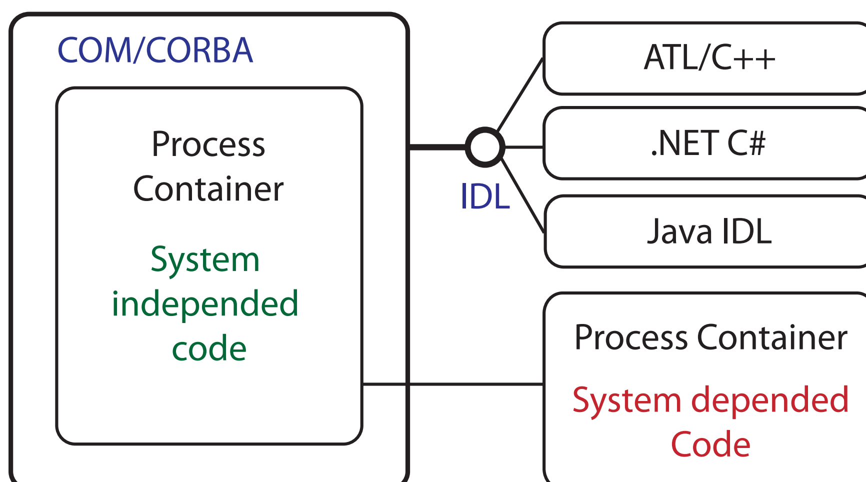
Komunikáciu medzi oboma aplikačnými kontajnermi nám zabezpečí vhodný komunikačný protokol, ktorý nám umožní prepojenie v rámci vzdialených systémov. Nič nám však nebráni v lokálnom prepojení oboch aplikačných kontajnerov v rámci jedného operačného systému.

3.2 Process Container - Kontajner procesov

Kontajner procesov je časť, ktorá bude namierená na čo najväčšiu množinu operačných systémov a bude systémovo závislá. Preto jej implementácia musí byť riešená tak, aby sme pri jej zavedení do nového operačného systému, vynaložili čo najmenšie úsilie. Teda snažiť sa zdrojový kód rozdeliť na systémovo nezávislý, ktorý by mal obsahovať väčšinu funkcionality a na systémovo závislý, v ktorom zabezpečíme prepojenie s konkrétnym operačným systémom. Taktiež treba umožniť použitie v rámci čo najväčšej množiny vývojových prostredí, teda zabezpečiť nezávislosť na programovacích jazykoch.

3.2.1 Systémovo nezávislá časť aplikačného kontajneru

Najvhodnejším programovacím jazykom bude jazyk C++. Medzi jeho výhody patrí to, že spolu so štandardnými knižnicami, ktoré pre náš účel postačujú, je preložiteľný na skoro všetkých systémových platformách. Jeho výsledný kód je optimalizovaný na konkrétnu architektúru.



obrázok 3.2: Vnútorná organizácia implementácie kontajneru procesov

3.2.2 Systémovo závislá časť aplikačného kontajneru

Pomocou vysunutých tried zabezpečíme prepojenie s konkrétnym operačným systémom. Triedy môžeme rozdeliť nasledovne:

- Triedy na prácu s vláknami.
- Triedy na prácu s reťazcami.
- Triedy obsluhujúce sieťovú komunikáciu.
- Triedy obsluhujúce vstupno-výstupné operácie.
- Triedy zabezpečujúce kontrolu práv.

3.2.3 Nezávislosť na programovacích jazykoch

Aby sme mohli v aplikáciách vytvárať schémy a následne ich prepájať s kontajnerom procesov, potrebujeme z neho vysunúť rozhranie, ktoré bude dostupné pre každé vývojové prostredie. Inak povedané, potrebujeme technológiu, ktorá nám zabezpečí dostupnosť v

rámci čo najväčšej množiny programovacích jazykov.

Pre riešenie tejto situácie bol vyvinutý IDL (interface description language), ktorý umožňuje popísať rozhranie. Následne ho môžeme pomocou technológie COM (Component Object Model [msc, , Kačmář, 2000]), alebo pomocou technológie CORBA (Common Object Request Broker Architecture [cor, b, cor, a, wik, b]) efektívne použiť v ľubovoľnom programovacom jazyku. Podmienkou je, aby vývojové prostredie ponúkalo súbor knižníc pre prácu s týmito technológiami. Napríklad pre MS Visual Studio* nám to zabezpečí súbor knižníc ATL (Active Template Library [msd,])

3.3 Desktop Container - Kontajner plochy

Kontajner plochy musí byť autonómny od operačného systému. Toto nám umožní jeden z interpretovaných programovacích jazykov. V našom prípade najlepšie zastrešenie ponúka Java Platform, Standard Edition [jav, b]. Integrované balíky (obr. 3.3) nám ponúkajú kompletné zastrešenie aplikačného kontajnera plochy. Optimalizáciu časovo náročných rutín môžeme zabezpečiť pomocou implementácie v jazyku C. Prepojenie medzi natívnym kódom a interpretáciou umožníme pomocou balíku JNI (Java™Native Interface[jav, a]).

Súčasťou kontajnera plochy je aj podpora dynamických objektov†. Dynamické objekty sa načítavajú buď z lokálneho systému alebo z centrálnej databázy dynamických objektov. Ak sa dynamický objekt nenachádza lokálne, mal by byť vyhľadaný v centrálnej databáze. Tým zabezpečíme ich šírenie a použitie v rámci komunity.

*Vývojové prostredie ponúkané firmou Microsoft

†ako neskôr uvidíme v popise modulov

obrázok 3.3: Zoznam balíkov v rámci J2SE

3.3.1 Centrálna databáza dynamických objektov

Nám slúži na ukladanie dynamických objektov (komponentov, kontajnerov, descriptorov, podmodulov . . .). Mala by byť reprezentovaná webovým serverom alebo databázou. Dynamické objekty sú potom prístupné pomocou svojho jedinečného čísla. Centrálna databáza by mala ponúkať užívateľské rozhranie, v ktorom by sa dali objekty vyhľadávať pomocou jedinečného čísla, kategórie alebo podľa kľúčových slov. Následne ponúknuť dokumentáciu pre popis rozhrania objektu. Takto ponúkneme vývojárom možnosť používať dynamické objekty v rámci svojej aplikácie alebo v prípade vytvorenia nového objektu, možnosť distribuovať ho v rámci komunity.

3.4 Komunikácia

Keďže aplikačný kontajner plochy je v roli klienta a aplikačný kontajner procesov v roli serveru, potom na komunikáciu potrebujeme vhodnú technológiu, ktorá nám zabezpečí ich spojenie. V rámci sietí WAN a LAN nám na tento účel postačuje internetový pro-

tokol TCP/IP.

Kapitola 4

Návrh modulov

V tejto kapitole zadefinujeme pre obidva aplikačné kontajnery vnútornú hierarchiu. Tá bude pozostávať z logicky oddelených častí - modulov, ktoré medzi sebou komunikujú. Každý modul obaľuje konkrétnu funkcionalitu, za ktorú zodpovedá a voči ostatným modulom ponúka nemenné rozhranie. Teda konečné rozhranie by malo byť navrhnuté ešte pred samotnou implementáciou. Rozšírenie modulu o novú funkcionalitu zabezpečíme podporou dynamických objektov.

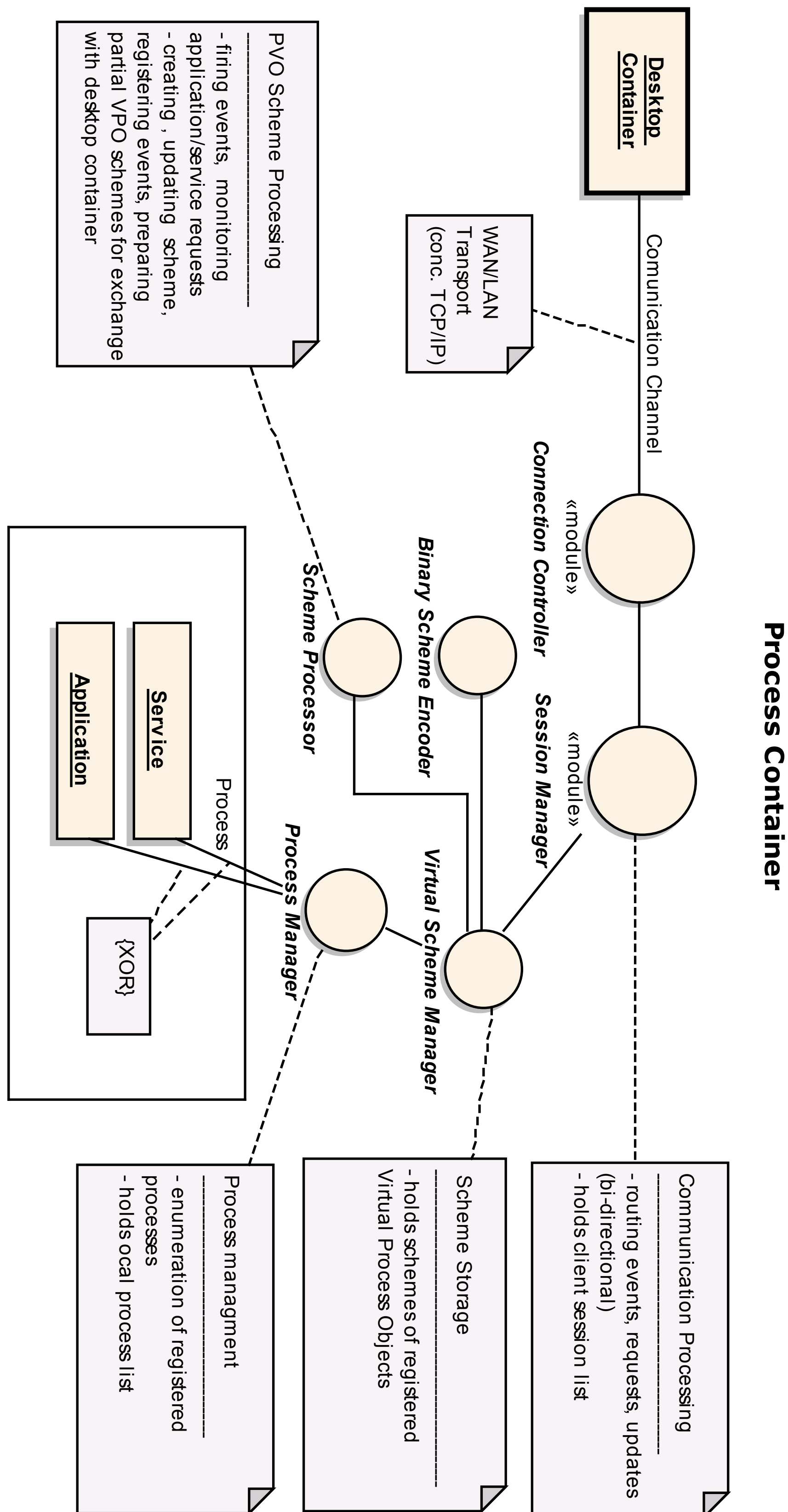
4.1 Moduly kontajneru procesov

4.1.1 Connection Controller

Modul ma za úlohu zabezpečiť sieťovú komunikáciu. Je v roli serveru, teda očakáva prichádzajúce spojenia. Ak spojenie bolo prijaté, informuje modul *"Session Manager"* o naviazaní spojenia.

Závislosti:

- **Session Manager**



obrázok 4.1: Vnútorná organizácia kontajneru procesov pomocou modulov

  - spätné volanie o naviazaní spojenia
  - spätné volanie o prijatí balíka

4.1.2 Session Manager

Modul má za úlohu spravovať relácie. Vždy po prijatí nového spojenia skontroluje, či je to požiadavka o novú reláciu alebo o obnovenie pôvodnej relácie. V oboch prípadoch kontroluje nastavenie práv, ktoré keď nie sú akceptované, tak vyšle požiadavku modulu *"Connection Controller"* na ukončenie spojenia. Ďalšou úlohou je smerovať balíky zo spojenia na modul *"Virtual Scheme Processor"* a naopak. Ak momentálne nemôže balík odoslať, uloží ho do fronty pre budúce možné odoslanie. Modul vykonáva nasledovné príkazy:

- Vytvorenie relácie
- Obnovenie relácie
- Zatvorenie relácie
- Kontrola práv
- Smerovanie balíkov

Závislosti:

- **Connection Controller**
  - Odosielanie balíkov z fronty
  - Zatváranie spojenia
- **Virtual Scheme Processor**
  - Odovzdávanie prijatých balíkov zo spojenia

4.1.3 Virtual Scheme Manager

Modul má za úlohu spracovávať binárne balíky, dekódovať príkaz a na základe príkazu vykonať jednu z nasledujúcich operácii:

- Vymenovanie dostupných schém
- Zaslanie schémy
- Vykonanie udalosti

Modul *"Binary Scheme Encoder"* použije, ak je potreba z balíku dekódovať schému a naopak. Na kontrolu správnosti a modifikáciu schém požadovaných modulom *"Process Manager"*, extrahovanie dát a udalostí sa využíva rozhranie modulu *"Session Processor"*. V prípade, že má vykonať udalosť, tak z nej extrahuje údaje a zasiela ich do aplikácie pomocou modulu *"Process Manager"*.

Závislosti:

- **Session Manager**
  - zasielanie odpovedí na udalosti
  - zasielanie zoznamu schém
  - zasielanie modifikovaných schém
- **Binary Scheme Encoder**
  - kódovanie schémy
  - dekódovanie schémy
- **Scheme Processor**
  - kontrola správnosti schémy

  - modifikácia schémy
  - extrahovanie dát

- **Process Manager**
  - zasielanie udalosti

4.1.4 Binary Scheme Encoder

Modul zabezpečuje kódovanie schém do a ich dekódovanie z binárnej optimalizovanej podoby.

Závislosti:

- **Virtual Scheme Manager**

4.1.5 Scheme Processor

Modul spracováva požiadavky na kontrolu správnosti schém, na ich modifikáciu a na extrahovanie dát.

Závislosti:

- **Virtual Scheme Manager**

4.1.6 Process manager

Modul ponúka rozhranie pomocou, ktorého aplikácie vykonávajú operácie na schémach, registrujú udalosti a reagujú na udalosti zaslaním odpovede. Modul sleduje existenciu registrovaných aplikácií. V prípade, že ich inštancia zanikne, automaticky zašle modulu *"Virtual Scheme Manager"* požiadavku na zrušenie prislúchajúcich schém. Jeho

dôležitou úlohou je spúšťať udalosti v rámci aplikácií.

Závislosti:

- **Virtual Scheme Manager**
  - zasielanie odpovedí
  - zasielanie požiadavky na aktualizáciu schémy
  - registrácia schém
  - zasielanie požiadavky na zrušenie schémy

4.2 Moduly kontajneru plochy

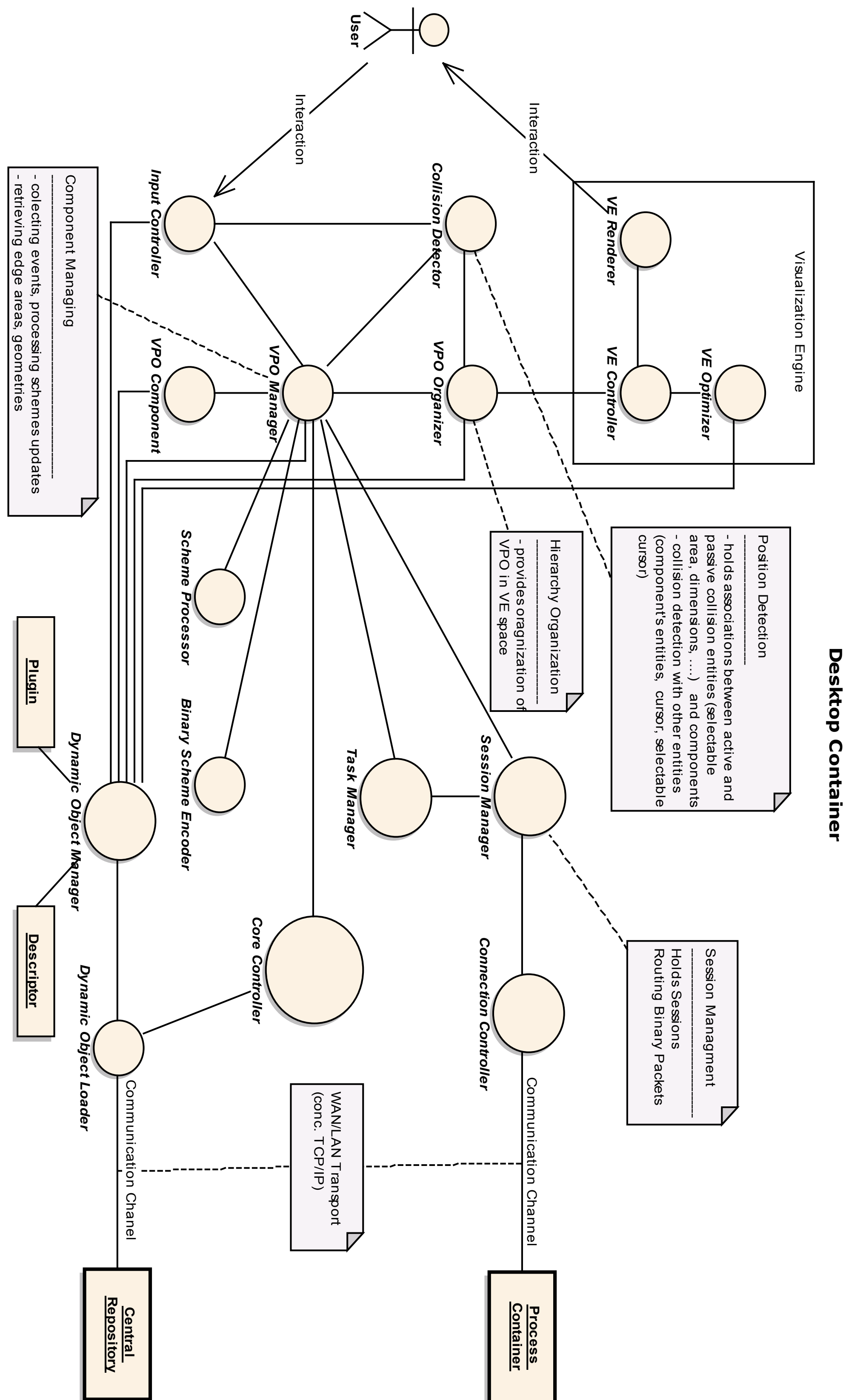
4.2.1 Connection Controller

Modul prijíma požiadavky modulu *"Session Manager"* na vytvorenie spojenia so vzdialeným systémom*. Ak spojenie bolo prerušené, snaží sa ho obnoviť. V prípade, že zlyhá, zasiela modulu *"Session Manager"* informáciu o prerušení spojenia.

Závislosti:

- **Session Manager**
  - Spätné volanie o prerušení spojenia
  - Spätné volanie o prijatí balíka

*znovu podotýkam, že sa môže jednať aj o lokálny systém



obrázok 4.2: Vnútorná organizácia kontajneru plochy pomocou modulov

4.2.2 Session Manager

Modul má za úlohu vytvárať a spravovať relácie. Od modulu*"Task manager"* prijíma požiadavku na vyžiadanie zoznamu procesov a im priradeným schém a požiadavku na vytvorenie relácie. Vždy po prijatí spojenia požiada kontajner procesov o schému zaslaním autorizačného balíka. V prípade, že dostane od modulu *"Connection Controller"* správu o prerušení spojenia, informuje o tom modul *"VPO Manager"*.Ten ho môže požiadať o obnovenie spojenia, alebo zrušenie relácie. Taktiež je jeho úlohou smerovať balíky zo spojenia na modul *"VPO Manager"* a naopak. Ak momentálne nemôže balík odoslať, uloží ho do fronty pre budúce možné odoslanie. Modul vykonáva nasledovné príkazy:

- Vytvorenie relácie
- Obnovenie spojenia
- Zatvorenie relácie
- Smerovanie balíkov
- Vyžiadanie zoznamu procesov a im priradených schém.

Závislosti:

- **Connection Controller**
  - Odosielanie balíkov z fronty
  - Zatváranie spojenia
- **Task Manager**
  - spätná volanie o prijatí zoznamu procesov a im príslušných schém.
- **VPO Manager**

  - Odovzdávanie prijatých balíkov zo spojenia
  - Zasielanie požiadavky na zrušenie schémy

4.2.3 VPO Manager

Modul ma úlohu vytvárať inštancie triedy *"VPO Component"*, ktoré zodpovedajú schémam. Modul prijíma binárne balíky, dekóduje ich príkaz a na základe príkazu vykoná jednu z nasledujúcich operácii:

- Vytvorenie schémy a následne objektu *"VPO Component"*
- Aktualizácia schémy v rámci objektu *"VPO Component"*
- Doručenie odpovede
- Zrušenie objektu *"VPO Component"* na základe požiadavky na zrušenie schémy

Predtým však zabezpečí dekódovanie schémy pomocou modulu *"Binary Scheme Encoder"*. Na vytvorenie schémy alebo na jej aktualizáciu, využíva modul *"Scheme processor"*. Taktiež ak chce vytvoriť udalosť pre aplikačnú logiku, vytvorí ju pomocou modulu *"Scheme processor"* a následne ju kóduje do binárnej podoby pomocou modulu *"Binary Scheme Encoder"*. Kódovanú schému pridá do balíka spolu s prislúchajúcim príkazom a zašle ju do aplikačnej logiky pomocou modulu *"Session Manager"*. Taktiež posiela správu o zrušení objektu *"VPO Component"* a odstránení schémy.

Modul musí zabezpečiť simuláciu systémových schém, ktoré produkuje Modul *"Core Controller"*. Princíp je rovnaký ako pri komunikácii s kontajnerom procesu, len nám nevzniká potreba narábať s reláciami. Schémy prijímame už v dekódovanej podobe.

Pomocou modulu *"Dynamic Object Manager"*, na základe prednastavenej témy, načítava deskriptory[†], ktoré obsahujú informáciu o tom, aký jedinečný kľúč dynamického objektu treba priradiť k prislúchajúcemu jedinečnému kľúču kategórie kontajnera alebo komponentu.

Z modulu *"VPO Organizer"* prijíma požiadavky na zber geometrie a hraničných oblastí od inštancií *"VPO Compnonent"*objektov. Okrem toho sa mu zasiela extrahovaná hierarchia v schéme, pomocou ktorej rozmiestňuje komponenty v priestore objektu.

Závislosti:

- **Scheme Processor**
  - kontrola správnosti schémy
  - modifikácia schémy
  - extrahovanie dát
  - vytvorenie udalosti
- **Binary Scheme Encoder**
  - kódovanie schémy
  - dekódovanie schémy
- **VPO Organizer**
  - prijímanie požiadavky na zber geometrie a hraničných oblastí
- **Input Controller**

[†]Definíciu deskriptoru uvedieme neskôr

  - prijímanie vstupných udalostí

- **Dynamic Object Manager**
  - načítavanie dynamických objektov

- **Task Manager**
  - spätné volanie o zrušení *"VPO Component"* objektu a odstránení schémy

- **Core Controller**
  - prijímanie systémovej schémy
  - prijímanie aktualizácie systémovej schémy
  - posielanie udalostí systémovej schémy
  - prijímanie odpovedí na udalosti systémovej schémy

Deskriptor

Deskriptor je štruktúra, ktorá nám hovorí, aký dynamický objekt máme načítať. Obsahuje dvojice: jedinečný kľúč dynamického objektu a kľúč kategórie komponentu alebo kontajneru. To znamená, že pre komponent alebo kontajner môžeme mať rôzne implementácie.

VPO Component

Objekt zabezpečuje tvorbu kontajnerov, komponentov pomocou dynamických objektov. Dynamické objekty získava z modulu *"Dynamic Object Manager"* cez jedinečný kľúč dynamického objektu. Stará sa o synchronizáciu dát v rámci komponentov a sleduje či u nich nenastala udalosť. V prípade prijatia odpovede ju podľa cesty doručí patričnému komponentu.

Pre proces vizualizácie získava od komponentov a kontajnerov ich geometrie a hraničné oblasti. Zabezpečuje interakciu s užívateľom pomocou prijatých vstupných udalostí z modulu *"Input Controller"*. Buď priamo, doručením do patričných komponentov alebo nepriamo, nastavením príznakov.

4.2.4 Binary Scheme Processor

Modul zabezpečuje kódovanie schém do a ich dekódovanie z binárnej optimalizovanej podoby.

Závislosti:

- **VPO Manager**

4.2.5 Scheme Processor

Modul spracováva požiadavky na kontrolu správnosti schém, na ich modifikáciu, extrahovanie dát a na vytvorenie udalosti.

Závislosti:

- **VPO Manager**

4.2.6 VPO Organizer

Modul zabezpečuje zber geometrií a hraničných oblastí.

Podľa hierarchie v schéme rozmiestňuje komponenty v priestore pomocou dynamického objektu pre rozmiestňovanie komponentov, čím vytvára kompletné vizuálne geometrie *"VPO Component"* objektov.

Zabezpečuje aj vzájomne rozmiestňovanie kompletných geometrií v priestore, na základe prijatých vstupných udalostí z modulu *"Input Controller"*. Pre túto operáciu využíva funkcionality dynamického objektu pre rozmiestňovanie kompletných geometrií.

Tieto dve operácie vykonáva aj pre hraničné oblasti a vytvára tak kompletné hraničné oblasti a ich rozmiestnenie v priestore. Následne po každej zmene sú zaslané modulu *"Collision Detector"*.

Prepočítaná kompletná geometria je zaslaná modulu *"VE Controller"*.

Závislosti:

- **VPO Manager**
  - zasielanie požiadavky na zber geometrie a hraničných oblastí
- **VE Controller**
  - zasielanie kompletnej geometrie
- **Collision Detector**
  - zasielanie kompletných hraničných oblastí
- **Dynamic Object Manager**
  - načítavanie dynamických objektov

4.2.7 Collision Detector

Modul si udržuje zoznam všetkých kompletných hraničných oblastí. Taktiež drží skupinu hraničných oblastí nástrojov selekcie. Tieto oblasti sú vytvorené na požiadavku modulu *"Input Controller"*. Z tohto modulu prijíma požiadavku na zmenu pozície a na detekciu kolízie konkrétnej hraničnej oblasti nástroja selekcie s množinou kompletných

hraničných oblastí. Následne mu zasiela cestu k zasiahnutému komponentu a zoznam relatívnych pozícií zásahu pre všetky objekty nachádzajúce sa na ceste.

Závislosti:

- **VPO Organizer**
  - prijímanie kompletných hraničných oblastí
- **Input Controller**
  - prijímanie požiadaviek na zmenu pozície a výpočet zásahov

4.2.8 Input Controller

Modul drží nástroje selekcie. Sú vytvorené pomocou dynamických objektov pre nástroje selekcie. Modul zabezpečuje priamu interakciu s užívateľom, následne žiada modul *"Collision Detector"* o výpočet zásahov. Výsledkom toho získa zoznam objektov v schéme. ktorým zasiela pomocou modulu *"VPO Manager"* vstupné udalosti.

Závislosti:

- **Collision Detector**
  - zasielanie požiadaviek na zmenu pozícií a výpočet zásahov
- **VPO Manager**
  - zasielanie vstupných udalostí
- **Dynamic Object Loader**
  - načítavanie dynamických objektov

4.2.9 VE Controller

Modul udržuje aktuálnu kompletnú geometriu. Zodpovedá za jej ohraničenie v rámci média, pomocou ktorého sa zobrazuje užívateľovi vizualizácia geometrie. Pomocou modulu *"VE Optimizer"* získava z kompletnej geometrie optimalizovanú čiastočnú geometriu, ktorú následne zasiela modulu *"VE Renderer"* na výsledne zobrazenie.

Závislosti:

- **VPO Organizer**
  - prijímanie kompletnej geometrie
- **VE Optimizer**
  - zasielanie požiadavky na výpočet čiastočnej geometrie
- **VE Renderer**
  - zasielanie čiastočnej geometrie na výsledné zobrazenie

4.2.10 VE Optimizer

Modul zabezpečuje optimalizáciu kompletnej geometrie, čo znamená, že odstraňuje pre užívateľa neviditeľné časti geometrie a tým vytvára optimalizovanú čiastočnú geometriu.

Závislosti:

- **VE Controller**
  - prijímanie požiadaviek na výpočet čiastočnej geometrie

- **Dynamic Object Loader**
  - načítavanie dynamických objektov

4.2.11 VE Renderer

Modul zabezpečuje vykreslenie geometrie na cieľové médium.

Závislosti:

- **VE Controller**
  - prijímanie čiastočnej geometrie

4.2.12 Dynamic Object Manager

Modul zabezpečuje načítanie dynamických objektov a deskriptorov schém. Vyhľadáva ich v lokálnej databáze pomocou jedinečného kľúča dynamického objektu, alebo pomocou mena deskriptoru. Ak sa mu to nepodarí, snaží sa ho pomocou modulu *"Dynamic Object Loader"* načítať z centrálnej databázy dynamických objektov. Po načítaní získa jeho certifikát a pomocou modulu *"Core controller"* čaká na potvrdenie inštalácie nového dynamického objektu. Ak je inštalácia potvrdená dynamický objekt je nakopírovaný do lokálnej databázy. Situácia, kde dynamický objekt nie je nájdený v centrálnej databáze, sa rieši rovnako ako pri nepotvrdení inštalácie a teda vrátením implicitného dynamického objektu *"Unknown"*.

Závislosti:

- **VPO Manager**
- **VPO Organizer**

- **Core Controller**
  - vyvolanie potvrdenia inštalácie
- **Input Controller**
- **VE Optimizer**
- **Dynamic Object Loader**

4.2.13 Dynamic Object Loader

Modul je určený pre stiahnutie dynamického objektu z centrálnej databázy.

Závislosti:

- **Dynamic Object Manager**
  - prijatie požiadavky na stiahnutie dynamického objektu

4.2.14 Core Controller

Modul slúži na vytváranie systémových hlásení pomocou simulácie schém v rámci modulu *"VPO Manager"*. Je napojený na všetky moduly.

Kapitola 5

Záver a budúca práca

Súčasnému užívateľovi nestačí len práca s lokálnymi službami, ale má záujem používať služby v rámci niekoľkých vzdialených systémov. Existujú rôzne riešenia, ktoré zabezpečujú vzdialený prístup oddelením vizualizácie od aplikačnej logiky, no nie sú ucelené do jednotného prostredia, ktoré by zabezpečilo vysokú adaptovateľnosť a platformovú nezávislosť.

Cieľom práce bolo analyzovať daný problém a navrhnúť model, ktorý by spĺňal všetky stanovené princípy. Abstraktným popísaním vizualných entít sme vytvorili pružný dátový model, ktorý umožňuje efektívne oddelenie vizualizácie od aplikačnej logiky.

Následne vhodnou voľbou použitých technológii a správnym návrhom vnútornej organizácie sme zabezpečili platformovú nezávislosť. Zavedením univerzálneho rozhrania sme umožnili použitie u väčšiny vývojových prostredí. Modulárnym návrhom a zavedením dynamických objektov sme zaručili podporu efektívnej opravy chýb a vysokú adaptovateľnosť zo strany užívateľa.

Spojením všetkých princípov dostávame kompletný návrh, čím sme pokryli väčšinu našich cieľov.

5.1 Budúca práca

Súčasne s prácou bola vyvíjaná implementácia, ktorá pri tvorbe návrhu napomáhala ku prekonávaniu problémov a voľbe vhodných technológií. Z dôvodu veľkej obsiahlosti a teda nekompletnej realizácie, nebola v práci zahrnutá. Plánujem na nej pokračovať formou kompletnej realizácie a rozšírenia návrhu o popis a implementáciu základných vizuálnych entít, čo mi umožní nájsť doposiaľ neobjavené chyby v návrhu.

Bibliografia

[msc,] *COM: Component Object Model Technologies*. Microsoft, http://www.microsoft.com/com/default.mspx.

[cor, a] *The CORBA & CORBA Component Model (CCM) Page*. http://ditec.um.es/ dsevilla/ccm/.

[cor, b] *CORBA Component Model, v4.0, official page*. Object Management Group, http://www.omg.org/technology/documents/formal/components.htm.

[jav, a] *Java™Native Interface (JNI)*. Sun, Microsystems, http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html.

[jav, b] *Java™Platform, Standard Edition*. Sun, Microsystems, http://java.sun.com/javase/.

[msd,] *MSDN - Microsoft Developer Network Library - ATL*. Microsoft, http://msdn2.microsoft.com/en-us/library/t9adwcde(vs.80).aspx.

[rpc,] *RPC: Remote Procedure Call protocol*. http://www.javvin.com/protocolRPC.html.

[wik, a] *Wikipédia - Client-server*. http://en.wikipedia.org/wiki/Client_server.

[wik, b] *Wikipédia - CORBA*. http://en.wikipedia.org/wiki/CORBA.

[wik, c] *Wikipédia - XML*. http://sk.wikipedia.org/wiki/XML.

[msx,] *XML Developer Center*. Microsoft, http://msdn2.microsoft.com/en-us/xml/default.aspx.

[Kačmář, 2000] Kačmář, D. (2000). *Programujeme v COM a COM+*. Computer Press. ISBN 80-7226-381-1.