PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Implementace grafického editoru s prvky vizuálního programování

2012 Josef Bláha

Anotace

Práce popisuje návrh a implementaci experimentálního grafického editoru, který reprezentuje kresbu pomocí prototypového objektového jazyka. Logické vazby mezi objekty lze vyjádřit pomocí výrazů a seskupování do kontejnerů.

Děkuji svému vedoucímu práce Mgr. Martinu Dostálovi, Ph.D. za jeho čas a povzbuzování při vzniku této práce.

Obsah

Seznam obrázků

1. Úvod

Cílem mé práce je navrhnout a implementovat experimentální grafický editor s podporou vizuálního programování – mělo by jít o software, který stojí na pomezí mezi grafickými editory a prostředími pro programování. Hlavním účelem bude vytváření grafických obrazců, diagramů apod., jaké lze nakreslit v tradičních vektorových editorech, navíc ale s možností zavést do kresby nějakou vnitřní logiku.

Editor, který s tímto zadáním vznikl, se jmenuje Pikta. Stejně tak budeme nazývat i jazyk, který slouží pro reprezentaci kresby.

1.1. Modelový problém

Abychom lépe pochopili zaměření editoru Pikta a mohli ho porovnat se stávajícími nástroji podobného zaměření, uveďme si příklad modelového problému, který budeme řešit.

Chceme vykreslit sadu klasických hracích karet francouzského typu. Každá z 52 karet je sice jiná, mají ale řadu společných rysů (jeden ze čtyř symbolů barev, stejné rozmístění obsahu pro kartu jedné hodnoty ve čtyřech barvách atd.). Pomocí Pikty by mělo být možné tyto společné rysy nějakým způsobem postihnout, aby se daly změnit stejně jednoduše jako jedna vlastnost jediné karty.

1.2. Stávající řešení

Nyní se podíváme na to, jak bychom modelový problém řešili pomocí dostupného softwaru, který má blízko k editoru Pikta. U každého nás budou zajímat jeho grafické možnosti, a také možnosti automatizace.

1.2.1. Vektorové editory

Jako první by nás nejspíš napadlo použít některý vektorový grafický editor jako například *Adobe Illustrator*, *CorelDRAW* nebo *Inkscape*. Tyto editory obsahují rozsáhlý arzenál nástrojů včetně např. seskupování a klonování objektů, které bychom v našem případě využili. Dokonce v nich lze pomocí skriptování dosáhnout jisté míry automatizace při vytváření kresby.

Zmíněné skriptování ale znamená přejít od vizuálních úprav do textového programovacího jazyka. Pokud skriptovací jazyk neovládáme, v našem případě bychom pravděpodobně na automatizaci rezignovali a vyřešili problém „ručně“.

Charakteristickým rysem vektorových editorů je principiálně jednoduchý model kresby s ohromným množstvím vlastností a nastavení, díky nimž lze maximálně ovlivňovat každý detail výsledné kresby. To s sebou nese také velkou obtížnost úprav v hotové kresbě.

1.2.2. MetaPost

Dalším kandidátem může být programovací jazyk zaměřený na tvorbu grafiky, jako je například *MetaPost*. Jedná se o jazyk pro popis vektorových obrázků (zejména grafů a diagramů) v typografickém systému LaTeX. Grafika se programuje pomocí relativně nízkoúrovňových příkazů kreslení čar, křivek a bodů a aplikace transformací. Můžeme také definovat funkce, které umožní snadné kreslení složitějších tvarů.

Naši karetní úlohu by nemuselo být obtížné v *MetaPostu* vytvořit. Je to ale opět (stejně jako u skriptování) podmíněné nutností naučit se textový programovací jazyk. Navíc se nejedná o WYSIWYG řešení – po každé změně programu je nutné ho znovu vykreslit.

1.2.3. Programovací jazyky pro UI

Můžeme také uvažovat nad použitím nástrojů, které se používají při tvorbě uživatelských rozhraní softwaru. Ty se totiž podobně jako Pikta pohybují mezi dvěma protichůdnými směry – vizuálními metodami pro kreslení grafiky a klasickým programováním – i když spíše tíhnou k tomu druhému. Jejich grafické možnosti obvykle nejsou nijak převratné, doprovodný program ve vyšším programovacím jazyce však nabízí úplnou automatizaci.

SK8 [1] bylo prostředí pro interaktivní vývoj multimediálních aplikací vyvíjené v minulém století společností Apple. Produkt již není dále vyvíjen a ani nikdy nebyl veřejně dostupný. Přesto bychom ho měli zahrnout do našeho srovnání, protože má několik zajímavých rysů.

Zásadní odlišností od jiných systémů pro tvorbu UI je, že v jeho jádru stál prototypový objektový systém. *SK8* je založen na metafoře jeviště a herců. Plocha na obrazovce počítače představuje jeviště, na které lze umisťovat herce – jak geometrické objekty, tak i interaktivní ovládací prvky UI, obrázky, video apod.

V reakci na události lze spouštět programy v předchůdci jazyka *AppleScript* [2]. Kromě handlerů událostí a změn vlastností objektů pomocí kódu obsahuje systém tzv. porty, které umožňují propojit několik vlastností různých objektů, které si pak vzájemně oznamují změny hodnot.

WPF [3] je knihovna pro tvorbu uživatelských rozhraní společnosti Microsoft dostupná jako součást *.NET Frameworku*. Kromě vizuálního vytváření a psaní kódu lze UI definovat pomocí značkovacího jazyka XAML.

[1] Čteme „skejt“.

[2] Není bez zajímavosti, že *AppleScript* lze použít při skriptování *Adobe Illustratoru*.

[3] *Windows Presentation Foundation*.

Ze zajímavých rysů *WPF* kromě obstojných grafických možností jmenujme kontejnery a binding. Kontejnery umožňují různé způsoby pozicování ovládacích prvků. Mimo klasické absolutní pozicování reprezentované prvkem *Canvas* lze použít i dvě varianty mřížky (*Grid* a *UniformGrid*) a panel, který obsah řadí za sebe – *StackPanel*. Tyto kontejnery představují elegantní způsob řešení často používaných rozmístění prvku; není nutné počítat správné absolutní souřadnice.

Jako binding se označuje technika pro svázání vlastností objektů, při kterém probíhá automatická vzájemná aktualizace při změně.

Obě dvě popsané technologie tvorby UI se staly hlavními zdroji inspirace pro editor Pikta. Rysy, které si z nich Pikta odnáší, jsou prototypy, kontejnery a vazby vlastností.

1.3. Prostor pro Piktu

V poli mezi vektorovými editory a zbytkem popsaných řešení, ve kterém je kladen důraz spíše na programování, zůstává prostor, který není pokryt. Sahá od místa, kde se začínají ztrácet grafické možnosti vektorových editorů, až do bodu, kde schopnosti systémů UI začínají být zastiňovány složitostí použití jejich programovacích jazyků. Tento prostor je přesně místo, kde může Pikta ukázat svou přednost – aplikaci jednoduchého programování.

Postup řešení modelového problému pomocí Pikty je popsán v části 4.2.

2. Jazyk systému Pikta

Pro reprezentaci grafických objektů a vztahů mezi nimi slouží v systému Pikta prototypový objektový jazyk. Množina objektů tohoto jazyka je grafickým editorem interpretována jako vykreslovaná scéna.

V této kapitole si podrobně popíšeme jazyk i jeho vizuální interpretaci.

2.1. Prototypové jazyky

Jako objekt se v objektově orientovaných programovacích jazycích označuje entita, která v sobě slučuje data a chování, a odpovídá určitému pojmu z modelovaného systému. Potřeba kategorizovat pojmy nás vede k tomu, že se snažíme v objektech nalézt hierarchii a jejich společné vlastnosti. Máme-li například červený obdélník a zelený čtverec, po analýze jejich vlastností dojdeme k tomu, že oba přes svou odlišnost odpovídají určitému nadřazenému obecnějšímu pojmu – obdélníku o neuvedené velikosti a barvě.

Způsob, jakým se reprezentuje tato nadřazená kategorie, rozděluje objektové jazyky na jazyky založené na třídách a prototypové jazyky. Jazyky založené na třídách zavádějí hierarchii na popisech objektů (třídách), prototypové jazyk přímo na objektech samotných.

Mezi výhody protypových jazyků patří jednoduchost definice nových pojmů a větší volnost při předefinování struktury nebo chování objektu z určitého konceptu ([1]). Reprezentace pojmů i konkrétních instancí jedním způsobem (objektem) naopak může způsobit, že nemusí být jisté, do které této skupiny objekt patří.

Podrobnější rozbor vlastností prototypových jazyků vzhledem k jazykům založených na třídách naleznete v [1].

2.2. Návrh jazyka

Umělé jazyky, mezi které patří i ty programovací, vznikají vždy za konkrétním účelem. Proto nejdříve popíšu, jaké požadavky stály na začátku vývoje jazyka Pikta.

1. *Objektová orientace* je zřejmým požadavkem vzhledem k tomu, že chceme popisovat kresbu složenou z grafických objektů. Jeden grafický objekt může přímo odpovídat jednomu objektu jazyka.

2. *Jednoduchost.* Protože se dá očekávat, že cílovou skupinou uživatelů systému Pikta nebudou programátoři, ale spíše běžní uživatelé, provázela návrh jazyka snaha o maximální jednoduchost. Ta je hlavním důvodem použití paradigmatu založeného na prototypech. Usnadní nám to použití dědičnosti, neboť nemusíme zavádět pojem třída, který by byl reprezentován odlišným způsobem než samotné objekty.

Dalším důsledkem tohoto požadavku je absence metod a zasílání zpráv (nebo obecněji řečeno absence vedlejšího efektu), jak je obvyklé v programovacích jazycích. Jejich funkci částečně suplují výrazy ve vlastnostech objektů.

Interpretace jazyka je čistě vizuální. Jediná definovaná syntaxe, se kterou přijde uživatel do styku, je pro zápisování výrazů. Tu si podrobněji popíšeme v následující části textu. Další textový zápis se objeví až při ukládání scény v editoru – scéna se uloží jako strukturovaná data ve formátu XML. Uživatel samotný však s tímto jazykem nepracuje.

2.3. Hodnoty a výrazy

Jazyk Pikta rozeznává tři základní datové typy *hodnot*: číslo, text a element. Ty jsou charakterizovány následovně:

Číslo Číselné hodnoty tvoří podmnožinu reálných čísel. Jsou implementovány jako čísla s plovoucí desetinnou čárkou ve dvojité přesnosti podle standardu IEEE 754.

Text Texty jsou libovolně dlouhé řetězce znaků. Jsou implementovány jako řetězce v kódování Unicode.

Element Posledním typem hodnoty je libovolný element. Elementy si podrobně popíšeme v následující části. Prozatím stačí vědět, že reprezentují netriviální objekty – grafické tvary, kontejnery, barvy atd.

Ačkoliv má každá hodnota vlastní datový typ, jeho dodržování není zcela striktní. Mezi čísly a texty existují implicitní [4] konverze:

- Každé číslo lze převést na text.
- Texty, které reprezentují čísla, lze převést na číslo. Např. hodnoty `"28"` a `"3,14"` lze implicitně převést na čísla.

Hodnota vznikne vyhodnocením *výrazu*. Nejednoduššími výrazy jsou konstanty:

- Číselné jako například `1024`, `-5`, `28,7`[5].
- Textové např. `"vyzáblé prasátko"`. Texty se zapisují způsobem vlastním pro programovací jazyky – ve dvojitých uvozovkách.

[4] Implicitní se nazývá konverze taková, která je provedena při výpočtu automaticky, kdykoliv je to nutné. Naproti tomu explicitní konverzi je třeba výslovně vyžádat. Jazyk Pikta explicitní konverze nepoužívá.

[5] Desetinná čísla se v editoru zapisují s desetinnou čárkou nebo tečkou podle místního nastavení operačního systému. Více o jazykových a kulturních aspektech naleznete v části 5.4.

Dále můžeme ve výrazech intuitivně používat základní aritmetické operátory `+`, `-`, `*`, `/` a `%` (modulo, zbytek po celočíselném dělení). Priorita těchto operátorů je taktéž intuitivní – násobení, dělení a zbytek mají přednost před sčítáním a odčítáním. V případě potřeby je možné využít závorky. Uveďme si několik příkladů:

```
20 * 13 + 5
20 * (13 + 5)
(15 + 3,14 * (2 + -50)) % 10
```

Editor Pikta definuje i několik základních funkcí, které můžeme ve výrazu volat. Volání funkce začíná jejím názvem, v závorce pak následují jednotlivé parametry oddělené středníkem[6].

`TentoElement()` vrací element, v jehož kontextu je výraz vyhodnocován.

`Pi()` vrací hodnotu π.

`Abs(číslo)` vrací absolutní hodnotu čísla.

`Sin(úhel)` vrací sinus zadaného úhlu v radiánech.

`Cos(úhel)` vrací kosinus zadaného úhlu v radiánech.

`Celé(číslo)` vrací celou část zadaného čísla.

`Pokud(podmínka; ano; ne)` Pokud má výraz v podmínce nenulovou hodnotu, vrací druhý parametr, má-li nulovou hodnotu, vrací třetí parametr.

`Hodina()` vrací aktuální hodinu (v rozsahu 0-24).

`Minuta()` vrací aktuální minutu (v rozsahu 0-59).

`Sekunda()` vrací aktuální sekundu (v rozsahu 0-59).

`Kontejner(element)` vrací kontejner zadaného elementu.

`SoučástNaPozici(kontejner; index)` vrací element, který je v kontejneru na zadaném indexu (počítáno od nuly).

`ŠířkaTextu(text; velikost)` vrací šířku textu při vykreslení se zadanou velikostí.

`VýškaText(text; velikost)` vrací výšku textu při vykreslení se zadanou velikostí.

[6]Běžně užívanou čárku nebylo možné použít z důvodu, že umožňujeme zadávat čísla s desetinnou čárkou. Vždy musíme být schopní rozlišit, jestli např. `Funkce(1,2)` volá variantu funkce s jedním nebo dvěma parametry

V neposlední řadě se můžeme ve výrazu odkázat na element či jeho vlastnost. Nejlépe si to ukážeme na příkladech:

`MůjElement` se vyhodnotí na element s uvedeným názvem.

`.nějakáVlastnost` představuje hodnotu vlastnosti `nějakáVlastnost` elementu, v jehož kontextu je výraz vyhodnocován.

`MůjElement.kontejner.šířka` kombinuje obě možnosti. Výraz se vyhodnotí na hodnotu vlastnosti `šířka` elementu, na který se vyhodnotí vlastnost `kontejner` elementu s názvem `MůjElement`.

Vyhodnocení výrazu vždy skončí, a to buď úspěchem – potom je výsledkem hodnota, anebo chybou. V případě chyby je součástí výsledku i zpráva popisující chybu.

2.4. Elementy a vlastnosti

Element je objekt složený z odkazu na *prototyp* a kolekce vlastností. *Vlastnost* je definována svým názvem a výrazem, který je v ní uložený.

Element může mít definován unikátní název, kterým se na něj můžeme odkázat. Není to však povinné.

Prototypem elementu může být jakýkoliv jiný element. Každý element musí mít definovaný prototyp. Výjimku tvoří speciální element pojmenovaný `Báze`, který prototyp nemá a je předkem všech elementů.

V souladu se zažitou terminologií objektově orientovaných paradigmat budeme prototyp elementu, protyp prototypu atd. nazývat *předky* daného elementu. Elementy, které jsou od určitého protypu odvozeny přímo i nepřímo budeme označovat jako jeho *potomky*.

Nový element může vzniknout pouze klonováním existujícího elementu. Elementy se obvykle klonují mělce[7]. Jsou ale případy, kdy mělké klonování jde proti intuici. V Piktě se to týká speciálních elementů, které se nazývají kontejnery. Pro klonování kontejnerů platí zvláštní pravidla popsaná v části 2.6. Po klonování už nelze prototyp nového elementu vyměnit za jiný.

2.4.1. Prototypy a delegování vlastností

Nyní si popíšeme jeden z principů protypové dědičnost nazývaný delegování.

Vlastnosti, které daný element přímo definuje, nazýváme *lokální*. Mimo lokálních vlastností každý element automaticky přejímá všechny vlastnosti svého prototypu. Má-li element definovanou lokální vlastnost a zároveň přejímá vlastnost se stejným názvem z prototypu, má přednost lokální definice vlastnosti.

[7] Mělké klonování znamená, že se klonuje pouze vlastní element, ale jeho součásti ne.

Příklad vidíme na obrázku 1. Prototypem elementu `Obdélník` je `Tvar`. Element `Obdélník` definuje vlastnosti `okraj`, `tloušťka`, `šířka`, `výška`, `výplň` a `zakulaceníRohů`. Vlastnosti `šířka`, `výplň` a `zakulacenírohů` definuje lokálně, ostatní přejímá ze svého prototypu.

`Obdélník`	Prototyp = `Tvar`
`výplň`	`Zelená`
`šířka`	`2 * .výška`
`zakulacenírohů`	`10`

`Tvar`	Prototyp = `Báze`
`výplň`	`Červená`
`okraj`	`Černá`
`tloušťka`	`.šířka / 10`
`šířka`	`200`
`výška`	`150`

Obrázek 1. Element a jeho prototyp

Podívejme se na hodnoty jednotlivých vlastností:

- `Obdélník.výplň` se vyhodnotí na barvu, kterou reprezentuje element `Zelená`.
- `Obdélník.šířka` se vyhodnotí na číslo `300`.
- `Obdélník.tloušťka` se vyhodnotí na `30`.
- `Tvar.šířka` se vyhodnotí na číslo `200`.
- `Tvar.tloušťka` se vyhodnotí na číslo `20`.
- `Tvar.zakulacenírohů` se vyhodnotí na chybu, protože element `Tvar` vlastnost `zakulacenírohů` nedefinuje.

Všimněte si, že vlastnost se vždy vyhodnocuje v kontextu určitého elementu. Vlastnost `tloušťka` je sice lokální v elementu `Tvar`, ale když se vyhodnocuje v elementu `Obdélník`, použije se jeho `šířka`.

2.4.2. Názvy elementů a vlastností

Názvy elementů a vlastností jsou tvořeny řetězci skládajících se výhradně z písmen, číslic a znaku podrtžítka. Přičemž platí, že název musí vždy začínat písmenem.

V názvech se nerozlišuje velikost písmen. Názvy `MůjNázev` a `můjNÁZEV` jsou ekvivalentní. Doporučená konvence pro názvy definuje tato pravidla:

- Je-li název složen z více slov, používá se tzv. velbloudí notace, kdy je první písmeno slova velké a ostatní písmena malá – `PříkladVelbloudíNotace`.
- Názvy elementů začínají velkým písmenem.
- Názvy vlastností začínají malým písmenem.

2.4.3. Výčtové typy

Někdy je nutné, aby obor hodnot vlastnosti byl nevelkou diskrétní množinou, kterou dost dobře nelze reprezentovat čísly. V programovacích jazycích se obyčejně nazývá *výčtový typ*. V jazyce Pikta se tento problém řeší tak, že pro jednotlivé hodnoty výčtu definujeme speciální pojmenované elementy a ty pak do vlastnosti přiřazujeme.

Příkladem může být vlastnost `Viditelný.horizontálníZarovnání`, do které lze vložit hodnoty (elementy) `Vlevo`, `NaStřed` a `Vpravo`. Jazyk nám se svými volnými pravidly nezabrání přiřadit do vlastnosti úplně jinou hodnotu, například číslo. Je na editoru, aby se s touto chybou vyrovnal. Učiní to způsobem popsaným v části 3.6.

2.5. Základní elementy a tvary

V této části si popíšeme základní elementy, které definuje editor Pikta. Nejdříve si zodpovíme otázku, jak v Piktě definujeme, že určitý element se má např. vykreslit jako elipsa.

Grafická interpretace elementu se vybírá podle jeho předka. Je definováno několik základních „praprototypů“, které představují jednotlivé kategorie elementů. Element reprezentuje barvu, je-li potomkem elementu `Barva`; element se vykreslí jako elipsa, je-li potomkem elementu `Elipsa` apod.

2.5.1. Barva

Element popisuje barvu v barevném schématu RGB s alfa kanálem určujícím průhlednost (0 průhledný, 255 neprůhledný). Je prototypem elementů `Průhledná`, `Bílá`, `Černá`, `Červená`, `Zelená`, `Modrá` a `Oranžová`.

`Barva`	Prototyp = `Báze`
`a`	255
`r`	200
`g`	200
`b`	200

2.5.2. Viditelný

Element definující základní vlastnosti vykreslitelných elementů. Vlastnosti `horizontálníPoloha` a `vertikálníPoloha` se týkají zarovnání elementu v buňce mřížky – viz 2.6.3.

Viditelný	Prototyp = Báze
horizontálníPoloha	Vlevo
vertikálníPoloha	Nahoru
kontejner	Kontejner(TentoElement())

2.5.3. Tvar

Společný prototyp základních tvarů.

Tvar	Prototyp = Viditelný
výplň	Průhledná
okraj	Černá
tloušťka	1
šířka	10
výška	10

2.5.4. Obdélník

Obdélník	Prototyp = Tvar
šířka	200
výška	100

2.5.5. Elipsa

Elipsa	Prototyp = Tvar

2.5.6. Text

Element reprezentuje textovou popisku. `velikostTextu` je hodnota v bodech. `výplň` určuje barvu textu. Velikost elementu se automaticky mění podle velikosti obsaženého textu.

Text	Prototyp = Tvar
okraj	Průhledná
text	"123"
velikostTextu	10
výplň	Černá
šířka	ŠířkaTextu(.text; .velikostTextu)
výška	VýškaTextu(.text; .velikostTextu)

2.5.7. Bitmapa

Element umožňuje vložit do scény bitmapu z externího souboru. Velikost elementu se nastavuje automaticky podle velikosti zdrojového obrázku. Pokud nastavíme jinou velikost, obrázek se roztáhne do celé velikosti elementu (poměr stran se nezachovává).

Do vlastnosti `zdroj` zadáváme cestu a název souboru s obrázkem. Cesta může být buď absolutní, nebo relativní vzhledem k umístění souboru scény. Podporované formáty jsou BMP, GIF, JPEG, PNG a TIFF.

`Bitmapa`	Prototyp = `Tvar`
`zdroj`	`""`
`šířka`	`ŠířkaBitmapy(.zdroj)`
`výška`	`VýškaBitmapy(.zdroj)`

2.6. Kontejnery

Kontejner je element, který může obsahovat další elementy a specifickým způsobem je rozmisťuje. Pikta má kontejnery tři:

Plátno Umisťuje elementy na zadané souřadnice.

Mřížka Umisťuje elementy do svých buněk. Definuje se počet sloupců a řádků; všechny buňky jsou stejně velké.

Štos Umisťuje elementy za sebou buď horizontálně, nebo vertikálně a zalamuje je podle své velikosti.

Vkládání elementů do sebe je implementováno pomocí elementu `Kolekce`. To je speciální element, jehož hodnoty vlastností se intepretují jako prvky kolekce. Vlastnosti jsou pojmenovány přirozenými čísly a tím je také definováno uspořádání kolekce.

Prototypem každé kolekce je element `Kolekce`. Klonování kolekcí z jiného prototypu není podporováno.

Element `Kontejner` je pomocí kolekce definován takto:

`Kontejner`	Prototyp = `Tvar`
`obsah`	klon elementu `Kolekce`
`šířka`	`300`
`výška`	`200`

2.6.1. Klonování kontejnerů

Protože skládání prvků do kontejnerů představuje kompozici místo běžné aso-

ciace, je nutné aplikovat jiný způsob klonování, než je mělké[8]. Klonujeme-li kontejner, přirozeně očekáváme, že se naklonuje i jeho obsah.

Při klonování kontejneru Pikta postupuje takto:

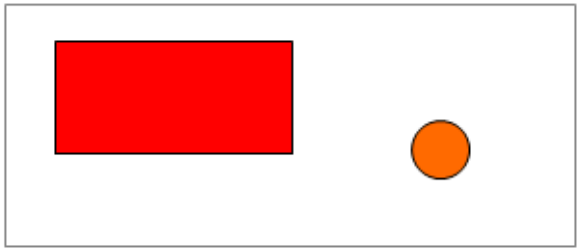
1. Klonuje kontejner obvyklým způsobem.
2. Klonuje element `Kolekce` a uloží ho do vlastnosti `obsah` nového kontejneru.
3. Pro každý element v původním kontejneru přidá do nového kontejneru jeho klon.

Ve zbytku této části si popíšeme jednotlivé kontejnery.

2.6.2. Plátno

Plátno je nejjednodušším kontejnerem (viz obrázek 2.). Své podelementy umisťuje na absolutní pozice určené jejich vlastnostmi `x` a `y`. Počátek souřadnic je v levém horním rohu, souřadnice se zvětšují směrem doprava a dolů.

Vlastnosti `x` a `y` nejsou definovány v žádném prototypu, protože jsou specifické pouze pro situaci, kdy je element umístěný v plátně.[9] Umístíme-li element na plátno, musíme mu tyto vlastnosti dodefinovat.



Obrázek 2. Plátno

2.6.3. Mřížka

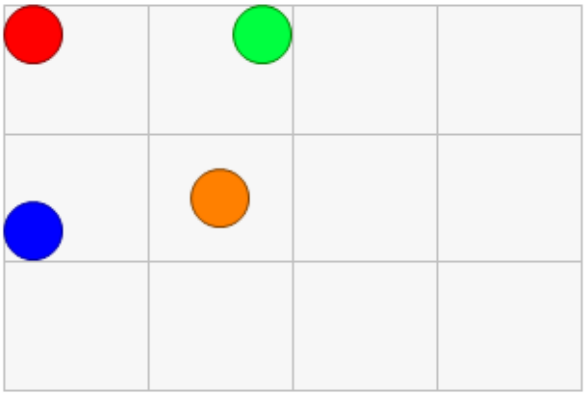
Mřížka je kontejner umisťující svoje podelementy do uniformních buněk (viz obrázek 3.). Počet buněk je určen hodnotami vlastností `řádky` a `sloupce`. Šířka buňky je určena jako `.šířka / .sloupce`, analogicky pak výška buňky.

Podobně jako v případě plátna mají i podelementy mřížky připojené vlastnosti – nazývají se `řádek` a `sloupec`. Hodnoty začínají od nuly a určují umístění elementu v mřížce.

Pozici elementu v buňce lze dále ovlivnit vlastnostmi pro horizontální a vertikální zarovnání, které mohou nabývat těchto hodnot:

[8] Více o tomto principu v [2]

[9] V knihovně WPF se tento princip nazývá *attached properties*.



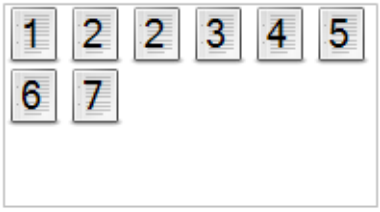
Obrázek 3. Mřížka

Vlastnost	Hodnoty
`horizontálníZarovnání`	`Vlevo`, `NaStřed`, `Vpravo`
`vertikálníZarovnání`	`Nahoru`, `NaStřed`, `Dolů`

Tabulka 1. Zarovnání

2.6.4. Štos

Štos je kontejner, který své podelementy řadí za sebe buď horizontálně, nebo vertikálně a zalamuje je podobně jako se zalamuje text na konci řádku (obrázek 4.). Orientaci štosu určuje jeho vlastnost `orientace`, která nabývá hodnot `Horizontální`, nebo `Vertikální`.



Obrázek 4. Horizontální štos

2.6.5. Zapouzdření

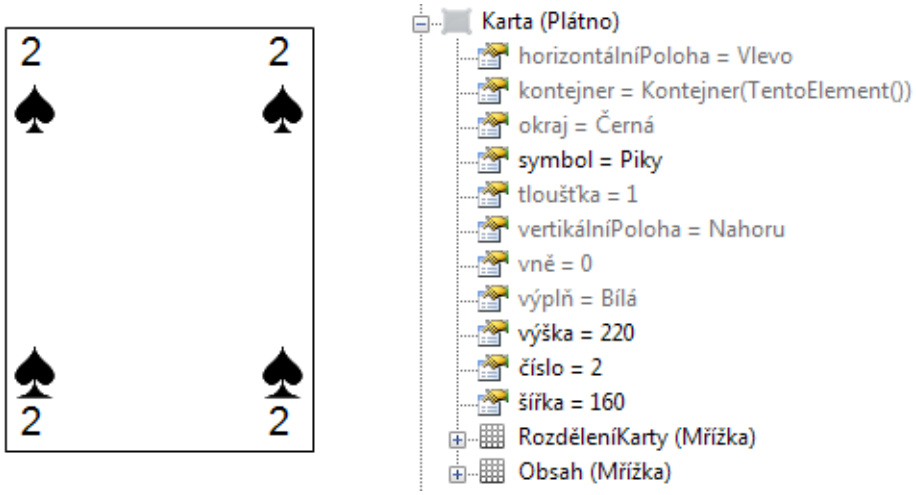
Kontejnery mají kromě uspořádání a umisťování elementů ještě jedno využití. Lze do nich „zapouzdřit“ složitější obrazec.

Na obrázku 5. vidíme příklad zapouzdření obrázku hrací karty[10]. Element `Karta` je potomkem plátna a obsahuje v sobě další prvky (mřížky pro rozmístění,

[10]Element `Karta` pochází z příkladu generování hracích karet v části 4.2.

4 bitmapy se symbolem, 4 texty s označením karty). Důležité vlastnosti, které charakterizují kartu, jsou její symbol (barva) a označení – `Karta` je má jako vlastnosti `symbol` a `číslo`. Bitmapy a texty, které karta obsahuje, jsou na tyto vlastnosti navázány.

Slovy objektových programovacích jazyků – zapouzdřili jsme abstraktní pojem hrací karty do jednoho objektu a implementační detaily (rozmístění hodnot do příslušných bitmap a textů) jsou skryty.



Obrázek 5. Zapouzdření hrací karty

2.7. Generátory

Zatím jsme si ukázali automatizaci výpočtu hodnot vlastností elementů z výrazů. Ještě chybí popsat nástroj pro automatizované vytváření samotných elementů. K tomuto účelu slouží v Piktě generátory.

Generátor je speciální neviditelný element, který popisuje vytváření dalších elementů. Vlastnosti prototypu generátoru a jejich výchozí hodnoty vypadají takto:

`Generátor`	Prototyp = `Báze`
`kontejner`	`Mřížka`
`počet`	`1`
`šablona`	`Báze`
`výsledek`	`Báze`

Obrázek 6. Prototyp generátoru

Spustíme-li v editoru generátor, provede následující:

1. Klonuje element ve vlastnosti `kontejner`.

2. Pokud je generátor v kontejneru, přidá klon do něho, jinak ho uloží do vlastnosti `výsledek`.

3. Vlastnost `počet` určí počet klonů elementu `šablonu`, které se do nového kontejneru přidají.

4. Každému z těchto klonů generátor dodefinuje vlastnost `i`, do které vloží jeho pořadové číslo počínaje 0.

Vlastnost `výsledek` se v editoru nepoužívá; generátor je vždy v kontejneru.

3. Editor Pikta

Editor Pikta je desktopová aplikace pro systém Windows umožňující vytváření a úpravy scén (kreseb). V této části práce naleznete příručku k jeho použití z uživatelského hlediska.

Příručka popisuje způsob ovládání editoru a všechny jeho funkce. Praktické ukázky spolu s postupy následují v kapitole 4.

3.1. Požadavky a instalace

Minimální systémové požadavky editoru jsou tyto:

- Procesor alespoň 1 GHz, 512 MB RAM, 600 MB místa na disku (pro instalaci .NET Frameworku).
- Operační systém Windows XP nebo novější.

3.1.1. Instalace editoru

Editor lze nainstalovat do počítače v několika málo krocích:

1. Instalaci spustíme programem *setup.exe*.
2. Pokud v počítači chybí .NET Framework 4, automaticky se nainstaluje.
3. Potvrdíme dotaz, zda chceme aplikaci instalovat, přestože se nepodařilo ověřit vydavatele. Důvodem tohoto upozornění je, že aplikace nemá digitální podpis – nedisponuji totiž potřebným certifikátem.
4. Editor se nainstaluje a spustí. Do nabídky Start se přidá zástupce.

3.1.2. Odinstalace

Editor se odinstaluje standardním způsobem pomocí ovládacího panelu Přidat nebo odebrat programy.

3.2. Práce se soubory a historií

Stejně jako jiné aplikace pro Windows poskytuje editor Pikta standardní funkce pro práci se soubory. Příkazy *Nový*, *Otevřít*, *Uložit* a *Uložit jako* v nabídce *Soubor* není třeba představovat. Scény editor ukládá do souborů s příponou *.pikta*.

Všechny operace provedené se scénou – změny vlastností, přidávání elementů, přesuny atd. – se ukládají do historie a lze je vrátit příkazem *Zpět*. Naopak vrácené akce se dají opět provést příkazem *Znovu*.

Historie akcí je dostupná pouze po dobu, kdy je scéna otevřená. Při ukládání scény se neukládá a při jejím zavření se zruší.

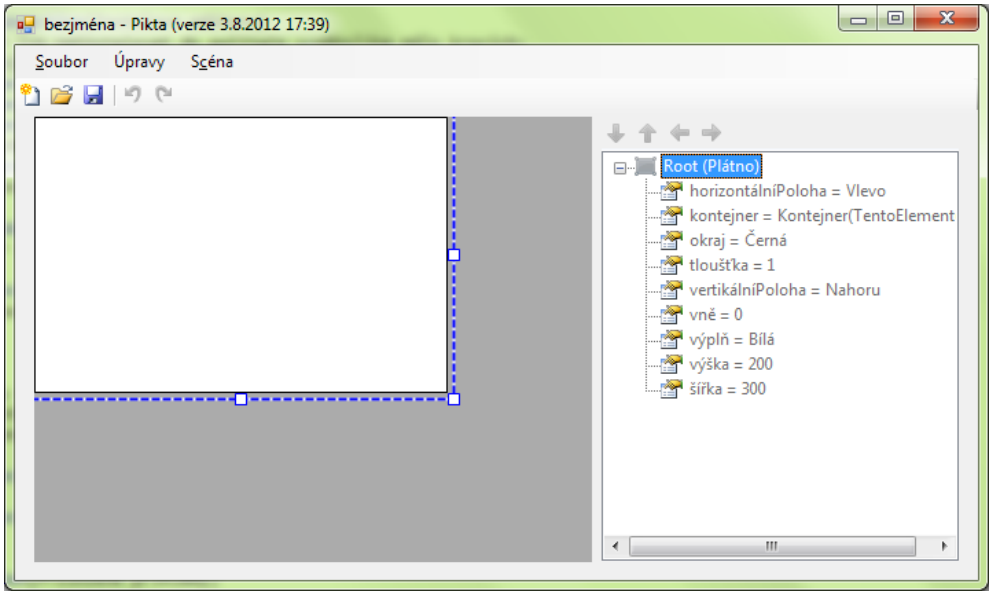
3.3. Export a tisk

Otevřenou scénu je možno z editoru vyexportovat do bitmapy ve formátu PNG. Exportuje se obraz elementu `Kořen` v poměru 1:1 a včetně alfa kanálu.

Dále je možné scénu vytisknout. Opět se tiskne pouze obraz elementu `Kořen` v poměru 1:1 a zarovnává se doprostřed stránky.

3.4. Scéna a její součásti

Při spuštění editoru se otevře nový dokument s prázdným plátnem. Vpravo je panel se stromem scény.



Obrázek 7. Editor s prázdnou scénou

Kresbu vytvářenou v editoru budeme nazývat *scéna*. Scéna je reprezentována stromovou strukturou elementů a její kořen tvoří speciální plátno nazvané `Kořen`. Element `Kořen` nelze přejmenovat, odstranit, ani ho nahradit jiným.

Panel stromu scény zobrazuje všechny elementy ve scéně spolu s jejich vlastnostmi. Elementy jsou zapisovány stylem *Název : NázevProtypu*. Nemá-li element název, zobrazí se místo něho hodnota `<anonymní>`. Základní typy grafických elementů jsou pro větší přehlednost také odlišeny ikonami.

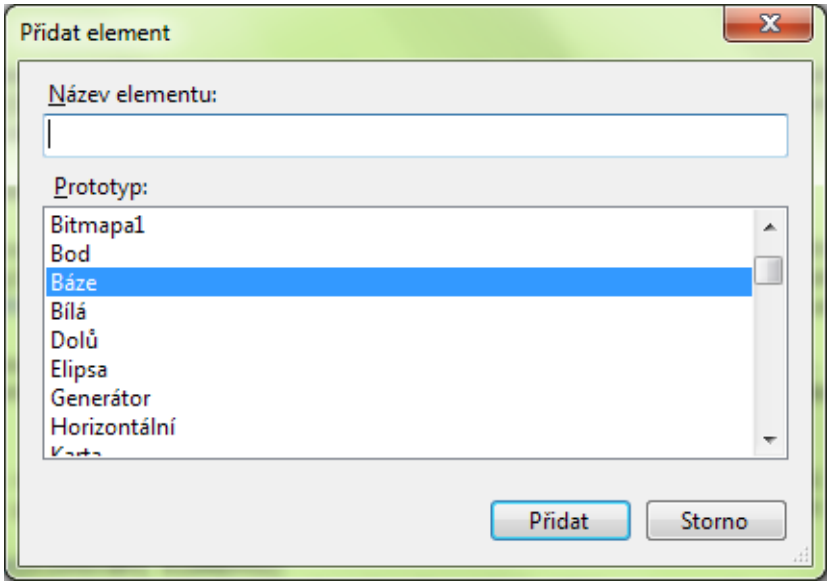
Vlastnosti jsou ve stromu zobrazeny ve formátu *vlastnost = výraz*. Je-li vlastnost v daném elementu definována lokálně, zobrazuje se černou barvou; je-li zděděná, vykresluje se šedě.

Pomocí panelu stromu scény lze provádět veškeré operace se scénou a jejími elementy. Operace lze vyvolávat pomocní místních nabídek elementu a vlastnosti. Výčet příkazů následuje.

3.4.1. Klonování elementu

Element lze klonovat dvěma způsoby. První možností je kliknout pravým tlačítkem myši na zvolený prototyp ve stromu a kliknout na příkaz *Klonovat*. Tento příkaz vytvoří klon vybraného elementu ve stejném kontejneru a přiřadí mu vygenerovaný název[11].

Druhou možností je klik pravým tlačítkem na kontejner a zvolení příkazu *Přidat element*. Příkaz otevře dialogové okno podobné jako na obrázku 8.



Obrázek 8. Dialogové okno Přidat element

V tomto okně je umožné zadat název nového elementu a vybrat prototyp ze seznamu všech pojmenovaných elementů, které jsou ve scéně, a navíc mnoha dalších, které editor definuje. V seznamu jsou i základní tvary a kontejnery – elipsa, bitmapa, plátno atd.

Necháme-li název elementu prázdný, vygeneruje se automaticky.

3.4.2. Odebrání elementu

Příkaz *Odstranit* v místní nabídce elementu provede odebrání elementu z jeho kontejneru a odstranění jeho jména. Výrazy, které na element odkazovaly, budou zneplatněny (viz část o reakcích na chyby 3.6.).

Je-li ale element protypem jiného elementu, jako prototyp zůstnane a ze scény tedy úplně nezmizí. Pouze se zobrazí jako anonymní.

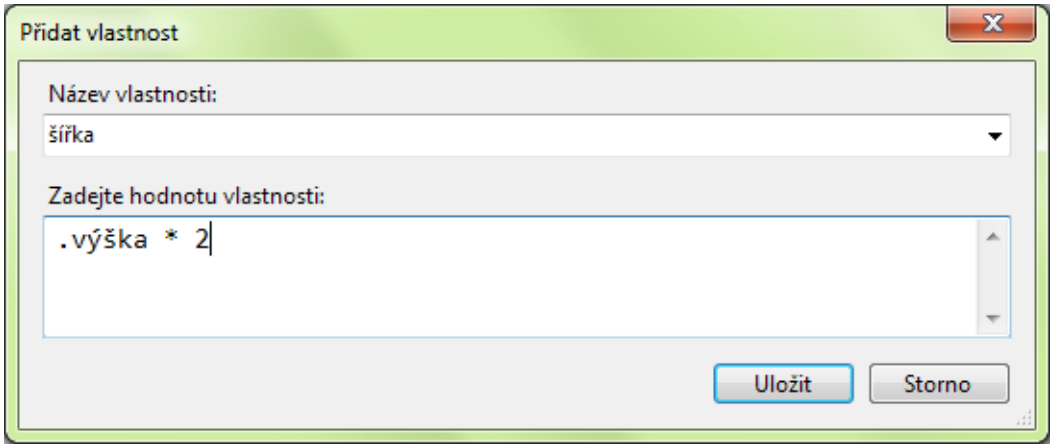
[11] Vygenerovaný název se skládá z názvu prototypu a pořadového čísla

3.4.3. Přejmenování elementu

Nový název můžeme elementu přiřadit příkazem *Přejmenovat*. Nutno ale poznamenat, že tato verze editoru nedokáže provést nahrazení všech odkazů na element novým názvem, proto může přejmenování zneplatnit některé výrazy. Ty je potřeba opravit ručně.

3.4.4. Přidání vlastnosti

Chceme-li definovat elementu vlastnost, použijeme příkaz *Přidat vlastnost*. Příkaz otevře dialogové okno, v němž je potřeba zadat název vlastnosti a výraz, který má obsahovat. Název vlastnosti můžeme také vybrat z rozbalovacího seznamu. V něm jsou uvedeny všechny vlastnosti předků elementu, které v něm ještě nebyly předefinovány.



Obrázek 9. Okno Přidat vlastnost

Vlastnost můžeme předefinovat také tak, že ji přímo upravíme – použijeme příkaz *Upravit* popsaný v odstavci níže na šedě zbarvené zděděné vlastnosti a zadáme její novou hodnotu.

3.4.5. Změna hodnoty vlastnosti

Hodnotu vlastnosti změníme příkazem *Upravit* (případně dvojklikem na vlastnost). Otevře se dialogové okno podobné tomu, pomocí kterého jsme vlastnost přidávali.

Je-li hodnotou vlastnosti barva, můžeme barvu vizuálně upravit pomocí příkazu *Upravit barvu*. Ten otevře standardní dialogové okno pro výběr barvy z Windows. Pomocí tohoto dialogového okna nelze zadat alfa kanál. Je-li to potřeba, lze ho potom upravit zvlášť přímo změnou hodnoty vlastnosti `a`.

Chceme-li jednu vlastnost svázat s druhou, můžeme namísto vypisování výrazu použít rychlejší přetažení myší. Má-li například výška tvaru vždy odpovídat

šířce, přetáhneme vlastnost `šířka` na vlastnost `výška`. Výraz pro výpočet výšky se změní na `.šířka`.

3.4.6. Odebrání vlastnosti

Vlastnost lze odebrat příkazem *Odebrat vlastnost*. Příkaz odebere definici vlastnosti z vybraného elementu. Definuje-li vlastnost i některý z předků elementu, bude se hodnota propagovat.

Pokud byla vlastnost definována poprvé v tomto elementu, je odstraněna úplně. To může způsobit zneplatnění výrazů, které na ni odkazovaly (viz část o reakcích na chyby 3.6.).

3.4.7. Změna pořadí elementů v kontejneru

Pomocí tlačítek se šipkami nahoru a dolů na panelu nástrojů stromu scény lze měnit pořadí elementů obsažených v kontejneru. Pořadí v kontejneru ovlivňuje překrývání při vykreslování elementů na plátně a v mřížce. Ve štosu určuje přímo pořadí a zalamování obsažených elementů.

3.4.8. Přesun elementu mezi kontejnery

Element lze přesunout do jiného kontejneru tak, že ho myší přetáhneme na požadovaný cílový kontejner.

Pro přesun elementu výše či níže v hierarchii kontejnerů slouží tlačítka se šipkami vlevo a vpravo na panelu nástrojů stromu scény. Tlačítko se šipkou doleva přesune element do kontejneru nadřazeného aktuálnímu. Tlačítko se šipkou doprava, přesune element do dalšího následujícího kontejneru na stejné úrovni.

3.4.9. Spuštění generátoru

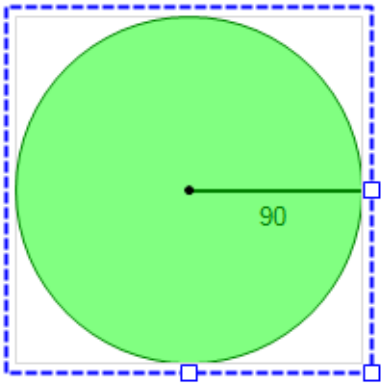
Pomocí příkazu místní nabídky *Spustit generátor* lze provést generování elementů podle vlastností vybraného generátoru.

Fungování generátorů jsme si teoreticky popsali v části 2.7. Prakticky si ho ukážeme na příkladu generování šachovnice v části 4.1.

3.5. Ovládání pohledu na scénu

Pohled na scénu zabírá největší část okna editoru. Zobrazuje všechny viditelné elementy scény v jejich grafické podobě.

Kliknutím na element ve scéně ho vybereme. Zároveň se vyhledá a zvýrazní ve stromu vpravo. Pokud klikneme na složitější element, který je složený z dalších, může se stát, že vybereme pouze některou součást. Celý element se nám nedaří vybrat, protože je úplně pokrytý součástmi. V takovém případě můžeme element vybrat tak, že na něj klikneme ve stromu scény.



Obrázek 10. Vybraný element

Je-li element obsažen v plátně, tažením myší můžeme změnit jeho pozici. Při změně pozice elementu se do jeho vlastností `x` a `y` uloží nové konstanty a přepíše se původní hodnota. Pokud byla pozice vypočítaná podle jiného elementu, vazba se zruší a přepíše se na konstantní.

Na hranách výběrového obdélníku jsou zobrazeny úchyty, jejichž tažením je možné změnit velikost elementu (vlastnosti `šířka` a `výška`). Opět platí to, co v předchozím odstavci – do vlastností se uloží konstanty.

Toto chování musíme mít na mysli, když konstruujeme objekty s vazbami na vlastnosti pozice a velikost. Když se například rozhodneme přidat do elementu reprezentujícím kružnici vlastnosti `středX`, `středY` a `poloměr`, měli bychom tyto vlastnosti brát jako sekundární a navázat je na hodnoty `x`, `y` a `šířka`, jinak přijdeme o možnost posouvat a zvětšovat element vizuálně.

V nabídce *Scéna* je přepínač *Periodická aktualizace*. Pokud je aktivovaný, dojde každou sekundu k automatickému překreslení scény – to je užitečné, pokud ve výrazu použijeme některou z časových funkcí (`Hodina()`, `Minuta()` nebo `Sekunda()`).

3.6. Reakce na chyby

Pokusíme-li se do vlastnosti přiřadit syntakticky špatný výraz nebo výraz, který nelze vyhodnotit, editor nám to nedovolí a zobrazí zprávu popisující chybu. Jsou ale případy, kdy editor chybu neoznámí a maskuje ji.

První takovým případem je situace, kdy se nějakou provedenou akcí zneplatní dříve platný výraz – např. odstraníme element, na který výraz odkazoval. Druhým případem jsou tzv. chyby interpretace. Když do vlastnosti `šířka` obdélníku uložíme výraz `"baf!"`, element je v pořádku, protože vlastnost nemá typ a tímpádem není omezena hodnota, kterou do ní lze uložit. Text `"baf!"` ale nelze převést na číslo, a je proto neplatnou hodnotou vlastnosti pro určení šířky obdélníku.

V obou těchto případech editor neupozorní na chybu a místo toho nahradí

chybnou hodnotu výchozí hodnotou pro danou vlastnost – např. pro vlastnost `šířka` je to `1`.

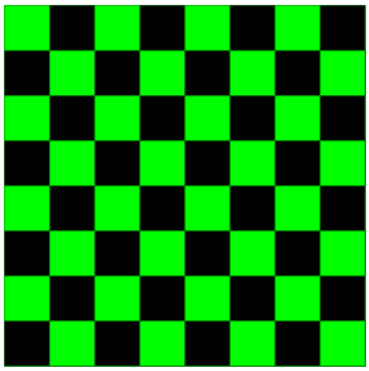
4. Přiložené příklady

Nyní si předvedeme, jak v editoru vytvořit několik složitějších scén. U každého z těchto příkladů je uveden postup, jak ho krok za krokem dokončit. Předpokládáme, že se čtenář seznámil s předchozími kapitolami o jazyce Pikta a ovládání editoru.

Scény s dokončenými příklady naleznete na přiloženém CD (viz příloha A.).

4.1. Šachovnice

Naším úkolem je vytvořit šachovnici, jako je na obrázku 11. Využijeme k tomu kontejner typu mřížka. Abychom políčko šachovnice nemuseli 64x ručně klonovat, použijeme také generátor.



Obrázek 11. Šachovnice

1. Nejdříve si připravíme prototyp mřížky. Naklonujeme element `Mřížka` a pojmenujeme klon `ŠablonaMřížky`.

2. Vlastnosti `sloupce` a `řádky` nastavíme na `8`.

3. Vlastnost `výška` nastavíme na `.šířka`. Pomůže nám to při zachování čtvercového tvaru šachovnice.

4. Nyní vytvoříme prototyp políčka šachovnice. Naklonujeme `Obdélník` a pojmenujeme klon `PolíčkoŠachovnice`.

5. Víme, že vygenerovaná políčka obdrží vlastnost `i` s pořadovým číslem. Protože toto číslo budeme potřebovat, definujeme vlastnost `i` už tady. Uložíme do ní hodnotu `0`.

6. Z pořadového čísla vypočítáme umístění políčka v mřížce. Do vlastnosti `sloupec` uložíme výraz `.i % ŠablonaMřížky.sloupce`, do vlastnosti

`řádek` uložíme `Celé(.i / ŠablonaMřížky.sloupce)`. Při určení řádku musíme oříznout desetinnou část výsledku. U sloupce to není potřeba, protože zbytek po dělení bude vždy celočíselný.

7. Vlastnost `okraj` nastavíme na `Průhledná`.

8. Vlastnost `výplň` nastavíme na hodnotu `Pokud((.řádek + .sloupec) % 2; Černá; Zelená)`

9. Nyní můžeme přistoupit k samotnému generování. Přidáme generátor pojmenovaný `GenerátorŠachovnice`. Jeho vlastnost `kontejner` nastavíme na `ŠablonaMřížky`, vlastnost `šablona` na `PolíčkoŠachovnice`.

10. Spustíme generování.

4.2. Hrací karty

Naším úkolem je vykreslit sadu klasických hracích karet francouzského typu. Scéna už nebude tak jednoduchá jako v předchozím příkladě, proto si musíme postup naplánovat.

4.2.1. Definování barev

Nejdříve nadefinujeme čtyři barvy.

1. Přidáme do scény vertikální štos a pojmenujeme ho `Barvy`.

2. Do štosu přidáme bitmapu, pojmenujeme ji `Piky` a pomocí vlastnosti `zdroj` načteme obrázek pikového symbolu.

3. Přidáme elementu vlastnost `barva` s hodnotou `Černá`. Tato vlastnost bude sloužit pro zbarvení popisek karty – musí být buď červená, nebo černá.

4. Podobným způsobem přidáme další 3 barvy – budou to elementy `Srdce`, `Káry` a `Kříže`.

4.2.2. Prototyp karty

1. Naklonujeme plátno a nový element pojmenujeme `Karta`. Změníme jí velikost na požadovaný tvar karty.

2. Přidáme jí vlastnost `barva` s hodnotou `Piky` a vlastnost `číslo` s hodnotou `2`.

3. Do karty přidáme mřížku pojmenovanou `RozděleníKarty` 2x2. Dále nastavíme `x = 0`, `y = 0`, `šířka = .kontejner.šířka` a `výška = .kontejner.výška`. Tím se mřížka roztáhne přes celou kartu. Mřížka bude sloužit pro rozmístění označení karty do jejích rohů.

   (a) Do mřížky `RozděleníKarty` přidáme další mřížku o dvou řádcích a jednom sloupci pojmenovanou `LN` (Levá Nahoře).

   (b) Do horní buňky nové mřížky přidáme bitmapu se zdrojem `.kontejner.kontejner.kontejner.symbol.zdroj` a patřičnou velikostí – bude ji potřeba zmenšit.

   (c) Do spodní buňky přidáme popisku s textem `.kontejner.kontejner.kontejner.číslo` a výplní `.kontejner.kontejner.kontejner.symbol.barva`. Zarovnáme popisku na střed buňky a zvětšíme písmo.

4. Podobným způsobem vytvoříme označení zbylých třech rohů.

5. Do elementu `Karta` přidáme další mřížku `Obsah`, která bude sloužit pro vložení symbolů barvy, které bývají na kartách uprostřed. Umístíme ji tak, aby měla od okraje karty dostatečně velkou mezeru (nebude pak překrývat již nakreslené symboly).

4.2.3. Reprezentace karetní řady

1. Do kořenu přidáme horizontální štos pojmenovaný `Řada` a nadefinujeme mu vlastnost `symbol = Piky`.

2. Do tohoto štosu naklonujeme element `Karta` a vlastnost `symbol` mu předefinujeme na `.kontejner.symbol`. Bude představovat eso – nastavíme mu `číslo = "A"`.

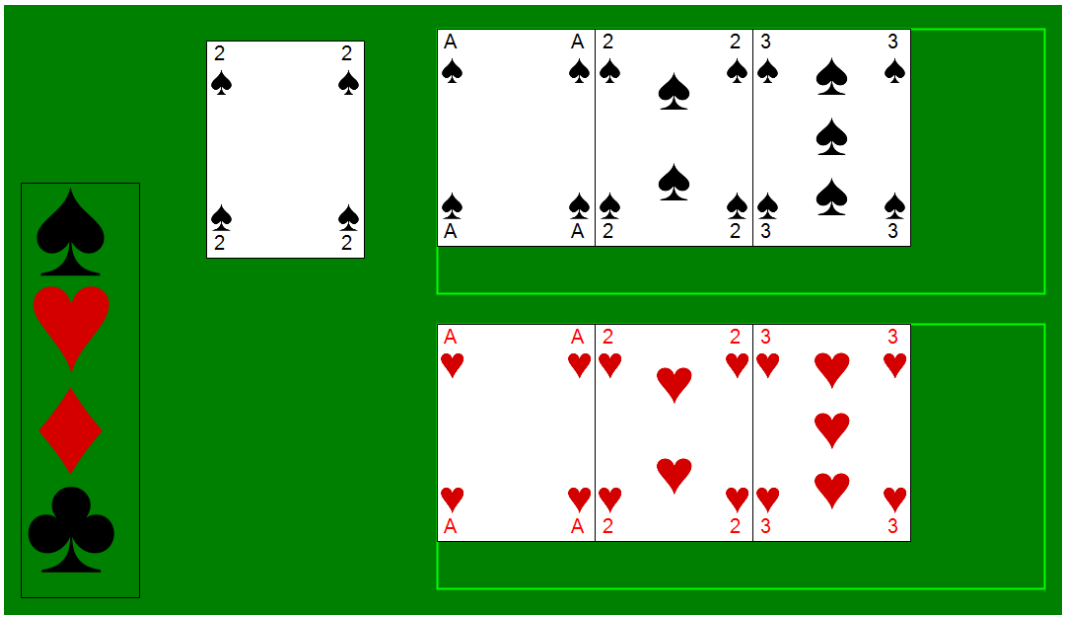
3. Eso naklonujeme opět do štosu, změníme `číslo = 2`. Do obsahové mřížky nové karty vložíme dva bitmapové symboly se zdrojem `.kontejner.kontejner.symbol.zdroj`, které budou o něco větší.

4. Podobným způsobem bychom mohli pokračovat při vytváření dalších karet.

5. Efekt karetní řady si můžeme vyzkoušet změnou vlastnosti `Řada.symbol`. Měly by se změnit symboly na všech kartách v řadě.

6. Celá karetní řada se dá po dokončení naklonovat a vznikne tak nová řada s jiným symbolem apod.

Konečný výsledek tohoto příkladu naleznete na přiloženém CD.



Obrázek 12. Karty

5. Popis implementace

Editor Pikta byl vyvinut v prostředí *Visual Studio 2012 RC* v jazyce *C#* na platformě *.NET Framework 4.0*.

Implementaci jsem rozdělil na dvě části. První tvoří knihovna jazyka Pikta, která definuje jazyk jako takový – elementy, vlastnosti, výrazy a jejich vyhodnocování. Knihovna není omezena na použití při popisu grafických scén. Pomocí knihovny lze vytvořit systém objektů, který je v této práci „shodou okolností“ interpretován jako grafika. Jazyková knihovna je popsána v části 5.2.

Druhou částí je samotný editor, který definuje základní tvary a kontejnery, poskytuje metody pro jejich vykreslování a v neposlední řadě řídí interaktivní práci se scénou. Implementace editoru je popsána v 5.3.

5.1. Resety a slepé uličky

Během vývoje aplikace jsem provedl několik zásadních změn jak v použité technologii, tak ve způsobu zobrazení scény, které měly za následek většinou úplné přepsání stávajícího kódu editoru.

5.1.1. Změny technologie vykreslování

První experimentální verze editoru vznikla pomocí knihoven *Windows Forms* a *GDI+*. Vykreslování přes *GDI+* funguje v tzv. *immediate módu*. To znamená, že se vykresluje pomocí sekvence grafických příkazů do bufferu, který operační

systém zkopíruje na obrazovku. Je-li potřeba obraz změnit, je potřeba provést znovu celé vykreslení[12]. Vykreslené grafické objekty nejsou interaktivní. Pokud chceme, aby např. reagovaly na klik myší, musíme napsat kód, který bude sledovat kliky myší nad celou vykreslovanou plochou a dokáže určit, na který prvek se kliklo.

Knihovna *WPF* nabízí o něco komfortnější práci s grafikou a navíc s ní lze jednoduše vytvořit interaktivní grafické objekty, které jsou schopny zpracovávat vstup uživatele samy. Proto jsem se rozhodl napsat editor znovu s pomocí *WPF* (tato verze je na obrázku 13.).

Knihovna *WPF* je primárně zaměřená na tzv. *retained módu*. To je způsob vykreslování, kdy aplikace popíše vykreslovanou scénu pomocí nějakého modelu a knihovna se stará o jeho vykreslování. Když je potřeba udělat změnu, aplikace změní model. V případě *WPF* je tímto modelem strom vizuálních objektů s kořenem v okně.

Z počátku jsem byl s *WPF* verzí editoru spokojený. Když jsem ale začal pracovat s netriviálními objekty (např. s Bézierovou křivkou tvořenou 10 a více řídícími body), narazil jsem na problém s výkonem – překreslování už nebylo plynulé a s rostoucí složitostí scény se dostalo až za hranici použitelnosti. Na vině nejspíše nebylo přímo vykreslování, ale změny objektového stromu.

Jediným řešením mohl být přechod na nižší úroveň vykreslování v immediate módu, což by ale znamenalo přijít o automatickou interaktivitu objektů a navíc výkon by ani tak nepředčil *GDI+*. Byl jsem nucen provést další reset editoru a vytvořit novou verzi pomocí *GDI+*.

Dá se podle této zkušenosti usoudit, že knihovna *WPF* je krok zpět? Myslím si, že rozhodně ne. Při vytváření složitých uživatelských rozhraní jsou její vlastnosti – jako například binding, styly a šablony – neocenitelné. Jako knihovna pro vykreslování grafiky se mi ale (nejenom v tomto projektu) neosvědčila.

Přesto *WPF* inspirovalo mnoho rysů Pikty, jak jsem popisoval v kapitole 1.2.3.

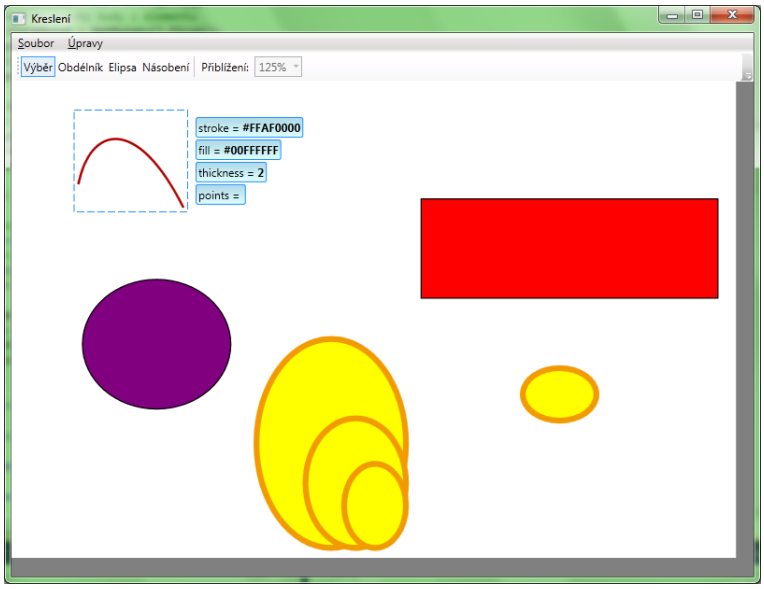
5.1.2. Změny zobrazení vlastností

V prvních verzích editoru jsem se pokoušel vyhnout se zažité tradici grafických editorů, kdy jsou vlastnosti vybraného objektu zobrazeny v samostatném okně na okraji obrazovky.

Seznam vlastností jsem zobrazoval přímo ve scéně, jak je vidět na obrázku 13.

Toto zobrazení je jednodušší v tom, že nerozmisťuje aspekty jednoho objektu do různých pohledů, ale nabízí je pohromadě na jednom místě. Problém ale nastane – jako vždy – s rostoucí složitostí. Je-li zobrazených vlastností mnoho, začnou ve scéně vadit, protože překrývají ostatní objekty. Další potíže může způsobit zobrazení vlastností více vybraných objektů, nebo objektů, které jsou mimo scénu či jsou příliš velké.

[12]Lze samozřejmě aktualizovat i pouze určitý výřez obrazu – nalezení objektů v takovémto výřezu může být ale komplikované, proto se většinou překresluje celý obraz.



Obrázek 13. Starší verze editoru

Proto jsem se rozhodl tento koncept opustit a vrátit se k tradičnímu zobrazení vlastností. Vlastnosti nyní zobrazuji ve stromu elementů v pravé části okna.

5.2. Implementace jazyka

Jazyková knihovna je implementována v projektu *Pikta*. Třídy v něm obsažené lze rozdělit do dvou velkých kategorií. Jedna modeluje elementy a vlastnosti, druhá řeší práci s výrazy.

5.2.1. Elementy a vlastnosti

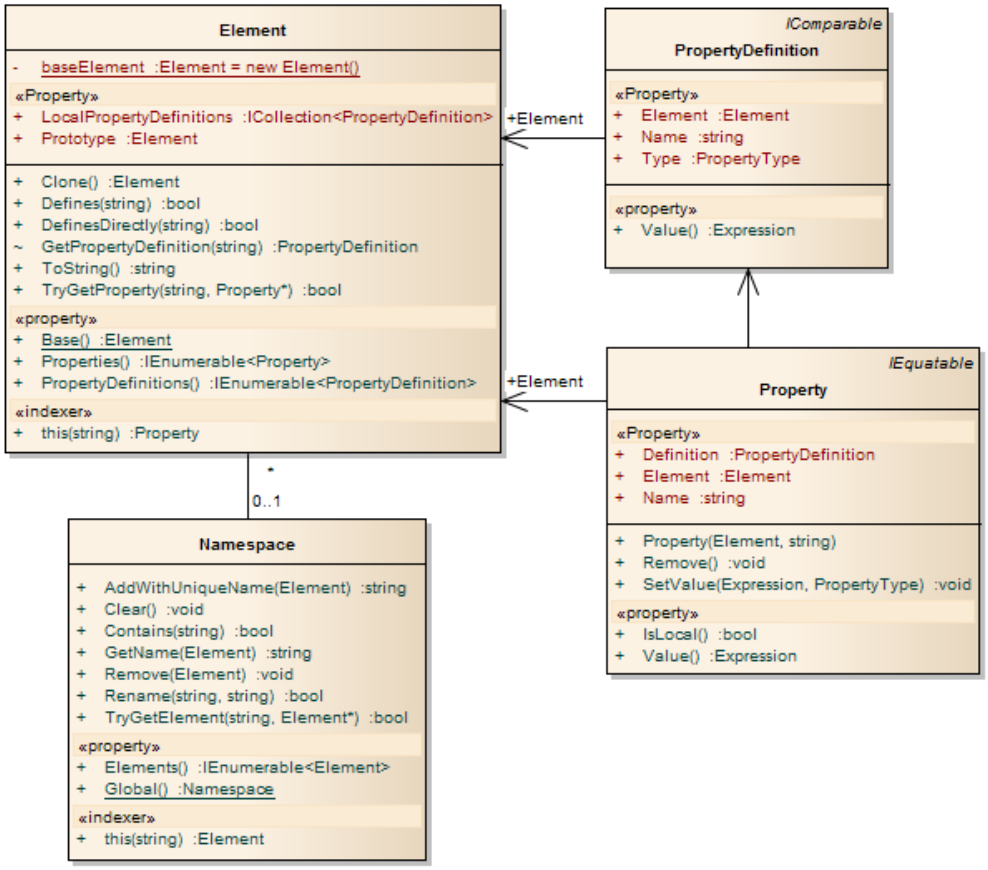
Tato část je implementována poměrně přímočaře a pochopit ji není složité. Hlavní třídy jsou zobrazeny v diagramu na obrázku 14.

Element Popisuje element jazyka. Obsahuje odkaz na prototyp a seznam lokálně definovaných vlastností.

PropertyDefinition Popisuje definici vlastnosti – její název a výraz – v určitém elementu.

Property Představuje (i nepřímo definovanou) vlastnost daného elementu.

Namespace Spravuje mapování názvů na elementy. Třída je používána jako singleton s jedinou instancí ve statické vlastnosti *Namespace.Global*. Toto zjednodušení nebylo příliš šťastnou volbou. Znemožňuje to v rámci jedné instance editoru otevřít dvě scény.



Obrázek 14. Hlavní třídy jazykové knihovny

Serializací elementů se zabývá třída *ElementSerializer*. Umožňuje načítat a ukládat elementy do formátu XML. Metoda *Load* slouží pro načtení elementů obsažených v souboru do jmenného prostoru. Používá se jak pro načítání scén (v tom případě je v souboru jen jeden element nejvyšší úrovně – kořen), tak pro načítání definičních souborů editoru. Metoda *Save* ukládá do souboru zadaný element.

Protože ukládané elementy mohou obecně tvořit graf, musí se na vhodných místech použít odkazy – např. když se ze dvou míst odkazuje na týž element, nesmí se uložit jako dva.

Příklad XML reprezentace scény z obrázku 2.:

```
<PiktaDocument>
  <Element id="1" name="Kořen" prototype="Plátno">
    <Property name="obsah">
      <Element id="2" prototype="Kolekce">
```

```
        <Property name="1073741823">
          <Element id="3" name="Obdélník1" prototype="Obdélník">
            <Property name="x" value="44" />
            <Property name="y" value="29" />
            <Property name="šířka" value="123" />
            <Property name="výška" value="58" />
            <Property name="výplň" value="Červená" />
          </Element>
        </Property>
        <Property name="1610612735">
          <Element id="4" name="Elipsa1" prototype="Elipsa">
            <Property name="šířka" value="30" />
            <Property name="výška" value=".šířka" />
            <Property name="x" value="229" />
            <Property name="y" value="70" />
            <Property name="výplň" value="Oranžová" />
          </Element>
        </Property>
      </Element>
    </Property>
    <Property name="šířka" value="326" />
    <Property name="výška" value="148" />
  </Element>
</PiktaDocument>
```

5.2.2. Výrazy

Větší část jazykové knihovny tvoří kód pro zpracování výrazů. Výrazy je potřeba parsovat z řetězců do stromové reprezentace, vyhodnocovat a v případě potřeby opět uložit do řetězce.

Reprezentace výrazů Výrazy jsou interně reprezentovány stromovou strukturou. Listy tohoto stromu tvoří konstanty a symboly (názvy elementů, vlastností a funkcí). Vnitřní uzly jsou n-ární operátory.

Jazyk obsahuje tyto druhy výrazů:

- Konstanta (číslo, text, element).
- Symbol (označení pojmenovaného elementu nebo vlastnosti).
- Binární aritmetické operátory (+, -, *, /, %).
- Unární operátor negace.
- Volání funkce.

Všechny třídy výrazů jsou implementovány jako neměnné (immutable).

Parsování výrazů Výrazy se parsují z řetězcové podoby pomocí třídy *Parser*, která využívá třídu *Lex*.

Třída *Lex* provádí syntaktickou analýzu a předává parseru seznam lexikálních symbolů. Většinu práce zde zastanou funkce pro práci s regulárními výrazy.

Třída *Parser* provádějící sémantickou analýzu a generující výrazový strom je o poznání větší. Pro sémantickou analýzu se nepoužívá žádný formální model, jedná se o ad-hoc analyzátor.

Opačný převod – ze stromu výrazů na řetězec – provádějí jednotlivé třídy výrazů samy pomocí metody *ToString*. Tento zpětně vygenerovaný zápis pravděpodobně nebude stejný jako ten, ze kterého výraz vznikl. Při převodu na text se přidávají nadbytečné závorky a n-ární aritmetické operace se převádějí na binární. Například z výrazu

```
20 + 10 + 30
```

parsováním a zpětným vygenerováním řetězce dostaneme výraz, který je sice ekvivalentní, ale o něco méně čitelný:

```
((20 + 10) + 30)
```

Této změně by šlo předejít dvěma způsoby: první by byl uložit si původní textovou podobu výrazu a použít ji (tím by se zachovalo i uživatelovo formátování). Druhý univerzálnější způsob by byl vytvořit algoritmus pro optimalizaci závorek.

Parser je jedním z příkladů problémů, při jejichž vývoji se výborně hodí podpora unit testů – více o tom v části 5.5.1.

Vyhodnocování Třída *Evaluation* provádí vyhodnocování výrazu v kontextu zadaného elementu. Vyhodnocování probíhá rekurzivně po celém výrazovém stromu.

5.3. Implementace editoru

Samotný editor Pikta, který staví na výše uvedené knihovně, se nachází v projektu *Pikta.Editor*. Editor pro GUI včetně vykreslování scény používá knihovnu *Windows Forms* (potažmo *GDI+*). Při popisu implementace editoru bych se chtěl zaměřit na dvě oblasti – zobrazení scény a historie akci – které mi přijdou nejzajímavější.

5.3.1. Zobrazení scény

Vykreslení samotné scény má na starosti třída *View*. Její hlavní zodpovědností je napojit se na předaný ovládací prvek z GUI a vykreslovat do něho obsah scény.

Vlastní vykreslování různých druhů elementů není ve třídě centralizované, ale deleguje se na potomky třídy *Visual*, přičemž pro každý druh vykreslovaného elementu existuje jeden tento „vizuální handler“.

Za poznámku stojí netriviální vykreslování bitmap v příslušném handleru - třídě *BitmapVisual*. Protože při každém vykreslení má handler pouze název souboru s bitmapou a bylo by neefektivní při každém překreslení bitmapu načítat a zase uvolnit, používá se na bitmapy keš (třída *BitmapCache*). Keš načtené bitmapy udržuje v paměti po jistý interval, a poté je automaticky uvolní.

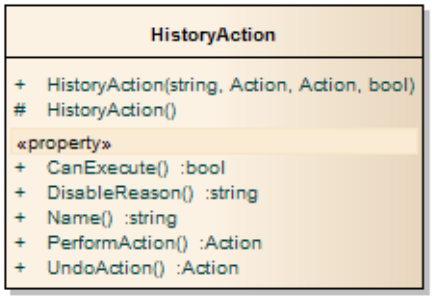
Třída *View* také ukládá obdélníky ohraničující vykreslené elementy, které jsou potřeba při zpracování vstupu od uživatele; zjišťování, na který element se kliklo apod.

Na třídu pohledu navazuje *InputProcessing*, která odchytává události práce s myší a řídí vybírání elementů a jejich přetahování.

Panel stromu scény tvoří samostatný ovládací prvek nazvaný *SceneTree*. Stará se o naplnění stromu scény podle aktuálního stavu, a také zpracovává uživatelský vstup.

5.3.2. Vracení a historie akcí

Provádění akcí v editoru je založeno na návrhovém vzoru *Command*. Každá akce je založená na třídě *HistoryAction* (viz obrázek 15.), která poskytuje název akce pro zobrazení v UI, důvod, proč případně není možné akci provést, a dvě metody, z nichž první akci provede a druhá ji vezme zpět.



Obrázek 15. Třída HistoryAction

Historii akcí udržuje klasický *HistoryManager* se dvěma zásobníky. Bohužel už jsem se k tomu nedostal, ale stálo by za to tyto akce ověřovat v unit testech.

5.4. Poznámky k lokalizaci

Každého uživatele zběhlého v programovacích jazycích hned po spuštění Pikty upoutají české názvy všech elementů, vlastností a funkcí. Záměrně jsem se rozhodl použít češtinu, abych Piktu více odlišil od běžných vývojových prostředí. Pikta

ale není v této oblasti průkopníkem. České názvy funkcí používá i např. *Microsoft Excel*.

V případě názvů elementů a vlastnosti mi zde čeština přijde naprosto přirozená. Trochu jiné je to u názvů funkcí – tam jsem se snažil vyhnout se použití sloves v rozkazovacím způsobu (Zaokrouhli, VraťPrvekNaPozici atd.), neboť se běžně v české komunikaci s počítači nepoužívají (Např. soubor neotvíráme příkazem Otevři, ale Otevřít.), ačkoliv by tu byly zcela na místě.

Nakonec jsem v názvech funkcí nepoužil sloveso ani jedno. Funkce se tedy maskují za jakési „rozšířené vlastnosti“.

5.5. Další prvky implementace

Rád bych se zmínil o několika zajímavých knihovnách a technikách, které jsou v implementaci použity. Tou první je použití unit testů k automatizaci testování kódu. Další dvě jsou knihovny usnadňující návrh a čitelnost kódu.

5.5.1. Unit testy

Unit testing je jeden ze způsobů automatizovaného testování softwaru na nejnižší úrovni samotnými vývojáři. Unit testy testují chování jednotlivých metod a tříd. Typický unit test je metoda s následující strukturou:

1. Inicializuje se testovaný objekt.
2. Stanoví se očekávaný výsledek testované metody.
3. Testovaná metoda se spustí s definovanými parametry.
4. Skutečný výsledek se porovná s očekávaným.
5. Na základě porovnání výsledků se určí, zda test prošel, nebo selhal.

Dostatečný počet unit testů pokrývající veškerou funkčnost slouží jako ochrana proti zanesení chyb do již napsaného kódu. Pokaždé, když přidáme novou funkci nebo provedeme refaktoring existujícího kódu, spuštěním všech unit testů se můžeme ujistit, že jsme neovlivnili žádnou jinou funkci.

V projektech *Pikta.Tests* a *Pikta.Editor.Tests* je několik unit testů, které pokrývají kritické části implementace Pikty – zejména práci s výrazy.

Více informací o využití unit testů při vývoji softwaru naleznete např. v [3].

5.5.2. Další knihovny

Reactive Extensions Knihovna pro modelování asynchronních proudů a dotazování se nad nimi. Více v [6].

Code Contracts Knihovna umožňující definovat predkondice, postkondice a invarianty se jejich propagování v hierarchii tříd. Více např. v [5].

Ookii Dialogs Pro zobrazování dialogových oken kompatibilních s *Windows Vista* se v editoru využívá volně dostupná knihovna *Ookii Dialogs*.

6. Možnosti dalšího rozvoje

Editor Pikta je v mnoha ohledech velmi nedokonalý a k možnosti reálného nasazení mu zbývá ještě dlouhá cesta. Na závěr uvedu několik úkolů, které by bylo dobré splnit při hypotetickém dalším vývoji aplikace.

Přidat další primitivní tvary a transformace V editoru chybí kompletní sada základních tvarů: čáry, křivky a polygony. Taktéž přidání grafických transformací by zvýšilo jeho použitelnost.

Odstranit omezení jazyka výrazů V jazyce chybí možnost zadat konstantní element a napsat výraz typu `Funkce().vlastnost`. Oba tyto nedostatky lze poměrně snadno obejít, přesto však při použití mohou překážet.

Přidat export do vektorů Export scény do vektorového formátu by přinesl možnost nad dokončenou kresbou v některém pokročilejším vektorovém editoru provést „postprodukci“.

Vyřešit internacionalizaci Umožnit překlad editoru do jiných jazyků a to včetně názvů elementů, vlastností a funkcí.

Zamyslet se nad metodami Stálo by za zvážení, kam by se editor dostal, kdyby se elementům přidalo chování v podobě metod nebo reakcí na události. Přestal by být grafickým editorem a přiblížil by se programovacím prostředím?

Závěr

Práce si kladla za cíl vytvořit inovativní grafický editor, který by pokryl mezeru mezi klasickými vektorovými editory a programovatelnými prostředími pro tvorbu UI. Vznikla v ní experimentální verze grafického editoru Pikta postavená na prototypovém objektovém systému s jednoduchými datovými typy a možností svazovat objekty pomocí aritmetických výrazů v jejich vlastnostech. Dále umožňuje seskupovat objekty do kontejnerů, které mohou sloužit jako jednoduché zapouzdření složitejší kresby.

Současná verze editoru Pikta demonstruje základní koncepty, pro nasazení do praxe se ale příliš nehodí, protože mu chybí grafické možnosti, na kterých by uživatelé objektovým systémem mohli stavět.

Conclusions

The thesis has set the goal to create an innovative graphics editor which would fill up the space between classic vector editors and programmable environments for UI creation. The experimental version of Pikta graphics editor was created based on the prototype object system with simple data types and the possibility to bind objects by arithmetic expressions in their properties. It also allows to group objects into containers which can serve as a simple encapsulation of a complex object graph.

The current Pikta graphics editor version demonstrates basic concepts, but it is not well suited for practice deployment. This is caused by lack of graphics options upon which the users can build using the object system.

Reference

[1] Laštovička, Jan. *Hranice prototypových programovacích jazyků.* Přírodovědecká fakulta Univerzity Palackého v Olomouci, 2011.

[2] Becerra, Gabriel. *Prototype Pattern: A Solution to Cloning Issues.* University of Calgary, Calgary, 2004.

[3] Beck, Kent. *Test-Driven Development: By Example* Addison-Wesley, Boston (MA), 2003.

[4] Apple Computer. *SK8 User Guide* Version 0.9, 1995. Dostupné online z `http://rubenkleiman.com/data/sk8/SK8_UserGuide.zip` [citováno 2012-08-07 23:58].

[5] Microsoft Corporation. *Code Contracts User Manual.* Dostupné online 2012-01-08 [citováno 2012-08-08 7:35].

[6] Cambell, Lee. *Introduction to Rx.* Dostupné online z `http://www.introtorx.com/` [citováno 2012-08-08 7:35].

A. Obsah přiloženého CD

V samotném závěru práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. závazné adresářové struktury, důležitých souborů apod.

`bin/`
: Instalátor editoru SETUP.EXE.

`doc/`
: Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace (v ZIP archivu), tj. zdrojový text dokumentace, vložené obrázky, apod.

`src/`
: Zdrojové kódy editoru Pikta.

`readme.txt`
: Instrukce pro instalaci a spuštění programu Pikta, včetně požadavků pro jeho provoz.

Navíc CD/DVD obsahuje:

`data/`
: Ukázková a testovací data použitá v práci a pro potřeby obhajoby práce.

U veškerých odjinud převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro materiály, u kterých toto není splněno, je uveden jejich zdroj (webová adresa) v textu dokumentace práce nebo v souboru `readme.txt`.