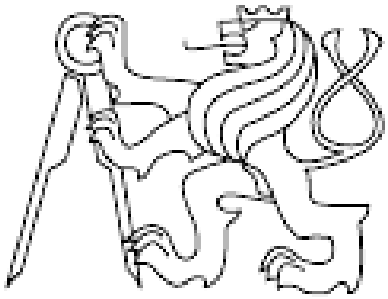
České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Vytvoření univerzálního uživatelského rozhraní pro vyvíjený EDA (electronic design automation) systém.

Ondřej Švejdar

Vedoucí práce: Ing. Petr Fišer

Studijní program: Informatika a výpočetní technika

leden 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 Bakově nad Jizerou dne 25.1.2006 .....

Abstract

A design and implementatnion of a user interface for the EDA system is proposed in this Diploma Thesis. This system will consist of separate algorithms. First type of algorithm is BOOM – algorithm for a Boolean minimalization. This algorithm was a model for the entire user interface design and its implementation was a test of this model.

The orginal user interface was a command line. BOOM algorithm was an executable file, whose parameters were passed throught the command line. This algorithm was integrated into single modular system. During this integration, orginal source code (written in C/C++) of this algorithm was respected in order to ensure the alorithm's reusability, so the source code can be used in other projects without binding to this system and user interface.

This system implements the management of algorithms by creating instances of algortithms, setting parameters of instances, executing them and providing a communication between the instances of algoritms. Executing more than one algorithm at once is not supported, because solving the problems involved, such as the access to common variables, treatment of inputs and outputs (for example two instances writing to the same output file), treatment of input parameters of already running algorithm, etc., is theme for another Diploma Thesis. The system is written to be not restricted to a single platform, ie., it can be executed from linux/unix system and the Windows system.

The user interface is an upper layer of this system. It consists of two parts. The first part performs all basic system operations accessible via shell. The second part is a graphical API, which allows creating of the graphic interface to the programmers. A form designer has been integrated, to simplify designing of the graphic interface.

The result of this work is a design and implementation of functional C++ modular system, adjusment of existing shell interface (Tcl) so it can work as an upper layer of this modular system, design and implementation of C++ API for graphics development, a design and a functionous implementation of the form designer (which is using C++ API for graphic design) to simplify the form design. Matter of course is a proper documentation.

Anotace

Tato práce je návrhem a implementací uživatelského rozhraní pro vyvíjený EDA systém. Tento systém se bude skládat z různých algoritmů. Prvním typem algoritmu je BOOM – algoritmus pro booleovskou minimalizaci. Tento algoritmus byl vzorem pro návrh uživatelského rozhraní a jeho implementací se tento návrh testoval.

Původním uživatelským rozhraním byla příkazová řádka. Algoritmus BOOM byl ve formě spustitelného souboru, kterému se přes příkazovou řádku předávaly parametry. Tento algoritmus byl začleněn do modulárního systému. Přitom bylo dbáno na to, aby se jeho zdrojový kód psaný v C/C++ příliš nezměnil, aby se dal kód použít i v jiných projektech bez vazby na tento systém a uživatelské rozhraní.

Tento systém realizuje vlastní správu algoritmů: vytváření instancí algoritmů, nastavování parametrů instance, spouštění instance algoritmu, komunikace mezi instancemi algoritmů. Není podporováno spouštění více algoritmů najednou, neboť řešení problému s tím spojených, tj. přístup ke společným proměnným, ošetření vstupů a výstupů (např. dvě instance algoritmu snažící se zapisovat do stejného výstupního souboru), ošetření zadávání vstupních parametrů již běžícího algoritmu, je téma na samostatnou diplomovou práci. Je psaný tak, aby nebyl omezený pouze na jednu platformu resp. aby se dal spouštět jak z unixových/linuxových systémů tak z systému Windows. Na obou těchto platformách byl i testován.

Uživatelské rozhraní je nadstavbou nad tímto systémem. Skládá se ze dvou částí. První část zpřístupňuje všechny základní operace systému pomocí shellu. Druhou částí je grafické API, pomocí kterého může programátor původních algoritmů vytvářet grafické rozhraní. Pro zjednodušení tvorby grafického rozhraní byl začleněn návrhář formulářů.

Výsledkem této práce je návrh a funkční C++ implementace modulárního systému, úprava existujícího shellového rozhraní (Tcl) tak, aby fungovalo jako nadstavba nad tímto modulárním systémem, návrh a implementace C++ API pro tvorbu grafiky a návrh a funkční implementace návrháře (který používá navržené C++ API pro tvorbu grafiky) pro usnadnění tvorby formulářů. Samozřejmostí je také řádná dokumentace.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

Cílem práce je vytvoření univerzálního uživatelské rozhraní pro vyvíjený EDA (electronic design automation) systém. Tento systém je soubor vzájemně nezávislých modulů. Každý tento modul je reprezentací algoritmu, který řeší konkrétní problém. Typickým příkladem tohoto algoritmu je BOOM, viz. [1], algoritmus pro booleovskou minimalizaci. Nad těmito moduly (algoritmy) lze provádět některé elementární operace: nastavování vstupních parametrů, spouštění, mohou spolu komunikovat apod. Uživatelské rozhraní má být nejvyšší vrstva celého systému, bude zaštiťovat všechny moduly a zpřístupňovat uživateli tyto elementární operace. Skládá se z několik částí: jednak je to rozhraní pro řízení celého systému, jednak rozhraní pro komunikaci s jednotlivými moduly, např. nastavování vstupních parametrů algoritmu. Základní úrovní rozhraní má být shell, pomocí kterého bude uživatel celý systém řídit. Dále rozhraní musí poskytovat prostředky, pomocí kterých může programátor modulů (algoritmů) vytvářet grafické rozhraní a umožnit tak uživateli např. grafické zadávání vstupních parametrů algoritmu.

2. Popis problému

Celý problém se dělí na několik základních částí:

- Návrh a implementace modulárního systému, který bude spravovat vlastní algoritmy. Tato část je klíčová, všechny ostatní návrhy z ní budou vycházet.
- Návrh a implementace shellového rozhraní, přes které bude moci uživatel kompletně spravovat tento modulární systém.
- Návrh a implementace grafického API, přes které bude moci programátor modulů vytvářet grafiku pro tvorbu formulářů.
- Pro usnadnění tvorby grafiky nad tímto API implementovat jednoduchého návrháře formulářů, který bude k dispozici pro programátora modulů a bude vhodně začleněn.

Požadavkem je, aby implementace žádné části nebyla omezena pouze na jednu platformu resp. aby se dala spouštět jak z unixových/linuxových systémů, tak z Windows. Všechny části musí být navržené tak, aby se daly snadno rozšířit, zejména se jedná o grafické API, kde musí jít snadno přidat nový ovládací prvek apod.

3. Modulární systém

3.1. Analýza

Původní stav systému je následující: algoritmy, jejichž kód je psaný v C/C++ existují jako konzolové aplikace, jejich vstupní parametry se jim zadávají při spuštění z příkazové řádky. Naším cílem je systém, ve kterém můžeme stejným způsobem přistupovat k více algoritmům stejného i různého typu najednou, nastavovat jejich vstupní parametry, spouštět je a umožnit mezi nimi interakci.

Nabízí se dvě řešení:

- ponechat stávající algoritmy tak jak jsou (tj. konzolové aplikace, kterým jsou předávány parametry z příkazové řádky) a modulární systém vytvořit jako aplikaci, která je bude s příslušnými parametry spouštět;
- používat přímo zdrojový kód algoritmů a upravit ho tak, aby vyhovoval modulárnímu systému;

První řešení má řadu výhod: je implementačně jednoduché, nemusí se provádět žádný další zásah do zdrojových kódů algoritmů. Nevýhodou je, že by se spolupráce omezovala pouze na prvotní nastavení parametrů, žádné další zásahy (pozastavení algoritmu, storno, výpis lokálních výsledků) by nebyly možné dokud by algoritmus nedoběhl.

Druhé řešení je implementačně složitější, aplikace musí být napsána v jazyce, ve kterém jsou psány zdrojové kódy algoritmů, tzn. C/C++, také se musí provést změny přímo ve zdrojových kódech algoritmů, ale neumožňuje žádané zásahy do průběhu algoritmu a celá aplikace je z vnějšího pohledu jednotná (netvoří ji několik relativně nezávislých spustitelných souborů jako v prvním případě). Další výhodou je, že systém informaci o „vstupních parametrech“ algoritmu může sdílet s každým algoritmem přímo na úrovni kódu. Z těchto důvodů jsem se rozhodl pro druhé řešení.

Vágní pojem algoritmus musíme přesně rozlišit na:

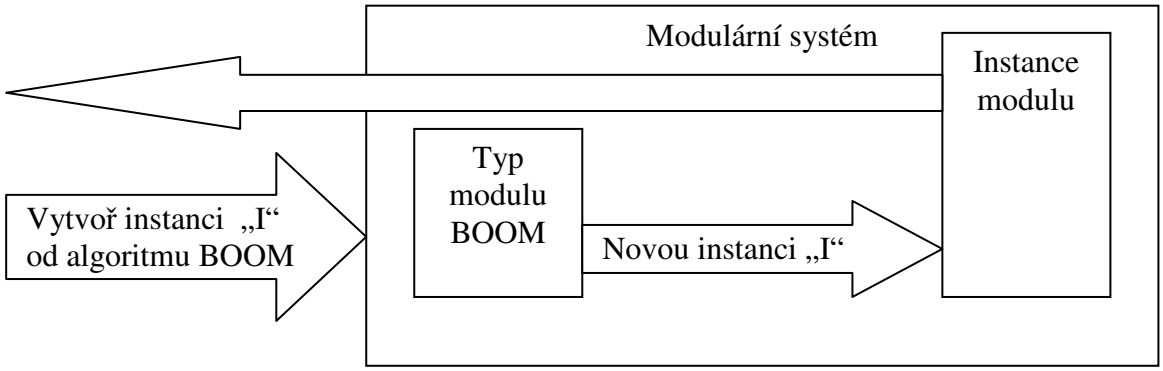
- typ algoritmu, dále označovaný jako **typ modulu**;
- instanci algoritmu, která reprezentuje vytvořený typ algoritmu v našem systému. Dále bude označovaná jako **instance modulu**. Aby byly jednotlivé instance modulů pro uživatele rozlišitelné, bude muset mít každá instance modulu v rámci systému unikátní identifikátor.

Toto rozlišení se dotýká i „vstupního parametru algoritmu“, který budeme přesněji rozlišovat na:

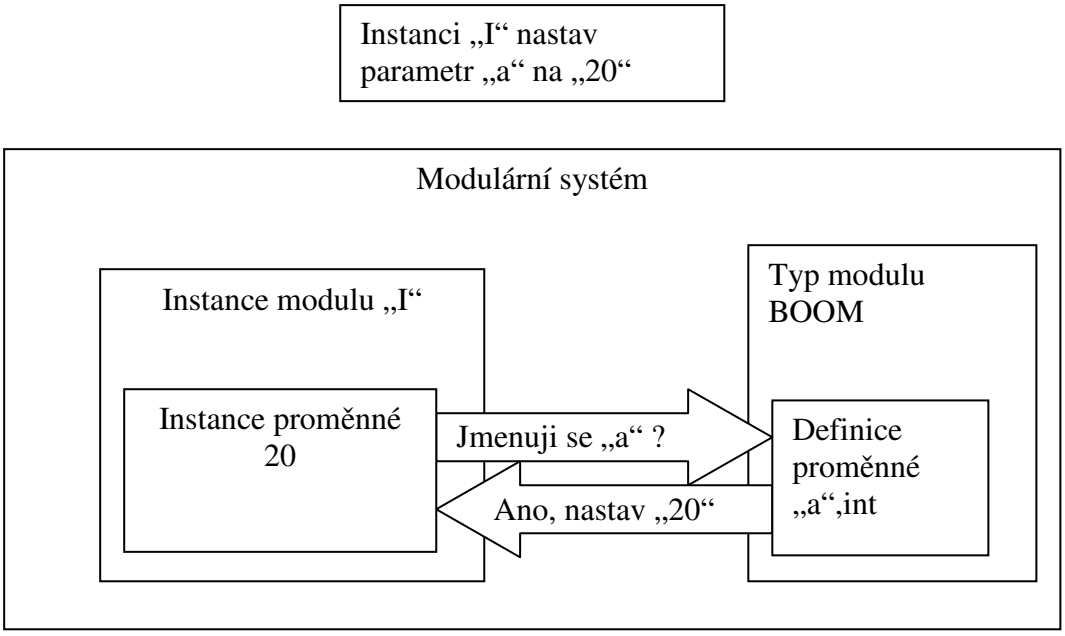
- název a typ vstupního parametru (typ vstupního parametru, vzhledem k faktu že se všechny parametry předávaly pouze přes příkazovou řádku, může být pouze jeden ze standardních C++ typů, tj. bool, int, double, string); tyto vlastnosti jsou spjaty s typem modulu; dále je budeme označovat jako **definici proměnné**;
- hodnota vstupního parametru; tato vlastnost je spjata s instancí modulu; dále ji budeme označovat jako **instanci proměnné**;

Základní funkce modulárního systému budou tyto:

- vytvoření nové instance modulu – vstupem bude typ modulu a jméno instance
- spuštění instance modulu – vstupem bude instance modulu
- nastavení parametrů instance modulu – vstupem bude instance modulu, jméno parametru a hodnota
- zaslání zprávy, adresátem mohou být buď všechny instance, instance určitého typu nebo jen konkrétní instance



Obr. 1 Vytváření nové instance modulu



Obr. 2 Nastavování parametrů instance modulu

3.2. Návrh

Při návrhu, a to nejen této části, ale i všech ostatních, jsem se rozhodl pro objektově orientovaný přístup.

Objektově orientovaný přístup přináší řadu výhod :

- *Abstrakce* – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.
- *Zapouzdření* – zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje *rozhraní*, pomocí kterého (a nijak jinak) se s objektem pracuje.
- *Dědičnost* – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou *dědit* z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do *tříd*, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).
- *Polymorfismus* – odkazovaný objekt se chová podle toho, jaký je jeho skutečný typ. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší. V praxi se tato vlastnost projevuje např. tak, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy (třídy, která přímo či nepřímo z této třídy dědí), která se může chovat jinak, než by se chovala instance rodičovské třídy, ovšem v rámci mantinelů, které jsou dané popisem rozhraní.

Další pojednání o OOP najdete na [5].

Již v analytické části jsem popsal několik základních objektů, je to instance modulu, typ modulu, definice proměnné a instance proměnné.

Instanci modulu navrhnu jako abstraktní třídu *IModule*. Bude obsahovat identifikátor, který bude unikátní v rámci celého systému. Dále bude obsahovat instance proměnné a přístupové metody k nim. Aby se dalo zjistit, jaký typ algoritmu reprezentuje, bude obsahovat odkaz na typ modulu (*ModuleType*). Pro příjem zpráv se vytvoří virtuální metoda *receiveMessage*, jejíž parametrem bude instance základní třídy od níž jsou všechny třídy odvozené a zároveň obsahuje typovou kontrolu. Pro spouštění vlastního algoritmu se vytvoří virtuální metoda *run*.

Typ modulu se vytvoří jako abstraktní třída *ModuleType*. Bude obsahovat virtuální metodu *getName,* která bude vracet vlastní jméno algoritmu, dále bude obsahovat virtuální metodu *getVariables*, která bude vracet seznam všech definic proměnných (*VarDef*) a také virtuální metodu *newModule*, která bude sloužit jako factory pro třídu *IModule*. Dále bude obsahovat seznam instancí třídy *IModule* jí náležících, a samozřejmě jednak přístupové metody k nim, jednak metodu na zaregistrovaní instance *IModule* do tohoto seznamu a metody pro posílání zpráv (buď všem instancím třídy *IModule* ze seznamu, nebo jenom těm s konkrétním unikátním identifikátorem). Při zaregistrování instance *IModule* se zároveň vytvoří seznam instancí proměnných (přesně podle jejich definic).

Programátor nového modulu bude muset pro začlenění svého modulu podědit tyto dvě třídy. Bude muset zařídit, aby virtuální metoda *newModule* vytvářela jeho podtřídu třídy *IModule* a aby virtuální metoda *getName* vracela jméno nového algoritmu. Dále musí vytvořit seznam definic proměnných (typicky statickým polem), který bude vracet virtuální metoda *getVariables.*

Definici proměnné bude reprezentovat třída *VarDef*. Jejím důležitým členem je typ, tj. typ, který bude mít instance proměnné. Protože jsou možné typy pevně dané (je to vlastně množina základních C typů), bude jej reprezentovat výčtovým typem. Další důležitou hodnotou je název vstupní proměnné. Třída může obsahovat ještě další členy: popis proměnné, výchozí hodnotu. Třída bude kromě obvyklých přístupových metod k výše jmenovaným členům obsahovat už jenom jednu metodu: metodu na vytvoření instance proměnné *instantiate*.

Instanci proměnné bude reprezentovat třída *VarInst*. Tato třída bude obsahovat odkaz na instanci třídy *IModule,* které patří, a odkaz na instanci třídy *VarDef*, která obsahuje její definici. Tato třída bude rodičovskou třídou pro třídy obsahující navíc vlastní proměnnou.

Třída	**Typ proměnné v C/C++**
VarBool	bool
VarChar	char
VarInt	it
VarDouble	double
VarString	string

tabulka 1 Typy instancí proměnných modulárního systému

Poslední třídou, bude třída *ModuleManager*. Bude reprezentovat modulární systém jako celek a jejím členem bude seznam typů modulů (*ModuleType*) a příslušné přístupové metody. Krom toho bude obsahovat metodu pro posílání zpráv všem instancím třídy *IModule*.

4. Shellové rozhraní

4.1. Analýza

Shellové rozhraní by mělo zpřístupňovat základní funkce modulárního systému (popsány jsou v 3.1). Krom toho a obecných požadavků (celý systém musí být multiplatformní) by samozřejmě mělo nabízet funkce shellu: vytváření proměnných, cykly, rozhodování, nahrazování parametrů apod.

Je nutné se rozhodnout, jakým směrem se vydat: vytvořit nový shell „from the scratch“ nebo použít a přizpůsobit nějaký existující shell. Nevýhoda vlastního shellu je velká implementační náročnost, nutnost tvorby vlastního řešení, které nebude dosahovat kvalit (a odladění) řešení existujících. Malou výhodou je, že neznamená žádnou další zátěž na „potřebné knihovny“. Proto jsem se rozhodl pro použití existujícího řešení.

4.1.1. Tcl

Je to zkratka vzniklá ze slov *Tool Command Langugage*. Označuje dvě skutečnosti: skriptovací jazyk a interpret, který je navržen tak, aby šel snadno začleněnit do aplikace. Tcl a k němu přidružené grafické uživatelské rozhraní, Tk, byly navrženy a vytvořeny profesorem John Ousterhout z University of California, Berkley. Je volně stáhnutelný z Internetu a je určen pro volné použití, dokonce i pro komerční aplikace. Tcl interpret byl naportován na prostředí UNIX, DOS, PalmOS, VMS, Windows, OS/2, NT a Macintosh. Grafické rozhraní bylo naportováno z X windows systém na Windows a Macintosh.

Podrobně si o Tcl můžete přečíst v [3].

Za základ pro implementaci rozhraní jsem použil skriptovací jazyk Tcl.

Tcl přináší tyto výhody:

- **je odzkoušený a odladěný –** jeho první verze jsou datovány rokem 1988;
- **mezi skriptovacími jazyky už je standardem -** což je další výhoda oproti tomu, kdyby se shell psal „from the scratch“ (uživatel se už nemusí učit syntax shellu jen pro použití v jedné velice speciální aplikaci);
- **má silnou vazbu na jazyk C -** knihovny Tcl lib a Tk lib umožňují přístup téměř ke všem vlasntostem shellu, ať už jsou to proměnné, události, příkazy pomocí C API;
- **je multiplatformní** Tcl/Tk bezchybně funguje na třech hlavních platformách - Linux, MS Windows a MAC OS;

4.2. Návrh

I když díky použití Tcl se nemusí navrhovat ani implementovat vlastní shell, stále ještě se musí upravit toto rozhraní tak, aby umožnilo správu našeho sytému. Je tedy nutné navrhnout sadu příkazů, kterými budeme ovlivňovat náš systém, zejména jsou to příkazy pro vytvoření nové instance modulu, spuštění instance modulu a nastavení parametrů instance modulu. Krom toho bude vhodné navrhnout příkazy pro informativní výpisy a nápovědu.

Příkaz	Argumenty	Popis
module	**typ_modulu, jméno_instance**	Vytvoří novou instanci modulu (instanci algoritmu) s unikátním identifikátorem

		jméno_ instance, tato instance bude odvozena od typu **typ_modulu** , pokud **typ_modulu** neexistuje nebo instance modulu s identifikátorem **jméno_ instance** již v systému existuje, ohlásí chybu
modlist		Vypíše v systému existující typy modulů.
vars[1]	**jméno_instance**	Vypíše hodnoty a názvy všech vstupních proměnných instance s identifikátorem **jméno_instance**, nebo ohlásí chybu pokud tato instance modulu neexistuje.
sets	**jméno_instance, jméno_proměnné, hodnota**	Nastaví instanci proměnné se jménem **jméno_proměnné** v instanci modulu **jméno_instance** hodnotu **hodnota**
run	**jméno_instance**	Spustí vlastní algoritmus na instanci modulu s identifikátorem **jméno_instance**
on	**jméno_instance, jméno_příkazu, param1, param2, …, paramN**	Vykoná „příkaz“ specifický pro ten který typ modulu. Tento příkaz se přeloží do zprávy (argumenty **jméno_příkazu, param1, param2, …, paramN**) která se pošle instanci modulu s identifikátorem **jméno_instance**
helps	**[jméno_příkazu]**	Vypíše nápovědu nad námi vytvořeným příkazem **jméno_příkazu**, nebo vypíše seznam námi vytvořených příkazů.
default	**jméno_instance**	Slouží pro zjednodušení přístupu, po jeho provedení se instance modulu s identifikátorem **jméno_instance** stane výchozí, takže se použije pokud u příkazů **vars**, **sets** či **run jméno_instance** neuvedeme

tabulka 2 Příkazy pro správu modulárního systému

[1] Tcl umožňuje správu svých proměnných tak, že jsou přímo slinkované s proměnou v C. To by sice trochu zjednodušilo implementaci, ale tím by se také stal modulární systém závislý na knihovnách Tcl. Proto ponechám proměnné modulárního systému oddělené, a tím pádem se také budou lišit příkazy pro nastavení/odečtení jejich hodnoty v shellu (vars, sets) od příkazů pro nastavení/otečtení shellových proměnných (set).

5. Grafické API

5.1. Analýza

Mým cílem je nabídnout programátorovi aplikace použitelné prostředky pro tvorbu grafiky. Musí se tedy použít některá z knihoven na tvorbu „okenní“ grafiky – např. známá Gtk,viz [2]. Důležité je, aby knihovna byla multiplatformní. Pro základ grafického rozhraní jsem se rozhodl použít knihovnu Tk lib.

Důvody pro její použití jsou následující:

- odpadá nutnost svazovat rozhraní s další extra knihovnou kvůli grafickým ovládacím prvkům. Tcl/Tk se dodává v jednom balíčku a Tcl jsme použili jako základ pro shellové rozhraní.
- odpadá nutnost složitě provazovat tok událostí shellu a grafiky (což by byla nutnost v případě použití knihovny Gtk). Tk jako nadstavba nad Tcl používá stejný tok událostí.

Nevýhodou je, že knihovna Tk lib nenabízí dostatek funkcí pro tvorbu grafiky přímo z jazyka C. Je koncipována spíše pro programátora nových Tk ovládacích prvků - widgetů. Má sice silnou podporu pro manipulování s okny, ale téměř postrádá jakékoliv metody pro práci s existujícími widgety, takže mnoho grafických ovládacích prvků je nutno vytvořit tím, že se v textové formě pošlou do skriptovacího rozhraní, což samozřejmě není optimální jak z hlediska rychlosti, tak z hlediska určité elegance kódu.

5.1.1. Úvod do problematiky tvorby grafiky v Tk

Tk je toolkit pro programování grafických uživatelských rozhraní. Byl navržen pro X window systém používaný na UNIX systémech a později byl naportovaný do prostředí Macintosh a Windows. Tk sdílí mnoho konceptů s ostatními toolkity na tvorbu *windows*. Tk poskytuje množinu Tcl příkazů, které vytváří a manipulují s tzv. *widgets*. Widget je okno v grafickém uživatelském rozhraní, které má určitý vzhled a chování. Termíny *widget* a *window* se často používají zaměnitelně. Mezi základní typy widgetů patří tlačítka, posuvníky, menu a textové okna.

Widgety jsou v Tk organizovány hierarchicky. Pro aplikaci hierarchická organizace widgetů znamená, že existuje tzv. primární okno (*primary window*) a uvnitř tohoto okna je obsaženo mnoho dalších synovských oken. Tato synovská okna mohou obsahovat další atd. Tak jako používá souborový systémem adresáře, které slouží jako kontejnery pro soubory a adresáře, používá hierarchický okenní systém okna jako kontejnery pro další okna. Podobné je i označení jednotlivých widgetů. Hlavní přístupový bod, tj. primární okno (v analogii s unixovým root adresářem ‘/‘) se značí ‘.‘. Tlačítko „ok“ umístěné v okně „hello_world“ bude mít unikátní identifikátor ‘.hello_world.ok‘ atd., analogie se souborovým systémem je zřejmá.

Widgety jsou pod kontrolou manažeru - *geometry manager* – který určuje jejich velikost a rozmístění na obrazovce. Dokud se geometry manager nedozví o widgetu, nebude widget namapován na obrazovku a tedy nebude vidět. Tk má mocné geometry managery, pomocí nichž se snadno vytváří hezké rozvržení ovládacích prvků. Základem je použití *frame* widgetů jako kontejnery pro další widgety. Jeden nebo více widgetů je vytvořeno a pak umístěno

v *frame* widgetu pomocí geometry manageru. Vkládáním *frame* widgetů do *frame* widgetů se může vytvořit komplexní rozvržení.

Existují tři základní typy manažeru (*geometry manager*):

- **pack** je constraint-based layout manager. To znamená, že ke každému widgetu jsou přiřazena určitá data (označovaná jako constraints – omezení) podle nichž se geometry manager rozhodne, kam který prvek umístí. Trochu připomíná BorderLayout známý z jazyka Java, tj. základní umístění nového prvku do kontejneru je ovlivňováno jednou ze čtyř stran, ke které bude přiléhat (top, right, bortím, left). Pack je ovšem mnohem složitější než BorderLayout, umožňuje roztahovaní prvků do různých stran (anchoring), nastavování vnitřního a vnějšího paddingu ovládacích prvků apod.
- **grid** je tabulkový manger, použijeme-li analogii z jazyku Java, připomíná GridLayout (kterým byl jeho návrh inspirován). Základem jeho algoritmu je, že plocha kontejneru je rozdělená do řádek a sloupců, kam umísťujeme jednotlivé widgety. Trochu nepříjemné je, že se nikde nenastavuje počet sloupců a řádků takto vzniklé tabulky, ale tento počet je dynamicky odvozen od "nejdále" položeného ovládacího prvku. Tzn. že když např. chceme pomocí tohoto manageru zobrazit widget *button* přesně vycentrovaný na prostředek okna, a chceme aby měl velikost 1/3 okna, musí se přidat *button* do políčka [1,1] (řádky i sloupce jsou indexované od 0), ale ještě se musí do políčka [2,2] přidat nějaký ovládací prvek (prázdný widget *frame*), protože teprve až potom se roztáhne mřížka layoutu do požadované velikosti 3x3.
- **place** je asi nejexaktnější layout manager. U každého prvku se specifikuje jeho [x,y] souřadnice v pixelech, a jeho výška a šířka v pixelech. Tyto souřadnice je též možno zadat jako relativní vzhledem k nadřazenému prvku. Zvláštností **place** manageru je také to, že se dá použít i k nastavení rozložení mezi jednotlivými okny.

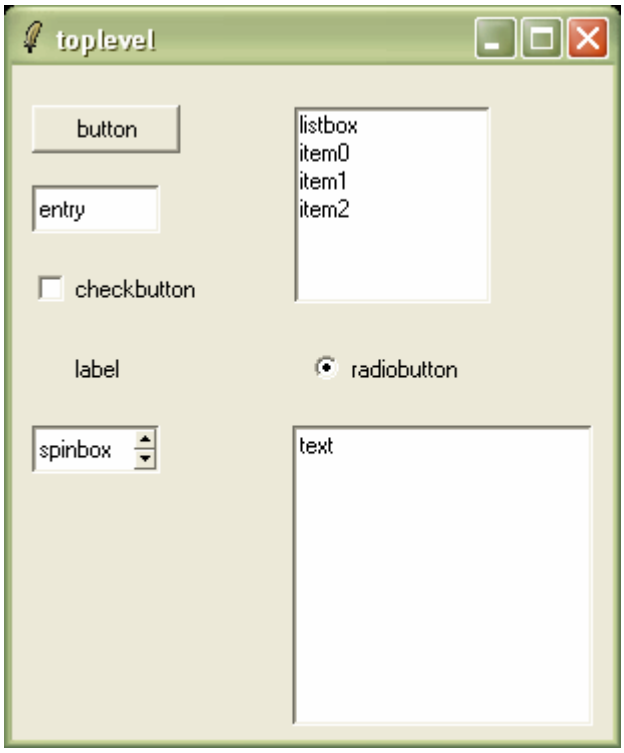
Tk aplikace je řízena tokem událostí, jako většina toolkitů na tvorbu okenních systémů. Tk widgety reagují na většinu událostí automaticky, takže programování aplikace zůstává jednoduché. Pro vlastní obsluhu událostí slouží příkaz *bind*, který každé události přiřadí Tcl příkaz – příkladem jsou pohyby myši, stisky klávesnice, změna velikosti, zavření okna.

Základní typy widgetů jsou:

- button – tlačítko
- checkbutton – přepínač (zaškrtávající tlačítko)
- entry – jednořádkový textový vstup
- label - popisek
- listbox – výběr ze seznamu více položek
- menu
- radiobutton – výběr ze skupiny možností
- spinbox – buď pro zadání celočíselného nebo reálného čísla, nebo pro výběr ze sady pevně stanovených hodnot
- scrollbar - posuvník
- text – víceřádkový vstup

Základní typy kontejnerových widgetů jsou:

- frame - panel
- toplevel – „okno“ ve běžném slova smyslu tj. reprezentuje okno na obrazovce



Obr. 3 Základní typy Tk widgetů, zobrazeno na systému Windows XP

Kromě těchto základních typů widgetů existují ještě rozšiřující balíčky. Nejznámější rozšíření jsou balíčky BWidgets a Iwidgets. Tyto balíčky se dodávají k většině instalací. Podle verze obsahují různé počty rozšiřujících widgetů (BWidget verze 1.6 obsahuje kolem 30 widgetů, Iwidgets verze 4.0.2 obsahuje kolem 60 widgetů !). Tyto balíčky nejsou zahrnuty v API poskytovaném programátorovi modulů pro svou rozsáhlost, widgety z nich se však dají vytvořit v případě potřeby přímo prostřednictvím shellu. Před tím je ovšem nutné tyto balíčky načíst. Tcl pro správu balíčků nabízí příkaz package.

Přesná syntax použití pro načtení balíčků Iwidgets a BWidgets z shellu je:

```
package require Iwidgets
package require BWidgets
```

Více viz [3].

5.1.2. Programátorský pohled na Tk

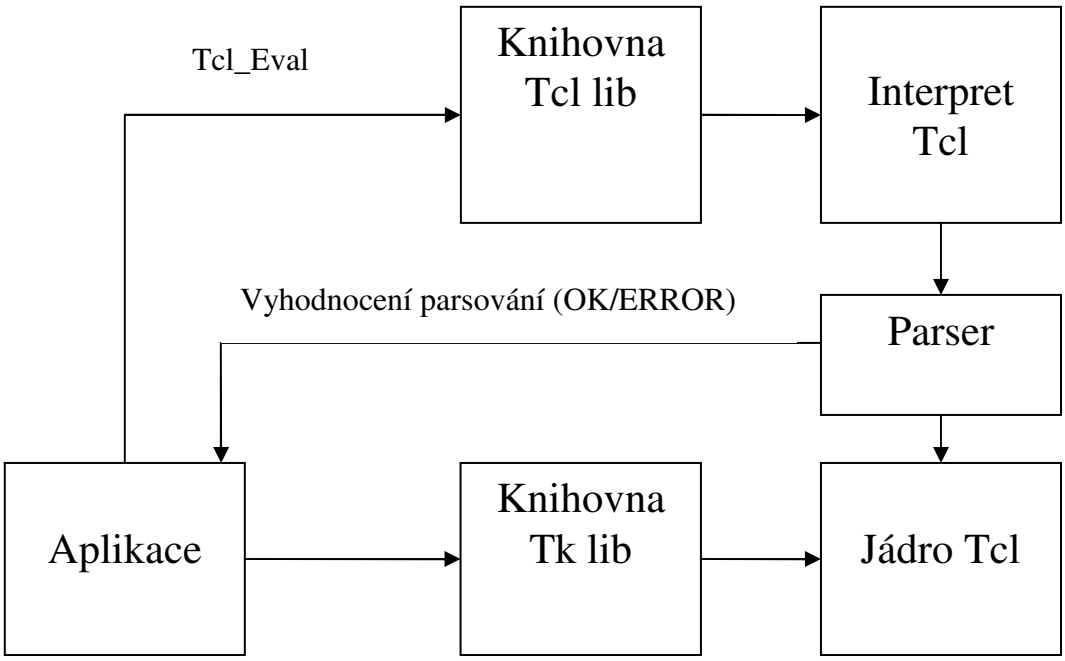
V zásadě existují dvě možnosti jak přistupovat k tvorbě grafického rozhraní prostřednictvím Tk. První je tvořit GUI přímo spouštěním příkazů v interpretu. Druhá je využít pomocí C API možností knihovny Tk lib. Její možnosti jsou však omezené.

V přehledu poskytuje Tk lib tyto základní funkce :

- vytváření toplevel widgetů, a jejich omezenou konfiguraci (velikost, okraje, kurzor), informace o jejich pozici
- nastavování parametrů widgetů
- podpora událostí na widgetech
- podpora obrázků všech rozšířených typů (png, bmp, jpg, gif, …)

- podpora jednotných identifikátorů
- podpora operací s barevnými schématy, fonty
- částečná podpora některých platformě-závislých záležitostí - ID managment u X-window, metody pro práci se známými hwnd identifikátory (window handle) systému Windows

Bohužel podpůrné C procedury knihovny Tk lib jsou orientované spíše na návrh nového typu widgetu než na nastavení parametrů existujícího. To se týká třeba podpůrných funkcí pro nastavování parametrů widgetu. To znamená, že při tvorbě grafického rozhraní pomocí Tk z aplikace se nevyhneme **přímé evaluaci skriptů v interpretu**.



Obr. 4 Rozdíl mezi použitím přímé evaluace a knihovny Tk

5.2. Návrh

Mým cílem je umožnit přes grafické rozhraní zadávání vstupních parametrů instancím modulu. Protože vlastní vzhled formulářů a nastavení vstupních algoritmů se bude lišit a bude v kompetenci tvůrců původních algoritmů, je důležité maximálně zjednodušit tvorbu formulářů.

Musíme tedy navrhnout vhodnou nadstavbu nad existujícími knihovnami tak, aby se programátor GUI nemusel uchylovat k přímé evaluaci v interpretu, i když tato možnost může zůstat zachována.

Tato nadstavba kromě odstranění přímé evaluace skriptů by měla:

- co nejvíce používat knihovní funkce Tk
- být rozšiřitelná
- obsahovat metody pro práci se všemi základními widgety, jejich konfiguraci a vytváření
- vzít ohled na to, jak jsou navrženy a strukturovány existující příkazy pro tvorbu GUI v interpretu
- být z hlediska programátora jednotná
- nabízet pohodlný přístup k obsluze událostí

Při návrhu jsem se rozhodl pro objektový přístup. Toto API jsem se snažil navrhnout tak, aby reprezentovalo co nejlépe logiku Tk a vnějškově bylo podobné balíčku AWT známého z Javy. To znamená pro každý jednotlivý typ widgetu vytvořit třídu, všechny tyto třídy podědit ze společného základu, reprezentujícího základní widget. Zvláštními třídami budou kontejnery, tj. widgety, které mohou obsahovat další widgety.

Další návrh rozdělím na dvě části:

- Návrh základních tříd reprezentujících základní widget a základní kontejner, realizujících přímou interakci s Tcl interpretem. Tato část bude důležitá pro programátora, který bude přidávat nové typy widgetů.
- Návrh tříd reprezentující jednotlivé typy widgetů. Tato část bude důležitá pro programátora aplikace.

Jména metod a struktur v Tk lib začínají prefixem Tk_, v Tcl lib prefixem Tcl_. Abych odlišil názvy našich nových tříd a metod od názvů metod a struktur knihoven Tk lib a Tcl lib, budu používat tuto konvenci: Jména našich tříd budou začínat pouze prefixem Tk popř. Tcl za kterým už nebude následovat podtržítko.

5.2.1. Návrh základního widgetu

5.2.1.1. Widget jako objekt

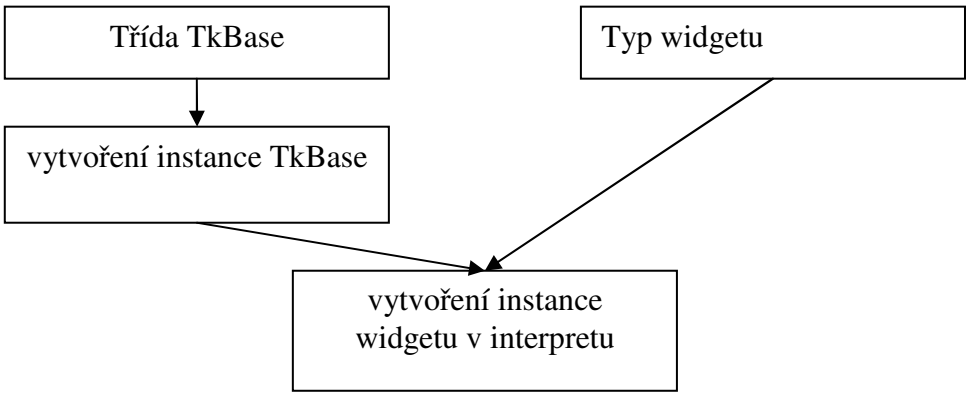
Podíváme-li se na Tk widget jako na objekt, mohou se určit následující společné vlastnosti :

- všechny mají stejný životní cyklus: nejdříve se musí vytvořit, poté se pomocí nějakého layout manageru musí zobrazit
- každý widget má svůj unikátní identifikátor
- každý widget má informaci o svém typu (button, frame, label, atd.)
- každý widget má sadu vlastností, které jsou specifické pro každý typ widgetu i když existují výjimky, jako je třeba barva pozadí; vlastností je celá řada, obsahují informaci o fontu ovládacího prvku, barevném nastavení, typu kursoru, textu zobrazeném na ovládacím prvku; mechanismus čtení a nastavovaní těchto vlastností je stejný.
- všechny typy widgetů reagují na základní množinu událostí
- všechny jsou umístěny v nějakém kontejneru, a to dokonce i *toplevel* widgety, které přináleží pod primární okno aplikace
- layout manager je založen na omezních - constrainsts; znamená to, že každému widgetu je přiřazen nějaký druh informace, podle které bude zobrazen..

Základní třídu reprezentující widget budeme označovat jako *TkBase.*
Výsledný UML diagram této třídy je v příloze UML diagramy, diagram 2 - Třída TkBase.

5.2.1.2. Životní cyklus

Životní cyklus třídy *TkBase* se musí podobat životnímu cyklu widgetu. Liší se od něj pouze jednou fází navíc. Když se vytvoří instance třídy *TkBase,* widget, který reprezentuje, ještě v interpretu nemusí existovat. Může se sice vytvořit automaticky při vytváření instance této třídy, ale narážíme zde pak na problém – identifikátor, pod kterým se widget vytváří je neměnitelný a musí odrážet umístění widgetu v stromové hierarchii. Mějme např. *toplevel* widget '.t'. Pokud bych vytvořil widget, který bude v tomto okně ležet, musí se nazvat ,.t.novywidget'. Jinými slovy, konstruktoru třídy *TkBase* by se musela předávat přesná strukturu rodičovských kontejnerů a aplikační programátor by si musel dávat pozor, aby vytvářel své formuláře přesně od hierarchicky „nejvyšších" widgetů k nejnižším



Obr. 5 Životní cyklus widgetu a TkBase

S tímto modifikovaným životním cyklem je možné ve fázi, kdy je vytvořena instance třídy *TkBase*, ale ještě neexistuje jako widget v interpretu, volně modifikovat hierarchická struktura widgetů. I po vytvoření instance widgetu je možné dál měnit hierarchickou strukturu s tímto omezením: **Instanci třídy TkBase, která již existuje v interpretu jako widget nelze přeřadit do jiného kontejneru.**

5.2.1.3. Unikátní identifikátor

Každý widget musí mít unikátní identifikátor – řetězec znaků. Proto musí i třída *TkBase* tento identifikátor obsahovat. Nebylo by vhodné nechávat vytvoření tohoto identifikátoru v rukou programátora aplikace, z důvodů:

- Znamená to (zbytečnou) práci pro programátora aplikace navíc.
- Na identifikátor jsou kladeny nároky co se týče formy, existují zakázané znaky, které nesmí obsahovat, a ty bychom museli kontrolovat. Programátor aplikace by si také musel pamatovat, které znaky jsou pro použití v identifikátoru přijatelné.

- Identifikátor nesmí kolidovat se jménem existujícího příkazu shellu (for,if, atd.)

Proto bude vhodné tento unikátní identifikátor generovat. Vytvořím třídu *TclGuid*, která bude obsahovat statický čítač, který po každém přečtení hodnoty inkrementuje. S tímto jednoduchým identifikátorem ale nevystačím, Tk interpret požaduje, aby identifikátor widgetu odrážel jeho umístění v hierarchické stromové struktuře. Vytvořím tedy metodu, která přečte strom až ke kořenu a z nalezených vygenerovaných identifikátorů poskládá výsledný identifikátor pro použití v interpretu.

5.2.1.4. Vlastnosti widgetu

Každý typ widgetu má specifickou sadu vlastností. Nastavování těchto vlastností je ale stejné – název vlastnosti, a hodnota vlastnosti se ve formě řetězce předá interpretu. Odečítání hodnoty vlastnosti je podobné, předáme interpretu název vlastnosti a interpret nám vrátí její hodnotu ve formě řetězce. Tyto vlastnosti jsou několika málo typů:

- standardní číselné typy – int, double
- pravdivostní hodnota – bool
- řetězec
- seznam
- proměnná shellu
- informace o barvě – ta může být buď reprezentována v RGB formě nebo v řetězci (bohužel platformě-závislém) který váže barvu na barevné nastavení systému
- rozměry a pozice obdélníku
- výčet (mnoho výčtových typů na nastavení okrajů, umístění ovládacího prvku a textu, apod.)

Různé typy vlastností mohu interpretovat tak, že vytvořím interface vlastností, který bude obsahovat dvě metody – inicializaci vlastnosti z řetězce, a výstup do řetězce. Pro každou jednotlivou vlastnost pak vytvořím vlastní třídu, která bude tento interface implementovat.

Třídou, kterou nebudu takto jednoduše implementovat, jsou výčty. Jejich vnitřní reprezentace je podobná – tvoří ji seznam povolených hodnot (řetězců) a ukazatel na aktivní hodnotu. Proto vytvořím abstraktní třídu, která bude implementovat výše uvedený interface a na konkrétních výčtových třídách ponechám pouze volbu sady hodnot.

Netriviální vlastností jsou také proměnné shellu.

Příklady jejich použití v Tk :

- u widgetu *radiobutton* zastupují roli skupiny. Všechny radiobuttony, které mají nastavenou stejnou shellovou proměnnou, patří do stejné skupiny. Chceme-li zjistit který radiobutton je zaškrtlý, musíme vypsat hodnotu této proměnné.
- u widgetu *entry* je jedinou metodou jak nastavit/získat hodnotu textu přistoupit k proměnné která je k němu přiřazena. Jedna je mu přiřazena vždy interpretem, její jméno je odvozeno od unikátního identifikátoru widgetu.

Problematice objektového přístupu k shellovým proměnných se budeme věnovat 7.3.2

5.2.1.5. Obsluha událostí widgetu

Běžnou metodou zpracování událostí v objektovém prostředí je vytvoření interface – tzv. posluchače, tento interface obsahuje jednu nebo několik prázdných virtuálních metod. Na ovládacím prvku samotném je kolekce těchto posluchačů a veřejné metody pro jejich přidávání a ubírání. Při vyvolání události se na všech posluchačích v kolekci zavolá příslušná metoda. Programátor aplikace pak vytvoří obsluhu události tak, že vytvoří vlastní třídu, která implementuje posluchače a instanci této třídy přidá do kolekce posluchačů. Já jsem se rozhodl pro stejné pojetí.

5.2.2. Návrh základního kontejneru, layout manageru

Základní třídu reprezentující kontejner budeme označovat jako *TkConteiner*, tato třída bude podtřídou třídy *TkBase*.

Mezi tzv. kontejnerové widgety se počítají widgety *frame* a *toplevel.* Kontejnerové widgety mají další rozšiřující vlastnosti:

- obsahují synovské widgety
- obsahují referenci na layout manager

Základní metody třídy *TkConteiner* jsou tyto:

- přidání synovského widgetu
- odebrání synovského widgetu

Layout manager si lze představit jako černou skříňku, od které chceme, aby umístila nebo odstranila widget z kontejneru.

- k přidání widgetu se musí layout manageru předat reference na kontejner, widget který chceme umístit a omezení pro umístění tohoto prvku.
- k odstranění widgetu se musí layout manageru předat reference na kontejner a widget který bude odstraněn

Vhodným návrhem je tedy interface - *TkLayout*, s dvěmi metodami pro přidání a odstranění widgetu.

Scénář přidání widgetu do kontejneru:

1. pokud je widget již vytvořen v shellu, nemůže se do kontejneru vložit, viz.5.2.1.2
2. přidej widget do kolekce synovských widgetů
3. jestliže je kontejner vytvořen, vytvoř i synovský widget
4. jestliže je nastaven layout manager, zavolej metodu pro umístění widgetu

Scénář odebírání widgetu z kontejneru:

1. pokud je widget umístěn, zavolej layout manager pro odebrání widgetu
2. odeber widget z kolekce synovských widgetů

Dále se bude *TkConteiner* lišit od třídy *TkBase* při vytváření v shellu. Scénář vytváření třídy TkConteiner je :

1. vytvoří se instance widgetu v shellu
2. projde se list všech synovských widgetů, které se rovněž vytvoří v shellu
3. pokud je nastaven layout manager, projde se znovu seznam všech synovských widgetů a provede se jejich rozmístění.

Kompletní UML schéma je v příloze UML diagramy, diagram 3 – Třída TkContainer.

5.2.3. Návrh jednotlivých typů widgetů

Jednotlivé typy widgetů vytvořím jako třídy, které se budou dědit buď z třídy *TkBase* nebo z třídy *TkContainer*. Přestože se implementace jednotlivých typů widgetů budou více či méně lišit, mým cílem je, aby se stejné metody mezi různými typy widgetů stejně jmenovaly, z důvodu přehlednosti pro programátora aplikace.

Jejich přehled najdeme v příloze UML diagramy, diagram 5 - Přehled základních typů widgetů.

Proto navrhnu vhodné společné interface pro:

- Změnu a nastavení stavu. Týká se widgetů, které mají dva stavy – aktivní, neaktivní.
- Přečtení a nastavení textu. Týká se skupiny widgetů jejichž obsah je reprezentován jednoduchým textovým řetězcem.
- Umístění scrollbarů (vertikálních, horizontálních). Týka se widgetů, které mohou mít scrollbar.

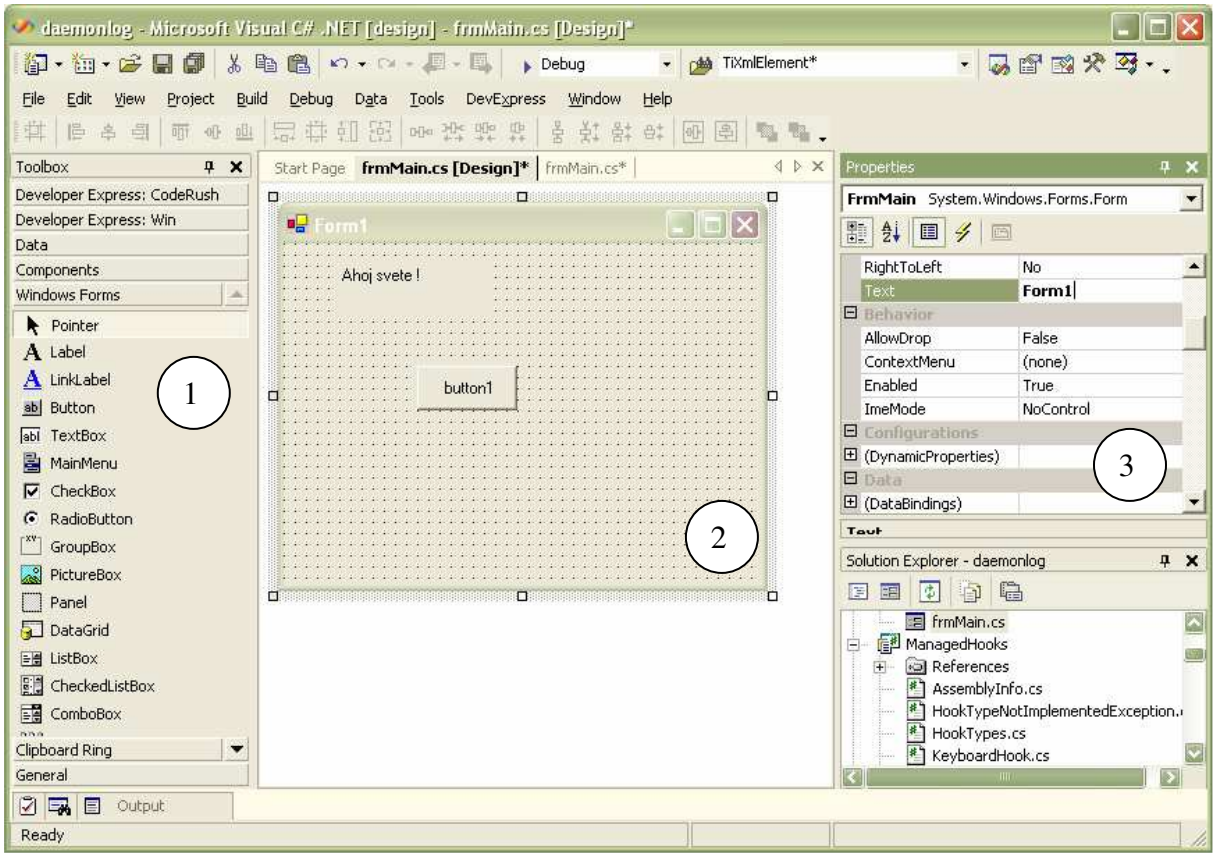
Tyto společné interface najdeme v příloze UML diagramy, diagram 4 - Základní typy widgetů a základní interface.

6. Designér

6.1. Analýza

Další ulehčení pro programátory GUI by bylo navržení designéru, který by pracoval metodou „what you see is what you get" a ulehčit tak programátorovi zdlouhavé psaní GUI kódem, z kterého hned není vidět výsledek. Podívejme se nejdříve na některé existující designéry.

6.1.1. Analýza designéru pro Microsoft Visual Studio .NET



Vysvětlivky :

1. **Toolbox - Seznam ovládacích prvků**
2. **Form - Formulář, kam ovládací prvky umisťujeme**
3. **Properties – Vlastnosti aktivního ovládacího prvku**

Obr. 6 Designér Microsoft Visual Studio .NET – návrh formuláře pro C#

Microsoft Visual Studio .NET je velice úspěšné komerční vývojové prostředí. Návrhář formulářů v tomto prostředí, jak ho vidíme na obrázku je velmi intuitivní a umožňuje rychlou tvorbu. Všimněme si, že nabídka ovládacích prvků je organizována přehledně hierarchicky do skupin a stejně tak jsou do skupin organizovány hierarchicky vlastnosti prvků. Uživatel navíc může u vlastností ovládacího prvku přepínat mezi hierarchickým zobrazením a čistě abecedním seznamem vlastností.

Uživatel komunikuje s návrhářem pomocí tří základních panelů – panel Toolbox, zobrazení vlastního formuláře a panel Properties.

Scénáře jeho použití jsou tyto:

- vkládání nového prvku:
  - uživatel si vybere z panelu Toolbox ovládací prvek;
  - uživatel najede myší na zobrazení formuláře a klikne;
  - příslušný ovládací prvek se vloží do formuláře, pokud se podíváme na okno formuláře zjistíme že je „tečkované“ tyto tečky udávají tzv. rozteč kreslicí mřížky a při vkládání se pozice vkládaného prvku zaokrouhlí k nejbližší tečce; můžeme tuto mřížku samozřejmě vypnout, ale u většiny formulářů je to velká pomoc;
  - aktivní prvek v panelu Toolbox se přepne na „šipku“ ta má speciální význam, není totiž ovládacím prvek, ale slouží pro výběr existujícího ovládacího prvku;
- změna aktivního prvku:
  - zde existují dva přístupy :
    - uživatel klikne na ovládací prvek přímo ve formuláři;
    - uživatel si ovládací prvek vybere ze seznamu (je nahoře v okně Properties) – výhodné pro malé/obtížně viditelné prvky;
- změna velikosti ovládacího prvku:
  - uživatel aktivuje ovládací prvek;
  - pak existují dvě možnosti:
    - tažením myši mu určí požadovanou, velikost která se opět zaokrouhlí do mřížky;
    - v Properties vybere vlastnost velikost a textově jí určí požadovanou hodnotu;
- změna umístění ovládacího prvku:
  - uživatel aktivuje ovládací prvek;
  - pak existují opět dvě možnosti:
    - tažením myši ho umístí do požadované pozice;
    - v Properties vybere vlastnost pozice a textově jí určí požadovanou hodnotu;
- nastavení vlastnosti ovládacího prvku:
  - uživatel aktivuje požadovaný ovládací prvek;
  - v panelu properties nastaví požadovanou vlastnost;
- smazání ovládacího prvku:
  - uživatel aktivuje ovládací prvek;
  - uživatel zmáčkne klávesu delete, popř. z vybere z menu;

Rozvržení ovládacích prvků na formuláři který vytváříme se „ukládá“ přímo do zdrojového kódu modifikací jedné metody na třídě která ho reprezentuje.

Výpis částí kódu která vytváří formulář na obr. 2

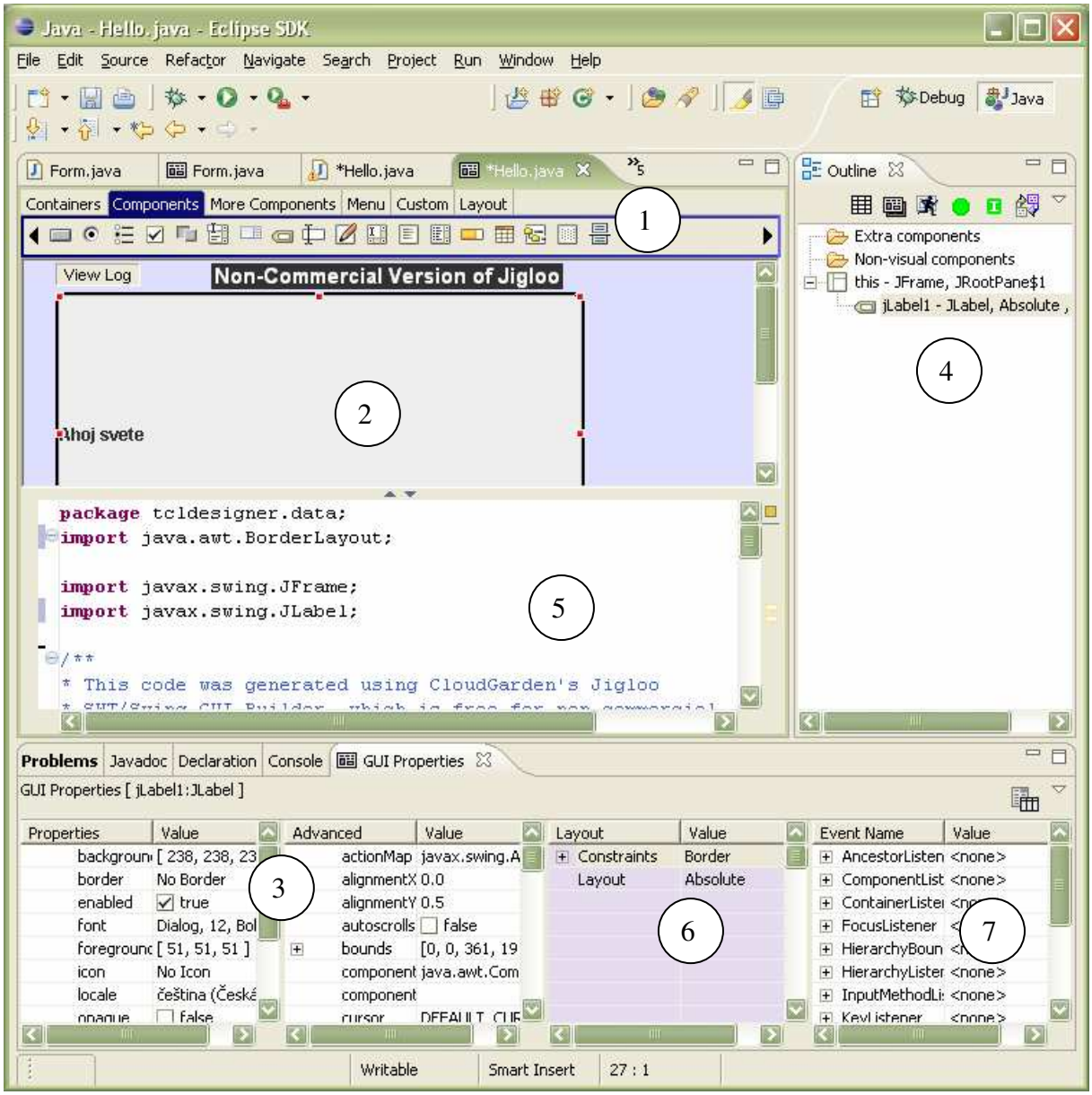
```
this.label1 = new System.Windows.Forms.Label();
this.SuspendLayout();
//
// label1
//
```

```
this.label1.Location = new System.Drawing.Point(40, 16);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(112, 40);
this.label1.TabIndex = 1;
this.label1.Text = "Ahoj svete !";

this.Controls.Add(this.label1);
this.ResumeLayout(false);
```

Funguje to ale i naopak, pokud upravíme kód (za předpokladu, že je přeložitelný), se změna promítne do rozložení ovládacích prvků v návrháři.

6.1.2. Analýza designéru Jigloo



Vysvětlivky:

1. **Toolbox – Seznam typů ovládacích prvků**
2. **Formulář**
3. **Properties – Panel vlastností ovládacího prvku**
4. **Seznam ovládacích prvků ve formuláři**
5. **Zdrojový kód formuláře**
6. **Seznam integritních omezení a layoutů**
7. **Seznam událostí napojených na ovládací prvek**

Obr. 7 Designér Jigloo – návrh jednoduchého formuláře

Jigloo je rozšíření vývojového prostředí Eclipse. Eclipse je jedno z nejrozšířenějších vývojových prostředích pro jazyk Java. Jigloo je k dispozici pro nekomerční použití zdarma.

Na první pohled si všimneme, že jeho ovládací prvky jsou rozmístěny chaotičtěji než u Microsoft Visual Studia. To je způsobeno jednak zřejmou, ale zbytečnou snahou vidět kód

i formulář najednou (např. panel 7 – panel událostí je nadbytečný, ty nás zajímají na úrovni kódu), pak také odlišným pojetím programátorského pohledu na grafiku v jazyku Java a C#. Zatímco dva hlavní balíčky javy na tvorbu grafiky – awt a swing používají layoutový přístup, WinForms (základní window toolkit jazyka C# v prostředí Windows) používá většinou absolutní rozmístění prvků. Proto zde musí být hierarchický strom který ukazuje, jaký layout má kontejner a jaké ovládací prvky v tomto kontejneru leží.

V srovnání s návrhářem z Visual Studia nalezneme další drobné nedostatky:

- Chybí nástroj „šipička“ pro výběr ovládacích prvků. Pokud vybereme ovládací prvek a pak si to rozmyslíme, musíme kliknout mimo okno formuláře (značně neintuitivní).
- Toolbox je umístěn nahoře a tak zatímco u Visual Studia si přečteme název ovládacího prvku, zde jsme odkázáni na pochopení ikony (jejich význam je ovšem pro trochu zkušenějšího programátora v jazyku Java naštěstí většinou zřejmý).
- U Visual Studia je formulář zobrazen tak, jak bude vykreslen na obrazovce (celé okno), u Jigloo jsme omezeni na zobrazení vnitřku.

Scénáře jeho použití jsou tyto:

- vkládání nového prvku:
  - uživatel si vybere z panelu Toolbox ovládací prvek;
  - uživatel najede myší na zobrazení formuláře a klikne;
  - zobrazí se dialog, kde se vyplní název prvku (u Visual Studia je to jedna z vlastností), potvrdí se pod který layout patří;
  - příslušný ovládací prvek se vloží do formuláře;
- změna aktivního prvku:
  - zde existují dva přístupy:
    - uživatel klikne na ovládací prvek přímo ve formuláři (nesmí být vybrán žádný ovládací prvek v panelu Toolbox);
    - uživatel si ovládací prvek vybere ze stromu;
- změna velikosti ovládacího prvku:
  - uživatel musí změnit typ layoutu, nebo integritní omezení ovládacího prvku;
- změna umístění ovládacího prvku:
  - uživatel musí změnit typ layoutu, nebo integritní omezení ovládacího prvku;
- nastavení vlastnosti ovládacího prvku:
  - uživatel aktivuje požadovaný ovládací prvek;
  - v panelu properties nastaví požadovanou vlastnost;
- smazání ovládacího prvku:
  - uživatel aktivuje ovládací prvek;
  - uživatel zmáčkne klávesu delete;
  - aktivuje se dialog „opravdu smazat“;
  - uživatel tento dialog potvrdí;

Vidíme, že scénář akcí je velmi podobný (je tu snad jen více potvrzování) scénáři akcí z Microsoft Visual Studia. Stejné je i „uložení“ formuláře: změny se aplikují přímo do příkazů ve zdrojovém kódu.

6.1.3. Shrnutí analýz

U obou dvou návrhářů je vidět řada stejných postupů a vlastností, kterých bude dobré se přidržet i u jakéhokoliv jiného návrháře – kvůli dodržování uživatelského standardu.

Základní ovládání návrháře se děje pomocí těchto oken:

- okno s nabídkou typů ovládacích prvků (vhodný je také nástroj „šipička“ použitý ve Visual Studiu) – tzv. toolbox
- okno se seznamem nastavitelných vlastností aktivního ovládacího prvku – tzv. property editor
- zobrazení vlastního formuláře tak jak ho uvidíme z aplikace

Základní scénář akcí je tento:

- vkládání nového prvku:
  - uživatel si vybere z panelu toolbox typ ovládacího prvku;
  - uživatel najede myší na zobrazení formuláře a klikne;
  - ovládací prvek se vloží do formuláře;
- změna aktivního prvku:
  - uživatel klikne v režimu „šipička“ na ovládací prvek na formuláři;
  - uživatel si tento prvek vybere ze seznamu;
- nastavení vlastnosti ovládacího prvku:
  - uživatel aktivuje požadovaný ovládací prvek;
  - v panelu properties nastaví požadovanou vlastnost;
- smazání ovládacího prvku:
  - uživatel aktivuje požadovaný ovládací prvek;
  - uživatel zmáčkne tlačítko delete (na klávesnici, nebo na některém panelu), zvolí příkaz „smaž“ v menu, popř. kombinaci těchto akcí; může být požadováno potvrzení smazání jak to známe v návrháři Jigloo;
  - ovládací prvek se smaže;

6.2. Návrh

Základním požadavkem návrhu designéru bude samozřejmě vnějškově dodržet uživatelské standardy pro návrhář.

Problém, který jsem doposud přehlížel, je jak oba dva analyzované návrháře uchovávali rozvržení ovládacích prvků na formuláři. Microsoft Visual Studio .NET i Jigloo uchovávali toto rozvržení přímo ve formě zdrojového kódu pro ten daný jazyk. C# a Java jsou interpretované jazyky. Není proto těžké vytvořit vazbu mezi zdrojovým kódem a zobrazeným formulářem. C/C++ ovšem interpretovaný jazyk není. Proto musíme navrhnout mechanizmus jak navržený formulář uložit, a jak k němu z aplikace přistupovat.

Jelikož navrhované formuláře, veškeré ovládací prvky (ostatně stejně tak i návrhář samotný) budou vytvořeny na našem objektovém rozhraní, tj. budou objekty, může se ukládání vyřešit jednoduše pomocí mechanizmu serializace objektů a načítání pomocí deserializace. Tím se sice dosáhne toho, že programátor aplikace bude již moci načíst formulář v objektové formě, ale programátor aplikace již nebude mít možnost dostat se na jeho jednotlivé ovládací prvky. Proto se musí ještě v návrháři umožnit přiřadit ovládacímu prvku identifikátor, který bude označovat jako jméno ovládacího prvku. Přes toto jméno se pak programátor aplikace již může dostat ke konkrétnímu ovládacímu prvku.

7. Implementace

Jazyk C++ je označován jako hybridní (objektově orientovaný přístup můžeme ale nemusíme využívat, s třídou nemůžeme pracovat jako s objektem), není sice úplně vhodný jazyk pro OOP, ale musím si s jeho omezenými možnostmi vystačit, protože jsme jím omezeni (je v něm psán zdrojový kód algoritmů). V první části se pokusím „obejít“ některé z jeho nedostatků. V dalších si povíme o specifických problémech v implementaci jednotlivých částí systému.

Krom jazyka C++ byly při implementaci použity tyto knihovny:

Knihovna	**Verze**	**Projektové stránky**
Tcl lib	8.4	www.activestate.com
Tk lib	8.4	www.activestate.com
Tk img	1.3	www.activestate.com
tinyxml	2.4.0	sourceforge.net/projects/tinyxml/

tabulka 3 Použité knihovny

7.1. Obecné problémy

7.1.1. Návrh základní třídy

Všechny třídy budou odvozeny od třídy *TObject*. Tu navrhnu tak, aby odstranila nevýhodu C++, fakt že samotná třída není objektem. Zároveň bude obsahovat metody pro jednoduchou správu paměti, které popíši později.

```
class TObject {
public:
      ...

      static const char* type;

      virtual const char* getType() const;
      virtual bool hasParent(const char* className) const;
      virtual bool compareTo(TObject* object) const;
      virtual const string& toString() const;
      virtual TObject* clone();

      bool typeOf(const char* type) const {
            return getType()==type;
      }

      ...
};
```

Mechanizmy které bude zajišťovat třída *TObject*:

- kontrolu typu: každá podtřída bude obsahovat statický veřejný prvek `type`, který je ukazatelem na řetězec obsahující název třídy; každá podtřída bude obsahovat virtuální metodu *getType()*, která u každé podtřídy vrátí tento ukazatel `type`. Další metodou navrhnutou pro podporu typové kontroly bude metoda *typeOf(const char*)*, která

porovnává typ třídy proti řetězci. Pro zrychlení se při testu na typ třídy nebude provádět porovnání textovým srovnáním ale porovnání ukazatelů, a to proto, aby tato často využívaná metoda pracovala rychle.

- kontrolu dědičnosti: virtuální metoda *hasParent(const char*)* provádí test zda třída je potomkem dané třídy. Opět se pro vyhodnocení používá porovnání ukazatelů.
- srovnání: virtuální metoda *compareTo(TObject*)* porovnává dva objekty;
- jednoduchý informativní výpis : virtuální metoda *toString()*;
- vytváření hluboké kopie : virtuální metoda *clone();*

7.1.2. Správa paměti

Častým nedostatkem aplikací psaných v C++ je tzv. „tečení paměti“, tj. že aplikace nevrací paměť, kterou alokuje. Stává se to běžně, zvlášť u větších projektů. Abych tomuto problému předešel, zvolil jsem vytvoření jednoduchého správce paměti. A to formou počítání referencí, i když je počítání referencí má své nevýhody – jedna zrušená reference může vést ke kaskádní dealokaci mnoha objektů (i když tomuto chování se dá zamezit např. pravidlem 2 dealokace za jednu alokaci, což se používá v aplikacích kde záleží na rychlosti), počítání referencí je vždy pomalejší než některý z algoritmů pro tracing garbage collection, protože pokaždé když se reference změní, musí se změnit i čítač. Tento typ správy paměti je také náročný na místo, u každého objektu musíme mít čítač referencí. Navíc vytváření tzv. kruhových referencí vede k tomu, že se taková oblast paměti nikdy neuvolní (resp. až do skončení programu).

Přesto jsem zvolil tohoto správce paměti, a to proto, že jeho naivní implementace je jednoduchá. Existují sice hotové správce paměti, používající některý z algoritmů tracing garbage collection, ale vzhledem k povaze problému (nejedná se o návrh aplikaci která musí nutně pracovat v reálném čase, není nutné zmenšit velikost alokované paměti na nejmenší možnou hodnotu) je počítání referencí vyhovující.

Více o správcích paměti v [6].

Prakticky správce paměti zrealizujeme přidáním odpovídajících členů do základní třídy *TObject* :

```
class TObject
{
      mutable int refCount;
public:
      TObject() {refCount=0;}
      void upcount() const {refCount++;}
      void downcount() const {
            refCount--;
            if (refCount==0) delete this;
      }

      ...
};
```

Dále navrhneme třídu *Ptr* která bude vlastní počítání referencí sama realizovat. Třída *Ptr* bude generická a bude obsahovat ukazatel na třídu `TObject`, pomocí kterého bude spravovat reference.

Zjednodušená třída *Ptr*, ukázka základního mechanizmu počítání referencí :

```
template<class T>
class Ptr
{
    TObject* data;
public:
      Ptr(T* data):data(data)  {
            data->upcount();
      }

      Ptr(const Ptr& ptr) {
            data=ptr.data;
            data->upcount();
      }
      ~Ptr()      {
            data->downcount();
      }

      operator T*(void) {
            return (T*)data;
      }

      T* operator->(void)  {
            return (T*)data;
      }

      const T* operator->(void) const
      {
            return (T*)data;
      }

      Ptr& operator=(const Ptr& ptr) {
            return operator=(ptr.data);
      }

      Ptr& operator=(TObject* ptr) {
            data->downcount();
            data=ptr;
            data->upcount();
            return *this;
      }
};
```

Více o problematice implementace počítání referencí najdete v [7].

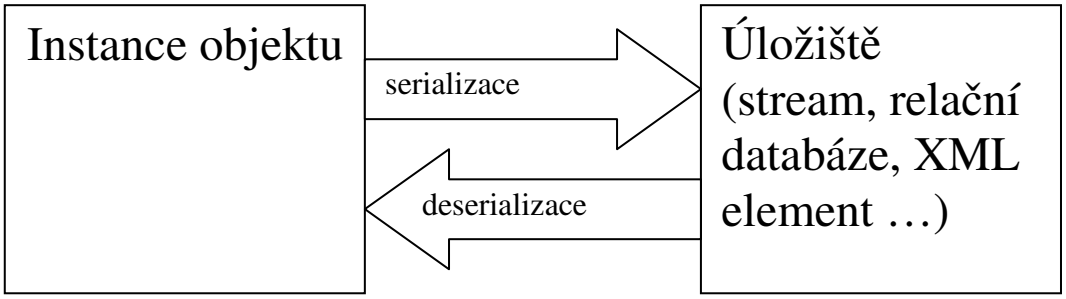
Další požadavky, které budeme mít na třídu *Ptr*, jsou:

- aby umožňovala porovnání s třídou *Ptr* vytvořenou dle stejného template (využije se metoda *compareTo(TObject*)* třídy *TObject*). Tím se umožní použití třídy *Ptr* ve standardních kontejnerových šablonách.
- možnost vytvořit instanci třídy *Ptr* s prázdným obsahem a ošetření nestandardních situací (dereference třídy *Ptr* s prázdným obsahem) příslušnými výjimkami.

7.1.3. Serializace a deserializace objektů

7.1.3.1. Úvod do problematiky

Mnoho instancí, které při svém programování používáme, mají celkem krátký „život“. Jsou nějakou cestou (konstruktorem, skrze reflexi, tovární metodou…) vytvořeny, poté jsou nějakým způsobem použity a pak když nejsou potřeba jsou odstraněny z paměti. Ovšem ne všechny instance jsou používány takto krátce a některé instance nesoucí potřebná data potřebujeme zachovat na delší dobu. To dokážeme zařídit tak, že onu instanci uložíme do nějakého perzistentního úložiště (relační databáze, soubor..). Takovémuto uložení instance do perzistentního úložiště se říká **serializace**. Opačnému procesu, tedy procesu pří kterém je z nějakého proudu vytvořena instance, se naopak říká **deserializace**.



Obr. 8 Serializace, deserializace objektu

Serializace bohužel nemá v C++ nativní podporu. Protože se ukázala jako jediná cesta k řešení problému s grafickým rozhraním, musela být navržena a implementována. Jako výstupní formát jsem zvolil soubor ve formátu XML. Ten je na rozdíl od binárního formátu náročnější na rozsah, ale je strukturovaný, dá se lehce zobrazit a zeditovat.

7.1.3.2. Implementace mechanizmu serializace a deserializace

Jako základ pro všechny serializovatelné objekty navrhneme třídu *TXmlObject* , která bude přímým potomkem třídy `TObject`. Bude muset obsahovat dvě virtuální metody, jednou bude sloužit pro uložení obsahu instance objektu do XML elementu, druhá pro načtení.

```
class TXmlObject:public TObject {

...

protected:
```

```
      virtual void doSerialize(TiXmlElement& element) const=0;

      virtual void doDeserialize(const TiXmlElement& element)=0;

...

};
```

S tím, co jsme zatím navrhl, můžeme být spokojeni jen částečně. První problém se týká serializace, která je tímto plně v rukou podtříd. Navíc by bylo jistě vhodné aby se při serializaci uchovával i název typu objektu. Toho docílím následující úpravou: přidáme veřejnou metodu, která bude volat serializaci na potomkovi.

```
public:

TiXmlElement& serialize() const
{
      TiXmlElement* element=new TiXmlElement(getType());
      doSerialize(*element);

      return *element;
}
```

Teď již můžeme být s návrhem serializace spokojeni, uchovává se nám informace o typu a zbytek je plně v rukou podtřídy.

Zůstává problém s deserializací. Když totiž máme nějaký XML element, jež vznikl serializací, nevíme na který typ třídy se má deserializace zavolat. Musíme tedy navrhnout nějaké „úložiště“, které nám přiřadí ke jménu typu třídu. Opět ale narážíme na fakt že třída v C++ není objekt. Musíme také vytvořit pro serializovaný XML element instanci té správné třídy.

Všechny tyto problémy můžeme elegantně vyřešit takto:

- pro každou podtřídu třídy `TXmlElement` navrhneme statickou metodu, která bude jeho „factory“ tj. po jejím zavolání se vytvoří instance dané podtřídy; factory musí mít stejnou formu, proto navrhneme jednotný typ; musí vracet typ `TXmlObject*` a nemůže mít žádné vstupní parametry, tedy :

  ```
  typedef TXmlObject* (*t_xml_factory)();
  ```

- navrhneme úložiště, kde bude uložena informace které třídě přísluší jaká „factory“

  ```
  private:
        typedef map<string,t_xml_factory> t_factory_map;
        static t_factory_map factoryMap;
  ```

- navrhneme metodu, která přidá jednu třídu do úložiště

  static void registerClass(const char* className,t_xml_factory method);

- navrhneme metodu, která toto úložiště naplní; toto je velkou nevýhodou,
  pro každou novou třídu kterou napíšeme musíme tuto metodu upravit a na její konec přidat registraci nové třídy

  ```
  static void registerAll();
  ```

- a konečně navrhneme metodu která nám pro libovolný XML element vrátí kýžený `TXmlObject`

  ```
  static TXmlObject* getXmlObject(const TiXmlElement& element);
  ```

  tato metoda se podívá do úložiště, zda existuje třída odpovídající názvu elementu (resp. faktory na vytváření instancí této třídy), vytvoří příslušnou instanci a na tuto instanci zavolá deserializaci

S tímto návrhem deserializace už můžeme být spokojeni, funguje tak jak jsme zamýšleli.

7.2. Implementace grafického API

7.2.1. Implementace systému událostí u widgetu

Tk poskytuje velké množství typů událostí. Připojení na obsluhu události probíhá pomocí příkazu interpretu bind. Jeho první parametr je unikátní identifikátor widgetu, druhý kód obsluhy události. V tomto kódu se před jeho vlastním provedením substituují parametry začínající znakem '%' za příslušné hodnoty události. Tyto parametry jsou jednak specifické pro každý typ události - pro stisknutí myši jsou to souřadnice bodu, pro stisk klávesy je to kód zmáčknuté klávesy a jednak obecné. Obecné parametry poskytují informace o události samotné.

Je to:
- sériové číslo události;
- identifikátor typu události;
- čas od spuštění aplikace v ms;
- unikátní identifikátor kořenového widgetu;
- unikátní identifikátor widgetu, který událost vyvolal;
- informaci zda šlo o uměle (programově) vyvolanou událost;

Základní typy událostí pro každý widget jsou:
- Activate,Deactivate – posílány každému widgetu když okno (tj. widget *toplevel*) změní stav;
- MouseWheel – aktivován točením kolečka u myši, pouze systém Windows;
- ButtonPress,ButtonRelease – aktivován stisknutím/uvolněním tlačítka myši;
- KeyPress,KeyRelease – aktivován stisknutím/uvolněním klávesy;
- Destroy – aktivován při odstranění ovládacího prvku;
- Motion – aktivován pohybem myši;
- FocusIn, FocusOut – aktivován když widget získá/ztratí focus;
- Enter, Leave – aktivován když kurzor vstoupí/vystoupí z plochy widgetu;

Základem připojení na událost z našeho rozhraní bude třída – *EBase*.
Bude obsahovat:
- unikátní identifikátor, tj. instanci třídy *TclGuid*;
- statické asociativní pole, které přiřazuje každému unikátnímu identifikátoru ukazatel na třídu *EBase*, položky tohoto asociativního pole se budou přidávat při volání konstruktoru třídy *EBase*, a naopak vyprazdňovat při volání destruktoru;

- odkaz na jednu instanci třídy *TkBase*, jejíž události bude obsluhovat;
- odkaz na seznam parametrů události;
- odkaz na jméno události, tak jak je známá shellu;
- statickou metodu *handleEvent*, která bude provádět obsluhu každé události, tato metoda bude vytvořena jako příkaz shellu;
- virtuální metodu *preprocessEvent,* která bude mít podobné parametry jako metoda *handleEvent*;
- metodu *createBind* která provede vlastní připojení;

Pro každý typ události vytvoříme dvě samostatné třídy. První bude třída realizující obsluhu, bude zděděná z třídy *EBase*. Bude obsahovat :

- statický seznam všech parametrů události a statický řetězec, který bude mít hodnotu stejnou jako jméno události v shellu; odkaz na tyto dva členy se bude předávat nadtřídě *EBase* při zavolání konstruktoru;
- seznam posluchačů;
- metodu pro přidání posluchače;

Druhá třídou bude interface představující posluchače daného typu události. Bude obsahovat jednu abstraktní virtuální metodu.

Kód posluchače události ButtonPressed:

```
class ButtonPressListener:public TObject
{
public:
      virtual void onButtonPress(EHeader& header,int button,const string&
state,int x,int y,int x_root,int y_root)=0;
};
```

Kód obslužné třídy události ButtonPressed:

```
class EButtonPress:public EBase {
      typedef vector<Ptr<ButtonPressListener> > t_listeners;
      t_listeners listeners;

      static const char* eventType;
      static const EField* eventFields[];
protected:
      int preprocessEvent(EHeader& header,Tcl_Interp* interp,int
objc,Tcl_Obj* CONST objv[]);
public:
      inline EButtonPress():EBase(eventType,eventFields) {}
      void addButtonPressListener(ButtonPressListener* listener)
      {
            listeners.push_back(listener);
            afterListenerRegistred();
      }
};
```

Připojení obsluhy události (metoda *createBind*) proběhne tak, že se v shellu spustí příkaz bind. Jeho parametry jsou uid widgetu, jméno události a příkaz provádějící obsluhu. Tímto příkazem je statická metoda *handleEvent* třídy *EBase*. Jeho parametrem je pak uid obslužné třídy a seznam všech přípustných parametrů daného typu události.

Pro událost ButtonPressed a widget s uid '.button' a obslužnou třídu s id 'id1' bude tento příkaz vypadat následovně:

```
bind .button <ButtonPressed> {handleEvent id1 %x %y …}
```

Scénář obsluhy události probíhá v krocích:

1. Dojde k události na ovládacím prvku. Například na widgetu '.button' došlo k události ButtonPressed na pozici [30,12] .
2. Shell zavolá obslužný příkaz.
   Pro náš příklad má tvar : `handleEvent id1 %x %y`
3. Shell provede náhradu parametrů v tomto příkazu konkrétními hodnotami.
   Po substituci bude mít příkaz tvar: `handleEvent id1 30 12`
4. Implementací shellového příkazu `handleEvent` je statická C procedura *handleEvent* třídy *EBase*
5. Tato procedura přečte identifikátor obslužné třídy, tedy 'id1', podívá se do asociativního pole po instanci třídy EBase s klíčem 'id1' a zavolá virtuální metodu *preprocessEvent*
6. Instance třídy *EBase* (konkrétně v našem příkladu je to instance třídy *EButtonPressed*) provede metodu *preprocessEvent*, ta zpracuje všechny parametry, které se jí předávají ve formě řetězců, projde seznam všech svých posluchačů a vyvolá na nich postupně metodu *onButtonPressed*.

Společné parametry událostí se zpracovávají už v metodě *handleEvent* a ukládají se do společné struktury *EHeader*. Reference na tuto strukturu se předává metodě *preprocessEvent*.

Třída *TkBase* pak bude potomkem všech typů obsluhujících tříd. Vazba se pro každý typ obslužné třídy zvlášť vytvoří při vytvoření widgetu reprezentovaného třídou *TkBase* v shellu. Všimněme si ještě, že tímto způsobem bychom v shellu pro každý widget vytvořili všechny typy obsluh. To by značně zpomalovalo grafické rozhraní. Vytvoření obsluhy potřebujeme až v momentě, kdy přidáme obsluhující třídě posluchače.

Čili na třídu *EBase* přidáme člen s pravdivostní hodnotou *anyListenerRegistred* a metodu *afterListenerRegistred.* Člen *anyListenerRegistred* bude mít výchozí hodnotu nepravda.
Metoda bude pracovat takto

1. pokud má *anyListenerRegistred* hodnotu nepravda, vyvolá se metoda *createBind*, která zaregistruje obsluhu události
2. hodnota *anyListenerRegistred* se nastaví na pravda

7.3. Implementace shellového rozhraní

7.3.1. Přímá evaluace v shellu

K přímé evaluaci v shellu nabízí knihovna Tcl lib tyto metody:

- ***Tcl_Eval***(*interp*,*scipt*) – *interp* je ukazatel na interpret, *script* je řetězec, který se má v interpretu provést; tato metoda vrací návratový kód vy formě čísla typu int.
  Význam návratového kódu :
  TCL_OK – vše proběhlo v pořádku
  TCL_ERROR – během provádění skriptu došlo k chybě

- ***Tcl_GetObjResult***(*interp*) – vrátí výsledek ve formě ukazatele na strukturu *Tcl_Obj*; pokud výsledkem *Tcl_Eval* je TCL_ERROR, tak je v této struktuře uloženo chybové hlášení

Struktura *Tcl_Obj* je základním úložným elementem. Obsahuje počítadlo referencí, ukazatel na obsah v podobě UTF-8 řetězce, informaci o délce tohoto řetězce a unii přes vnitřní reprezentace. Důležité je, aby programátor aplikace nikdy nepřistupoval k prvkům *Tcl_Obj* přímo, ale zprostředkovaně pomocí metod knihoven Tcl nebo Tk. Hodnota jednotlivých položek struktury totiž nemusí být aktuální a knihovní metody provedou nejdříve „regeneraci“ této struktury a poté příslušnou operaci.

```
typedef struct Tcl_Obj {
    int refCount;
    char *bytes;
    int length;
    Tcl_ObjType *typePtr;
    union {
        long longValue;
        double doubleValue;
        VOID *otherValuePtr;
        struct {
            VOID *ptr1;
            VOID *ptr2;
        } twoPtrValue;
    } internalRep;
} Tcl_Obj;
```

Protože přímá evaluace skriptů není nikde „vidět“, bylo by vhodné pro účely ladění a vývoje zaznamenávat skript který se v interpretu pouští a výsledek jeho evaluace. Proto vytvoříme třídu *TclInterp*, která bude reprezentovat instanci interpretu.
Bude obsahovat:

- metody pro nastavení logování buď do výstupního proudu (např. cout,cerr) nebo do souboru;
- metodu *eval* pro evaluaci vlastního skriptu;
- sadu metod pro vrácení výsledků, kromě vrácení výsledku ve formě struktury *Tcl_Obj* bude vracet výsledek ve formě řetězce, celého čísla, apod.

7.3.2. Proměnné shellu jako objekty

Už kvůli faktu že některé z nastavitelných vlastností widgetu jsou proměnné shellu, musíme vytvořit obalující třídu nad těmito proměnnými. I když z uživatelského hlediska jsou všechny proměnné Tcl téměř stejného typu – řetězec[2], (výjimkou jsou seznamy a pole), bude lépe pokud je rozdělíme do základních typů a ušetříme si tak opakované psaní konverzních rutin. Nadtřídou pro všechny třídy reprezentující jednotlivé typy proměnných bude třída *TclVar*. Společnou vlastností proměnných všech typů je unikátní pojmenování v shellu. Zde použijeme opět tak jako u třídy *TkBase* instance třídy *TclGuid*.

Změny v hodnotách proměnných shellu je také možné sledovat přímo z C, což je důležité pro vytvoření systému reakce na událostí. Tcl lib nabízí pro sledování změn, nastavování a odečítání hodnot proměnných mnoho metod.

[2] Vnitřní reprezentace je složitější, viz Tcl_Obj v 7.3.1

Pro naši implementaci jsou důležité zejména :

- *Tcl_ObjGetVar2* – vrátí hodnotu proměnné jako *Tcl_Obj*
- *Tcl_ObjSetVar2* – nastaví hodnotu proměnné, vstupem je struktrura *Tcl_Obj*
- *Tcl_TraceVar* – nastaví callback funkci, která se bude volat při každém přístupu k proměnné
- *Tcl_UntraceVar* – zruší trasování proměnné

Pokud je spouštěna callback funkce je trasování dočasně vypnuto, a tak se nemusíme starat o případnou nekonečnou smyčku, ke které by došlo kdybychom během provádění callback funkce změnili proměnnou.

Naše implementace trasování proměnných bude podobná implementaci systému událostí. Tedy třída *TclVar* bude obsahovat:

- unikátní identifikátor
- statické asociativní pole, které přiřadí unikátnímu identifikátoru ukazatel na instanci třídy *TclVar*, položky do tohoto pole se budou přidávat při volání konstruktoru třídy *TclVar*, a ubírat se budou při volání destruktoru této třídy
- statickou metodu *catchVar*, která bude přijímat callback volání po přístupu k proměnné; tato metoda ze jména tj. uid proměnné najde v asociativním poli ukazatel na instanci třídy *TclVar*, a zavolá metodu této instance *catchWrite*

Dále navrhneme interface *TclVarListener*, reprezentující posluchače změny proměnné. Bude obsahovat jednu abstraktní virtuální metodu, jejímž parametrem bude ukazatel na instanci třídy *TclVar*.

```
class TclVarListener:public TObject
{
public:
      virtual void valueUpdated(TclVar* variable)=0;
};
```

Třída *TclVar* bude mít seznam těchto posluchačů. Metoda *catchWrite* této třídy ho projde, na každou instanci třídy *TclVarListener* zavolá metodu *valueUpdated* a jako parametr jí předá ukazatel na svoji instanci.

Potomci třídy *TclVar* budou realizovat jednotlivé typy proměnných.

Typ proměnné v shellu	**C/C++ obdoba**	**Reprezentující třída**
pravdivostní hodnota	bool	*TclBoolVar*
celé číslo	int	*TclIntVar*
reálné číslo	double	*TclDoubleVar*
Řetězec	string	*TclStringVar*
Seznam	list<string>	*TclListVar*

tabulka 4 Třídy reprezentující proměnné shellu

Každá tato třída bude mít metodu pro zjištění obsahu *get* a metodu pro nastavení obsahu *set*. Výjimkou je třída reprezentující seznam. Ta bude obsahovat metody podobné standardní kontejnerové šabloně *list*. Metody *get* a *set* našich tříd budou volat metody *Tcl_ObjGetVar2* a *Tcl_ObjSetVar* z knihovny Tcl lib a provádět výstupní (metoda *get*) či vstupní (metoda *set*) konverze.

Viz též UML schéma problematiky v příloze UML diagramy, diagram 6 - Proměnné shellu jako objekty.

7.3.3. Vlastní vytvoření Tcl shellu

Abychom mohli Tcl shell vůbec použít, musíme nejdřív podniknout tyto kroky:

1. Předat řízení do rukou Tcl, to se provede tak, že se přímo ve funkci *main* zavolá funkce *Tk_Main*:

```
int main( int argc, char * argv[] )
{
   Tk_Main(argc, argv, Tcl_AppInit);
   return 0; // není podstatné ale předejde chybovým hlášením překladače
}
```

2. Vytvořit metodu (*Tcl_AppInit*) která provede vlastní inicializaci, tato metoda musí být typu *Tcl_AppInitProc*,tedy :

```
typedef int Tcl_AppInitProc(Tcl_Interp *interp);
```

V těle této metody provedeme:

- inicializaci shellu (*Tcl_Init*), inicializaci grafiky (*Tk_Init*), pokud z nějakých důvodů (špatná verze knihoven, neexistence grafického rozhraní apod.) nelze inicializaci provést, aplikace se ukončí;
- přidání vlastních příkazů do shellu. K tomu nabízí Tcl metodu *Tcl_CreateObjCommand*. Jejími parametry jsou jméno příkazu a C funkce předem daného typu. Tato funkce pak bude obsluhovat „příkaz“ shellu. Její parametry jsou velmi podobné funkci *main* tj. dostane pole argumentů a jeho délku. Její návratová hodnota (typu int) se předá zpátky shellu jako návratová hodnota příkazu (typicky předdefinované konstanty *TCL_ERROR*,*TCL_OK*). Tyto funkce vytvoříme jednu pro každý z našich navržených příkazů.

7.3.4. Příkazy pro modulární systém

Jednotlivé příkazy jsou implementovány jako statické funkce, které volá s příslušnými parametry shell. Těmto funkcím se předává seznam parametrů a ukazatel na instanci třídy *ModuleManager*.

7.3.4.1. Příkaz module

Syntax: **module** module_type jméno_instance. Funkce: vytvoření instance modulu.

Příkaz **module** přes instanci třídy *ModuleManager* vyhledá typ modulu se stejným jménem. Pokud tento typ modulu existuje, vytvoří pomocí „factory“ medoty *newModule* novou instanci modulu. Pak se pokusí zaregistrovat tuto metodu do systému (resp. do příslušné instance třídy *ModuleType*). Pokud instance se stejným jménem již v systému existuje, vytvoří se vyjímka kterou funkce realizující příkaz zachytí, vypíše do konzole a ukončí se.

7.3.4.2. Příkaz modlist

Syntax: **modlist**. Funkce: výpis typů modulů.

Příkaz **modlist** projde instanci třídy *ModuleManager* a do seznamu si poznamená jméno každého typu modulu, který je v ní obsažen. Tento seznam zobrazí na výstupu.

7.3.4.3. Příkaz vars

Syntax: **vars** jméno_instance. Funkce: vypíše proměnné instance modulu.

Příkaz **vars** vyhledá zda v systému existuje instance se jménem jméno_instance. *ModuleManager* projde každý v něm obsažený typ modulu. Každý typ modulu projde všechny své zaregistrované instance. Protože je jméno unikátní v momentě, kdy se najde instance s jménem, hledání končí. Pokud se instance najde, zavolá se metoda třídy *IModule* pro výpis proměnných, ta projde všechny obsažené instance proměnné a odkazy na jejich definice a sestaví z nich výstup.

7.3.4.4. Příkaz sets

Syntax: **sets** jméno_instance jméno_proměnné hodnota.. Funkce: nastaví hodnotu shellové proměnné.

Vyhledání instance probíhá stejně jako u příkazu **vars**. Dále se nastaví na nalezené instance modulu instance proměnné řetězcem (metoda *fromString*). Pokud tento řetězec nebyl platný pro daný typ proměnných vrátí metoda chybovou hodnotu, která se zobrazí. V opačném případě je ještě potřeba upozornit modulární systém na změnu proměnné. To se provede pomocí broadcastu zprávy „proměnná byla změněna“ – instancí třídy *MesgVarChanged*.

7.3.4.5. Příkaz run

Syntax: **run** jméno_instance. Funkce: spustí instanci modulu.

Vyhledání instance probíhá stejně jako u příkazu **vars**. Dále se nalezenou instanci modulu (tj. instanci třídy *IModule*) zavolá metoda *run*.

7.3.4.6. Příkaz on

Syntax: **on** jméno_instance do arg1 arg2 argN. Funkce: spustí příkaz specifický pro daný typ modulu.

Vyhledání instance probíhá stejně jako u příkazu **vars**. Dále vyhledá typ modulu (tj. instanci třídy *ModuleType*) pod kterou náleží nalezená instance. Pak projde seznam příkazů třídy *ModuleType*. Pokud nenajde příkaz jménem „do“ (viz seznam argumentů) vykonávání příkazu **on** skončí. V opačném případě se zkontroluje arita příkazu (zda počet argumentů odpovídá definici) a pokud arita vyhovuje zabalí se celý specifický příkaz (tj. „do“) a jeho argumenty to třídy *MesgCommand* a ta se pak pošle příslušné instanci modulu jako zpráva.

7.3.4.7. Příkaz helps

Syntax: **helps** [jméno_příkazu]. Funkce: poskytuje nápovědu na naše nové příkazy.

Nápověda je vytvořena tak, aby nebyla přímo v kódu. Měla by se dát jednoduše upravit, přeložit apod. Podívejme se na strukturu nápovědy k jednomu příkazu.

Položky které musí obsahovat jsou:

- jméno příkazu;
- syntax jeho správného použití;
- popis co příkaz dělá;
- zkrácený popis toho, co příkaz dělá;
- seznam nápovědy k argumentům;

Nápověda k jednomu argumentu musí obsahovat:

- jeho jméno;
- nápovědu k tomuto algoritmu;

Kompletní návrh tříd pro nápovědu k našim příkazům shellu je v příloze s UML diagramy, diagram 1 - Třídy pro nápovědu k příkazům.

Také budeme potřebovat ukládat tyto položky ve vhodném strukturovaném formátu do souboru tak, aby šel editovat, použít různé typy znakových sad apod. S výhodou použijeme formátu XML. Tento soubor musí mít také pevně určenou a ověřitelnou strukturu, protože ho dáváme k „volné editaci“ uživateli. K tomu se používá notace DTD.

Pro náš případ bude vypadat takto:

```
<!ELEMENT commands (command*)>
<!ELEMENT command (arg*)>
<!ATTLIST command name  CDATA #REQUIRED
                  usage CDATA #REQUIRED
                  desc  CDATA #REQUIRED
                  shortDesc CDATA #REQUIRED
>
<!ATTLIST arg     name  CDATA #REQUIRED
                  desc  CDATA #REQUIRED
>
```

O formátu XML,DTD se můžete více dozvědět na [4].

8. Testování

8.1. Portabilita

Náš modulární systém je zadán jako multiplatformní, tj. měl by jít spustit na více platformách. Původním vývojovým prostředím byl systém Windows a překladač Visual Studio .NET. Pro otestování na Linuxovém systému byl vybrán systém Debian s kernelem verze 2.6.8 a kompilérem gcc verze 3.3.5.

Při portování kódu se objevily následující problémy:

- „not“ je předdefinovaný symbol a tudíž jeho použití jako argumentů ve funkci vedlo k chybě překladače. Tento problém byl odstraněn přejmenováním příslušných argumentů.
- funkce pow a div mají argumenty int *pow*(int,int), long *pow*(long,long) atd. Tyto funkce (použity např. v kódu algoritmu BOOM) měli argumenty typu *pow*(long, int) atd. což překladače pod Windows (jmenovitě Visual Studio .NET, Borlandu C++ Builder) tolerovaly, ale gcc to striktně ohlásilo jako chybu. Tento problém byl odstraněn explicitním přetypováním vnitřních parametrů na rozšiřující typ.
- konce řádků na konci souborů, gcc vyžaduje aby soubor končil prázdnou řádkou, jinak zahlásí varování. Tento problém byl odstraněn přidáním prázdné řádky na každý konec souboru.
- jiný přístup k výčtovým typům, pokud v třídě *MyClass* definuji výčtový typ *MyEnumeration* a v ní mám např. položku *Item* tolerujími překladače pod windows přístup stylem *MyClass::MyEnumeration::Item*, zatímco gcc vyžaduje *MyClass::Item* (tato notace je přípustná i pro Windows překladače). Tento problém byl odstraněn sjednocením přístupu k výčtovým typům tak, aby vyhovoval notaci gcc.
- knihovny Tcl/Tk se pod windows distribuují v jednom velkém balíku, pod linuxem je možnost stáhnout každý balík zvlášť. Při tomto přístupu se může stát, že chybí částečně některá součást (konkrétně se jedná o knihovnu tkimg) kterou aplikace potřebuje, ale rozběhne se i bez její existence;
- změněn přístup k souborům, zatímco windows označuje adresáře znakem „\” u linuxu je to “/”. Tento problém byl vyřešen použitím maker pro rozlišení cílového systému (např. unixové systémy mají standartně předdefinován symbol __unix__).
- grafické elementy mají jinou velikost. To není záležitost jen portability, ale spíše uživatelského nastavení toho kterého systému. Jedná se zejména o jiné velikosti fontů. Je důležité vzít tuto skutečnost v potaz při navrhování formulářů (ponechat dostatek místa).

8.2. Usability Test

Tato část nebyla využita jen pro potřeby této diplomové práce, a proto je psána v anglickém jazyce.

8.2.1. Executive Summary

Usability study showed that users generally were missing a proper feedback and a more detailed help. All users were able to finish their tasks.

The most visible problems included:

- Participants were not able to determine if something actually happened after a successful execution of a shell command.
- Participants were not able to get the shell command they wanted fast enough.
- Participants were depressed by non-existing drag & drop mechanism.

8.2.2. Introduction

The usability test's first ambition was to examine how well a user can execute a simple algorithm using the shell interface. The test should prove that a user can execute this algorithm only by using shell help.

The second ambition was to examine designing a form using the form designer. The test should prove that this designer can be controlled intuitively.

Two participants were selected and asked to perform three tasks. All usability notes were noted down and analyzed in the following sections.

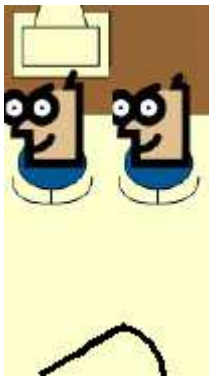
8.2.3. Participants

Part.	Sex	Experience with any shell	Programming experience	Form designer experience
P1	M	Yes	Average	Average
P2	F	No	None	Low

tabulka 5 Participants

8.2.4. Test setup description

Usability tests were performed in one workroom. Very simple schema is bellow.



Obr. 9 Test setup

8.2.5. Application setup

The first two tasks were aimed on shell interface. So for this first task text console was maximalized. To track the console, logger was installed.
As the last task the form designer was executed with no forms opened.

8.2.6. Participants warm-up

Participants were informed about basic facts on the module system, and how to get the help. Participants were asked to play with console and try entering some commands.

8.2.7. Task summary

Three complex tasks were focused on the following activities:

1. Execution of a simple "sum two numbers" algorithm
2. Execution of a simple sorting algorithm
3. Designing a form for algorithm from task 1.

8.2.8. Task analysis

8.2.8.1. Task 1 - Execution of simple "sum two numbers" algorithm

Objective
Using implemented "add" algorithm for summing two numbers calculate "3 + 7"

Expected steps
Participant will create algorithm "add" instance.
Participant will examine algorithm parameters.
Participant will set algorithm instance parameters to "3" and "7"
Participant will execute algorithm.
Participant will get result.

Expected results
Participants will experience difficulties with exact command syntax.
All participants will be able to finish the task.

Results
All participants have finished the task.
Both participants had experienced difficulties and not only with the command syntax.

Notes

Command helps shows available command list or help on a particular command. Participants are excepting help to include short info about commands.

Participants are missing feedback. Commands "run" for running algorithm, "sets" for setting variables if they are successfully executed produces no text output. Participants complains, that they are excepting something like "running algorithm 1" or "setting variable a to 2" etc.

P1 is missing a command for getting module-specific command list (list of module-specific command list is displayed only after writing an incorrect module-specific command). P1 is also requiring some help for this module-specific commands. Both participants are trying to get help on module type. Both participants have intuitevly written "helps add" trying to get help to summing algorithm, which didn't work.

Help is not specific in the matter, that instance names should be unique. P1 is trying to create two algorithm instances of the same name.

P1 uses “default” command that speeds up writing, P2 fails to get its meaning.

P1 missed command “vars” in help list. It is because there is no newline at the end of the help list and “vars” is the last command in the help list. So he didn’t get it because his eyes skipped this row.

P2 didn’t understand that arguments enclosed in [] bracers are not required.

P2 is confused with names “module type”,“module instance”. She prefers names more real, like “algorithm”. She is also confused by difference between “module type” and “module instance”. She is repeating using module type instead of module instance – for example she tried to execute “run add”.

P2 is complaining about feedback after typing command with wrong syntax. System default behavior after command with wrong syntax is to write down the right syntax. She would like shell to write down something like “You typed this wrong!” and after that write the right syntax.

P2 is complaining about shell at all. She said things like “How can people use computers without graphic” and other, mostly rude words about console.

8.2.8.2. Task 2 - Execution of simple sorting algorithm

Objective
Using implemented “compare” algorithm for sorting three numbers participants will sort numbers “3”,”5”,7”

Expected steps
Participant will create a compare algorithm instance.
Participant will examine algorithm parameters.
Participant will set algorithm instance parameters to “3”, “5” and “7”
Participant will execute the algorithm.
Participant will get the result.

Expected results
This task is similar to Task 2. Users will finish it quickly without much help on commands.
All participants will be able to finish the task.

Results
P1 has fulfilled the expectations.
P2 again encountered problems with syntax. She rolled the console back to see how she typed the commands right.

Notes
P1 sets algorithm variables as if they’re required to be real numbers. He types “3.0”, “5.0”, “7.0” instead of simple “3”,”5”,”7”.

P2 tries to type variables like formulas. She tries “set instance a=2” instead of correct “set instance a 2”.

8.2.8.3. Task 3 - Designing a form for algorithm from Task 1

Objective
Use form designer to design a form to enter variable values and report the result of the algorithm from Task 1.

Expected steps
Participant will create a new window.
Participant will place three text-entries into the window.
Participant will create a label for naming the entries.
Participant will create a binding between this entries and module variables.

Expected results
Participant will have difficulties with creating the binding.
All participants will be able to finish the task.

Results
P1 created crude form but without the binding.
P2 managed to create form but then she placed randomly basic graphic elements without real meaning.

Notes
Both participants are really missing drag & drop style to place graphic elements.

P2 is missing tooltip on graphic elements. She also doesn’t know what every single graphic does.

P1 doesn’t like style how graphic elements are deleted from the form. The system does it after clicking the “delete” button. He would like “backspace” or “delete” key to do this job. He quotes it with “Clicking on delete button is evil, mark my words”.

P1 complains about no undo mechanism.

P1 doesn’t like label icon.

P1 doesn’t like the way the graphic elements are selected (filled by red color) he would like something more decent. P2 doesn’t get the meaning of the red elements – “Why is this red? It is little bit shitted up”.

Participants are excepting after pressing “enter” key in properties editor the property to change not after it lose focus.

Participants would like after placing graphic element to change element selection tool to “arrow”.

Participants are excepting change of property value after pressing “enter”, not after property editor lose focus.

8.2.9. Problem list and solution suggestion

Command help
The help command displays only available command list. Participants are excepting the help to include a short info about commands. They also missed the last command in help overview because help dump was not separated by newline. Some participants also require help on help. Another note was that help doesn’t directly state that module instance name must be unique.

Include short info about what command do to help list. Help list should be ending with a newline. Try considering some help on help and some overview over the modular system in general.

Shell command feedback
Participants are complaining about improper feedback. They expect some feedback on each successful command execution and on each syntax error.

Write down something like “command successfully executed” after each successful command execution and write down not only correct syntax after each syntax error, but also some message like “syntax error”.

Module-specific command
Participants are missing help on module specific commands.

Give some form of help either on module specific commands or on module types in general.

Confusion between “module type” and “module instance” terms
Participants are confused about where to use module type and where to use module instance. They also try to use module type in commands such as “sets” or “vars” instead of module instance.

State the meaning more clearly in help. Try to revise name of some commands. Also consider reserving module type names so they can be used to access module instance if module type has only one instance.

Setting variables
Users are trying to set variables in formula style. “sets instance a=2“ instead of “set instance a 2”.

Consider implementation of both methods of setting variables.

Drag&drop
Users are excepting that placing graphic elements on the form should be in drag&drop style.

Consider implementation of this mechanism.

Tooltips
There is only image information about what specific graphic element does.

Consider adding tooltips to all icons.

Element deletion
Participants are excepting that “delete” key would delete the selected graphic element.

Consider adding “delete” key as trigger of deleting graphic element.

Element selection
Currently, selected element is filled with red color. Users find this distracting.

Consider some more decent way to state that element has been selected.

Property editor
Users are excepting that after pressing “enter” key the property will change its value.

Consider implementing “enter” key press as trigger to changing property value.

Undo mechanism
Users are complaining about no way to undo action.

Consider implementing some undo mechanism.

Graphic element placing
After placing a graphic element in the form, the tool which selects what graphic element will be added to the form next doesn’t change. So if user clicks again into the form, new graphic element of the same type would be added.

Consider changing tool to “selecting arrow” after graphic element has been placed.

8.2.10. Suggestions for further usability test

- Higher number of participants.
- Participants experienced with Tcl
- Participants experienced with some designer

9. Závěr

Vytyčené cíle – tj. implementace modulárního systému, shellového rozhraní, grafického API i designéru se podařilo splnit. Uživatelská studie naznačuje určité změny které by bylo vhodné provést. Kromě malých úprav shellového rozhraní, rozšíření nápovědy a posílení zpětné vazby shellových příkazů, jsou to zejména změny v návrháři. Uživatelé zejména postrádali implementaci drag&drop mechanizmu pro umísťování grafických prvků a umísťování tooltipů. Bohužel jak události drag&drop, tak tooltipy nejsou základním balíčkem, nad kterým je návrhář postaven podporovány. Proto by se mohl další vývoj orientovat směrem na rozšíření vytvořeného API tak, aby obsahovalo podporu pro tooltipy a drag&drop mechanizmus. Také by se stálo zamyslet nad použitelností systému pro méně zkušené uživatele, popř. uživatele nemající vůbec žádné zkušenosti se shellovým rozhraním. Řešením by byla např. možnost nějakého zjednodušeného režimu, ve kterém by uživatel pracoval jen s jedním typem algoritmu nebo jen s jednou instancí algoritmu.

Krom toho je možno provést značné změny s celým modulárním systémem. Je to zejména spouštění více instancí algoritmů najednou buď pomocí threadů nebo pomocí procesů a ošetření stavů s tím spojených, oddělení vlastního modulárního systému od uživatelského rozhraní tak aby šel spouštět jako klient/server popř. ještě lépe, aby šel modulární systém distribuovat na více počítačů a automaticky podle vytížení jednotlivých počítačů rozděloval zátěž. Dalším mechanizmem, který nebyl implementován je mazání vytvořených instancí.

Grafické API tak jak je vytvořeno poskytuje jen základní widgety. Widgety z rozšiřujících balíčků (BWidgets, Iwidgets) jsou mnohem pokročilejší. Poskytují např. přímou podporu pro zmiňovaný drag&drop mechanizmus, přímou podporu pro tooltipy, atd. Tyto balíčky jsou však velice rozsáhlé a jejich celková integrace by byla implementačně náročná. Řešení by bylo buď zamyslet se nad tím, které widgety jsou nejpotřebnější a doplnit je nebo postupně rozšiřovat grafické API přímo programátory kteří budou potřebovat nové ovládací prvky.

10. Seznam literatury

[1] FIŠER P. : *BOOM – The Boolean Minimizer* [online stránky projektu]. Last updated 24 November 2005 [cit. 25.1.2006]. Dostupné na <http://service.felk.cvut.cz/vlsi/prj/BOOM/>.

[2] *GTK* [online stránky projektu]. Last updated 16 January 2006 [cit. 25.1.2006]. Dostupné na <http://www.gtk.org/>.

[3] *Tcl Developer Xchange* [online stránky podpory]. ActiveState, [cit. 25.1.2006]. Dostupné na <http://www.tcl.tk>.

[4] QUIN L.: *Extensible Markup Language (XML)* [online stránky vývojové skupiny]. Konsorcium W3C, November 2005 [cit. 25.1.2006]. Dostupné na <http://www.w3.org/XML/>.

[5] *Objektově orientované programování.Wikipedie* [online encyklopedie]. [cit. 25.1.2006]. Dostupné na <http://cs.wikipedia.org/wiki/Objektově_orientované_programování>.

[6] *Garbage Colletion. Wikipedie* [online encyklopedie]. [cit. 25.1. 2006]. Dostuptupné na <http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)>.

[7] *Templates and smart pointers* [online stránky podpory]. Microsoft [cit. 25.1. 2006]. Dostupné na <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/_core_templates_and_smart_pointers.asp>

[8] Aktive Tcl – Online Docs, verze 8.4.12.0 [online manuál]. ActiveState, December 2005 [cit. 25.1.2006]. Dostupné na <http://aspn.activestate.com/ASPN/docs/ActiveTcl>.

[9] WELCH B. B., JONES K., HOBBS J. : *Practical Programming in Tcl and Tk*, fourth edition. Upper Saddle River, Prentice Hall PTR, 2003.

A. Seznam použitých zkratek

API – Application Programming Interface
BOOM – Boolean Minimalizer
EDA - Electronic Design Automation
GUI – Graphic User Interface
Tcl - Tool Command Langugage
Tk – (GUI) Toolkit
XML – Extended Markup Language

B. UML diagramy

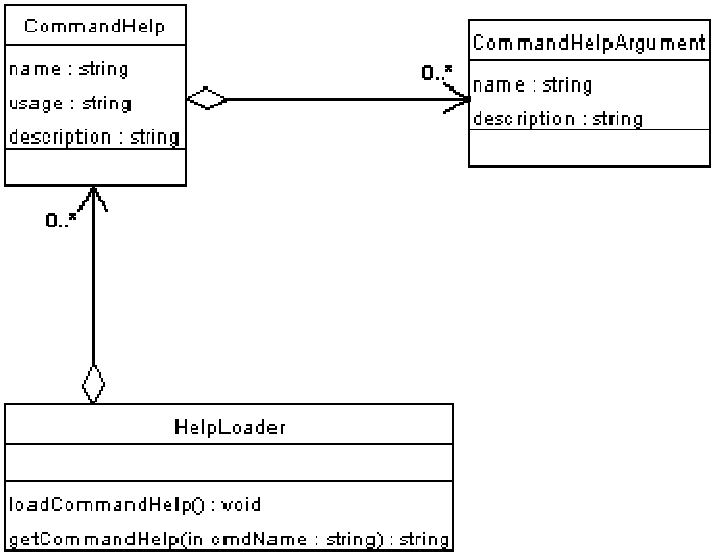


diagram 1 - Třídy pro nápovědu k příkazům

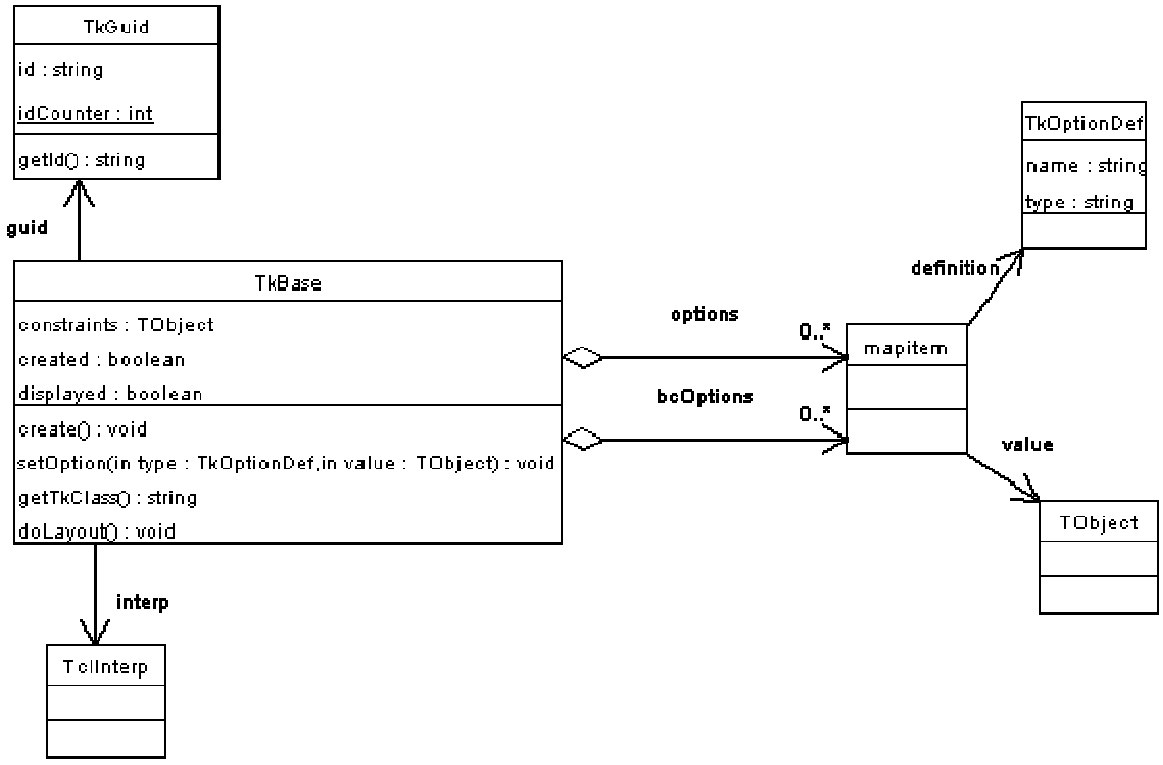


diagram 2 - Třída TkBase

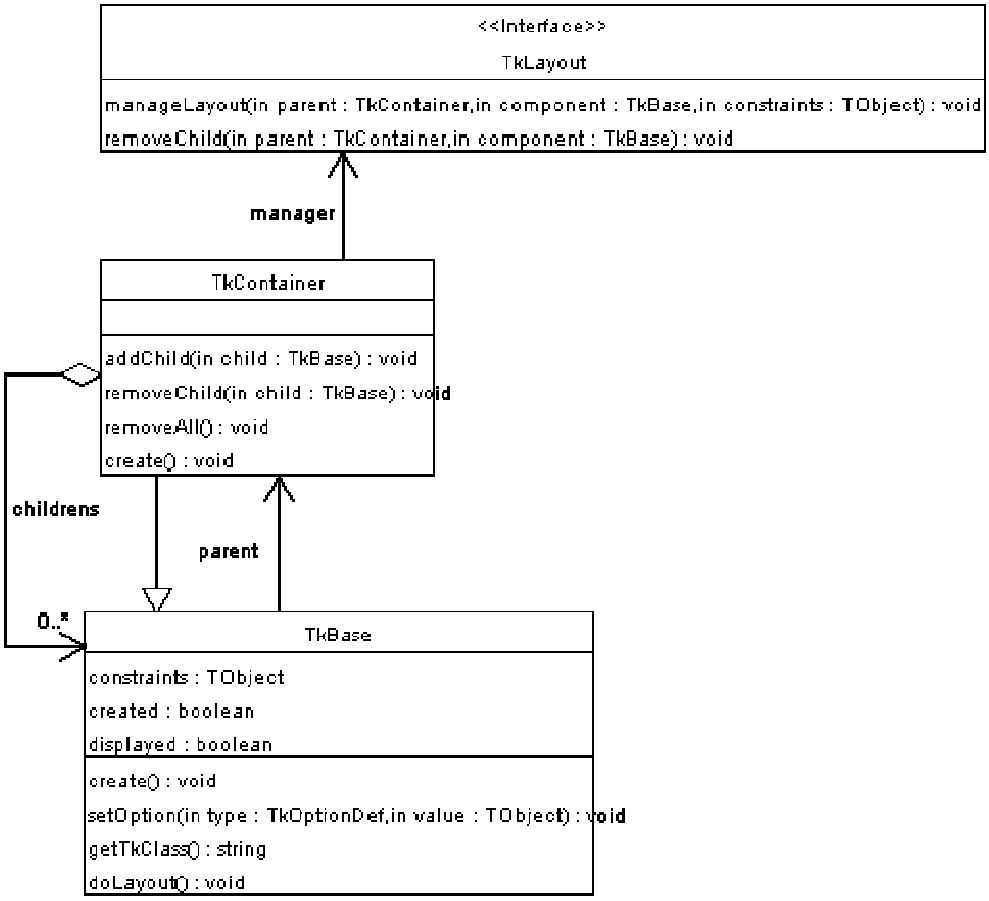


diagram 3 – Třída TkContainer

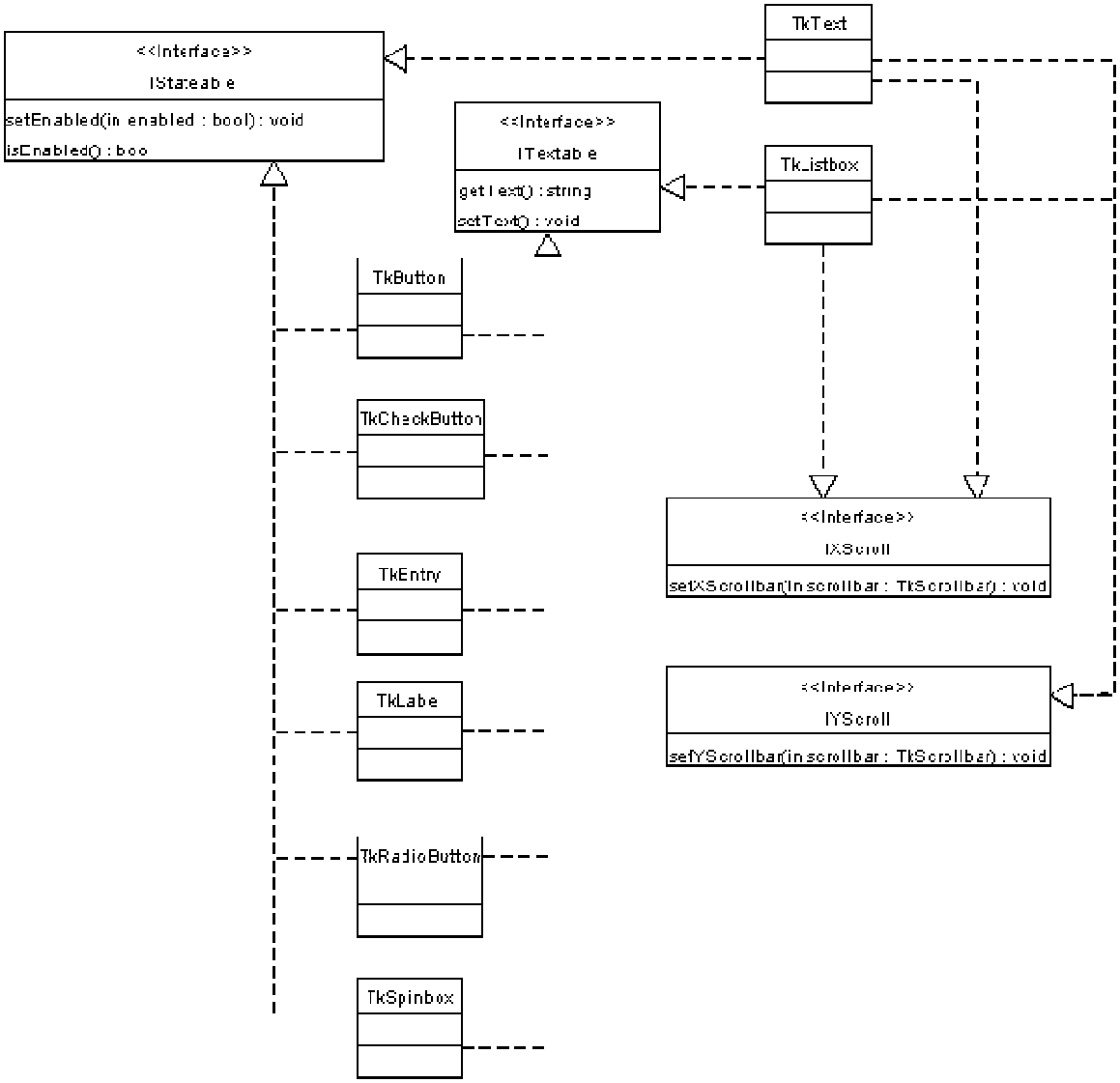


diagram 4 - Základní typy widgetů a základní interface

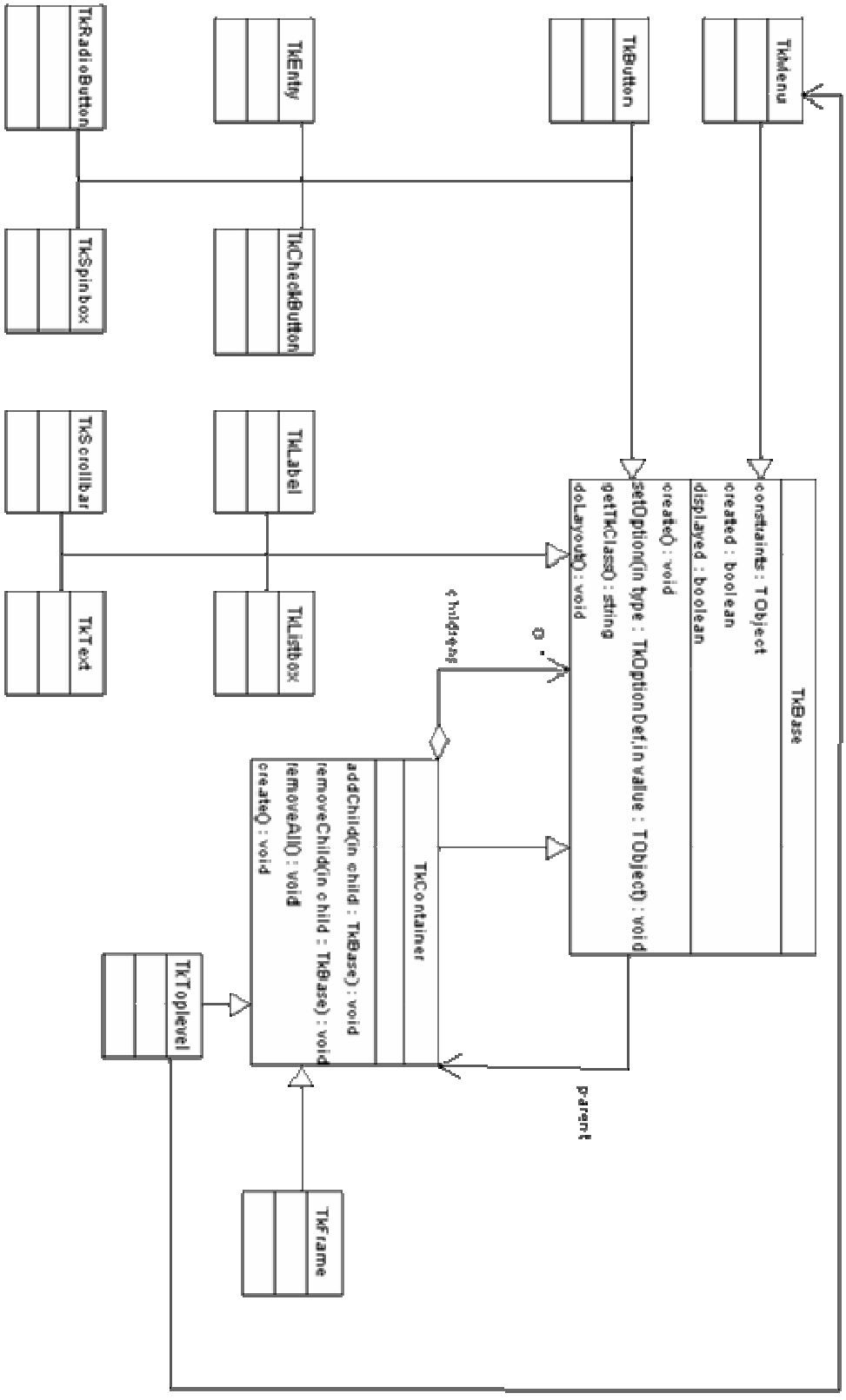


diagram 5 - Přehled základních typů widgetů

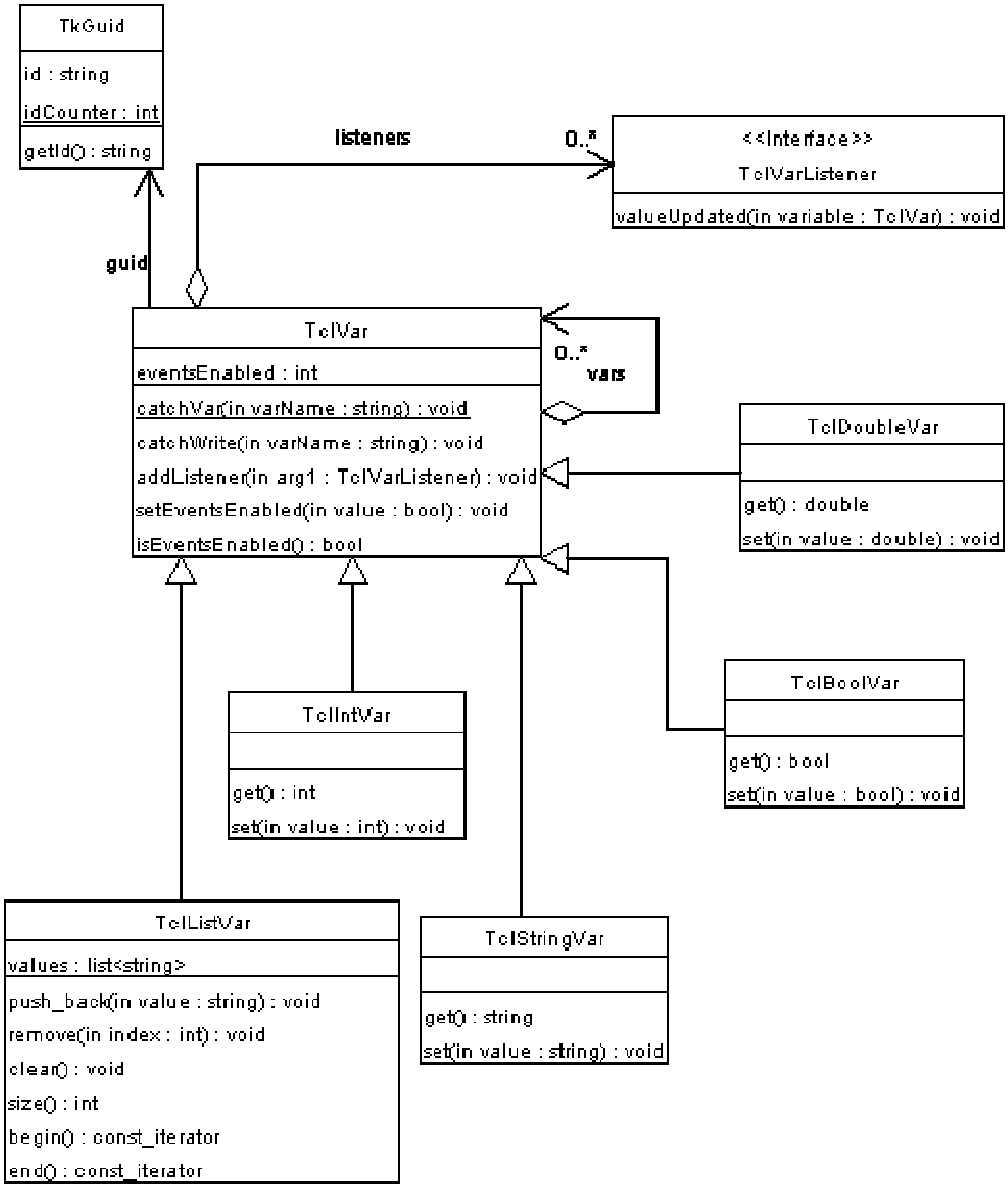


diagram 6 - Proměnné shellu jako objekty

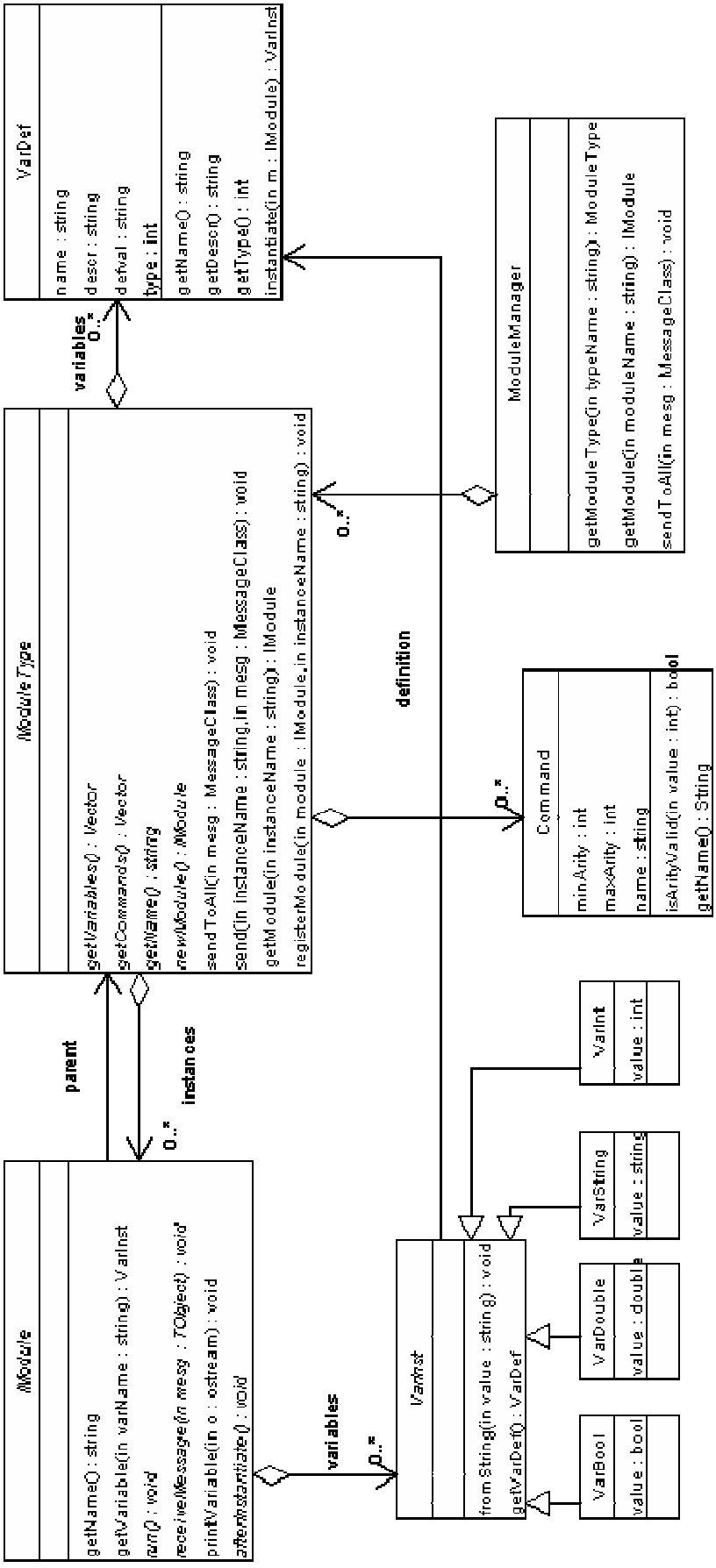


diagram 7 - Modulární systém

C. Programmer's handbook

1. How to write new class

All classes in our system should be subclasses of class *TObject* or of one of its child. Class *TObject* provides general object management methods. We should at least override its method *getType*() which is used for type control, exactly like this:

```
class MyClass:public TObject {
public:
  static const char* type;
  const char* getType() const {
    return type;
  }
};

static const char* MyClass::type="MyClass";
```

For type control is used pointer comparison, so we must have public static pointer (type in our example). This pointer should hold string exactly matching to the name of class, so there won't be any confusion.

Other methods, which you can override:

- `virtual bool hasParent(const char* className) const` – check if object has parent class named className
- `virtual bool compareTo(TObject* object) const` – should be true, if "this<object" in desired meaning. Default implementation throws runtime exception "method not implemented". You must override this method if you mean to use *Ptr<YourClass>* in STL container classes.
- `virtual const string& toString() const` – simple text dump. Default implementation returns name of class.
- `virtual TObject* clone()` – creates and returns deep-copy of object

2. How to use Ptr class

Ptr class is used as smart pointer, which encapsulates *TObject* class and its subclasses. It manages reference counting and provide access to *TObject* (or its subclass) methods.
Special methods are:

- `TObject* asObject()` – return encapsulated class as *TObject*
- bool isNull() const – true if inner pointer to *TObject* is NULL

Suppose we have class *MyClass*:

```
class MyClass:public TObject {
public:
  int foo;
  void bar();
}

...

Ptr<MyClass> i;
cout<<i.isNull()<<endl; // true
i->foo=8; // will throw runtime exception ExceptionNullPtr
i=new MyClass(); // reference count = 1
```

```
cout<<i.isNull()<<endl; // false
i->foo=7; // OK
{
  Ptr<MyClass> a=i; // correct expression, a and i are equal, ref. count=2
  cout<<a->foo<<endl; // 7
} // a is disposed => reference counter is decreased by one
i->bar(); // one method to call bar
((MyClass*)i)->bar(); // another method to call bar
((MyClass*)i.asObject())->bar(); // another method to call bar (compare)
i=new MyClass(); // correct, previous value will be disposed
```

This *Ptr* mechanism is implemented in all classes for creating GUI, so it is perfectly correct to make expressions like:

```
 Ptr<TkFrame> frame=new TkFrame();
frame->setLayoutManager(new GridLayout());
```

Pointer to manager is stored in *Ptr* reference and so if any layout manager was set, it would be automatically disposed and this new *GridLayout* would be disposed as well when reference counter of frame reach zero.

Be careful to avoid reference cycles. Reference cycles are disadvantage of reference counting, because they allocate memory, which is never disposed.

Example:

```
Ptr<MyClass> a=new MyClass();
a=a; // AVOID THIS!

Ptr<MyClass> a=new MyClass(),b=new MyClass();
a=b;
b=a; // AVOID THIS!
```

3. How to add new module type

1. We must create a new subclass of class *IModule* (definition in file "modulemanager/imodule.h"). This class represents module instance. We should override following methods:

   - `virtual void run()` – method serving as "main start" for the module instance, this method is pure virtual so we have to implement it
   - `virtual void receiveMessage(TObject* mesg)` – method for message receiving, all events are directed to this method
   - `void afterInstantiate()` – method is called after module instance is registred into system, typical usage is creating graphic

   Simply example of our own module instance is module for summing two numbers *TestAdd* (complete example can be found at directory testsuite, files testadd.h and testadd.cpp) :

   ```
   class TestAdd:public IModule {
     bool wasCounted; // if true, we have valid result
     int c;
   public:
     void afterInstantiate();
   ```

```
  void receiveMessage(TObject* mesg);
  void run();
};

void TestAdd::afterInstantiate() {
  wasCounted=false;
  c=0;
}

void TestAdd::receiveMessage(TObject* mesg) {
  if (mesg->typeOf(MesgCommand::type)) {
    MesgCommand* command=(MesgCommand*)mesg;
    if (wasCounted) cout<<c<<endl;
    else
      cout<<"UNDEFINED"<<endl;
  }
  if (mesg->typeOf(MesgVarChange::type))
    wasCounted=false; // variable has been changed => invalidate
}

void TestAdd::run() {
  int a=toInt(getVariable("a"));
  int b=toInt(getVariable("b"));

  c=a+b;
  wasCounted=true; // result is valid
}
```

2. Next we must create new subclass of the class *ModuleType* (definition in file "modulemanager/moduletype.h"). This class serves as factory for *IModule* and holds list of all *IModule* instances with same type. Class *ModuleType* contains information about what variables will use module, which commands can be called on it and what is the module name. We must override following methods :

   - `const string& getName() const` – method returns name of module, this name should be unique, should not contain any special characters (especially spaces) because we will use this name in shell, for the matter of uniformity is recommended to use lowercase letters only
   - `Ptr<Command>* getCommands() const` – method returns array of commands, valid for specific module. If method return value is NULL, system treats module as there are no specific module commands
   - `Ptr<VarDef>* getVariables() const` – method returns array of variable definitions of module, or NULL in case that module hasn't got any variable definitions
   - `IModule* newModule() const` – method serves as factory for creating new module instances

   Simple example for our TestAdd module:

```
class TestAddLoader:public ModuleType {
  static Ptr<VarDef> variables[];
  static Ptr<Command> commands[];

  static string name;
public:
```

```
  const string& getName() const {
    return name;
  }

  Ptr<Command>* getCommands() const {
    return commands;
  }

  Ptr<VarDef>* getVariables() const {
    return variables;
  }

  IModule* newModule() const {
    return new TestAdd();
  }
};


string TestAddLoader::name="add";

Ptr<VarDef> TestAddLoader::variables[]= {
  new VarDef("a","Defines first  number",VarDef::type::VAR_INT,"0"),
  new VarDef("b","Defines second number",VarDef::type::VAR_INT,"0"),

  0 // NULL pointer is used for signal end-of-array, always include
};

Command TestAddLoader::commands[]= {
  new Command("get","Returns result"),

  0 // NULL pointer is used for signal end-of-array, always include
};
```

VarDef is class, defining the variable. Constructor has following form:

```
VarDef(const char* name,const char* descr,const type vd_type,const
char* defval)
```

Parameters are:

- name – name of variable
- descr – description of variable (for user)
- vd_type – variable type (one of VAR_INT, VAR_DOUBLE, VAR_STRING, VAR_BOOL, that is one value from enumeration VarDef::type)
- defval – string containing default value (this parameter can be empty)

Command is class, defining the command, which can module accept. Constructor has following form:

```
Command(const string& name,int minArity,int maxArity)
```

Parameters are:

- name – name of command
- minArity – minimal number of parameters for this command
- maxArity – maximal number of parameters for this command (if maxArity is not defined, it's value is set to be equal to minArity)

minArity and maxArity parameters are not required. If they are both undefined, they are set to zero.

3. We must pass this module type to the module manager. This is done in file "main.cpp", in array types:

```
ModuleType* types[]=
{
      new BoomLoader(),
      new TestAddLoader(), // our newly added module type
      new TestCompareLoader(),

      0
};
```

4. How to add new widget type

1. We must decide if this new widget is container or base widget. If this widget is container, we will create it as subclass of *TkContainer,* if not, we will create it as subclass of *TkBase.* We must override following methods:
   - const Ptr<TkOptionDef>* getOptionAll() const – return list of all options defined for this widget
   - const char* getTkClass() const – return name of widget type in Tk

```
class MyWidget:public TkBase {
  const static Ptr<TkOptionDef> OPT_TEXT;
  const static Ptr<TkOptionDef> OPT_2;
  const static Ptr<TkOptionDef> OPT_3;

  const static Ptr<TkOptionDef> optAll[];
  const static char* tkClass;

  const Ptr<TkOptionDef>* getOptionAll() const {
    return optAll;
  }
  const char* getTkClass() const {
    return tkClass;
  }
};

const char* MyWidget::tkClass="labelbutton";
```

2. Class *TkOptionDef* is used for defining widget options. Widget should have all available options defined as public static members, this options are used for setting

properties of any Tk widget. We should make methods for setting/getting options, so application programmer can avoid them.

TkOptionDef has following constructor:

```
TkOptionDef(const string& name,const char* optionType,bool quote,bool
simpleOption);
```

Parameters are:

- name – name of option (from Tk)
- optionType – pointer to class, which define this option
- quote – option will be enclosed in quotes (default true)
- simpleOption – option is simple that is it can be represented as string, non-simple options are for example shell variables, references to other widgets and so on

For example, we will make methods for setting/getting option OPT_TEXT, which is used for setting/getting text of label in label part of labelbutton.

```
const string& getLabel() const {
  return getStrOption(MyWidget::OPT_TEXT);
}

void setLabel(const string& label) {
  setOption(MyWidget::OPT_TEXT,new TkStringOption(label));
}
```

5. How to add new layout manager

New layout manager should be subclass of *TkLayout*:

```
class TkLayout:public TXmlObject {
public:
      virtual void manageLayout(TclInterp* interp, TkContainer* parent,
TkBase* component, TObject* constraints) const;
      virtual void removeComponent(TclInterp* interp, TkContainer* parent,
TkBase* component) const;
};
```

Method *manageLayout* is called when new component is to be displayed. Method *removeComponent* is called when component should be removed. These two methods must be overridden.

6. How to create simple "Hello world" form using our GUI

First, we will create form and set form's title to "Hello world".

```
Ptr<TkTopLevel> frmHello=new TkTopLevel(interp);
frmHello->create();
frmHello->doLayout();
frmHello->setTitle("Hello world !");
```

Second, we will add a button to this form. As layout, we will use *TkGridLayout* manager.

```
TkButton* button=new TkButton(interp,"Say hello to the world");
frmHello->addChild(button, new TkGridConstraints(0,0));
frmHello->setLayoutManager(new TkGridLayout());
```

So we have displayed form with button. Next, it would be nice to add some action to the button press.

```
class HelloButtonPressListener:public ButtonPressListener
{
public:
  void onButtonPress(EHeader& header,int button,const string& state,int
x,int y,int x_root,int y_root) {
    cout<<"Hello world for pixel at ["<<x<<","<<y<<"]"<<endl;
};

button->addButtonPressListener(new HelloButtonPressListener());
```

We're done. The console would write coordinates of pixel in button whenever we click on button.

7. How to use form created in designer

1. Forms from designer are stored in form database. This database is represented in class *TkGui*, which can be accessed via *ModuleManager*. This forms can be accessed via their names. Suppose, we would like to create form for entering module variables after module has been instantiated. We've designed this form before and named it "myform".

   ```
   void OurModule::afterInstantiate()
   {
     Ptr<TkGui> gui=getManager()->getGui();
     Ptr<TkTopLevel> window=gui->cloneSourceForm("myform");
     ...
   }
   ```

   We have now reference to *TkTopLevel*. Note that it wouldn't be best idea to modify stored form directly – if we modify stored form, and save form database, we will store also the modification. So we get form's deep clone instead.
2. We would like to display it. Stored forms are nor created, nor displayed. Forms are also in window state "WITHDRAWN" which means, they are invisible. So we must create form, display it and change state to "NORMAL".

   ```
   window->create();
   window->doLayout();
   window->setWindowState(TkWinStateOption::opts::normal);
   ```

3. If we designed some widgets in form to be bound to the module directly, we must also activate this bind. This is done by *attachToModule* method.

   ```
   window->attachToModule(this);
   ```

4. This can be all. If we need to access some specific widgets in form, to give them additional listeners or modify them we can access them via their names. For example, we've designed entry named "txtInput".

```
TkEntry* entryInput=(TkEntry*)window->getChildByName("txtInput");
```

D. User's handbook

1. Tcl shell

1.1 How it works

All except the most recent version of tcl (ver 8.0) are completely string based interpretive. The newest version includes some binary phases to increase speed. The language is procedure based, with all operations taking a list of string parameters. Even the built in tcl commands are approached in this fashion. The string parameters are handled in a list, with list operatives being built into the language. When dealing with C the items in the list become strings in the *argv[] array which is supplied to the procedure, in a similar fashion in which the operating system hands the command line parameters to the *main()* function of a program.

The list of parameters is interpreted before being passed to the procedure which is called. Any variables which are in the parameter list are replaced with their contents. This replacement is similar to "glob" replacement by a shell, where a "*.html" in a command is replaced by a list of all of the files ending in the ".html" extension.

Once a procedure has received its arguments, it processes and then returns. A numeric value is returned by most built in commands, with '1' indicating success and '0' indicating failure. Some of the more complex commands can also return string values.

1.2 Variables and the "set" operator

Variables in tcl/tk are "declared" by setting them to a value. Each variable can be "set" multiple times, and exists within scope until "unset". When setting a variable all that is required is the variable name. When using the variable the name must be preceded by a "$".

1.3 The "puts" command and the comments symbol "#"

A few quick notes to get you started: the comment symbol in tcl/tk is "#" and it works up until the next end of line character. The selection of "#" is rather convenient as it allows tclsh to ignore the line which is used by UNIX to determine what shell to run. The "puts" command takes a string and prints it to the screen. Now for your first tclsh program. Not wanting to break with tradition, here is "hello world":

```
#!/.software/local/.admin/bins/bin/tclsh

#the following outputs the text "hello world" to the terminal
puts "hello world"
```

The output looks like this:

```
hello world
```

Notice that the "puts" automatically prints the new line character for you at the end of the string. From here on in, the header won't be bothered with, as it should be understood. If you are trying to run the above piece of code don't forget to make the file executable. Also bear in mind the *#!/.software/local/.admin/bins/bin/tclsh* line should be replaced with the full path location of your tclsh interpreter.

1.4 Putting things in variables

Now that we can put things to the screen, how about playing with some variables. The following program uses the "set" operator to put the string "world" in the variable "stuff". Notice that the "$stuff" is interpreted by the "puts" command, and the contents are printed, not the word "stuff".

```
set stuff world
puts "hello $stuff"
```

Simple enough. So what if you actually want a "$" to show up in your text? Like in C the back-slash ("\") character allows you to display "non-printable" characters. The following code will print: "hello stuff"

```
puts "hello \$stuff"
```

Like the "set" operator there is also an "unset" operator. The "unset" command removes the variable from existence, and any further references to that variable will cause an error. The following code will print the error: "can't read `stuff`: no such variable".

```
set stuff world
unset stuff
puts "hello $stuff"
```

1.5 Number representation

Everything seen so far has been dealing with text. There is a good reason for this, everything in tcl is text. This is not to say that numbers can not be handled, but to do so, tcl requires special commands. Numbers are strings with *just* numbers in them. A mix of both numbers and letters will most often result in tcl interpreting the string as a word.

1.5.1 The "incr" command

The "set" and "unset" operators work just as they did above with strings when dealing with numerics. The "incr" operator takes a variable and increments it by a specified value. The following piece of code plays with a variable called "number" to give you an example of the "incr" operator.

```
set number 3
puts "the number is $number"

incr number 2
puts "the number is now $number"

# if you don't give "incr" the second parameter it defaults to 1
incr number
puts "the number is now $number"
```

Notice that the "incr" operator is given the name "number" *without* the "$", this is because we don't want the interpreter to give "incr" the value in "number", but the name of the variable itself.

1.5.2 Number Representation

Tcl also handles negative numbers, scientific, hexadecimal, octal, and floating point. The following code shows off all of these things:

```
set number 3.14
puts "pi is almost $number"

set number -7e2
puts "a negative scientific number: $number"

set number 0xFFFF
puts "all good programmers like hex: $number"

set number 01234
puts "just like C a leading `0` means octal: $number"
```

1.6 Interpretation using [] & {} and the "expr" operator

The square and brace brackets have special meaning in tcl, and effect how the interpreter handles the string contained within. The brace brackets force no interpretation, and everything inside is passed to a command as it appears. There are numerous places where a programmer might not want interpretation to happen, and this will become more clear in the sections on procedures and controls.

1.6.1 The Square Brackets "[" and "]"

The square brackets mean evaluation. In essence any string inside of the square brackets is treated as a tcl script and run by the interpreter. This parallels the back-quote in many shell programming languages.

1.6.2 The "expr" command

A useful operation is the "expr" command, this is short for expression and it returns the results of an equation contained in the string passed to it. The combination of the square brackets and the "expr" operator allows manipulation of variables. The following code sets a variable called "number" to 3, and then sets a variable "stuff" to be 10 more than what is contained within "number". Notice that the contents of "number" are not effected.

```
set number 3
set stuff [expr $number + 10]
puts "stuff contains: $stuff"
```

The above code works by evaluating the script between the square brackets, and replacing this code with the returned value from the script. The script has its variables interpreted before it is run, so the "expr" command sees the string: "3 + 10". The "expr" command takes the string, realizes that we want the numbers added together and returns a string containing the resulting "13". This resulting string replaces where the square brackets were, and so we get "set stuff 13", which of course puts the value "13" in the variable called "stuff".

1.6.3 Semi-colons

The semi-colon character allows the programmer to issue multiple commands on the same line. This becomes useful if we wish to put a couple of operations inside of a set of square brackets. The following code outputs "hello world" to the screen as well as the contents of "stuff".

```
set number 3
set stuff [puts "hello world"; expr $number + 10]
puts "stuff contains: $stuff"
```

The return value from a script within square brackets is whatever the *last* line in the script returns. If we had put the "puts" after the "expr" in the above example, the contents of stuff would have been the empty string, since "puts" returns the empty string.

The semi-colon is a neat trick, but it really isn't necessary. It is perfectly legal to have end of line characters inside of square brackets. The following code does the same thing as the above example.

```
set number 3
set stuff [puts "hello world"
expr $number + 10]
puts "stuff contains: $stuff"
```

1.7 Procedures

What good is a programming language without being able to call procedures? Tcl supports the basic procedure stuff that you would expect, in fact all of the commands which have been looked at so far have just been procedures which are provided for you by the language. Calling a procedure is as simple as calling any of the commands, just give the name of the procedure plus any arguments you wish to pass it. The procedure must be defined *before* its first calling. Recursive procedures are supported.

1.7.1 Defining A Procedure: "proc"

The "proc" key-word is what allows you to define a procedure. After "proc" you give the name of the procedure, the name of arguments and then the script which is to be executed when the procedure is called. The arguments *must* be there, even when you don't need any. There are a couple of ways of dealing with this: give the name "args" and then ignore it, or use a set of brace brackets. The script which is executed when the procedure is called is also contained in brace brackets, this follows from the fact that we do not want the contents interpreted during the definition of the procedure, only during execution. The following code is a new twist on our familiar "hello world"

```
# the following code defines the procedure "hey_world"
proc hey_world {} {
  puts "hello world"
}

# the following code calls "hey_world"
hey_world
```

1.7.2 Arguments

Accessing the arguments passed to a routine can be done in two ways. If you know how many arguments you want, then simply specify them inside of brace brackets. The second method is using the key-word "args" instead of an argument name, this puts all of the arguments in a variable called "args" which can then be accessed as a string or list. The following example takes exactly two arguments:

```
proc print {one two} {
  puts "var one: $one"
  puts "var two: $two"
}

print hello world
print "hey there you big" "fat globe you"

# the following line will cause an error
print hey there you big fat globe you
```

Notice that specifying items inside of quotes tells tcl that the whole thing is an argument. The second call to print passes two arguments, the third call to print attempts to pass seven arguments and thus fails.

1.7.3 Variable number of arguments

By using the "args" key-word in your argument list, you obtain a sort of catch-all. All of the arguments that aren't taken up by variables before the "args" are put into a variable called "args". This variable can be accessed like any other.

```
proc print {one args} {
  puts "var one: $one"
  puts "var two: $args"
}

print hello world

# the following line no longer causes an error
print hey there you big fat globe you
```

The section on strings discusses some ways of getting at each of the space-separated items in the "args" variable. Tcl also includes a fair amount of list processing techniques which are also useful, these can be found by looking at the web sites mentioned, or in the man pages.

1.7.4 Returning a value

The return value of a procedure is implicitly the return value of the last statement in the script. If you wish to force leaving of a procedure early, or return a specific value that isn't returned by some other line, use the key-word "return". This works just like the return statement in C. A simple procedure to add two numbers and return the sum is given below.

```
proc add2 {num1 num2} {
  return [expr $num1 + $num2]
}
```

1.8 String substitution, commands & regular expressions

Strings in tcl/tk are a lot like those in C. We have already seen examples of variable substitution using the "$" operator, but this is not the only special character. As was mentioned in the introduction to variables the "\" character can precede a "$" to print a "$" and not have it interpreted as a special character. Tcl also provides a fairly standard array of "\" characters:

Character	
\$	the "$" character -- instead of variable substitution
\f	form feed
\a	audible alert
\b	back space
\n	new line
\r	carriage return
\t	tab
\xHH	any character represented by 0xHH hex (H ranging from 0 - F)
\ddd	any character represented by 0ddd octal (d ranging from 0-7)

Any character that is put after a back-slash that isn't in the above list is replaced by just the character. Thus "\\" is just "\", and "\p" is just "p". The "\\" is useful when you want a "\" to show up in your output.

1.8.1 String Commands

Everything in tcl is based on strings, so it is fitting that there be a series of string manipulation routines. The string routines are all based on a single command "string" which takes as its first parameter an indicator of what you would like to do with the string. Features exist for finding the length, ranges, and doing comparisons. The following code illustrates some of these features

```
set string1 "this is my most fine string"
set string2 "this is another string"

set size [string length $string1]
puts "this size is $size"

# comparison returns 0 for equal, 1 for larger, -1 for smaller
puts [string compare $string1 $string2]
puts [string compare $string2 $string1]
puts [string compare $string1 $string1]

# range returns a part of a string -- it starts at 0
puts [string range $string1 8 14]

# index returns the nth character of a string
puts [string index $string1 16]
```

The "string compare" operator evaluates greater than as coming later in the alphabet. Whenever any counting is done on strings (e.g. the "index" and "range" commands) the first character is accessed using "0". This mimics the C behaviour of strings as arrays. Other "string" operations include "first", "last", and "match", which all look for sub-patterns in the string. The "tolower" and "toupper" operators return an all lower case or upper case version of the string, respectively. The "trim", "trimleft", and "trimright" operators are used to remove white-space from the left, right, or both sides of the string. The trim operators can also be given specific characters to remove instead of the white-space.

1.8.2 Regular Expression Operator "regexp"

The "regexp" operator allows some of the rudimentary regular expression operations to be performed on a string. The typical regular expression operators are supported: ".", "^", "$", "\", "[...]", "(...)", "*", "+", "?", and "|". The "regexp" operator takes an argument for the expression to be evaluated and an argument for the string to be operated on. The "regexp" command returns a '1' or '0' indicating success and failure. Another command called "regsub" does the regular expression operations using substitution. For more information on either of these see the man pages.

1.9 Control Flow

Control flow in tcl/tk is performed using procedures. In most cases the control flow statement takes several scripts as arguments, some of the scripts being conditions for execution and some for the script to be executed if the condition is true.

1.9.1 The "if" Statement

The "if" statement takes two scripts as arguments, the first is the condition, and the second is what is to be run if the condition is true. The "if" statement can also be paired with "elsif" and "else" statements. The following code shows an example

```
if {$i < 3} {incr n}
elsif {$i >3} {incr n 2}
else {incr n 3}
```

This simple example shows the usage of each of the key-words. The resulting scripts to be executed can be multiple lines and multiple commands. Be careful that the opening brace bracket is on the same line as the "if", "elsif", or "else" key-word, otherwise it won't work.

1.9.2 The "while" Statement

The "while" statement takes two scripts as arguments, the first is the condition, and the second is the script that is executed multiple times until the condition is no longer met. If the condition is not met in the first place then the execute script is *not* run.

```
set i 0
while {$i < 3} {puts "hello"; incr i }
```

This example will print the word "hello" three times.

1.9.3 The "for" Statement

The "for" statement is unnecessary, as everything that can be done with a "for" can be done with a "while". The "for" statement, builds in some of the common things that happen in a "while" as a short-cut for the programmer. The "for" statement behaves in a similar fashion to its C counterpart. This command takes four scripts. The first script is where variables are initialized, the second is the stopping condition, the third script is for incrementing, and the last is the actual body of the loop.

```
for {set i 0} {$i < 3} {incr i} {puts "hello"}
```

This funky little one line example does the same thing as the previous example using a "while" statement, just a little more compactly.

1.9.4 The "foreach" Statement

The "foreach" statement is very useful when trying to process lists or parse. This command takes three arguments, the first is a variable name, the second is a string (or list), and the third is a script. For each item in the list the third script gets executed. Upon each iteration of the loop the first variable gets the next word in the list. The following code prints the words "one", "two", and "three" on separate lines.

foreach i "one two three" { puts $i }

A common use for this statement is to process the "args" variable in a subroutine. The next example shows a procedure which returns the sum of a series of numbers.

```
proc sum args {
  set total 0
  foreach i $args { set total [expr $total + $i] }
  return $total
}

puts "the total is [sum 1 2 3 4 5 6]"
```

1.9.5 The "switch" Statement

The "switch" statement is a shortcut for multiple if-elsif clauses, using the same condition variable. The structure of the "switch" is a little different from the other control statements. It also takes an argument and a script, but the script has to contain a specific format. The following code fragment should help to make this clear.

```
switch $x {
  a -
  b {puts "got an a or b"}
  c {puts "definitely got a c"}
}
```

In the above fragment the contents of variable "x" is compared to the string "a". If the value of "x" is "a" then perform "-", which in this case means to execute the next script which is available. The next available script here is that following the "b" condition. If the "a" to contents of "x" comparison fails, then the "switch" compares against the next item, which in

this case is the "b". This process continues until a match is found, or the end of the "switch" statement is found. For example, if the contents of "x" was "a", then "got an a or b" would be output to the screen. Like with the "if" statements, multi-command, multi-line scripts can be used within the "switch" as long as the opening brace bracket is on the same line as the condition.

1.10 Error handling via catch and the "exec" command

Tcl does not handle errors very well, the addition of the tk toolkit actually makes the error handling even uglier. Typical response of tcl to errors is to bail out. Tk is a little better in that it brings up an exception window, but there is not much that can be done after that. In the most frequent cases the error has been caused from a syntax or illegal operation, and so forcing the exit of a program is almost expected, so why the concern? A useful aspect of most scripting languages is an interface out to the command line so that programs can be run from the operating system. The problem arises in what these programs return. Many UNIX programs are not very careful about their exit codes, and the meaning of the codes varies from program to program. Unfortunately a non-zero error code is interpreted by tcl as a fatal error. The "exec" command takes a string which is the command to pass through to the operating system. The following example uses the UNIX date program to print the date to the screen.

```
puts "the date is: [exec date]"
```

The date program is relatively well behaved, if it had returned with a non-zero exit code the tcl program would have stopped. The solution to this is the "catch" statement. The "catch" operation catches any exceptions thrown by a command, and returns '1' or '0' indicating success or error of the command. An added bonus is that "catch" can also take a variable for the output of a command. The following code snipet is a safe version of the previous example.

```
catch {exec date} stuff
puts "the date is: $stuff"
```

2. Module system commands

2.1 Module instantiation

For module instantiation is the only command - **module**.
Syntax is: **module** module_type name

- module_type is type of module, you can list all registred module types using command **modlist**
- name is name of instance, it is your's to decide but the instantiation will fail if there is already module instance with the same name

Another command is **default**.
Syntax is: **default** name

- name is the name of instance you've created using command **module**, this command will make selected instance default, so if you don't specify in command **vars**, **sets** and **run** module instance, default instance will be used.

2.2 Module variables

Any module can have set of variables. You can list them or set their values.
Command **vars** handles listing.
Syntax is: **vars** name

- name is the name of module instance

Command **sets** handle setting.
Syntax is: **sets** name varName value

- name is the name of module instance
- varName is the name of variable (use vars to get list of all variables)
- value is new value of variable

As you may have noted, commands for setting module variables are slightly different from Tcl. That is because they are stored in different places and accessed in different way.

2.3 Module commands

Basic module command is **run**. This command run algorithm in module instance.
Syntax is: **run** name

- name is the name of module instance

Another command is **on**. Module instances can have various commands for example for writing results to console etc.
Syntax is: **on** name do [arg1 arg2 argN …]

- name is the name of module instance
- do is the name of command specific for given module
- arg1 … argN are arguments of command do

2.4 Module system help

Command for help is **helps**.
Syntax is: **helps** [commandName]

- commandName is name of command, on which we want help, if not specified **helps** write list of available commands

Note that help support is only for the module system commands.

E. Obsah CD

Umístění adresáře/souboru	Obsah
/src	zdrojové soubory, Makefile
/doc	dokumentace tříd
/lib/win	potřebné knihovny pro systém Windows
/lib/linux	potřebné knihovny pro systém linux
/bin/win	binární soubory pro systém Windows
/bin/debian-i386-2.6.8	binární soubory pro systém Debian (platforma i386, kernel 2.6.8)
/dp.doc	tato diplomová práce
/dp.pdf	tato diplomová práce