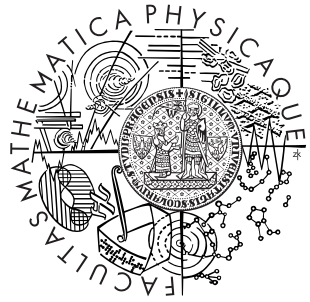
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Zdeněk Louženský

Animace algoritmů vnějšího třídění

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Rudolf Kryl,
Studijní program: Informatika, Programování

2006

Na tomto míste bych rád poděkoval vedoucímu mé bakalářské práce RNDr. Rudolfu Krylovi za podnětné připomínky a rady při vývoji Animatoru i při vypracování této bakalářské práce.

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

V Praze dne 10. 8. 2006 Zdeněk Louženský

Obsah

Název práce: Animace algoritmů vnějšího třídění
Autor: Zdeněk Louženský
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Rudolf Kryl
e-mail vedoucího: Rudolf.Kryl@mff.cuni.cz

Abstrakt: Tato bakalářská práce popisuje hlavní komponenty softwarového díla Animator. Program Animator umožňuje uživatelům studovat algoritmy vnějšího třídění. Uživatelé mohou algoritmy trasovat nebo je nechat animovat a pozorovat pásky s tříděnými prvky pomocí různorodě zaměřených pohledů také nazývaných zobrazení. Průběh algoritmu mohou ovlivňovat přenastavováním hodnot proměnných nebo změnou prvků uložených na páskách. Také se mohou zaměřit na zkoumání efektivity algoritmu pomocí statistik, které jsou v průběhu třídění shromažďovány. Animator je dále možné rozšiřovat o nové algoritmy a obohacovat o nové pohledy na pásky s tříděnými prvky.
Klíčová slova: Animator, animace, vnější třídění

Title: Animation of file sorting algorithms
Author: Zdeněk Louženský
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Rudolf Kryl
Supervisor's e-mail address: Rudolf.Kryl@mff.cuni.cz

Abstract: This bachelor's thesis describes main components of the Animator software project. Animator offers studying of file sorting algorithms. Users can single-stop algorithms or watch tapes with sorted elements on variable oriented views. Algorithm progress may be influenced by changing variable values or elements placed on tapes. Users also may orient on examining algorithm efectivity using statistics collected during algorithm progress. It is possible to extend Animator by new views or file sorting algorithms.
Keywords: Animator, animation, file sorting

Předmluva

Tato práce pojednává o softwarovém díle Animator (spustitelná verze je na přiloženém CD k této práci). V následujících kapitolách se budu věnovat jeho prezentaci a popisu hlavních komponent.

V první kapitole představuji projekt. Nahlížím na historii vzniku tohoto projektu a nabízím seznam nástrojů a předností, kterými tento projekt, dle mého názoru, předčí ostatní.

Druhá kapitola se zabývá popisem nejdůležitějších objektů, jejich propojení a principu fungování animace.

Kapitola třetí je věnována popisu objektů reprezentujících pásky a postupu jak s těmito objekty pracovat.

V kapitole čtvrté se hlouběji zaměřuji na vizualizaci animace. Popisuji třídy, které jsou při vizualizaci používány a budoucím autorům nových pohledů na tříděná data předkládám praktické informace, které se mi podařilo během vývoje projektu nasbírat.

Poslední velká kapitola je věnovaná animovaným algoritmům. Detailněji je zde rozebírán, z mého pohledu, největší problém, na který jsem během vývoje tohoto projektu narazil. Dále se tato kapitola zabývá objekty reprezentujícími jednotlivé algoritmy. Nabízí stručný postup a přehled nástrojů, jenž je možné při obohacování prostředí o nové algoritmy využívat.

V závěrečné kapitole zhodnocuji svůj projekt, nově získané zkušenosti a podávám možnosti jeho využití.

Kapitola 1

Představení projektu Animator

1.1 Základní popis

Program Animator slouží uživatelům jako pomůcka při studiu algoritmů vnějšího třídění. Snaží se jim přiblížit základní principy algoritmů vnějšího třídění, usnadnit jim pochopení konkrétních algoritmů, ale i poskytnout informaci o efektivitě algoritmu. Uživatelé mohou algoritmy trasovat nebo je nechat animovat a pozorovat pásky s tříděnými prvky pomocí různorodě zaměřených pohledů. Průběh algoritmu mohou ovlivňovat přenastavováním hodnot proměnných nebo změnou prvků uložených na páskách. Nebo se mohou zaměřit na zkoumání efektivity algoritmu pomocí statistik, které jsou v průběhu třídění shromažďovány. Animator je dále možné rozšiřovat o nové algoritmy a obohacovat o nové pohledy na pásky s tříděnými prvky.

1.2 Vznik projektu

První seznámení s projektem proběhlo na konzultaci s RNDr. Rudolfem Krylem na jaře roku 2005. RNDr. Rudolf Kryl měl představu, že výsledkem projektu bude aplikace, která bude animovat nějakou podmnožinu algoritmů přednášenou posluchačům prvních ročníků na Matematicko-fyzikální fakultě UK v Praze. Jeho představa mě zaujala a začal jsem přemýšlet, kterou sadu algoritmů bych si zvolil. Původně mi přišli nejbližší algoritmy vnitřního třídění, ale na některé z dalších konzultací jsme se dohodli, že tento projekt se zaměří na algoritmy vnějšího třídění. Kvalitních animátorů algoritmů vnitřního třídění má student možnost na internetu najít velké množství, ale těch které jsou zaměřeny na vnější (souborové) třídění zdaleka tolik není.

1.3 Historie vývoje

Od prvního okamžiku, kdy jsem se nad projektem začal zamýšlet, jsem kladl velký důraz především na názornost celé animace a efektnost vizualizace. Tato myšlenka nebyla úplně v souladu s důrazem RNDr. Rudolfa Kryla na obecnost a budoucí rozšiřitelnost celého projektu o nové animace a algoritmy. Měl jsem pocit, že čím

obecnější prostředí pro animaci algoritmů navrhnu, tím méně bude výsledná animace názorná. Domnívám se, že později se mi podařilo nalézt přijatelný kompromis mezi mou představou o názornosti animace a rozšiřitelností projektu. Rozšiřitelnost projektu považuji, s odstupem času, za naprosto nezbytnou.

První verze aplikace, kterou jsem RNDr. Rudolfu Krylovi představil, umožňovala sledování průběhu algoritmu pouze na datech, která algoritmus modifikuje a statistikách, které jsou během třídění shromážděny. Animaci bylo možné krokovat po uměle vytvořených krocích, kterými byly např.: dočtení některé z pásek nebo utvoření nového běhu na pásce. Tato verze nabízela pěkný pohled na pásky s tříděnými prvky, animace byla plynulá a poskytovala dobrý vizuální dojem. Studenta však stavěla do pozice diváka této animace s malou možností ovlivňovat její průběh.

Při představení této verze RNDr. Rudolf Kryl namítl, že úkolem této aplikace není pouze poskytnutí hezkého pohledu na tvořící se setříděnou poslopnost prvků, ale že by měla studentovi umožnit i bližší zkoumání algoritmů, které animuje. Naznačil mi, že pokud bude chtít student pochopit princip fungování nějakého algoritmu, měl by alespoň vidět jeho zdrojový kód a stav, ve kterém se algoritmus právě nachází. Nebo ještě lépe, kdyby mohl některé části algoritmu studovat detailněji. Dalším problémem, který vyvstal, byla definice toho, co přesně je krok animace a jestli má něco společného s krokem algoritmu.

S RNDr. Rudolfem Krylem jsme usoudili, že současnou aplikaci bude nutné rozšířit o možnost trasování zdrojového kódu algoritmu a sledování hodnot některých význačných proměnných. Tím se také vyřešil problém s definicí kroku animace, ten je odvozený od toho, jakým způsobem uživatel algoritmus aktuálně trasuje (tedy má spojitost s krokem algoritmu). Varianty, které byly v souvislosti s tímto rozšířením rozebírány, detailněji popisuji v kapitoje věnované animovaným algoritmům.

Behem celého vývoje projektu jsem byl postaven před řadu problémů a směrů, kterými by se projekt mohl dále ubírat. Některé jsou detailneji rozebrány v následujících kapitolách této práce.

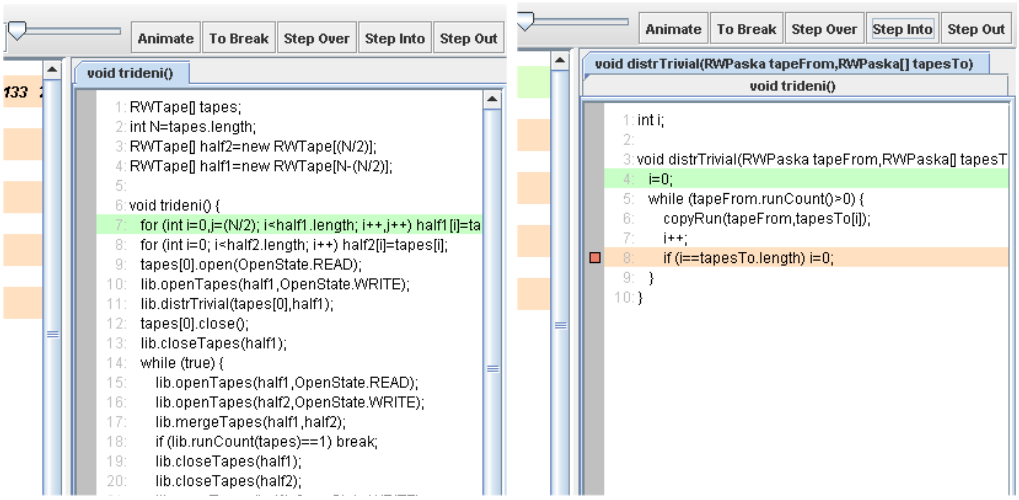
1.4 Co nabízí současná verze

Většina programů, které se zabývají animací algoritmů, se charakterem svého návrhu podobají filmu a uživatele staví do role jeho pasivního diváka, který nemá možnost větším způsobem tento film ovlivnit. Takové programy se většinou omezují na změnu rychlosti probíhající animace. Zřídka umožňují sledovat zdrojový kód algoritmu, natožpak ho trasovat, měnit hodnoty proměnných nebo přepínat mezi pohledy animace. Podobnou představu o svém budoucím projektu jsem měl i já. Naštěstí tato představa nepřetrvala a současná podoba mé aplikace nabízí uživatelům i další nástroje, pomocí kterých můžou průběh algoritmu detailně zkoumat a dokonce ovlivňovat. Hlavní přednosti a nástroje, kterými popisovaný program disponuje jsou uvedeny v následujícím seznamu:

- trasování algoritmu

  Uživatel má možnost nechat animaci algoritmu spustit a sledovat okem pasivního diváka jako film, ale také se může rozhodnout, že bude algoritmus zkoumat

detailněji. Součástí aplikace je panel, na kterém je možné sledovat kód animovaného algoritmu a provádět jeho trasování.
Po spuštění animace algoritmu je na tomto panelu možné sledovat hlavní podprogram algoritmu s deklaracemi proměnných. Uživatel má možnost provádět trasovací operace *Step Over* (přechod na další řádek kódu prováděného podprogramu), *Step Into* (vstup do podprogramu, který je na řádku volán nebo přechod na další řádek podprogramu) a *Step Out* (výstup z prováděného podprogramu zpět do podprogramu odkud byl tento volán). Při vstupu do nějakého podprogramu je na panelu otevřena nová záložka s kódem tohoto podprogramu a deklaracemi proměnných používaných tímto podprogramem. Počínaje tímto okamžikem je možné detailně zkoumat tento podprogram. Po jeho opuštění je záložka zavřena. Uživatele by mohlo odradit neustálé mačkání tlačítka pro přechod na další řádek algoritmu. Proto má možnost spustit animaci algoritmu. Aplikace pak provádí trasování algoritmu jako by uživatel opakovaně prováděl operaci *Step Over*. Mezi dvěma provedeními operace *Step Over*, aplikace vždy chvilku vyčká, aby si uživatel mohl uvědomit, k jakým změnám došlo. Tato čekací doba je volitelná. Uživatel však nemusí algoritmus trasovat po řádcích, má možnost k řádkům umisťovat breakpointy a algoritmus nechat zastavit až v okamžiku dosažení některého z nich. Uživatel má také možnost algoritmus kdykoliv přerušit a spustit ho znovu od začátku nebo si nechat animovat jiný algoritmus.

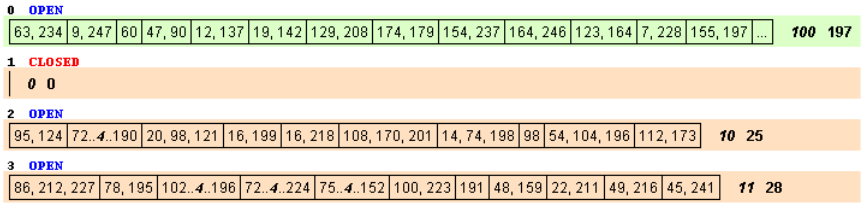


Obrázek 1.1: Ukázka panelu pro sledování a trasování algoritmu (aktuálně prováděný řádek podprogramu zvýrazněn zeleně, breakpoint červeně)

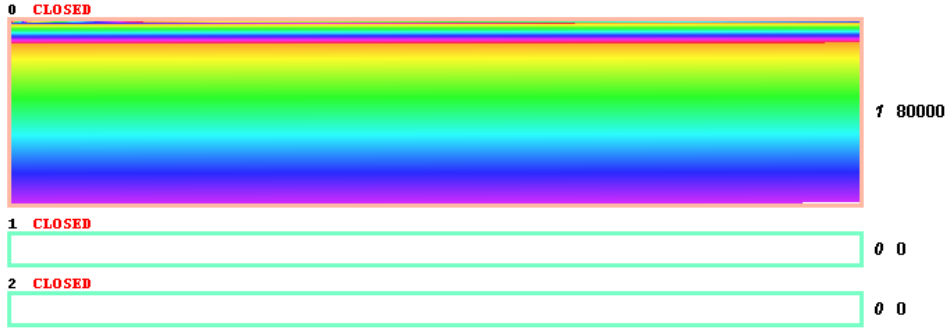
- různorodě zaměřené pohledy na pásky s prvky

  Aplikace umožňuje sledovat pásky s tříděnými prvky v několika různorodě zaměřených pohledech, nazývaných zobrazení. Tyto pohledy je možné kdykoliv v průběhu třídění mezi sebou přepínat. Prostřednictvím některých je možné sledovat běhy a konkrétní prvky, které je tvoří, některé umožňují porovnávat

délky běhů a některé mají uživateli nabídnout hezký vizuální dojem z průběhu algoritmu. Objektový návrh aplikace je navíc koncipován tak, aby bylo možné přidávat pohledy nové.



Obrázek 1.2: Ukázka zobrazení (barevné pruhy představují pásky, černé obdélníčky jsou běhy, některé jsou vyobrazeny ve zkrácené formě, na konci pásek počty běhů a prvků na pásce)



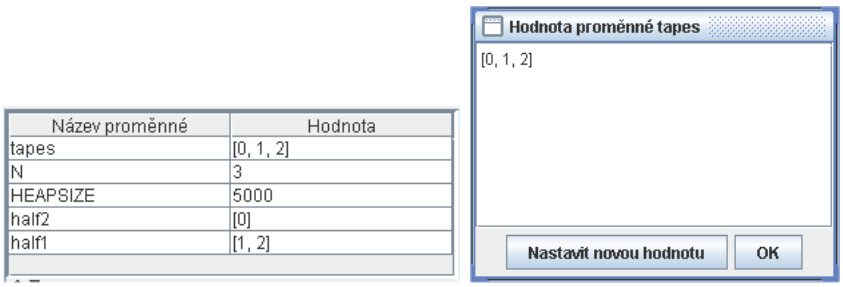
Obrázek 1.3: Ukázka zobrazení setříděné posloupnosti prvků (každý prvek pásky představován čtverečkem jehož barva závisí na velikosti prvku)

- možnost sledovat a měnit hodnoty proměnných a tříděných prvků

  Pomocí tabulky proměnných (Obrázek 1.4) má uživatel možnost sledovat jejich hodnoty. V případě, že by se hodnota proměnné nevešla do buňky tabulky, může uživatel na tuto buňku kliknout myší a zkoumat hodnotu proměnné v otevřeném okně (vhodné například u proměnných polí). V tomto okně je taktéž možné hodnotu proměnné nastavovat.
  Některá zobrazení umožňují měnit hodnoty prvků na páskách. Například zobrazení na obrázku 1.2. Po kliknutí na některý z běhů jsou v novém okně zobrazeny všechny prvky zvoleného běhu (Obrázek 1.5). V tomto okně je zároveň možné prvky běhu měnit.

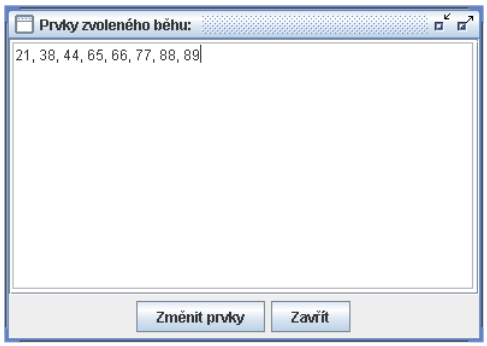
- detailní statistiky o průběhu algoritmu

  V průběhu třídění prvků jsou shromažďovány statistiky. Jednak jsou sbírány statistiky souhrnné, které zaznamenávají počty čtení a zápisů prvků do souboru, počty otevření a zavření souboru, za celou dobu běhu algoritmu pro každou pásku. Tyto statistiky je možné sledovat prostřednictvím tabulky, která je



Název proměnné	Hodnota
tapes	[0, 1, 2]
N	3
HEAPSIZE	5000
half2	[0]
half1	[1, 2]

Obrázek 1.4: Tabulka proměnných, hodnota proměnné v otevřeném okně



Obrázek 1.5: Okno s prvky zvoleného běhu

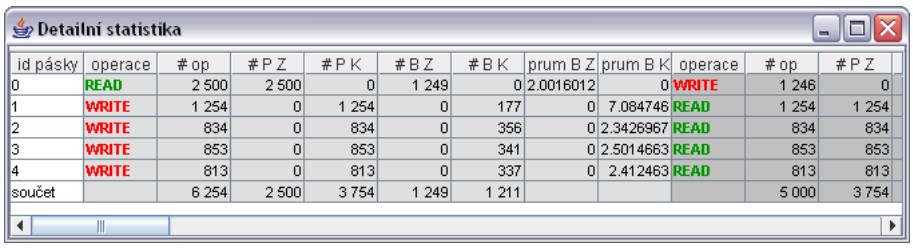
na obrázku 1.6. Také jsou zaznamenávány detailní statistiky v fázích (časový úsek od okamžiku otevření souboru pásky po jeho uzavření), ve kterých se páska nachází. Tyto statistiky zaznamenávají režim v jakém byla páska v dané fázi otevřena (čtení, zápis), počet provedených operací v této fázy, počty prvků a běhů na začátku fáze a na jejím konci a průměrné délky běhů. Podrobné statistiky je možné sledovat v okně, které je na obrázku 1.7.
Veškeré statistiky jsou vyhodnocovány aktuálně. Na základě nich je možné porovnávat efektivitu odlišných algoritmů.

id pásky	# čtení	# zápisů	# č+z	# otevření	# zavření	# o+z
0	539 893	539 893	1 079 786	12	12	24
1	360 107	360 107	720 214	11	11	22
2	311 554	311 554	623 108	12	12	24
3	308 362	308 362	616 724	12	12	24
4	280 084	280 084	560 168	12	12	24
součet	1 800 000	1 800 000	3 600 000	59	59	118

Obrázek 1.6: Tabulka se souhrnnými statistikami

- volitelná vstupní data

  Před spuštěním algoritmu si uživatel zvolí charakter a velikost vstupních dat. Může si zvolit data náhodně generovaná, klesající posloupnost prvků, může si nastavit kolik běhů má být z daného počtu prvků vygenerováno nebo si nechá prvky načíst z externího souboru, který si pro tento účel připraví.



id pásky	operace	# op	# P Z	# P K	# B Z	# B K	prum B Z	prum B K	operace	# op	# P Z
0	READ	2 500	2 500	0	1 249	0	2.0016012	0	WRITE	1 246	0
1	WRITE	1 254	0	1 254	0	177	0	7.084746	READ	1 254	1 254
2	WRITE	834	0	834	0	356	0	2.3426967	READ	834	834
3	WRITE	853	0	853	0	341	0	2.5014663	READ	853	853
4	WRITE	813	0	813	0	337	0	2.412463	READ	813	813
součet		6 254	2 500	3 754	1 249	1 211				5 000	3 754

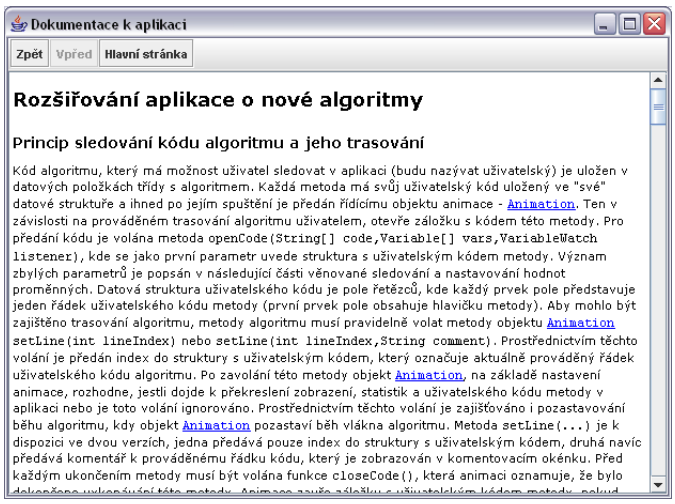
Obrázek 1.7: Okno s detailními statistikami (fáze jsou odlišeny barvou pozadí buněk)

- některé algoritmy jsou parametrizované

  U některých algoritmů je vhodné, aby si uživatel před jeho spuštěním zvolil hodnoty některých vstupních parametrů. Příkladem takového algoritmu může být algoritmus, který pro počáteční distribuci běhů využívá pomocnou datovou strukturu halda. U takového algoritmu je vhodné, aby si uživatel mohl nastavovat velikost použité haldy.
  Uživatel si před spuštěním animace algoritmu také nastavuje, kolik pásek bude algoritmus při třídění využívat. Proto je samozřejmostí všech algoritmů, že jsou přizpůsobeny pro práci s N páskami ($N > 2$).

- dokumentace přístupná z nápovědy programu

  Nezapomínejme na tuto nezanedbatelnou přednost. Mnoho programů má dokumetaci těžko dostupnou (například pouze z internetu) nebo po ní uživatel musí dlouze pátrat, pokud vůbec nějaká existuje.



Obrázek 1.8: Ukázka okna s dokumentací k projektu

Podrobnější přehled funkcí a možností jednotlivých nástrojů je možné nalézt v uživatelské dokumentaci projektu.

Kapitola 2

Jádro projektu

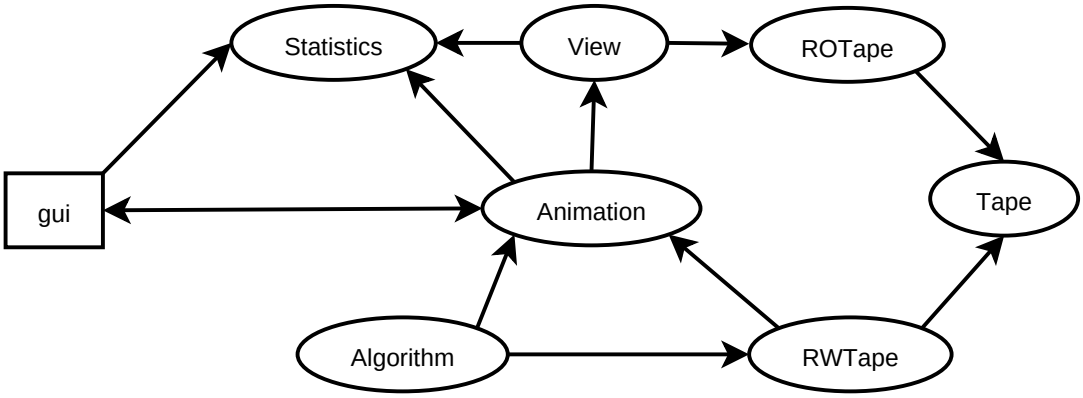
V této kapitole představím objekty tvořící „jádro“ projektu. Zaměřím se na objekty umožňující animaci algoritmů, shromažďování statistických informací, objekty reprezentující algoritmy a jednotlivá zobrazení a objekty sloužící pro práci s páskou. Popíšu jejich propojení a vzájemnou komunikaci.

2.1 Princip fungování animace

Než začnu detailně popisovat výše zmíněné objekty, zjednodušeně nastíním princip na jakém je založeno animování algoritmů. Každý animovaný algoritmus je potomkem abstraktní třídy `Algorithm`, která je potomkem třídy `Thread`. To znamená, že při vytvoření instance třídy algoritmu je vytvořeno vlákno, ve kterém třídící algoritmus poběží. Toto vlákno je spuštěno v okamžiku, když si uživatel aplikace spustí animování tohoto algoritmu. Kód algoritmu, který má uživatel možnost v aplikaci sledovat (budu nazývat uživatelský kód algoritmu) je uložen v datových strukturách třídy algoritmu. Každá metoda algoritmu, kterou je možné v aplikaci sledovat má svoji strukturu s kódem. Po zavolání této metody je předána struktura s uživatelským kódem objektu `Animation`, který rozhodne o jeho zobrazení, v závislosti na prováděné trasovací operaci uživatele. Volaná metoda pak pravidelně volá synchronizační funkce řídícího objektu `Animation`, prostřednictvím kterých je nastavován aktuálně prováděný řádek uživatelského kódu algoritmu. V případě, že je volání provedeno z metody, která je trasována, objekt `Animation` uspí vlákno algoritmu a tím pozastaví jeho vykonávání. V opačném případě je toto volání ignorováno a algoritmus běží dál. Pokud dojde k uspání vlákna, je nastaven aktuálně prováděný řádek uživatelského kódu, aktualizováno zobrazení a statistiky. Potom je buď běh vlákna algoritmu obnoven nebo animace vyčká s obnovením na provedení trasovací operace uživatelem.

2.2 Objekty a jejich komunikace

Objekty tvořící jádro projektu a nastínění jejich vzájemné komunikace je nejlépe vystiženo schématem na obrázku 2.1.



Obrázek 2.1: Schéma komunikace hlavních tříd projektu (Třída od níž vede šipka k druhé třídě, volá některé z metod druhé třídy.)

Z obrázku je patrné, že většina objektů komunikuje skrze centrální objekt `Animation`. Tato třída je nejdůležitějším článkem celé soustavy. Uchovává v sobě veškerá nastavení animace. Každá operace, kterou uživatel ovlivňuje animaci je předána tomuto objektu, který ji zpracuje. S tímto objektem komunikuje algoritmus (potomek abstraktní třídy `Algorithm`), který je animován, pásky, které jsou algoritmem využívány (objekty `RWTape`), zobrazení (potomci abstraktní třídy `View`) i objekt shromažďující statistiky průběhu třídění (`Statistics`) a balíček objektů zajišťujících uživatelské rozhraní aplikace (`gui`).

Animovaný algoritmus volá metody `Animation` pro zajištění jeho sledování a trasování. Ten navíc, v závislosti na nastavení animace, rozhoduje o překreslení zobrazení. Animovaný algoritmus pro práci s páskami používá objekty `RWTape`. Ty `Animation` oznamují operace s páskou, které algoritmus provádí a ten je dále předává objektu shromažďujícímu statistiky. Vedení této komunikace přes objekt `Animation` se zdá být nadbytečné, ale umožňuje projekt snadno rozšiřovat o nové objekty vyžadující informace o operacích prováděných s páskami. Navíc tím, že je komunikace vedena přes prvek, který v sobě uchovává veškerá nastavení animace, umožňuje objektu `Statistics` oznamovat jen ty operace, které jsou v souladu s těmito nastaveními. Zobrazení (potomek abstraktní třídy `View`) pro načítání dat z pásek volá metody objektů `ROTape` a pro získávání statistických informací o průběhu algoritmu metody objektu `Statistics`. Objekty `ROTape` a `RWTape` pro práci se souborem prvků využívají metody objektu `Tape`.

Někdo by mohl namítnout, že tím, že je většina komunikace vedena přes objekt `Animation` (někdy nadbytečně), bude tento model pomalý a bude zdržovat běh aplikace. Dovoluji si ale připomenout, že projekt který v této práci popisuji, má význam výukový a prezentační a u těchto projektů není na rychlost kladem tak velký důraz. Tím, že je velká část komunikace neustále vedena přes objekt `Animation` navíc umožňuje jeho budoucí snadné rozšiřování. Nově přidávané objekty je možné „napojovat“ právě na objekt `Animation`.

Přednosti návrhu

- možnost dalšího rozšiřování o nová zobrazení

  Objektový model je navržen tak, aby bylo možné projekt snadno rozšiřovat o nová zobrazení (potomky třídy `View`). Těm je věnována kapitola „Pohledy na tříděná data“.

- možnost dalšího rozšiřování o animované algoritmy

  Projekt je, kromě nových zobrazení, možné obohatit i o animace dalších algoritmů (potomky třídy `Algorithm`). Detailnější informace o obohacování aplikace o nové algoritmy a návody jak postupovat je možné nalézt v kapitole „Animované algoritmy“.

- možnost dalšího rozšiřování o nové nástroje

  Nastavení animace je centralizováno a taktéž nejdůležitější komunikace je vedena skrze objekt `Animation`. Proto je možné tento model snadno rozšiřovat, „napojováním“ nových objektů na objekt `Animation`.

Objekt Animation

Již mnohokrát jsem se zmínil o funkci tohoto objektu. Nyní se zaměřím na některé partie detailněji.

Animovaný algoritmus volá metody tohoto objektu pro předávání struktur s uživatelským kódem svých metod a následně pro synchronizaci jeho kódu s kódem uživatelským. `Animation` si ve svých datových položkách udržuje informaci o výšce zásobníku volaných podprogramů algoritmu. Po zavolání nějakého podprogramu algoritmu je výška zásobníku zvýšena, při jeho ukončení snížena. Kromě této hodnoty si ještě udržuje výšku zásobníku volaných podprogramů, do kterých uživatel vstoupil (operace *Step Into*) a trasuje je. Hodnoty těchto proměnných umožňují animaci řídit trasovací operace *Step Over* a *Step Out*. Při provádění trasovací operace *Step Over* objekt řídící animaci ignoruje všechna synchronizační volání algoritmu, dokud není výška zásobníku volaných podprogramů stejná jako výška zásobníku volaných podprogramů, do kterých uživatel vstoupil. `Animation` zareaguje tedy až na volání synchronizační metody, které je provedeno z aktuálně trasovaného podprogramu uživatelem. Obdobně je umožněna i operace *Step Out*, při které je navíc snížena výška zásobníku volaných podprogramů, které uživatel trasuje.

Dalším důležitým úkolem objektu `Animation` je přidělování zámku nad objektem pásky. Od otevření pásky zobrazením po její uzavření není algoritmu umožněno s páskou pracovat (kvůli jiné pozici v souboru). Objekt animace sice volá metody pro překreslení zobrazení v okamžiku, kdy je vlákno algoritmu uspáno, ale k překreslení dochází například ještě v případě, kdy byla plocha zobrazení zakryta nějakým jiným oknem a poté odkryta. V tomto případě, by mohlo dojít k tomu, že se začne překreslovat zobrazení a tedy dojde k otevření pásky zobrazením a před jejím zavřením by s páskou začal pracovat i algoritmus. `Animation` proto těmto objektům přiděluje zámek nad páskou a případně druhé vlákno uspí a spustí až v okamžiku, kdy je páska „volná“.

Objekt Statistics

Shromažďování statistických informací o využívání pásek algoritmem zajišťuje objekt `Statistics`. Využívá k tomu dvě hashmapy (jedna pro souhrnné a druhá pro detailní statistiky), jejichž klíčem jsou reference pásek a hodnotami struktury se statistickými informacemi. Pokaždé, když algoritmus provede nějakou operaci s páskou (objektem `RWTape`), informuje ho prostřednictvím objektu `Animation` o této operaci a předá mu referenci pásky. Statistiky si na základě této reference aktualizují hodnoty příslušných proměnných. Obdobným způsobem probíhá komunikace, při které `Statistics` předává nasbírané hodnoty zobrazení nebo objektům uživatelského rozhraní, které zobrazují tyto hodnoty v tabulkách.

Objektům reprezentujícím pásky, zobrazení a animované algoritmy jsou věnovány samostatné kapitoly této práce.

Rozdělení objektů do jednotlivých balíčků

Pro ucelení přehledu objektů, které tvoří jádro projektu uvádím jejich seznam a zároveň zařazení v jednotlivých balíčcích:

- balíček `animator:` `Animation`, `Statistics`, `DataGenerator` (slouží pro generování vstupních dat algoritmu)
- balíček `tapes:` `Tape`, `ROTape`, `RWTape`
- balíček `views:` `View` + potomci třídy `View`
- balíček `algorithms:` `Algorithm`, `ExternalSortingLibrary` (viz kapitola věnovaná animovaným algoritmům)
- balíček `algorithms.generated:` třídy s animovanými algoritmy

Součástí projektu je ještě balíček `gui`, který obsahuje objekty zajišťující uživatelské rozhraní aplikace.

Kapitola 3

Objekty reprezentující pásky

Objektový model mého projektu disponuje třemi objekty pro práci se soubory prvků. Z historických, ale i didaktických důvodů, tyto objekty nazývám pásky. Jsou jimi třídy `Tape`, `RWTape` a `ROTape`.

3.1 Charakteristika objektů

Objekt `Tape` je z výše uvedených nejpodstatnější. Tato třída obsahuje jako datovou položku soubor, ve kterém jsou prvky umístěné na pásce uloženy a poskytuje metody pro práci s tímto souborem. Umožňuje číst a zapisovat prvky do souboru, poskytuje metody pro jeho otevírání, zavírání a zjištění konce souboru. Objekt si ale také udržuje informace o datech, která v sobě uchovává. Dokáže poskytnout údaje o počtech prvků a běhů na pásce, dokáže indikovat začátek dalšího běhu. Také poskytuje metody, které vrací aktuální prvek, který má být z pásky přečten (neposouvá pozici na pásce) nebo metody pro změnu již uložených prvků na pásce (aby je mohl měnit uživatel). Většina metod je vyhotovena ve dvou verzích - první určená animovaným algoritmům a druhá zobrazením. To značně usnadnilo implementaci tohoto objektu.

Zobrazení i animované algoritmy by pro práci s páskou mohli přímo volat jim určené metody tohoto objektu. V takovém případě by ovšem mohlo snadno docházet k chybám spočívajícím v zaměnění metody určené pro algoritmus a metody, kterou může volat zobrazení. Ulehčení rozlišení těchto metod a zamezení vzniku podobných chyb jsem se snažil zabránit. Do objektového modelu jsem zařadil třídy `RWTape` a `ROTape`.

Třída `RWTape` nabízí pouze metody využitelné algoritmem. Jsou to metody pro čtení a zápis prvků na pásku, otevření a zavření souboru pásky a metody poskytující informace o prvcích uložených na pásce. Metody tohoto objektu volají jim příslušné metody objektu `Tape` určené pro algoritmy. Kromě toho navíc ještě informují objekt `Animation` o právě prováděné operaci (čtení prvku, otevření souboru). Ten je dále vyhodnocuje a informuje o nich další objekty, například statistiky.

Zobrazení získávají informace o páskách a jejich prvcích prostřednictvím objektů `ROTape`. Metody tohoto objektu umožňují číst prvky z pásky, poskytovat informace o datech na pásce a měnit prvky na pásce uložené. Ukázalo se, že měnit již uložené

prvky na pásce je uživatelsky nejpřívětivější prostřednictvím zobrazení. Narozdíl od objektu `RWTape`, neposkytuje metody pro zápis dalších prvků na pásku. Metody tohoto objektu volají příslušné metody třídy `Tape` určené zobrazením.

Zde se nabízí otázka, proč oddělení metod není zajištěno pomocí dědičnosti. Objekt `Tape` by mohl poskytovat nějaké základní metody pro práci s páskou a jeho potomci `ROTape` a `RWTape` by poskytovali metody specifické pro algoritmy a zobrazení. Uvědomme si ale, že zobrazení ukazuje prvky, se kterými pracuje algoritmus a že soubor, ve kterém jsou prvky uloženy je datovou položkou třídy `Tape`. V případě, že by třídy `RWTape` a `ROTape` byly potomky této třídy, pracovali by s odlišným souborem dat. Pokud by soubor nebyl datovou položkou třídy `Tape`, ale byla by jí předána reference na tento soubor, šlo by oddělení metod řešit i pomocí dědičnosti.

3.2 Třída Tape

Než jsem se pustil do implementace objektu `Tape`, zvažoval jsem několik možností na jakém principu by měl fungovat a v jaké formě budou prvky umístěné na pásce uloženy. Rozhodoval jsem se mezi ukládáním prvků do vnitřní paměti počítače, do sekvenčního souboru a do souboru s přímým přístupem. Varianta s ukládáním prvků do vnitřní paměti počítače by nabízela nejrychlejší dobu vybavování prvku z pásky a přístup ke všem prvkům na pásce s konstantní časovou složitostí. Vnější třídění je sice založené na sekvenčním přístupu k prvkům, ale v případě tohoto projektu kromě algoritmu pracuje s páskou i zobrazení. Je tedy nutné často měnit aktuální pozici na pásce. Program má ale charakter výukového a prezentačního a proto není vhodné, aby výrazně obsazoval vnitřní paměť počítače, k čemuž by došlo v případě, že by si uživatel nechal setřídit větší množinu prvků. Což je pro vnější třídění charakteristické. Variantu s uložením prvků ve vnitřní paměti počítače jsem proto zavrhl. Bylo tedy nutné rozhodnout se mezi sekvenčím souborem a souborem s přímým přístupem. Se souborem je většinou pracováno sekvenčně, ale také dochází k přesunu aktuální pozice při předávání pásky mezi algoritmem a zobrazením. V případě, kdy by byl použit sekvenční soubor, každý přesun pozice k počátku souboru by znamenal zavření a znovuotevření souboru. Což vyžaduje nemalou režii operačního systému. Jeho použití by také nebylo vhodné v případě, kdy páska pracuje v režimu *READBACK*. V tomto režimu jsou prvky z pásky čteny od jejího konce k začátku. Před otevřením pásky v tomto režimu by musel být celý soubor „převrácen“. Při použití souboru s přímým přístupem je možné nastavovat pozici v souboru vždy před prvek, který má být přečten. Rozhodl jsem se proto, že pro uložení prvků na pásce použiji soubor s přímým přístupem. Soubor je vytvořen ve složce operačního systému pro dočasné soubory a smazán po ukončení *Java Virtual Machine*. Jelikož je s ním pracováno sekvenčně, je pro práci se souborem používán pomocný buffer.

Páska, kromě metod pro ukládání a čtení prvků a zavírání a otevírání souboru, nabízí i metody poskytující informace o počtech prvků a běhů na pásce. Hodnoty těchto veličin si páska drží ve svých proměnných a při práci s páskou tyto hodnoty aktualizuje. Při každém přečtení prvku z pásky, je snížena hodnota proměnné udržující informaci o počtu prvků na pásce. Také je zkontrolováno, jestli po přečtení

tohoto prvku, prvkem následujícím nezačíná další běh. V takovém případě jsou aktualizovány i hodnoty proměnných, které uchovávají počet běhů na pásce a proměnná indikující začátek nového běhu. Obdobně jsou hodnoty upravovány při zápisu prvků na pásku.

Při předávání pásky mezi algoritmem a zobrazením je uložena pozice souboru pásky a hodnoty výše zmiňovaných proměnných. Z hodnot těchto proměnných je poté pozice obnovena. Poskytování aktuálního prvku, který má být z pásky přečten je řešeno tak, že si páska vždy načítá jeden prvek dopředu.

Při implementaci jsem zvažoval i alternativu, která by hodnoty výše zmíňovaných veličin vždy dopočítala. Tuto alternativu jsem velmi záhy zamítl, protože by pokaždé musela projít všechny prvky na pásce, což by bylo časově velmi náročné.

3.3 Postup práce s páskou

Protože je pro ukládání dat na pásku použit soubor, není těžké si domyslet jak se má postupovat při práci s páskou. Před prvním čtením nebo uložením prvku na pásku je nutné pásku nejprve otevřít. K tomu slouží metody `open(OpenState openState)` objektu `RWTape` a `open()` objektu `ROTape`. Metodě `open(OpenState openState)`, se jako parametr předává režim otevření. Režimy jsou následující: čtení, zápis, zápis na konec souboru (append) a čtení pásky pozpátku od jejího konce - umožňuje implementovat některé algoritmy používané v minulosti, kdy se data opravdu na pásky ukládala a doba přetočení pásky na začátek třídění velmi zpomalovala.

Po otevření je, v závislosti na zvoleném režimu, možné prvky z pásky číst (metody `read()`), zapisovat (jen algoritmus - metoda `write(int element)`), získávat informace o počtech prvků a běhů na pásce (metody `runCount()` a `elementCount()`) a další. Po ukončení práce s páskou je nutné ji zavřít metodou `close()`.

Pokud je páska otevřená zobrazením, není možné volat žádné metody určené animovaným algoritmům. Aktuální pozice v souboru se liší od té, na které s páskou pracuje algoritmus. V případě nekorektní práce s páskou je vyvolána některá z běhových výjimek, například `EmptyTapeException` v případě čtení dat z pásky, na které již nejsou žádné další prvky.

Detailní přehled metod pro práci s páskami je k nahlédnutí v programátorské dokumentaci k třídám, které je reprezentují.

Kapitola 4

Pohledy na tříděná data

V této části mé práce se budu věnovat pohledům na pásky s tříděnými prvky, třídě která je předkem všech těchto pohledů a poté se zaměřím na přidávání nových pohledů do aplikace.

4.1 Třída View

Každé zobrazení, které je možné sledovat, je potomkem abstraktní třídy `View`. Nabízím její zdrojový kód, který je pro účely této práce upravený (jsou u něj odstraněny těla metod):

```
public abstract class View  extends JComponent implements
MouseListener{
    protected ROTape[] tapes;
    protected Statistics stat;
    protected int N;
    protected String name;
    protected String helpFile;

    /** metoda kterou se inicializuji promenne */
    public void initialize(ROTape[] tapes,Animation anim,Statistics stat) {}
    /** metoda je volana pri pozadavku o~prekresleni komponenty */
    public final void paint(Graphics g) {}
    /** metody ktere by mel autor noveho zobrazeni naimplementovat */
    protected void smaz(Graphics2D g) {}
    protected abstract void vykresli(Graphics2D g);

    /** metody rozhrani MouseListener */
    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}

    /** metody pro ziskavani nazvu zobrazeni a souboru s~napovedou */
    public final String getViewName() {}
    public final String getHelpFileName() {}
}
```

Třída `View` je potomkem grafické komponenty `JComponent`. To s sebou přináší několik výhod: komponenta umožňuje kreslit na svůj povrch, komponenta sama pozná, kdy je jí nutné překreslit (např. při odsunutí okna, které ji zakrývalo) a sama volá metody pro překreslení. Komponenta používá dvojitý kreslicí buffer, což přináší její rychlejší překreslování. Programátor zobrazení může využívat metody poskytující informace o rozměrech komponenty a další.

Metoda `initialize(ROTape[] tapes,Animation anim,Statistics stat)` slouží k inicializaci proměnných této třídy. Je volána při každém spuštění animace nějakého algoritmu. Inicializace proměnných by právě proto neměli být prováděny v konstruktoru třídy, ale v této metodě.

Při každém požadavku o překreslení komponenty je volána metoda `paint(Graphics g)`. Pokud je zobrazení již inicializováno, jsou volány metody `smaz(Graphics2D g)` a poté `vykresli(Graphics2D g)`. Metoda `smaz(Graphics2D g)`, jak její název napovídá, má za úkol překreslit současný povrch komponenty barvou jejího pozadí. Metoda `vykresli(Graphics2D g)` by měla obsahovat instrukce zajišťující nakreslení nového „snímku animace“. V případě, že zobrazení ještě inicializováno nebylo, je na povrch komponenty vykreslena informační předdefinovaná zpráva „Zvolte algoritmus“.

Aby bylo zobrazení schopné reagovat na kliknutí a další akce myši, musí být naimplementována příslušná metoda rozhraní `MouseListener`, které třída `View` implementuje. Třída dále obsahuje metody vracející název zobrazení a název souboru s nápovědou k tomuto zobrazení. Soubor nápovědy musí být ve formátu HTML (*HyperText Markup Language*) a měl by obsahovat popis grafických prvků a hodnot, které je na tomto zobrazení možné sledovat. Tuto nápovědu je možné prohlížet v aplikaci. S názvem zobrazení se uživatel má možnost setkat například při přepínání mezi jednotlivými zobrazeními.

Zobrazení může, kromě informací a dat uložených na páskách přístupných skrze proměnnou `tapes`, kreslit i tabulky a grafy založené na aktuálních statistikách posbíraných během dosavadního běhu algoritmu, které je možné načítat z objektu `Statistics` - proměnná `stat`.

4.2 Rozšiřování o nové pohledy

Jediná metoda, kterou musí autor nového zobrazení povinně naimplementovat je metoda `vykresli(Graphics2D g)`. Je čistě pouze na autorovi pohledu, jak bude graficky zpracován a co bude jeho obsahem. Doporučuji však několik rad, které se mi během programování zobrazení, které jsou dostupné v aktuální verzi aplikace, podařilo nasbírat.

Každé zobrazení by mělo být co nejvíce názorné. Uživatel se tak nemusí dlouho zamýšlet nad tím, na co nahlíží. Po jednom přečtení dokumentace (která by měla být k zobrazení napsána vždy) by mu mělo být jasné, co symbolizují použité grafické elementy a jaký význam mají hodnoty, které je možné na zobrazení sledovat.

Pokud zobrazení nějakým způsobem od sebe odlišuje pásky, je vhodné aby u těchto pásek bylo napsáno i jejich označení, to je možné získat metodou `getCode()`. Uživa-

teli výrazně usnadní orientaci mezi páskami, tato označení se totiž objevují ve statistikách i části aplikace sloužící ke sledování hodnot proměnných. Dále by u každé pásky měly být vyobrazeny hodnoty počtů běhů a prvků na pásce. Obecně o každé pásce by mělo být vyobrazeno co nejvíce informací, které je možné zobrazit.

V případě, že páska zobrazuje zkrácenou formu některých subjektů (například jen části běhů), mělo by být umožněno prohlédnout si takový subjekt i v nezkrácené podobě, pokud ovšem je to rozumně proveditelné. Má smysl se dívat na všechny prvky běhu, který má délku padesát, ale zobrazit prvky běhu s délkou milion prvků by asi nebylo ani dost dobře proveditelné.

Java nepatří mezi jazyky, které by byly „rychlé“ a ani kreslení grafiky není nejrychlejší. Je tedy vhodné, aby autor zobrazení, které bude kreslit velké množství grafických prvků (řádově desetitíce a více), tyto elementy nekreslil přímo na povrch komponenty, ale nakreslil je nejdříve do nějakého pomocného bufferu (například `BufferedImage`) a ten potom nechal kompletně vykreslit na komponentu. Doba vykreslování snímku se tím výrazně zkrátí.

Kapitola 5

Animované algoritmy

Jakým způsobem je v rámci konceptu mého projektu řešeno animování algoritmů, jak se k této variantě dospělo, jakým způsobem současné prostředí správně obohatit o nové algoritmy a jaké nástroje k tomu využít popíšu v této kapitole.

5.1 Vývoj, problém a jeho řešení

V úvodní kapitole této práce jsem se zmínil, že mé původní představy o podobě tohoto projektu byly odlišné. V původních myšlenkách měl tento program sloužit spíše jako vizuální pomůcka při studování algoritmů vnějšího třídění. Jeho původním cílem bylo, aby uživatel pochopil základní principy vnějšího třídění. Na zkoumání jednotlivých algoritmů a rozdílů mezi nimi nebyl kladen až tak velký důraz. Proto původní verze nenabízela možnost trasování, dokonce ani sledování zdrojového kódu algoritmu. Zaměřovala se spíše na co nejlepší ztvárnění tříděných dat při průběhu animace algoritmu. S odstupem času jsem rád, že tato představa nepřetrvala až do současné verze.

Bohužel k přehodnocení mé původní představy došlo až v době, kdy velká část podstatných objektů byla naprogramována a nebylo tedy možné navrhovat projekt od začátku znovu. Zde se bohužel potvrdilo přísloví „dvakrát měř a jednou řež“. Původní verzi bylo tedy nutné rozšířit takovým způsobem, aby zásah do již hotového „jádra“ projektu nebyl příliš razantní. Zároveň bylo nutné poskytnout budoucím uživatelům aplikace dostatečně silný nástroj při sledování a trasování algoritmu. Dalším bodem, na který byl kladen velký důraz, bylo nalezení kompromisu mezi programátory algoritmů, které bude aplikace umět animovat a uživateli této aplikace. Bylo nutné programátorům nabídnout alespoň částečně přívětivý způsob, jakým se budou algoritmy programovat, při zachování dostatečných možností sledování a trasování algoritmu uživatelem aplikace. Pokud by kompromisní mez nebyla zvolena vhodně, mohlo by se stát, že by pro programátory bylo příliš složité a tím pádem i neatraktivní přidávat nové algoritmy. Na druhé straně v případě, že by aplikace nenabízela dostatečné trasovací možnosti, by toto rozšiřování nejspíš nemělo smysl.

Při prvním zamyšlení nad problémem mě napadly dva základní směry, kterými se vydat (kód algoritmu, který je možné sledovat v aplikaci nazývám uživatelský):

- uživatelský kód bude v Javě
- uživatelský kód bude zapsaný v pseudojazyku

Obě varianty s sebou přinášejí potřebu mít uživatelský kód uložený v nějaké datové struktuře a nějakým způsobem ho synchronizovat se skutečně prováděným zdrojovým kódem algoritmu. Této skutečnosti se při rozšiřování stávajícího „jádra“ projektu již nebylo možné vyhnout. Samozřejmě, že lepší alternativou by bylo naprogramovat interpret nějakého definovaného pseudojazyka, ve kterém by bylo umožněno psát algoritmy uživatelům aplikace a interpretovat je za běhu. To by ale znamenalo doslova přepracování celého projektu. Ale zpět k výše zmíněným variantám. Pokusím se shrnout jejich výhody a nevýhody. U obou variant je nutné napsat kód algoritmu dvakrát - jednou jako kód zdrojový, podruhé jako uživatelský. U obou variant musí být zajištěna synchronizace kódů. V případě, že si zvolíme variantu s uživatelským kódem v pseudojazyce, bude nutné ho zavést definicí. Tedy bude potřeba i nějaký nástroj, který bude kontrolovat korektnost uživatelského kódu vůči této definici. Na druhou stranu algoritmus popsaný dobře nadefinovaným pseudojazykem bude určitě daleko čitelnější než algoritmus popsaný Javou. Java je v mnoha věcech specifická a ne každý tento jazyk zná, i když její popularita stále roste.

Pro uživatelský kód v Javě bude možné využít zdrojový kód algoritmu. Navíc pokud bude součastí prostředí nástroj (budu jej označovat preprocesor), který naplní datové struktury s tímto kódem místo programátora, nebude muset, kromě několika málo instrukcí pro tento nástroj, psát nic navíc. Dokonce se nebude muset starat ani o provazování obou kódů, vše za něj může udělat preprocesor. Programátor tedy de facto napíše jen zdrojový kód algoritmu. V případě, že by algoritmus byl popsán pseudojazykem, tak silný nástroj bych programátorům nemohl nabídnout. Programátoři by museli napsat algoritmus dvakrát a ještě se prostřednictvím nějakých instrukcí postarat o synchronizaci obou kódů. Rozhodl jsem se pro varintu s uživatelským kódem v Javě.

Nyní je již vytyčeno, že kód zobrazovaný aplikací bude v Javě a že se v podstatě bude jednat o kopii zdrojového kódu tohoto algoritmu, vyhotovenou s pomocí nějakého preprocesoru. Ještě bylo potřeba zamyslet se nad tím jak bude kód zobrazovaný v aplikaci vypadat. Jestli se bude zobrazovat **celá třída s algoritmem jako jeden celek** nebo bude **uživatelský kód rozdělen na metody a zobrazovat se bude aktuálně prováděná metoda, případně s nějakými globálními deklaracemi** nebo zda-li se bude zobrazovat **jen metoda hlavní** - tedy metoda, která je zavolána při spuštění algoritmu, případně ještě nějaká další varianta.

Možnost, kterou jsem uvedl jako třetí, tedy bude se zobrazovat jen metoda hlavní, vyloučím hned na začátku. Metoda, kterou se spouští algoritmus vůbec nemusí obsahovat to podstatné z celého algoritmu a může volat další metody, které teprve budou z hlediska studia algoritmu těmi zajímavými. Tomu by se dalo předejít tím způsobem, že by se bucoucím autorům nových algoritmů vydala nějaká doporučení, kterými se mají při programování nového algoritmu řídit. Jedním z nich by se mohlo zajistit například to, aby hlavní cyklus algoritmu byl implementován v metodě, která je volána po jeho spuštění. Nedovedu si ale představit, že bych studoval nějaký algoritmus na jeho hlavním cyklu a neměl ponětí nebo si pouze domýšlel, co asi tak

dělají některé metody, které jsou v tomto cyklu volány.

Dá se říci, že přesným opakem třetí varianty je varianta první, kdy uživatel může sledovat celou třídu. Z uživatelského kódu této varinty se určitě dá vyčíst, co která metoda dělá a jakým způsobem. Otázkou je, jestli by se nedal studovat algoritmus ještě lépe? Co kdyby se po spuštění algoritmu uživateli zobrazila jen hlavní metoda, která by obsahovala hlavní cyklus algoritmu a uživatel by si při trasování algoritmu mohl zvolit zda-li vstoupit do podprogramu a nechat si zobrazit jeho kód. V okamžiku, kdy by se ocitl v podprogramu mohl by trasovat tento podprogram a po jeho dokončení by se jeho kód skryl a uživatel by pokračoval v trasování hlavního cyklu. Většině studentů by stačilo takovýto podprogram trasovat jen jednou. Uživatel by viděl vždy jen to, co je aktuálně zajímavé a na co by se měl soustředit. Jeho pozornost by nebyla rozptylována desítkami nebo stovkami řádků dalšího kódu, které ho v daný okamžik nezajímají. Samozřejmě, že když by se uživatel nacházel v některém podprogramu, bylo by mu umožněno přepnout se i na uživatelský kód podprogramů, ze kterých do aktuálně prováděného vstoupil. To co tu popisuji je druhá varianta, která se doplní nějakými doporučeními, kterými by se měl autor algoritmu řídit při jeho programování. Tuto varianta jsem si nakonec také zvolil. V následujících podkapitolách už se budu věnovat technickým detailům a samotným rozšířením.

5.2 Třída Algorithm

Každý algoritmus, který aplikace animuje je potomkem abstraktní třídy `Algorithm`. Zde nabízím její zdrojový kód (opět ochuzený o těla metod):

```
public abstract class Algorithm extends Thread {
    protected final RWTape[] tapes;
    protected final int N;
    protected final Animation anim;
    public static String name;
    protected ExternalSortingLibrary lib;
    protected String helpFileName;

    /** predany reference na pasky a ridici objekt animace */
    public Algorithm(RWTape[] tapes,Animation anim) {}

    /** metoda ktera obsahuje vlastni kod algoritmu */
    public abstract void trideni() throws Exception;

    /** metody pro praci s~parametry algoritmu */
    public static String[] getParams() {}
    public void setParam(String caption,String value) {}

    /** metoda je volana ke spusteni vlakna algoritmu */
    public final void run() {}

    /** metoda vraci nazev souboru s~napovedou */
    public String getHelpFileName() {}
}
```

Nejpodstatnější metoda celé třídy je metoda `trideni()`. Tuto metodu musí autor nového algoritmu pokaždé naimplementovat a právě tato metoda je zavolána při spuštění algoritmu. Při bližším pohledu na její definici, je možné si povšimnout, že metoda „vyhazuje“ obecnou výjimku. Budoucí autoři nových algoritmů by neměli zachytávat žádné běhové výjimky, které mohou být v průběhu algoritmu vyvolány. Tato, na první pohled trochu zvláštní skutečnost, má své opodstatnění. Aplikace uživateli umožňuje nastavovat hodnoty některých proměnných. Aby bylo možné v aplikaci zobrazovat chybové hlášky, ke kterým dojde špatnou změnou hodnoty proměnné, je nutné, aby výjimky byly propagovány vně všechny metody, které jsou v průběhu algoritmu volány. Animací jsou pak zachyceny a zpracovány.

Další metody, které jsou ve třídě obsaženy jsou metody pro práci s parametry algoritmu. Programátor algoritmu má možnost každý svůj algoritmus parametrizovat nějakou hodnotou, kterou nastaví uživatel aplikace před spuštěním tohoto algoritmu. Každý parametr je vhodně pojmenován, aby uživatel věděl, jakou hodnotu zadává. Pro práci s parametry slouží metody `getParams()` a `setParam(String caption,String value)`. První z nich prostředí předává pole řetězců s názvy parametrů, které má uživatel možnost nastavit. Druhá je volána prostředím pro nastavování hodnot těchto parametrů v algoritmu. Metoda `getParams()` je statická, oproti tomu metoda `setParam(String caption,String value)` statická není. Metoda `getParams()` je totiž volána, ještě před vytvořením objektu algoritmu,
kdy ještě není jasné jestli se uživatel rozhodne teto algoritmus animovat. Naopak metoda `setParam(...)` je volána až v okamžiku, kdy už uživatel tento algoritmus zvolí a je tedy vytvořena instance jeho třídy. Hodnota parametru je předávána jako `String`, tak jak ji zadá uživatel aplikace a je na programátorovi, aby si hodnotu převedl na správný typ, případně vyvolal nějakou běhovou výjimku, kterou aplikace zobrazí uživateli. Pro tuto funkci mi přišlo zbytečné definovat nějaké vlastní datové typy nebo sadu metod rozlišených podle typu, pro který jsou určeny.

Metoda `getHelpFileName()`, jak sám název napovídá, vrací název souboru s nápovědou k tomuto algoritmu, hodnotu proměnné `helpFileName`. Soubor s nápovědou k algoritmu musí být ve formátu HTML (*HyperText Markup Language*) a platí pro něj obdobná doporučení, jaká jsem uvedl pro nápovědu k zobrazením. Nápověda by měla detailně popisovat práci algoritmu, případně i význam některých proměnných.

Proměnná `tapes` představuje pole referencí na objekty pásek se kterými algoritmus pracuje, `N` udává jejich počet (tedy `tapes.length`), `anim` je reference na centrální objekt řídící animaci. Její metody jsou volány pro zajištění synchronizace uživatelského a zdrojového kódu. Statická proměnná `name` slouží pro uložení názvu algoritmu a proměnná `lib` je reference na knihovnu metod vnějšího třídění, kterou má autor algoritmu možnost při programování využít. Té se budu podrobněji věnovat v následující kapitole.

5.3 Rozšíření aplikace o nový algoritmus

V této podkapitole se zaměřuji na rozšiřování aplikace o nové algoritmy, popisuji jaké nástroje k tomu může autor algoritmu použít a stanovuji několik bodů, které by měl

každý nově přidaný algoritmus splňovat. Nebudu zde již více specifikovat položky třídy `Algorithm`, to jsem, myslím dostatečně, naznačil v kapitole předchozí.

Parametry algoritmu

Každý algoritmus může být parametrizován nějakými hodnotami. Příkladem takového algoritmu může být například algoritmus, který pro svou počáteční distribuci běhů používá datovou strukturu halda. U takového algoritmu je vhodné, aby uživatel mohl nastavit velikost použité haldy. Na následující ukázce demonstruji použití výše zmiňovaných metod pro prácí s parametry, pro nastavení velikosti haldy:

```
public static String[] getParams() {
    String[] params={"Velikost haldy"};
    return params;
}

public void setParam(String caption,String value) throws Exception {
    try {
        if (caption.equals("Velikost haldy")) HEAPSIZE=Integer.valueOf(value);
    }
    catch (Exception ex) {
        throw new Exception("Velikost haldy musi byt cele cislo!");
    }
}
```

Špatně zadané hodnoty parametrů je možné prostředí oznamovat pomocí běhových výjimek.

Zásady dobře studovatelného algoritmu

Než se zaměřím na postup při implementaci algoritmu, vydám několik doporučení, jejichž cílem je zajištění, co možná největší podobnosti uživatelského kódu od různých autorů algoritmu a zároveň snaha o lepší čitelnost a snazší studovatelnost kódu:

- hlavní cyklus algoritmu a jeho „jádro“ by měl být umístěn přímo v metodě `trideni()`
- při implementaci využívat již implementované metody knihovny `ExternalSortingLibrary` (na tu se zaměřím o několik řádků níže)
- používat omezené množství inicializačních metod na počátku algoritmu
- pro metody se specifickou funkcí (distribuční) používat specifické prefixy (`distr`)

Knihovna funkcí

Už několikrát jsem se zmínil o objektu `ExternalSortingLibrary`. Jedná se o knihovnu metod realizujících množství základních operací vnějšího třídění (kopírování prvků a běhů, slévání běhů, slévání pásek, metody pro různé druhy distribucí, atp.) a metod využitelných při práci s páskami (otevření několika pásek, zjištění pásky

s nejmenším prvkem). Prostřednictvím funkcí této knihovny má programátor možnost svůj algoritmus doslova „poskládat“ z jejich volání, výrazně si tím ulehčit práci a zároveň uživateli zjednodušit studování algoritmu. Je totiž docela možné, že už bude vědět co tato metoda provádí. Příklad použití této knihovny opět uvedu na příkladu. Jedná se o triviální algoritmus, který nejprve data z první pásky distribuuje na všechny ostatní a z nich slévá zpět na pásku první. Tento cyklus je opakován až do okamžiku, kdy je na páskách setříděná posloupnost prvků.

```
public void trideni() throws Exception {
    for (int i=1; i<tapes.length; i++) otherTapes[i-1]=tapes[i];
    while (true) {
        tapes[0].open(OpenState.READ);
        lib.openTapes(otherTapes,OpenState.WRITE);
        lib.distrTrivial(tapes[0],otherTapes);
        lib.closeTapes(tapes);
        tapes[0].open(OpenState.WRITE);
        lib.openTapes(otherTapes,OpenState.READ);
        lib.mergeTapes(otherTapes,tapes[0]);
        if (tapes[0].runCount()==1) break;
        lib.closeTapes(tapes);
    }
    lib.closeTapes(tapes);
}
```

Na příkladu je vidět, že naprogramování algoritmu s využitím knihovny `ExternalSortingLibrary` je opravdu snadné a rychlé. Obdobným způsobem by autor mohl vytvořit sadu podobných algoritmů lišících se například jen v distribuční fázi.

Sledování hodnot proměnných

Autorům nových algoritmů je umožněno zvolit si proměnné, jejichž hodnoty bude možné v průběhu animace sledovat a případně i měnit. Pokud se tak rozhodnou, musí naimplementovat potřebné metody rozhraní `VariableWatch` a vyplnit datovou strukturu se seznamem těchto proměnných (ta je poté předána objektu řídícímu animaci). Interface `VariableWatch` obsahuje metody pro získávání a nastavování hodnot proměnných základních datových typů, jejich polí a typů reprezentujících pásky. Bohužel programátor musí počítat s tím, že tímto způsobem je možné sledovat pouze proměnné, které jsou datovými položkami třídy.

Typické použití opět předvedu na příkladu. Uvažujme algoritmus, který jsem popsal výše a řekněme, že budeme chtít sledovat hodnotu proměnné `otherTapes`. Nejprve vytvoříme novou datovou položku třídy - strukturu s proměnnými, jejichž hodnoty budeme chtít sledovat:

```
private Variable[] vars_trideni={
    new Variable("otherTapes",Variable.Type.RWTAPEARRAY,"new RWTape[N-1]")
};
```

Tato struktura je potom předána třídě `Animation` a ta na základě ní vytvoří deklarace těchto proměnných, o které je doplněn uživatelský kód, a také umožní sledování a nastavování hodnot těchto proměnných prostřednictvím volání metod rozhraní

`VariableWatch`. Struktura je pole proměnných typu `Variable` a k jejímu vytvoření je použit konstruktor, jehož prvním parametrem je název sledované proměnné (ten je potom předáván jako parametr metodám rozhraní), dále typ proměnné a řetězec s nějakými počátečními inicializacemi, těmi je doplněna deklarace proměnné v uživatelském kódu. Pokud by programátor chtěl předat proměnnou, která má být uvedena pouze v deklaracích a nechtěl by sledovat hodnotu takové proměnné nebo by sledování takového datového typu proměnné nebylo podporováno rozhraním, může jako typ proměnné uvést `Variable.Type.OTHER`.

Nyní už stačí programátorovi naimplementovat metodu `getRWTapeArrayVar(String name)`, která vrací hodnotu proměnné na základě jejího názvu, který je předán parametrem `name`. Pokud by programátor chtěl umožnit i změnu této proměnné, musel by ještě naimplementovat metodu `setRWTapeArrayVar(String name,Integer[] codes)`. Prvním parametrem je opět název proměnné, druhým je pole identifikačních kódů pásek, které jsou aplikací zobrazovány. Pro získání tohoto kódu slouží metoda pásky `getCode()`.

Doplnění algoritmu o synchronizační volání

Algoritmus, který postupně v této kapitole utvářím už je téměř hotový. Jeho zdrojový kód už je napsaný, proměnné, které se mají sledovat jsou nastaveny. Teď už chybí pouze vygenerování struktury s uživatelským kódem tohoto algoritmu a doplnění zdrojového kódu o synchronizační volání prostředí. V tomto ohledu jsem se snažil programátorům opět vyjít maximálně vstříc. Součástí projektu je nástroj (nazývaný preprocesor), který se na základě minima instrukcí o toto postará. Jediné co musí programátor udělat, pokud chce umožnit sledování uživatelského kódu metody, je před začátek a za konec kódu metody doplnit speciální komentáře určené preprocesoru. Každý komentář preprocesoru začíná dvojicí znaků `PP`, po nich následuje mezera a další instrukce. Před začátek kódu metody se doplní instrukce `METHOD`, případně doplněná o instrukci `WATCH strukturaSledovanychPromennych`. Za koncem metody musí být instrukce `METHODEND`. Seznam všech použitelných instrukcí a jejich detailnější popis je možné nalézt v dokumentaci k projektu v části věnované použití preprocesoru.

Po spuštění preprocesoru je vygenerována nová třída umístěná do balíčku `algorithms.generated` (programátor může dále upravovat původní kód třídy v balíčku `algorithms`), doplněná o synchronizační volání metod a s naplněnými strukturami pro uživatelský kód. Volání mají následující tvar: před provedením prvního příkazu metody je volána metoda animace `openCode(...)`, které jsou předány struktury s uživatelským kódem metody, seznam sledovaných proměnných a reference na aktuální ojekt (kvůli volání metod rozhraní `VariableWatch`). Animace na základě tohoto volání pozná, že došlo ke vstupu do nějakého podprogramu a v závislosti na jejím nastavení se patřičně zachová - zobrazí kód podprogramu v aplikaci a umožní jeho trasování nebo ne. Poté následují instrukce algoritmu doplněné o volání synchronizační metody `setLine(int lineIndex)`, která prostřednictvím parametru `lineIndex` předává index aktuálně prováděného řádku uživatelského kódu.

Před výstupem z podprogramu je zavolána metoda `closeCode()`, signalizující výstup z této metody.

Tím je vše potřebné pro animaci algoritmu hotovo. Nově přidanou třídu s algoritmem již není nutné nikde registrovat, protože aplikace k načtení tříd s algoritmy využívá funkcí *Java Reflection API*.

Kapitola 6

Závěrečné zhodnocení

Vývoj projektu Animator byl pro mne cennou zkušeností. Poprvé jsem vytvářel dílo tak velkého rozsahu a po tak dlouhou dobu. Během jeho vývoje jsem se musel vypořádat s řadou problémů. Řešení některých spočívalo v drobných úpravách projektu, jiné si vyžadovaly rozsáhlejší zásah do jeho modelu. Poučil jsem se, že rozsáhlé projekty, jako je tento, je vhodné nejdříve důkladně promyslet a naplánovat do největších detailů a teprve potom se pustit do jejich implementace. V průběhu vývoje aplikace jsem se seznamoval s novými technikami programování. Detailněji jsem poznal jazyk Java a některá jeho úskalí, se kterými jsem byl nucen se vypořádat. V neposlední řadě pro mě byla nová a obohacující zkušenost konzultovat mé představy o projektu a stěžejní problémy, na které jsem při jeho vývoji narazil, s vedoucím této práce RNDr. Rudolfem Krylem.

Výsledkem mého snažení je práce, která by mohla zaujmout svým pojetím a možnostmi široké spektrum studentů. Mohla by pomoci studentům, kteří se snaží pochopit základy algoritmů vnějšího třídění, ale i studentům, jenž se věnují zkoumání a porovnávání těchto algoritmů již delší dobu. Tato práce nabízí řadu nástrojů, kterými je možné animovaný algoritmus podrobovat detailnímu zkoumání. Na některých nástrojích je možné dokumentovat rozdílné pojetí různých algoritmů, některé se spíše zaměřují na vystihnutí principů vnějšího třídění. Domnívám se, že těmito možnostmi a nástroji předčí ostatní aplikace stejného zaměření a proto věřím, že bude studenty využívána.