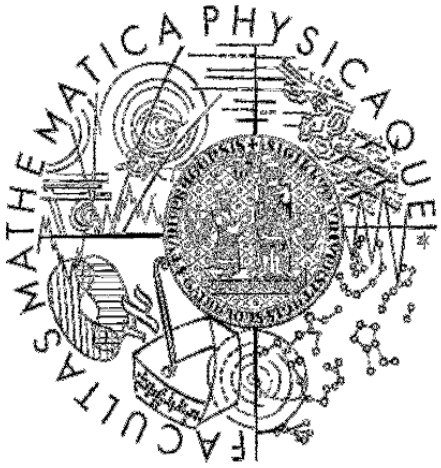
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Pavel Římský

Prostředí pro animaci algoritmů – interpret a řízení animace

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Rudolf Kryl

Studijní program: Informatika, programování

2006

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne Pavel Římský

Obsah

Název práce:	Prostředí pro animaci algoritmů – interpret a řízení animace
Autor:	Pavel Římský
Katedra:	Kabinet software a výuky informatiky
Vedoucí bakalářské práce:	RNDr. Rudolf Kryl
e-mail vedoucího:	rudolf.kryl@mff.cuni.cz

Abstrakt

Tato bakalářská práce pojednává o některých hlavních komponentách softwarového projektu AAnim. Projekt AAnim je prostředí (vytvořené v jazyce Java) pro vizualizaci algoritmů, které umožňuje psát vlastní programy v pseudokódu (jazyk *AL*), krokovat je, nastavovat breakpointy, sledovat hodnoty proměnných pomocí tabulky i pomocí obrázku, měnit hodnoty proměnných za běhu algoritmu a ukazovat průběh výpočtu konkrétních algoritmů ve speciálním panelu (tzv. *animátoru*). Hlavní důraz byl kladen na snadnou rozšiřitelnost – jazyk AL lze rozšiřovat o nové funkce, procedury a datové typy (pomocí tzv. *modulů*), lze přidávat nové zobrazovače proměnných a animátory. Tato práce pojednává o správci modulů, interpretu, řízení animace, editoru kódu, hlavních aspektech GUI a shrnuje zkušenosti autora nabyté při tvorbě tohoto projektu.

Klíčová slova

AAnim, animace, algoritmy, interpret

Title:	Environment for Animation of Algorithms - Interpreter and Animation Control
Author:	Pavel Římský
Department:	Department of Software and Computer Science Education
Supervisor:	RNDr. Rudolf Kryl
Supervisor's e-mail address:	rudolf.kryl@mff.cuni.cz

Abstract

This work deals with some of the main components of the AAnim software project. The AAnim project is an environment (written in Java) for visualisation of algorithms, the user can write their own programs in pseudocode (the *AL* language), set breakpoints, observe current values of variables in a table and also in a figure, modify the values of the variables when the program is running and watch the process of the algorithm in a special panel (called *animator*). The main emphasis has been put on extendability – it is possible to extend the AL language with new functions, procedures and data types (using so called *modules*), it is possible to add new renderes of variables' values and add new animators. This work deals with module management, interpreter, animation control, code editor, main GUI aspects and summarises experience acquired by the author during creation of the project.

Keywords

AAnim, animations, algorithms, interpreter

Předmluva

Softwarové dílo AAnim bylo vytvářeno dvěma studenty - Pavlem Římským a Petrem Štěpánem. Vzhledem k provázanosti prací jsou první dvě kapitoly bakalářských prací společné. V příloze A lze pak nalézt bližší informace o autorství jednotlivých částí AAnimu.

Abychom čtenáři umožnili získat alespoň základní představu o funkcích AAnimu, zařadili jsme první kapitolu, v níž se snažíme přiblížit základní dovednosti programu a používané pojmy. V druhé pak prezentujeme obecné informace o projektu (hlavní komponenty, popis vývoje atp.).

Pro úplné pochopení dalšího textu doporučujeme nejdříve přečíst uživatelskou dokumentaci dostupnou z prostředí programu AAnim (na CD přiloženém k této práci). Bylo by sice možné ji exportovat a po úpravách ji vytisknout, ale považovali jsme to za samoúčelné. Z nápovědy AAnimu je také dostupná generovaná programátorská dokumentace detailně popisující použité třídy a rozhraní.

Tato bakalářská práce pojednává o hlavních komponentách, jejichž autorem je Pavel Římský.

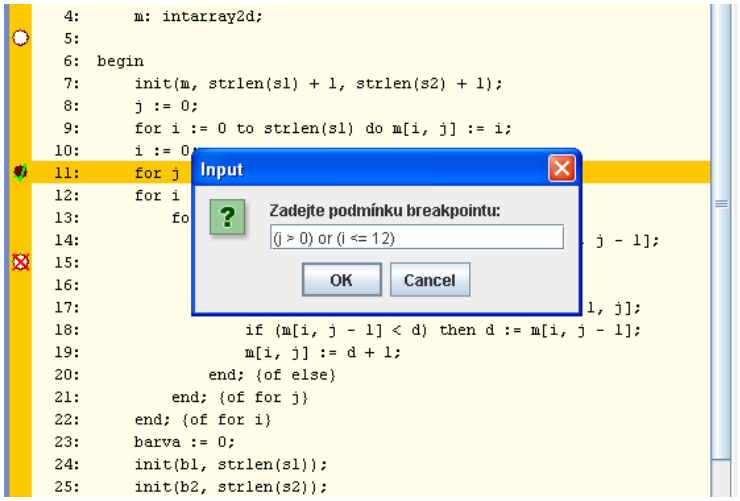
V kapitolách 3-10 popisuji objektový návrh komponent, které jsem vytvořil. Kromě objektového návrhu se zabývám i jeho motivací a zvažovanými alternativami.

Kapitola 1. Představení programu AAnim

V této úvodní kapitole čtenáři pomocí velkého množství screenshotů představíme program AAnim z uživatelského hlediska. Každý obrázek má za cíl ukázat jednu konkrétní dovednost tohoto programu.

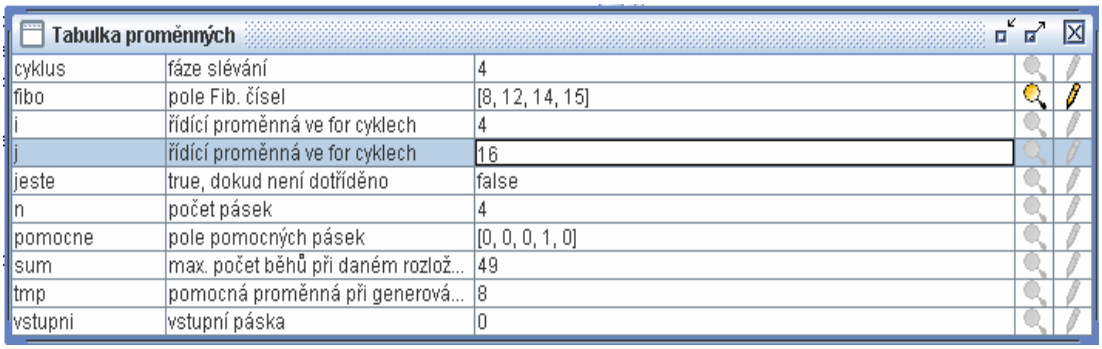
1.1 Zkoumání již existujících algoritmů

- **sledování, která část zdrojového kódu algoritmu se zrovna provádí, nastavování breakpointů a podmíněných breakpointů**



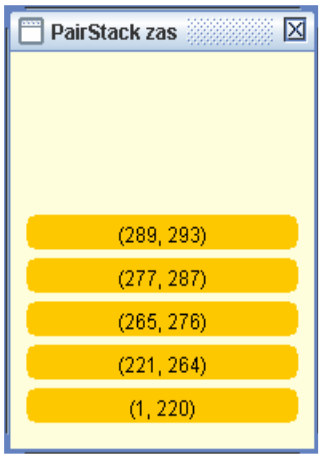
Obrázek 1.1.1: Nastavování breakpointů (levý pruh), podmínek breakpointů (dialog v popředí) a sledování, která část zdrojového kódu se provádí (oranžově zvýrazněný řádek)

- **sledování a změna aktuální hodnoty proměnných za běhu algoritmu**

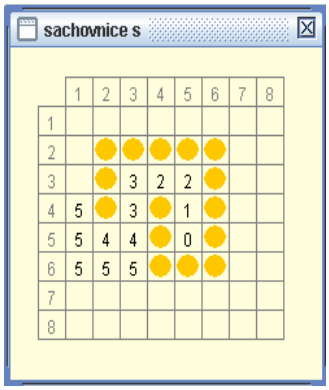


Obrázek 1.1.2: Sledování hodnot proměnných pomocí tabulky, hodnota proměnné `j` je právě měněna

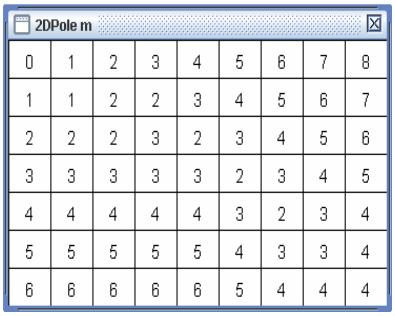
- **u některých datových typů sledování aktuální hodnoty proměnných pomocí speciálních zobrazovačů (tzv. *rendererů*)**



Obrázek 1.1.3: renderer typu `PairStack` *(zásobník intervalů)*

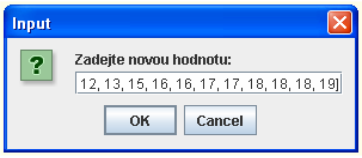


Obrázek 1.1.4: renderer typu `sachovnice`

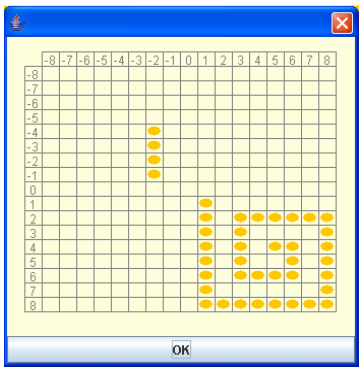


Obrázek 1.1.5: renderer typu `2DPole`

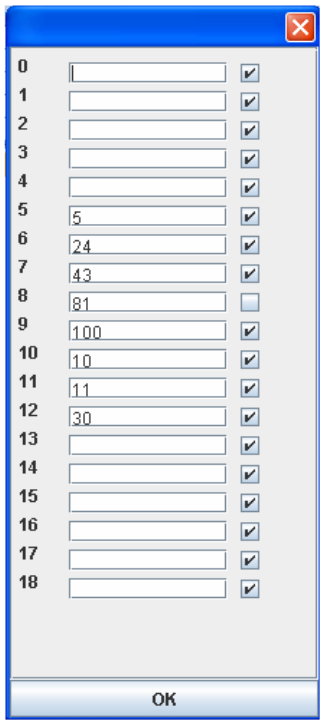
- **u některých datových typů změna aktuální hodnoty proměnných pomocí speciálních editorů**



Obrázek 1.1.6: editor typu `intarray`

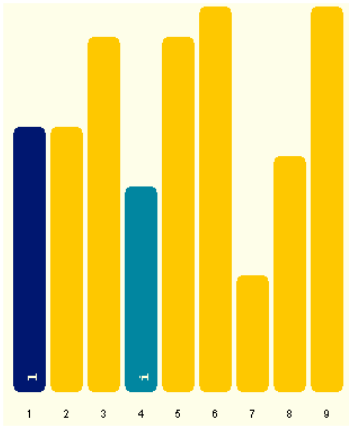


Obrázek 1.1.7: editor typu `polepozic`

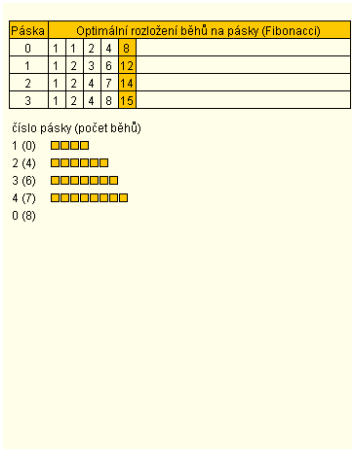


Obrázek 1.1.8: editor typu `hashtable`

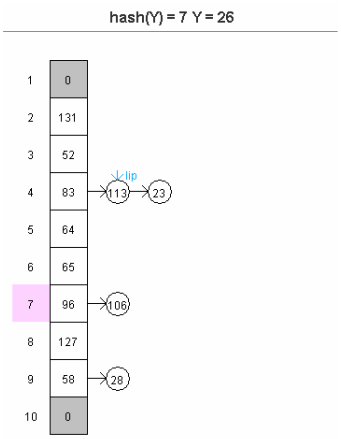
- **animování velkého množství algoritmů v samostatném panelu (tzv. *animátoru*)**



Obrázek 1.1.9: algoritmus vnitřního třídění



Obrázek 1.1.10: algoritmus polyfázového třídění

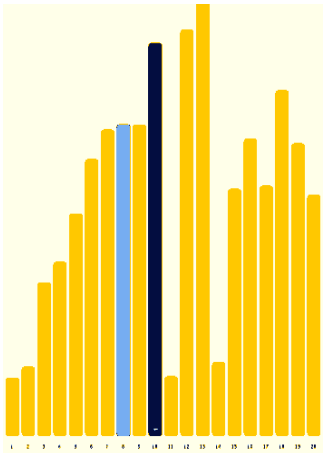


Obrázek 1.1.11: hashování s oblastí přetečení

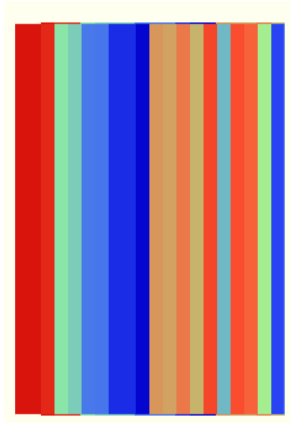
- **vybírání z více animátorů pro jeden algoritmus**

```
 1:  {TRIDICI ALGORITMUS INSERTSORT}
 2:
 3:  input a: intarray;
 4:
 5:  var i, j, tmp: int;
 6:      posunvpravo: bool;
 7:
 8:  begin
 9:      if sizeof(a) = 0 then a := randomSequence(20, 255);
10:      for i := 2 to sizeof(a) do begin
11:          tmp := a[i];
12:          j := i - 1;
13:          while (j > 0) and (a[j] > tmp) do begin
14:              a[j + 1] := a[j];
15:              j := j - 1;
16:          end;
17:          a[j + 1] := tmp;
18:      end;
19:  end;
```

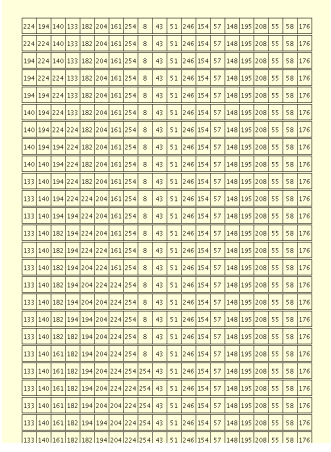
Obrázek 1.1.12: Zdrojový kód algoritmu vnitřního třídění (konkrétně insertsort)



Obrázek 1.1.13: vnitřní třídění pomocí sloupečků

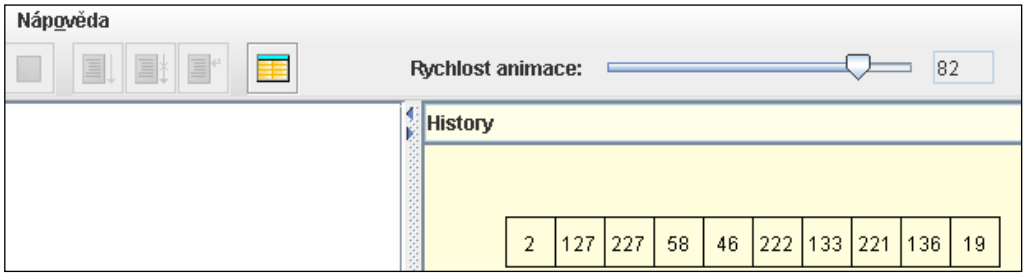


Obrázek 1.1.14: vnitřní třídění jako proužky barev



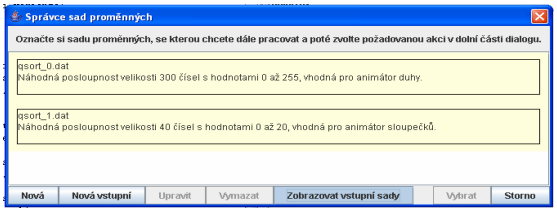
Obrázek 1.1.15: vnitřní třídění jako historie výpočtu

- **nastavování rychlosti provádění algoritmu**

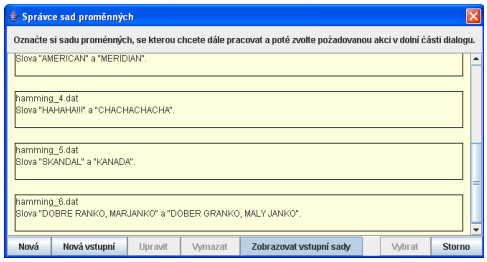


Obrázek 1.1.16: nastavování rychlosti provádění algoritmu

- **výběr z nabídky ukázkových vstupních dat algoritmů**

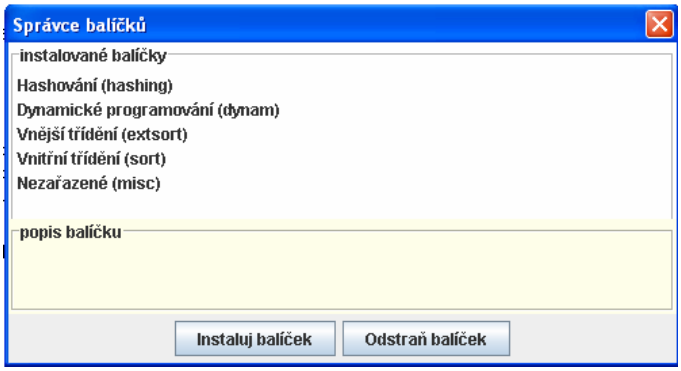


Obrázek 1.1.17: vstupní data pro algoritmus `quicksort`



Obrázek 1.1.18: vstupní data pro algoritmus Hammingova vzdálenost

- **rozšiřování AAnimu o nové algoritmy, animace či zobrazovače a editory hodnot proměnných instalací tzv. *balíčků***



Obrázek 1.1.19: Správce balíčků

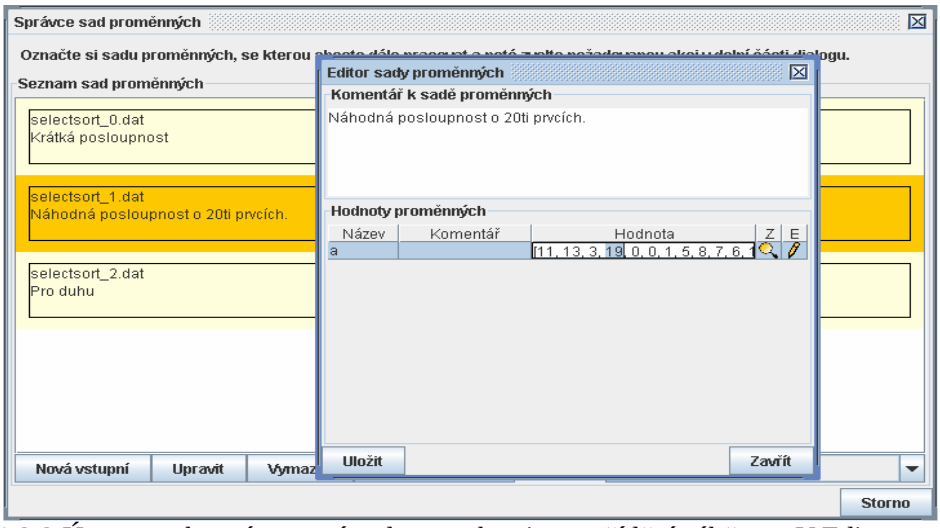
1.2 Psaní vlastních algoritmů

- **vytváření vlastního zdrojového kódu v AL (AAnim Language), lehce zvládnutelném, Pascalu podobném jazyku**

```
 1:  input
 2:      a: intarray;
 3:  var
 4:      i, minindex, j, tmp: int;
 5:  begin
 6:      if sizeof(a) = 0 then a := randomSequence(100, 255);
 7:      for i := 1 to sizeof(a) do begin
 8:          minindex := i;
 9:          for j := i + 1 to sizeof(a) do begin
10:              if a[j] < a[minindex] then
11:                  minindex := j;
12:          end;
13:          tmp := a[i];
14:          a[i] := a[minindex];
15:          a[minindex] := tmp;
16:      end;
17:  end;
```

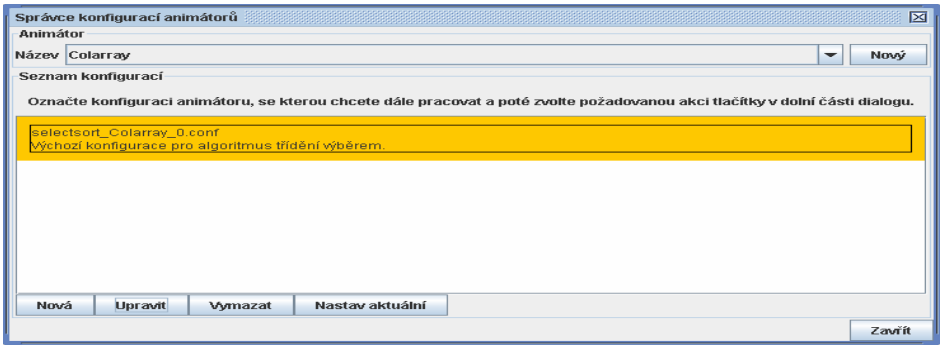
Obrázek 1.2.1 Algoritmus třídění výběrem v AL

- **vytváření, úprava a ukládání vstupních sad algoritmu v Editoru sad proměnných**

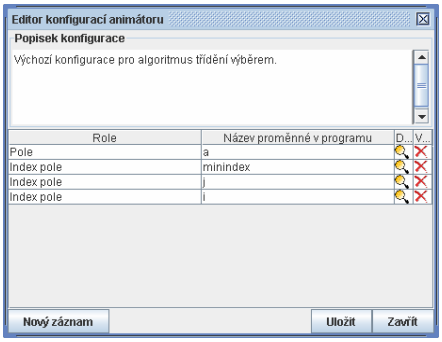


Obrázek 1.2.2 Úprava vybrané vstupní sady pro algoritmus třídění výběrem. V Editoru sad proměnných lze pohodlně upravovat hodnoty proměnných vstupní sady

- **možnost využít pro animace svého vlastního algoritmu již existujících animátorů vytvořením konfigurace animátoru – souboru, v němž je vybraným proměnným z programu přiřazena jistá role v animaci**

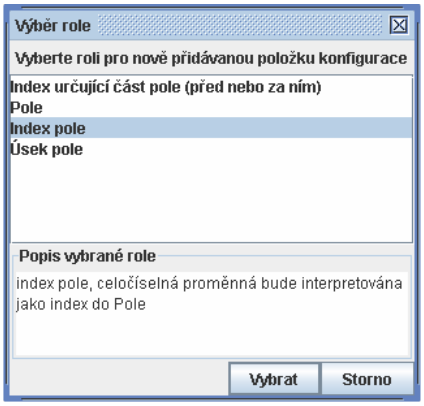


Obrázek 1.2.3 Správce konfigurací animátorů umožňuje vytvářet, mazat a upravovat již existující konfigurační soubory animátorů.

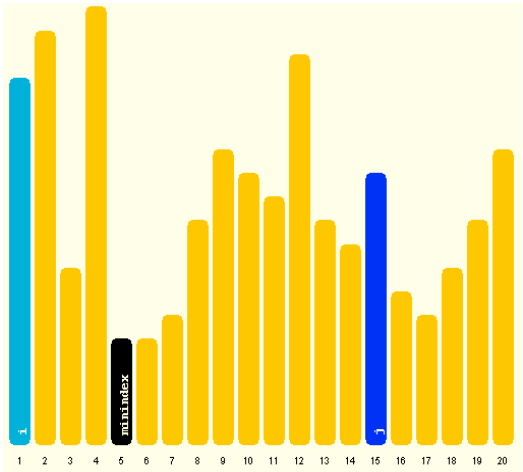


Každá proměnná, kterou animátor využívá, má přiřazenu při animaci nějakou roli. V Editoru konfigurací animátoru lze s těmito přiřazeními (položkami konfiguračního souboru) pracovat.

Role však může mít další atributy, které ovlivňují její chování při animaci, např. vzhled. Tyto atributy lze pro každou roli editovat v dialogu Detaily o roli.



Při přidávání nových položek do konfigurace se nejprve uživateli nabídne seznam animátorem nabízených rolí včetně jejich krátkého popisu.



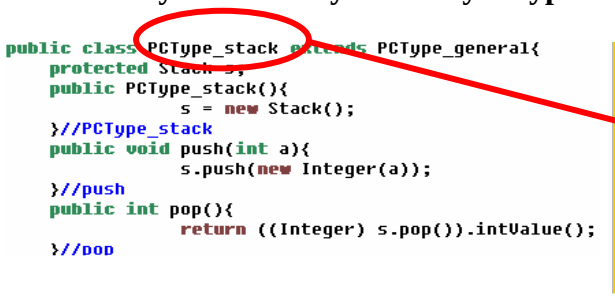
A konečně výsledek: animátor zobrazuje průběh algoritmu na základě informací z konfiguračního souboru.

Obrázek 1.2.4 Konfigurace animátoru

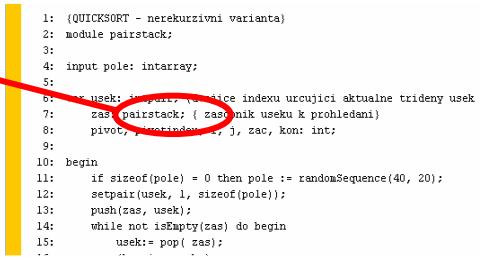
1.3 Rozšiřování AAnimu

Každý, kdo ovládá Javu, může rozšířit jazyk AL o nové datové typy, operace na datových typech, zobrazovače typů, editory typů, funkce či procedury (balíku takovýchto rozšíření říkejme *modul*), a to naprosto nezávisle na zbytku zdrojového kódu AAnimu (autor vůbec nemusí mít k dispozici zdrojový kód AAnimu). Stejně snadno se dá AAnim rozšířit o nové animátory.

- **vytváření nových datových typů**

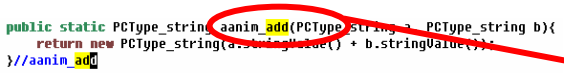


Obrázek 1.3.1:Výňatek ze zdrojového kódu Javy, kde se definuje nový datový typ.

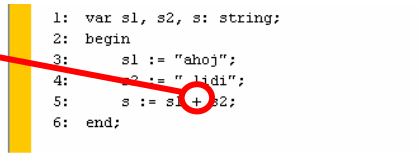


Obrázek 1.3.2: Použití takto definovaného typu v pseudokódu.

- **definování operací na těchto typech**



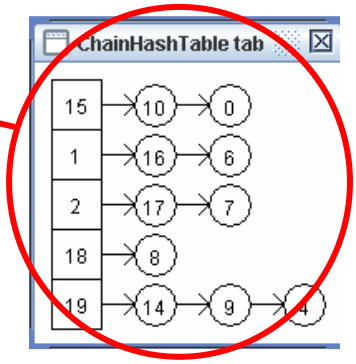
Obrázek 1.3.3:Výňatek ze zdrojového kódu Javy, kde se definuje operace sčítání na řetězcích.



Obrázek 1.3.4:Použití takto definované operace v pseudokódu.

- **definování vlastních zobrazovačů a editorů vlastních datových typů**

```
class ChainHashTableRenderer extends DefaultTypeRenderer{
    //hodnota, jež se vykresluje v zobrazovači
    private PCType_chainhashtable table;
    //konstanty určující vzhled
    private final int MARGIN= 10, ITEMSIZE= 30, CHAINITEMSIZE= 25 ;
    private final int ARROWLEN= 20, ARROWING= 9;
    private final Color bgColor= Color.white, emptyColor= Color.lightGray,
    fullColor= Color.white, fontColor= Color.black;
```



Obrázek 1.3.5:Výňatek ze zdrojového kódu Javy, kde se definuje zobrazovač hashovací tabulky s oblastí přetečení.

Obrázek 1.3.6: A jeho použití při animaci.

- **definování vlastních funkcí a procedur - rozšiřování jazyka AL**

```
public static void aanim_mergerunsto(PCType_chainhasharray tapes, PCType_int destination){
    int x = destination.intValue();
    for (int i=0; i < tapes.numtapes(); i++){
        if (i == x)
            tapes.get(i).addRuns(1);
        else
            tapes.get(i).removeRun();
    }//for
}//aanim_mergerunsto
```

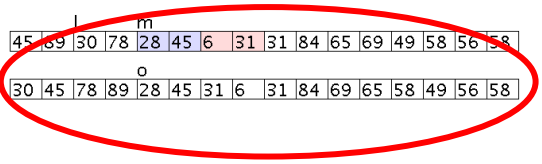
```
while pomocne[cyklus mod (n+1)]) > 0 do
mergeRunsTo(pomocne, (n+cyklus) mod (n+1));
```

Obrázek 1.3.7:Výňatek ze zdrojového kódu Javy, kde se definuje procedura `mergerunsto`.

Obrázek 1.3.8: Použití takto definované procedury v pseudokódu.

- **definování vlastních animátorů**

```
/** Animátor nerekurzivní verze algoritmu vnitřního třídění slučováním */
public class Merge extends General {
    private MyPanel panel = new MyPanel();
    private RectArray first= new RectArray();
    private RectArray second= new RectArray();
```



Obrázek 1.3.9:Výňatek ze zdrojového kódu Javy, kde se definuje animátor Merge.

Obrázek 1.3.10: A jeho použití při animaci.

Kapitola 2. Projekt AAnim

Základní popis

Program AAnim (*Algorithm Animation*) slouží jako pomůcka studentům při studiu jednodušších algoritmů. Zahrnuje vývojové prostředí, které umožňuje editaci algoritmu v pseudokódu, jeho krokování a sledování hodnot proměnných. AAnim je možné rozšířit o moduly (napsané v jazyce Java) dodávající do pseudokódu další funkce a procedury. K algoritmu lze také připojit tzv. *animátory*, panely, které animují průběh algoritmu.

Vznik projektu

Projekt AAnim začal vznikat na jaře roku 2005. RNDr. Rudolf Kryl potřeboval vytvořit několik animací algoritmů, které se přednášejí nebo by se mohly přednášet v kurzech programování na Matematicko-fyzikální fakultě UK v Praze. Po několika konzultacích se doktor Kryl rozhodl, že pro fakultu by byl vhodný program, který by dokázal nejenom zobrazovat průběh animace, ale šly by v něm nové animace i vytvářet. Algoritmy by se popisovaly jednoduchým jazykem, který by vycházel z Pascalu (dále v textu tento jazyk budeme nazývat *pseudokód* nebo *jazyk AL*). Pseudokód by měl být rozšiřitelný o nové funkce a procedury, přičemž ty by se nemusely psát přímo v pseudokódu, ale mělo by být možné o ně jazyk snadno obohatit.

Vzhledem k tomu, že se mělo jednat o poměrně náročný projekt, RNDr. Kryl usoudil, že by jej mohl svěřit hned dvěma studentům. Každý z nich by byl odpovědný za jednu část projektu. Po několika dalších konzultacích došlo k dohodě, že student Petr Štěpán vytvoří kompilátor pseudokódu (lexikální a syntaktickou analýzu) a student Pavel Římský vytvoří sémantickou analýzu a interpret.

Kompilátor a interpret však tvoří pouze jádro programu. Program se skládá z velkého množství komponent. Tyto komponenty jsou detailněji popsány v samostatné části každého z tvůrců programu.

Slepé uličky ve vývoji AAnimu

Původně jsme měli v úmyslu vytvořit několik na sobě nezávislých aplikací či apletů, každý z nich by zobrazoval jeden konkrétní algoritmus. Neuvažovali jsme však o tom, že by se tyto algoritmy definovaly v pseudokódu – původní návrh počítal s tím, že logika animovaných algoritmů by byla obsažena přímo ve zdrojovém kódu v Javě. RNDr. Kryl však již za začátku projevil zájem o univerzálnější nástroj, který by byl snadno rozšiřitelný.

Konkrétnější podoba našeho projektu se začala rýsovat teprve po několika sezeních s RNDr. Krylem. Nejprve měl být jazyk pseudokódu velmi jednoduchý, obsahovat pouze datový typ pro celé číslo a logickou hodnotu. Je zřejmé, že pro předvádění pokročilejších algoritmů tyto dva typy nestačí. Co víc, pokud měl být náš projekt univerzálně použitelným nástrojem, bylo potřeba umožnit uživateli, aby si definoval vlastní datové typy. Nelze totiž dopředu předvídat, jaké různé algoritmy bude kdo chtít naším programem animovat a jaké datové typy budou tyto algoritmy používat

(například zásobník je potřeba pro algoritmus quicksort, fronta pro některé grafové algoritmy).

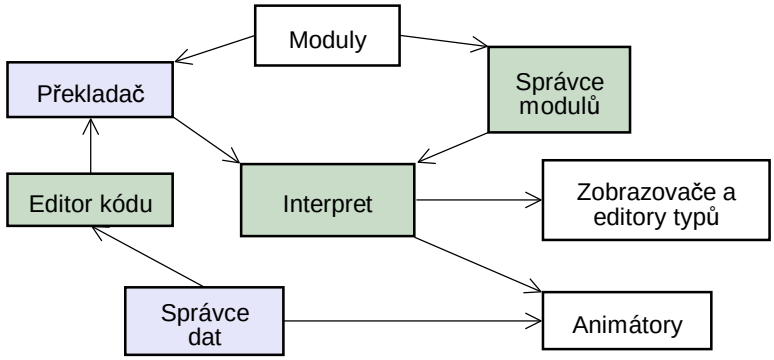
Jedním z prvních (s odstupem času lze říct že kuriózních) návrhů, jak by mohl náš program vypadat, byl generátor zdrojového kódu pro jazyk Java. Princip by byl takový, že autor animace by napsal algoritmus v pseudokódu, své vlastní datové typy, procedury a animace by napsal v Javě a náš program by z pseudokódu vygeneroval Javovský zdrojový kód, který by připojil k Javovskému kódu napsanému autorem animace. Tím by vznikl Javovský aplet, který by animoval určitý algoritmus. Vzhled tohoto apletu by mohl autor ovlivnit vhodnou konfigurací. Jednalo by se de facto o „továrnu na animace“. Cílem by bylo uživateli ulehčit psaní apletu. Za normálních okolností by se totiž programátor musel starat o řízení rychlosti animace, krokování, zobrazování pseudokódu a zvýrazňování prováděných řádků a spoustu dalších technických záležitostí. S naším generátorem apletů by se programování výrazně ulehčilo – část kódu obstarávající tuto nutnou režii by byla automaticky generovaná.

V zásadě se nejednalo o špatný nápad. Samozřejmě by uživatel neměl možnost pseudokód editovat (leda že by si potom vygeneroval nový aplet). Paralelně s tímto návrhem jsme však již pracovali na alternativě, k níž jsme se nakonec přiklonili. Na smetišti dějin pak návrh generátoru skončil po otázce RNDr. Kryla „Jakou by to mělo výhodu oproti tomu druhému řešení?“ a naší odpovědi „žádnou“.

Během celého vývoje projektu jsme byli stavěni před další a další rozhodnutí, kudy se má projekt dále ubírat. I zde jsme narazili na několik slepých uliček. Ty jsou detailněji popsány v samostatných částech obou autorů.

Hlavní komponenty AAnimu

AAnim sestává z mnoha komponent. Přikládáme velmi zjednodušené schéma (obrázek 2.1) a krátké popisy nejdůležitějších z nich, abychom zjednodušili čtenáři orientaci v dalším textu. Podrobnější informace lze nalézt v dalších kapitolách prací obou autorů (viz přílohu Zodpovědnost za jednotlivé části AAnimu).



Obrázek 2.1: Hlavní komponenty AAnimu – zeleně zvýrazněné komponenty vytvořil Pavel Římský

Specifika programu

Většina ostatních programů, které mají za cíl přiblížit studentovi princip fungování algoritmu pomocí animace jeho průběhu, se charakterem svého návrhu podobá filmu a studenta staví spíše do role pasivního pozorovatele s malými možnostmi ovlivnit průběh animace. Ty se ve většině případů omezují na změnu rychlosti probíhající animace či výběr z několika autorem předem připravených kolekcí vstupních dat. Zdrojový kód, pokud je vůbec možné ho zobrazit, potom slouží spíše k informaci uživateli, v jaké části algoritmu se právě vizualizace nachází.

My jsme se snažili (po různých peripetiích popsaných v předchozí části) navrhnout AAnim tak, aby poskytl studentovi prostředky k hlubšímu a aktivnímu zkoumání principu algoritmu. Hlavní vlastnosti a nástroje, které mu to umožní, shrnuje následující seznam.

- interpretovaný pseudokód

  Student může modifikovat již existující algoritmy a vidět, jak provedené změny ovlivní chování algoritmu, nebo vytvářet své vlastní programy.

- zobrazení a změna hodnot proměnných v průběhu algoritmu

  Pomocí tabulky proměnných lze zobrazovat i upravovat hodnoty jednodušších typů přímo, pro složitější typy existuje mechanismus zobrazovačů a editorů, které může uživatel využít i při absenci předpřipravené animace, tedy například pro své vlastní algoritmy.

- vstupní sady

  Načítaní iniciálních hodnot vstupních proměnných využijí zejména autoři algoritmů, kteří mohou připravit dostatečně „vysvětlující“ data.

- konfigurovatelné animační moduly (*animátory*)

  Hlavní prostředek vizualizace algoritmů, na základě hodnot proměnných, jež mu jsou poskytovány prostředím, vytváří animaci algoritmu. Možnost sdílet jeden animátor mezi více algoritmy a pomocí konfiguračních souborů přizpůsobit např. různému pojmenování proměnných stejného významu, zjednodušuje práci autorům animátorů, neboť je činí univerzálnějšími. Naproti tomu jeden algoritmus může být vizualizován více animátory, což může zlepšit studentovo porozumění danému programu.

Podrobnější výčet funkcí a dalších možností, jak rozšířit AAnim pomocí Javy, nalezne čtenář v dokumentaci (přístupné z nápovědy AAnimu).

Použití Javy v AAnimu

Ačkoliv jazyk Java nepatří k nejuniverzálnějším, pro tento typ projektu nabízí řadu výhod. Nejprve si shrňme obecné klady Javy. Java eliminuje chyby programátora, je jednoduchá. To umožňuje, aby se programátoři soustředili převážně na užitečnou práci a neztráceli tolik času hledáním chyb. Další výhodou Javy je bohatá standardní knihovna. Není třeba složitě instalovat přídavné moduly, takřka vše, co programátor bude při své práci potřebovat, je součástí standardní knihovny Javy. Tato knihovna je navíc podrobně a přehledně zdokumentovaná. Za vývojové nástroje a vývojové

prostředí se nemusí nic platit a programy napsané v Javě běží pod velkým množstvím běžně používaných operačních systémů.

Nyní je třeba si zodpovědět otázku, zda se nevýhody Javy nějak zásadně projeví na našem projektu. Javu řadíme mezi interpretované jazyky. Psát interpret v interpretovaném jazyce se může zdát nerozumné. Proto jsme se museli zamyslet nad tím, jestli rychlost Javy nebude představovat problém. Jelikož náš projekt má sloužit k vizualizaci algoritmů a typickým uživatelem bude student, který se snaží pochopit princip tohoto algoritmu, rychlost interpretace není podstatná. Algoritmy budou typicky krokovány, aby byl uživatel schopný zamyslet se nad jejich principem, často si budou uživatelé nastavovat breakpointy. Jelikož je projekt již téměř hotový, lze zpětně konstatovat, že jsme se v předpokladu nemýlili. Rychlost Javy opravdu není omezujícím faktorem.

Malá univerzalita Javy u tohoto typu projektu nevadí. Žádné speciální operace se nepoužívají. Naopak se ukázaly užitečné některé speciální vlastnosti Javy. Jde především o takzvané *Reflection API*, tedy možnost uložit zkompilovanou třídu do souboru a pomocí standardního rozhraní zjišťovat, jaké obsahuje metody a jakého jsou typu, a navíc vytvářet instance těchto metod. Reflection API je využito pro rozšiřování pseudokódu o nové funkce, procedury, datové typy a animace. Nutno ovšem podotknout, že ne vždy používání knihovny Javy urychluje práci. Existují i výjimky. O tom blíže pojednává část o editoru kódu.

Kapitola 3. Správa modulů

AAnim je možné rozšířit o takzvané moduly – přidáním modulu se jazyk AL (tak se jmenuje pseudokód používaný AAnimem) obohatí o nové datové typy, funkce a procedury. Z uživatelského hlediska jsou moduly popsány v uživatelské dokumentaci programu AAnim. Část nazvaná *popis jazyku pseudokódu* popisuje, jak jsou datové typy, funkce a procedury modulu přístupné z pseudokódu. Části nazvané *definice nových modulů* a *definice vlastních datových typů* popisují konvence pro vytváření tříd modulu.

Alternativy pro implementaci modulů

Mechanismus modulů lze implementovat několika způsoby. Základní kritéria, pro jaký způsob se rozhodnout, byla tato:

1. Musí to být jednoduché pro autora modulu.
2. Nemělo by se opakovat příliš mnoho stejných informací na různých místech.
3. Mělo by se předcházet tomu, aby chyba autora modulu narušila funkčnost AAnimu.
4. Již hotové moduly musí být možné snadno rozšířit.
5. Těla funkcí a procedur by měla být napsána v Javě, ne v pseudokódu (kvůli rychlosti).

První (nepoužitá) alternativa – modul sám rozhodne

První možností bylo pro každý modul vytvořit třídu a v ní metodu `executeFunction`, která by dostala v prvním parametru název funkce a v druhém seznam (např. `ArrayList`) jejích parametrů. Kdykoliv by uživatel v pseudokódu uvedl volání funkce `fn` s parametry `p1`, `p2` a `p3`, zavolala by se tato metoda, jako název funkce by se jí předalo „fn“ a seznam parametrů by byl `[p1, p2, p3]`. Tělo metody `executeFunction` by vypadalo takto:

```
if (nazevFce == "fce1"){
    /* proveď fci 1 */
}else if (nazevFce == "fce2"){
    /* proveď fci 2 */
}else if(nazevFce == "fce3"){
    /* proveď fci 3 */
}else...
```

Prvky seznamu parametrů by byly objekty nějaké třídy, která by zapouzdřovala jejich hodnotu a typ. Při takovéto implementaci by se dalo řešit i přetěžování funkcí. V klauzuli `if` by se netestoval jen název funkce, ale i počet a typy předaných parametrů. Je však potřeba vyřešit, jak při sémantické analýze ověřit, zda existuje funkce s nějakým názvem a typy parametrů. Taková informace by se musela nacházet ještě na nějakém jiném místě. Buď v samostatném souboru, nebo by se ve třídě definovala obdobná metoda, která by však funkci nevykonávala, ale testovala by její existenci.

Původní návrh navíc předpokládal, že třída modulu bude obstarávat i samotnou animaci - definuje se v ní metoda, která vrátí panel animátoru (objekt třídy `JPanel`). Tak by v podstatě pro každý modul (tedy každou sadu funkcí) existovala jen jedna animace. Jeden modul je však typicky využíván velkým množstvím algoritmů, zatímco téměř každý algoritmus má vlastní animátor. Proto jsme se rozhodli moduly od animací oddělit.

Tato alternativa porušuje podmínky 1 až 3. Tělo metody `executeFunction` by se s rostoucím počtem funkcí a procedur stávalo stále méně přehledným a pro uživatele by definování nových funkcí a procedur nebylo jednoduché. Navíc kvůli nutnosti kontrolovat existenci funkce by se testování názvu funkce a jejích parametrů v kódu dvakrát opakovalo. Kdyby si autor modulu nedal pozor na to, aby šla funkce vykonat, právě když ji sémantická analýza označila jako existující, modul by se nechoval korektně (např. algoritmus by se úspěšně přeložil, ale při běhu by zjistil, že nějaká funkce neexistuje).

Druhá (použitá) alternativa

Jazyk Java umožňuje uložit zkompilovanou třídu do souboru a pomocí standardního rozhraní zjišťovat, jaké obsahuje metody, jakého jsou typu a navíc vytvářet instance těchto metod a tyto instance volat. Toho využívá i náš projekt.

Každému modulu odpovídá třída Javy, jejíž autor musí dodržovat jisté konvence. Tyto konvence se týkají pojmenování samotných tříd, pojmenování jejich metod, modifikátorů tříd a jejich metod (veřejnost, statičnost) a umístění do balíčků. Podrobnější informace o těchto konvencích najdete v uživatelské části dokumentace k programu AAnim pod titulkem *definice nových modulů* a *definice vlastních datových typů*.

Tato alternativa se ukázala daleko pružnější než první zmíněná. Pro programátora je velmi pohodlná, píše totiž definice funkcí a procedur úplně stejně, jako by tvořil normální Javovský program. Konvence jsou poměrně jednoduché na zapamatování. Sémantická analýza se provádí snadno. Informace se neopakují zbytečně na více místech, tudíž je eliminováno riziko, že se autor modulu dopustí chyby, která ovlivní chod celého AAnimu. Rozšíření o nový podprogram se provede jednoduše přidáním nové metody do třídy modulu.

Správa modulů je řešena třídami z balíčku aanim.moduleman. Nejdůležitější třída tohoto balíčku se jmenuje `Module` a reprezentuje jeden modul. Její konstruktor přijímá jediný parametr, a to název modulu. Po vytvoření nové instance třídy `Module` se podle daných pojmenovacích konvencí najde třída modulu, zjistí se, jaké má metody, a ověří se, že splňují jisté podmínky (popsané v uživatelské dokumentaci). Vytvoří se instance třídy `Method`, které budou tyto metody reprezentovat, a uloží se do tabulky metod modulu. Tabulka metod modulu je spravována třídou `SubroutineTable`. Třída `SubroutineTable` si pamatuje všechny kombinace parametrů přetížených podprogramů a je schopna najít vhodné přetížení podle pravidel popsaných v uživatelské dokumentaci pod titulkem *popis jazyka pseudokódu*.

I datovým typům odpovídají třídy Javy. Každý datový typ musí být potomkem třídy `PCType_general`. Objektová hierarchie navíc určuje pravidla pro implicitní

přetypování. Typ `B` je přetypovatelný na typ *A*, právě když třída `PCType_`*B* je potomkem třídy `PCType_A`. Je-li v kódu algoritmu importován modul `X`, pak lze v pseudokódu použít právě ty datové typy, které jsou v aspoň jednom podprogramu modulu (speciální metodě třídy modulu nebo jejím předkovi) použity jako parametry nebo návratové typy. Lze tedy sdílet jeden datový typ mezi více moduly.

V modulech lze definovat čtyři typy podprogramů: funkci, proceduru, přiřazení do proměnné a zápis do pole. Každý z těchto typů podprogramu je reprezentován třídou Javy, jejíž název končí na „`Props`“ (balíček aanim.moduleman). Některé funkce mají speciální význam, překladač na jejich volání totiž převede výrazy s operátory. Tak například výraz `a + b` překladač přeloží jako by na jeho místě bylo psáno `add(a, b)`. Popis, které operátory odpovídají kterým funkcím lze najít v uživatelské dokumentaci. Přiřazení do proměnné (`AssignmentProps`) reprezentuje podprogram, který zajistí, že lze provádět operaci a `:= b` na určitém typu proměnné `a` a výrazu `b`. Nebyl by dobrý nápad, kdyby se zápis a `:= b` převáděl (obdobně jako u `a + b`) na volání procedury, například na `assign(a, b)`. Důvodem je možnost implicitního přetypování parametrů u procedury. Nechť bychom měli například proceduru `assign(real, real)`. Pak by ji šlo zavolat s parametry `(int, real)`, a tak přiřadit číslo typu real do čísla typu int! To je pochopitelně nežádoucí. Proto přiřazení je chápáno jako samostatný typ podprogramu a přetypovat lze pouze jeho druhý parametr. Z obdobných důvodů je jako samostatný typ podprogramu vnímáno přiřazení do pole. U zápisu `a[i1, i2, i3] := b` lze implicitně přetypovat parametry napravo od přiřazovátka a uvnitř hranatých závorek, ale v žádném případě ne typ proměnné `a`.

Předávání parametrů a jeden nepříjemný důsledek

Při volání podprogramu se postupuje následovně: vyhodnotí se parametry (uvedené v závorkách nebo vpravo či vlevo od přiřazovátka) a hodnoty se předají metodě `invoke` příslušné třídy podprogramu (třídy, jejíž název končí na `Props`). Hodnotou parametru je vždy (z pohledu Javy) objekt typu `PCType_general` nebo odvozeného typu. Jestliže je parametrem podprogramu proměnná, pak se nevytváří žádný nový objekt typu `PCType_general`, ale předá se rovnou ten objekt, který AAnim používá k reprezentaci hodnoty proměnné běžícího algoritmu. Tento mechanismus v kombinaci s tím, že objekty se v Javě předávají referencí, lze využít k implementaci podprogramů, které pracují s parametry předávanými odkazem.

Objekt typu `PCType_general` má totiž obvykle definované metody měnící jeho hodnotu (`setValue()`, `fromString()` a jiné). Pokud je zavoláme v metodě, která odpovídá nějakému pseudokódovskému podprogramu, a zároveň uživatel v pseudokódu tento podprogram zavolal s parametrem-proměnnou (nikoliv složitějším výrazem), změní se hodnota této proměnné. Takto musí být implementovány přiřazení a zápis do pole.

Nepříjemným důsledkem je fakt, že v sémantické analýze nelze nijak ověřit, jestli je někde vyžadován parametr-proměnná nebo je povolen i složitější výraz. Nechť máme například definovanou proceduru `NSDNSN`, která bere čtyři parametry. První dva jsou čísla *a*, *b*, další dva mají být proměnné předávané odkazem, do nichž se po

řadě uloží největší společný dělitel a nejmenší společný násobek. Tuto funkci bude možno zavolat i takto: `NSDNSN(5, 3, 12, 7)`. Program se přeloží, při běhu se nezhroutí, ale zavolání procedury se nijak neprojeví. A to proto, že AAnim z parametrů 7 a 12 vytvoří nové (dočasné) objekty typu `PCType_general`, na nich provede metoda odpovídající proceduře `NSDNSN` změnu hodnoty, ale tyto objekty pak zaniknou.

Až tak nepříjemný důsledek to není, uživatel si jen musí dát větší pozor na smysluplnost toho, co píše, a být si vědom, že podobný zápis je možný a chová se tak, jak se chová.

Kapitola 4. Interpret a jeho komunikace s ostatními komponentami

Základní principy

Popišme si, co se děje po spuštění algoritmu a během jeho provádění. Spustí-li uživatel algoritmus napsaný v pseudokódu, proběhne nejprve jeho kompilace. Vstupem kompilátoru je kód algoritmu a výstupem derivační strom. Derivační strom je reprezentován objekty třídy `Command` (nebo odvozených tříd) z balíčku aanim.translator. Třída `Command` je abstraktní a představuje obecný příkaz; její potomci pak konkrétní typy příkazů (if, for, blok begin..end, volání procedury, přiřazení, zápis do pole). Derivační strom jako celek je také příkaz, konkrétně blok begin..end (hlavní blok begin..end algoritmu).

Všichni potomci třídy `Command` mají dvě stěžejní metody – konstruktor a metodu `execute`. Konstruktor provádí nejen inicializaci, ale i sémantickou analýzu. Zjistí-li sémantickou chybu, vyvolá výjimku (třída `ETypecheck`). Parametry konstruktoru jsou mimo jiné syntaktické elementy, z nichž se příkaz skládá. Jaké elementy to konkrétně jsou, to záleží na typu příkazu (na tom, o jakého potomka třídy `Command` se jedná). Například konstruktor třídy `IfCommand`, reprezentující příkaz if, bere tři hlavní parametry, a to jsou podmínka (objekt typu `Expression` – výraz), důsledek (objekt typu `Command` – příkaz provádějící se v případě, že podmínka byla splněná) a alternativu (příkaz provádějící se v případě, že podmínka splněná nebyla).

Konstruktoru kompilátor předává i dodatečné informace: pozici ve zdrojovém kódu algoritmu, kde se daný příkaz nachází, použitý modul a typy proměnných deklarovaných v hlavičce kódu algoritmu. Pozice ve zdrojovém kódu (řádek a znak) je využívána v běhových chybových hláškách a hláškách sémantické analýzy a také při zvýrazňování řádků zdrojového kódu běžícího algoritmu. Typy použitých proměnných jsou využívány při sémantické analýze, stejně tak použitý modul.

Samotnou interpretaci zajišťuje metoda execute. Jejím jediným parametrem je objekt třídy `Environment` (v dalším textu proměnná `env`). Třída `Environment` uchovává informace o aktuálních hodnotách proměnných a také se stará o správné krokování. Typické tělo metody execute vypadá následovně:

```
1:  env.nextCommand();
2:  env.cmdEntered(line, c);
3:  if (env.isStopped()) return;
4:  /* vykonání příkazu */
5:  env.varsChanged();
```

Na prvním řádku zavoláme metodu třídy `Environment`, která zajišťuje správné krokování. Metoda `nextCommand` se uspí, dokud není možný přechod na další příkaz. Přechod na další příkaz je možný tehdy, jestliže uživatel stiskl tlačítko pro provedení dalšího kroku nebo je zapnuté automatické krokování a uběhl patřičný interval. Veškerou logiku spojenou s časováním obstarává třída `Environment`. Pokud uživatel algoritmus úplně zastaví, nečeká se na žádný signál, ale volání metody se rovnou vrátí.

Na druhém řádku se volá metoda `cmdEntered`, která oznámí, že se mohlo změnit číslo řádky, na níž se nachází prováděný příkaz. Třída `Enviroment` pak obeznámí další komponenty AAnimu, například editor kódu, aby zvýraznil právě prováděný řádek. V metodě `cmdEntered` se navíc ověří, jestli na aktuálním řádku není nastaven breakpoint. Pokud ano, zastaví se automatické krokování (je-li nastaveno) a metoda `cmdEntered` se uspí, dokud uživatel nestiskne tlačítko pro další krok nebo nezapne automatické krokování a neuběhne nastavený interval. Na třetím řádku se ověří, jestli uživatel algoritmus nezastavil. Pokud ano, provádění metody se ukončí. Pak následuje provádění příkazu. To je závislé na typu příkazu.

Je-li to potřeba, oznámíme třídě `Environment`, že došlo ke změně hodnot některých proměnných. K tomu slouží metoda `varsChanged`. Všude, kde se volá metoda `varsChanged`, měla by se zavolat i metoda `varChanged(String)`, která oznámí třídě `Environment`, že se změnila jedna konkrétní proměnná, a řekne která. Aby se snadno dalo zjistit, která proměnná se změnila a která ne (parametry funkcím a procedurám se předávají referencí, může se tak změnit v podstatě libovolná proměnná), v těle metody `execute` se nejprve uloží řetězcové reprezentace všech proměnných, které by se mohly změnit (získají se metodou `PCType_general.fromString()`) a po vykonání příkazu se tyto uložené řetězce porovnají s nově získanými.

V implementacích cyklů (`WhileCommand`, `ForCommand`) se před každým vykonáním těla cyklu znovu volají metody `cmdEntered` (aby se ve zdrojovém kódu zvýraznil řádek s hlavičkou cyklu) a `nextCommand` (aby na hlavičce chvíli zůstal). Rovněž se ověřuje, že algoritmus nebyl zastaven.

Po kompilaci, kdy už je derivační strom vytvořen a úspěšně proběhla sémantická analýza, se vytvoří nová instance třídy `Environment` (proměnná `env`). Této instanci se předají názvy a typy všech proměnných, proměnné v pseudokódu deklarované klíčovým slovem `input` se označí jako vstupní. Nastaví se některé důležité parametry, jako je rychlost krokování. Pokud algoritmus používá vstupní proměnné, je uživateli nabídnuto vybrat vhodnou vstupní sadu. Dále je objekt `env` informován o nastavených breakpointech, tabulce proměnných se předá `env` jako zdroj dat. Nakonec se vytvoří objekt reprezentující animátor, vytvoří se nové vlákno (třída `RunningAlgorithm`) a v tomto novém vlákně se zavolá metoda `execute` těla algoritmu.

Uživatel může ovlivnit běh algoritmu pomocí ovládacích prvků hlavního okna. Hlavní okno volá příslušné metody třídy `Environment`. Jedná se především o metody `goOn`, `setAutosteps`, `setStepsInterval` a `stop`. Tyto metody jsou popsány ve vygenerované programátorské dokumentaci k AAnimu. Ovládací prvky zase mohou být informovány o průběhu algoritmu (změně hodnot proměnných, změně řádku prováděného příkazu, ...). Stačí se zaregistrovat jako listener těchto událostí. K tomu slouží metody `addCommandEnterListener`, `addVarChangedListener` a `addVarsChangedListener`. Tyto metody včetně příslušných rozhraní, která musí listener implementovat, jsou rovněž popsány ve vygenerované programátorské dokumentaci.

Alternativní postupy, jak vytvořit interpret

Jednou z možností by bylo místo derivačního stromu používat mezikód. Jedná se však o daleko složitější způsob implementace, pro tento typ projektu až zbytečně složitý. Navíc by se v něm špatně reprezentovala čísla řádků a jedné instrukci pseudokódu by odpovídalo více instrukcí mezikódu, tudíž by se špatně definovalo, co znamená posun o jeden příkaz vpřed.

Ukončení vlákna

Pomocí metody `Environment.stop()` se v objektu, na němž je zavolaná, nastaví příznak, že provádění algoritmu má skončit. V metodě `Command.execute()` se pomocí metody `Environment.isStopped()` testuje, zda je tento příznak nastaven. Proto, jakmile algoritmus zastavíme pomocí `Environment.stop()`, zanedlouho se vlákno prováděného algoritmu ukončí. Může však dojít k následující situaci: V použitém modulu je definován nějaký podprogram, který je implementován chybně – cyklí se nebo jeho provádění trvá neúměrně dlouho. V takovém případě se může stát, že se začne provádět metoda odpovídající tomuto podprogramu, mezitím zavoláme `Environment.stop()`, ale protože metoda v modulu netestuje, zda se má algoritmus ukončit, vlákno stále běží. K ničemu takovému nedojde, používáme-li „slušně vychované“ moduly, tedy moduly, jejichž podprogramy se provádějí jen chvíli. I kdyby náhodou někdo zavolal takový chybně implementovaný podprogram, stále se s tím umí AAnim vypořádat. Pokud totiž v jeho hlavním okně stiskneme tlačítko pro ukončení běhu algoritmu, zavolá se `Environment.stop()` na `Environment` právě běžícího algoritmu a pak se na vlákno algoritmu čeká pomocí metody `Thread.join(int)`. Timeout se nastaví na několik málo sekund. Jakmile se volání této metody vrátí, otestuje se, jestli vlákno stále běží (`Thread.isAlive()`). Pokud ano, znamená to, že některý podprogram se vykonává příliš dlouho. V takovém případě se vypíše informace pro uživatele, že použitý modul je chybně implementovaný a vlákno je ukončeno násilím metodou `Thread.stop()`. Jedná se o metodu, která je označená jako deprecated. Toto použití je však ospravedlnitelné, jelikož jiná možnost, jak ukončit vlákno, neexistuje. Aby překladač nehlásil varování, je metoda `Thread.stop()` volána krkolomně přes Reflection API.

Kapitola 5. Editor kódu

Proč speciální editor?

Editor kódu je vizuální prvek, který se za normálních okolností nachází v levé části hlavního okna AAnimu. Není to obyčejné textové pole; umí zvýrazňovat řádek, na němž začíná právě vykonávaný příkaz, umí zvýraznit řádek s chybou, automaticky číslovat řádky, obsahuje pruh pro nastavování breakpointů a umožňuje jednoduché automatické odsazování.

Editor kódu je napsán speciálně pro potřeby AAnimu; jako editor kódu není použita žádná komponenta z knihovny Javy ani žádná externí knihovna. Jedná se o výjimku potvrzující pravidlo – ne vždy se vyplatí používat knihovny Javy. Java sice obsahuje velké množství komponent, které by potenciálně mohly posloužit jako editor kódu (`JEditorPane`, `JTextPane`), ale pokud by tyto komponenty měly sloužit k tak speciálním účelům, jak je v AAnimu požadováno, znamenalo by to přijmout celou filosofii autorů těchto komponent a ovládnout složité rozhraní. Pokud bychom chtěli svou komponentu rozšířit o novou funkci nebo její vzhled přizpůsobit k obrazu svému, většinu práce bychom strávili luštěním dokumentace a nemohli se soustředit na opravdovou a užitečnou práci. Pokud by se nepovedlo přijmout filosofii autorů knihovních komponent, editor by možná fungoval ale nemuselo by být dosaženo valné čistoty kódu.

Naproti tomu vytvořit komponentu „od píky“ je nesporně pracnější, avšak je-li jednou hotová, rozšiřuje se velmi snadno. Přidat podporu zvýrazňování řádků a podporu breakpointů se ukázalo jako nesmírně triviální.

Implementace editoru kódu

Komponenta editoru kódu je reprezentována třídou `CodeEdit` z balíčku aanim.codeedit. Je to přímý potomek třídy `JComponent`. `JComponent` je obecná vizuální komponenta Javy. Předefinováním její metody `paint(Graphics)` lze dosáhnout požadovaného vzhledu, implementací metod pro obsluhu událostí lze pak dosáhnou patřičného chování.

Nejdůležitější členská proměnná editoru kódu se jmenuje `lines` a uchovává aktuální text na všech řádkách editoru. Navíc si pro každou řádku pamatuje, je-li pro ni nastaven breakpoint, a pokud ano, jaká je podmínka jeho provedení. Dále si editor pamatuje číslo řádku a pozici na řádku, kde se nachází kurzor. Pamatuje si, která část textu byla uživatelem vybraná (označená myší nebo šipkami se stisknutou klávesou Shift), jaké je číslo řádku, kde se nachází právě prováděný příkaz, jaké je číslo řádku, který má být označen jako chybný, a jaký je momentální stav kurzoru (je zobrazen, není zobrazen – automaticky měněno časovačem).

Je ještě třeba zmínit, že AAnim (a týká se to především editoru kódu) pracuje ve dvou režimech. V režimu editace lze editovat kód pseudokódu, v tomto režimu se editor nachází, jestliže algoritmus zrovna neběží. Pokud algoritmus běží, nelze editovat jeho kód, jelikož jsou průběžně vyznačovány právě prováděné řádky a editace by způsobila zmatek. Když algoritmus běží, říkáme, že se editor nachází v režimu předvádění. Režim lze nastavit voláním metody

`setEditable(boolean)`. Třída `CodeEdit` odchytává stisky kláves a reaguje na ně. Například je-li stisknuta klávesa s nějakým tisknutelným znakem, vloží se tento znak za aktuální pozici v textu (zavoláním vhodné metody objektu `lines`). Podobně `CodeEdit` reaguje na ostatní události klávesnice, tak, aby to odpovídalo tomu, co od textového editoru očekáváme.

Metoda `paint(Graphics)` vykresluje na základě aktuálních hodnot členských proměnných text. Nekreslí však úplně všechny řádky. Nejprve si zjistí (voláním `getVisibleRect()`), která část textového editoru je viditelná. (Díky tomu, že textový editor lze scrollovat, nemusí být viditelné všechny řádky.) Viditelnou část textu pak vypíše. Jinou barvou (konkrétně šedou) vykreslí pozadí vybraného textu, nakonec nakreslí tenký proužek kurzoru (pokud má být zobrazen). Do levého oranžového pruhu se kreslí ikonky breakpointů. Je rozlišováno několik stavů breakpointu. Pokud breakpoint není vůbec nastaven, není pro něj zobrazena žádná ikonka. Nastavíme-li breakpoint a editor se zrovna nachází v režimu editace, breakpoint se dostane do stavu *potenciální*, což znamená, že pokud projde kontrolou, může se stát aktivním. Potenciální breakpointy jsou zobrazovány jako tmavě červené puntíky. Kontrola proběhne po úspěšné kompilaci. Kontrolu provádí metoda `CodeEdit.checkBreakpoints(HashMap, Module, HashSet)`.

Pokud se editor nachází v době nastavení breakpointu v režimu předvádění, proběhne kontrola bezprostředně po nastavení breakpointu. Kontrola breakpoint označí buď jako platný (na řádku, kde je nastaven, začíná příkaz), nebo oznámí, že na řádce, kde je nastaven, nezačíná žádný příkaz, případně že v podmínce breakpointu je chyba.

Třída `CodeEdit` umožňuje práci se schránkou. Má implementovány metody `copy()`, `cut()` a `paste()` s obvyklou sémantikou. Schránka není lokální, data lze přenášet i z nebo do jiných aplikací.

Editor umí provádět akce undo a redo. Důraz byl kladen na jednoduchost, a proto je jejich implementace ta nejjednodušší, jaká může být. Používá k tomu takzvaný *undo kontext*, což je třída, která uchovává text v editoru, číslo aktuálního řádku a pozici na řádku. Kontext se ukládá na dva zásobníky – undo zásobník a redo zásobník. Před jakoukoliv změnou textu v editoru se nejprve aktuální stav uloží na undo zásobník jako řetězec. Při provádění akce undo se nejprve na vrchol redo zásobníku uloží aktuální obsah editoru a v editoru se nastaví text nacházející se na vrcholu undo zásobníku (a z undo zásobníku se vrchol vyjme). Akce redo probíhá opačně: na undo zásobník se uloží aktuální obsah editoru a obsah editoru se nastaví podle vrcholu redo zásobníku (který se pak smaže).

Pokud chce být některý z objektů informován o změnách undo a redo zásobníku, stačí, když bude implementovat rozhraní `UndoStackListener` a zaregistruje se v editoru kódu pomocí metody `CodeEdit.addUndoStackListener(UndoStackListener)`. Prostřednictvím metody `undoStackChanged` se při každé práci s některým ze zásobníků oznámí všem takto registrovaným objektům, že byl přidán prvek do některého zásobníku nebo z některého zásobníku byl prvek smazán. Objekt pak může pomocí metod `CodeEdit.isUndoStackEmpty()` a `CodeEdit.isRedoStackEmpty()` testovat, jestli je některý ze zásobníků prázdný. Toho využívá hlavní okno AAnimu – když se vyprázdní undo zásobník,

disabluje se tlačítko Zpět v menu Editace, když se vyprázdní redo zásobník, disabluje se tlačítko Opakovat v menu Editace.

Možnosti, jak vylepšit editor kódu

Editor kódu byl vyvinut, aby co nejlépe sloužil potřebám AAnimu. Zpětně lze říct, že se to povedlo. Samostatně použitelný nicméně není. Přitom by se dal využít i v mnoha jiných projektech, kde je potřeba editovat zdrojové kódy. Byly by k tomu však potřeba jisté úpravy. Pokusím se sepsat stručný přehled takových úprav, o kterých by se dalo do budoucna uvažovat.

Za prvé, editor obsahuje některé grafické prvky, které nemusejí najít uplatnění v jiných aplikacích. Jedná se zejména o proužek s breakpointy a číslování řádků. Rozumné by bylo, kdyby se tyto komponenty osamostatnily a pouze se poskytla metoda `CodeEditu`, která by je informovala o počtu řádků. Umožnilo by to krom jiného samostatné scrollování textu, aniž by se při tom hýbaly breakpointy a čísla řádků.

Nyní je editor závislý na některých třídách AAnimu, například na třídách kompilátoru, které zkompilují výraz určující podmínku breakpointu. Stačilo by však definovat nějaký interface, který by předepisoval metody provádějící vyhodnocení výrazu a toto rozhraní by mimo jiné implementovaly i třídy AAnimu, které výraz vyhodnocují. Mohly by jej však implementovat i jiné třídy v projektech nesouvisejících s AAnimem. Samozřejmě je omezující, že lze označit pouze řádek s právě vykonávaným příkazem (oranžově) a řádek s chybou (červeně). Pro obecné použití by se muselo definovat označení libovolného řádku libovolnou barvou.

Další vylepšení se týká nikoliv editoru jako takového, ale způsobu jeho použití. V některých aplikacích, AAnim nevyjímaje, trvá kompilace zdrojového kódu zlomek sekundy (u pseudokódu je to doba prakticky nepostřehnutelná). Proto by se možná vyplatilo provádět kompilaci průběžně a uživatele o chybách informovat na stavovém řádku. Tím pádem by se mohla provádět kontrola platnosti breakpointů průběžně a nebyly by třeba potenciální breakpointy.

Jednoduchá implementace akcí undo a redo zatím nevadí. Text v pseudokódu bývá typicky krátký a zdržení při ukládání na zásobník není znatelné. Pro náročnější aplikace by se však musela implementace vylepšit – například na zásobník ukládat pouze řádky, v nichž došlo ke změně. Dále by bylo vhodné, kdyby se na zásobník neukládal obsah editoru před každou změnou, ale pouze při významnějších změnách (např. smazání jednoho slova, napsání slova, vymazání bloku), tak, jak je to u moderních editorů zvykem.

Kapitola 6. Stručně o grafickém uživatelském rozhraní

Balíček aanim.gui

Nejdůležitější třídy grafického uživatelského rozhraní (dále jen *„GUI“*) jsou umístěny v balíčku aanim.gui. V tomto balíčku najdeme i hlavní okno AAnimu – je reprezentováno třídou `AAnimGUI`. Je to v podstatě hlavní třída celého AAnimu, jelikož právě v ní je umístěna metoda main. Členskými proměnnými třídy `AAnimGUI` nejsou pouze grafické prvky, ale také několik objektů, které spravují kompilaci, interpretaci a práci se vstupními (případně i výstupními) daty algoritmů. Kompilaci zajišťuje objekt kompilátoru, který je deklarovaný takto:

```
private Translator t;
```

Je používán tehdy, je-li potřeba algoritmus v pseudokódu přeložit. Před každým spuštěním, pokud se kód algoritmu od posledního spuštění změnil, je třeba provést kompilaci. Kompilaci řídí metoda `AAnimGUI.compile()`. Nejprve vytvoří novou instanci kompilátoru a předá jí zdrojový kód algoritmu. Při spuštění algoritmu předá nově vytvořenému vláknu tělo algoritmu (`t.getBody()`).

Při interpretaci jsou využívány proměnné `env` a `th` deklarované takto:

```
private Environment env;
private Thread th;
```

Instance třídy `Environment` je vytvořena při spuštění algoritmu a zaniká

1. vytvořením nového algoritmu (v menu: Soubor → Nový)
2. otevřením nového algoritmu (Soubor → Otevřít)
3. zkompilováním aktuálního algoritmu (Algoritmus → Zkompilovat)

Nové vlákno `th` vznikne při spuštění algoritmu a zanikne buď tím, že algoritmus sám skončí, nebo tehdy, je-li zastaven uživatelem. Vstupní a výstupní data algoritmu řídí objekt `ak` deklarovaný takto:

```
private AlgorithmKit ak;
```

Podrobnosti o balíčku aanim.algorithmkit jsou uvedeny ve vygenerované dokumentaci a v práci kolegy Štěpána.

Dále třída `AAnimGUI` zapouzdřuje spoustu ovládacích prvků – tlačítka, menu, combobox, slider, a tak dále. Pochopit jejich funkci není těžké, stačí nahlédnou do vygenerované programátorské dokumentace. U tlačítek a položek menu je pouze vhodné zmínit, že AAnim se může nacházet v různých stavech (před kompilací, mezi kompilací a spuštěním, režim předvádění, po ukončení algoritmu ale před kompilací nebo otevřením jiného) těmto stavům také odpovídá to, která tlačítka jsou enablovaná a která disablovaná. O logiku ohledně enablování a disablování tlačítek se starají metody, jejichž název začíná na „`enableAndDisableButtons`“. Jsou blíže popsány ve vygenerované dokumentaci.

Hlavní okno je dále rodičovským oknem pro panel animátoru. Objekt animátoru (členská proměnná `animator`) se vytvoří, jakmile je známo, jaký konfigurační soubor se použije, což je buď při otevření souboru s algoritmem (implicitní konfigurace) nebo při změně výběru v comboboxu v ovládacím panelu (podrobnější informace o konfiguračních souborech animátoru najdete v uživatelské dokumentaci). Objekt animátoru se vytváří pomocí Reflection API. Jméno animátoru se získá z comboboxu v ovládacím panelu a podle pojmenovávacích konvencí (popsány v uživatelské dokumentaci) se vytvoří nová instance třídy `Animator_general`. Panel animátoru se umístí do pravé části hlavního okna. Při spuštění algoritmu se nejprve animátoru předá objekt třídy `Environment` – právě vytvořené prostředí. Pak se volá metoda `Animator_general.run()`, ta říká animátoru, že se má inicializovat. Pokud se změnily hodnoty proměnných, zavolá se metoda `Animator_general.refresh()`, která animátoru řekne, že se má aktualizovat. Po skončení algoritmu se zavolá metoda `Animator_general.stop()`, je-li to třeba, animátor provede úklid.

Další GUI komponentou je okno s nápovědou. Jedná se jednoduchý prohlížeč HTML souborů nápovědy AAnimu. Historie stránek (funkce Vpřed a Zpět) je implementována pomocí dvou zásobníků, kam se ukládají URL navštívených stránek.

Potíže při konstrukci GUI

Jednou z nejtěžších věcí při konstrukci GUI bylo rozvrhnou vizuální komponenty tak, aby bylo rozložení přehledné, uživateli přineslo co nejvíce pohodlí a fungovalo na co nejširším okruhu hardware.

Tabulka proměnných

Ve starších verzích byla umístěna pod panelem animátoru. Protože jsme však usoudili, že tabulku proměnných bude využívat jen málo uživatelů a nejdůležitější je přeci jen animace, rozhodli jsme se umístit tabulku do zvláštního okna (objektu třídy `JInternalFrame`), které se zobrazí jen na žádost uživatele.

Rychlost animace

Původně se rychlost animace nastavovala tak, že se vybrala jistá položka menu, ta zobrazila dialog a v něm se zadala prodleva mezi kroky animace. To však znemožňovalo rozumně nastavovat rychlost animace za běhu, neboť výběr položky menu by uživatele zdržoval. Proto se vytvořil slider, který se umístil pod animátor. Když si však uživatel nechal zobrazit pouze zdrojový kód a animátor ho nezajímal, neměl jak nastavit rychlost animace. Animátor roztažený dole po celé šířce okna by zase zabíral neúměrně mnoho místa. Proto se nakonec ocitl v ovládacím panelu.

Málo místa na displayi

Jedná se o věčný problém. Spousta animací by k názornému zobrazení průběhu výpočtu potřebovala více prostoru na displayi. Jedno z řešení bylo nechat uživatele nastavit velikost písma v editoru kódu. Zvolí-li uživatel menší písmo, zúží se editor

kódu a zbude víc místa pro animátor. U animátoru přirozeného slučování to pomohlo. Dalším řešením je umístění animátoru do scrollpanu a ty části animátoru, kde se něco zajímavého děje, ukazovat pomocí automatického scrollování (metoda `Component.scrollRectToVisible()`). To pomohlo například u animátoru B-stromu.

Co zbývá vyřešit

Potíží je, že okna rendererů se při prvním otevření objevují stále v levém horním rohu. Bylo by dobré se do budoucna zamyslet nad nějakým inteligentním automatickým umístěním těchto oken, ale tak, aby to uživateli nebylo na obtíž.

Kapitola 7. Pozorovatelé hodnot proměnných

Třída `Environment` udržuje aktuální hodnoty proměnných. Pomocí metod

```
Environment.addVarChangeListener(VarChangeListener)
```

a

```
Environment.addVarsChangeListener(VarsChangedListener)
```

se u ní mohou zaregistrovat objekty, které chtějí být informovány o změně hodnot proměnných. Těmto objektům říkejme *pozorovatelé*. Pozorovatelé mohou získat aktuální hodnoty proměnných pomocí metody `Environment.getVarValue(String)`, jejímž parametrem je název proměnné. Pozorovateli jsou: tabulka proměnných, renderery hodnot proměnných, animátor a také editor hodnot proměnných. Tabulka proměnných a editor mohou, kromě zjišťování hodnot proměnných, tyto hodnoty i měnit.

Podívejme se na to, kdo spravuje renderery a editory. V balíčku aanim.vartable najdeme dvě třídy. Tabulku proměnných (`VarTable`) a její model (`VarTableModel`). Tabulka proměnných je grafický prvek, pouze zobrazuje data, která jí poskytuje model. Model tabulky si udržuje přehled o všech proměnných, s nimiž algoritmus pracuje. Navíc slouží jako správce rendererů a editorů. Pokud chce uživatel zobrazit nějaký renderer, tabulka zavolá metodu `VarTableModel.renderRow(int)` a jako parametr jí předá číslo řádku tabulky, kde je proměnná, pro níž se má zobrazit renderer. Model tabulky zjistí, jestli byl renderer již někdy zobrazen. Pokud ne, vytvoří novou instanci rendereru, zapamatuje si ji a zobrazí. Pokud už renderer vytvořen byl (ale uživatel jej zavřel), pouze jej opět zobrazí. Obdobně model tabulky zachází s editory. Hodnoty proměnných se nastavují metodou `Environment.setVarValue(String, PCType_general)`, které se předá název a nová hodnota.

Zanikne-li environment, zanikají i všechny renderery a editory (zavřou se jejich okna). Postará se o to metoda `VarTableModel.discardRenderers()`, která je volaná ze třídy AAnimGUI ve stejnou dobu, kde zaniká environment. Podrobnější informace o rendererech a editorech jako takových najdete v práci kolegy Štěpána, autora balíčku aanim.datatypes.

Kapitola 8. Stručně o dalších komponentách AAnimu

Správce balíčků

Z uživatelského hlediska jsou balíčky popsány v uživatelské dokumentaci, najdete tam i návod, jak balíček vytvořit. Třídy pro správu balíčků (AAnimu) jsou umístěny v balíčku (myšleno Javovském) aanim.packageman. Pro podrobnější informace nahlédněte do vygenerované programátorské dokumentace. Do budoucna bude potřeba se hlouběji zamyslet nad těmito aspekty balíčků:

- Současný stav: Pokud balíček *A* závisí na balíčku `B` verze *n*, závislost bude splněna i tehdy, pokud bude nainstalován balíček `B` verze `m`, kde `m` > `n`. Jak ale zabránit autorovi balíčku, aby z verze *m* odstranil některé věci, které byly ve verzi *n* a balíček `A` na ně spoléhá? Zatím autorovi balíčku nic nebrání, aby kromě přidávání funkcionalit do nové verze také ubral něco, co v předchozích verzích bylo. Pokud se na to spoléhají jiné balíčky, mohlo by dojít k nekonzistenci.
- Povolit či nepovolit rekurzivní odstraňování závislých balíčků? Nyní, pokud uživatel odstraňuje balíček, na němž závisí jiné balíčky, dojde k chybě a uživatel bude vyzván, aby odstranil závislé balíčky ručně. Možná by se hodilo, kdyby se toto provádělo automaticky. Avšak takový „velký úklid“ uživatel asi nebude provádět často a rekurzivní odstraňování je potenciálně nebezpečné.

Smartfile

Jedná se o třídu, která se stará o inteligentní práci s otevřeným souborem. Podrobnosti hledejte ve vygenerované programátorské dokumentaci.

Co se jinam nevešlo

V balíčku aanim.utils se nachází několik tříd se statickými členskými proměnnými a metodami, které se používají v celém AAnimu a logicky nepatří do jiného balíčku. Podrobnosti hledejte ve vygenerované dokumentaci. Za zmínku stojí třída `fileUtils`, která se stará o systémově nezávislé čtení souborů. Soubory, jako jsou zdrojové kódy, konfigurace animací, vstupní sady, a tak dále, se ukládají v kódování `iso-8859-2` a toto kódování je dodržováno bez ohledu na to, v jakém operačním systému AAnim zrovna běží.

Kapitola 9. Některá poučení plynoucí z vývoje AAnimu

Poučení 1: Reflection API je docela užitečné

V AAnimu je využíváno na mnoha místech – při implementaci nových datových typů, při implementaci modulů i animátorů. Jeho použití je relativně snadné, ale má i svá úskalí. Problém činí zejména při tzv. refaktorování, tedy přejmenovávání proměnných, tříd či metod. Zatímco klíčová slova použitá normálním způsobem vývojové prostředí bez problému rozpozná a nahradí, jsou-li použita ve volání metod Reflection API, nahradit je nemůže. To se musí provádět ručně. Konkrétní příklad: V projektu jsme používali v názvech balíčků i velká písmena (dle konvencí mají mít balíčky názvy složené jen z malých písmen). Proto jsme se rozhodli balíčky přejmenovat. Vývojové prostředí prošlo všechna použití názvů balíčků a bezpečně je nahradilo variantou s malými písmeny. Až na případ, kdy byl název balíčku (nebo jeho část) použit ve volání metody Reflection API jako parametr typu String. Tam se přejmenování muselo provést ručně. Aby se řetězce s názvy balíčků neopakovaly v kódu mnohokrát, jsou definovány jako konstanty ve třídě aanim.utils.Names.

Poučení 2: Dovolit komukoliv rozšiřovat program je problematické

Moduly, datové typy a animátory může psát kdokoliv, stejně tak může kdokoliv vytvářet balíčky. Je značně problematické zabránit tomu, aby chyba autora modulu, animátoru či balíčku ohrozila celý AAnim. Jednak se špatně ukončuje vlákno, v němž se zacyklila nějaká procedura modulu, pak zůstává nedořešená otázka ohledně verzí balíčků a závislostí diskutovaná výše.

Poučení 3: V Javě lze vyvíjet prakticky kdekoliv

Za rok, co pracujeme na tomto projektu, jsme vystřídali na různé platformy a v různá vývojová prostředí. Operační systémy: Windows 98, Windows XP, Gentoo Linux, Mandriva Linux, Fedora Core Linux.Vývojová prostředí: Unix shell & javac & vim, Příkazový řádek & javac & gvim, NetBeans, Eclipse, JBuilder a další. Přechod mezi jednotlivými platformami či prostředími neprovázely větší problémy.

Kapitola 10. Co přinesl projekt AAnim?

Projekt AAnim je připraven k praktickému použití ve výuce. Výběr algoritmů a animací byl připraven za velmi vydatné spolupráce s RNDr. Rudolfem Krylem, který měl jak k animacím, tak k samotnému programu AAnim, spousty užitečných připomínek, které jsou do projektu zapracovány.

AAnim byl již od samého začátku vyvíjen tak, aby bylo velmi snadné jej rozšířit a obohatit o nové dovednosti. Jak ukazuje naše osobní zkušenost, přidání průměrně složitého algoritmu včetně vytvoření modulu a animátoru nezabere více než den práce pro jednoho programátora. Dá se očekávat, že s přibývajícím množstvím modulů bude odpadat nutnost psát moduly nové a budou se používat ty již existující. Přidání jiné varianty téhož algoritmu je velmi snadné, stačí pouze pozměnit zdrojový kód algoritmu v pseudokódu. Pro jednu třídu algoritmu může sloužit jediný animátor (krásně to ilustrují algoritmy vnitřního třídění).

Pro vytváření nových modulů a animátorů není potřeba kompletní zdrojový kód AAnimu a uživatel, který si bude chtít modul či animátor nainstalovat, nemusí nic kompilovat – nepotřebuje ani kompilátor Javy. Stačí, když si stáhne z Internetu balíček AAnimu (*.apk), což je jeden soubor, a nainstaluje jej několika kliknutími myši.

Možná rozšíření a vylepšení AAnimu

Do budoucna by se dal AAnim ještě více vylepšovat. V úvahu přichází například:

Uživatelské procedury a funkce

Jazyk AAnimu používá pouze funkce a procedury definované v modulu, nemůže si psát vlastní. Rozšířit AAnim tak, aby to uměl, je poněkud pracné, ale v zásadě tomu nic nebrání. Spousta algoritmů by se tak zpřehlednila a šly by animovat i rekurzivní algoritmy.

Importování více modulů v jednom algoritmu

V současnosti lze v algoritmu specifikovat pouze jeden modul, který se bude používat. Jistě by bylo možné používat v jednom algoritmu funkce, procedury a typy z více modulů současně. Bude však potřeba vyřešit některé aspekty, například: co když bude ve více modulech funkce se stejným názvem a stejnými typy parametrů?

Generování dokumentace k modulům a animátorům

Pro autory modulů a animátorů by bylo užitečné, kdyby existoval nástroj, který by generoval dokumentaci k modulům a případně i animátorům na základě Javovských zdrojových kódů tříd modulů a animátorů. Programátorovi by to šetřilo čas a bylo by zajištěno, že na nic důležitého nezapomněl.

Příloha A. Zodpovědnost za jednotlivé části AAnimu

V této příloze je shrnuto, kdo z autorů je zodpovědný za které části projektu AAnim. Najdete tu informace o autorství jednotlivých komponent AAnimu, modulů a animátorů. Na některých komponentách pracovali oba studenti společně, v takovém případě bude tato informace podrobněji rozvedena.

Základní komponenty AAnimu

Komponenty vytvořené Pavlem Římským

- editor kódu (celý balíček aanim.codeedit)
- třídy reprezentující prostředí, v němž algoritmus běží (balíček aanim.environment)
- část grafického uživatelského rozhraní
  - téměř celý balíček aanim.gui (Petr Štěpán v něm prováděl změny pouze tehdy, pokud to bylo nutné, aby se projevily některé úpravy, které prováděl ve svých komponentách)
  - tabulka proměnných (balíček aanim.vartable), drobné úpravy v rodičovských třídách editorů a rendererů
  - balíček aanim.utils, až na dvě metody ve třídě `CGU`
- správa modulů (celý balíček aanim.moduleman)
- správa balíčků AAnim (celý balíček aanim.packageman)
- třída `aanim.smartfile.Smartfile`
- sémantická analýza a interpretace derivačního stromu (třídy `Command` a `Expression`, Petr Štěpán v nich podnikl drobné úpravy)

Komponenty vytvořené Petrem Štěpánem

- překladač -- lexikální a syntaktická analýza (většina balíčku aanim.translator)
- jednotná správa dat algoritmu (balíček aanim.algorithmkit)
- mechanismus rolí v animátorech (`aanim.animators.Role`, související části v `aanim.animators.general.General`)
- správce konfigurací animátoru (balíček aanim.animators.configedit)
- správce sad proměnných (balíček aanim.setman)
- mechanismus zobrazovačů a editorů typů (balíček aanim.datatypes)

Moduly

Moduly vytvořené Pavlem Římským

- extsort

- general (spolu s Petrem Štěpánem)
- hashtable
- polyphase
- radixsort
- sachovnice

Moduly vytvořené Petrem Štěpánem

- btree
- general (spolu s Pavlem Římským)
- hashing
- heap

Animátory

Animátory vytvořené Pavlem Římským

- colarray (spolu s Petrem Štěpánem)
- hamming
- hashtable
- history
- polyphase
- radixsort
- rainbow
- tapes

Animátory vytvořené Petrem Štěpánem

- btree
- chainhash
- colarray (spolu s Pavlem Římským)
- heap
- heaparray
- matrixmultiply
- merge