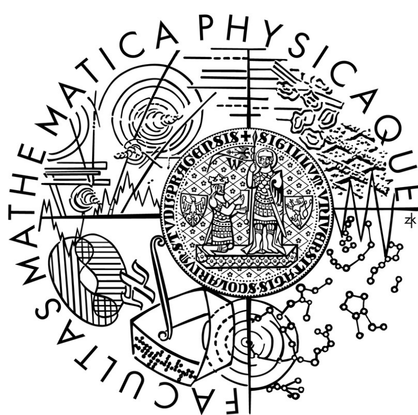
Univerzita Karlova v Prahe
Matematicko-fyzikálna fakulta

BAKALÁRSKA PRÁCA



Maroš Vranec

Vizualizace algoritmů

Kabinet software a výuky informatiky
Vedúci bakalárskej práce: Mgr. Martin Senft
Študijný program: Informatika

2006

Poďakovanie

Ďakujem môjmu vedúcemu, Mgr. Martinovi Senftovi, za pomoc hlavne pri návrhu a za cenné pripomienky počas celého vývoja.

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím uvedenej literatúry. Súhlasím so zapožičaním práce a s jej zverejnením.

V Prahe dňa 7.8.2006 Maroš Vranec

Obsah

OBSAH...........3

KAPITOLA 1 – ÚVOD...........6

1.1 Motivácia...........6
1.2 Cieľ...........6
1.3 Alternatívne projekty...........6
1.3.1 JDSL a net.datastructures...........6
1.3.2 JDSL Visualizer...........6
1.3.3 Algovision...........7
1.3.4 CCAA a ďalšie...........7
1.3.5 Diplomová práca Ondřeja Hanouska...........7

KAPITOLA 2 – UŽÍVATEĽSKÁ PRÍRUČKA...........8

2.1 Dátové štruktúry...........8
2.1.1 Strom...........8
2.1.2 Graf...........9
2.2 Animované dátové štruktúry...........9
2.2.1 Animované primitívne typy...........9
2.2.2 Animované pole a animovaný spojový zoznam...........10
2.2.3 Animovaný strom...........10
2.2.4 Animovaný graf...........10
2.3 Obaly...........10
2.3.1 Bežné obaly...........10
2.3.2 Obal pre číslo a reťazec...........11
2.4 Ostatné možnosti knižnice...........11
2.4.1 Krokovanie...........11
2.4.2 Podpora pre skiny...........12
2.4.3 Pseudokód...........12
2.5 Použitie knižnice - prakticky...........12
2.6 Radix sort...........18
2.6.1 Inicializácia poľa na zotriedenie...........18
2.6.2 Inicializácia 10 spojových zoznamov...........19
2.6.3 Triedenie...........19
2.6.4 Ďalšie použité vlastnosti knižnice...........19
2.7 Alfa beta...........20
2.7.1 Tvorba náhodného stromu...........20
2.7.2 Alfa beta...........20
2.8 Kruskalov algoritmus...........20

KAPITOLA 3 – ANALÝZA A NÁVRH...........21

3.1 Správa požiadaviek...........21
3.1.1 Počiatočné požiadavky...........21
3.1.2 Zmena ukážkových aplikácií knižnice...........21
3.1.3 Použitie knižnice cez obaly...........21

3.1.4 Krokovanie algoritmov a ďalšie....22
3.2 Návrh knižnice....22
3.2.1 Pôvodný návrh knižnice....22
3.2.2 Zlom v návrhu....22
3.2.3 Návrh pred obalmi....23
3.2.4 Obaly....23
3.3 Detaily návrhu....23
3.3.1 Rozhrania....24
3.3.2 Implementácia stromu a grafu....24
3.3.3 Animované dátové štruktúry....24
3.3.4 Obaly....25
3.3.5 Ostatné možnosti knižnice....25

KAPITOLA 4 – IMPLEMENTÁCIA....26

4.1 Použité technológie....26
4.1.1 Vývojové nástroje....26
4.1.2 Programovací jazyk....26
4.2 Zvládnuté technické prekážky....26
4.2.1 Grafický výstup v Jave....26
4.2.2 Layouty vo Swingu....26
4.2.3 Práca s viacerými vláknami....27
4.2.4 Iniciácia appletov....27
4.2.5 Reflexia....27
4.3 Testovanie....28
4.3.1 Test-driven development....28
4.3.2 Testy v knižnici animelib....28
4.4 Dokumentácia....29
4.4.1 JavaDoc....29

KAPITOLA 5 – ZÁVER....30

5.1 Porovnanie s alternatívnymi projektmi....30
5.2 Prínos knižnice....30
5.3 Splnenie cieľa....30
5.4 Možnosti ďalšieho vývoja....31

LITERATÚRA....32

PRÍLOHY....33

Animované objekty....33
Obaly....34
Obsah CD....34

Názov práce: Vizualizácia algoritmov
Autor: Maroš Vranec
Katedra (ústav): Kabinet software a výuky informatiky
Vedúci bakalárskej práce: Mgr. Martin Senft
e-mail vedúceho: Martin.Senft@mff.cuni.cz

Abstrakt: Knižnica animelib je určená pre užívateľsky prívetivú vizualizáciu algoritmov. Dôraz je kladený na to, aby programátor nemusel písať veľa kódu, ktorý sa týka vizualizácie, a tiež nemusel príliš prispôsobovať jeho vlastný kód (teda kód algoritmu, ktorý chce vizualizovať). Zaoberá sa vizualizáciou základných dátových štruktúr (zoznam, spojový zoznam, strom a graf). Práca s knižnicou je takmer transparentná, jediná potrebná vec je zaregistrovať dátové štruktúry, ktoré sa majú zobraziť. Knižnica naviac poskytuje rozšírené operácie na objektoch, ktoré dovoľujú lepšie animovať.

Kľúčové slová: Algoritmus, vizualizácia, applet, dátové štruktúry, Java.

Title: Visualization of algorithms

Author: Maroš Vranec
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Martin Senft
Supervisor's e-mail address: Martin.Senft@mff.cuni.cz

Abstract: Animelib is Java library, which visualizes data structures. It is designated for user-friendly visualization of algorithms. User-friendlyness is done in two ways – programmer of algorithm has much less graphics code to write (none ideally), and user – viewer should be pleased with what he sees.

Keywords: Algorithm, visualization, applet, data structures, Java.

Kapitola 1 – Úvod

1.1 Motivácia

Na internete je vidieť mnoho appletov, ktoré istým spôsobom vizualizujú algoritmy. Tieto applety boli robené ako jednorázové ukážky algoritmu, a ich vizualizačná logika ostala použitá iba raz.

Zdá sa preto užitočné vytvoriť nástroj, ktorý dostatočne uľahčí tvorbu takýchto prezentácií algoritmov, a zároveň je dosť robustný a modifikovateľný, aby dokázal pokryť čo najviac algoritmov. Tak, aby sa programátori nemuseli zaťažovať zobrazovacou logikou (jej zvládnutie môže byť náročné), ale sústredili by sa len na samotný algoritmus.

1.2 Cieľ

Cieľom tejto bakalárskej práce je vytvoriť knižnicu, ktorá sa dá pohodlne integrovať do ľubovoľného programu (v Jave) a umožňuje programátorsky čo najjednoduchšiu vizualizáciu algoritmu. Použitie je také jednoduché, že stačia dve-tri volania knižnice, a vizualizácia je hotová. Tým je zaručená pohoda na strane užívateľa – programátora (ďalej len programátora).

Nezabúda sa ani na druhý typ užívateľa – diváka prezentácie, ktorý by mal, pokiaľ možno, pochopiť algoritmus z jeho vizualizácie. Výstup, ktorý sa dostane až k divákovi je teda čo najkvalitnejší. Je tiež maximálne modifikovateľný, ak by mal programátor špeciálne požiadavky na samotné zobrazenie.

Okrem toho knižnica ponúka ďalšie funkčnosti, ktoré pomáhajú vizualizácii, ako krokovanie v algoritme, podpora pre skinovanie a podpora pre pseudo-kód.

1.3 Alternatívne projekty

1.3.1 JDSL a net.datastructures

Na internete je každému k dispozícii veľa open-sourceových projektov zameraných na dátové štruktúry. Z najznámejších stojí za zmienku knižnica JDSL ([4]) a knižnica net.datastructures ([3]). Žiadny z týchto projektov sa však nevenuje ich vizualizácii. Rozhranie, ktoré na prácu s dátovými štruktúrami ponúkajú, je podobné tomu z knižnice animelib. Obsahujú aj podporu pre dátové štruktúry, ktoré v animelib nie sú (napríklad pre hešovacie tabuľky).

1.3.2 JDSL Visualizer

Nad knižnicou JDSL bol postavený vizualizátor dátových štruktúr, teda JDSL Visualizer ([4], http://www.cs.brown.edu/cgc/jdsl/explorations/jdslviz/jdslviz.html). Tento projekt sa snáď najviac podobá knižnici animelib, ktorá je predmetom tejto bakalárskej práce. Avšak kým JDSL Visualizer sa sústreďuje na vizualizáciu dátových štruktúr, a poskytuje tomu primerané užívateľské prostredie, animelib sa zameriava na

vizualizáciu *algoritmov*, a maximálne uľahčenie práce programátora, ktorý chce nejaký ten algoritmus prezentovať.

1.3.3 Algovision

Algovision ([1]) je projekt od pána Prof. RNDr. Luděka Kučeru, DrSc.. Zameriava sa na vizualizáciu algoritmov. Projekt je dosť veľký, obsahuje vizualizácie vyše 15 algoritmov. Algoritmy sú vizualizované pomocou appletov v Jave. Rozdiel oproti animelib, je okrem iného v obtiažnosti naprogramovania vizualizácií. Pomocou animelib je vizualizácia niektorých algoritmov otázkou pár minút. V projekte Algovision sú algoritmy pomerne rôznorodé, takže sa zrejme musela programovať vizualizačné logika jednotlivo. Nevýhoda animelib je (okrem iného) v tom, že svojimi dátovými štruktúrami nedokáže pokryť toľko algoritmov ako (prakticky neobmedzený) Algovision.

1.3.4 CCAA a ďalšie

CCAA alebo Complete Collection of Algorithm Animations ([10]) je zbierka odkazov na ďalšie stránky s mnohými animáciami algoritmov. Vo všeobecnosti sa dá na internete (aj mimo neho) nájsť animovaný takmer ľubovoľný algoritmus, ktorý by niekto mohol potrebovať prezentovať. Oproti animelib majú tú výhodu, že môžu mať lepší grafický výstup, a dokážu vizualizovať ľubovoľný algoritmus. Výhoda animelib je v tom, že programátor nie je zaťažovaný zobrazovacou logikou.

1.3.5 Diplomová práca Ondřeja Hanouska

Pán Ondřej Hanousek vytvoril Javovskú knižnicu na vizualizáciu kompresných algoritmov ([7]). Podobne ako animelib obsahuje časť vizualizačnej logiky, aby ju nemusel kompletne celú písať programátor. Keďže sa špecializuje na kompresné algoritmy, zrejme nad animelib vyniká v ich grafickej prezentácii. Animelib má zase výhodu v tom, že napríklad krokovanie tam a späť v algoritme je implementované v rámci knižnice, takže kód algoritmu je oveľa prehľadnejší.

Kapitola 2 – Užívateľská príručka

V tejto kapitole sa postupne rozoberie, aké triedy knižnica ponúka, a ukáže sa implementácia štyroch praktických príkladov použitia knižnice.

Kapitola je rozdelená na päť častí. Prvá časť sa zaoberá implementáciou dátových štruktúr, ktoré v Jave chýbajú (teda strom a graf). Druhá sa venuje triedam, ktoré majú na starosti vykresľovanie. Tretia je o obaloch, čo je spôsob, ktorým knižnica minimalizuje zmenu v programátorovom kóde. Štvrtá časť je o rôznych „vychytávkach", ktoré sú súčasťou knižnice, ako napríklad krokovanie, podpora pre skiny, pseudokód, atď. Piatu časť tvoria štyri popísané praktické ukážky použitia knižnice.

2.1 Dátové štruktúry

Problém Javy, ak sa chcete zaoberať klasickými algoritmami, sú chýbajúce triedy pre strom a graf. Animelib preto ponúka svoje vlastné rozhrania a implementácie pre prácu s nimi. Ak máte hotové nejaké vlastné implementácie, tak ich stačí adaptovať (adaptovaním je myslené použitie GoF design patternu Adapter – vytvorí sa nová trieda, ktorá implementuje nové rozhranie, na ktoré sa chce programátor adaptovať, a vnútorne budú metódy vykonané pomocou pôvodnej implementácie... viac v [8] alebo [9]) na rozhranie z animelib, a tým je integrácia knižnice do vášho kódu hotová.

2.1.1 Strom

Knižnica animelib ponúka rozhranie stromu a jednu jeho implementáciu. Implementácia stromu neobsahuje žiadne ďalšie metódy oproti rozhraniu, takže sa o nich bude písať o oboch, ako o strome.

Strom kvôli jednoduchosti nerozširuje graf. Dôvod je ten, aby programátor, ktorý rozhranie stromu implementuje, nebol nútený zbytočne implementovať aj metódy rozhrania grafu. Ak by chcel vytvoriť dátovú štruktúru, ktorá je použiteľná jak v algoritmoch pre stromy, tak v algoritmoch pre grafy, implementuje obidve rozhrania. Ale knižnica ho k tomu nenúti, a ak chce použiť strom len ako strom, tak môže implementovať len jedno rozhranie (robí to tak aj implementácia stromu, ktorú ponúka knižnica animelib).

Strom sa vytvára s koreňom (alebo prázdny, neskôr sa nastaví koreň). Ostatné prvky sa do stromu pridávajú. Robí sa to pomocou metód na pridanie detí rodičovi. Každý uzol stromu je zároveň rodičom svojich detí, a zároveň dieťaťom svojho rodiča. Výnimkou sú koreň alebo listy, ktoré nemajú rodiča alebo deti.

Okrem pridávania uzlov stromu sa môžu uzly aj odoberať. Ak odoberaný uzol nie je listom, tak sa odoberie aj celý jeho podstrom.

Ukážka interfacu stromu (nekompletné rozhranie – kompletné zdrojové kódy si možno pozrieť na priloženom CD v adresári CD/src):

```
public interface Tree < E > {
   Position < E > getRoot();
   Position < E > getParent(final Position < E > child);
```

```
    List < Position < E > > getChildren(Position < E > parent);
        // ...
    Position < E > addChild(Position < E > parent, final E child);
    int getDepth();
    boolean isLeaf(final Position < E > node);
}
```

2.1.2 Graf

V animelib je jedno rozhranie pre graf a dve jeho implementácie – jedna je orientovaný a druhá neorientovaný graf. Rozdiel je v práci s hranami. Ak pridáte hranu do neorientovaného grafu, tak je to to isté, ako by ste pridali dve hrany do grafu orientovaného. Ďalej sa preto bude o obidvoch implementáciách aj o rozhraní hovoriť ako o grafe.

Graf má ohodnotené vrcholy aj hrany. Ohodnotené môžu byť čímkoľvek.

Do grafu sa dajú pridávať a odoberať vrcholy. Tie sa potom dajú prepájať a rozpájať pomocou hrán.

Ukážka interfacu grafu (nekompletné rozhranie):

```
public interface Graph < V, E > {
    void addVertex(V newVertexElement);
    void deleteVertex(Position < V > vertex);
    List < ? extends Position < V > > getAllVertices();
    // ...
    Position < V > getStartVertex(final Position < E > edge);
    Position < V > getFinishVertex(final Position < E > edge);
    E getLength(final Position < E > edge);
    List < ? extends Position < E > > getEdges();
}
```

2.2 Animované dátové štruktúry

Programátor sa s animovanými dátovými štruktúrami nemusí vôbec stretnúť ak nechce, ale príde tak o metódy, ktoré mu môžu pomôcť vylepšiť výstup (napr. zvýraznenie dôležitej cifry čísla).

Väčšinou sú to kolekcie, ktoré rozširujú bežné dátové štruktúry, ale programátor sa u nich nevyhne zobrazovacej logike, a musí pracovať napr. s AnimatedString namiesto obyčajného Stringu (viac o Stringu v [6]). To je väčšinou nevyhovujúce, a práve z toho dôvodu vznikli obaly, o ktorých bude reč neskôr.

2.2.1 Animované primitívne typy

Za animované primitívne typy sa u knižnice animelib považuje animované číslo a animovaný reťazec. Animované číslo je len špeciálny animovaný reťazec, ktorý umožňuje niektoré operácie naviac, ako napríklad zvýraznenie cifry, a pod.

Animované primitívne typy umožňujú, okrem svojho zobrazenia v podobe obdĺžnika s textom, zvýraznenie samého seba (ktoré je voliteľne animované, ako aj od-výraznenie) a samozrejmá je možnosť meniť ich hodnoty.

2.2.2 Animované pole a animovaný spojový zoznam

Animované pole dedí od obyčajného ArrayListu ([6]), a umožňuje do seba pridať ľubovoľné ďalšie animované objekty. Zobrazenie je jednoduché, prvky sa zobrazia v rade po sebe (alternatívne pod sebou).

Animované pole ponúka tzv. automatické zvýrazňovanie položky, s ktorou sa aktuálne pracuje, čo znamená, že sa o zvýrazňovanie väčšinou nemusíte starať. Animované sú niektoré operácie na poli, napr. pridávanie novej položky poľa.

Animovaný spojový zoznam sa od animovaného poľa líši tým, že zobrazuje aj „ukazateľ" na ďalší objekt v podobe šípky.

2.2.3 Animovaný strom

Animovaný strom sa zobrazí tak, že sa hore vykreslí koreň a pod ním hierarchia jeho potomkov (sú aj ďalšie možnosti, ako uzly rozložiť – programátor má na výber). Všetky uzly, ktoré sú v rodičovskom vzťahu, budú prepojené hranou.

Na vylepšenie vizualizácie sú tu metódy, ktorými sa vizualizuje cesta stromom (pomocou „vyhľadávacej guličky"). Týmto spôsobom možno zobraziť vyhľadanie toho správneho uzla v strome, ktoré je využívané v mnohých algoritmoch.

2.2.4 Animovaný graf

Animovaný graf najprv náhodne (alebo do kruhu – voľba je na programátorovi) umiestni vrcholy, a tie potom pospája hranami. Vypíše aj ohodnotenie hrán. U orientovaného grafu je naviac zobrazený smer hrán.

2.3 Obaly

Obaly výrazne zjednodušujú prácu, ktorú musí programátor vykonať, aby sa jeho algoritmus animoval.

Knižnica animelib má obaly pre všetky dátové štruktúry, ktoré vizualizuje, ale okrem nich sa dá obaliť aj ľubovoľný objekt (potomok triedy java.lang.Object ([6])), pomocou obalu pre reťazec.

2.3.1 Bežné obaly

Bežné obaly v knižnici animelib sú všetky obaly, okrem dvoch – obalu pre číslo a obalu pre reťazec.

Každý bežný obal dedí od triedy, ktorú obaľuje, preto sa môže do kódu napísať niečo ako:

```
mojePole = new AnimatedWrapperArrayList(mojePole);
```

Ďalej sa pracuje s premennou mojePole, ale tá už bola obalená, a animelib ju teda vizualizuje. Všetko bez vedomia programátora, on len používa bežné operácie nad ArrayListom.

Väčšina parametrov, ktoré by programátor chcel dať na vedomie knižnici, sa dá predať pomocou konštruktorov, ktoré obaly ponúkajú. Napríklad súčasťou akého grafického kontajneru by mala byť vizualizovaná dátová štruktúra.

2.3.2 Obal pre číslo a reťazec

Tieto dva obaly sú trochu špeciálne, čo je dané tým, že od normálneho java.lang.Integeru ([6]) a java.lang.Stringu sa nedá dediť. To znamená, že nemožno jednoducho používať obal namiesto pôvodnej premennej.

Každý obal implementuje rozhranie Wrapper, ktoré obsahuje metódu getWrapper(), ktorá u bežných obalov vráti referenciu na this, ale u obalu pre číslo a reťazec vráti referenciu na pôvodný objekt. Väčšinou sa to používa k interným účelom, a ak chce programátor animovať číslo alebo reťazec samotný, mal by použiť AnimatedNumber alebo AnimatedString.

Obal na reťazec je naviac zaujímavý tým, že dokáže obaliť ľubovoľný objekt. Grafická reprezentácia potom bude pomocou animovaného reťazca textovej reprezentácie pôvodného objektu (cez metódu toString(), ktorú má každý objekt).

2.4 Ostatné možnosti knižnice

2.4.1 Krokovanie

Knižnica rieši aj „krokovanie“ algoritmu, čím sa myslí prechádzanie algoritmom tam a späť. Krokovanie je v knižnici riešené tak, že sa ukladá história stavov dátových štruktúr, aby sa užívateľ mohol vracať v algoritme.

Knižnica automaticky pridá do grafického kontajnera dve tlačítka – krok a krok späť, pre posun v algoritme a pomocné prvky ckeck-box a combo-box ([6]) na ďalšie nastavenia.

Programátor môže určiť krok algoritmu aj sám, a to pomocou volania metódy pause(int level) na singletone ([8], [9]) AnimeApp. Tlačítko „krok“ potom funguje tak, že len odpauzne vlákno algoritmu, aby mohlo pokračovať.

Pri kroku späť sa knižnica sama vracia v histórii stavov algoritmu. Keďže za jeden „krok“ algoritmu sa toho môže stať viac, tieto prechody už nie sú animované. Z toho dôvodu sa divákovi algoritmu zamkne check-box „animate“, pretože stráca opodstatnenie.

Keď sa divák vráti do posledného zatiaľ vykonaného kroku algoritmu, knižnica opäť umožní vláknu algoritmu, aby ďalej bežalo, a pokračovalo tak ďalej s jeho animáciou.

Ako už bolo spomenuté, je možné použiť viac úrovní krokovania. Divák má potom možnosť si combo-boxom vybrať úroveň, po ktorej sa chce pohybovať, a všetky nižšie úrovne sú vtedy ignorované. Tento prvok knižnice sa zrejme použije u zložitejších algoritmov, kedy programátor na viaceré miesta kódu vpíše niečo ako:

```
AnimeApp.getInstance().pause(3)
```

, alebo iné číslo, čím zaručí, že na danom mieste sa má algoritmus zastaviť na úrovni 3. Čím vyššia úroveň kroku, tým viac sa toho musí za daný krok vykonať. Na najnižšej úrovni (0) krokuje sama knižnica, pri každej zmene dátových štruktúr. Teda programátor sa môže vkladaniu takýchto inštrukcií vyhnúť, ak je algoritmus dostatočne jednoduchý (všetko krokovanie nechá na knižnicu).

2.4.2 Podpora pre skiny

Grafický výstup je v knižnici modifikovateľný pomocou „skinov“. Programátor si môže doplniť vlastné skiny, stačí mu implementovať rozhranie Skin z knižnice. V knižnici má predpripravené 3 skiny.

Pomocou skinov je možné nastaviť všetky druhy farieb, ktoré knižnica použije na výstup, použité fonty písma, a štýl okraja jednotlivých grafických prvkov.

Skin sa mení zavolaním metódy na singletone AnimeApp:

```
AnimeApp.getInstance().setSkin(new SeaSkin());
```

2.4.3 Pseudokód

U vizualizácii algoritmov sa často využíva vypísanie pseudo-kódu s tým, že sa v pseudo-kóde stále vyznačí, v akom štádiu sa algoritmus práve nachádza. Animelib podporuje takýto pseudo-kód, programátor do kódu na miesto, ktoré sa pseudo-kódu týka, vpíše niečo ako:

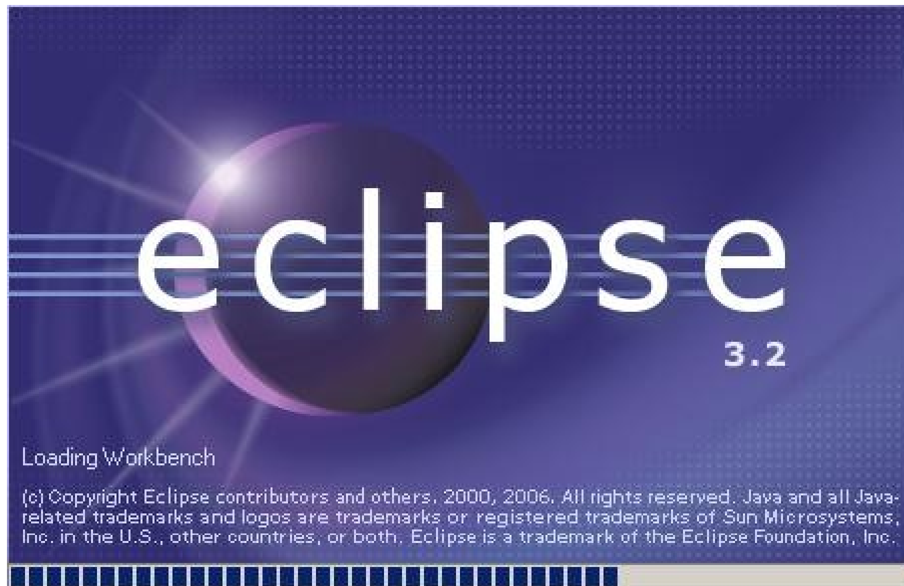
```
AnimeApp.getInstance().pseudoCode("Initializing...");
```

Knižnica sa automaticky postará o aktualizáciu okna s pseudo-kódom (podľa porovnania zhody reťazcov pseudo-kódu). Podľa týchto reťazcov pseudo-kódu bude teda automaticky vyznačené, ktorý z nich je práve aktuálny.

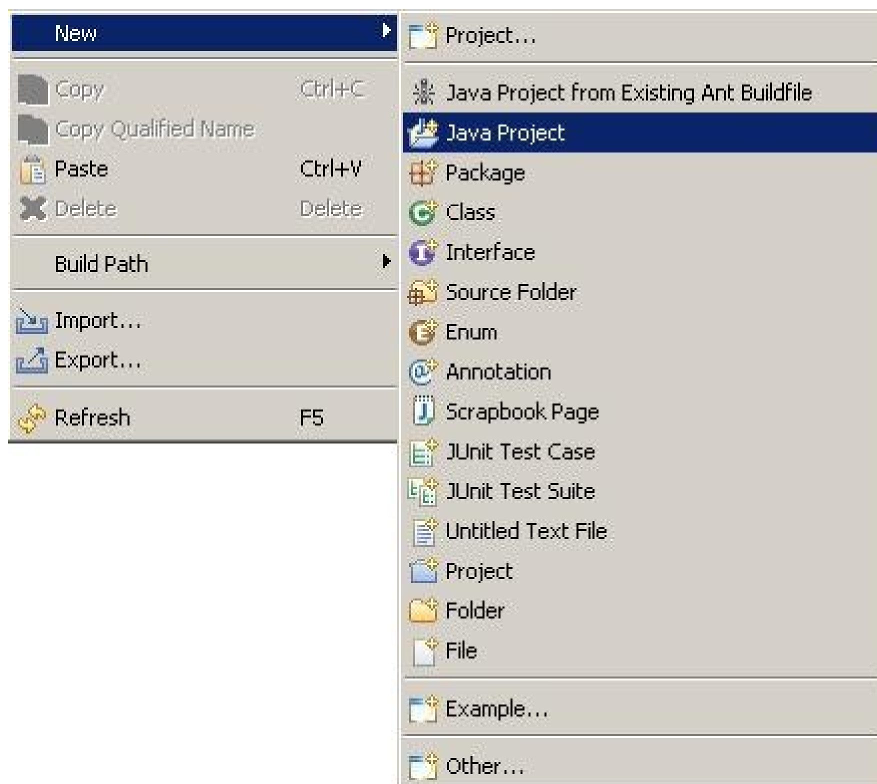
2.5 Použitie knižnice - prakticky

Predstavme si programátora, ktorý chce pomocou animelib vizualizovať quicksort (viac o algoritme v [12]). Keďže je lenivý, prvé, čo spraví, je, že si zapne internetový vyhľadávač a stiahne si nejakú implementáciu.

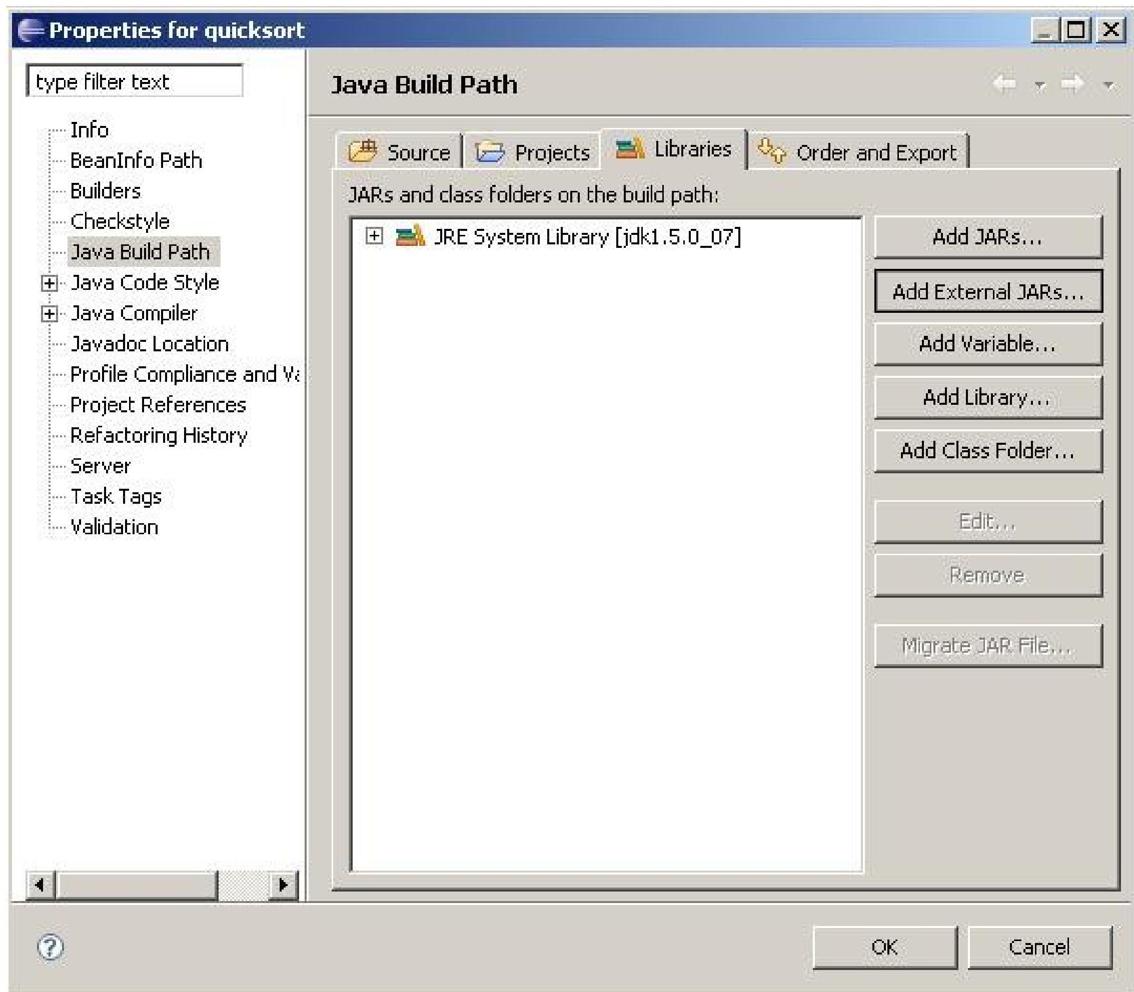
Animelib už má stiahnutú, takže si zapne svoj nástroj na programovanie Javy.



Vytvorí si nový Javovský projekt.



Prekliká sa wizardom pre nový projekt. Pre použitie knižnice animelib je nutné ju pridať do classpath. V prostredí Eclipse ([11]) sa to robí nastavením tzv. Build Path:



Po pridaní animelib.jar do build path potrebuje programátor vytvoriť triedu, ktorá bude predstavovať applet. Pre jednoduché applety môže dediť od AnimelibApplet z knižnice animelib.

Má teda predgenerovanú metódu run() od AnimelibAppletu, do ktorej vpíše nájdený algoritmus quicksort.

• • •

```
public class QuicksortApplet extends AnimelibApplet {

   @Override
   protected void run() {
      ArrayList elements;

      elements = new ArrayList();
      for (int i = 0; i < 10; ++i) {
         elements.add(new Integer((int) (Math.random() * 100)));
      }

       quickSort(elements);
   }

   public void quickSort(ArrayList elements) {
      if (!elements.isEmpty()) {
         this.quickSort(elements, 0, elements.size() - 1);
      }
   }

   private void quickSort(ArrayList elements, int lowIndex, int highIndex) {
      int lowToHighIndex;
      int highToLowIndex;
```

• • •

Doteraz nenapísal ani riadok týkajúci sa animácie (snáď až na dedenie od AnimelibAppletu, čo mu umožnilo vyhnúť sa kódu appletu). Až teraz to príde – použitie knižnice:

• • •

```
public class QuicksortApplet extends AnimelibApplet {

   @Override
   protected void run() {
      ArrayList elements;

      elements = new ArrayList();
      for (int i = 0; i < 10; ++i) {
         elements.add(new Integer((int) (Math.random() * 100)));
      }

      elements =
         new AnimatedWrapperArrayList(elements, BoxLayout.X_AXIS, false, this,
            true, false, null, "sorted array");

      quickSort(elements);
   }

   public void quickSort(ArrayList elements) {
      if (!elements.isEmpty()) {
         this.quickSort(elements, 0, elements.size() - 1);
      }
   }
```

• • •

Hotovo. Dopísal tri riadky (parametre vedel podľa dokumentácie), ale ak teraz spustí applet, uvidí vykreslené jeho pole čísel (ak by triedil reťazce, tak by sa vykreslili reťazce, ak by to bolo dvojrozmerné pole, vykreslí sa dvojrozmerné pole, atď.), a môže krokovať algoritmom tam a späť.

Takúto vizualizáciu zrejme bude chcieť vylepšiť. Napríklad tým, že zvýrazní pivotov, alebo aj nejak inak, záleží na ňom. V takom prípade zavolá rozšírené metódy na animovanej verzii poľa, asi takto (zvýraznenie čísla s indexom index):

```
((List < Animation >) ((Wrapper) elements).getAnimated()).get(index)
        .highlight();
```

Sú potrebné dve pretypovania z toho dôvodu, že tieto rozšírené metódy sú skryté pred programátorom, aby ho čo najmenej obťažovali.

2.6 Radix sort

Spolu s naprogramovaním boli pripravené tri applety algoritmov, ktoré predvádzajú, čo knižnica dokáže, a nasledujúce podkapitoly vysvetľujú, ako to možno jednoduchým spôsobom dosiahnuť.

Prvým ukážkovým appletom je viacnásobný radix sort (viac o algoritme v [12]). Algoritmus bol vybraný z toho dôvodu, že sa jednoducho implementuje a je zaujímavý.

Konkrétne ide o trojnásobný radix sort s desiatimi spojovými zoznamami, ktoré predvádzajú samotné triedenie. Triedi sa pole čísel, ktoré sú menšie ako 1000 (aby 3-násobný radix sort stačil).

Použitie knižnice je určite najlepšie vidieť zo zdrojového kódu, ktorý je možné nájsť na priloženom CD v adresári CD/src/appletpack/RadixSortApplet.java.

2.6.1 Inicializácia poľa na zotriedenie

Programátor nejakým spôsobom vytvorí pole čísel. Naplní ho niekoľkými náhodnými celými číslami. Toto pole je pole čísel, ktoré chce zotriediť.

Keďže ho chce vizualizovať (toto pole je zaujímavé počas celého algoritmu), obalí ho do obalu knižnice pomocou konštruktora. Zároveň si do referencie, s ktorou pracuje dosadí tento nový objekt (pretože má rovnaké rozhranie, je to možné), a ďalej pracuje s ním.

Na to, aby bolo pole vizualizované už nie je potrebné nič ďalšie robiť, všetko zariadi práve skonštruovaný obal poľa z knižnice. Pole je od tohto momentu vizualizované.

2.6.2 Inicializácia 10 spojových zoznamov

Logicky sa ničím nelíši od predošlej inicializácie poľa na zotriedenie, je to tu uvedené z toho dôvodu, že sa vlastne bude vizualizovať *pole* spojových zoznamov čísel. Takéto pole ide samozrejme obalom z knižnice obaliť, a tá v konštruktori cez reflexiu zistí, akého typu sú prvky poľa, aby automaticky obalila a vizualizovala všetko potrebné. Programátor má teda v kóde navyše opäť len jeden konštruktor. Je to nevyhnutné z toho dôvodu, že chce toto pole vizualizovať. Knižnica sa to nejako musí dozvedieť, a toto je ten spôsob.

Prakticky to teda vyzerá tak, že programátor nejakým spôsobom nainicializuje 10 spojových zoznamov, dá ich do svojho poľa, a potom si toto pole obalí obalom z knižnice (t.j. zavolá jeden konštruktor), aby sa vizualizovalo. O nič viac sa nemusí starať, môže ďalej pracovať s obalom, ako by to bol pôvodný objekt, a knižnica všetko vizualizuje.

2.6.3 Triedenie

Samotné triedenie už v kóde vyzerá, ako pôvodný kód, a všetko je knižnicou automaticky zobrazované.

Ak však chce vizuálnu stránku nejako vylepšiť, môže. Musí o svojich objektoch vedieť, že sú to v skutočnosti obaly jeho originálnych objektov, a teda pretypovanie na rozhranie Wrapper počas behu programu prejde. Na rozhraní Wrapper už má metódu getAnimated(), ktorá vráti animovanú verziu originálneho objektu. Ak vie, akého je typu (dozvie sa to z referenčnej príručky – číslu odpovedá AnimatedNumber, reťazcu AnimatedString, atď.), môže opäť pretypovať, a volať špecifické pomocné grafické operácie. Napríklad v radix sorte sa hodí zvýraznenie číslice, podľa ktorej sa triedi.

2.6.4 Ďalšie použité vlastnosti knižnice

Táto ukážka algoritmu využíva zo všetkých štyroch najviac ďalších vlastností knižnice, pretože v nej sú použité skiny...

```
AnimeApp.setSkin(new SandSkin());
```

Vyššia úroveň pauzovania (kód spôsobí, že každé volanie metódy pause() bude pauzovať na úrovni 1):

```
AnimeApp.getInstance().getPausingManager().setUserPausing(1);
AnimeApp.getInstance().getPausingManager().setPausingLevel(1);
```

Pseudo kód:

```
AnimeApp.getInstance().pseudoCode("preparing data structures");
```

A rozšírené grafické možnosti:

```
// Highlight the digit.
((List < AnimatedNumber >) ((Wrapper) all.get(digit)).
      getAnimated()).get(all.get(digit).size() - 1).
            setHighlightedDigit(imp + 1);
```

2.7 Alfa beta

Alfa beta (viac o algoritme v [13]) bola vybraná preto, lebo sa často používa na prehľadávanie stromov (napr. hier), a išlo o to ukázať vizualizáciu stromu. Navyše sa jednoducho implementuje.

2.7.1 Tvorba náhodného stromu

Pre alfa betu je potrebné vytvoriť náhodný strom hĺbky aspoň 4, aby na ňom bolo niečo vidieť. Programátor musí použiť rozhranie stromu z knižnice, pretože v Jave neexistuje zabudovaná podpora pre strom.

Programátor má k dispozícii implementáciu stromu (implementáciu rozhrania Tree), ale ak by mu nevyhovovala – ak by mal vlastnú implementáciu, tak môže použiť návrhový vzor Adapter (ktorý je stručne vysvetlený v časti 2.1, podrobne v [8], [9]), a prispôsobiť si implementáciu na toto rozhranie.

V ukážkovom príklade to je spravené tak, že programátor používa implementáciu z knižnice. Táto implementácia je implementácia normálneho (neanimovaného) stromu. V pomocnej metóde naplní strom do hĺbky 4, a nakoniec ho, keďže ho chce zobraziť, obalí do obalu pre stromy. Tým sú jeho starosti o vizualizáciu vyriešené.

2.7.2 Alfa beta

Algoritmus alfa-bety je naprogramovaný, nie je na ňom takmer žiaden kód navyše (až na jediný príkaz, ktorý obalí strom) a vizualizácia by v pohode fungovala. Ale ak chce programátor zlepšiť výstup, opäť môže použiť rozšírenia knižnice, ktoré zlepšujú grafický výstup, tentoraz je vhodné stromu povedať, ktorým uzlom sa programátor zaoberá, čím sa začnú vizualizovať prechody medzi uzlami:

```
((AnimatedGeneralTree < AnimatedNumber >)
 ((Wrapper) tree).getAnimated()).highlightNode(parent);
```

2.8 Kruskalov algoritmus

V tomto príklade sa pracuje s grafom, teda knižnica podporuje aj grafy. Ak by mal programátor svoju implementáciu grafu, tak sa musí opäť pomocou vzoru Adapter (stručné vysvetlenie je v časti 2.1, podrobné v [8], [9]) prispôsobiť môjmu rozhraniu. Inak to bohužiaľ nejde, pretože knižnica nemôže dopredu vedieť, akú implementáciu bude programátor používať.

V ukážke Kruskalovho algoritmu (viac o algoritme v [12]) sa pracuje s grafom, odoberú, a potom sa pridávajú hrany. Je v ňom aj pole hrán, ktoré sa automaticky obalia pomocou obalu pre reťazec (zobrazená je ich textová reprezentácia). Rozloženie hrán je zvolené do kruhu, pretože tu vyzerá lepšie ako náhodné.

Kapitola 3 – Analýza a návrh

3.1 Správa požiadaviek

3.1.1 Počiatočné požiadavky

Počiatočné požiadavky na funkčnosť knižnice boli vizualizácia základných dátových štruktúr, teda napr. spojového zoznamu, poľa, stromu a grafu.

Tiež bola požiadavka na to, aby bola knižnica použiteľná na viacerých platformách, menovite na Windowsoch a systémoch typu UNIX. Programovací jazyk bol zvolený C++.

Pôvodná predstava bola taká, že sa budú (ako ukážky) vizualizovať primárne kompresné algoritmy.

3.1.2 Zmena ukážkových aplikácií knižnice

Po nejakom čase sa ukázalo, že návrh knižnice nie je príliš šťastný, a nastali veľké zmeny v návrhu. Z hľadiska požiadaviek sa zmenilo len to, že knižnica nebude orientovaná primárne na kompresné algoritmy, ale bude radšej robená všeobecne, pre ľubovoľné algoritmy. Táto zmena vyplývala z nadobudnutých skúseností s naprogramovaním kompresných algoritmov, ktoré sa ukázalo zbytočne zložité, keďže malo ísť len o ukážku aplikácie knižnice.

Namiesto Huffmanovho kódovania, LZ77 a ďalších, budú teda ukážkové algoritmy radix-sort, alfa-beta a Kruskalov algoritmus, ktoré nie sú o nič menej zaujímavé.

Výrazná zmena tiež nastala vo voľbe použitých technológií, pretože sa prešlo na programovací jazyk Java a technológie s ňou súvisiace.

3.1.3 Použitie knižnice cez obaly

Neskôr, keď už mala knižnica istú podobu, a dokázala vizualizovať vytýčené algoritmy, vznikali nové požiadavky, ktoré zlepšovali jej použitie z hľadiska programátora algoritmu.

Dátové štruktúry, ktoré sú súčasťou knižnice, majú fungovať ako obaly neanimovaných (bežných) dátových štruktúr z Javy s rovnakým rozhraním. To znamená, že programátor algoritmu obalí svoju dátovú štruktúru jej ekvivalentom z knižnice animelib, a potom používa jej obal (ktorý má rovnaké rozhranie, a teda sa s ním úplne rovnako pracuje). Knižnica už sama zariadi, že všetky zmeny sa budú vizualizovať. Nutný zásah do kódu programátora je teda minimálny (len obalenie dátových štruktúr, ktoré sa majú vizualizovať). Výsledok je taký, že rozdiel medzi algoritmom, ktorý nie je vizualizovaný a vizualizovaným ekvivalentom, je (napríklad) 5 riadkov, ktoré volajú knižnicu.

Programátorovi ostane k dispozícii aj všetka vizualizačná logika, aby mohol vizualizáciu svojho algoritmu zlepšovať.

3.1.4 Krokovanie algoritmov a ďalšie

Krokovanie algoritmov tu znamená asi toľko, že keď skončí jeden krok algoritmu, beh programu sa pozastaví, a po kliknutí užívateľa – diváka algoritmu na tlačítko „Step" sa znova rozbehne.

Knižnica si sama pamätá predošlé stavy dátových štruktúr, a dokáže sa vracať na predošlé stavy algoritmu. Implicitne knižnica krokuje sama (pri každej zmene dátovej štruktúry), ale programátor si môže nastaviť vyšší stupeň krokovania, a krokovanie knižnice sa vtedy vypne.

Knižnica tiež podporuje skiny, pseudo-kód, globálne vypnutie/zapnutie animácie, a pod.

3.2 Návrh knižnice

Návrh je na knižnici asi najdôležitejší, musí byť taký, aby bola použiteľná. U knižnice animelib prešiel návrh mnohými zmenami, zmeny v návrhu trvali prakticky počas celého vývoja.

3.2.1 Pôvodný návrh knižnice

Pôvodne bola vizualizácia algoritmov robená ako program, ktorý dokázal interpretovať zdrojový kód napísaný v jazyku C. Keďže bol robený v C++, tak sa pomocou knižnice wxWidgets (www.wxWidgets.org) podarilo spraviť verziu, ktorá fungovala na oboch požadovaných platformách, a zvládala animáciu jednoduchých algoritmov, ktoré používali dátové štruktúry len polia a jednoduché typy.

Použiteľnosť takéhoto programu však bola mizerná, pretože programátor algoritmu musel vedieť, ako majú presne vyzerať dátové štruktúry, aby sa správne vizualizovali. Napr. binárny strom musel byť uložený v poli podľa presne určených pravidiel. Ťažko si predstaviť programátora, ktorý by takýto program použil.

Navyše už voľba jazyka C na interpretovanie nie je najšťastnejšia, a naviac to už bolo implementované inde.

3.2.2 Zlom v návrhu

Nevýhody pôvodného návrhu boli očividné. Vymyslel sa teda nový návrh.

Programovací jazyk sa zmenil na Javu, čo nie je až také podstatné. Projekt dostal konečne podobu knižnice, ktorá pomáha vizualizácii. Hlavná myšlienka bola taká, že knižnica bude obsahovať špeciálne dátové štruktúry, ktoré okrem toho, že sa dajú normálnym spôsobom používať, vizualizujú to, čo sa s nimi robí.

Ukážky algoritmov sa v tomto prípade zvolili applety, aby boli prístupné z internetu pomocou prehliadača a JRE.

Multiplatformovosť sa vďaka Jave výrazne zlepšila, prakticky stačí, aby platforma podporovala Javu, a tá je v súčasnosti pomerne dosť rozšírená. Takisto sa zlepšila aj

použiteľnosť, programátor už pracuje so stromom ako stromom, atď. A implementácia takejto knižnice je podstatne jednoduchšia, keďže odpadá parsovanie kódu v jazyku C.

Tento zlom v návrhu je to najdôležitejšie, čo sa knižnici počas jej vývoja stalo. Chyby v predošlom návrhu nastali kvôli mojim malým skúsenostiam s navrhovaním aplikácii, a tiež kvôli nedostatku komunikácie s vedúcim.

3.2.3 Návrh pred obalmi

Celá knižnica teda zatiaľ vyzerala tak, že v nej bolo pár dátových štruktúr, ktoré mali síce podobné rozhranie, ako ich originálne ekvivalenty (napr. java.util.LinkedList ([6])), ale napr. prvkami takého animovaného spojového zoznamu mohli byť len ďalšie animované objekty. Lenže keď toto videl programátor, tak sa mu nepáčilo, že sa musí učiť pracovať s mojimi animovanými objektmi. Nepáčilo sa mu to preto, lebo rozhranie síce bolo podobné, ale nebolo úplne rovnaké, ako rozhranie pre prácu s bežnými objektmi.

Animované dátové štruktúry dedili od ich neanimovaných ekvivalentov, a obsahovali jednu grafickú komponentu, do ktorej sa vykresľovali. K pôvodnému rozhraniu bolo pridaných niekoľko pomocných operácií nad nimi, ktoré mali zlepšovať výstup.

3.2.4 Obaly

Pre programátora sú pripravené obaly bežných dátových štruktúr, ktoré stačí na jednom mieste v algoritme obaliť, a odvtedy by ich mal programátor používať. Výhoda oproti animovaným dátovým štruktúram je tá, že rozhranie majú úplne rovnaké, ako originály. Priamo prístupné nie sú ani žiadne operácie navyše, aj keď programátorovi aj teraz ostala možnosť dostať sa až k animovaným ekvivalentom, a tým môže zlepšiť animáciu.

Obaly sú riešené tak, že v sebe vnútri obsahujú jak originálnu dátovú štruktúru, tak jej animovaný ekvivalent, a ešte sú sami obalom, ktorý vracia ako výsledok operácii ďalšie obaly (pretože prvkami obalu poľa Integerov musia byť obaly Integerov). Pri každej operácii na obale sa zmenia všetky tri dátové štruktúry, a teda každá operácia sa animuje, a zároveň mení aj originálny objekt (ak by chcel náhodou programátor s vizualizáciou objektu prestať).

Predošlý návrh, ktorý fungoval, samozrejme nebol zahodený (ani nemohol byť, pretože animované dátové štruktúry sa používajú vnútri obalov), a programátorovi teda stále ostáva možnosť pracovať s animovanými štruktúrami. Týmto spôsobom má jednoduchšiu kontrolu nad výstupom algoritmu, ale písanie algoritmov je o niečo pracnejšie. Je na ňom, ktorý spôsob si zvolí. Všeobecne vhodnejšie sa zdá použitie obalov, pretože sa môže sústrediť na algoritmus (zásah do kódu algoritmu je na pár miestach).

3.3 Detaily návrhu

Z predošlého textu má zrejme programátor len nejasnú predstavu o tom, ako je celá knižnica organizovaná a ktorá trieda čo dokáže, a ako to robí. Z toho dôvodu bude ešte stručne popísané, čo je kde umiestnené a ako to funguje.

3.3.1 Rozhrania

Knižnica ponúka dve rozhrania, ktoré chýbajú v Jave, je to Tree a Graph. Bola snaha o čo najvšeobecnejší návrh, aby boli použiteľné všade. Výsledok je taký, že u stromu má každý vrchol ľubovoľný počet potomkov, a graf je orientovaný, chýbajú mu však možno viacnásobné hrany, ale tie sa zle vizualizujú.

S týmito rozhraniami súvisí ďalšie rozhranie Position, ktoré reprezentuje pozíciu v dátovej štruktúre. V grafe to je vrchol, v strome uzol, atď. Na Position je zaujímavé to, že má jedinú metódu, ktorá pristupuje k prvku, ktorý je uložený vnútri. Ide o vzor Decorator, ktorý mení rozhranie ([8], [9]). Position existuje z toho dôvodu, že graf a strom majú svoju implementáciu Position, v ktorej potrebujú isté pomocné operácie zohľadňujúce ich štruktúru (väčšinou podľa vzoru Composite – uzol stromu môže byť rodičom a zároveň potomkom ďalších uzlov stromu, viac v [8], [9]).

Ďalej tu je rozhranie Animation. Implementujú ho všetky triedy, ktoré reprezentujú animované verzie dátových štruktúr (napríklad animovaný spojový zoznam). Animation poskytuje základné metódy, ktoré sú spoločné pre všetky animované dátové štruktúry, ako napríklad vypnutie/zapnutie animovania, nastavenie rýchlosti animovania, zvýraznenie, atď. Ako príklad možno uviesť animovaný strom animovaných reťazcov, ktorý chce povedzme nastaviť rýchlosť svojej animácie na 30%.

```
animatedTree.setSpeedOfAnimation(0.3f);
```

Takýto príkaz nastaví rýchlosť animácie stromu aj všetkých jeho prvkov na 30% z počiatočnej rýchlosti.

Veľmi podobným je ďalšie rozhranie, Wrapper, ktoré spĺňa podobnú funkčnosť, ale pre všetky obaly. Obsahuje spoločné metódy, ako vrátenie animovanej verzie dátovej štruktúry, vrátenie obalu, a pod.

3.3.2 Implementácia stromu a grafu

Knižnica ponúka implementáciu stromu a grafu, keďže v Jave sa tieto dátové štruktúry nenachádzajú. Ide o prostú implementáciu odpovedajúcich rozhraní. Obidve implementácie majú svoju vlastnú vnútornú implementáciu Position, podľa vzoru Composition ([8], [9]). U stromu sa táto trieda volá Node, a u grafu je to Vertex a Edge (graf má ohodnotené vrcholy aj hrany).

3.3.3 Animované dátové štruktúry

Ako už bolo spomenuté, všetky animované dátové štruktúry implementujú Animation, ktorá obsahuje isté spoločné metódy. Animovaných tried, s ktorými sa potom dá pracovať, je šesť: animované číslo, reťazec, pole, spojový zoznam, strom a graf.

Všetky animované dátové štruktúry sú organizované tak, že obsahujú vnútornú pomocnú triedu, ktorá dedí od grafickej komponenty Swingu ([5], [6]), JComponent, a stará sa o vykresľovanie. Hlavná logika vykresľovania je teda v nej.

Animovaný reťazec kreslí do upraveného JLabelu ([6]), ktorému pridáva pozadie (JLabel svoje pozadie normálne nekreslí). Vykresľovacia logika sa tiež stará o zvýrazňovanie celého reťazca. Animovaný reťazec dedí od animovaného objektu.

Animované číslo - jeho vykresľovacia komponenta rozširuje komponentu animovaného reťazca, pretože toho majú veľa spoločného. Vlastne jediná vec, ktorou sa číslo líši od reťazca je vnútorné uloženie hodnoty. Navonok vyzerá číslo ako reťazec číslic. Animované číslo však má nejakú tú funkčnosť navyše – napríklad umožňuje zvýrazňovať číslice (hodilo sa pri ukážke radix sort), alebo zobraziť číslo v inej, ako desiatkovej sústave.

Animovaný spojový zoznam kreslí do JPanelu (obyčajný obdĺžnikovitý grafický kontajner, detaily v [6]), po rade za sebou svoje prvky. Medzi prvky pridáva šípky.

Podobné je animované pole, ktoré vykresľuje taktiež do JPanelu, ale bez šípok. Obsahuje podporné metódy, ktoré umožňujú lepšie nastaviť výstup, napríklad zarovnanie veľkostí všetkých prvkov, alebo vertikálne, či horizontálne pole.

U animovaného stromu a grafu sa tiež používa JPanel, ale tentoraz určuje rozloženie prvkov samotná štruktúra, pretože u týchto dátových štruktúr je bežné rozloženie z Javy nevyhovujúce. U grafu si môžeme vybrať z dvoch verzií rozloženia grafu – do kruhu a náhodné. Tieto dva spôsoby boli zvolené kvôli jednoduchej implementácii. Ďalšie spôsoby vykreslenia možno do knižnice jednoducho doplniť – stačí implementovať LayoutManager, čo je Javovské rozhranie pre layout managery ([5], [6], http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html). Takúto implementáciu potom stačí predať konštruktoru animovaného grafu.

3.3.4 Obaly

Všetky obaly (okrem obalu na reťazec obalu na celé číslo) sú robené tak, že dedia od triedy, ktorú aj obaľujú. Vnútri agregujú tri referencie na túto triedu. Jedna referencia je animovaná verzia, ktorá sa má pri každej zmene zmeniť. Druhá je originálny objekt, ktorý bol obalený, aby sa aj uňho pri každej zmene obalu prejavili zmeny. A tretia referencia je väčšinou this, ale u obalu na reťazec je to odkaz na pôvodnú objekt (aby táto referencia bola vždy rovnakého typu, ako obaľovaný objekt). Táto tretia referencia – ak ide napríklad o pole, obsahuje prvky, ktoré sú zase automaticky obalené.

Z toho vyplýva, že pri konštrukcii obalov sa použije nejaká tá reflexia nad pôvodnými objektmi, aby sme boli schopní vytvoriť im odpovedajúce obaly.

3.3.5 Ostatné možnosti knižnice

Ostatné možnosti knižnice, teda globálne funkcie, ako vypnutie/zapnutie animácie, podpora pre skiny, pre krokovanie, pre pseudo-kód, sú riešené pomocou globálneho singletonu AnimeApp, na ktorom sú metódy, pomocou ktorých sa každá z uvedených oblastí rieši. Napríklad globálne sa animácia vypne takto:

```
AnimeApp.getInstance().globallyDontAnimate();
```

Kapitola 4 – Implementácia

4.1 Použité technológie

4.1.1 Vývojové nástroje

Keďže knižnica je napísaná v Jave, nie je žiadny problém zohnať voľne dostupné programátorské nástroje na jej vývoj. Za zmienku stoja hlavne Eclipse (www.eclipse.org), NetBeans (www.netbeans.org), IntelliJ IDEA (www.jetbrains.com/idea) a Jcreator (www.jcreator.com). Animelib bola robená v Eclipse, kvôli robustnosti, dobrej rozšíriteľnosti pomocou plug-inov a slušnej rýchlosti.

Do Eclipse bol doinštalovaný plug-in, ktorý pomáhal zlepšiť kvalitu kódu (Checkstyle, http://checkstyle.sourceforge.net/), plug-in, ktorý zlepšil výkon Eclipse (Keep Resident, http://suif.stanford.edu/pub/keepresident/), a plug-in, ktorý pomáhal sledovať, ako postup v projekte (Metrics, http://metrics.sourceforge.net/).

Testovanie bolo robené pomocou knižnice Junit (http://www.junit.org/).

Na automatizované zostavenie ukážok a zabalenie knižnice do balíčka (JAR, [5]) bol použitý voľne dostupný Ant (http://ant.apache.org/).

4.1.2 Programovací jazyk

Ako bolo už veľakrát spomenuté, použitým programovacím jazykom je Java. V súčasnosti ide o jeden z najpopulárnejších programovacích jazykov (podľa niektorých štatistík je dokonca najpopulárnejší, napríklad podľa [12] vedie v rebríčku SERP). Hlavným motívom pre výber Javy bola práve požiadavka na multiplatformovosť. Ale zlepšilo to aj celkovú použiteľnosť knižnice. Ako druhý jazyk pripadal v úvahu C++. To by ale knižnica mala príliš veľa problémov s multiplatformovým grafickým výstupom.

Konkrétna verzia Javy je 5, kvôli lepšej typovej kontrole na úrovni kompilátora (generics, [6]) oproti predošlým verziám, a kvôli tomu, lebo to bola najnovšia verzia Javy.

4.2 Zvládnuté technické prekážky

4.2.1 Grafický výstup v Jave

Na grafický výstup bolo použité grafické rozhranie Swing z Javy ([5], [6]). Časom sa ukázalo zopár technických prekážok, ktoré bolo nutné zvládnuť.

4.2.2 Layouty vo Swingu

Swing má zaujímavo vyriešené ukladanie grafických komponent do grafického kontajneru. Je tu zopár preddefinovaných tzv. layout managerov ([5], [6]), ktorí sa starajú o umiestnenie všetkých komponent v rámci jedného kontajneru. Programátorovi

potom stačí volať len metódu add nad kontajnerom, a komponenty sa umiestňujú *automaticky*.

Problém je trochu v tom, že každá komponenta má nadefinované 3 veľkosti – minimálnu prípustnú, maximálnu prípustnú, a preferovanú. Každý layout manager sa potom pozrie na určité z nich (u rôznych managerov je to rôzne), a podľa toho sa rozhodne, aká bude skutočná veľkosť.

4.2.3 Práca s viacerými vláknami

Pri programovaní bolo použitých viacero vlákien, už len kvôli tomu, že sa programovali aj applety. Osobitné vlákna sa v knižnici používajú napríklad pri zvýraznení animovaných objektov.

4.2.4 Iniciácia appletov

Tu bol menší problém s tým, že pri štarte appletu ([5], [6]) je nevyhnutné spustiť mu osobitné vlákno, inak sa „štartovanie appletu" nikdy neskončí.

4.2.5 Reflexia

Počas väčšiny projektu to vyzeralo, že sa reflexii v Jave ([5], [6]) bude možno vyhnúť, ale potom prišli obaly, a reflexia sa ukázala nevyhnutná. Na najvyššej, užívateľskej (z pohľadu knižnice), úrovni síce programátor obaľuje sám, je však nevyhnutné, aby aj prvky obaleného objektu boli obalené. Keďže obaly musia byť vytvárané dynamicky, reflexia je nevyhnutná. Ide o reflexiu pomerne jednoduchú, stačí len zistiť typ obaľovanej premennej.

Metóda, ktorá sa stará o reflexiu, pretože je rekurzívne volaná z konštruktorov obalov (a z ďalších metód, ktoré automaticky obaľujú), vyzerá takto:

```
/**
 * Converts regular object to its wrapped version.
 *
 * @param object
 *          Object to be wrapped.
 * @return Wrapper object of original object.
 */
public Wrapper convertToAnimatedWrapper(final Object object) {
   // null -> null.
   if (object == null) {
      return null;
   }

   // Integer -> AnimatedWrapperInteger.
   if (object instanceof Integer) {
      return new AnimatedWrapperInteger((Integer) object);
   }

   // Number -> AnimatedWrapperNumber.
   if (object instanceof Number) {
      return new AnimatedWrapperNumber((Number) object);
   }

   // LinkedList -> AnimatedWrapperLinkedList.
   if (object instanceof LinkedList) {
      return new AnimatedWrapperLinkedList < Object >((LinkedList) object);
   }
```

```
        // atď...

        // Object -> AnimatedWrapperString.
        return new AnimatedWrapperString(object);
    }
```

Obalenie ArrayListu ArrayListov čísiel teda vyzerá takto: Programátor zavolá konštruktor obalu na ArrayList, ktorému ako parameter predá obaľované pole. Knižnica vie, že ide o pole, teda zavolá vyššie uvedenú metódu nad každým prvkom, a zapamätá si obalené prvky, ktoré pridá do nového poľa. Vnútorne (v uvedenej metóde) sa zase pomocou reflexie volajú konštruktory AnimatedeWrapperArrayListov, a potom obalov na Integery, všetko rekurzívne, a pre programátora transparentne. Orientáciu si programátor môže určiť len na najvyššej úrovni, pretože zvyšok je už obaľovaný automaticky. ArrayList je implicitne zobrazovaný horizontálne, teda ak chce programátor vidieť 2D pole, obalí si najvyššiu úroveň vertikálne (pomocou parametra obalu).

4.3 Testovanie

Testovanie je dôležitou súčasťou vývoja softwaru, niektoré metodológie ho dokonca stavajú na popredné miesta (je to súčasť tzv. Best practicies ([14])). Na testovanie animelib boli použité unit testy pomocou knižnice JUnit .

4.3.1 Test-driven development

Pri vývoji knižnice bola použitá technika Test-driven development (TDD, [14]). Popularizovala ju metodika Extrémne programovanie, ale v súčasnosti sa využíva aj vo viacerých iných metodikách.

Ide o to, že najprv sa napíšu testy, a potom sa implementuje funkčnosť tak, aby tieto testy prešli. Každý test môže buď prejsť, alebo zlyhať, a funkčnosť je hotová, keď všetky testy prejdú.

Výhoda písania testov je zrejmá. Okrem otestovania všetkých funkčností testy pomáhajú pri vývoji, pretože nútia programátora písať bezchybný kód. Za správny kód sa považuje práve taký, ktorý prejde všetkými testmi.

4.3.2 Testy v knižnici animelib

Testy sa píšu jednoducho, stačí si prejsť funkčné požiadavky na nejakú triedu, a podľa nich napísať kód, ktorý by mal fungovať, keby už funkčnosť bola hotová (implementovaná).

Na automatizované testy bola použitá bežne dostupná a známa knižnica JUnit, ktorá má pre prostredie Eclipse plug-in s pekným grafickým výstupom.

V rámci vývoja bolo napísaných celkovo niečo okolo 70 testov, ktoré testovali všetky funkčnosti knižnice. Nakoniec všetky dopadli úspešne.

4.4 Dokumentácia

4.4.1 JavaDoc

V Jave je štandardom písať dokumentáciu formou JavaDocu (http://java.sun.com/j2se/javadoc/). Ide o špeciálne predpisy komentárov pred metódami, triedami a pod. Zdrojový kód sa potom dá za vstup utilite JavaDoc, ktorá z komentárov vygeneruje dokumentáciu (v takmer ľubovoľnej podobe – implicitná je HTML dokumentácia, ale pomocou Docletov sa dá vygenerovať takmer hocičo – PDF dokumentácia JavaDocu nerobí žiadny problém). Vygenerovanú dokumentáciu je možné nájsť na priloženom CD v adresári CD/docs/api.

Kapitola 5 – Záver

5.1 Porovnanie s alternatívnymi projektmi

Pokus prepísať triediaci algoritmus quicksort (ktorý bol už hotový v Jave) do vizualizovanej podoby trval asi 5 minút. Bolo potrebné zmeniť typ triedeného poľa z Vectoru (zastaralý) na List (alebo na ArrayList), a každé volanie elementAt na odpovedajúci get. Potom už len stačilo obaliť pôvodné pole obalom z animelib, a vizualizovali sa (a boli krokovateľné) všetky zmeny, ktoré v poli nastali. Ak by bolo treba vylepšiť vizualizáciu, pridalo by sa zvýraznenie bloku, ktorý sa práve triedi.

Pre porovnanie, ak by chcel podobnú funkčnosť dosiahnuť programátor sám, bez pomoci akýchkoľvek knižníc, musel by viečo vedieť o grafike v Jave (o Swingu), niečo o ovládaní vláknien (kvôli krokovaniu) a musel by celú vizualizačnú funkčnosť sám naimplementovať, čo by mohlo zabrať niekoľko hodín.

Knižnice ako JDSL alebo net.datastructures by mu pomohli, ak by pracoval so stromami alebo grafmi, ale ani tak by zaňho nevizualizovali algoritmus. Až JDSL Vizualizer poskytuje podobnú funkčnosť, ako knižnica animelib, ale práca s ním nie je až taká jednoduchá, a programátor musí doplniť do svojich dátových štruktúr niekoľko metód špecifických pre JDSL Vizualizer navyše. Okrem toho má slabšiu podporu napríklad pre krokovanie alebo skiny. Základná vizualizácia by sa s ním teda dala dosiahnuť asi za hodinu.

5.2 Prínos knižnice

Knižnica animelib vypĺňa medzeru v knižniciach, ktoré sa zaoberajú dátovými štruktúrami, pretože máloktorá z nich sa venuje vizualizácii algoritmov. Je užitočná, ak chceme bez veľkej námahy animovať algoritmus. Výhoda je, že sa nemusíme zaoberať zobrazovacou logikou, tá je za nás riešená knižnicou.

5.3 Splnenie cieľa

V rámci tejto bakalárskej práce bola vytvorená knižnica animelib slúžiaca pre pohodlnú tvorbu prezentácií algoritmov. Dôraz bol kladený na minimálne obťažovanie užívateľa – programátora, a čo najväčšiu užívateľskú prívetivosť grafického výstupu.

Pre vizualizáciu naprogramovaného algoritmu zapísaného v jazyku Java stačí pár príkazov, ktoré volajú knižnicu, a tá sa už postará o samotnú vizualizáciu a krokovanie v algoritme.

Knižnica je pripravená pre použitie vývojármi, ktorí majú záujem na vizualizácii svojho algoritmu, a chcú napísať čo najmenej kódu. Ak programátor zistí, že je preňho vhodná (že obsahuje všetky dátové štruktúry, ktoré on potrebuje), môže mu výrazne uľahčiť prácu.

5.4 Možnosti ďalšieho vývoja

Knižnica je ľahko rozšíriteľná o ďalšie špeciálnejšie dátové štruktúry, ktoré by do nej mohli pribudnúť. Grafický výstup by sa dal zlepšiť. Tiež by sa dali ako sprievodný projekt pripojiť ďalšie applety a mohlo by vzniknúť slušné množstvo ukážkových algoritmov. Knižnicu by šlo prerobiť do Javy s nižšou verziou, aby boli applety spustiteľné aj pre užívateľov napríklad s Javou 1.4, ktorých je v súčasnosti (2006) ešte stále dosť.

Literatúra

[1] Prof. RNDr. Luděk Kučera, DrSc.: Algovision, http://kam.mff.cuni.cz/~ludek/Algovision/Algovision.html

[2] Craig Larman (2001): Applying UML and Patterns - An Introduction To Object-Oriented Analysis And Design And The RUP. Prentice Hall.

[3] Michael Goodrich, Roberto Tamassia, Eric Zamore (2003): net.datastructures – http://net3.datastructures.net

[4] Roberto Tamassia, Robert Cohen, Michael Goodrich a kol. (2000):JDSL – http://www.jdsl.org/

[5] Sun Microsystems (1995): Java Tutorial - http://java.sun.com/docs/books/tutorial/

[6] Sun Microsystems (2004): Java 2 Platform Standard Edition 5.0 API Specification – http://java.sun.com/j2se/1.5.0/docs/api/

[7] Ondřej Hanousek (2005): Kompresní algoritmy, jejich implementace a vizualizace, http://service.felk.cvut.cz/courses/36KOD/vizualizace/index.html

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1997): Design Patterns CD – Elements of Reusable Object-Oriented Software. Addison-Wesley.

[9] ing. Miloš Dvořák (2003): Návrhové vzory, http://objekty.vse.cz/Objekty/Vzory

[10] Peter Brummund a kol.: Complete Collection of Algorithm Animations, http://www.cs.hope.edu/~alganim/ccaa/

[11] The Eclipse Foundation: Eclipse, www.eclipse.org

[12] Wikipedia, The Free Encyclopedia, www.wikipedia.org

[13] Bruce Moreland (2001): Computer Chess – Programming Topics, http://www.seanet.com/~brucemo/topics/topics.htm

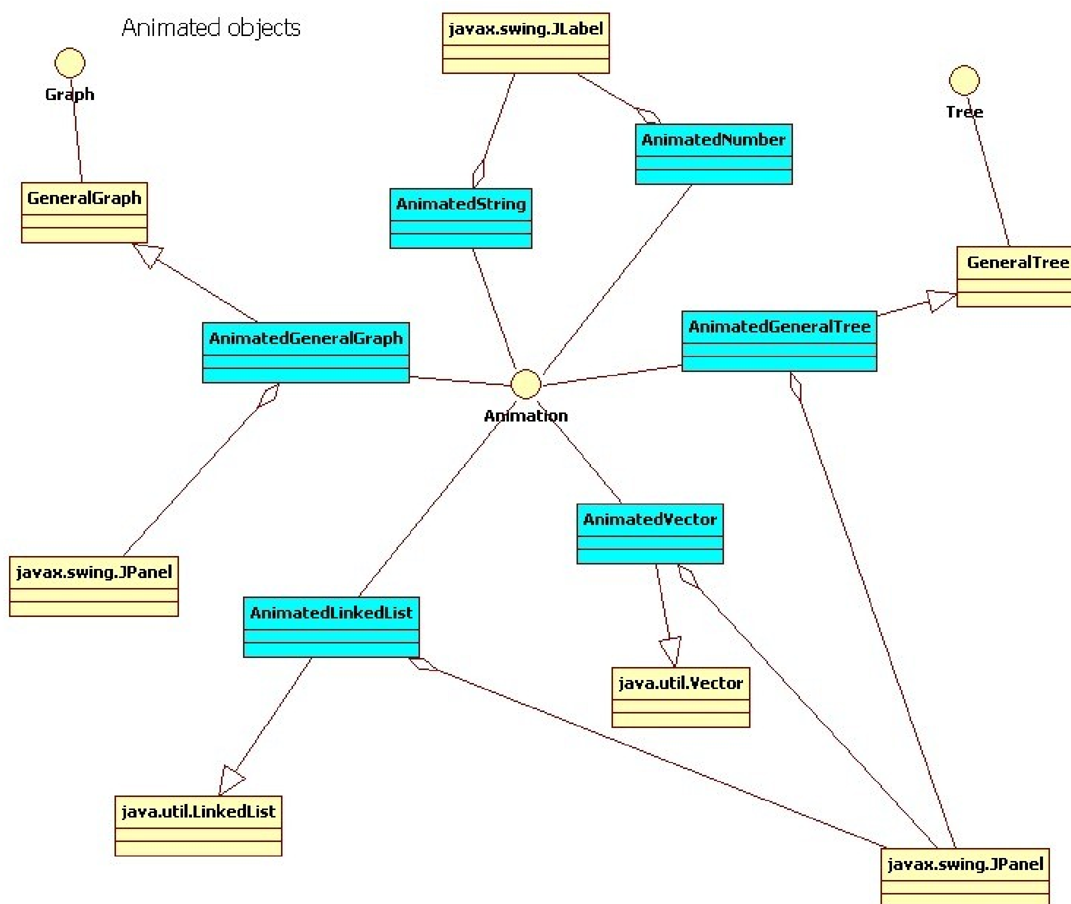
[14] Steward Baird (2002): Sams Teach Yourself Extreme Programming in 24 Hours

Prílohy

V prílohe nájdete návrhové diagramy knižnice vo forme class diagramov z UML ([2]).

Animované objekty

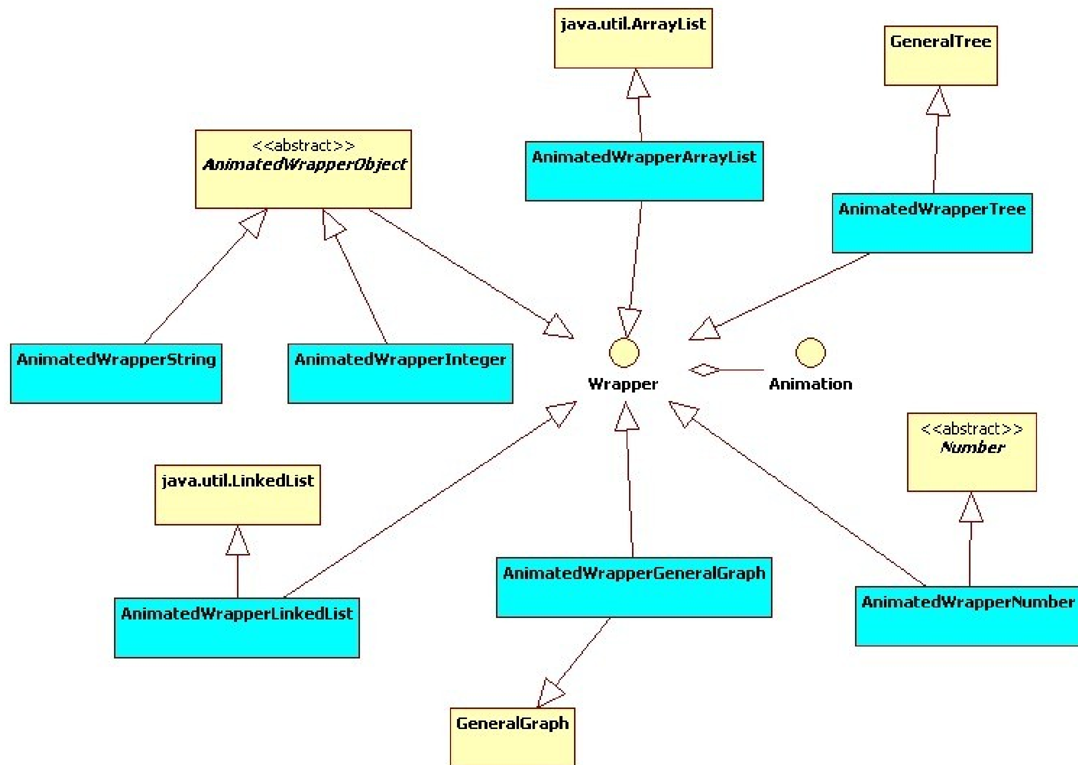
Všetky animované objekty implementujú rozhranie Animation, ktoré je zobrazené v strede. Väčšina z nich potom dedí od odpovedajúcich neanimovaných implementácií, a obsahuje nejakú grafickú komponentu, do ktorej vykresľuje.



Obaly

Všetky obaly implementujú rozhranie Wrapper umiestnené v strede. Dedia od dátových štruktúr, ktoré zároveň obaľujú. Každý obal obsahuje tiež animovaný ekvivalent dátovej štruktúry, ktorú obaľuje. Ten si vytvára sám a sám doňho premieta zmeny, ktoré sa s ním udejú.

Wrapper objects



Obsah CD

CD obsahuje:

- ukážky appletov v adresári CD/bin/apps
- použiteľnú knižnicu, ktorú možno integrovať do Javovského projektu v adresári CD/bin/animelib
- kompletné zdrojové kódy v adresári CD/src
- referenčnú príručku (JavaDoc) v adresári CD/docs/api
- tento text v adresári CD/docs/manual