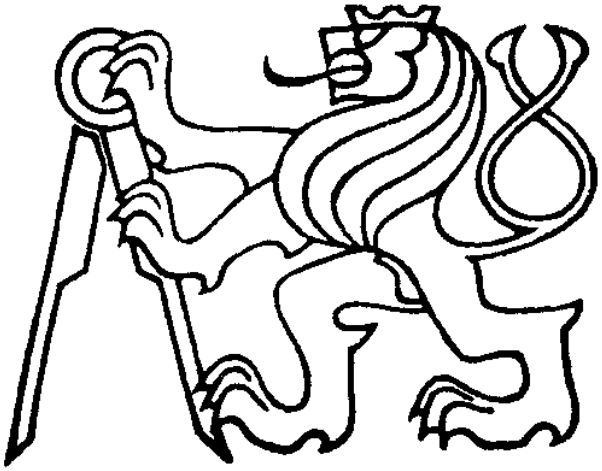
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

BAKALÁŘSKÁ PRÁCE

Objektově orientované nástroje vizuálního programování

Libor Smékal



1 9 9 7

Obsah

Úvod

V současné době se můžeme na trhu setkat hned s několika softwarovými produkty, které nám umožňují vytvářet aplikace s grafickým uživatelským rozhraním na velmi vysoké úrovni. Pokud se zaměříme na vývoj aplikací pro MS Windows (příp. Windows 95 nebo Windows NT), jedná např. o Borland C++ , Visual C++, Watcom C++ a také o systém Borland Delphi, který patří k nejpopulárnějším. Všechny tyto systémy umožňují vizuální návrh aplikace, také je všeobecně známo, že ve všech systémech se při návrhu aplikace využívá **objektově orientovaného programování** (dále jen **OOP**). Je tu ovšem ještě jeden systém, který umožňuje stejně snadný vizuální návrh aplikace pomocí OOP, přesto se však od předchozích poněkud liší. Jedná se o systém **Visual Works / Smalltalk-80 (**dále jen **Visual Works.)**, kterým se budeme v této práci zabývat především. Ve stručnosti bude popsána jeho architektura, návrh aplikace a srovnání s návrhem v Borland Delphi (verze 2.0, příp. 3.0). Vizuální návrh se opírá především o práci s vizuálními komponentami a právě popisu jejich tvorby ve Visual Works bude věnována hlavní část této práce.

Základní rysy jazyka Smalltalk

Cílem této kapitoly není popsat programování v jazyku Smalltalk. Kapitola je zde zařazena proto, aby čtenáři, který se ještě se Smalltalkem nesetkal, umožnila pochopit části kódu, které se budou vyskytovat dále. V této kapitole jsou tedy uvedeny ve stručnosti opravdu jen základní rysy, vše další je uvedeno v [1].

Programovací jazyk Smalltalk je důsledně objektový, vše (znaky, čísla, řetězce, ...) jsou objekty.

Objekty ve Smalltalku

V následující tabulce jsou uvedeny základní objekty.

OBJEKT	PŘÍKLAD OBJEKTU ZAPSANÉHO VE SMALLTALKU	POPIS PŘÍKLADU
Číslo	`1`	Číslo 1
Znak	`$a`	Znak 'a'
Pole	`#(1 2 3)`	Pole obsahující čísla 1, 2, 3
Řetězec	`'abc'`	Řetězec obsahující znak 'a', znak 'b' a znak 'c'

Objekty uvedené v tabulce odpovídají základním datovým typům většiny procedurálních programovacích jazyků. Mezi základní a běžně používané objekty ve Smalltalku však patří i takové, které obdobu v jiných jazycích nemají. My se teď zaměříme jen na dva z nich, protože je budeme využívat dále v textu při popisu vizuálních komponent.

Prvním z nich je bod (`Point`). Jde o objekt, který má dvě složky (souřadnice) `x`, `y`. Zápis v jazyku Smalltalk má tvar `x@y`. Objekt `1@2` tedy představuje bod, jeho složkou x je číslo 1 a složkou y číslo 2.

Druhým je objekt `OrderedCollection`. Jde o kolekci prvků, na níž je definováno uspořádání. Můžeme stejně jako u pole určit pořadí jednotlivých prvků, rozdíl je však při přístupu k jednotlivým prvkům. Zatímco u pole využíváme při přístupu k jednotlivým prvkům indexace, u uspořádané kolekce můžeme také určit první resp. poslední prvek, přidat prvek na

začátek resp. konec, určit předchůdce resp. následníka zadaného prvku kolekce. Právě operace přidání prvku nám umožňují měnit velikost kolekce, což je další rozdíl oproti poli (pole má svoji velikost danou při vzniku).

Zprávy

Pokud chceme pracovat s objektem, máme ve Smalltalku pouze dvě možnosti (dva typy výrazů) - můžeme jej pojmenovat nebo mu poslat zprávu.

Na poslání zprávy zareaguje objekt vyvoláním příslušné metody (pokud taková existuje, jinak je hlášena chyba) a vrácením výsledku, což je samozřejmě zase nějaký objekt.

Zprávy dělíme na

1) **unární**

Unární zprávy jsou alfanumerické konstanty, zapisují se za příjemce zprávy. Tyto zprávy nemají parametry. Zápis

```
a value
```

znamená poslání unární zprávy `value` objektu `a`.

2) **binární**

Binární zprávy jsou jedno nebo dvouznakové konstanty zapsané za příjemce zprávy následované jedním parametrem. Zápis

```
3 + 5
```

znamená poslání binární zprávy + s parametrem `5` objektu 3.

Také výše zmíněný zápis bodu 1@2 představuje vlastně poslání binární zprávy @ s parametrem 2 objektu 1.

3) **slovní**

Slovní zprávy jsou alfanumerické konstanty, které se při zápisu píší za příjemce zprávy a jsou rozděleny na tolik částí, kolik mají parametrů. Každá část musí být zakončena dvojtečkou (Právě podle dvojtečky snadno rozeznáme jedno nebo dvojznakovou zprávu s jedním parametrem od zprávy binární). Zápis

```
a value:3
```

znamená poslání slovní zprávy `value:` s jedním parametrem 3 objektu `a`.

Zápis

```
x between:1 and:5
```

znamená poslání slovní zprávy `between:and:` se dvěma parametry objektu `x`. Prvním parametrem je číslo 1, druhým číslo 5.

Při vykonávání programu se nejprve vyhodnocují unární zprávy (mají nejvyšší prioritu), potom binární zprávy a nakonec slovní zprávy (mají nejnižší prioritu). Toto pořadí lze, stejně jako v jiných programovacích jazycích, měnit pomocí kulatých závorek.

Pojmenování objektů

Pojmenování představuje druhou možnost, jak pracovat s objekty. Zápis

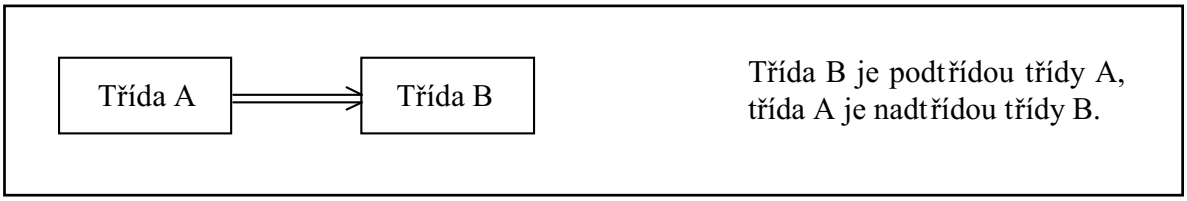
```
x := y
```

znamená, že objekt `y` je nyní přístupný i pod jménem `x`. Je důležité uvědomit si, že pod označením `x` a `y` se skrývá tentýž objekt, operátor `:=` způsobil, že jména `x` a `y` označují totéž paměťové místo !

Grafické vyjádření vztahů a interakce mezi objekty

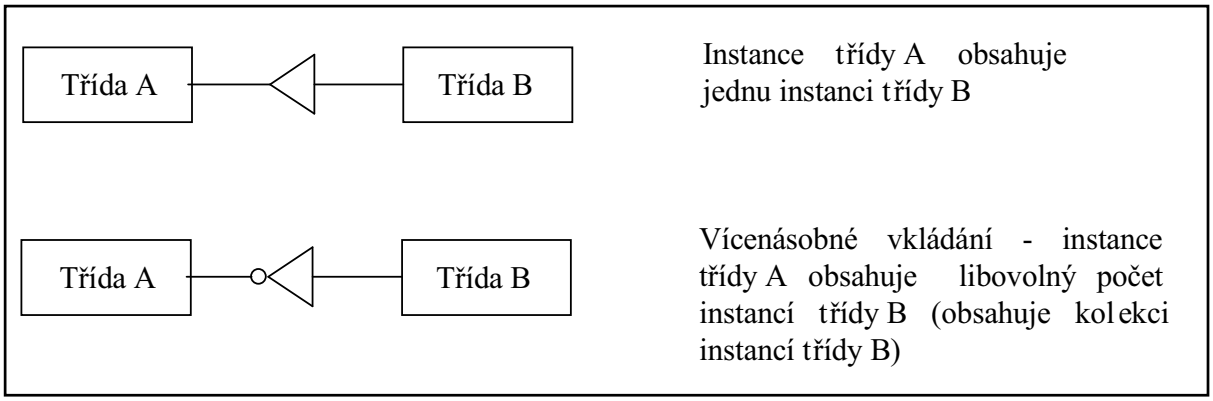
Při popisu architektury systému Visual Works budeme využívat základních vztahů mezi objekty. Jedná se o dědičnost, skládání a závislost. K jejich vyjádření budeme dále v textu používat následující značení:

Dědičnost



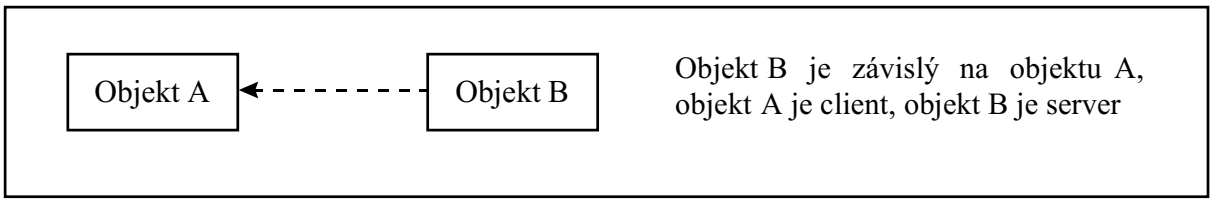
Obr. 1

Skládání



Obr. 2

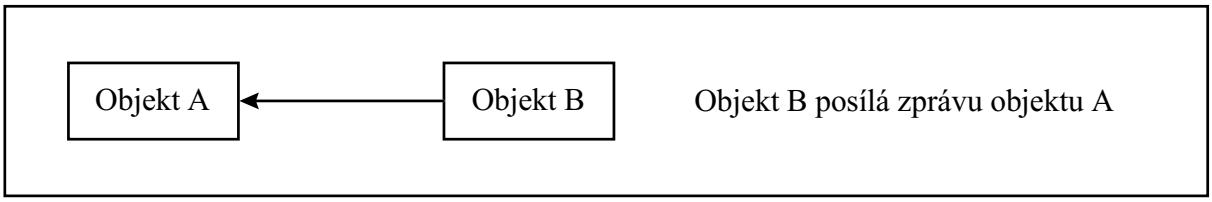
Závislost



Obr. 3

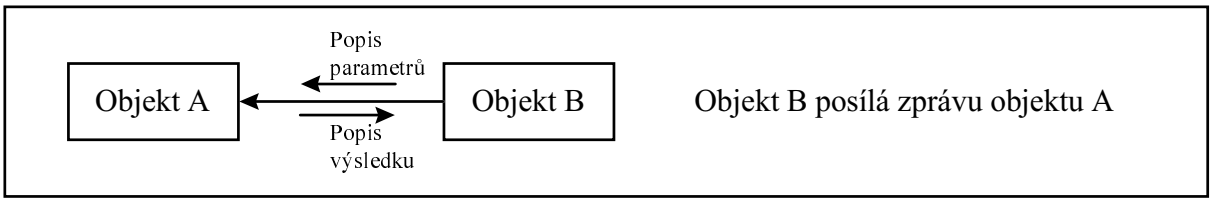
Poslání zprávy

Pro popis interakce mezi objekty budeme ještě potřebovat vyznačit poslání zprávy. Při zasílání více zpráv může být u jednotlivých šipek označujících poslání zprávy číslo, které bude označovat pořadí vyslání dané zprávy.



Obr. 4

Pokud bude nutné blíže specifikovat parametry zprávy nebo vrácený výsledek, bude poslání zprávy vypadat následovně:

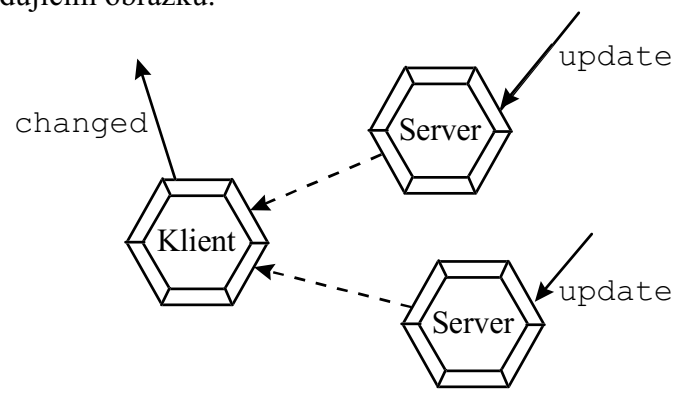


Obr. 5

Vazba závislosti mezi objekty

Vazba závislosti, narozdíl od dědičnosti nebo skládání, v jazycích typu Object Pascal nebo C++ neexistuje, ve Smalltalku je však velmi často využívána. Má také zásadní význam v architektuře MVC a proto se jí teď budeme zabývat trochu podrobněji.

Pro pochopení této vazby si lze představit, že závislý objekt - server sleduje chování objektu - klienta, na němž je závislý, a reaguje na jeho chování daným způsobem. Přitom ale client o serveru (příp. serverech) vůbec nemusí vědět. Chování serveru a klienta je znázorněno na následujícím obrázku.



Obr. 6 - Vazba závislosti mezi objekty

Na obrázku jsou také vyznačeny dva druhy zpráv, které se při závislosti mezi objekty uplatňují. První je zpráva `changed`, kterou vysílá klient, pokud chce cokoliv „oznámit“ svým serverům. Na této zprávě je zajímavé to, že není poslána žádnému konkrétnímu

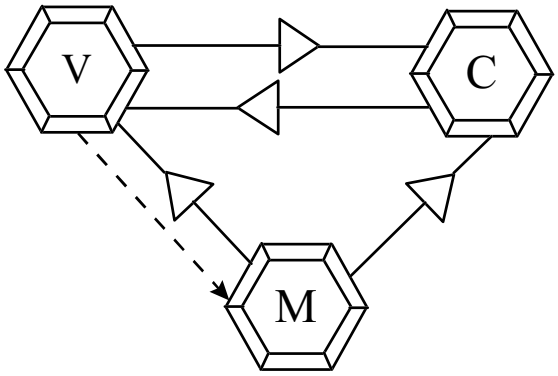
příjemci. Protože však ve Smalltalku lze poslat zprávu jen někomu, realizuje se to tak, že zprávu `changed` posílá klient sám sobě. Toto vyslání způsobí, že se u všech serverů vyvolá metoda `update`. V praxi se většinou zpráva `changed` posílá s parametrem kvůli bližšímu určení zprávy. Navázání vazby závislosti se uskutečňuje tak, že se klientovi zašle zpráva `addDependent:`, jako parametr se uvede daný server.

Díky tomuto mechanismu je možné, aby klient zasílal zprávy serverům, přitom se však nikde v kódu jeho metod adresy nebo jiná identifikace serverů neobjevuje. Proto je ve Smalltalku možné, aby se nově vytvořený objekt za běhu aplikace připojil (navázal vazbu závislosti) k nějakému již dříve vytvořenému objektu a začal např. zobrazovat jeho data.

Architektura MVC

Architektura MVC je základním prostředkem, pomocí něhož je realizováno **Grafické uživatelské rozhraní (Graphics User Interface,** dále jen **GUI).** Zkratka MVC je utvořena z prvních písmen třech objektů, které tuto architekturu tvoří. Jedná se o **M**odel, **V**iew a **C**ontroller.

Základní úlohou view je v grafické nebo textové podobě zabezpečit zobrazení aktuálního stavu modelu. Zároveň také ale view musí spolupracovat s controllerem, který zprostředkovává interakci s uživatelem, většinou pomocí myši nebo prostřednictvím klávesnice. Na základě zpráv od controlleru může view měnit stav modelu. Také controller může přímo spolupracovat s modelem.



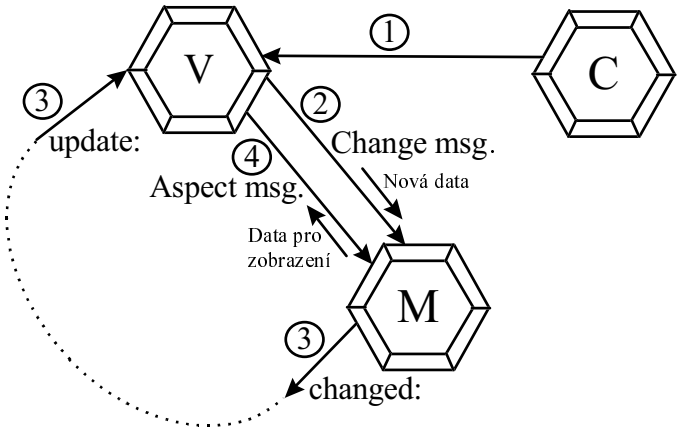
Obr.7 - Vazby skládání a závislosti v MVC

Nyní k vlastní architektuře MVC. Jak je vidět z obrázku, view a controller jsou obousměrně propojeny vazbou skládání. Toto poměrně těsné spojení je možné, protože při vývoji MVC komponenty (tlačítka, editačního boxu, ...) se view i controller vyvíjí současně, view ví, kdo je jeho controller a jak vypadá (jaké má metody). Totéž platí i obráceně. Z těchto důvodů interakce mezi view a controllerem nemusí být blíže specifikovaná.

Poněkud jiná situace ovšem nastává mezi modelem a view a také mezi modelem a controllerem. Model většinou představuje data aplikace, která chceme zobrazovat a ovládat pomocí již dříve napsaného view a controlleru. Také kvůli tomu není vazba mezi modelem a jeho dvěma partnery tak těsná. Model je vložen do view i do controlleru a zároveň zde existuje vazba závislosti mezi modelem a view. Komunikace mezi modelem a view probíhá prostřednictvím tzv. aspect zpráv, change zpráv a pomocí vazby závislosti.

Aspect zprávou žádá view model o data, která má zobrazit. Tato data jsou navrácena jako výsledek této zprávy. **Change** zprávu zasílá view modelu, jestliže mu potřebuje předat

nová data. Data jsou předávána jako parametr této zprávy. Pokud došlo v modelu ke změně dat, musí se to view dozvědět kvůli případnému překreslení. Proto model v tento moment využije vazbu závislosti, vyšle zprávu **`changed`** a to způsobí vyvolání metody **`update`** u všech view.



Obr. 8 - Příklad komunikace mezi objekty MVC

Jak může vypadat interakce mezi objekty MVC je vidět na obrázku. Daná situace reprezentuje např. zadávaní telefonního čísla pomocí editačního boxu. Nyní si popíšeme jednotlivé kroky:

1. Uživatel zadal celé telefonní číslo a právě stiskl klávesu ENTER. Naznačenou zprávou controller oznamuje view, že byla stisknuta tato klávesa.
2. View se díky zprávě od controlleru dozví, že bylo ukončeno zadávání dat a pošle modelu zprávu typu change. Parametrem této zprávy je řetězec zadaných znaků představující telefonní číslo.
3. Zpráva od view vyvolá příslušnou metodu modelu, v které se data upraví (např. zkontroluje se, že řetězec s tel. číslem obsahuje jen povolené znaky, upraví se do požadovaného formátu, ...) a uloží do instanční proměnné. Protože uvnitř modelu došlo ke změně dat, musí model vyslat (poslat sám sobě) zprávu `changed:`. To díky vazbě závislosti způsobí, že se u všech jeho serverů vyvolá metoda `update:`.
4. Vyvolání metody `update:` u view znamená, že model změnil svůj stav. Proto view zašle modelu zprávu typu aspect. Model jako výsledek této zprávy předá zpět upravená data ze své instanční proměnné a view se postará o jejich patřičné zobrazení.

Při interakci mezi objekty MVC nemusí vždy dojít ke všem těmto krokům. Např. po svém vytvoření view jen požádá o data, která má zobrazit a proběhne tedy jen krok 4. Nebo může dojít ke změně dat jejich načtením ze souboru, model pak jen žádá o překreslení a proběhnou kroky 3 a 4.

Podrobnější rozbor architektury MVC

Následující odstavce by měli objasnit, co je třeba k vytvoření vlastního view a controlleru. View i controller lze samozřejmě vytvořit mnoha různými způsoby, zde bude předveden jeden z nich. Zaměříme se především na popis metod, které je třeba předefinovat nebo které jsou důležité pro správnou funkci.

Důležité je, aby nově definované view bylo potomkem třídy `View`, obdobně nový controller potomkem třídy `Controller`. Tím bude zajištěna správnost základních funkcí našich nových tříd. Nové view by mělo obsahovat třídní metodu, která zajistí vytvoření instance. Parametry této metody bude:

- model
- symbol určující, která zpráva je aspect zprávou
- symbol určující, která zpráva je change zprávou

Důležité je zajistit, aby vytvořená instance zavolala svoji zděděnou metodu `model:` , která provede uložení modelu do své instanční proměnné (tzn. vytvoří vazbu skládání mezi modelem a view), zároveň však vyvolá následující kroky:

- Navázání vazby závislosti mezi modelem (klient) a view (server) - view pošle modelu zprávu `addDependent:` s parametrem `self` (proměnná **`self`** označuje ten objekt, součástí jehož rozhraní je kód, ve kterém je tato proměnná použita, v tomto případě tedy view).
- Zaslání zprávy `model:` controlleru, který si do své instanční proměnné uloží model, který byl parametrem zprávy.

V předchozích odstavcích bylo popsáno vytvoření vazby závislosti a skládání mezi modelem a view a vazby skládání mezi modelem a controllerem. Z obr. 7 je vidět, že zbývá ještě vytvořit obousměrnou vazbu skládání mezi view a controllerem. To proběhne tak, že při začleňování view do systému je view systémem požádáno zprávou `getController` o controller. To způsobí vyvolání následujících kroků:

- Zavolá se metoda `defaultControllerClass,` výsledkem je třída controlleru pro toto view.
- Dojde k vytvoření controlleru , vytvořený controller je uložen do instanční proměnné
- Controlleru je poslána zpráva `view:` s parametrem `self,` controller si uloží do své instamční proměnné view

Všechny tyto akce mohou proběhnout díky tomu, že naše nové třídy byly odvozeny pomocí dědičnosti ze tříd systému. Jedinou metodou, kterou je většinou třeba předefinovat, je metoda **`defaultControllerClass`**. Musíme to udělat tehdy, jestliže chceme použít jiný controller než standardní třídu `Controller`.

Důležité metody view

- **`displayOn:`** - Metoda charakteristická pro všechna view. Je volána vždy, když je třeba view překreslit. V této metodě tedy dochází k vlastnímu grafickému výstupu. Zobrazení se provádí na tzv. `graphicsContext`, který je parametrem této metody.
- **`defaultControllerClass`** - Jak již bylo zmíněno výše, tato metoda vrací jako výsledek třídu controlleru určeného pro toto view.
- **`update:`** - Metoda je vyvolána tehdy, jestliže došlo ke změně stavu modelu. Je v ní třeba zajistit (např. pomocí aspect zprávy) vyzvednutí nových dat a překreslení view.

Důležité metody controlleru

- **`controlLoop` -** Tahle metoda se nepředefinovává, ale je důležitá pro pochopení funkce controlleru. Její kód vypadá následovně:

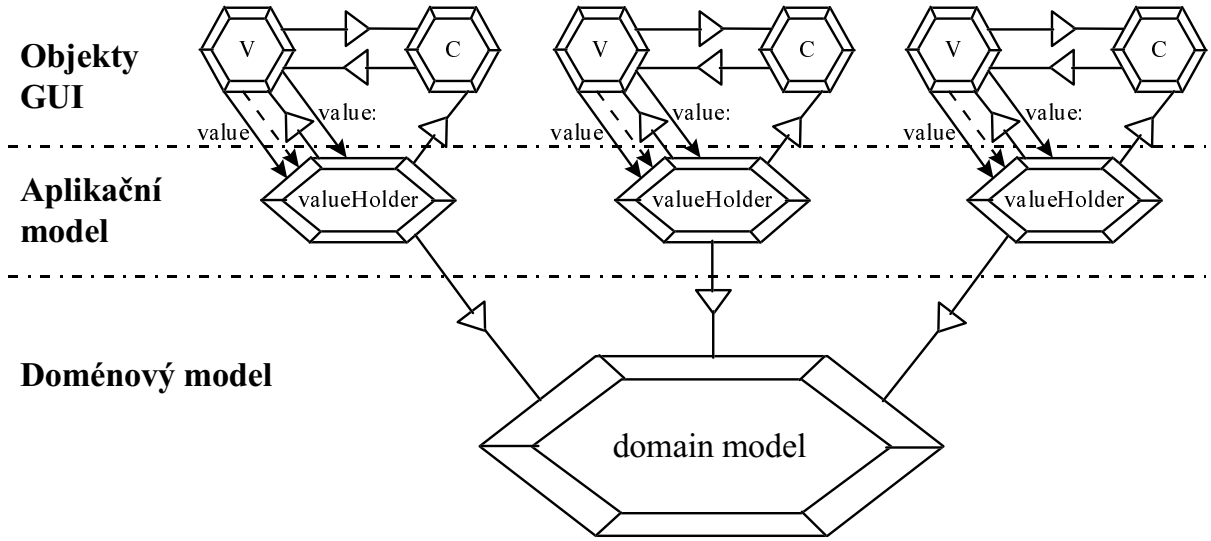
```
[self pool.
self isControlActive]
     whileTrue:
          [self controlActivity]
```

- **`controlActivity`** - V této metodě probíhá vlastní testování vstupu. K tomu se používá objekt `sensor`, který nám umožní přístup ke klávesnice a k myši.
- **`isControlActive` -** Výsledek (true nebo false) určuje, zda bude prováděno vlastní testování. Někdy je potřeba ji předefinovat.

Struktura aplikace v systému Visual Works

Struktura aplikace vytvořené v systému Visual Works plně vychází a využívá filozofie architektury MVC. Aplikaci tvoří tři části:

a) doménový model (domain model), v kterém jsou uložena data a který řídí běh aplikace
b) **aplikační model** (application model) - tvořen objekty, v nichž jsou uložena data pro vstup a výstup
c) **objekty GUI** (graphics user interface objects) - zajišťují vstup a výstup

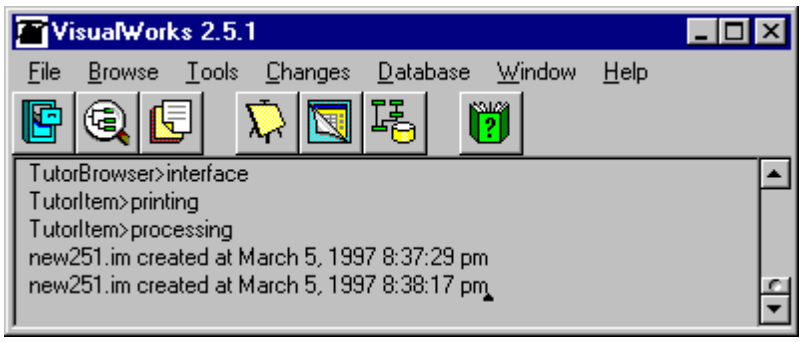


Obr. 9 - Typická struktura aplikace v systému Visual Works

Z obrázku je vidět, že v architektuře MVC objektů uživatelského rozhraní se jako model vyskytuje instance třídy `ValueHolder` (příp. třídy odvozené z třídy `ValueModel`). Třída `ValueHolder` je potomkem třídy `ValueModel`. Tato třída implementuje jednak zprávu `value`, která vrací jako výsledek data uložená ve valueModelu a dále pak zprávu `value:`, která slouží ke změně dat valueModelu. Zpráva `value:` také způsobí vyvolání zprávy `changed:` a tak jsou všechny závislé objekty informovány o změně dat. Zpráva `value`, resp. `value:` je zde vlastně aspect, resp. change zprávou architektury MVC.

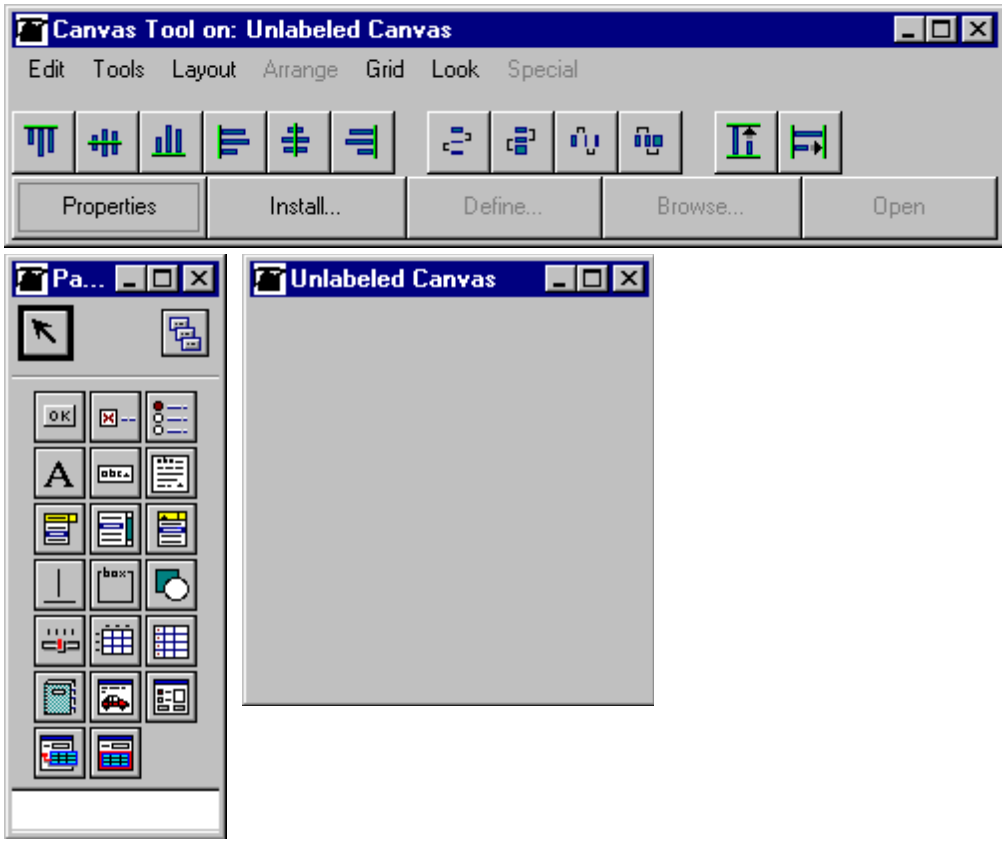
Tvorba aplikace v systému Visual Works

Základní okno systému Visual Works vypadá následovně:



Obr. 10 - Základní okno systému Visual Works

Pro tvorbu nové aplikace slouží volba menu Tools/New Canvas nebo ikona představující malířské plátno. Canvasem je označováno jedno okno aplikace. Po vyvolání této volby se objeví tři další okna - okno s nástroji, paleta vizuálních komponent a nový prázdný canvas.



Obr. 11 - Okno s nástroji, paleta vizuálních komponent a nové okno aplikace

Nyní si vysvětlíme funkci pěti základních tlačítek v okně nástrojů:

Properties - Slouží k nastavení všech parametrů označené vizuální komponenty a jejího připojení k doménovému modelu.

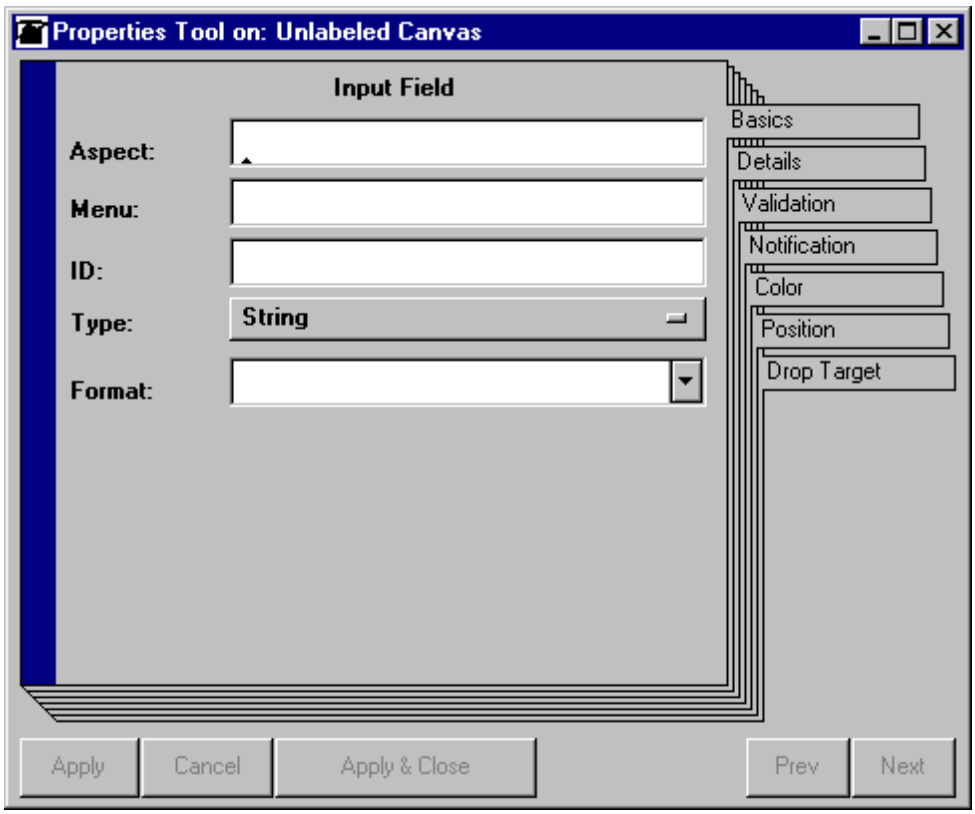
Install - Slouží k vygenerování třídní metody, v které bude uložena informace o navrženém canvasu. Po stisku tohoto tlačítka je nutno zadat příslušnou třídu, třídní metodu (standardně je nabízena metoda `windowSpec`), kategorii třídy a také, zda chceme vygenerovat aplikaci, dialog, formulář nebo databázovou aplikaci. Podle toho se zvolí nadtřída naší nové třídy.

Define - Pomocí tohoto tlačítka lze automaticky vygenerovat instanční proměnné, v kterých jsou uloženy valueHoldery jednotlivých vizuálních komponent (viz obr. 9). Nagenerovány jsou také metody poskytující daný valueHolder jako svůj výsledek. V těchto metodách může být zajištěna i počáteční inicializace příslušné instanční proměnné.

Browse - Toto tlačítko otevře Hierarchy Browser nad naší třídou.

Open - Spustí vygenerovanou aplikaci (aby se projevily všechny změny, které jsme provedli, je nejprve nutné použít Install)

Nyní k popisu vlastního návrhu aplikace. Nejdříve je třeba umístit potřebné vizuální komponenty na plochu. To provedeme tak, že kliknutím do palety vybereme požadovanou komponentu, poté klikneme na canvas a komponenta se zobrazí. Dalším kliknutím ji umístíme. Nyní musíme nastavit vlastnosti komponenty. To se provede tak, že jí kliknutím vybereme a zvolíme *Properties*. Pokud zvolíme *Properties* např. pro vizuální komponentu *Input Field*, otevře se nám následující okno:



Obr. 12 - *Properties* okno vizuální komponenty Input Field

Okno obsahuje několik karet, po otevření je aktivní karta *Basics*, která slouží především pro připojení komponenty do systému. Tato karta je samozřejmě pro různé komponenty různá, z hlediska funkčnosti je však třeba u většiny komponent vyplnit jen

položku *aspect*, příp. *action*. Do položky *aspect* je nutno zadat symbol, který určuje zprávu, pomocí níž je doménový model dotazován na valueHolder (příp. instanci jiné podtřídy `ValueModel`) dané vizuální komponenty. Je to přesně ta zpráva, která byla zmiňována při vysvětlování funkce tlačítka *Define*. Jedná se tedy o jinou aspect zprávu, než o které jsme se bavili v souvislosti s architekturou MVC !

U některých komponent (např. u tlačítka - Action Button) se nezadává symbol určující aspect, ale action zprávu. Tato zpráva je doménovému modelu zaslána při aktivaci komponenty (např. při stisku tlačítka).

Další karty v okně bývají zařazeny podle toho, zda mají u dané komponenty smysl. Většinou je přítomna karta *Details* (lze zadat např. font, stav po inicializaci, orámování, průhlednost, ..), *Color* (nastavení barev), *Position* (specifikace umístění v okně). Na kartách *Notification* a *Validation* můžeme zadat tři zprávy - entry, change a exit. Jde o zprávy, které jsou v souvislosti s akcemi plynoucích z jejich názvu zasílány doménovému modelu. Zprávy specifikované v okně *Validation* musí vracet `true`, jestliže jsou příslušné akce dovoleny, jinak vrací `false`. *Notification* zprávy jsou zaslány, až když příslušná akce nastane. Stejným způsobem jsou zasílány i čtyři zprávy (entry, over, exit, drop) zadávané na poslední běžné kartě, kterou je karta *Drop target*.

Po specifikaci vlastností vizuálních komponent již můžeme vygenerovat kód odpovídající našemu návrhu pomocí volby *Install*. Tlačítkem *Define* lze automaticky vygenerovat aspect metody a nadefinovat jim odpovídající instanční proměnné. Poté již můžeme naši aplikaci spustit stisknutím tlačítka *Open*.

Nástroje pro vizuální návrh

V okně s nástroji můžeme nalézt mnoho užitečných pomůcek, které nám velice usnadní vizuální návrh aplikace. Ty nejdůležitější jsou lehce přístupné pomocí lišty s ikonami.

Obr. 13 - Lišta s ikonami v okně nástrojů

V první skupině jsou nástroje, pomocí nichž lze zarovnat komponenty tak, jak je na ikonách znázorněno. V prvním případě jsou na vodorovnou přímku zarovnány horní okraje komponent, v druhém a ve třetím pak středy a dolní okraje. Čtvrtou až šestou ikonu jsou na svislou přímku zarovnány levé okraje, středy a pravé okraje. Zarovnání se provádí tak, že postupně označíme myší (držíme přitom klávesu Shift) všechny ikony, které chceme srovnat a pak stiskneme ikonu žádaného zarovnání. To se provede podle ikony, která byla označena jako první.

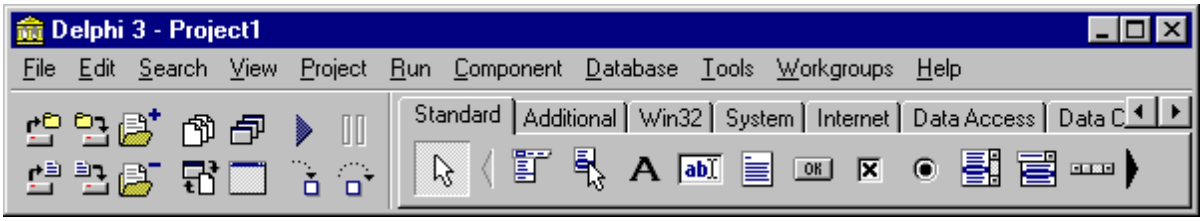
Další skupina čtyř ikon slouží k nastavení vzdáleností. První resp. třetí ikona umožňuje nastavit stejnou vzdálenost mezi komponentami ve vodorovném resp.svislém směru, druhá resp.čtvrtá umožňuje nastavit stejnou vzdálenost mezi středy komponent. Poslední dvě ikony umožní nastavit stejný rozměr komponent opět v obou směrech.

V menu pak můžeme nalézt ještě mnoho dalších nástrojů (operace s rastrem, seskupování, Image Editor, Menu Editor, ...).

Tvorba aplikace v systému Delphi

Systém Delphi si během krátké doby od svého vzniku dokázal získat velký počet příznivců. Pokud se budeme ptát, jak je to možné, nabízí se hned celá řada odpovědí. Pomocí Delphi lze opravdu rychle a snadno napsat běžné aplikace. Velice šťastným krokem při vývoji systému bylo zvolení Borland Object Pascalu výchozím jazykem, protože tento jazyk je dnes de facto standardem objektového Pascalu. A v neposledním řadě to byla tradice firmy Borland v oblasti vývoje kompilátorů, která přispěla k velkému úspěchu tohoto produktu. Proto nyní věnujeme tuto kapitolu popisu práce s tímto systémem.

Po otevření nového projektu se nám na obrazovce objeví čtyři okna. Prvním z nich je základní okno systému .



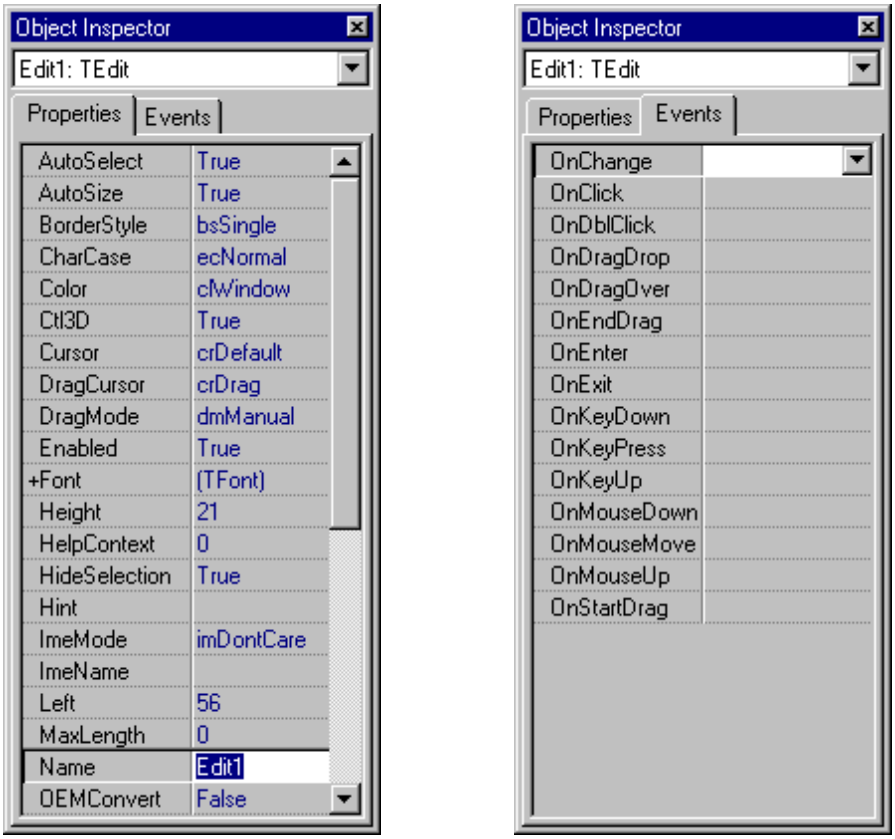
Obr. 14 - Základní okno systému Delphi 3

V pravé části okna se nachází paleta s vizuálními komponentami. Pro větší přehlednost se komponenty podle své funkce a svých vlastností nachází na různých kartách. V levé části okna můžeme nalézt ikony, které reprezentují nejběžnější a nejpoužívanější operace. Jedná se o zavření a otevření souboru nebo projektu, přidání či odebrání souboru z projektu, spuštění nebo přerušení (pause) běhu aplikace, krokování se zanořováním se do funkcí a bez zanořování. Zbývající čtyři ikony umožňují otevírat, případně se přepínat mezi dvěma dalšími okny - novým prázdným formulářem a oknem se zdrojovým kódem tohoto formuláře. Tuto lištu s ikonami i lištu s vizuálními komponentami si uživatel může libovolně upravit (přidat, odebrat ikonu metodou *drag and drop*).

Jak jsme se již zmínili, dalším oknem je formulář (představuje jedno okno aplikace) a jemu odpovídají okno se zdrojovým kódem jednotky tohoto formuláře. Delphi tedy generuje zdrojový text tak, že každému formuláři odpovídá jedna jednotka (unit) jazyka Pascal uložená v jednom souboru na disku.

Posledním oknem, které se po otevření projektu objeví, je okno nástroje *Object Inspector*. O jeho úloze bude řeč v následujících odstavcích.

Nyní k popisu vlastního návrhu. Vizuální komponentu vložíme na formulář tak, že ji vybereme kliknutím na liště a umístíme kliknutím na formulář. Dalším krokem je specifikace vlastností označené komponenty (označení komponenty se provede kliknutím na ni) v okně *Object Inspector*. Příklad provedeme opět nad komponentou editačního boxu.



Obr. 15 - Dvě karty okna Object Inspector

Okno Object Browser obsahuje dvě karty. Na kartě *Properties* se specifikují vlastnosti komponenty a její začlenění do systému. Tyto vlastnosti jsou opět pro různé komponenty různé, společná je vlastnost *Name*, která určuje jméno proměnné ve zdrojovém kódu, pod kterým je komponenta vložena do objektu formuláře. Blíže však v následující kapitole o struktuře aplikace. Druhá karta *Events* slouží ke specifikaci metod formuláře, které se vyvolají při příslušných akcích (událostech). Při pohledu na obrázek je vidět, že je možno navíc oproti systému Visual Works ošetřit zprávy související s pohybem myši a stisknutím klávesy. Ošetření se provádí tak, že zadáme do položky název metody a dvakrát klikneme. Delphi v tom okamžiku přidá danou metodu a přepne nás do okna se zdrojovým kódem, kde je již tato metoda s prázdným tělem přichystaná pro dopsání.

Po zadání všech vlastností a doplnění příslušných metod již můžeme pomocí volby Run (na liště nebo v menu) aplikaci přeložit a spustit.

Struktura aplikace v Delphi a její srovnání se strukturou aplikace ve Visual Works

Po přidání našeho editačního boxu změní Delphi definici třídy formuláře následujícím způsobem (v *Object Inspectoru* byl box pojmenován jako Edit1, formulář jako Form1):

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Ze zdrojového kódu je vidět, že vizuální komponenta byla vložena do třídy TForm1 (byla vytvořena vazba skládání). Nyní lze (v metodách třídy TForm1) pracovat s editačním boxem následujícím způsobem:

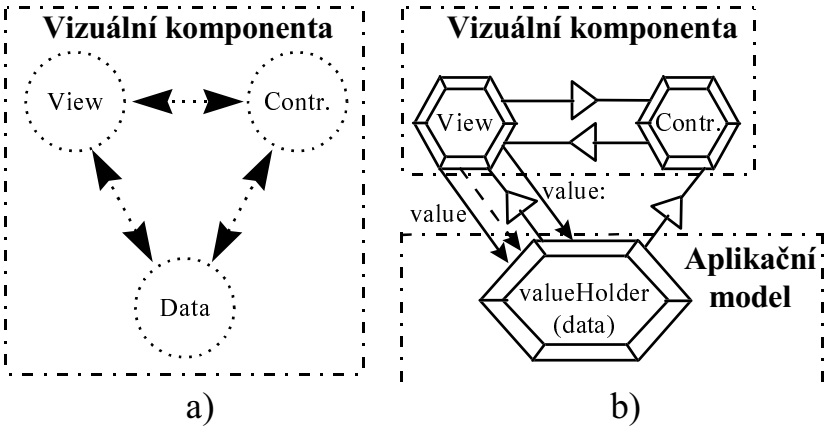
Změna barvy se provede takto:

```
Edit1.Color := clRed
```

Změna zobrazovaného textu se provede takto:

```
Edit1.Text := 'Nový text'
```

Nyní si dobře uvědomme, co jsme vlastně udělali. Změnili jsme barvu, tedy vlastnost view (budeme nyní i pro odpovídající části komponenty Delphi používat terminologii běžnou ve Smalltalku). Pak jsme ale úplně stejně změnili text boxu, tedy data komponenty ! Z toho ovšem plyne, že také **data jsou součástí vizuální komponenty !!!** Vizuální komponentu v Delphi tedy tvoří view, controller (protože komponenta umí reagovat na události myši a klávesnice) a data. Naproti tomu vizuální komponentu systému Visual Works tvoří jen view a controller, data (valueHoldery) jsou součástí aplikačního modelu !



Obr. 16 - Vizuální komponenta a) v Deplhi b) ve Visual Works

Šipky na obrázku vizuální komponenty systému Delphi naznačují blíže nespecifikovanou vzájemnou interakci.

Tato uspořádání ovšem vedou k velice závažným závěrům. Ve Smalltalku lze vytvořit fungující aplikaci bez grafického uživatelského rozhraní (komunikující s okolím např. pomocí zpráv) a k ní později připojit (i za chodu) libovolný počet vizuálních komponent, aniž bychom museli v této aplikaci cokoliv měnit. V Delphi ovšem musíme, má-li být naše aplikace v budoucnu vizuální, komponenty začlenit do programu, aby jsme měli kde uchovávat data. Pokud bychom chtěli danou komponentu nahradit jinou, musí tato umět všechny zprávy původní komponenty a musí u ní být i stejný přístup k datům. Vše se samozřejmě musí znovu překompilovat a spustit.

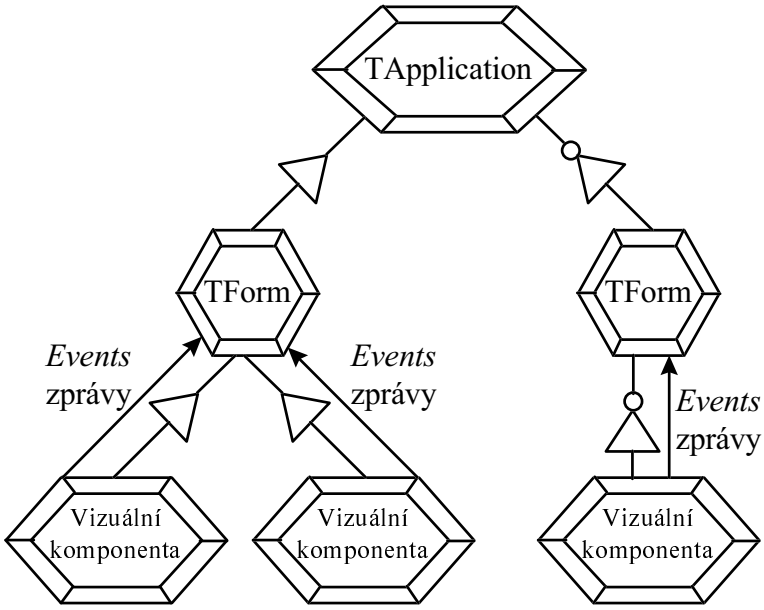
Další rozdíly zjistíme, pokud si uvědomíme, že vizuální komponenta (a s ní i data) byla vložena do formuláře, který reprezentuje okno aplikace. To znamená, že **datový model není tvořen podle svého vzoru v reálném světě, ale podle rozmístění komponent do oken** !!!

Tuto skutečnost potvrzuje i to, že všechny metody odpovídající akcím (events) komponenty jsou generovány jako metody formuláře. Vyvolání určité události u komponenty tedy znamená poslání zprávy formuláři, v kterém je tato komponenta umístěna. Tuto

skutečnost ilustruje následují část zdrojového kódu odpovídající události `OnChange` našeho editačního boxu:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin

end;
```

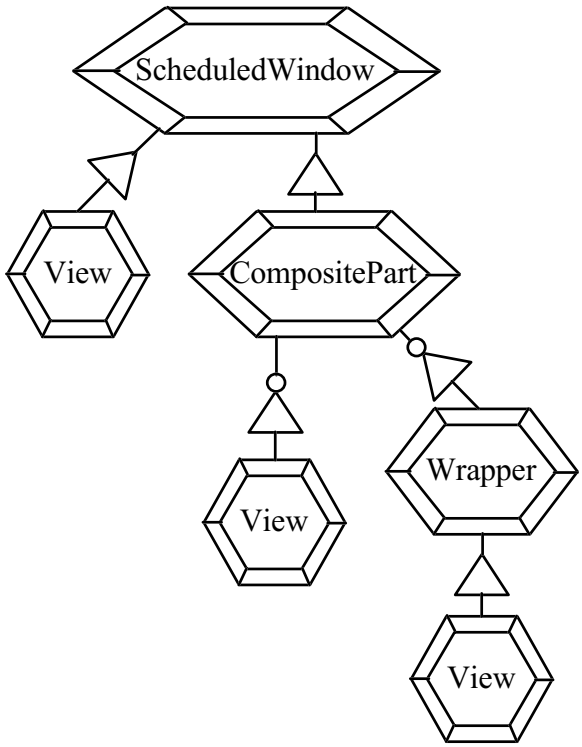


Obr. 17 - Struktura aplikace v systému Delphi

Vložení formuláře se provádí metodou `CreateForm` třídy `TApplication`. Tato akce se provádí v hlavním programu, který je uložen v souboru s příponou *dpr* (Delphi Project).

Úvod do problematiky vizuálních komponent

Jak vytvořit vizuální prvek - view, připojit jej ke spolupracujícímu modelu a controlleru, bylo popsáno v kapitole pojednávající o architektuře MVC. Znalosti nabité v této kapitole nám postačují k tomu, abychom si mohli udělat libovolnou komponentu splňující nároky architektury MVC. V této a následujících kapitolách se však budeme snažit o něco navíc - udělat z této komponenty vizuální komponentu systému Visual Works, která bude nabízena v paletě vizuálních komponent, kterou budeme moci pomocí myši umístit na canvas a kterou lze připojit a parametrizovat v okně *Properties*. Tím se ale otvírá široký okruh problémů spojených se začleněním view do struktur systému. Pokud bychom sami „ručně" vkládali view do okna, bylo by to zřejmě jedním z následujících typických způsobů:

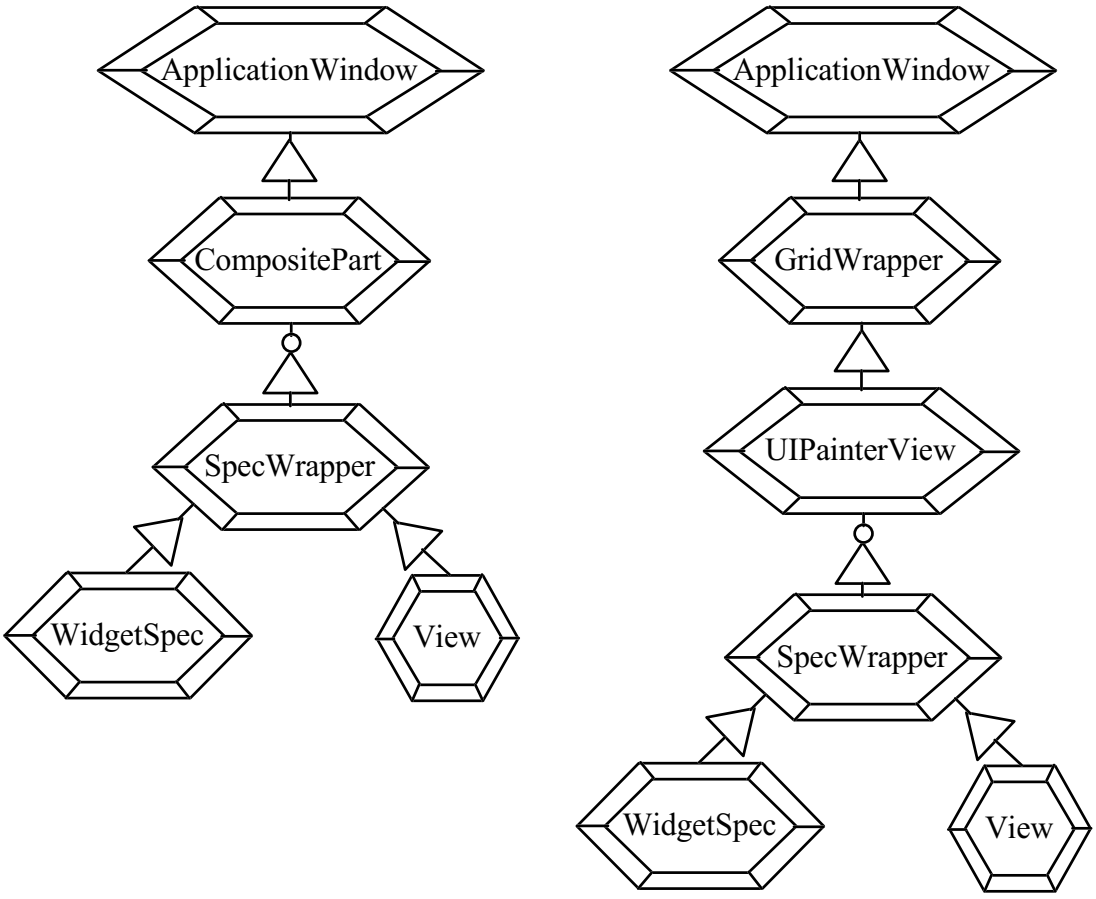


Obr. 18 - Různé způsoby vložení view do okna

Z obrázku je vidět, že do okna lze vložit jen jeden vizuální prvek. Může jím být už přímo view, ale častěji to bývá tzv. kontejner (instance třídy CompositePart) , pomocí něhož lze v jednom okně zobrazit libovolný počet view. View nemusí být do kontejneru, příp. okna vkládána přímo, ale mohou být „zabalena" do instancí třídy `Wrapper`. Tato třída a její podtřídy umožňují definovat ohraničení view (např. kvůli ořezávání), zarámovat view, přidat posunovací lišty apod. Je třeba ještě dodat, že na tomto obrázku (a všech následujících obrázcích týkající se tohoto problému), není kvůli přehlednosti zakreslena vazba skládání opačným směrem. Instance třídy `VisualPart` (a instance všech jejich podtříd, tedy i `View, Wrapper, CompositePart)` obsahují proměnnou `container`, v které je udržován rodičovský (parent) objekt, do kterého je tato instance vložena.

Při používání vizuálních komponent má aplikace vygenerovaná systémem Visual Works sice podobnou, ale přece jen trochu jinou hierarchii view v okně. Tato hierarchie je navíc jiná při běhu aplikace (run-time) a při návrhu (design-time). Je zřejmě jasné, že při návrhu bude situace složitější, protože view musí nejen plnit svou funkci, tedy zobrazovat, ale

musí být zajištěna funkce nástrojů pro vizuální tvorbu a také vazba do okna *Properties*, pomocí kterého dané view nastavujeme.



Obr. 19 - Hierarchie view ve Visual Works a) při běhu aplikace b) při návrhu aplikace

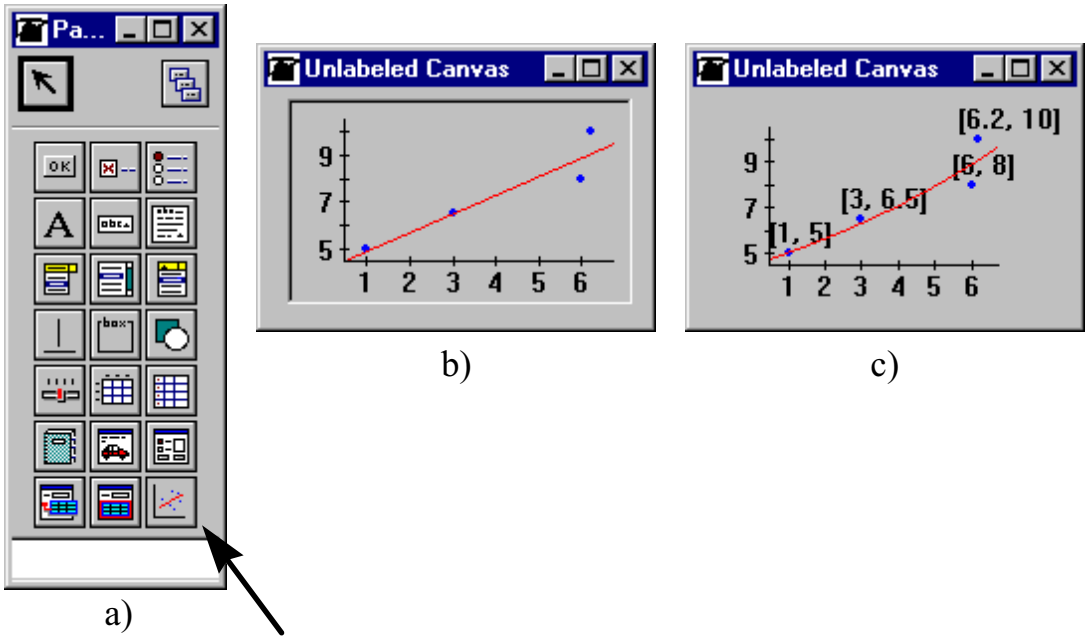
Nejprve si všimneme hierarchie při běhu aplikace. Je vidět, že Visual Works používá místo třídy `ScheduledWindow` její podtřídu `ApplicationWindow`, která má rozšíření související právě s tvorbou uživatelského rozhraní. Daleko podstatnější změna se nachází v dolní části obrázku - view je vloženo do objektu `SpecWrapper` a spolu s ním instance třídy **`WidgetSpec`**. Pro nás se tato třída a její příp. podtřídy stanou v dalším textu středem pozornosti, protože umět napsat podtřídu této třídy odpovídající danému view (komponentě MVC) znamená umět vytvořit vizuální komponentu systému Visual Works.

Nyní ale ještě k případu b). Instance třídy `GridWrapper` zajišťuje při návrhu správnou funkci rastru. Objekty `SpecWrapper` jsou vkládány do instance třídy `UIPainterView`, která zde plní úlohu své nadtřídy `CompositePart`. Navíc ale slouží jako view objektu **`UIPainter`**, který řídí celý průběh návrhu.

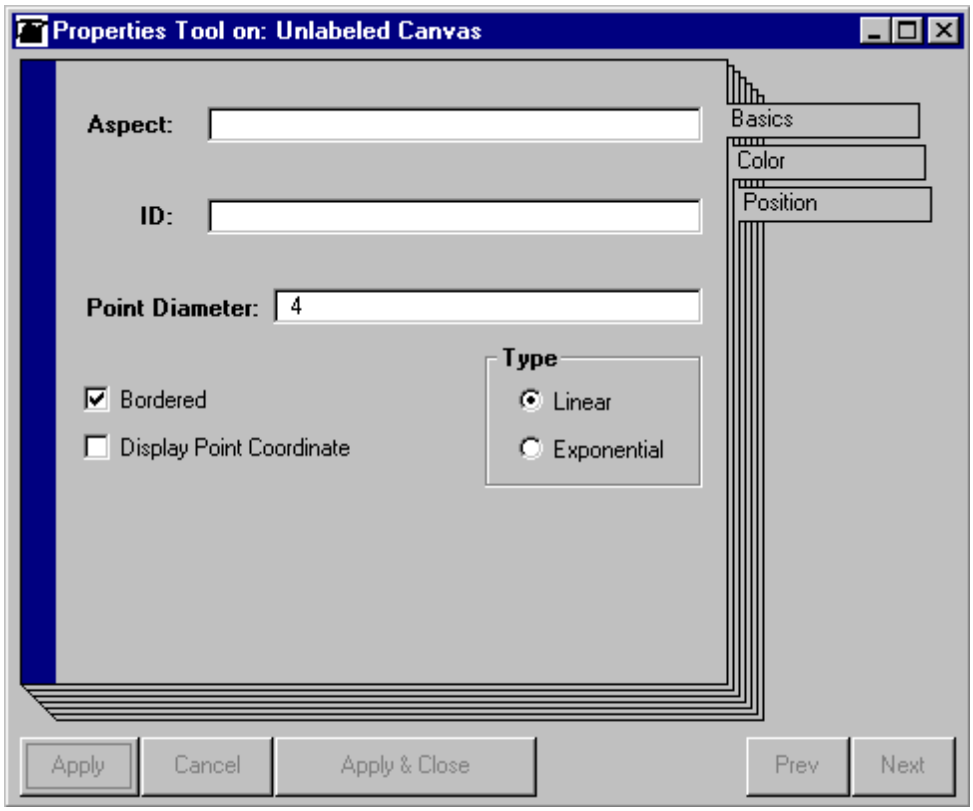
Závěrem nutno dodat, že view nebývají do objektu `SpecWrapper` vkládány přímo, ale bývají ještě „zabaleny“ do instancí tříd `BorderedWrapper` nebo `BoundedWrapper` podle toho, zda mají nebo nemají být zarámovány.

Příklad vizuální komponenty

Jedním z výsledků této práce je implementace vizuální komponenty systému Visual Works (jedná se o třídy RegressFceView a RegressFceSpec) a příkladu použití této komponenty (třída RegressFceTest). V dalších odstavcích se budeme při objasňování problematiky na tento příklad odkazovat, proto si jej nyní trochu přiblížíme.



Obr. 20 - a) Ikona vizuální komponenty b), c) různě nastavená vizuální komponenta



Obr. 21 - Karta *Basics* okna *Properties*

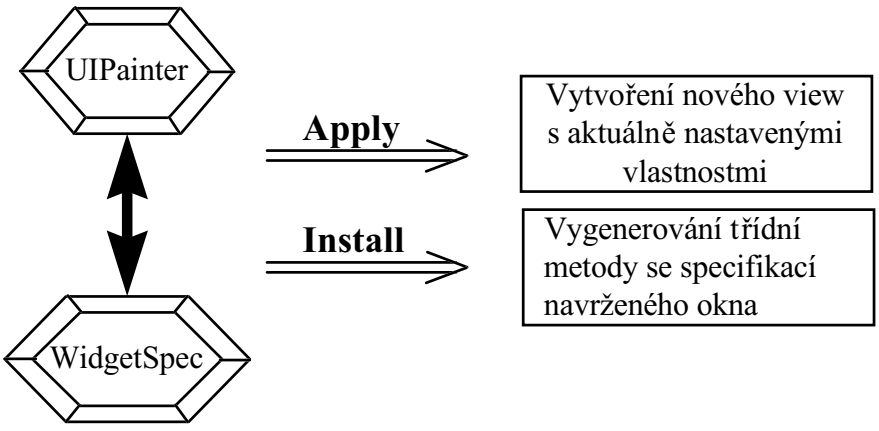
Komponenta umí zobrazit lineární nebo exponenciální regresivní funkci, vstupem (modelem) je kolekce bodů (nebo přesněji `ValueHolder` obsahující danou kolekci bodů). Po zavedení do systému přibude na paletě s vizuálními komponentami ikona reprezentující naši novou komponentu (viz. obr. 20a). Na obrázku 20b je její vzhled odpovídající nastavení karty *Basics* v okně *Properties* na obr. 21. Na obr. 20c je vzhled vizuální komponenty, jestliže přepneme přepínače *Bordered*, *Display Point Coordinate* a *Type*. Přepínač (Check Box) *Bordered* určuje, zda má být komponenta zarámovaná, *Display Point Coordinate* udává, jestli se mají zobrazovat souřadnice bodů a volbou *Type* (2x Radio Button) zvolíme, zda se jedná o lineární či exponenciální regresivní funkci. Poslední volbou speciální pro tuto komponentu je *Point Diameter*, která určuje velikost zobrazovaných bodů. Z obr. 21 je ještě vidět, že do okna jsou zařazeny další dvě standardní karty - *Color* a *Position*.

Stejně jako při popisu příkladu komponenty MVC je i zde nutno upozornit, že způsobů, jak vytvořit vizuální komponentu, je mnoho a zde bude předveden jeden z nich. Hlavní důraz bude kladen na vysvětlení principu a těch vlastností, které by měla splňovat každá vizuální komponenta.

Třída WidgetSpec a její podtřídy

Jak již bylo řečeno, tato třída (příp. její podtřídy) je při tvorbě vizuálních komponent jednou z nejdůležitějších. Popišme si tedy nejdřív její úlohu a následně z toho plynoucí strukturu.

Při „ručním“ vytváření a začleňování view (komponenty MVC) do programu zavoláme třídní metodu, která má za úkol vytvořit danou instanci. Parametry této zprávy bývá jednak model a také další vlastnosti (properties) dané instance. Musí se stanovit umístění v okně, view příp. zarámovat, přidat lišty apod. Při vizuálním návrhu je v okamžiku vytváření view také třeba znát všechny tyto údaje. Proto je nutno všechny vlastnosti a způsob připojení vhodně zadat, uložit do příslušného objektu a po ukončení návrhu vygenerovat specifikaci navrhovaného okna. Všechny tyto akce řídí objekt `UIPainter`, který uvedené údaje ukládá do instance některé z podtříd třídy `WidgetSpec`. `UIPainter` si ale také při prvním vytvoření view na začátku návrhu z této instance vyzvedne přednastavené (default) vlastnosti view.



Obr. 22 - Vizuální návrh aplikace

Na obrázku je znázorněno, co je předmětem našeho zájmu. Jde především o vzájemnou interakci mezi objekty `UIPainter` a `WidgetSpec`. Dále je třeba specifikovat, co se má stát při potvrzení změn stisknutím tlačítka *Apply* v okně *Properties*. Poslední akce - vygenerování metody se specifikací okna stisknutím tlačítka Install se nás, z hlediska návrhu vizuální komponenty, již netýká, vše je zajištěno tím, že naše třída bude podtřídou třídy WidgetSpec.

Nyní si již můžeme vyjmenovat, co všechno musí naše nová podtřída (`RegressFceSpec`) třídy `WidgetSpec` obsahovat:

1. Instanční proměnné, v kterých jsou uloženy vlastnosti komponenty a metody zajišťující přístup k těmto proměnným.
2. Inicializaci instančních proměnných z bodu 1.
3. Specifikaci vizuálního návrhu jednotlivých karet okna *Properties*
4. Třídní metody zajišťující spolupráci s objektem `UIPainter`.
5. Metody zajišťující akce spojené s akceptováním (*Apply*) daného nastavení.
6. Specifikaci ikony zastupují komponentu v paletě vizuálních komponent.
7. Metodu zajištující přidání ikony z bodu 6 do palety vizuálních komponent.

Implementace vizuální komponenty

V této kapitole bude popsána implementace jednotlivých bodů z předchozí kapitoly. Nejdříve je ovšem nutno vytvořit novou třídu `RegressFceSpec` jako podtřídu třídy `WidgetSpec`.

1. Přidáme instanční proměnné reprezentující vlastnosti komponenty. V našem případě jde o proměnné `pointDiameter`, `displayPointCoordinate` a `type`. Dále vytvoříme metody (public interface methods), které umožní zpřístupnit a nastavit danou proměnnou. Jako příklad si uvedeme metody pro proměnou `type`
Metoda pro zpřístupnění:

```
type
     ^type
```

Metoda pro nastavení:

```
type: aVal
     type := aVal
```

Instanční proměnnou `bordered`, v které by byla uložena informace o orámování, a příslušné metody nedefinujeme. Důvody, které nás k tomu vedou, budou objasněny v následujícím bodu.

2. Inicializace proměnných reprezentující jednotlivé vlastnosti se provede v metodě `initialize`. Může vypadat následujícím způsobem:

```
initialize
     super initialize.
     pointDiameter := 4.
     type := #linear.
     displayPointCoordinate := false
```

Některé vlastnosti nabývající dvou stavu (boolean) , jako např. průhlednost, ale také orámování (bordered), jsou natolik běžné, že již byly implementovány ve třídě `NamedSpec`, která je nadtřídou třídy `WidgetSpec`. Každé vlastnosti odpovídá jeden bit hodnoty uložené v instanční proměnné `flags`. Vlastnost orámování je reprezentovaná bitem 3 (pokud číslujeme bity od nuly), příslušné metody se jmenují `hasBorder:` a `hasBorder` . Pokud chceme některou z těchto vlastností zinicializovat, musíme předefinovat metodu `defaultFlags`, která vrací inicializační hodnotu instanční proměnné `flags`. Nastavení orámování po zapnutí odpovídá této metodě:

```
defaultFlags
     ^8
```

příp. lze pro větší názornost použít binárního zápisu hodnoty:

```
defaultFlags
     ^2r1000
```

3. Každou nestandardní kartu okna *Properties* navrhneme jako zvláštní okno. Při generování nesmíme zapomenout změnit název třídní proměnné ze standardního `windowSpec` na jméno odpovídající vytvářené kartě. To může být libovolné, ale většinou se tvoří tak, že ke jménu karty přidáme „EditSpec“, takže třídní metoda se specifikací karty *Basics* se bude jmenovat `basicsEditSpec`.

V našem příkladu je jedinou nestandardní kartou právě karta *Basics*. Rozmístění editačních boxů je patrné z obr. 21. První dva boxy - aspect a ID patří opět ke standardním a jejich ošetření je již implementováno v třídách `WidgetSpec` a `NamedSpec`. Symboly odpovídající jejich *aspect* zprávám jsou `#model` a `#name`. Do položky *Menu* zadáme u obou editačních boxů symbol `#fieldMenu`. Všechny další editační boxy reprezentují již námi přidané vlastnosti a jako jejich *aspect* položku vyplníme symboly odpovídající patřičným přístupovým metodám, tedy `#pointDiameter`, `#hasBorder`, `#displayPointCoordinate`. U dvou komponent typu Radio Button je *aspect* symbolem `#type`, select položku vyplníme v prvním případě symbolem `#linear`, ve druhém `#exponential`.

4. **Třídní** metody třídy `RegressFceSpec` spolupracující s objektem UIPainter

a) metoda `addBindingsTo:env for:inst channel:aChannel`

Tato metoda slouží k předání informací uložených v instanci třídy `RegressFceSpec` objektu `UIPainter`. `UIPainter` volá tuto metodu vždy, když potřebuje zobrazit uvedené informace, tedy např. při každém překreslení karty *Basics*.
Parametry zprávy jsou následující objekty:

`env` - instance třídy `UIPainter`
`inst` - instance třídy `RegressFceSpec`
`aChannel` - instance třídy `ValueHolder` držící objekt `RegressFceSpec`

V těle metody se musí zavolat stejná metody nadtřídy, dále pak pro každou proměnnou z bodu 1 je nutno zajistit její předání např. pomocí následujícího výrazu (uveden pro proměnnou `type`):

```
env at: #type put: (self
                adapt: inst
                forAspect: #type
                channel: aChannel)
```

b) Metoda `componentName`

Vrací název komponenty, který se zobrazí, když na ni klikneme v paletě.

```
componentName
     ^'Regression'
```

c) Metoda `placementExtentBlock`

Metoda obsahuje blok s jedním parametrem, tělem bloku je bod, jehož souřadnice *x* a *y* udávají příslušné inicializační rozměry vizuální komponenty.

```
placementExtentBlock
     ^[:bldr | 180@110]
```

d) Metoda `specGenerationBlock`

Metoda obsahuje blok, který se podílí na vytvoření instance této třídy. Blok má dva parametry - instanci třídy `UIPainterController` a instanci třídy `Point`, která určuje umístění komponenty. Metoda může vypadat např. takto:

```
specGenerationBlock
     ^[:ctrlr :point | RegressFceSpec new layout:(
          (ctrlr gridPoint: point)
               extent: (ctrlr currentMode value class
                    placementExtentFor:RegressFceSpec
                    inBuilder: ctrlr builder))]
```

e) Metoda `slices`

Výsledkem této metody je pole karet, které se mají zobrazovat v okně *Properties*. Vložení námi vytvořené karty *Basics* a standardních karet *Color* a *Position* do tohoto okna zajišťuje následující metoda:

```
slices
     ^#(  (Basics basicsEditSpec)
          (Color propSpec ColorToolModel)
          (Position propSpec PositionToolModel) )
```

5. Při akceptování změn v okně *Properties* (nebo přesněji vždy při návrhu, když je třeba vytvořit view, tedy i při prvním umístění komponenty do okna) se vyvolá metoda `dispatchTo:with:`. Může vypadat následujícím způsobem:

```
dispatchTo: policy with: builder
     ^policy regressFce: self into: builder
```

První parametr `policy` představuje instanci třídy `UILookPolicy`, druhý parametr `builder` je instancí třídy `UIBuilder`. Tato třída řídí vytváření uživatelského rozhraní podle specifikace (`UISpecification`, třída `WidgetSpec` a tedy i naše

RegressFceSpec je jejím potomkem) vytvořené třídou UIPainter. K vytvoření a inicializaci view využívá UIBuilder právě služeb třídy UILookPolicy. Je vidět, že metoda dispatchTo:with: volá metodu třídy UILookPolicy, kterou je regressFce:into:. Tato metoda provádí vlastní vytvoření a začlenění view a musíme ji do této třídy (do protokolu metod *building*) přidat. Jak je vidět z kódu metody dispatchTo:with:, parametry jsou instance naší třídy RegressFceSpec a UIBuilder. Protože jde o poměrně dlouhou metodu, budeme ji komentovat po částech:

```
regressFce: regressFceSpec into: builder
     | component mdl |
     mdl := regressFceSpec modelInBuilder: builder.
     component := RegressFceView new model: mdl.
     component pointDiameter: regressFceSpec pointDiameter.
     component type: regressFceSpec type.
     component displayPointCoordinate:regressFceSpec
displayPointCoordinate.
     builder component: component.
```

V této první části je patrné vytvoření jak view (nebo spíš celé komponenty MVC, tedy i modelu, příp. controlleru), tak i předání vlastností z instance třídy RegressFceSpec novému view. Je samozřejmě nutné implementovat u view metody, pomocí nichž se toto předání provádí. Na závěr je celá komponenta MVC poskytnuta objektu UIBuilder.

```
     self
          setDispatcherOf: component
          fromSpec: regressFceSpec
          builder: builder.
     regressFceSpec hasBorder
          ifTrue: [builder wrapWith: (self borderedWrapperFor:
regressFceSpec)]
          ifFalse: [builder wrapWith: (self simpleWrapperFor:
regressFceSpec)].
     builder applyLayout: regressFceSpec layout.
     builder wrapWith: (self simpleWidgetWrapperOn: builder
spec: regressFceSpec)
```

První výraz této části zajistí vytvoření, inicializaci a napojení objektu UIDispatcher. Úkolem tohoto objektu je zajišťovat překlad událostí na odpovídající zprávy, které jsou pak komponentě zasílány. Dále je postaráno jednak o předání umístění komponenty a také především o „zabalení“ komponenty do objektů typu Wrapper, tak jak to bylo zmíněno v souvislosti s obrázkem 19.

6. Ikona je vytvořena v barevném a černobílém provedení pomocí nástroje *Image Editor*, který lze vyvolat z okna nástrojů. Jako výsledek jsou vygenerovány příslušné dvě třídní metody paletteIcon a paletteMonoIcon.

7. Začlenění ikony do palety vizuálních komponent se provádí při inicializaci třídy. Třídní metoda initialize vypadá následovně:

```
initialize
     (UIPalette activeSpecsList includes: #RegressFceSpec)
```

```
ifFalse: [UIPalette activeSpecsList add: #RegressFceSpec]
```

Předchozích sedm bodů dává návod na vytvoření podtřídy třídy `WidgetSpec`. K vytvoření celé vizuální komponenty je však třeba mít ještě komponentu MVC. Vytvoření této části bylo popsáno v kapitole pojednávající o architektuře MVC, zbývá tedy jen upozornit, že je třeba přidat metody, pomocí nichž jsou view předávány jeho parametry. Jsou to ty metody, které byly v této souvislosti zmíněny v bodě 5.

Závěr

V první části této práce byly objasněny principy architektury MVC a popsány vazby mezi jejími základními částmi - modelem, view a controllerem. U všech těchto prvků byly také zdůrazněny a vysvětleny metody, které je třeba vytvořit při jejich implementaci. Dalším krokem bylo objasnění struktury aplikace systému Visual Works a popsán návrh této aplikace. Totéž bylo provedeno pro systém Delphi 3. Srovnáním těchto dvou systémů jsem dospěli k následujícím závěrům:

Způsob práce při vizuálním návrhu se příliš neliší, oba systémy poskytují podobné nástroje pro rozmisťování, zarovnávání a další manipulaci s vizuálními komponentami. Velice dramatický rozdíl je však ve struktuře výsledné aplikace. U aplikace vytvořené ve Visual Works dochází především díky využití architektury MVC k oddělení datové a zobrazovací části. Tím je umožněn objektový návrh datového modelu přesně podle všech zásad objektového programování. Naproti tomu Delphi podporuje návrh takové aplikace, která téměř nerozlišuje datovou a zobrazovací část. Součástí většiny základních vizuálních komponent jsou jejich data. Aplikace je generovaná tak, že komponenty (se svými daty) z jednoho formuláře (okna) jsou vloženy do objektu tohoto formuláře. Dochází tak k vytvoření datového modelu, který nerespektuje reálnou skutečnost, ale rozmístění dat při jejich zobrazení. Systém Delphi tedy jen používá objekty, ale negeneruje objektově orientovanou aplikaci.

Výhody objektově orientované aplikace se projeví již při jejím vytváření. Aplikaci je možno spustit a testovat hned po vytvoření její datové části, která je na zobrazovací nezávislá. K této fungující aplikaci je možno (i za chodu, bez jakéhokoliv kompilování) připojit libovolný počet vizuálních prvků. Dobře navržené objekty mohou být základem jiné aplikace a z toho plyne velká znuvupoužitelnost a větší produktivita. Objektově orientovaný přístup však přináší především jiný způsob myšlení a pohledu na věc, což může být pro mnohé velkou překážkou.

Druhá část této práce obsahuje rozbor problematiky vizuálních komponent systému Visual Works. Cílem této části je popsat jejich začlenění do systému a především způsob jejich implementace. Ta byla předvedena na příkladu vizuální komponenty, která umí zobrazit lineární a exponenciální regresivní funkci pro danou množinu bodů. Při popisu implementace komponenty bylo ukázáno, jak umožnit zadání nových parametrů komponenty (např. zda zobrazit souřadnice bodů), ale také jak začlenit běžné vlastnosti získané prostřednictvím dědičnosti (např. zda má být komponenta orámovaná, zadávání barev, pozice apod.). Celý popis je veden tak, aby nejen objasnil funkci výše uvedené komponenty, ale aby mohl sloužit jako návod pro tvorbu libovolné vizuální komponenty.

Použitá literatura

[1] LaLonde W., Pugh J.; Inside Smalltalk; Prentice - Hall, 1990.

[2] Manuál k systému Visual Works.

[3] Osier D., Grobman S., Batson S. - Teach yourself Delphi 2 in 21 days,

Sams Publishing, 1996

[4] Cantú M.- Mistrovství v Delphi, Computer Press, 1995

[5] Richta K., Sochor J. - Softwarové inženýrství 1, Vydavatelství ČVUT, Praha 1996

[6] Hindls R., Kaňoková J., Novák I. - Statistické metody, VŠE Praha, 1995

[7] Jarošová E. - Statistika B, VŠE Praha, 1995

Příloha - Výpis zdrojového kódu příkladu vizuální komponenty

```
ApplicationModel subclass: #RegressFceTest
	instanceVariableNames: 'pointArray x y arrayIndex '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'UI-RegressFce'!


!RegressFceTest methodsFor: 'initialize-release'!

initialize
	pointArray := ValueHolder with: (OrderedCollection
					with: 1 @ 5
					with: 3 @ 7
					with: 6 @ 6
					with: 7 @ 9).
	arrayIndex := 1 asValue.
	arrayIndex onChangeSend: #arrayIndexChanged to: self! !

!RegressFceTest methodsFor: 'aspects'!

arrayIndex
	"This method was generated by UIDefiner.  Any edits made here
	may be lost whenever methods are automatically defined."

	^arrayIndex!

pointArray
	"This method was generated by UIDefiner.  Any edits made here
	may be lost whenever methods are automatically defined.  The
	initialization provided below may have been preempted by an
	initialize method."

	^pointArray!

x
	"This method was generated by UIDefiner. Any edits made here
	may be lost whenever methods are automatically defined. The
	initialization provided below may have been preempted by an
	initialize method."

	^x isNil
		ifTrue: [x := (self pointArray value at: self arrayIndex value) x asValue]
		ifFalse: [x]!

y
	"This method was generated by UIDefiner. Any edits made here
	may be lost whenever methods are automatically defined. The
	initialization provided below may have been preempted by an
	initialize method."

	^y isNil
		ifTrue: [y := (self pointArray value at: self arrayIndex value) y asValue]
		ifFalse: [y]! !

!RegressFceTest methodsFor: 'actions'!

accept
	| aIdx pArr |
	aIdx := self arrayIndex value.
	pArr := self pointArray value.
	aIdx == (pArr size + 1)
		ifTrue: [pArr add: (Point x: self x value y: self y value)]
		ifFalse: [pArr at: aIdx put: (Point x: self x value y: self y value)].
	self pointArray value: pArr!

arrayIndexChanged
	| aIdx pArr |
	aIdx := self arrayIndex value.
	pArr := self pointArray value.
	(aIdx between: 1 and: pArr size)
		ifTrue:
			[x value: (pArr at: aIdx) x.
			y value: (pArr at: aIdx) y]
		ifFalse: [aIdx == (pArr size + 1)
				ifTrue:
```

```
                                    [x value: 0.
                                    y value: 0]
                            ifFalse: [self arrayIndex value: pArr size + 1]]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

RegressFceTest class
       instanceVariableNames: ''!


!RegressFceTest class methodsFor: 'interface specs'!

windowSpec
       "UIPainter new openOnClass: self andSelector: #windowSpec"

       <resource: #canvas>
       ^#(#FullSpec
              #window:
              #(#WindowSpec
                     #label: 'Unlabeled Canvas'
                     #bounds: #(#Rectangle 7 245 546 448 ) )
              #component:
              #(#SpecCollection
                     #collection: #(
                            #(#RegressFceSpec
                                   #layout: #(#Rectangle 240 13 517 189 )
                                   #model: #pointArray
                                   #pointDiameter: 4
                                   #type: #exponential
                                   #displayPointCoordinate: true )
                            #(#LabelSpec
                                   #layout: #(#Point 21 87 )
                                   #label: 'Array Index:' )
                            #(#InputFieldSpec
                                   #layout: #(#Rectangle 111 87 211 105 )
                                   #colors:
                                   #(#LookPreferences
                                          #setBackgroundColor: #(#ColorValue #white ) )
                                   #model: #arrayIndex
                                   #type: #number )
                            #(#InputFieldSpec
                                   #layout: #(#Rectangle 111 13 211 31 )
                                   #colors:
                                   #(#LookPreferences
                                          #setBackgroundColor: #(#ColorValue #white ) )
                                   #model: #x
                                   #type: #number )
                            #(#InputFieldSpec
                                   #layout: #(#Rectangle 111 50 211 68 )
                                   #colors:
                                   #(#LookPreferences
                                          #setBackgroundColor: #(#ColorValue #white ) )
                                   #model: #y
                                   #type: #number )
                            #(#ActionButtonSpec
                                   #layout: #(#Rectangle 111 129 211 174 )
                                   #model: #accept
                                   #label: 'Accept'
                                   #defaultable: true )
                            #(#LabelSpec
                                   #layout: #(#Point 79 13 )
                                   #label: 'x:' )
                            #(#LabelSpec
                                   #layout: #(#Point 79 50 )
                                   #label: 'y:' ) ) ) )! !

WidgetSpec subclass: #RegressFceSpec
       instanceVariableNames: 'pointDiameter type displayPointCoordinate '
       classVariableNames: ''
       poolDictionaries: ''
       category: 'UI-RegressFce'!


!RegressFceSpec methodsFor: 'private'!

defaultFlags

       ^8!

defaultModel
       ^ValueHolder with: (OrderedCollection
                     with: 1 @ 5
```

```
                        with: 3 @ 6.5
                        with: 6 @ 8
                        with: 6.2 @ 10)!

dispatchTo: policy with: builder
        ^policy regressFce: self into: builder! !

!RegressFceSpec methodsFor: 'initialize-release'!

initialize
        super initialize.
        pointDiameter := 4.
        type := #linear.
        displayPointCoordinate := false! !

!RegressFceSpec methodsFor: 'accessing'!

displayPointCoordinate
        ^displayPointCoordinate!

displayPointCoordinate: aBoolean
        displayPointCoordinate := aBoolean!

pointDiameter
        ^pointDiameter!

pointDiameter: aVal
        aVal > 0 ifTrue: [pointDiameter := aVal]!

type
        ^type!

type: aVal
        type := aVal! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

RegressFceSpec class
        instanceVariableNames: ''!


!RegressFceSpec class methodsFor: 'class initialization'!

initialize
        "Put me into UIPalette's list of Specs"

        (UIPalette activeSpecsList includes: #RegressFceSpec)
                ifFalse: [UIPalette activeSpecsList add: #RegressFceSpec]! !

!RegressFceSpec class methodsFor: 'private-interface building'!

addBindingsTo: env for: inst channel: aChannel
        super
                addBindingsTo: env
                for: inst
                channel: aChannel.
        env at: #pointDiameter put: (self
                        adapt: inst
                        forAspect: #pointDiameter
                        channel: aChannel).
        env at: #type put: (self
                        adapt: inst
                        forAspect: #type
                        channel: aChannel).
        env at: #displayPointCoordinate put: (self
                        adapt: inst
                        forAspect: #displayPointCoordinate
                        channel: aChannel)!

componentName
        ^'Regression'!

placementExtentBlock

        ^[:bldr | 180@110]!

slices
        ^#(     (Basics basicsEditSpec)
                (Color propSpec ColorToolModel)
                (Position propSpec PositionToolModel) )!

specGenerationBlock
```

```
	^[:ctrlr :point | RegressFceSpec new layout: ((ctrlr gridPoint: point)
			extent: (ctrlr currentMode value class
					placementExtentFor:RegressFceSpec
					inBuilder: ctrlr builder))]! !

!RegressFceSpec class methodsFor: 'interface specs'!

basicsEditSpec
	"UIPainter new openOnClass: self andSelector: #basicsEditSpec"

	<resource: #canvas>
	^#(#FullSpec
		#window:
		#(#WindowSpec
			#label: 'Unlabeled Canvas'
			#bounds: #(#Rectangle 18 290 369 521 ) )
		#component:
		#(#SpecCollection
			#collection: #(
				#(#LabelSpec
					#layout: #(#Point 13 23 )
					#label: 'Aspect:' )
				#(#InputFieldSpec
					#layout: #(#Rectangle 75 23 323 41 )
					#colors:
					#(#LookPreferences
						#setBackgroundColor: #(#ColorValue #white ) )
					#model: #model
					#menu: #fieldMenu )
				#(#LabelSpec
					#layout: #(#Point 39 69 )
					#label: 'ID:' )
				#(#InputFieldSpec
					#layout: #(#Rectangle 75 69 323 87 )
					#colors:
					#(#LookPreferences
						#setBackgroundColor: #(#ColorValue #white ) )
					#model: #name
					#menu: #fieldMenu )
				#(#LabelSpec
					#layout: #(#Point 12 115 )
					#label: 'Point Diameter:' )
				#(#InputFieldSpec
					#layout: #(#Rectangle 108 114 323 132 )
					#colors:
					#(#LookPreferences
						#setBackgroundColor: #(#ColorValue #white ) )
					#model: #pointDiameter
					#type: #number )
				#(#GroupBoxSpec
					#layout: #(#Rectangle 214 141 323 214 )
					#label: 'Type' )
				#(#RadioButtonSpec
					#layout: #(#Point 227 161 )
					#model: #type
					#label: 'Linear'
					#select: #linear )
				#(#RadioButtonSpec
					#layout: #(#Point 227 185 )
					#model: #type
					#label: 'Exponential'
					#select: #exponential )
				#(#CheckBoxSpec
					#layout: #(#Point 12 161 )
					#model: #hasBorder
					#label: 'Bordered' )
				#(#CheckBoxSpec
					#layout: #(#Point 12 185 )
					#model: #displayPointCoordinate
					#label: 'Display Point Coordinate' ) ) ) )! !

!RegressFceSpec class methodsFor: 'resources'!

paletteIcon
	"UIMaskEditor new openOnClass: self andSelector: #paletteIcon"

	<resource: #image>
	^CachedImage on: (Image extent: 26@26 depth: 3 bitsPerPixel: 4 palette:
(MappedPalette withColors: ((Array new: 6) at: 1 put: ColorValue black; at: 2 put:
(ColorValue scaledRed: 6553 scaledGreen: 6553 scaledBlue: 6553); at: 3 put: (ColorValue
scaledRed: 4112 scaledGreen: 4112 scaledBlue: 4112); at: 4 put: ColorValue lightGray; at: 5
```

```
put: ColorValue blue; at: 6 put: ColorValue red; yourself)) usingBits: (ByteArray
fromPackedString: '@@@@@@@@@@@@@@@@@@@@@@@@DQDQDQDQDQDQDQD2@@@@@ADQDQDQDQDQDQDQL
@@@@@QDQDQDQDQDQDQDSH@@@@@DQ@QDQDQDQDQDQD2@@@@@ADPDQDQDQDQDQDQL
@@@@@QDADQDQDQDQDQDSH@@@@@DQ@QDQDQPQDQDQD2@@@@@ADPDQDQDQDTDQDQL
@@@@@QDADQDQDQDUTQDSH@@@@@DQ@QDQDQDUTQDQD2@@@@@ADPDQPQDUTQDQDQL
@@@@@QDADQDQTQDTDQDSH@@@@@DQ@QDQUQDQDQDQD2@@@@@ADPDQUQEADQDQDQL
@@@@@QDAEQDQDQDQDQDSH@@@@@DQ@QDQDQEADQDQD2@@@@@ADPDQDQDQDQDQDQL
@@@@@QDADQDQDQDQDQDSH@@@@@DQ@QDQDQDQDQDQD2@@@@@ADP@@@@@@@@@@DQL
@@@@@QDQDQDQDQDQDQDSH@@@@@DQDQDQDQDQDQDQD2@@@@@CL3L3L3L3L3L3L3L
@@@@@"H"H"H"H"H"H"H"H@@@@@@@@@@@@@@@@@@@@@@@@@@@@b')!

paletteMonoIcon
	"UIMaskEditor new openOnClass: self andSelector: #paletteMonoIcon"

	<resource: #image>
	^CachedImage on: (Image extent: 26@26 depth: 1 bitsPerPixel: 1 palette:
MonoMappedPalette blackWhite usingBits: #[0 0 0 0 127 255 255 128 127 255 255 128 127 255 255
128 119 255 255 128 119 255 255 128 119 255 255 128 119 255 255 128 119 255 223 128 119 251
255 128 119 255 231 128 119 255 159 128 119 190 127 128 119 249 223 128 119 247 255 128 119
203 255 128 119 63 255 128 118 254 255 128 119 255 255 128 119 255 255 128 119 255 255 128
112 0 3 128 127 255 255 128 127 255 255 128 127 255 255 128 0 0 0 0])! !

SimpleView subclass: #RegressFceView
	instanceVariableNames: 'pointDiameter type displayPointCoordinate x1 y1 x2 y2 mX mY
dX dY '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'UI-RegressFce'!


!RegressFceView methodsFor: 'initialize-release'!

initialize
	super initialize.
	pointDiameter := 4.
	type := #linear.
	displayPointCoordinate := false! !

!RegressFceView methodsFor: 'private'!

initialModel
	^ValueHolder with: (Array
			with: 1 @ 5
			with: 3 @ 6.5
			with: 6 @ 8
			with: 6.2 @ 10)! !

!RegressFceView methodsFor: 'coord-transformation'!

backTransformX: aX
	^aX - x2 / mX + x1!

computeCoefficient: aGC
	| mVal |
	mVal := self model value asSortedCollection: [:a :b | a x < b x].
	x1 := mVal first x.
	mX := mVal last x - x1.
	x1 := x1 - (mX * 0.1).
	mX := 1.2 * mX.
	dX := mX.
	mVal := mVal asSortedCollection: [:a :b | a y < b y].
	y1 := mVal first y.
	(aGC widthOfString: y1 printString)
		< (aGC widthOfString: mVal last y printString)
		ifTrue: [x2 := 10 + (aGC widthOfString: mVal last y printString)]
		ifFalse: [x2 := 10 + (aGC widthOfString: y1 printString)].
	mY := mVal last y - y1.
	y1 := y1 - (mY * 0.1).
	mY := 1.2 * mY.
	dY := mY.
	y2 := self bounds extent y - 20.
	self displayPointCoordinate
		ifTrue: [mX := self bounds extent x - x2 - 32/ mX.
			mY := 34 - self bounds extent y / mY]
		ifFalse:
			[mX := self bounds extent x - x2 - 8 / mX.
			mY := 28 - self bounds extent y / mY]!

transform: aPoint
	^(mX * (aPoint x - x1) + x2) rounded @ (mY * (aPoint y - y1) + y2) rounded!
```

```
transformX: aX
	^(mX * (aX - x1) + x2) rounded! !

!RegressFceView methodsFor: 'displaying'!

displayAxesOn: aGC
	| base i numbering k |
	self foregroundColor isNil
		ifTrue: [aGC paint: ColorValue black]
		ifFalse: [aGC paint: self foregroundColor].
	aGC displayLineFrom: (self transform: x1 @ y1)
		to: (self transform: x1 + dX @ y1).
	aGC displayLineFrom: (self transform: x1 @ y1)
		to: (self transform: x1 @ (y1 + dY)).
	base := 10 ** (dX log - 0.32) floor.
	i := (x1 / base) ceiling * base.
	numbering := (dX / base / 4) floor.
	numbering == 0 ifTrue: [numbering := 1].
	k := 0.
	[i < (x1 + dX)]
		whileTrue:
			[| pom |
			pom := self transform: i @ y1.
			aGC displayLineFrom: pom + (0 @ 2) to: pom - (0 @ 2).
			k \\ numbering == 0 ifTrue: [aGC displayString: i printString at: pom -
(((aGC widthOfString: i printString)
							/ 2) rounded @ -16)].
			k := k + 1.
			i := i + base].
	base := 10 ** (dY log - 0.32) floor.
	i := (y1 / base) ceiling * base.
	numbering := (dY / base / 3) floor.
	numbering == 0 ifTrue: [numbering := 1].
	k := 0.
	[i < (y1 + dY)]
		whileTrue:
			[| pom |
			pom := self transform: x1 @ i.
			aGC displayLineFrom: pom - (2 @ 0) to: pom + (2 @ 0).
			k \\ numbering == 0 ifTrue: [aGC displayString: i printString at: pom -
((aGC widthOfString: i printString)
							+ 5 @ -6)].
			k := k + 1.
			i := i + base]!

displayBackgroundOn: aGC
	self backgroundColor isNil
		ifFalse:
			[aGC paint: self backgroundColor.
			(Rectangle origin: 0 @ 0 extent: self bounds extent - (1 @ 1))
				displayFilledOn: aGC]!

displayOn: aGC
	| mVal |
	mVal := self model value asOrderedCollection.
	self computeCoefficient: aGC.
	self displayBackgroundOn: aGC.
	self displayPoints: mVal on: aGC.
	self displayRegression: mVal on: aGC.
	self displayAxesOn: aGC!

displayPoints: aMVal on: aGC
	aGC paint: self selForegroundColor.
	aMVal
		do:
			[:p |
			| trP |
			trP := self transform: p.
			aGC displayDotOfDiameter: pointDiameter at: trP.
			self displayPointCoordinate
				ifTrue:
					[aGC paint: self foregroundColor.
					aGC displayString: '[' , p x printString , ', ' , p y
printString , ']' at: trP- (10@5).
					aGC paint: self selForegroundColor]]!

displayRegression: aMVal on: aGC
	| sumX sumY sumSquareX sumXY b0 b1 pomArray |
	type == #exponential
		ifTrue:
			[pomArray := OrderedCollection new: aMVal size.
```

```
                        aMVal do: [:a | a y > 0 ifTrue: [pomArray add: a x @ a y ln]]]
                ifFalse: [pomArray := aMVal].
        sumX := 0.
        sumY := 0.
        sumSquareX := 0.
        sumXY := 0.
        pomArray
                do:
                        [:a |
                        sumX := sumX + a x.
                        sumY := sumY + a y.
                        sumSquareX := sumSquareX + (a x ** 2).
                        sumXY := sumXY + (a x * a y)].
        b0 := sumY * sumSquareX - (sumXY * sumX) / (aMVal size * sumSquareX - (sumX ** 2)).
        b1 := aMVal size * sumXY - (sumX * sumY) / (aMVal size * sumSquareX - (sumX ** 2)).
        aGC paint: self selBackgroundColor.
        type == #exponential
                ifTrue:
                        [| lines |
                        lines := OrderedCollection new: (self transformX: dX)
                                                        + 2.
                        (self transformX: x1)
                                to: (self transformX: x1 + dX)
                                do:
                                        [:trX |
                                        | x |
                                        x := self backTransformX: trX.
                                        lines add: (self transform: x @ (b1 * x + b0) exp)].
                        aGC displayPolyline: lines]
                ifFalse: [aGC displayLineFrom: (self transform: x1 @ (b1 * x1 + b0))
                                to: (self transform: x1 + dX @ (b1 * (x1 + dX) + b0))]! !

!RegressFceView methodsFor: 'accessing'!

displayPointCoordinate
        ^displayPointCoordinate!

displayPointCoordinate: aBoolean
        displayPointCoordinate := aBoolean!

model
        super model isNil
                ifTrue: [^self initialModel]
                ifFalse: [^super model]!

pointDiameter
        ^pointDiameter!

pointDiameter: aVal
        aVal > 0 ifTrue: [pointDiameter := aVal]!

type
        ^type!

type: aVal
        aVal == #exponential
                ifTrue: [type := aVal]
                ifFalse: [type := #linear]! !

!RegressFceView methodsFor: 'visual properties'!

selBackgroundColor
        | pom |
        pom := self widgetState colors.
        (pom notNil and: [pom selectionBackgroundColor notNil])
                ifTrue: [^pom selectionBackgroundColor]
                ifFalse: [^ColorValue red]!

selForegroundColor
        | pom |
        pom := self widgetState colors.
        (pom notNil and: [pom selectionForegroundColor notNil])
                ifTrue: [^pom selectionForegroundColor]
                ifFalse: [^ColorValue blue]! !
RegressFceSpec initialize!


!UILookPolicy methodsFor: 'building'!

regressFce: regressFceSpec into: builder
        | component mdl |
```

```
mdl := regressFceSpec modelInBuilder: builder.
component := RegressFceView new model: mdl.
component pointDiameter: regressFceSpec pointDiameter.
component type: regressFceSpec type.
component displayPointCoordinate: regressFceSpec displayPointCoordinate.
builder component: component.
self
       setDispatcherOf: component
       fromSpec: regressFceSpec
       builder: builder.
regressFceSpec hasBorder
       ifTrue: [builder wrapWith: (self borderedWrapperFor: regressFceSpec)]
       ifFalse: [builder wrapWith: (self simpleWrapperFor: regressFceSpec)].
builder applyLayout: regressFceSpec layout.
builder wrapWith: (self simpleWidgetWrapperOn: builder spec: regressFceSpec)! !
```