České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Využití technologií GTK+/libgda/CORBA pro vývoj aplikací nad relační databází

Jiří Veruněk

Vedoucí práce: Ing. Michal Valenta, Ph.D.

Studijní program: Elektrotechnika a informatika magisterský

Obor: Výpočetní technika

leden 2007

Diplomová práce: Využití technologií GTK+/ libgda/ CORBA pro vývoj aplikací nad relační databází.

České vysoké učení technické v Praze
Jiří Veruněk

Vydáno 28. prosince 2006

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu. Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 28. 12. 2006

Abstrakt

Tato práce po teoretické stránce rozebírá různé přístupy k tvorbě aplikací nad relační databází pomocí volně šířitelných nástrojů a implementuje ukázkovou aplikaci využitím jednoho z těchto přístupů.

This diploma thesis is focused on discussing of several approaches to the relational database application development using a free software. It also contains an demonstrating application based on one of them.

Obsah

Úvod xiii
1. Motivace 15
1.1. Proč volně šířitelné nástroje? 15
1.2. Nekompatibilita protokolů 15
1.3. Použité technologie 15
1.3.1. libgda 15
1.3.2. GTK+ 16
1.3.3. CORBA 17
2. Rozbor problematiky 25
2.1. První návrh 25
2.1.1. Dotazovací jazyk 27
2.1.2. Interpretace v C++ 29
2.1.3. Problémy s Any v C++ 31
2.1.4. Interpretace v C++ – pokračování 35
2.1.5. Zhodnocení 38
2.2. Druhý návrh 38
2.2.1. Sestavení spojení 41
2.2.2. Zhodnocení 42
2.3. Třetí návrh 42
2.3.1. Architektura 43
2.3.2. Oddělení funkční části aplikace od graf. rozhraní? 45
2.3.3. Popis vzdálené komunikace a formát dat 48
2.3.4. IDL popis 50
3. Ukázková aplikace 55
3.1. Konfigurace prostředí 60
4. Zhodnocení 63
Bibliografie 65

Seznam obrázků

2.1. Architektura aplikace při použití aplikačně nezávislé vrstvy 25
2.2. Architektura aplikace při použití čistě relačního řešení 38
2.3. Model vzdáleně zpřístupněného uživatelského rozhraní 43
3.1. Schéma ukázkové aplikace 55
3.2. E-R model 57
3.3. Přihlášení 58
3.4. Seznam knih 58
3.5. Obsah košíku 59
3.6. Přehled objednávek 59
3.7. Konfigurace zdroje dat 61

Seznam tabulek

1.1. IDL typy a jejich odpovídající C++ typy 24

Seznam příkladů

1.1. Ukázka CORBA IOR v řetězcové podobě 19
1.2. Dekódovaná CORBA IOR 20
2.1. Jednoduché dotazy v Object Query Language. 27
2.2. Jednoduchý dotaz ve Smalltalku. 28
2.3. Ukázka práce s CORBA DII v C++ 29
2.4. Implementace accessoru vracejícího Any 30
2.5. Ukázka práce s typem Any 31
2.6. Vložení a vyjmutí C++ typu char do, resp. z Any 32
2.7. Chybné a správné vyjmutí řetězce z Any 34
2.8. Vyjmutí předka objektu z Any 35
2.9. Implementace zdvojených accessorů 36
2.10. Příklad specifikace relace v SQL a rozhraní objektu v IDL 39
2.11. Rozhraní sestavení spojení v IDL 42

Úvod

Cílem této práce je zhodnotit použití volně šiřitelných nástrojů pro vývoj aplikací nad relační databází. Důraz byl kladen také na umístění *co největší části funkcionality na server*, aby se eliminovala nutnost opakované implementace při vývoji nového typu klienta, a také aby dodatečná změna v systému nevyžadovala úpravu všech již existujících klientů.

Při tvorbě programů se hojně využívá objektů, zatímco jako datová úložiště slouží relační databáze. Je nutné provádět netriviální transformace mezi objektovým a relačním datovým modelem. Zadání této práce vzniklo před více než 3 lety, v té době nebyly implementace objektově relačních mapování tak používané a známé, jako je tomu dnes. V současné době bychom využili spíše je než v této práci popsaná řešení. Přesto má tato práce smysl, jelikož popisuje zvolené technologie a pokouší se je zhodnotit na různých přístupech v tvorbě aplikací.

Původně se tato práce zabývala myšlenkou vytvoření objektové vrstvy nad relační databází pro splnění výše uvedených cílů. Výše zmíněná funkcionalita by tedy byla na serveru umístěna v metodách objektů. Tato cesta se ukázala jako nepoužitelná v důsledku zvoleného implementačního jazyka C++ a s tím souvisejících problémů. Poznatky získané během „výzkumu“ této slepé větve jsou v práci obsaženy.

Pro rozhraní mezi serverem a klienty byl zvolen systém CORBA (Common Object Request Broker Architecture), který představuje poměrně efektivní způsob vzdáleného volání metod objektů. Způsob práce s ním se však v jednotlivých návrzích radikálně měnil, až nakonec dospěl do vzdálené komunikace na úrovni prvků grafického rozhraní.

Součástí práce je ukázková aplikace využívající dalé popsané přístupy.

Kapitola 1. Motivace

V současné době se při vývoji databázových aplikací implementují některé funkce přímo do databázového stroje pomocí procedur (stored procedures). Ty však činí aplikaci závislou na konkrétním databázovém stroji, jelikož tyto procedury nejenže nejsou standardizovány, ale mnoha stroji ani nejsou podporovány. V následujícím textu popsaná řešení se snaží vyhnout jakýmkoliv závislostem na produktech konkrétních výrobců a lze je považovat za alternativní způsob implementace databázových aplikací. Jelikož na použitý relační databázový stroj nejsou kladeny žádné požadavky, lze použít téměř jakýkoliv – i ten nejjednodušší.

1.1. Proč volně šířitelné nástroje?

Volně šířitelné nástroje zažívají v poslední době obrovský rozmach. Důvodem jsou nejen ušetřené licenční poplatky, ale jelikož jsou tyto nástroje často dodávány i se zdrojovým kódem, tak také možnost nahlédnout do zdrojových kódů, příp. je upravit nebo se podílet na jejich vývoji. U jiných nástrojů může hrozit v případě ukončení vývoje, radikální změně licenčních podmínek nebo poplatků až nutnost přepsání existujících aplikací.

1.2. Nekompatibilita protokolů

Relační databáze používají standardizovaný definiční a dotazovací jazyk SQL. Jelikož však nikdo nestandardizoval protokol, kterým klient komunikuje s databázovým strojem, jsou tyto protokoly od jednotlivých výrobců nekompatibilní a klientské aplikace napsané pro jeden databázový stroj nejsou schopny připojit se ke stroji od jiného výrobce. Pro odstranění tohoto nedostatku vzniklo mnoho různých knihoven, které se snaží poskytnout klientské aplikaci unifikovaný přístup ke všem databázovým strojům a tím zlepšit přenositelnost – například ODBC, JDBC, apod. Aplikace využívající takovou knihovnu je schopna pracovat nad kterýmkoliv databázovým strojem, nicméně se může v budoucnu stát, že se změní programátorské rozhraní (API) této knihovny nebo se jen tato knihovna neuchytí a stejně bude potřeba aplikaci přepsat. Podmínkou použití některé podobné knihovny je samozřejmě, aby pro ni existoval „ovladač“ schopný pracovat s požadovaným databázovým strojem.

1.3. Použité technologie

1.3.1. libgda

V této práci byla jako knihovna unifikující přístup k databázi zvolena libgda (GNOME Database Access). Na rozdíl od podobné a mnohem starší ODBC, která implementuje *průnik* funkcí podporovaných jednotlivými stroji, jde libgda cestou *sjednocení*. Při pokusu o použití na konkrétním RDBMS nepodporované funkce je tedy ohlášena chyba.

Knihovna libgda je napsána v programovacím jazyku C a není tedy příliš obtížné vytvořit pro ní mapování i do jiných jazyků. Původně byla ukázková aplikace psána v jazyku C++, později bylo vše přepsáno do jazyka Python, v obou případech šlo jen o různé nadstavby této knihovny.

1.3.2. GTK+

Pro tvorbu grafického rozhraní byla zvolena knihovna *GTK+*. Tato v jazyku C implementovaná knihovna vznikla pro potřeby vývoje grafického editoru GIMP, odtud název GTK+ (GIMP Toolkit). Je multiplatformní, avšak nepoužívá nativní grafické ovládací prvky (widgets) cílové platformy. Se standardně dodávaným vzhledem působí na cílove platformě poněkud nezvykle. Avšak její použití nevyžaduje zakoupení licence a to ani pro tvorbu komerčních programů, narozdíl například od knihovny QT firmy Trolltech.

Nad GTK+ vzniklo několik nadstaveb pro různé programovací jazyky – například gtkmm pro C++, PyGTK pro Python, apod. Bohužel jejich autoři vytvořili mnohdy pouhá mapování, místo aby se snažili vytvořit vrstvu, která by práci s knihovnou přiblížila zvyklostem cílového jazyka.

Grafická uživatelská rozhraní lze s použitím knihovny GTK+ vytvářet několika způsoby. Buď poskládáme jednotlivé grafické prvky do sebe programově nebo vizuálně pomocí návrháře Glade. Program Glade lze ale také používat více způsoby. První možností je nechat si s ním vygenerovat kusy zdrojového kódu – hlavičky metod v C nebo C++, a doimplementovat jen jejich těla, která se k nim jen připojí. Druhou možností je uložit do souboru vzhled aplikace ve formátu XML a s pomocí knihovny libglade si ji za běhu podle tohoto souboru nechat sestavit. Tento přístup přináší následující výhody:

- Odpadá nutnost překladu celé aplikace při změnách grafického rozhraní u překládaných jazyků.
- Návrháři grafického rozhraní nemusejí být programátory.

Použití vizuálního návrháře zrychluje tvorbu aplikací s grafickým uživatelským rozhraním. Časem mi ovšem došlo, že mnohem důležitější než použití vizuálního návrháře jsou inteligentní objekty reprezentující prvky grafického

rozhraní a podpora jazyka pro inteligentní objektový návrh.

1.3.2.1. Definice rozmístění prvků

Vzájemné rozmístění prvků grafického rozhraní se v GTK+ provádí vkládáním těchto prvků do kontejnerů – horizontální box, vertikální box a mřížka. Smyslem těchto kontejnerů je rozdělit definovaným způsobem volný prostor při změně velikosti okna. Pokud by byly prvky umístěny na pevných souřadnicích, nashromáždili by se prvky v levém horním rohu a zbylou oblast až do rozměrů okna by vyplnil celistvý volný prostor.

Ať už do sebe vkládáme prvky programově nebo prostřednictvím návrháře, je nutné pochopit způsob rozdělování tohoto prostoru. U každého kontejneru nastavujeme parametry *homogenní* (homogeneous) a *rozestup* (spacing).[1] Parametrem *homogenní* říkáme, zda má být přidělený prostor rozdělen rovnoměrně mezi všechny prvky, které kontejner obsahuje, či nikoliv. Parametrem rozestup definujeme vzdálenost v obrazových bodech o kterou se každé 2 sousední prvky kontejneru od sebe oddálí.

Každý prvek vkládaný do kontejneru má také své parametry – *expandovat* (expand), *vyplnit* (fill) a *obložení* (padding). Není-li kontejner homogenní, rozdělí se volný prostor mezi ty prvky, u kterých je nastaven parametr expandovat. Nemá-li žádný z nich tento parametr nastaven, vyplní volný prostor zbylou oblast stejně, jako kdyby prvky byly umístěny na pevných souřadnicích. Je-li kontejner homogenní, na hodnotě tohoto parametru nezáleží.

Parametrem *vyplnit* říkáme, zda má prvek vyplnit přidělený prostor nebo si má ponechat minimální velikost a pouze se umístit „do středu" přiděleného prostoru. Nevyplňuje se však celý přidělený prostor. Z tohoto prostoru se nejprve odebere *obložení*. U horizontálního boxu je tato vzdálenost odebrána jen zleva a zprava. Obdobně je tomu u vertikálního boxu. U mřížky jsou definovány 2 parametry – každý pro jeden směr. Ze zbylého prostoru je ještě odebrána *šířka okraje* prvku, tentokrát ze všech směrů. Až po této operaci zbylý prostor je vyplněn prvkem při nastaveném parametru vyplnit.

Vypadá to složitě, ale není. Je mnohem lepší si to vyzkoušet v návrháři než to zde popisovat. Nicméně je důležité alespoň tušit význam těchto parametrů.

1.3.3. CORBA

Při vývoji aplikací je vhodné oddělit část vykonávající požadované funkce od

[1] Tyto anglické názvy, resp. jejich české překlady v lokalizovaném prostředí, nalezneme v návrháři Glade.

uživatelského rozhraní. Zvyšuje se tím nezávislost jedné části na druhé, která v budoucnu v případě potřeby umožní snadnější výměnu některé části, a zároveň zvyšuje znovupoužitelnost v případě požadavku implementace dalších uživatelských rozhraní. V klasických aplikacích bývá model implementován ve formě knihovny, která se připojí k části pracující s uživatelským rozhraním. V případě distribuované aplikace můžeme model umístit na server, který se tím stává tzv. aplikačním serverem, a přistupovat k němu z klientských aplikací některými z prostředků síťové komunikace – BSD sokety, CORBA, Sun RPC, XML RPC, Java RMI, apod. Cílem této práce je prozkoumat systém CORBA, který byl zvolen pro své vlastnosti:

- Odstraňuje nutnost nákladného vývoje vlastního protokolu jako v případě použití BSD soketů.
- Je poměrně efektivní ve srovnání s ostatními technologiemi vzdáleného volání metod – XML RPC, Sun RPC, apod.
- Při volání metody objektu uloženého lokálně v rámci jednoho Object Request Brokeru, je toto volání vykonáno přímo. Odpadá tím režie spojená s tzv. marshallingem.
- Umožňuje spolupráci objektů implementovaných v různých programovacích jazycích díky standardizovaným mapováním jazyka IDL do těchto jazyků.

1.3.3.1. Stručný popis

Základním pilířem CORBA jsou moduly ORB (Object Request Broker), které provádí překlad volání metod a jejich parametrů do síťové komunikace. Nejčastěji se CORBA používá tak, že ve speciálním jazyce IDL (Interface Definition Language) vytvoříme pro každý objekt popis *rozhraní*, kterým lze k objektu vzdáleně přístupovat. Rozhraní je popsáno prototypy metod (hlavičkami) – specifikujeme název metody, typ návratové hodnoty, typy parametrů a také zda daný parametr je vstupní, výstupní nebo vstupně-výstupní. Typem parametru a návratové hodnoty mohou být kromě objektů specifikovaných rozhraními také primitivní datové typy, speciální typ sekvence libovolného typu a také typ Any. V místě, kde se očekává datový typ Any, lze přiřadit libovolný datový typ, ve většině implementací je ale nutné provést explicitně konverzi. Mezi primitivní datové typy patří například řetězec, struktura (záznam) a různé typy čísel. Rozhraní můžeme uzavřít do *modulu*, což je pouze jmenný prostor. V IDL popisu se nevyskytují žádné instanční proměnné nebo omezení přístupových práv k metodám. Jazyk IDL podporuje vícenásobnou dědičnost rozhraní.

Překladačem jazyka IDL dodáváného k ORB vygenerujeme z tohoto univerzálního popisu rozhraní, tzv. *stub* a *skeleton* v požadovaném programovacím jazyku. Stub je reprezentací vzdáleného objektu na straně volající strany, jeho úkolem je ve spolupráci s ORB převést volání metody do síťové komunikace, tomuto procesu se někdy říká *serializace*. Serializaci parametrů metody pak říkáme *marshalling*. Skeleton je reprezentací objektu na straně implementační, jeho úkolem je ve spolupráci s ORB provést opačný proces – *deserializaci* s *unmarshallingem* parametrů. Objekt s implementací metod, jejichž prototypy byly definovány v IDL, nazýváme *servantem*. V některých programovacích jazycích skeleton deleguje požadavky na servanta, v C++ a Pythonu servant dědí ze skeletonu. V dynamických programovacích jazycích ještě existuje možnost nechat vytvořit stuby a skeletony až za běhu programu importováním IDL popisu.

Další možností práce s CORBA je použítí tzv. DII (Dynamic Invocation Interface), případně DSI (Dynamic Skeleton Interface). Při tomto způsobu sice není potřeba vytvářet popis rozhraní v jazyce IDL, ale veškeré informace o prototypu metody, které bychom jinak specifikovali IDL popisem, je nutné dodat v okamžiku jejího zavolání! Tento způsob volání metod je také několikanásobně pomalejší než použití staticky vygenerovaných stubů a skeletonů.

1.3.3.2. CORBA IOR

Chceme-li s objektem něco provádět, musíme na něj nejprve získat referenci. Pomocí této reference ORB ví, kde má požadovaný objekt hledat. Reference v systému CORBA označujeme IOR (Interoperable Object Reference). Lze je převést do řetězcové podoby, přepravit nějakým způsobem na požadované místo a převést zpět na referenci. Další možností je zaregistrovat IOR do jmenné služby.

Příklad 1.1. Ukázka CORBA IOR v řetězcové podobě

```
IOR:010000001c00000049444c3a52656d6f74654755492f536572766572
4170703a312e3000010000000000000064000000010102000c0000003139
322e3136382e312e330084bd00000e000000fe135f3f4400000802000000
000000000200000000000000080000000100000000545441010000001c00
000001000000010001000100000001000105090101000100000009010100
```

IOR se skládá ze tří částí:

- *Repository ID* Standardizovaná položka, slouží k typové kontrole.

- *Endpoint Info* Standardizovaná položka, poskytuje informace pro klienta. U IIOP (Internet InterORB Protocol) obsahuje například IP adresu a port.
- *Object Key* Proprietární položka obsahující informace pro server, jedná se například o adresu objektu v adresním prostoru serveru.

Poznámka

Položek Endpoint Info a Object Key muže IOR obsahovat více. ORB klienta si pak zvolí pro něj nejvhodnější protokol, kterým kontaktuje ORB serveru. Proto má IOR tak ohromnou délku.

Programem **ior-decode-2** dodávaném k brokeru omniORB lze výše uvedenou řetězcovou podobu IOR dekódovat. Jedná se o referenci rozhraní ServerApp modulu RemoteGUI.

Příklad 1.2. Dekódovaná CORBA IOR

```
Object ID: IDL:RemoteGUI/ServerApp:1.0
IOP_TAG_INTERNET_IOP: GIOP 1.2 192.168.1.3:48516
    object_key (14) 'fe135f3f44000008020000000000'
    Unknown component 0

    Unknown component 0x1
```

1.3.3.3. Jmenná sužba – COS Naming Service

Práci se získáváním referencí ulehčuje jmenná služba. Jakoukoliv referenci můžeme svázat se jménem, a potom ji z jiných počítačů podle tohoto jména získat. Prostor jmenné služby si můžeme představit jako strom, jehož každý vnitřní uzel reprezentuje kontext a každému kontextu lze pod „řetězcem“ zaregistrovat jiný kontext nebo referenci (IOR).

Poznámka

Jelikož je možné zaregistrovat kontext, který je již zaregistrován pod jiným jménem, je jmenný prostor ve skutečnosti grafem a ne stromem, nicméně ve většině případů této možnosti nevyužijeme.

Jméno je složeno z *komponent*, které určují uzly stromu na cestě z kořene. Každá komponenta se skládá z atributů *id* a *kind*. Do atributu *id* zadáváme libovolný identifikátor. *Kind* se používá pro popis objektu, formát tohoto atri-

butu také není definován a je pouze na aplikaci, k čemu tento atribut využije. Při vyhledávání objektu ve jmenné službě se vyžaduje shoda obou těchto atributů.

Hovoříme-li o jménu, máme na mysli buď *„strukturovanou“* nebo *řetězcovou* reprezentaci.

- *Strukturovaná reprezentace* je definována v IDL jako sekvence struktur NameComponent.
- *Řetězcová reprezentace* jména je například `RemoteGUI.Framework/Example.Application`. Zde je pod jménem `RemoteGUI.Framework` zaregistrován kontext a v něm je pod jménem `Example.Application` zaregistrována IOR. Lomítko odděluje jednotlivé komponenty, tečka odděluje výše popsané atributy id a kind. Mezi těmito reprezentacemi lze převádět pomocí metod `to_name` a `to_string`, které jsou trochu nelogicky umístěné v objektu NamingContext, přesněji v NamingContextExt.

Metody jmenné služby jsou ve svém standardizovaném mapování příliš nízkoúrovňové, práce s nimi je velmi nepohodlná a je tedy nutné si vytvořit vlastní objekt s metodami vyšší úrovně. S metodami `resolve` a `resolve_str`, které vracejí IOR dle strukturované, resp. ve druhém případě řetězcové reprezentace jména, se pracuje ještě dobře – je možné jim předat celou cestu od kořene (jméno). S ostatními metodami je to horší – při registraci objektu nebo kontextu je například nutné, aby existující kontexty na cestě již byly vytvořeny. Je tedy nutné předtím postupně všechny kontexty na cestě od kořene explicitně otestovat, zda již existují, a pokud ne, tak je vytvořit. V tomto případě tedy nelze předat celé jméno, ale vždy pouze jeho jednu komponentu.

Jméno je pouze lokálním identifikátorem v rámci jednoho jmenného serveru. CORBA ani neposkytuje žádný systém kořenových jmenných serverů obdobných například službě DNS. Je zřejmé, že získání umístění serveru jmenné služby se musí řešit jiným způsobem – statickým nastavením v konfiguračním souboru všech ORB.

V konfiguračním souboru ORB v části InitRef lze nastavit několik IOR a každou z nich svázat s identifikačním řetězcem, pomocí kterého ji lze z aplikace získat metodou `resolve_initial_references(řetězec)` objektu ORB. Jelikož má jmenná služba také svoji IOR, lze takto svázat i ji, v tomto případě bývá identifikačním řetězcem `NameService`. S tímto jménem buď můžeme svázat přímo řetězcovou reprezentaci IOR nebo specifikaci objektu ve formátu *corbaloc*, resp. *corbaname*.

Formát *corbaloc* vznikl, jelikož se člověku s řetězcovou podobou IOR špatně pracuje. Umožňuje specifikovat cestu k objektu pomocí cesty k ORB, ve kterém se tento objekt nachází, a *klíče* pomocí kterého ORB tento objekt dohledá. V případě IIOP je cesta specifikována doménovým jménem, resp. IP adresou, a portem. Klíčem je položka *Object Key*, která již byla zmíněna v popisu struktury CORBA IOR. Většinou v této položce IOR bývá adresa paměti, nicméně v případě jmenné služby je v této položce řetězec `NameService`.[2] Velmi zjednodušená definice formátu corbaloc vypadá asi takto:

```
corbaloc:[PROTOKOL]:[SERVER[:PORT]][/OBJECT_KEY]
```

Známe-li tedy doménové jméno serveru, na kterém běží jmenná služba, je vhodné do konfiguračních souborů ORB zadat místo IOR například následující záznam:

```
InitRef = NameService=corbaloc::corbans.domena.cz/NameService
```

Neuvedeme-li žádný protokol, použije se IIOP. Existuje také protokol *rir*, který je ekvivalentem zavolání již dříve popsané metody `resolve_initial_references(OBJECT_KEY)`.

Poznámka

Speciální DNS záznam typu SRV se serverem provozujícím jmennou službu CORBA pro danou doménu definován není.

Formát *corbaname* vznikl rozšířením corbaloc, aby bylo možné specifikovat objekt zaregistrovaný ve jmenné službě.

```
corbaname:[PROTOKOL]:[SERVER[:PORT]][#JMÉNO_VE_JMENNÉ_SLUŽBĚ]
```

Reference objektů ve formátech corbaloc a corbaname lze použít všude, kde jsme doposud hovořili o IOR. Z toho plyne, že ve jmenné službě lze zaregistrovat reference jiných jmenných služeb[3] ve formátu corbaloc, vytvořit tak

---

[2] Program **ior-decode-2** tento řetězec vypíše v šestnáctkové soustavě jako `4e616d6553657276696365`, jelikož nedokáže s jistotou určit, zda se nejedná o adresu paměti.

[3] Nekonečnou smyčku registrací refence ve formátu corbaname na sebe sama vytvořit nelze, protože ORB se nejdříve pokouší objekt získat a až potom vytváří záznam ve jmenné službě.

z nich hierarchii a ve všech ORB nastavit cestu ke jmenné službě nejvyšší úrovně. Pokud nám nestačí jediný server jmenné služby, jeví se toto řešení jako vhodné. Získávání objektů z neprovázaných jmenných služeb by totiž bylo poměrně těžkopádné a samozřejmě bychom na každém ORB museli zadat referenci na každou z nich.

1.3.3.4. POA manažer

Nyní, když již máme referenci vzdáleného objektu, můžeme zavolat metodu. K tomu ovšem musí na vzdáleném ORB běžet proces, který bude přijímat žádosti o spuštění metod a delegovat je na servanta. Tento proces se nazývá POA[4] Manager. K pouhému volání metod vzdálených objektů není POA manažer potřeba.

Poznámka

Uvědomme si, že distribuovanou aplikaci nemusíme rozdělit jen na stroje, které metody pouze volají (klient), a na jiné, které je pouze vykonávají (server). Aktivujeme-li POA manažera na všech strojích, mohou být objekty aplikace rozmístěny na všech strojích a serverová část aplikace tedy může volat i metody objektů klientské části.

1.3.3.5. Mapování CORBA IDL na C++

Pro pochopení příkladů uvedených v této práci je nutné se zmínit o mapování rozhraní specifikovaných v IDL do C++. Mějme například interface Employee. K reprezentaci tohoto objektu můžeme v C++ použít buď proměnnou typu Employee_var nebo Employee_ptr. Rozdíl mezi nimi je pouze v tom, že Employee_var automaticky uvolní paměť, jakmile proměnná nebude potřeba. U Employee_ptr musíme uvolnit paměť explicitně zavoláním `CORBA::release()`. Stejnou správu paměti zajišťuje v novějších C++ překladačích šablona auto_ptr. Ta však v době vzniku mapování IDL nebyla příliš rozšířená. Přiřazujeme-li _var do _var, vytváří se hluboká kopie. Přiřazujeme-li _ptr do _var, paměť po _ptr uvolnit nesmíme, jelikož _var tím převzala její vlastnictví a je odpovědná za její uvolnění. Přiřazujeme-li _var do _ptr, kopie se nevytváří, o uvolnění paměti se postará _var. V následující tabulce si všimněte, že na typ _ptr je mapován pouze IDL typ interface, ostatní IDL typy, včetně těch, co nejsou v tabulce uvedeny vůbec, jsou mapovány na standardní primitivní typy jakyka C++.

[4] Akronym POA znamená Portable Object Adapter. V době před CORBA 2.2 se používal BOA (Basic Object Adapter), jeho nevýhodou bylo, že nestandardizoval, jak má vypadat skeleton, nebylo tedy možné bez problémů přenést serverovou část k ORB od jiného výrobce.

Tabulka 1.1. IDL typy a jejich odpovídající C++ typy

typ v IDL	**typ v C++**	**obalující typ v C++**
string	char *	CORBA::String_var
any	CORBA::Any	CORBA::Any_var
interface foo	foo_ptr	class foo_var
struct foo	struct foo	class foo_var
union foo	class foo	class foo_var
typedef sequence<X> foo[10];	class foo	class foo_var
typedef X foo[10];	typedef X foo[10];	class foo_var

Přiřazujeme-li const char * do String_var, vytvoří se kopie. Pokud však přiřazujeme char *, kopie se nevytvoří a přiřadí se pouze ukazatel. Jelikož některé starší překladače implementují řetězcové literály typem char *, musíme každé takovéto přiřazení pro jistotu přetypovat na const char *.

Kapitola 2. Rozbor problematiky

V následujících částech práce budou zmíněny tři možné přístupy k realizaci ukázkové aplikace. První návrh se zabývá vytvořením objektově relačního mapování, druhý čistě relačním řešením a třetí je založen na vzdáleném zpřístupněním prvků grafického rozhraní. Ukázková aplikace, kterou je informační systém internetové knihkupectví, je pouze prototypem na kterém chci zhodnotit použitelnost těchto přístupů, neklade si za cíl být kompletním řešením, které by šlo nasadit do praxe.

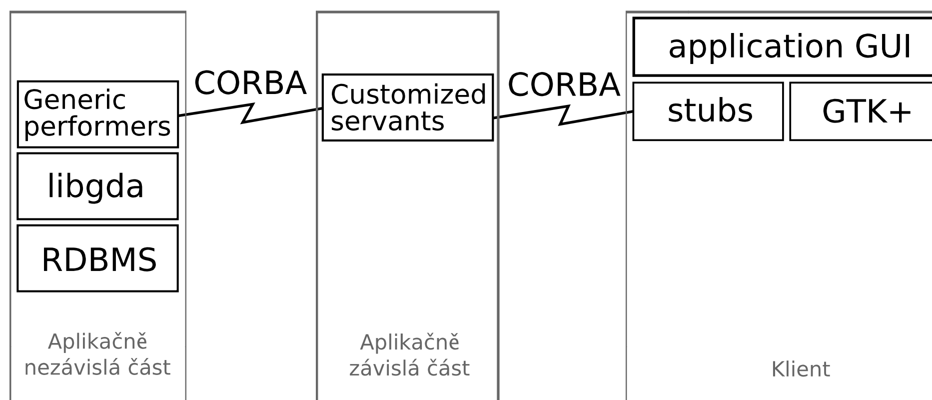
2.1. První návrh

V prvním návrhu bylo záměrem vytvořit aplikačně nezávislou část, kterou by bylo možné použít i pro jiné úlohy, a s použitím této části pak implementovat konkrétní ukázkovou aplikaci. Jedním ze způsobů, jak tento úkol splnit je vytvořit vrstvu, která bude mapovat objektovou databázi na relační.[1] Pro klienta by se tato vrstva chovala jako objektová databáze a přitom vlastní data by byla uložena v relační databázi. Stejný přístup používají v současné době velmi používaná objektově relační mapování. V době, kdy jsem se tímto problémem začal zabývat, tomu tak nebylo.

Objektové databáze umožňují implementovat metody s dotazy přímo na serveru z důvodu vyšší rychlosti, příp. bezpečnosti. Aby bylo možné tyto metody i klienta implementovat v libovolném programovacím jazyku, je vhodné systém postavit na dvou CORBA rozhraních. Jedno zprostředkuje komunikaci mezi klientem a aplikačně závislou částí, druhé mezi aplikačně závislou a aplikačně nezávislou částí. Jelikož toto druhé rozhraní bude probíhat lokálně na serveru, nemělo by způsobovat vyšší zpoždění – již dříve jsme si uvedli, že volání metody v rámci jednoho ORB se provádí přímo bez režie spojené s marshallingem.

Obrázek 2.1. Architektura aplikace při použití aplikačně nezávislé vrstvy

[1] Relační databáze jsou totiž nejrozšířenější. Volně šířitelné objektové databáze ani skoro neexistují.



Při tvorbě aplikačně závislé části bychom si pomocným programem, který by získal schéma relační databáze přes libgda, nechali vygenerovat IDL popis objektové databáze. Tento popis bychom mohli ručně obohatit o hlavičky dodatečných metod zmíněných v předchozím odstavci. Vygenerované IDL by totiž obsahovalo pouze tzv. *accessory*, tj. metody přiřazující instančním proměnným hodnotu, resp. tuto hodnotu vracející. Instanční proměnné by v relační databázi byly mapovány na „sloupce“, tj. položky záznamu.

IDL popis bychom pak překladačem jazyka IDL přeložili do jazyka, ve kterém bychom chtěli implementovat těla metod. Bohužel musíme kromě dodatečných metod implementovat i *accessory*, jelikož není možné dědit ze vzdáleného objektu. Ty by však pouze delegovaly požadovanou akci na „obecného vykonávače“ – předaly by mu dotaz, který má vykonat. Formát tohoto dotazu je definován dotazovacím jazykem, kterému se podrobněji věnuje následující kapitola. Referenci na vykonávače by objekty získaly například pomocí jmenné služby.

Před delegováním první akce na obecného vykonávače je nutné sestavit spojení s objektovou databází. Při obdržení požadavku připojení k databázi by objekt aplikačně nezávislé části vytvořil objektovou reprezentaci databáze tím způsobem, že například pro relaci *Employee* by si ze jmenné služby[2] vyžádal instanci zaregistrovanou jako *EmployeeClass*, jejíž metoda `new` vrací instanci třídy `Employee`.

Poznámka

Hovoříme stále o instancích, jelikož v CORBA IDL není možné specifikovat třídní metodu.

Reference těchto objektů by byly do jmenné služby zaregistrovány aplikačně

[2] Je samozřejmě možné neregistrovat vše ve jmenné službe a volat jen metodu `getObjectNamed_("EmployeeClass")` nějakého objektu, který by například mohl udržovat „seznam“ těchto objektů.

závislou částí při startu. Instance třídy `Employee` by poté byla „spojena“ s aktuálním záznamem relace.

Toto spojení je možné provést následujícími způsoby.

- *Předat instanci pouze hodnotu primárního klíče.* Toto řešení je nejjednodušší. Nevýhodou je nutnost přístupu do relační databáze s každým přístupem do instanční proměnné, což by se patrně projevilo v nízké rychlosti celého řešení.
- *Předat instanci hodnotu primárního klíče i položek záznamu.* Nevýhodou tohoto řešení jsou problémy se zajištěním konzistence dat držených těmito objekty s daty uloženými v databázi.

2.1.1. Dotazovací jazyk

Dotazovacím jazykem objektové databáze by mohl být Object Query Language, který vychází z jazyka SQL. V takovém případě by z pohledu klientské aplikace byl v systému přítomen objekt odpovědný za provádění dotazů. Bylo by samozřejmě nutné implementovat interpret tohoto jazyka.

Příklad 2.1. Jednoduché dotazy v Object Query Language.

```
SELECT p.name
FROM p in People
WHERE p.age > 26
```

```
SELECT t.name
FROM ( SELECT t From Tutors WHERE t.salary > 300 ) r,
     r.students s
WHERE s.fee > 30;
```

První dotaz vypíše jména všech lidí starších 26 let, druhý jména všech lektorů s platem vyšším než 300, jejichž některý student platí více než 30.

Druhou možností je použití dotazovacího jazyka odvozeného ze Smalltalku, který je používán například v databázi GemStone. Samozřejmě by se nejednalo o plnohodnotný Smalltalk, pouze chci demonstrovat odlišný přístup k dotazům.

Příklad 2.2. Jednoduchý dotaz ve Smalltalku.

```
tutors select: [ :aTutor |
  (aTutor salary > 300) and: [
    aTutor students contains: [ :aStudent |
      aStudent fee > 30
    ]
  ]
] collect: [ :aTutor |
  aTutor name
]
```

Poznámka

Ponechme stranou, že by se v objektovém modelu ve skutečnosti asi nevyskytoval výše zmíněný `collect:`. Nezajímají nás totiž jména, ale celé objekty. Při tvorbě GUI bychom pak pro zobrazení jejich jmen použili Adaptory, které by dokázali jména z těchto objektů získat. Chtěl jsem pouze uvést stejný dotaz v OQL i ve Smalltalku.

V tomto případě zde neexistuje centrální vykonávač dotazů, na kterého by se klient obracel, jako v předchozím případě. Relace jsou reprezentovány objekty (kolekcemi), které rozumí zprávám `select:`, `collect:`, `inject:into:`, `contains:`, apod.

- `select:` aBlock – vrací kolekci prvků vyhovujících podmínce.
- `collect:` aBlock – vrací kolekci návratových hodnot bloku vyhodnocovaném s prvky původní kolekce.
- `inject:` initialValue `into:` aBlock – vyhodnocuje blok s postupným předáním všech prvků kolekce jako parametr tohoto bloku s mezivýsledkem inicializovaným na počáteční hodnotu `initialValue`.

Tyto metody přijímají jako jeden ze svých parametrů blok. V CORBA IDL bychom tento blok reprezentovali řetězcem, který bychom museli interpretovat. Všechny objekty v tomto dotazu musíme nahradit řetězcovou reprezentací CORBA IOR.

V příkladu 2.2 – „Jednoduchý dotaz ve Smalltalku.“ žádný objekt za IOR nahrazovat na straně klienta nemusíme, `aTutor` je totiž parametr bloku, za který budou na serveru postupně nahrazovány prvky kolekce.

2.1.2. Interpretace v C++

Při implementaci interpretu dotazovacího jazyka v programovacím jazyku C++ naražíme na problém, který spočívá v nemožnosti C++ zavolat metodu, jejiž název je předán jako parametr například ve formě řetězce. Tento problém by se dal vyřešit pomocí *Dynamic Invocation Interface*, které slouží k volání metod, pro něž neexistuje specifikace v IDL. Bohužel musíme specifikovat i typy parametrů a návratové hodnoty.

Poznámka

Další možnou cestou by bylo vytvoření nějakého registrátora, do kterého bychom museli explicitně zaregistrovat ukazatele (paměťové adresy) všech metod. Při požadavku zavolání funkce bychom z tohoto registrátora získali ukazatel, který bychom museli přetypovat na správnou funkci a zavolat. Tomuto nepříliš bezpečnému řešení se věnovat nebudu.

Příklad 2.3. Ukázka práce s CORBA DII v C++

```
CORBA::String_var arg = (const char *) "parametr";
CORBA::Request_var req = obj -> _request("název_metody");

req -> add_in_arg() <<= arg;
req -> set_return_type(CORBA::_tc_string);
req -> invoke();

if ( req -> env() -> exception() ) {
  cout << "Exception was thrown!" << endl;
  return;
}

const char *ret;
req -> return_value() >>= ret;
```

V dotazu reprezentovaném řetězcem ovšem typy nemáme, a tak interpret nemůže vědět, jaké typy má DII dodat. Pokud by specifikace datových typů byla součástí dotazu, asi by se s celým systémem špatně pracovalo. Tyto problémy nastávají u všech volání metod, ať už se jedná o přístupové delegující metody (accessory), metody dodatečně přidané nebo metody kolekcí, které jsme zmínili v sekci 2.1.1 – „Dotazovací jazyk“. Při implementaci metody kolekce dokonce narážíme na problém, že nevíme jakého typu budou objekty, které bude tato kolekce obsahovat.

CORBA poskytuje datový typ *Any*, který dokáže zapouzdřit hodnotu jakéhokoliv typu, ať už se jedná o instanci třídy nebo hodnotu primitivního datového typu (řetězec, číslo, apod.). Any kromě této hodnoty obsahuje i tzv. *kód typu* (type code). Účelem any je umožnit bezpečně typově předat parametr libovolného typu. Typová bezpečnost je zajištěna právě pomocí kódů typů, jejichž nesoulad je detekován za běhu programu. Je vyžadována úplná shoda těchto kódů – není tedy možné například vložit short a vyjmout long. Any bývá někdy přirovnávána k obecnému ukazateli jazyka C *void* *, na rozdíl od něj však zároveň zachovává typovou kontrolu.

Bohužel tvůrci CORBA při standardizaci mapování typu Any do různých programovacích jazyků nemysleli na snadnost používání, a tak se použití tohoto typu stává doslova noční můrou nejen v C++, ale dokonce i v dynamicky typovaných jazycích jakým je například Python. Chceme-li hodnotu libovolného typu překonvertovat na typ Any, je nutné to provést explicitně a ještě dodat typ hodnoty. Obdobný problém musíme řešit při obráceném procesu. OmniORBpy, jeden z brokerů jazyka Python, naštěstí obsahuje nestandardizované metody `from_any` a `to_any`, které velmi výrazně ulehčují práci s tímto typem. Explicitní konverzi sice neodstraňují, ale není již nutné dodávat datový typ konvertované hodnoty a dokonce lze těmito metodami jednoduše převést i vnořené kolekce.

Můžeme tedy přistoupit k tomu, že bychom typ Any použili u všech metod pro parametry i pro návratové hodnoty. Již tedy víme, jaký typ DII dodat. Do Any bychom vkládali primitivní typy i reference objektů.

Poznámka

Také by bylo možné používat jen reference objektů, ale k tomu by bylo nutné vytvořit kompletní objektovou hierarchii zpřístupněnou vzdáleně, která by nahradila primitivní typy. To by bylo nejen ohromně pracné, ale nejspíš bychom i zvyšili režii vlivem vzdáleného přístupu k úplně všem objektům.

Přístupová metoda (accessor) by mohla být implementována v C++ tak, jak je uvedeno v příkladu 2.4 – „Implementace accessoru vracejícího Any“. Již jsme se zmínili, že tyto metody delegují požadavek na „obecného vykonávače“.

Příklad 2.4. Implementace accessoru vracejícího Any

```
CORBA::Any *
Employee::name()
{
  CORBA::Object_var thisIOR = this -> _this(); // vrací IOR
```

```
  return genericPerformer -> standardMethod_on_("name",
                                               thisIOR);
}
```

K masivního použítí typu Any jako je tomu v tomto příkladu je vhodné poznamenat, že v místě každého volání metody naší pseudoobjektové databáze je nutné získat z typu Any konkrétní typ, který očekáváme, a to nejen například při volání metody `select:`, ale i při každém volání accessoru. Toto musíme řešit nejen v aplikačně závislé části při implementaci dodatečných funkcí, ale i v klientovi – viz příklad 2.5 – „Ukázka práce s typem Any“.

Příklad 2.5. Ukázka práce s typem Any

```
char *
SomeObject::someFunction()
{
  CORBA::Any_var any;
  char *name;
  // Employee_var employee = ...

  any = employee -> name();
  if (! any >>= name)
    throw SOME_EXCEPTION();
  return name;
}
```

V předchozím příkladu jsme použili zvláštní C++ operátor >>=, ten slouží k získání hodnoty z typu Any. Konverze proběhla úspěšně, je-li výsledek true. K opačnému procesu, tj. ke vložení hodnoty do typu Any, slouží operátor <<=, v některých jazycích je nutné dodat i typ vkládané hodnoty.

Poznámka

Možná se ptáte, proč byly definovány právě tyto operátory a ne << a >>. Důvodem je priorita operátorů v C++, <<= a >>= ji mají stejně nízkou jako operátor přiřazení, zatímco u << a >> je příliš vysoká.

2.1.3. Problémy s Any v C++

Situace je však bohužel ještě komplikovanější. Vkládání, resp. vyjmutí, hod-

noty je v C++ založeno na přetěžování operátorů. Z hlediska použití výše uvedených operátorů můžeme datové typy rozdělit do několika skupin.

U první skupiny datových typů jsou operátory *definovány implicitně*. Do této skupiny patří typy short, unsigned short, long, unsigned long, float, double, string (s proměnnou délkou), wstring a any,[3] příp. dodatečně přidané typy, které nemusí být některými ORB podporovány – long long, unsigned long long, long double.

Typům druhé skupiny jsou výše zmíněné operátory *definovány IDL překladačem*, což znamená, že je potřeba mít dostupný IDL popis těchto typů – v C++ dokonce již v době překladu. V tomto případě je pak možné používat operátory stejným způsobem jako výše uvedené. Není-li však tento popis dostupný, je nutné v C++ konstruovat any mnohem složitěji pomocí tzv. *DynAny*. Sem patří uživatelsky definované typy struct, enum a typedef (alias jiného typu). K posledně jmenovanému je potřeba dodat, že při vkládání do Any, se do jejího kódu typu nevloží toto nově definované jméno, ale pouze identifikátor standardního typu. Typová kontrola tedy nezachytí případ, kdy například do any vložíme alias typu string a vyjmeme z ní úplně jiný alias také typu string. Přesněji řečeno IDL překladač vygeneruje správný kód typu, ale při vkládání hodnoty do Any není implicitně použit. Pokud to vyžadujeme, je nutné jej po vložení hodnoty do Any předat metodou `type`.

Třetí skupinou jsou primitivní datové typy, u kterých jsou sice tyto operátory implicitně definovány, nicméně je nutné vkládanou hodnotu přetypovat nebo spíše obalit pomocným typem. Několik IDL typů je totiž mapováno na jeden C++ typ, a proto selhává mechanizmus přetěžování operátorů. Konkrétně IDL typy char, boolean a octet odpovídají v C++ typu char. Způsob práce ukazuje příklad 2.6 – „Vložení a vyjmutí C++ typu char do, resp. z Any“. Mezi další typy s těmito problémy patří wchar, jehož přímé vložení do any nemá definované chování – nestandardizovaný překladač přeloží typ wchar_t na int. U řetězcových typů string, wstring s limitem délky (bounded strings) je nutné předat tento limit parametrem v konstruktoru pomocného typu. Podobně se také pracuje s typem fixed.

Poslední skupina je kombinací druhé a třetí – operátory jsou definovány IDL překladačem, ale je nutné i vkládanou hodnotu přetypovat, resp. obalit pomocným typem. Sem patří například pole fixní délky array, jelikož v C++ jsou pole degenerovány na ukazatel na první prvek.

Příklad 2.6. Vložení a vyjmutí C++ typu char do, resp. z Any

[3] Je možné vložit Any do jiného Any.

```
CORBA::Any a;
CORBA::Boolean  b = 0;
CORBA::Char     c = 'x';
CORBA::Octet    o = 0xff;

a <<= CORBA::Any::from_boolean(b);
a <<= CORBA::Any::from_char(c);
a <<= CORBA::Any::from_octet(o);

a <<= b;      // Chyba při překladu!
a <<= c;      // Chyba při překladu!
a <<= o;      // Chyba při překladu!
```

```
CORBA::Any      a;
CORBA::Boolean  b;
CORBA::Char     c;
CORBA::Octet    o;
if (a >>= CORBA::Any::to_boolean(b)) {
    // Any obsahuje boolean ...
} else if (a >>= CORBA::Any::to_char(c)) {
    // Any obsahuje char
} else if (a >>= CORBA::Any::to_octet(o)) {
    // Any obsahuje octet
} else {
    // Any obsahuje hodnotu jiného typu.
}
a >>= b;    // Chyba při překladu!
a >>= c;    // Chyba při překladu!
a >>= o;    // Chyba při překladu!
```

Doposud jsme hovořili pouze o přiřazování proměnných do Any. Přiřadit literál (např. 34, "Novák", 3,14, apod.) je možné také, ale v případě číselného literálu je nutné jej přetypovat na konkrétní IDL typ. Standardně je typem celočíselného literálu typ int, ten je ovšem na různých architekturách různě velký – je definován jako jeden z typů short, long a long long. Identifikátory těchto typů se při vložení literálu do Any přiřadí do jejího kódu typu a při pokusu o vyjmutí hodnoty z Any na jiné architektuře pak selže typová kontrola. Ke stejnému problému dochází u čísel s pohyblivou řádovou čárkou. V případě vkládání řetězcového literálu přetypovávat nemusíme. Proto je výhodnější literál nejprve vložit do proměné a tu pak přiřadit do Any. Jako typy proměnných použijeme CORBA::UShort, CORBA::Long, CORBA::Double, apod.

Aby problémů nebylo příliš málo, obsahuje mapování CORBA IDL na C++ spoustu různých vyjímek způsobených explicitní správou paměti, na které je třeba pamatovat. Vzpomeňme si na nutnost přetypování řetězcového literálu

na typ const char * při přiřazování do proměnné typu String_var z důvodu, že některé starší překladače tyto literály implementují typem char *. Pokud však místo do String_var přiřazujeme řetězcový literál do Any, přetypovávat nemusíme, jelikož dojde k vytvoření hluboké kopie v případě const char * i char *.

Další problém se správou paměti nastává při vyjímání hodnoty z Any. Není možné vyjmout z Any hodnotu do proměnné typu _var, jelikož by došlo k pokusu o dvojnásobné uvolnění paměti proměnnými typů Any a _var. Operátor >>= nám v podstatě vrací ukazatel na paměť, kterou stále vlastní Any, neměli bychom ji tedy modifikovat a ani ji nesmíme použít mimo oblast platnosti proměnné Any, kdy došlo k uvolnění paměti! Následující příklad demonstruje tyto problémy na vyjmutí řetězce. Na stejné problémy při vyjímání z Any nesmíme zapomenout ani u *objektů* a typů struct, union a sequence. Při vyjímání objektů z Any očekává operátor C++ referenci[4] na typ _ptr, zatímco u ostatních zde uvedených typů očekává ukazatel na jejich primitivní typ.

Pro vložení hodnoty do Any máme k dispozici dva přetížené operátory – jeden pro vložení hodnoty C++ *referencí* a druhý C++ *ukazatelem*. Při vložení C++ referencí _var nebo _ptr dojde k vytvoření hluboké kopie (u objektů zavoláním metody `_duplicate`), zatímco při vložení C++ ukazatelem na _ptr převezme Any odpovědnost za uvolnění paměti[5], na kterou ukazuje tento ukazatel. Tento druhý přístup je vhodný, pokud chceme do Any vložit hodnotu „proměnné“ nebo spíše předem neznámé délky vrácenou návratovou hodnotu nějaké funkce – v této funkci došlo k alokování paměti pro tuto hodnotu. Pokud tento přístup použijeme, nesmíme od okamžiku vložení ukazatele na proměnnou do Any, tuto proměnnou použít (například v případě reference objektu zavolat její metodu)!

Příklad 2.7. Chybné a správné vyjmutí řetězce z Any

```
CORBA::Any a;
a <<= "Hello";
CORBA::String_var msg;
a >>= msg;              // NELZE! Dvojnásobné uvolnění paměti!
```

```
CORBA::Any a;
a <<= "Hello";
const char * msg;              // const char *, nikdy char *
a >>= msg;                     // V pořádku.

msg[0] = 'h';                  // Chyba, msg je konstantní!
```

[4]Pro snížení zmatku v pojmech označuji C++ operátor & jako C++ referenci
[5]U objektů dojde k zavolání metody `release`.

```
CORBA::String_var copy(msg); // Vytvoření hluboké kopie.
copy[0] = 'h';               // V pořádku.
```

Řekli jsme si, že typová kontrola vyžaduje přesnou shodu kódů typu. V případě předávání refencí objektů to znamená, že implicitně není možné místo objektu předat jeho potomka. Tento problém se obchází získáním reference objektu z Any metodou `to_object` a přetypováním tak, jak je uvedeno v příkladu 2.8 – „Vyjmutí předka objektu z Any“.

Příklad 2.8. Vyjmutí předka objektu z Any

```
CORBA::Any a;
Test::SubClassA_var sub = ...;
a <<= sub;                   // Vložení instance potomka
CORBA::Object_var obj;
a >>= CORBA::Any::to_object(obj);  // Vyjmutí jako Object
Test::ClassA_var c;                // Reference na předka
c = Test::ClassA::_narrow(obj);
  // Bezpečné přetypování na předka
```

Jako třešnička na dortu zde působí další vyjímka – v CORBA od verze 2.3 je po vyjmutí reference objektu z Any metodou `to_object` *nutné uvolnit paměť*, proto v předchozím příkladu přiřazujeme do Object_var. Připomeňme si, že u typů struct, union a sequence, a dále také u vyjmutí refence objektu klasickou cestou, tomu je přesně naopak – paměť uvolnit nesmíme. Tedy do verze CORBA 2.2 to bylo jednotné, teď již není!

2.1.4. Interpretace v C++ – pokračování

Vraťme se k příkladu 2.5 – „Ukázka práce s typem Any“. Bude-li v Any hodnota primitivního datového typu, je bohužel nutné při získávání hodnoty z typu Any program rozvětvit podle typu této hodnoty a implementovat obslužné funkce pro všechny možné datové typy, které se v Any mohou objevit. Toto je nutné provádět v jazycích, ve kterých se primitivní datové typy CORBA mapují na skutečné primitivní datové typy daného jazyka, s každým z těchto typů se pracuje jinak.

Za zde uvedený převod z typu Any je bohužel zodpovědný i klient, což komplikuje práci s databází. Uvědomíme-li si, že hlavním důvodem, který nás dovedl k použití typu Any, byla interpretace netypovaného dotazovacího jazyka,

nabízí se řešení v podobě zdvojených accessorů. Accessor vracející primitivní datový typ by byl volán klientem, příp. aplikačně závislou částí serveru, zatímco accessor vracející typ Any by byl volán interpretem dotazů v aplikačně nezávislé části. Samozřejmě by jeden accessor využíval ke své činnosti druhého – viz příklad 2.9 – „Implementace zdvojených accessorů“.

Příklad 2.9. Implementace zdvojených accessorů

```
char *
Employee::nameAsString()
{
  char *name;

  CORBA::Any_var any = this -> name();
  if (! any >>= name)
    throw SOME_EXCEPTION();
  return name;
}
```

```
CORBA::Any *
Employee::name()
{
  CORBA::Object_var thisIOR = this -> _this(); // vrací IOR

  return genericPerformer -> standardMethod_on_("name",
                                                thisIOR);
  // standardMethod_on_ vrací Any
}
```

Metoda `standardMethod_on_` by získala název třídy z IOR předaném v parametru *`thisIOR`*, který by se měl shodovat s názvem relace a vrátila by typ Any s hodnotou atributu (sloupce). Získání názvu třídy můžeme v C++ provést jedním z následujících způsobů:

- Implementovat u všech objektů metodu `className`, jež by vracela řetězec s názvem třídy, tato metoda by musela být „obecným vykonávačem“ zavolána pomocí CORBA DII, jelikož aplikačně nezávislá část nemá k dispozici IDL popis aplikace.
- Máme-li k dispozici Any, je možné z ní metodou `type` získat kód typu, jehož metoda `kind` by měla vracet *tk_objref* a metoda `name` vrátí název třídy.

Poznámka

K problémům s datovými typy je vhodné ještě zmínit, že některé novější ORB podporují přístup k tzv. *Interface Repository*, pomocí kterého by mělo jít získat kompletní IDL popis objektu nebo jakékoliv jiné hodnoty podle *repository ID*. Tato položka je součástí kódu typu a ten je součástí any. Abychom mohli kontaktovat Interface Repository, stačilo by nám tedy mít any s vloženou hodnotou. Je samozřejmě zbytečné tuto funkci používat ke zjištění názvu třídy – z any to jde mnohem jednodušeji výše popsaným způsobem. V současné době není tato funkce příliš rozšířená.

Použití zdvojených accesorů přináší do návrhu důležitou vlastnost – klient i aplikačně závislá část očekává jeden konkrétní datový typ, není možné přijmout hodnotu jiného typu. Pokud bychom se s tímto omezením nechtěli spokojit, je nutné volat přímo accesory vracející Any a v každém místě tohoto volání následně rozvětvit program podle typu této hodnoty.

V dynamicky typovaném jazyku s primitivními datovými typy, jakým je například Python, by bylo možné vytvořit obecnou metodu, která by vrátila hodnotu primitivního datového typu vyjmutou z Any. Jelikož se s každým primitivním datovým typem pracuje jinak, musela by i v tomto případě volající strana očekávat hodnotu konkrétního datového typu nebo se rozvětvit výše popsaným způsobem.

Ve staticky typovaných jazycích, jakými jsou například C++ a Java, je potřeba nejen nějaký typ očekávat, ale je nutné ho i metodě dodat. Ať už tento typ „dodáme“ svázáním názvu funkce s konkrétním typem návratové hodnoty nebo použitím syntaktické konstrukce šablon C++, výsledek je skoro stejný. Obecnou funkci, která by vyjmula hodnotu z Any a správně přetypovanou ji vrátila, nelze ve staticky typovaném jazyku vytvořit.

Pokud bychom nepoužívali primitivní typy, tj. kdyby všechny naše typy byly objekty s velmi dobře navrženým rozhraním, mohla by obecná metoda vrátit objekt a klient by se mohl spolehnout na polymorfismus. Jelikož je v C++ polymorfismus podmíněn dědičností, musel by existovat předek všech objektů reprezentujících typy. Vytvoření kompletní objektové hierarchie zpřístupněné vzdáleně jsme již dříve zavrhli z důvodu pracnosti a možného zvýšení zpoždění. Bylo by také možné vytvořit jednoduchou sadu objektů reprezentujících typy pouze lokálně a na které by byly mapovány primitivní typy CORBA. Tato sada objektů by samozřejmě byla implementována v nějakém programovacím jazyku a nešla by tedy využít pro klienty napsané v jiných jazycích.

Poznámka

> Skoro to vypadá, že v případě objektů neplatí výše popsané omezení staticky typovaného jazyka, ale není tomu tak. Ve skutečnosti je vráceným typem vždy předek, pouze je na něm možné volat metody definované v potomkovi mechanizmem virtuálních metod, což mimochodem není nic jiného než překladačem vytvořené rozvětvení programu podle „typu objektu“, kde v každé větvi je zavolána správná metoda.

Z výše popsaného lze shrnout následující závěr. Jednoduše pracovat s objektovou databází lze pouze objektovým modelem výpočtu plně využívajícího polymorfismu. K tomu je nejen potřeba místo primitivních datových typů používat jejich objektové varianty, ale stejně důležité je, že tyto objekty musí mít velmi propracované jednotné rozhraní. Například programovací jazyk Python je sice dynamicky typovaný, ale jeho objekty mají často různá rozhraní, takže se programátor nemůže spolehnout na polymorfismus a musí větvit program podle typu objektu podobně jako v případě použití primitivních typů.

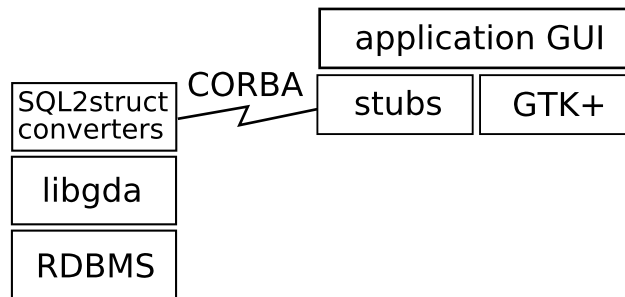
2.1.5. Zhodnocení

Absence čistého objektového modelu výpočtu nebyla samozřejmě jediným důvodem, proč jsem od tohoto přístupu upustil. Příčinou byl spíše souhrn všech komplikací a vyjímek popsaných v tomto textu. Netvrdím, že by popsaným způsobem nešlo pseudoobjektovou databázi implementovat, ale vzhledem k pracnosti pochybuji, že se vynaložené úsilí vyplatí. Aplikačně nezávislá vrstva v podobě objektově relačního mapování dostupného přes CORBA by nám sice možná ušetřila práci při vývoji dalších aplikacích, vytvořili bychom si však spoustu problémů, které toto řešení odsuzují k praktické nepoužitelnosti. O alternativním způsobu implementace pojednává následující kapitola.

2.2. Druhý návrh

Po problémech, které se objevily v prvním návrhu, jsem opustil myšlenku vytvoření produktu, který by se dal využít i při vývoji dalších aplikací, a přešel jsem k jednoúčelovému čistě relačnímu řešení. Tento způsob návrhu byl od počátku zamýšlen jako „záchranná síť“ pro případ, že by se vynořily problémy znemožňující implementaci jiným způsobem. Podobný přístup je použit v knize Linux – programujeme profesionálně [PLP].

Obrázek 2.2. Architektura aplikace při použití čistě relačního řešení



Při tomto způsobu implementace je každá relace reprezentována jedním CORBA objektem, pomocí kterého s ní bude klient pracovat. Již zde není potřeba dvou CORBA rozhraní jako v předchozím případě. Server komunikuje s databází pomocí dotazů v jazyce SQL a výsledky předává klientovi ve formě struktur. Tyto struktury v sobě obsahují všechny sloupce relace včetně primárních a cizích klíčů. U dotazů, které vyžadují spojení více relací, je toto spojení možné provést již v databázovém serveru dle SQL dotazu a serverová aplikace vytvoří z výsledků strukturu s údaji. Klientovi se tím usnadní práce, jelikož si nemusí provádět spojení výsledků z různých relací sám. Ve výsledku dotazu se samozřejmě mohou nacházet cizí klíče, pomocí kterých si může klient provést dodatečná spojení. Obsahuje-li konceptuální model databáze ISA hierarchii, je vhodné, aby server prováděl spojení relací a zasílal klientovi atributy ze všech relací, které tuto hierarchii tvoří.

Příklad 2.10. Příklad specifikace relace v SQL a rozhraní objektu v IDL

```
CREATE TABLE Book (
  bookId SERIAL PRIMARY KEY,
  isbn CHAR(13),
  language CHAR(2),
  title VARCHAR,
  yearOfIssue SMALLINT,
  releaseNumber SMALLINT,
  pagesCount SMALLINT,
  price SMALLINT,
  piecesCount SMALLINT,
  publishingId INTEGER REFERENCES Publishing
);
```

```
interface Book {
  exception NO_SUCH_BOOK {};
  typedef long bookId_t;
  typedef sequence<bookId_t> bookIdList_t;
  struct book_t {
```

```
    bookId_t bookId;
    string isbn;
    string language;
    string title;
    unsigned short yearOfIssue;
    unsigned short releaseNumber;
    unsigned short pagesCount;
    unsigned short price;
    unsigned short piecesCount;
    Publishing::publishingId_t publishingId;
  };
  void create(in book_t book, out bookId_t bookId)
          raises (SQL_ERROR);
  void delete(in bookId_t bookId)
          raises (NO_SUCH_BOOK);
  void set(in book_t book);
  void get(in bookId_t bookId, out book_t book)
          raises (NO_SUCH_BOOK);
  void getBooksWrittenBy(in Person::personId_t authorId,
                         out bookIdList_t bookIdList)
          raises (Person::NO_SUCH_PERSON);
  void getBooksByTitle(in string title,
                       out bookIdList_t bookIdList);
  void getBookByISBN(in string isbn, out bookId_t bookId);
};
```

Pojďme se podívat na metody z příkladu 2.10 – „Příklad specifikace relace v SQL a rozhraní objektu v IDL“. První čtyři metody pracují se strukturami. Jelikož se vyskytují u všech objektů reprezentujících relace, měli bychom je dobře pochopit.

- `create` vytvoří záznam v relaci a vrátí hodnotu primárního klíče. Tato hodnota není ve struktuře předaném vstupním parametrem obsažena.
- `delete` smaže záznam v databázi podle hodnoty primárního klíče.
- `set` modifikuje záznam v databázi. V tomto případě musí být samozřejmě hodnota primárního klíče ve struktuře vyplněna.
- `get` vrátí záznam databáze ve formě struktury podle hodnoty primárního klíče.

Další metody by sloužily pro vyhledávání záznamů podle hodnot ostatních atributů. V uvedeném příkladu se jedná o metody `getBooksWrittenBy`, `getBooksByTitle`, `getBookByISBN`. Tyto metody vrací jako výsledek

dotazu hodnotu primárního klíče nalezeného záznamu nebo kolekci těchto klíčů, pokud může být nalezených záznamů více. Požaduje-li klient kompletní záznamy, může je z těchto klíčů získat metodou `get`. Bylo by možné samozřejmě vracet tyto kompletní záznamy rovnou, ale vzhledem k tomu, že výše uvedené metody `set`, `get`, `delete` pracují s hodnotami primárních klíčů, je možná užitečnější tento nízkoúrovňový přístup.

Poznámka

Metoda `getBooksWrittenBy` má jako vstupní parametr hodnotu primárního klíče relace Autor, která není ve struktuře book_t obsažena. Je tomu tak, protože mezi entitami Autor a Kniha je vztah M:N a při transformaci konceptuálního modelu do relačního tedy vznikla relace Autor_Kniha, jejíž objekt je třeba kontaktovat, chceme-li zjistit autory dané knihy. Můžeme přidat do objektu Book metodu, která by něco takového udělala, museli bychom ale pak implementovat další metody, pomocí kterých bychom například některému autorovi chtěli odebrat autorství dané knihy. Jaké metody dotazů bude objekt podporovat, záleží pouze na nás.

2.2.1. Sestavení spojení

Před provedením první operace se samozřejmě musíme k databázi připojit. Tento proces může probíhat následujícím způsobem:

1. Server po startu zaregistruje ve jmenné službě CORBA referenci objektu *Database*.
2. Klient si tuto referenci ze jmenné služby vyžádá a metodou `loginAs_with_on_` požádá o připojení ke konkrétnímu zdroji dat[6] libgda.
3. Proběhlo-li připojení úspěšně, je mu vrácen objekt `Connection`, který je pouze *továrnou*[7] (Object Factory) objektů spravujících jednotlivé relace.

[6] Na počítači, který se připojuje k databázi přímo pomocí funkcí knihovny libgda, je nutné vytvořit tzv. zdroj dat (datasource). Konfigurace se provádí pomocí aplikace **gnome-database-properties**. Zadává se zde IP adresa nebo doménové jméno databázového serveru, typ databázového systému, název databáze, jméno uživatele, příp. heslo. Samozřejmě musí pro náš databázový systém existovat modul, v současnosti existují moduly pro Firebird, MS SQL a Sybase, MySQL, ODBC, Oracle, PostgreSQL, LDAP, XML.

[7] Pod továrnou chápejme objekt, který má za úkol vytvořit instanci jiného objektu dle zavolané metody. V našem případě vytváříme instanci objektu pouze jednou a při opakovaném vyžádání vracíme referenci na již existující instanci.

Příklad 2.11. Rozhraní sestavení spojení v IDL

```
interface Connection {
  Book book();
  Publishing publishing();
  PaymentMethod paymentMethod();
  Order order();
  Person person();
  Employee employee();
  Department department();
  Author_Book author_Book();
  Book_Order book_Order();
  //Transaction transaction();

  void closeConnection();
};

interface Database {
  exception LOGIN_FAILED {};
  Connection loginAs_with_on_(in string username,
                              in string password,
                              in string datasource)
    raises (LOGIN_FAILED);
};
```

2.2.2. Zhodnocení

Uvedený přístup je sice použitelný, ale ze strany klienta se s ním nepracuje příliš snadno. Při podrobnějším pohledu se nemohu zbavit pocitu, že zde CORBA rozhraní svojí nízkoúrovnovostí nepřináší žádné podstatné zlepšení oproti přímému použití libgda a dalo by se skoro říct, že je zde tedy zbytečné. Začal jsem se proto zabývat myšlenkou přemístění CORBA rozhraní na vyšší úroveň. Nepřišel jsem na lepší řešení, které by zároveň splnilo cíle této práce, než použít CORBA pro vzdálené zpřístupnění uživatelského rozhraní.

2.3. Třetí návrh

Cílem tohoto návrhu bylo pomocí CORBA rozhraní vzdáleně zpřístupnit prvky grafického uživatelského rozhraní, tzv. widgets. Tento přístup nám také umožňuje odstranit závislost klientské části na konkrétní aplikaci, tzn. že klient vůbec nic netuší o aplikaci, kterou vykonává. Veškerá logika programu je v „aplikačním serveru“, ke kterému se klientská část připojuje. Úkolem klient-

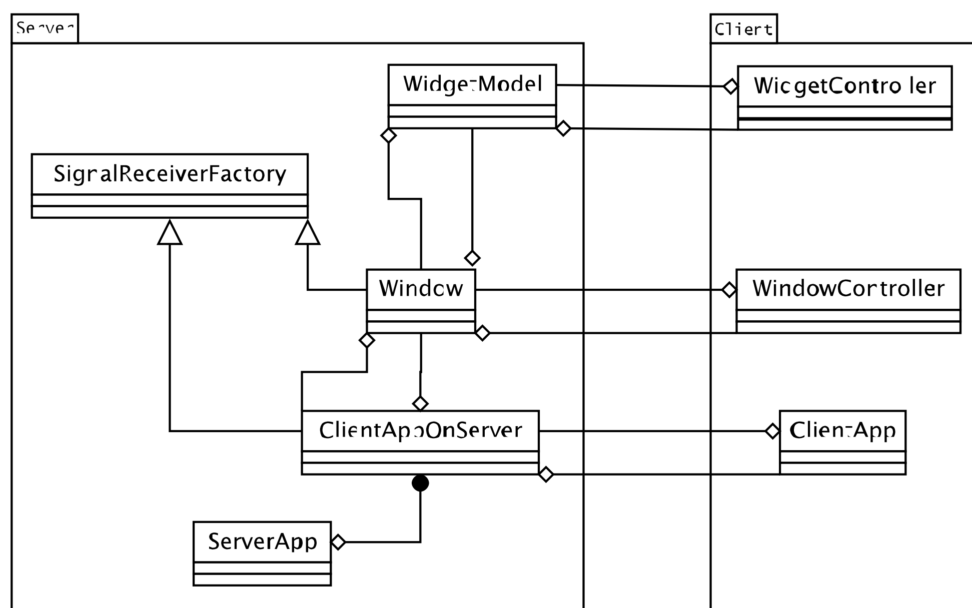
ské části je sestavit dle dodané specifikace uživatelské rozhraní, zaregistrovat v aplikačním serveru jím požadované prvky grafického uživatelského rozhraní, informovat ho delegováním tzv. *signálů* o událostech, které na těchto prvcích nastanou, a také vykonávat akce s prvky rozhraní dle požadavků aplikačního serveru.

Důraz byl kladen také na nezávislost komunikace mezi aplikačním serverem a klientskou částí na konkrétní použité grafické knihovně, klientská část tedy provádí transformaci této komunikace do funkcí dané knihovny. Součástí této práce je ukázková aplikace s implementací klientské části pro grafickou knihovnu GTK+. Pro jiné grafické knihovny, například QT firmy Trolltech, je třeba implementovat novou klientskou část. Jakmile však tyto části budou vytvořeny a odladěny, nebude potřeba do nich zasahovat a bude je možné používat pro libovolné aplikace.

Kromě výše uvedených výhod má tento přístup samozřejmě i své nevýhody, kterými je očekávaná nižší rychlost celé aplikace způsobená právě komunikací na úrovni grafického rozhraní. Na tento handicap byl od počátku brán ohled a pro jeho zmírnění bylo úpravou komunikace sníženo zpoždění. I přes tuto úpravu nelze popisovaný přístup doporučit do vysokozátěžových provozů.

2.3.1. Architektura

Obrázek 2.3. Model vzdáleně zpřístupněného uživatelského rozhraní



Popišme si podrobně architekturu aplikace dle obrázku 2.3 – „Model vzdáleně zpřístupněného uživatelského rozhraní“. Klientská část si po startu od jmenné služby CORBA vyžádá referenci na objekt *ServerApp*, kterého požádá o připojení. Objekt ServerApp spravuje seznam připojených klientských částí, pro každou z nich je vytvořen jeden objekt *ClientAppOnServer*, jehož reference je vrácena jako návratová hodnota při úspěšné žádosti o připojení. Klientská část v žádosti o připojení také předává v parametru referenci objektu ClientApp, aby mohla být serverem později kontaktována.[8] Objekty ClientApp a ClientAppOnServer tedy od tohoto okamžiku znají své reference a mohou si navzájem volat metody. Objekt ClientApp nicméně ještě nedokončil svoji inicializaci. Jakmile tak učiní a bude připraven přijímat požadavky od serveru, informuje o tom objekt ClientAppOnServer zavoláním metody `run`.

Kdykoliv bude server chtít vytvořit nové okno, zavolá metodu `openWindow` objektu ClientApp a předá mu potřebné parametry – jméno okna, název XML souboru s popisem uživatelského rozhraní vygenerovaným programem Glade a dále seznam jmen prvků grafického rozhraní, které je potřeba propojit se serverem. Jméno okna je potřeba dodat, jelikož tento soubor může obsahovat více oken.

Poznámka

Vlastní získání XML souboru není v této práci řešeno, není to důležité. Je možné ho zpřístupnit libovolnými prostředky pro sdílení souborů a otvírat ho „ze sítě“, nebo by mohl být jeho obsah zaslán přes CORBA. V ukázkové aplikaci se nachází v adresáři klientské části.

Tento popis samozřejmě obsahuje v definici jednotlivých prvků rozhraní i jejich jména, a právě některá z těchto jmen jsou předána v parametru metody `openWindow`. Klientská část vytvoří uživatelské rozhraní z tohoto popisu pomocí funkce knihovny libglade. Pro jiné grafické knihovny by bylo potřeba vytvořit její obdobu načítající stejný vstupní soubor nebo poskytovat popis rozhraní v jiném univerzálním XML a dle tohoto popisu explicitně vytvářet každý prvek rozhraní i pro GTK+. Třetí a nejspíš nejhorší možností je poskytovat serverem více popisů uživatelského rozhraní, pro každou knihovnu grafického rozhraní jeden.

Po vytvoření okna dochází k napojování požadovaných prvků uživatelského rozhraní. Pro každý prvek je na klientské části vytvořen tzv. *Controller* a na serveru *SignalReceiver* (příjemce signálů). Úkolem objektů Controller je delegovat signály na objekty SignalReceiver a zároveň sjednocovat rozhraní prvků uživatelského rozhraní různých grafických knihoven. Reference na SignalReceiver je získána z názvu prvku uživatelského rozhraní metodou `get-`

[8] Je potřeba, aby POA manažer byl aktivován i na klientské části, viz 1.3.3.4 – „POA manažer“.

`SignalReceiverFor` zděděnou z třídy *SignalReceiverFactory*. Okno je také objektem typu SignalReceiver a také má svůj Controller, vše zde uvedené platí tedy i pro něj.

Jakmile Controller získá referenci na SignalReceiver, zaregistruje si u prvku uživatelského rozhraní pro každý signál buď svou delegující metodu nebo přímo metodu objektu SignalReceiver. Kdykoliv později nastane událost na prvku grafického rozhraní, dojde k zavolání této metody. Metoda objektu SignalReceiver jen zavolá pro konkrétní prvek specifickou metodu objektu Window – například metoda `clicked` volá `clickedOnCancelButton`. SignalReceiver totiž zná jméno prvku, pro kterého byl vytvořen a i kdyby ho neznal, je mu předán v parametru metody `clicked`.

Aby serverová část aplikace nebyla zbytečně obtěžována událostmi, které ji nezajímají, je vhodné neregistrovat všechny signály, ale pouze ty, které server požaduje. Mohli bychom tedy pro každý napojovaný prvek dodat klientské části i seznam signálů, které je nutné zaregistrovat. V ukázkové aplikaci je seznam signálů napsán přímo v kódu klientské části, ale je samozřejmě možné vše předat dalším parametrem při vytváření okna metodou `openWindow` nebo se na tyto seznamy dotazovat jednotlivými objekty Controller po získání objektu SignalReceiver.

2.3.2. Oddělení funkční části aplikace od graf. rozhraní?

2.3.2.1. Systém závislých objektů

Prvky grafického rozhraní knihovny GTK+ si v sobě drží hodnoty, nejsou tedy bezestavové. Pokud bychom chtěli vytvořit aplikaci, ve které by bylo odděleno uživatelské rozhraní od vlastních dat, které toto rozhraní zobrazuje, je nutné aby prvek rozhraní měl signály, na které bychom se mohli napojit a dozvědět se tak o změně hodnoty, a zároveň naše data musí být schopna informovat o své změně, abychom je následně mohli zapsat do prvku grafického rozhraní. Předpokládejme, že všechny grafické knihovny, které chceme s naší aplikaci používat, poskytují u každého prvku dostatečnou množinu signálů, abychom mohli tuto první podmínku splnit. Podmínku schopnosti dat informovat o své změně je při dodržení jejich znovupoužitelnosti nebo nezávislosti na grafickém rozhraní možné vyřešit implementací principů známých z architektury MVC (Model-View-Controller), resp. návrhového vzoru Observer (pozorovatel). V takovém případě by objekty Controller byly závislými na některém objektu serveru, který by držel vlastní data – *modelu*. Controller by se u svého modelu zaregistroval jako tzv. *závislý objekt*. Model by při změně dat speciální metodou informoval své závislé objekty, že se změnila data. Cont-

roller by na tuto změnu zareagoval vyžádáním aktuálních hodnot a jejich zapsáním do prvku uživatelského rozhraní. Pokud by data při své změně místo pouhého oznámení o změně rovnou předala aktuální hodnoty, zaneslo by to do dat již dříve zmíněnou závislost na grafickém rozhraní.

Jelikož nechceme mít žádnou funkční část aplikace v klientské části, měly by objekty Controller reagovat na informaci o změně dat vždy stejným způsobem, nejspíše zasláním metody `getValue`. Na serveru bychom měli *Adaptory*, které by prováděly transformaci volání této metody do konkrétní metody modelu získávající požadovaná data. Controller by nebyl závislý na modelu, ale na Adaptoru a Adaptor na modelu.

Ukažme si nyní, jak by mohl v konkrétním případě vypadat objekt Controller. Jeho úkolem je předat svému modelu, resp. adaptoru, hodnotu právě zadanou uživatelem. Mějme například prvek *Entry* knihovny GTK+, který slouží pro vkládání jedné hodnoty ve formuláři. U tohoto prvku můžeme použít signál *changed*, který je emitován při každém zadaném znaku. Přímým delegováním tohoto signálu na SignalReciver bychom způsobili příliš mnoho CORBA komunikace, proto by bylo lepší si jen v Controlleru poznamenat, že došlo ke změně hodnoty a jakmile editace hodnoty bude dokončena, vyžádat si z prvku grafického rozhraní aktuální text a odeslat ho celý najednou. O tomto „dokončení editace“ se můžeme nechat informovat signálem *focus_out_event*, který je emitován při ztrátě fokusu, tj. například při kliknutí na jiný prvek. Tento signál se na rozdíl od předchozího vyskytuje u všech prvků grafického rozhraní.

Běžné programovací jazyky závislost neobsahují, museli bychom ji do *nových* objektů implementovat, což ale není příliš obtížné. Přidat metodu do *existujícího* objektu lze v programovacím jazyku Python pouze přidáním nové třídy s touto metodou do třídní proměnné `__bases__` obsahující seznam předků – ani toto nám ale v tomto případě příliš nepomůže, jelikož je potřeba i modifikovat existující metody, aby došlo k zavolání nově přidané metody při každé změně dat. Uvědomme si také, že tento jazyk není čistě objektový, jelikož obsahuje primitivní typy a ty by bylo nutné obalit objekty, aby mohla data o své změně informovat závislé objekty.

Použití principů z architektury MVC naráží na několik problémů. Pokud půjde řetěz závislých objektů i přes CORBA komunikaci, mohlo by dojít ke *zvýšení odezvy aplikace* v důsledku příliš velkého počtu zasílaných zpráv mezi objektem Controller a jeho modelem. Z tohoto pohledu by bylo lepší *ukončit řetěz závislosti na serveru* a mezi klientem a serverem posílat hodnoty pomocí metod `setValue` a `valueChangedTo`. Druhá z uvedených metod je vyvolána pouze při zadání dat uživatelem a samozřejmě ne při přiřazení hodnoty metodou `setValue`.

Uvědomme si ještě jeden problém související s tím, co bude hodnotou v parametrech výše uvedených metod. Pokud bychom objektovou architekturu aplikace rozšířili i do CORBA komunikace, tj. dopravovali bychom mezi klientem a serverem v parametrech těchto metod reference objektů, příp. jejich kolekce, došlo by k dramatickému zvýšení zpoždění aplikace způsobeného vzdálenou komunikací. Zde je jasně vidět, že pro efektivní přenos dat mezi klientem a serverem je potřeba použít primitivní typy CORBA. Při zobrazování obsahu relace s R řádky a S sloupci nám při přenosu primitivních typů stačí *1 zpráva*. Pokud však relaci reprezentujeme kolekcí R objektů, kde každý z nich reprezentuje jeden řádek relace, a pokud pro každý sloupec existuje v těchto objektech 1 metoda, je potřeba poslat *R krát S zpráv*.

2.3.2.2. Skutečně objektový návrh aplikace?

Z předchozích odstavců vyplývá, že pro tvorbu objektových aplikací by bylo možné systému závislých objektů použít, nejlépe pouze lokálně v serverové části, a pro komunikaci s klientskou části tyto objekty překládat do primitivních typů CORBA.

V případě aplikací nad relační databází by to znamenalo překládat primitivní typy z relační databáze do objektů a zase zpět na primitivní typy pro grafické rozhraní. I toto na první pohled nesmyslné řešení by mělo své výhody, protože by se díky objektové povaze dat zjednodušila architektura konkrétní aplikace. Daní tohoto řešení je ovšem nutnost implementace objektů, které jsou běžné v systémech umožňujících tvorbu aplikací s objektovým návrhem, mezi které patří Smalltalk. Implementace těchto objektů vyžaduje jejich dokonalou znalost a jejich následné použití vyžaduje od uživatele rámcového systému velké zkušenosti s tímto typem návrhu. Ani jedním zatím bohužel v dostatečné míře nedisponuji.

Vzhledem k výše uvedenému a také cíli této práce zhodnotit tvorbu aplikací nad relační databází, bylo zvoleno řešení neprovádět dvojnásobný překlad. Ukázková aplikace je „pseudoobjektovou" aplikací pracující s daty relační povahy, což se projevuje v zanesení závislostí na formátu dat skrz celou aplikaci. Tím se komplikuje návrh, jelikož „věci" spolu související nejsou na jednom místě, a dochází tak k prodražení pozdějších úprav aplikace.

2.3.2.3. Opravdu to s relační povahou dat nelze?

I když pracujeme s daty relační povahy a neprovádíme tedy v našem případě dvojnásobný překlad, neznamená to pravděpodobně, že by nešlo oddělit funkční část aplikace od uživatelského rozhraní. Pokud se části aplikace budou chovat jako takové „kontejnery" těchto relačních dat a signály uživatelského rozhraní se budou překládat na funkce aplikace, které data mezi těmito

kontejnery přenáší a zpracovávají, je možné ovládat aplikaci i bez uživatelského rozhraní. Problém je pouze s překreslením uživatelského rozhraní při změně dat.

Pokud by kontejner i uživatelské rozhraní byly objekty, bylo by možné i použít závislostí, přestože vlastní data, která celou aplikací proudí, objekty nejsou.

Nevyžadujeme-li oddělení uživatelského rozhraní od funkční části aplikace, závislosti nepotřebujeme, pokud se spolehneme na sekvenční provádění programu a pokud se data aplikace budou modifikovat pouze z jednoho místa, tj. z podnětu uživatelského rozhraní – to pak bude odpovědné za vyvolání událostí způsobujících překreslení prvků zobrazujících změněná data. Je ale potřeba, aby data byla do prvku uživatelského rozhraní naplněna „z venku" – prvek si nedokáže data získat sám, jak je obvyklé ve výše zmíněných objektových systémech, ale naplní je do něho objekt, který se o změně kontejneru dozvěděl.

Použití závislostí by také možná nakonec vedlo k plnění „z venku", je totiž otázkou, zda by granularita objektů byla dostatečně malá, aby si prvky uživatelského rozhraní mohly data z kontejneru získávat samy, a zda by výsledkem nebyl řetěz jednoúčelových objektů získávajících tato data z kontejnerů a transformujících je do podoby vyžadované prvkem rozhraní. Takové řešení by postrádalo smysl, jelikož by tyto objekty nutně musely být silně závislé na *formátu* neobjektových dat a tedy špatně znovupoužitelné.

Přístup s kontejnery, bez objektové závislosti a s plněním prvků „z venku" byl zvolen v ukázkové aplikaci.

2.3.3. Popis vzdálené komunikace a formát dat

Ať už bude architektura serverové části jakákoliv, na rozhraní vzdálené komunikace ani na klientskou část pracující s primitivními typy to nebude mít vliv. Víme, že pro každý z požadovaných prvků uživatelského rozhraní je klientskou částí vytvořen objekt Controller. U prvků, které zobrazují uživateli data, mají tyto objekty metody `setValue` a `getValue`, pomocí kterých nastavíme nebo naopak získáme hodnotu z prvku.

Pro snížení zpoždění aplikace a také hlavně pro podporu různých architektur serverové části byla druhá z uvedených metod později nahrazena metodou `valueChangedTo` u objektu SignalReceiver. Hodnota zadaná uživatelem je tedy odeslána na server místo toho, aby se na ni musel ptát v době, kdy ji bude

potřebovat. Snížené zpoždění jde na úkor zvýšeného počtu zaslaných zpráv. Tato metoda je však nezbytná, pokud je serverová část navržena skutečně objektově a nebo si jen udržuje hodnotu ve „vyrovnávací paměti“. Připomeňme si, že použitá grafická knihovna musí poskytovat dostatečnou množinu signálů, se kterou je možné tuto metodu implementovat.

Všechny tři zmíněné metody mají jako návratovou hodnotu, resp. parametr, již dříve popsaný datový typ Any, ve kterém je uložena hodnota jednoho z následujících typů:

- *jednoduchý primitivní typ* – použijeme u prvku Entry.
- *CORBA sekvence jednoduchých typů* – například u prvku ComboBox.
- *CORBA sekvence CORBA záznamů* – u prvků zobrazujících víceřádková vícesloupcová data, tj. v našem případě relace relační databáze.

V posledním případě CORBA záznam reprezentuje záznam (řádek) relace. CORBA sekvence reprezentuje několik záznamů relace. Klíči CORBA záznamů jsou názvy „sloupců“ relace. Pro implementaci aplikace byl zvolen programovací jazyk Python, ve kterém je CORBA záznam mapován na typ dict, který umožňuje přiřadit hodnotu libolnému až za běhu aplikace definovanému klíči. Controller v klientské části tuto sekvenci záznamů transformuje do formátu požadovaném prvkem uživatelského rozhraní.

Některé prvky uživatelského rozhraní umožňují výběr hodnoty z několika nabízených. Příkladem může být prvek, který je známý pod názvem ComboBox. Podobně je tomu u prvku zobrazující relaci, pokud bychom rádi uživateli umožnili vybrat záznam relace, se kterým později provede nějakou akci. Objekt Controller má u těchto prvků metodu `getSelection`, resp. jejich SignalReceiver metodu `selectionChangedTo`.[9] Výběr ze seznamu nabízených hodnot představuje bezpečnostní problém, jelikož upravená klientská část by mohla serveru podvrhnout nesprávná data. Proto je nutné v serverové části kontrolovat, zda vybraná hodnota byla předtím nabízena. Tuto kontrolu provádí *filtr*, jehož metoda `filterValue` je zavolána objektem SignalReceiver po získání hodnoty z objektu Controller.

Je zřejmé, že CORBA rozhraní se bude u některých prvků lišit, u některých bude shodné. Doposud jsme probrali prvky zobrazující uživateli nějaká data, dále prvky umožňující navíc výběr z těchto dat a zbývají nám tlačítka. U tlačítek je nejdůležitější vlastností informovat serverovou část o kliknutí. K této

[9] Bylo by možné použít původní metody `valueChangedTo` a `getValue`, jelikož v obou případech se jedná o hodnotu zadanou uživatelem, pouze v jednom je množina hodnot omezena výběrem. Nejspíš bychom tím ale způsobili zmatek v tom, že u některých prvků by `getValue` vracela něco jiného, než jsme do ní přiřadili pomocí `setValue`.

činnosti není potřeba na serverové části registrovat referenci na objekt Controller, pokud ho serverová část nebude nikdy chtít kontaktovat například pro zneaktivnění.

2.3.4. IDL popis

V následujícím IDL popisu si všimněte, že objekty popsané rozhraními `SignalReceiver` očekávají v signálových metodách název prvku rozhraní, přestože by jej měly znát, jelikož pro něj byly vytvořeny – zavolání metody `getSignalReceiverFor`. Tato duplicita zde umožňuje postavit model aplikace i tak, že serverová část bude mít pro všechny prvky uživatelského rozhraní stejného příjemce signálů.

```
module RemoteGUI
{
  exception UNKNOWN_WIDGET_NAME {};

  interface ClientAppOnServer;
  interface ClientApp;
  interface SignalReceiver;
  interface Column;

  typedef sequence<Column> columns;
  typedef sequence<string> stringSeq;

  interface ServerApp
  {
    ClientAppOnServer registerClientApp(in ClientApp app);
  };

  interface ClientApp
  {
    void quit();
    void openWindow(in string windowName, in string gladeFile,
                    in stringSeq widgetsToConnect);
  };

  interface SignalReceiverFactory
  {
    SignalReceiver getSignalReceiverFor(in string widgetName)
         raises (UNKNOWN_WIDGET_NAME);
  };

  interface ClientAppOnServer: SignalReceiverFactory
  {
    void run();
  };

  interface WidgetController
```

```
{
};
```

Poznámka

Rozhraní `WidgetController` je abstraktní. Musí zde být, přestože je prázdné, protože níže uvedené metodě `registerWidgetController` předáváme objekty s různými rozhraními `Controller`. Tato rozhraní proto dědí z tohoto abstraktního rozhraní.

```
interface ValueController: WidgetController
{
  any getValue();
  void setValue(in any value);
};

interface WindowController: WidgetController
{
  void close();
};

interface SelectionInListController: ValueController
{
  any getSelection();
};
```

Metoda `getSelection` není potřeba, pokud se vybraná hodnota posílá zprávou `selectionChangedTo` při každé změně výběru.

```
interface SignalReceiver
{
  void registerWidgetController(in WidgetController ior);
  //sequence<string> getSignalsToEmit();
};

interface WindowSignalReceiver: SignalReceiver,
                                SignalReceiverFactory
{
  void attemptedToClose(in string widgetName);
  void postOpen(in string widgetName);
};
```

Metoda `attemptedToClose` je zavolána, když uživatel klikne na zavírací tlačítko okna. K žádnému zavření okna nedojde. Pokud si serverová část přeje okno skutečně zavřít, musí zavolat metodu `close` rozhraní `WindowController`.

Metoda `postOpen` je zavolána po otevření okna a napojení prvků uživatelského rozhraní. Slouží zde podobně jako metoda `run` pro synchronizaci klientské a serverové části. Jelikož používáme pouze synchronní[10] zasílání zpráv, nejsou tyto metody nutné, nicméně zjednodušují práci.

```
  interface ButtonSignalReceiver: SignalReceiver
  {
    void clicked(in string widgetName);
  };

  interface ValueSignalReceiver: SignalReceiver
  {
    void valueChangedTo(in string widgetName, in any value);
  };

  interface SelectionInListSignalReceiver: SignalReceiver
  {
    void selectionChangedTo(in string widgetName,
                            in any selection);
  };

  interface MultiRowSignalReceiver: SelectionInListSignalReceiver
  {
    columns getColumns();
  };

  interface Column
  {
    string getLabel();
    string getKey();
    CORBA::TypeCode getValueType();
    boolean getVisibility();
  };
};
```

Všimněte si rozhraní `MultiRowSignalReceiver` a `Column`. Použijeme je u prvků uživatelského rozhraní zobrazujících uživateli víceřádková vícesloupcová data. Řekli jsme si, že v těchto případech probíhá přenos dat mezi serverovou a klientskou částí v datovém typu any obsahujícím CORBA sekvenci CORBA záznamů.

Metoda `getColumns` vrací specifikaci sloupců v podobě sekvence objektů

[10] Asynchronní zasílání zpráv je možné pomocí CORBA Messaging Service, které ale není příliš rozšířeno. Lze také použít OMG Event Service a OMG Notification Service, které zprostředkovávají komunikaci mezi producenty a konzumenty přes kanál, aby o sobě nemuseli vědět a zlepšila se škálovatelnost. Těžkopádně lze také asynchronně zasílat zprávy pomocí Dynamic Invocation Interface. Poslední metodou, která je ale velmi nespolehlivá, a proto se nedoporučuje, jsou jednocestné metody definované klíčovým slovem *oneway* v IDL.

popsaných rozhraním `Column` s následujícími metodami:

- `getKey` vrací *klíč*, kterým se má klientská část aplikace dotázat CORBA záznamu pro získání hodnoty tohoto sloupce.
- `getValueType` je zde pouze z důvodu, že prvek použité knihovny uživatelského rozhraní vyžaduje specifikovat jakého typu bude hodnota, kterou do něj budeme vkládat.
- `getLabel` vrací pouze název sloupce, který má být zobrazen uživateli v záhlaví.
- Pomocí metod `getVisibility` poskytuje serverová část přednastavený seznam viditelných sloupců. Vychází se zde z předpokladu, že prvek uživatelské rozhraní umožňuje uživateli zobrazit, resp. skrýt sloupce dle jeho přání za běhu aplikace. Sloupce, jejichž metoda `getVisibility` vrací hodnotu `false`, nebudou uživateli viditelné, avšak budou se přenášet mezi serverovou a klientskou částí. Tímto „neviditelným“ sloupcem může být primární klíč relace nebo i jen sloupec s dodatečnou informací, která by při zobrazení snížila přehlednost.

  Pokud bychom se nechtěli na tuto funkci spoléhat, je samozřejmě možné uživatelské rozhraní postavit na principu tzv. *master-detail*, kde při výběru záznamu relace v jednom prku uživatelského rozhraní dojde k zobrazení podrobností o tomto výběru v jiném prvku.

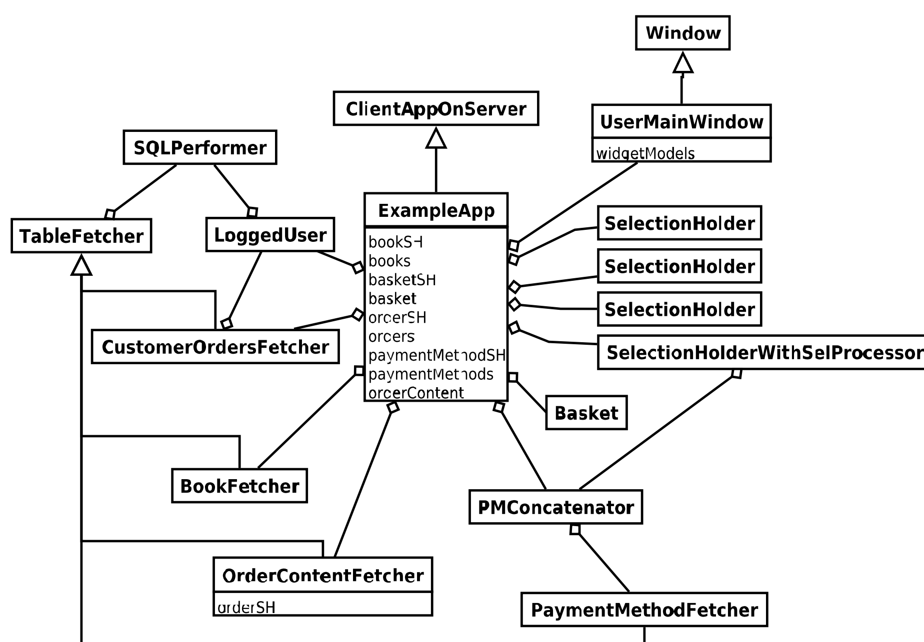
  ### Poznámka

  V ukázkové aplikaci nelze uživatelem zvolit viditelnost sloupců, jelikož jsem nepřišel na způsob, jak této v mnoha GTK+ aplikacích vyskytující se funkce docílit.

Kapitola 3. Ukázková aplikace

Cílem ukázkové aplikace je ověřit praktickou použitelnost řešení popsaného v části 2.3 – „Třetí návrh“. Aplikace je fragmentem informačního systému internetového knihkupectví. Na obrázku 3.1 – „Schéma ukázkové aplikace“ je uvedeno schéma části aplikace, ve které si zákazník vybírá knihy, přidává je do košíku, volí způsob placení, potvrzuje objednávku a také nechává vypsat své starší objednávky.

Obrázek 3.1. Schéma ukázkové aplikace



Popišme si nejdůležitější objekty

- UserMainWindow – reprezentuje, jak je z názvu patrné, hlavní okno aplikace. Stejně jako všechny objekty Window má i tento za úkol vyvolat akci z podnětu signálu na prvku uživatelského rozhraní a následně se postarat o překreslení dotčených prvků. Metoda `clickedOnBuyButton` například vyvolá **app.addSelectedBookToBasket()** a „překreslí“ obsah košíku. Při překreslování se přistupuje nepřímo k objektům `SignalReceiver` udržovaných v instanční proměnné `widgetModels`.

- SelectionHolder – udržuje aktuálně vybranou hodnotu ze seznamu. Smyslem není jen snížit zpoždění aplikace tím, že není nutné kontaktovat klientskou část serveru pro získání této hodnoty, ale také umožnit oddělení

aplikace od uživatelského rozhraní. Pokud by tato „cache“ neexistovala, musela by výše uvedená metoda `addSelectedBookToBasket()` v objektu `ExampleApplication` přistupovat pro získání vybrané hodnotu přistupovat k objektu `SignalReceiver` přes `UserMainWindow` (objekt uživatelského rozhraní), zatímco nyní se obrací na SelectionHolder. Zmínili jsme, že ukázková aplikace nemá skutečně objektový návrh, SelectionHolder je tedy kontejner, který drží hodnotu primitivního typu, přesněji řečeno jejich kolekci, protože rámcový systém počítá s možností výběru několika řádků najednou.

- ValueHolder má podobný účel jako SelectionHolder – jeho smyslem je udržovat hodnotu, kterou by bylo nutné získávat z klientské části metodou `getValue`. Rozdíl mezi nimi je kromě rozhraní objektů (API) především v tom, že SelectionHolder by si měl skládat seznam, z jehož hodnot si jednu drží jako vybranou, a měl by být na tomto seznamu závislý, aby se dozvěděl například o odstranění hodnoty ze seznamu – měl by to být spíše `SelectionInList`. Jelikož ukázková aplikace závislosti neobsahuje, je za zneplatnění výběru při změně seznamu odpovědný ten, kdo změnu vyvolal. Těmito jsou objekty `TableFetcher` a `Basket`.

- TableFetcher – má za úkol získat data z relační databáze SQL dotazy, dodatečně tato data zpracovat a vrátit je ve formě seznamu. K relační databázi přistupuje prostřednictvím objektu `SQLPerformer`, kterému předává SQL dotazy. Zde se bohužel ukazuje, že jazyk SQL není na některé typy úloh vhodný. Chceme-li například získat seznam knih a u každé z nich *zřetězit její autory*, neobejdeme se bez „dodatečného zpracování“ dat, které lze implementovat buď v procedurálním rozšíření databázového stroje nebo mnohem pomaleji až v aplikaci. V prvním případě bychom mohli elegantně využít indexů jazyka PL/SQL. V této práci jsme se procedurálního rozšíření vzdali, a tak musíme data dodatečně zpracovávat až v aplikaci. V naší konkrétní úloze si můžeme vypsat knihy bez autorů a ke každé z nich dalším dotazem získat autory, zřetězit je a zahrnout do výsledku. To je stejně elegantní řešení jako při použití indexů, ale nejspíš pomalejší, protože mezi databázovým strojem a aplikací přenášíme *N dotazů místo jednoho*. Z tohoto důvodu se nabízí velmi těžkopádné, ale možná rychlejší řešení vypsat knihy, které mají pouze jednoho autora, i s tímto autorem a zřetězování autorů provádět jen u „zbylých“ knih. Výhodou tohoto řešení je nižší počet dotazů 2+N-M na úkor dvou velmi komplikovaných. Tento přístup byl použit v objektu `BookFetcher`.

  Objekty TableFetcher se samozřejmě starají i o *vytváření nových záznamů* například při potvrzení objednávky. V takovém případě musí odněkud získat nově vygenerovanou jedinečnou hodnotu primárního klíče. Je-li primární klíč typu SERIAL, dojde při definici relace k vytvoření tzv. *sekven-*

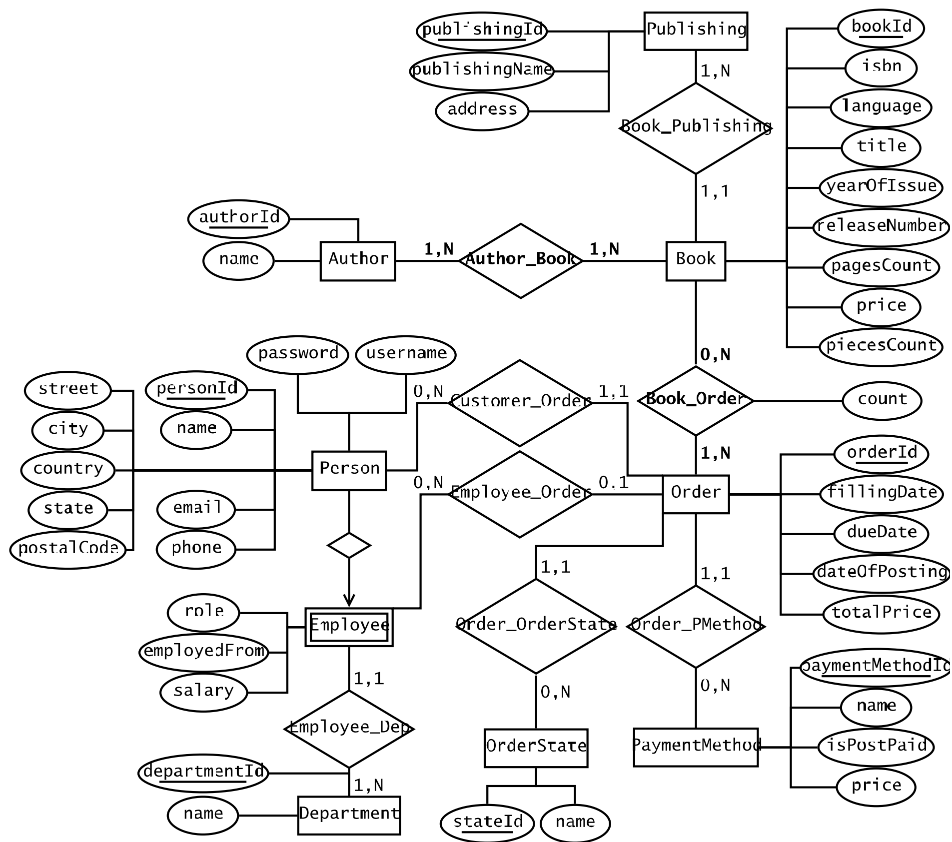
ce, která udržuje unikátní hodnotu. Databázové stroje většinou poskytují PL/SQL funkci, která vrací aktuální hodnotu sekvence a automaticky provádí inkrementaci. Název této funkce není standardizován a knihovna libgda bohužel neposkytuje funkci, která by přístup sjednocovala, a tak se naše aplikace stává závislou na konkrétním databázovém stroji – v případě PostgreSQL musíme provést SQL dotaz

```
SELECT nextval('_order_orderid_seq');
```

- PMConcatenator – má za úkol transformovat data do formy prezentované uživatelským rozhraním. V ukázkové aplikaci se v prvku *ComboBox* zobrazuje způsob platby a jeho cena, v tomto objektu tedy dochází ke zřetězování názvu a ceny, například *dobírka - 70 Kč*. Na tomto případu je přesně vidět nevýhoda absence čistě objektového návrhu. To, co by šlo elegantně vyřešit nějakou metodou `printString` přímo na objektových datech, vyžaduje v případě dat relační povahy odtržení transformací od vlastních dat, vytvoření umělých částí systému provádějících tyto transformace a co je nejhorší – zanesení závislosti na formátu dat a jejich prezentace i do jiných částí systému. Objekt `SelectionHolderWithSelectionProcessor` udržuje tuto zřetězenou hodnotu a při požadavku na vrácení aktuální hodnoty, musí kontaktovat objekt `PMConcatenator` pro získání původních nezřetězených hodnot.

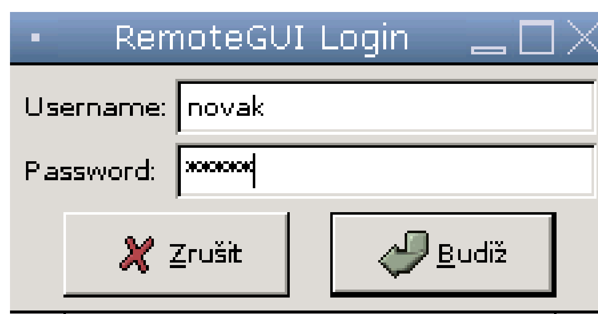
Více informací o architektuře rámcového systému a ukázkové aplikace lze rozpoznat z přiložených zdrojových kódů.

Obrázek 3.2. E-R model

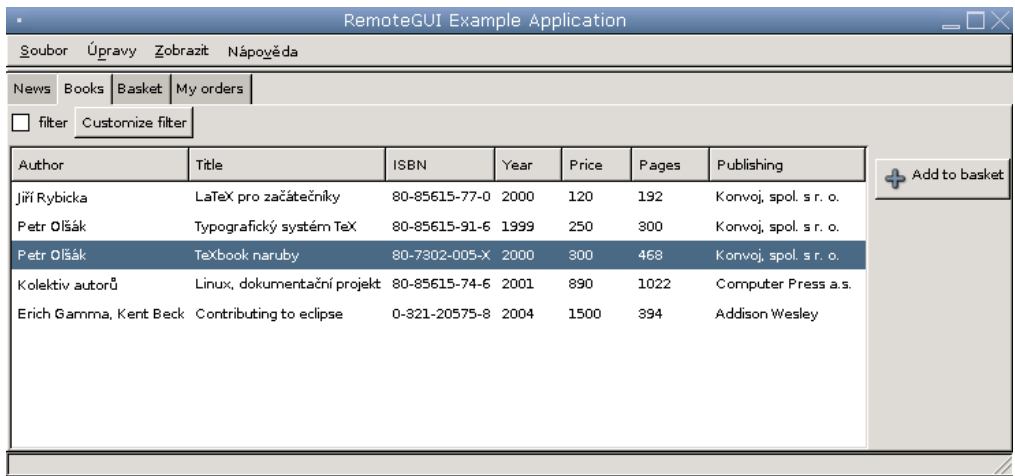


Následující obrázky vzhledu aplikace nemá příliš smysl popisovat.

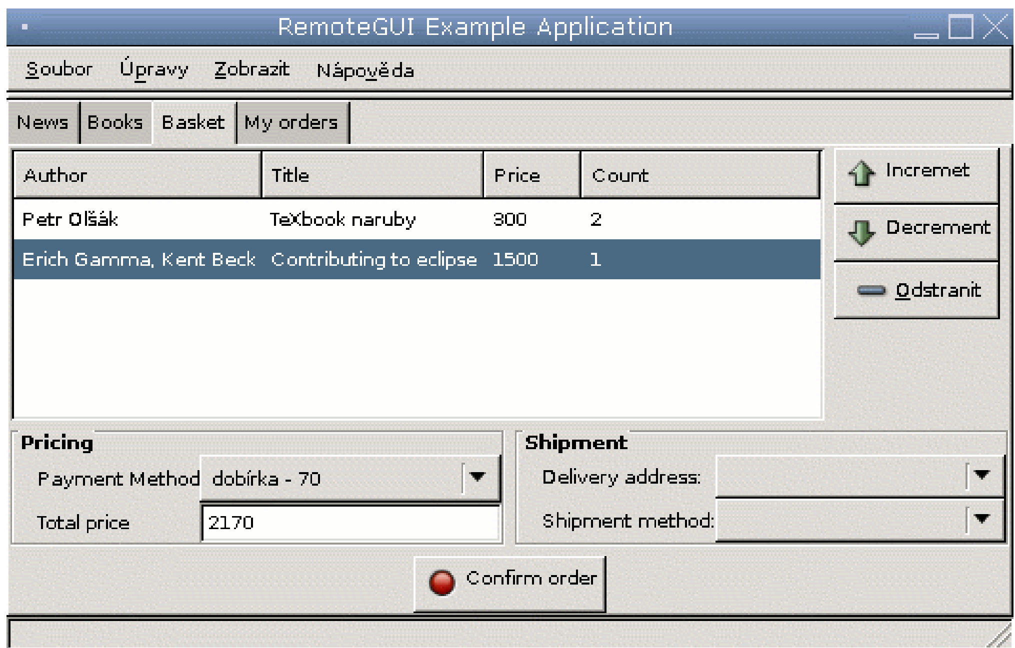
Obrázek 3.3. Přihlášení



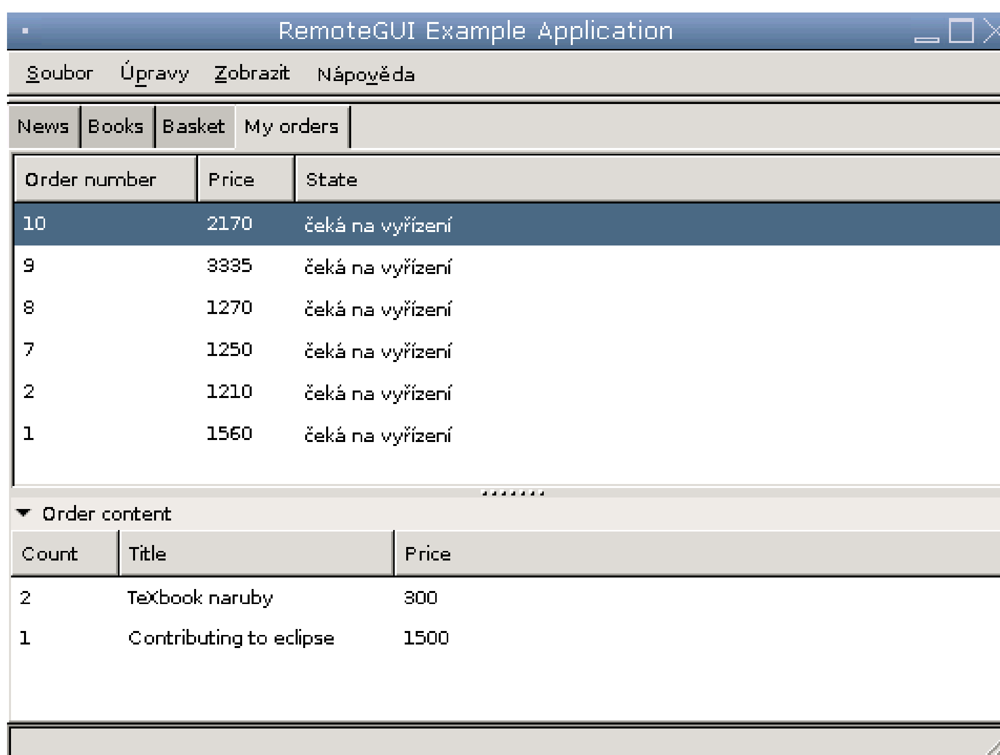
Obrázek 3.4. Seznam knih



Obrázek 3.5. Obsah košíku



Obrázek 3.6. Přehled objednávek



3.1. Konfigurace prostředí

Instalací použitých technologií se nebudu zabývat, protože je závislý na použité distribuci. Pro zprovoznění ukázkové aplikace je potřeba nainstalovat následující programy.

- PostgreSQL
- omniORB
- omniORBpy
- Python
- gnome-python-extras

O prvotní konfiguraci databázového stroje, při které se vytváří úložiště databází, se většinou postará systémový skript (služba) při svém prvním spuštění. Pro případ, že k tomu u vaší distribuce nedojde, uvádím zde následující postup.

```
PGROOT=/var/lib/postgres
export PGROOT
```

```
groupadd -g 88 postgres
useradd -u 88 -g postgres -d $PGROOT -s /bin/bash postgres
mkdir -p $PGROOT/data
chown postgres.postgres $PGROOT/data
su postgres -c "/usr/bin/initdb -D $PGROOT/data"
su postgres -c "/usr/bin/pg_ctl -D $PGROOT/data \
  -l /var/log/postgresql.log -W start"
```

Způsob spuštení služby databázového stroje je také závislý na použité distribuci, a proto předpokládám, že databázový stroj již běží. Nejprve vytvoříme uživatele a vlastní databázi.

```
su postgres -c psql
postgres=# CREATE ROLE some_user NOSUPERUSER NOCREATEDB
  NOCREATEROLE INHERIT LOGIN;
postgres=# CREATE DATABASE "RemoteGUI_example";
```

Nyní se připojíme k databázi, vytvoříme v ní relace a naplníme je ukázkovými daty.

```
psql -d RemoteGUI_example some_user
RemoteGUI_example=> \i sql/createdb.sql
RemoteGUI_example=> \i sql/createdata.sql
```

Nyní na stroji se serverovou částí klientské aplikace spustíme program **gnome-database-properties**, pomocí kterého nastavíme zdroj dat dle obrázku 3.7 – „Konfigurace zdroje dat“.

Obrázek 3.7. Konfigurace zdroje dat

Vlastnosti zdroje dat

Základní | Tabulky

Název zdroje dat: RemoteGUIexample

Poskytovatel: PostgreSQL

Připojovací řetězec: DATABASE=RemoteGUI_example;HOST=localhost

Popis: Ukázková aplikace

Jméno uživatele:

Heslo:

Vrátit | Zavřít

Spustíme jmennou službu CORBA příkazem **omniNames**. Na všech ORB, které budou přistupovat ke jmenné službě, nastavíme cestu k její referenci, tj. zapíšeme do souboru `/etc/omniORB.cfg` následující řádek.

```
InitRef = NameService=corbaloc::localhost/NameService
```

Nyní již zbývá jen spustit nejprve serverovou část klientské aplikace a potom klientské části.

Po spuštění klientské části je uživatel vyzván k zadání přihlašovacích údajů. SQL skripty, které jsou součástí této práce, vytvořily uživatele *novak* s heslem *heslo*. Ukázková aplikace neumožňuje registrovat nového uživatele.

Kapitola 4. Zhodnocení

Uvedli jsme si tři přístupy tvorby aplikací nad relační databází. Třetí přístup spočívající ve vzdáleném zpřístupnění prvků uživatelského rozhraní se jeví jako v praxi poměrně dobře použitelný a splňuje v této práci vytyčené cíle. Místo pracné implementace vrstvy získávající data z relační databáze bychom však v současné době mohli použít některé z existujících knihoven objektově relačních mapování. Přestože mají své nedostatky, je s nimi tvorba aplikací jednodušší než zde použité řešení. Zadání této práce je více než 3 roky staré, v té době nebyly používany v takové míře jako je tomu dnes.

Místo vzdáleného zpřístupnění prvků grafického rozhraní lze použít webového rozhraní. S příchodem specifikace web 2.0 a knihovny AJAX snad dochází k odstranění jeho uživatelské nepřívětivosti, a proto by takové řešení možná bylo více perspektivní. Zde použitý přístup je však také použitelný a jeho obdoby jsou pravděpodobně využívány v komerčních systémech – například v software firmy SAP.

Architektura CORBA sice ulehčuje vývoj distribuovaných aplikací, nicméně způsob, jakým se s ní v různých programovacích jazycích pracuje, není příliš přívětivý. Snadnost používání bohužel asi nebyla nejdůležitějším cílem při standardizaci.

Knihovna *libgda* je také přínosem pro tvorbu aplikací nad relační databází, protože nás zbavuje závislosti na databázovém stroji konkrétního výrobce. Bohužel neposkytuje jednotné rozhraní pro generování unikátních hodnot pro primární klíče, to je ale jen drobná vada, která nijak závažně nekomplikuje přenos aplikace na databázový stroj jiného výrobce.

Tvorba uživatelského rozhraní pomocí knihovny GTK+ a vizuálního návrháře Glade je velmi snadná, horší je to se způsobem práce s jejími prvky uživatelského rozhraní. Pokud programátor nemyslí objektově, nemusí mu tento způsob práce vadit. Některé úkony jako například zobrazení kontextového menu odvozeného od vybraném řádku nejsou příliš jednoduché, ale to se časem nejspíš změní. Rád bych ale zmínil, že existují lepší principy tvorby uživatelského rozhraní, které vyžadují čistě objektový návrh aplikace.

Použití programovacího jazyka Python usnadnilo implementaci třetího návrhu. Jelikož je tento jazyk dynamicky typovaný, bylo možné snadno postavit klientskou část nezávislou na konkrétních primitivních typech. V CORBA rozhraní předáváme hodnoty libovolného typu v typu any. Bez dynamické typové kontroly bychom museli program rozvětvit podle tohoto typu. Tím, že tvoříme parametry metod dynamicky, však vytváříme obrovské problémy aplikacím implementovaným ve staticky typovaných jazycích, které by pří-

padně s naší aplikací rády komunikovaly. Jelikož tyto aplikace nemají k dispozici IDL popis rozhraní, musejí používat dynamické nástroje CORBA – DII, DynAny a Typecode. Poslední dva slouží k tvorbě Any bez dostupného IDL popisu a ke zjištění typu hodnoty obsažené v Any. Tyto dva nástroje nebyly v této práci podrobně vysvětleny, protože jejich použití je doslova noční můrou.

Dynamická typová kontrola jazyka Python v Any spoléhá na položku Repository ID, která je součástí kódu typů. Bohužel CORBA povoluje tuto položku nevyplnit, a v těchto případech se dynamická typová kontrola nemá o co opřít. Proto se doporučuje i v případě jazyka Python přetypovat objekt získaný z any pomocí metody `_narrow` na konkrétní třídu.

V této práci nebylo řešeno zabezpečení spojení mezi klientskou a serverovou částí. CORBA ale umožňuje spojení šifrovat přes SSL. Podrobně jsem se tímto nezabýval, nebylo to mým cílem, ale nepředpokládám, že by měl být příliš velký problém toto zprovoznit. Dalším problémem, který nebyl řešen, je lokalizace.

Celkově mohu tedy v této práci zvolené technologie zhodnotit jako použitelné nicméně nepříliš vhodné.

Bibliografie

The omniORB version 4.0 User's Guide. http://omniorb.sourceforge.net.

Michi Henning a Steve Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley. 1999. 0-201-37927-9.

CORBA: Core Specification. Object Management Group. 2004.

[PLP] Neil Mathew a Richard Stones. *Linux – programujeme profesionálně*. Computer Press. 2001. 80-7226-532-6.

GemStone GemORB User's Guide. 2000.