

Návrhové vzory

a jejich využití při výuce algoritmů a
datových struktur

Bakalářská práce

Thomas Wanli

Knihovna JU - PF



3 1 1 5 1 7 1 7 3 3

Vedoucí bakalářské práce
RNDr. Jaroslav Icha

Jihočeská univerzita
Pedagogická fakulta
Katedra informatiky

České Budějovice 2005

Poděkování

Děkuji mému vedoucímu práce za laskavé vedení a velmi užitečné rady, které mě dost pomohly při tvorbě této práce. Děkuji též všem učitelům, kteří se snažili nás něco naučit a připravit do praxe.

Anotace

Tato práce je o návrhových vzorech a jejich využití při implementaci datových struktur. Návrhový vzor je koncepce pro tvorbu kvalitního objektově orientovaného softwaru; jde o konkrétně popsaný postup určený pro řešení typického problému se kterým se můžeme potýkat při psaní programového kódu. Návrhových vzorů existují stovky, co problém to návrhový vzor.

Tato práce je konkrétně o využití těchto návrhových vzorů pro řešení problémů s návrhem datových struktur. Nebudu se tedy zabývat přímo použitím návrhových vzorů při tvorbě aplikací jako takových, ale tím jak je používat pro tvorbu datových struktur, které tvoří základní stavební kameny každé aplikace. Na druhé straně v tomto materiálu bude popsáno několik aplikací, které demonstrují konkrétní praktické použití té které datové struktury případně návrhového vzoru.

Nejdříve popíši o co jde v koncepci návrhových vzorů. Ukážeme si konkrétní aplikaci s využitím jednoho takového vzoru. Dále v kapitole *Návrhové vzory a datové struktury* popíšeme blíže, jak nám návrhové vzory mohou pomoci při návrhu a tvorbě datových struktur. Toto rozvedeme podrobněji na příkladu sedmi vzorů, které jsou obsaženy v kapitole *Katalog návrhových vzorů*.

Katalog návrhových vzorů tvoří podstatnou část mé práce. Slovo *katalog* zní neobvykle. Nejčastěji si ho spojujeme s představou přehledu výrobků určitého druhu např. stanů nebo jiného zboží, ale představa katalogu problémů při programování a jejich řešení je nezvyklá. Takové katalogy ovšem skutečně existují. Jde o to, že každý návrhový vzor má skutečně jméno jako nějaký výrobek. Dále obsahuje popis problému, se kterým je spojený jako výrobek, má svůj popis a přirozeně řešení tohoto problému. A kdo ho vyrobil? Počítačová odborníci, kteří zjistili, že některé problémy při návrhu programů se opakují s „železnou“ pravidelností. Proto je pojmenovali a napsali obecnou šablonu pro jejich řešení. Ale o tom blíže v následující kapitole.

Jako implementační jazyk budu používat Javu a to verzi 4.1. Java je výlučně objektově orientovaný jazyk a proto je pro popis návrhových vzorů ideální. Zdůrazňuji, že návrhové vzory jsou záležitostí objektově orientovaného programování, nikoli procedurálního programování.

Součástí práce jsou čtyři aplikace, které ukazují praktické uplatnění návrhových vzorů při implementaci datových struktur. Dále zdrojové kódy zjednodušené hierarchii datových struktur, které se používají pro implementace těchto aplikací. Na různých místech práce se na tyto aplikace odkazují. Všechny fragmenty kódu zde uvedené jsou vyňaty z těchto zdrojů. Aplikace a datové struktury najdete na příloženém CD. Popis obsahu CD najdete v příslušné příloze.

Text práce v elektronické podobě, zdrojáky aplikací a jejich spustitelné soubory můžete též najít na adrese:

<http://mujweb.cz/Pocitace/tomwanli/diplomka>

Obsah

ÚVOD

Co je návrhový vzor?	2
Pozorovatel	2
Zobrazení stavu čítače:	3
Návrhové vzory a datové struktury	10

KATALOG NÁVRHOVÝCH VZORŮ

OBJEKTOVÉ VZORY	12
1. Iterátor	12
Implementace iterátorů pomocí pole	12
Cvičení	16
2. Pozice	17
Hierarchie pozičních kontejnerů	19
Poziční iterátory	22
Cvičení	24
3. Lokátor	25
Lokátory v prioritní frontě	27
Aplikace Díář	28
Implementace prioritní fronty s využitím lokátorů	31
Cvičení	35
TŘÍDNÍ VZORY	36
4. Adaptér	36
Postfixový kalkulátor	38
Poziční iterátor jako adaptér	42
DequeStack	42
Cvičení	44
5. Komparátor	45
Porovnávání datumů v diáři	47
Cvičení	49
6. Dekorátor	50
Použití BVS uzlu podporujícího vzor Dekorátor v AVL stromech	52
Cvičení	54
7. Šablonová metoda	55
Kreslení binárních stromů pomocí Eulerova průchodu	58
Cvičení	64

ÚVOD

Co je návrhový vzor?

Jedním ze základních stavebních kamenů efektivního objektově orientovaného programování je snaha o tvorbu znovupoužitelného kódu; tj. kódu, který můžeme použít **znovu** v budoucích aplikacích a to bez jeho modifikace nebo alespoň s minimálními úpravami. Když nalezneme dobré řešení, tak ho musíme používat, jestliže to jde, opakovaně. Je totiž velkou chybou řešit každý problém od píky, musíme se snažit používat řešení, která v minulosti fungovala.

Zkušební návrháři se tím řídí. Jenže zašli ještě dál: přišli s nápadem tyto problémy pojmenovat a navrhnout obecné šablony pro jejich řešení. Tyto šablony mohou programátoři využít vždy, když narazí na určitý problém řešitelný danou šablonou. Jde o **vzory** návrhu řešení problémů.

Tento bod nám pomůže ilustrovat analogie. Autoři románů a divadelních her málokdy vymýšlejí zápletky úplně od začátku. Místo toho se řídí vzory, např. „hrdina s tragickou vadou“ (Macbeth, Hamlet atd.) nebo „romantický román“ (nespočet milostných románů). Stejně tak se objektově orientovaní návrháři řídí vzory, např. „obohatit třídy tak, aby se funkce snadno přidávaly a odstraňovaly“ a „oddělit grafické uživatelské rozhraní od výpočetní části programu“.

Návrhový vzor je tedy abstrakce návrhového problému i jeho řešení: každý návrhový vzor systematicky nazývá a vyhodnocuje důležitý a v objektově orientovaných systémech se opakující návrhový problém.

Vzor se skládá z těchto prvků:

1. **Název vzoru** vystihuje podstatu problému.
2. **Problém** popisuje, kdy se má vzor používat. Vysvětluje řešený problém.
3. **Řešení** popisuje prvky, z nichž se návrh skládá a jakým způsobem spolu komunikují.

Například jedním z běžných problémů v programátorské praxi je, jak oddělit výpočetní část programu od grafického uživatelského rozhraní. Ukážeme si, jak lze řešit tento problém pomocí konkrétního návrhového vzoru. Tím je...

Pozorovatel

Název vzoru: *Pozorovatel – Observer.*

Problém:

Jak jsem uvedl výše, jedním z častých problémů v programátorské praxi je, jak oddělit výpočetní část programu od grafického uživatelského rozhraní (část programu popisující jeho vzhled). „Oddělit“ výpočetní část od uživatelského rozhraní znamená, uložit výpočet do samostatné třídy či tříd, které jsou na způsobu zobrazení **naprosto nezávislé**. Třída, která zajistí zobrazení výsledků, pak jen vytvoří a použije objekty těchto „výpočtových“ tříd.

Tento problém řeší vzor **Pozorovatel**. A nejen to: pozorovatele můžeme použít i tam, kde chceme reprezentovat data výpočetní části několika grafickými formami (můžeme tatáž data reprezentovat tabulkou, koláčovým nebo sloupcovým grafem). Jak na to, popíšeme v následující části.

Řešení:

K dosažení nezávislosti výpočetní třídy na třídě grafického uživatelského rozhraní nebo obecně nezávislosti jakékoliv třídy na jiných třídách, potřebujeme definovat rozhraní, přes které budou tyto (vzájemně nezávislé) třídy komunikovat a způsob této komunikace.

Návrh vzoru se tedy bude skládat ze dvou klíčových rozhraní: rozhraní pozorovatele a rozhraní subjektu. Výpočetní třída je **subjektem** a grafické uživatelské rozhraní nebo grafická komponenta tohoto rozhraní **pozorovatelem** stavu objektu výpočetní třídy (subjektu). Co to znamená? Prakticky to znamená, že **kdykoli dojde ke změnám v datech výpočetní třídy, tak se to projeví graficky ve vzhledu grafické komponenty**: pozorovatel (grafická komponenta) je **upozorněn**, kdykoli subjekt (výpočetní třída) mění svůj stav, tj. svá data. Reakcí pozorovatele je, že synchronizuje svůj stav (vzhled) podle stavu subjektu. Stav grafické komponenty je závislý na stavu výpočetní třídy.

Když budeme chtít, aby výpočetní třída byla reprezentována několika grafickými formami, tak stačí prostě přidat tolik pozorovatelů, kolik bude třeba. Princip komunikace zůstane stejný: když tito pozorovatelé odebírají upozornění na změnu dat, tak synchronizují svůj stav podle stavu výpočetní třídy.

Tento způsob komunikace je také znám jako „vydávat – odebírat“. Subjekt je vydavatelem upozornění na změnu svých dat. Pozorovatel nebo pozorovatelé tato upozornění odebírají a podle jejich obsahu mění svůj vzhled.

Implementace rozhraní bude vypadat takto:

```
//Rozhraní výpočetní části programu.
public interface Subject {
    public void registerInterest(Observer obs);
    public void notifyObservers(Object arg);
}

//Rozhraní grafické komponenty
(pozorovatele)
public interface Observer {
    public void sendNotify(Object arg);
}
```

Metodou `registerInterest(Observer obs)` pozorovatel `obs` se přihlásí k odebrání upozornění na změnu dat: aby grafická komponenta byla schopna automaticky měnit svůj vzhled podle stavu výpočetní třídy, tak se musí předtím zaregistrovat jako pozorovatel této třídy – **grafická komponenta není automaticky pozorovatelem**. Po registraci subjekt zpravuje pozorovatele o změnách v datech, kdykoli k nim dojde metodou `notifyObservers(Object arg)`, přičemž `arg` je objektem změny. Pozorovatel pak mění svůj vzhled podle nových hodnot obsažených v `arg` metodou `sendNotify(Object arg)`. Tato metoda je volána uvnitř těla `notifyObservers(Object arg)`. Tato metoda jí předá parametr `arg`.

Jak uvidíme v následujícím programu, kód výpočetní třídy neobsahuje žádnou konkrétní instanci pozorovatele: volání metody, která registruje pozorovatele (`registerInterest()`) s parametrem reprezentujícím konkrétní instanci pozorovatele, se uskutečňuje v kódu třídy grafického uživatelského rozhraní. To umožňuje použití výpočetní třídy i v jiných aplikacích, které mají odlišné uživatelské rozhraní. **Výpočetní třída je znovupoužitelná.**

Zobrazení stavu čítače:

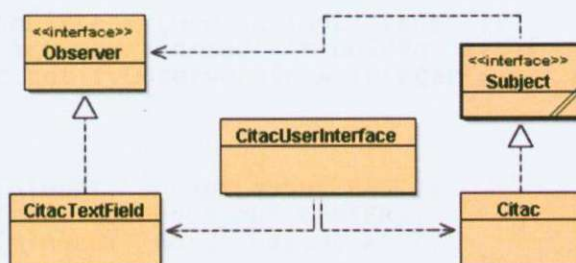
Napišeme program, na kterém si objasníme mechanismus práce vzoru popsany výše. Program zobrazí okno ukazující stav čítače, který inkrementuje či dekrementuje svoji hodnotu po stisku tlačítka „+1“ nebo „-1“ uživatelem:



Nejdříve si ukážeme, jak snadno lze pomocí vzoru Pozorovatel oddělit uživatelské rozhraní tohoto programu od výpočetní části, která se o inkrementaci resp. dekrementaci stará. Potom si ukážeme, jak lehce se díky tomuto vzoru můžeme postarat o reprezentaci stavu čítače více formami: kromě textfieldu to bude scrollbar.

Dříve než se do toho pustíme, musíme si udělat základní představu o tom, jaké třídy budou figurovat v našem programu: chceme-li oddělit výpočet od grafického rozhraní, musíme uložit výpočetní část do třídy *Citac* a GUI programu do třídy *CitacUserInterface*. *CitacUserInterface* bude využívat objekt třídy *Citac*.

Podle filozofie vzoru Pozorovatel *Citac* bude implementovat rozhraní subjektu, ale pozor: *CitacUserInterface* jako takové není pozorovatelem, ale obsahuje grafickou komponentu, která je tímto pozorovatelem – textfield případně scrollbar: po každé když dojde ke změně dat čítače, tak se zobrazí nová hodnota v **textfieldu** nebo se **změní poloha jezdce scrollbaru**. Proto by bylo dobré, navrhnout textfield i scrollbar jako samostatné třídy, které implementují rozhraní pozorovatele. Takže náš diagram tříd v případě jednoho pozorovatele by mohl vypadat takto:



Na třídách tohoto diagramu¹ si vysvětlíme, jak funguje Pozorovatel. Začneme implementací třídy *CitacUserInterface*:

```

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
 * Implementuje uživatelské rozhraní objektu Čítač, který má textové pole
 * ukazující stav čítače a dvě tlačítka "+1" a "-1".
 */

public class CitacUserInterface {
    /**
     * Hlavní program. Spouští uživatelské rozhraní.
     */
    public static void main(String[] args) {
        new CitacFrame();
    }
}

//Okno čítače
class CitacFrame extends JFrame {
    private Citac citac; //Objekt výpočtové třídy
    private CitacTextField ctf; //textové pole v němž se objevuje
                                //aktuální hodnota čítače

    private JButton plusBT, minusBT;

    public CitacFrame() {
        setTitle("Čítač");
    }
}

```

¹ Tento a následující diagramy v práci jsou vytvořené v programovém prostředí *BlueJ*. Obdelníky reprezentují třídy nebo rozhraní a šipky mezi nimi reprezentují vazby mezi třídami: přerušovaná šipka s trojúhelníčkem na konci říká, že třída, ze které šipka míří, implementuje rozhraní do nějž šipka vstupuje. Šipka bez trojúhelníčku říká, že třída, ze které šipka míří, používá třídu do níž šipka vstupuje. A konečně existuje ještě nepřerušovaná šipka s trojúhelníčkem, která znázorňuje dědičnost.

```

/**Od Javy verze 5.1, už nemusíme vytvářet kontejner do nějž bychom
    umístili grafické komponenty uživatelského rozhraní: tyto
    komponenty, už můžeme vkládat přímo do objektu typu Frame. */

Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout(10,0));

Citac citac = new Citac(0);
ctf = new CitacTextField("0");
plusBT = new JButton("+1");
minusBT = new JButton("-1");
citac.registerInterest(ctf);           //registrace pozorovatele
                                       textfield

plusBT.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        citac.notifyObservers(new Integer(citac.plusJedna()));
    }
});

minusBT.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        citac.notifyObservers(new Integer(citac.minusJedna()));
    }
});

contentPane.add(plusBT, BorderLayout.EAST);
contentPane.add(ctf, BorderLayout.CENTER);
contentPane.add(minusBT, BorderLayout.WEST);
pack();
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setResizable(false);
setVisible(true);
}

```

Jak vidíme třída `CitacFrame` skutečně používá objekt výpočetní třídy: obsahuje členskou proměnnou `citac` typu `Citac`. `CitacFrame` nejdříve volá metodu `citace.registerInterest()`, kterou se registruje pozorovatel, pak volá další metody `citace`, kdykoli uživatel stiskne tlačítko: když uživatel stiskne tlačítko „+1“ tak volá metodu `citace.plusJedna()` a když stiskne „-1“ tak volá metodu `citace.minusJedna()`.

Grafická komponenta neboli členská proměnná, která se zaregistruje jako pozorovatel stavu `citace` je `ctf` typu `CitacTextField`, jestliže dojde k inkrementaci resp. dekrementaci, změní tato komponenta svůj stav: objeví se nová hodnota v textovém poli. Jak vidíme tato registrace proběhne dříve, než se objeví okno programu na obrazovce příkazem `setVisible(true)`. Registrace `ctf` nebo jakéhokoliv jiného pozorovatele proběhne tak, že `citac` si ho přidá do svého seznamu pozorovatelů, kterým pak posílá upozornění na změnu svých dat. To vidíme v implementaci třídy objektu `citac`:

```

import java.util.*;
import observer.Subject;
import observer.Observer;
/**
 * Třída, která přičítá nebo odečítá k určitému číslu jedničku.
 * Implementuje rozhraní Subject, aby uměla komunikovat se svým případným
 * grafickým rozhráním.
 */
public class Citac implements Subject {
    private ArrayList observers;
    private int hodnota;
    public static final int MIN_HODNOTA = -5;
    public static final int MAX_HODNOTA = 5;

```

```

/**
 * konstruuje čítač s počáteční hodnotou a prázdným seznamem
 * pozorovatelů.
 * @param hodnota počáteční hodnota, která se má inkrementovat nebo
 * dekrementovat.
 */
public Citac(int hodnota) {
    observers = new ArrayList();
    setHodnota(hodnota);
}

/**
 * nastavuje hodnotu čítače na hodnotu uvedenou v parametru této
 * metody.
 * @param hodnota hodnota na kterou se má nastavit čítač.
 */
public void setHodnota(int hodnota) {
    this.hodnota = hodnota;
    if(this.hodnota < MIN_HODNOTA) //pokud hodnota čítače překročila
        minimální přípustnou hodnotu
        this.hodnota = MIN_HODNOTA; //nastav ji na přípustné minimum.
    if(this.hodnota > MAX_HODNOTA) //pokud hodnota čítače překročila
        maximální přípustnou hodnotu
        this.hodnota = MAX_HODNOTA; //nastav ji na přípustné maximum.
}

/**
 * vrací aktuální hodnotu čítače.
 * @return aktuální hodnotu čítače.
 */
public int getHodnota() {
    return this.hodnota;
}

/**
 * přičítá k aktuální hodnotě čítače jedničku.
 * @return nová hodnota čítače.
 */
public int plusJedna() {
    setHodnota(++this.hodnota);
    return this.hodnota;
}

/**
 * odečítá od aktuální hodnoty čítače jedničku.
 * @return nová hodnota čítače.
 */
public int minusJedna() {
    setHodnota(--this.hodnota);
    return this.hodnota;
}

/**
 * registruje pozorovatele výpočetní části programu.
 * @param obs pozorovatel, který má přijímat upozornění na změnu dat
 * subjektu.
 */
public void registerInterest(Observer obs) {
    observers.add(obs);
}

/**
 * zpravuje pozorovatele o změně dat.
 * @param arg objekt změny.
 */
public void notifyObservers(Object arg) {
    Iterator it = observers.iterator();
    while(it.hasNext()) {
        Observer obs = (Observer)it.next();
        obs.sendNotify(arg); //pozorovatel zaktualizuje svoje hodnoty.
    }
}
}

```

Vidíme, že registrování pozorovatelů proběhne vykonáním těla metody `registerInterest()`:

```
observers.add(obs);
```

Proměnná `observers` je seznam, do nějž se pozorovatelé přidávají. Je členskou proměnnou třídy `Citac` a při inicializaci objektu této třídy (v našem případě `Citac`) je prázdný.

Všimněte si, že kód třídy `Citac` skutečně neobsahuje žádnou konkrétní instanci pozorovatele: v kódu uvedeném výše není žádná zmínka o tom, že pozorovatelem objektu `Citac` je nějaké textové pole. Třída `Citac` se stará pouze o posílání změněných dat pozorovatelům, kteří jsou „skrytí“ v seznamu `observers`. **To nám umožní použít třídu `Citac` s jiným uživatelským rozhraním bez jakékoliv její změny. Je nezávislá.** O jaké konkrétní pozorovatele jde, to vidíme v kódu třídy uživatelského rozhraní `CitacUserInterface`.

Když uživatel stiskne tlačítko („-1“ nebo „+1“), tak se změní stav výpočetní třídy `Citac`: v reakci na tuto událost (stisknutí tlačítka) se vykoná metoda `actionPerformed`. Ta volá metodu `minusJedna()` nebo `plusJedna()` podle toho, jaké tlačítko uživatel stiskl. Vlastní nastavení čítače na novou hodnotu vykoná metoda `setHodnota()`. Nová hodnota se předá metodě `notifyObservers()` prostřednictvím objektu změny `arg`. Ta se volá přímo z těla `actionPerformed`. Metoda `notifyObservers()` zpraví svého pozorovatele (textfield `ctf`) o změně v datech. Vlastní synchronizaci stavu textfieldu podle stavu čítače provede metoda `sendNotify()`. To vidíme v implementaci třídy objektu `ctf`:

```
public class CitacTextField extends JTextField implements Observer {

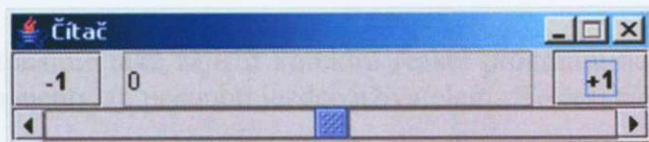
    public CitacTextField(String nula) {
        setColumns(18);
        setEditable(false);
        setText(nula);
    }
    //metoda, která aktualizuje stav textového pole.
    public void sendNotify(Object arg) {
        setText(arg.toString());
    }
}
```

Jak vidíme novou hodnotu zobrazí komponenta prostým přepsáním textu pole.

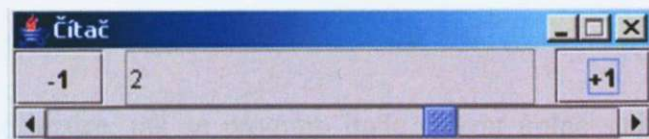
Takže obecně mechanismus komunikace výpočetní třídy s GUI programem na základě návrhového vzoru *Pozorovatel* probíhá takto: uživatel zadá požadavek. Uživatelské rozhraní volá metodu z výpočetní třídy, která dokáže požadavek zpracovat. Tato metoda po vykonání výpočtu předá výsledek nebo výsledky objektu `arg`, který náleží metodě `notifyObservers(Object arg)`. Ta upozorní pozorovatele na změnu dat a předá jim `arg` obsahující změněná data prostřednictvím metody `sendNotify(Object arg)`, která vzápětí zaktualizuje jejich vzhled podle nových dat. Ovšem aby tento mechanismus vůbec fungoval, musí výpočetní třída zaregistrovat pozorovatele už předtím než se GUI zobrazí.

Nyní si ukážeme jak přidat dalšího pozorovatele: budeme chtít reprezentovat stav čítače další grafickou komponentou. Tou bude scrollbar.

Jezdec scrollbaru se bude nacházet ve středu scrollbaru, při hodnotě čítače 0. Když uživatel stiskne tlačítko „+1“, tak jezdec se posune doprava, a když stiskne tlačítko „-1“, tak se jezdec posune doleva. Vzhled programu s textfieldem a scrollbarem a prezentací hodnoty čítače scrollbarem ilustruje následující obrázek:



Před kliknutím na tlačítko „+1“



Po kliknutí

Jediné, co se na kódu aplikace změní je, že tam přidáme novou třídu `CitacScrollbar` reprezentující nového pozorovatele scrollbar. Tu pak zakomponujeme do třídy uživatelského rozhraní `CitacUserInterface` prostým přidáním členské proměnné typu `CitacScrollbar` – `csb` a samozřejmě ji iniciujeme. Potom stačí `csb` zaregistrovat jako dalšího pozorovatele metodou `registerInterest()`. Kód registrování pozorovatelů bude vypadat takto:

```
citac.registerInterest(ctf); //CitacTextField pošle zprávu čítači, že chce
                             //pozorovat jeho data.
citac.registerInterest(csb); //CitacScrollbar pošle zprávu čítači, že chce
                             //pozorovat jeho data.
```

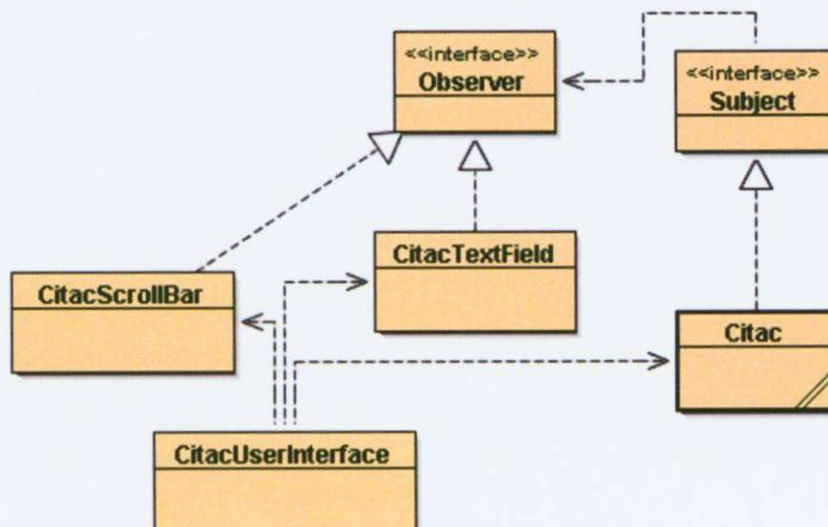
Kód nově přidané třídy vypadá takto:

```
import javax.swing.*;
import observer.Observer;
import observer.Subject;

public class CitacScrollbar extends JScrollbar implements Observer {
    public CitacScrollbar(int hodnota, int min, int max) {
        super(JScrollbar.HORIZONTAL, hodnota, 0, min, max);
    }

    //metoda, která aktualizuje stav jezdce posuvníku.
    public void sendNotify(Object arg) {
        int pozice = ((Integer)arg).intValue();
        setValue(pozice);
    }
}
```

Pro jasnější představu uvádím diagram tříd rozšířené aplikace:



Samozřejmě, že musíme také zajistit korektní reakci programu na událost, která se týká nově vytvořené komponenty, tj. posunutí jezdce uživatelem. To řeší následující kód:

```
csb.addAdjustmentListener(new AdjustmentListener(){
    public void adjustmentValueChanged(AdjustmentEvent evt){
        citac.setHodnota(evt.getValue());
        citac.notifyObservers(new Integer(citac.getHodnota()));
    }
});
```

Když uživatel posune jezdce, tak se program bude chovat úplně stejně, jako když uživatel klikne na tlačítka, tj. posunutím jezdce doprava se inkrementuje hodnota čítače a posunutím doleva se tato hodnota dekrementuje. Probíhá to opět na principu „vydávat – odebírat“. Proměnná `citac` odesílá upozornění o změně hodnoty a pozorovatel `textfield` je odebírá a synchronizuje svůj stav podle nové hodnoty.

Opět zdůrazňuji, že kód třídy `Citac` se nezmění. To mimo jiné znamená, že kód obsluhy jednoho pozorovatele (odeslání objektu změny pozorovateli) se neliší od kódu obsluhy více pozorovatelů. Třída `Citac` má seznam (`observers`), do kterého můžeme uložit jednoho nebo více pozorovatelů. Obsluhu pozorovatelů zajišťuje metoda `notifyObservers()`. Ta to dělá tak, že prochází všechny pozorovatele v seznamu `observers` prostřednictvím tzv. iterátoru a posílá jim objekt změny zavoláním metody `sendNotify()`:

```
/**
 * zpravuje pozorovatele o změně dat.
 * @param arg objekt změny.
 */
public void notifyObservers(Object arg) {
    Iterator it = observers.iterator();
    while(it.hasNext()) {
        Observer obs = (Observer)it.next();
        obs.sendNotify(arg);    //pozorovatel zaktualizuje svoje
                                //hodnoty.
    }
}
```

Iterátor je návrhový vzor, který popisuje, jakým způsobem zpracovat kolekci objektů. V tomto případě kolekci pozorovatelů, ale o tom až později.

Návrhové vzory a datové struktury

Při práci s datovými strukturami se můžeme potýkat s řadou nejrůznějších problémů. Jak urychlit vyhledávání v kontejneru a najít žádaný prvek bez procházení kontejnerem prvek po prvku? Jistě existují optimalizované datové struktury, které toto dokáží rychle, ale jejich nejlepší čas je $O(\log n)$ -např. AVL stromy. Ale jsou jiné struktury u kterých vyhledávání proběhne nejrychleji za $O(n)$, což je neefektivní.

Potřebujeme také vytvářet typově zaměřené kontejnery, které by uměly ukládat pouze objekty určitého typu a vyjímat je bez nutnosti přetypování na původní typ. Nevýhodou používání kontejnerů v jazyce *Java* je totiž ztráta typové informace po uložení objektu do kontejneru².

Jak naimplementovat datovou strukturu slovník (mapa) bez nutnosti vytváření zvláštního slovníku pro každý typ klíčů, na jejichž základě se porovnávají objekty ve slovníku, což by bylo neefektivní, protože by to přinášelo spoustu duplicitního kódu?

Tyto problémy a další nám pomohou vyřešit návrhové vzory: když potřebujeme rychle přistupovat k prvkům kontejneru, tak použijeme návrhový vzor **Lokátor**, který umožňuje přímý přístup k prvkům kontejneru, tj. v $O(1)$ čase a nezáleží na tom, jestli kontejner je spojový seznam nebo strom.

Pro parametrizaci kontejnerů použijeme vzor **Adaptér**, který tento problém řeší tak, že upraví (adaptuje) metody obecného kontejneru k použití pro implementaci metod parametrizovaného kontejneru. Pomocí návrhového vzoru **Komparátor** zase vyřešíme problém porovnání klíčů ve slovníku. Ten to řeší tak, že vytváří speciální rozhraní *Comparator*, který se bude starat o porovnávání klíčů. Jak toto vše funguje popíšeme v katalogu vzorů, který je na následujících stránkách.

Dříve než začneme, musím objasnit dvě věci:

Za prvé musím zdůraznit, že návrhové vzory se nepoužívají pouze pro řešení problémů tvorby datových struktur. Přesněji konkrétní návrhový vzor se může použít, jak na řešení konkrétního problému s datovou strukturou, tak i na problémy vzniklé při tvorbě aplikací. Například *adaptér* se nemusí použít jenom pro účel popsany výše. Vezmeme si například grafický editor (viz [2] strana 145), jenž umožňuje uživateli vytvářet a uspořádávat grafické prvky (čáry, mnohoúhelníky, text atd.) do obrázků a diagramů. Klíčovou abstrakcí grafického editoru je grafický objekt, který je schopen se sám vykreslit. Tuto abstrakci definuje rozhraní *Shape*, které implementují třídy pro každý druh grafického objektu: třída *LineShape* pro čáry, třída *PolygonShape* pro mnohoúhelníky atd.

Třídy *LineShape* a *PolygonShape* se poměrně snadno implementují avšak implementace třídy *TextShape*, která zobrazuje a edituje text, je daleko obtížnější. Přitom by již dostupná souprava nástrojů na uživatelská rozhraní mohla poskytnout důmyslnou třídu *TextView* k zobrazování a editaci textu. Jenže problém je v tom, že rozhraní třídy *TextView* neodpovídá rozhraní *Shape*, tudíž ji nemohu použít přímo. **Řešení spočívá právě v použití adaptéru, protože ten umí převádět rozhraní jedné třídy na rozhraní jiné třídy.** V tomto případě to znamená, že adaptér přizpůsobí metody třídy *TextView* pro implementaci metod třídy *TextShape*. **Tudíž povaha problému rozhoduje o tom, který návrhový vzor použít a ne povaha programu.**

V katalogu u každého vzoru se zaměříme na použití vzoru s určitou datovou strukturou.

² Od Javy verze 5.0 to již neplatí. Ta obsahuje parametrizované kontejnery.

Za druhé vzory se dělí na **objektové** a **třídní**. Objektové vzory se aplikují na objekty tříd jako takové a třídní vzory se aplikují na samotné třídy. Třídní vzory propojují třídy objektů různými vazbami s určitým účelem nebo rozšiřují možnosti existujících tříd.

Náš katalog tedy bude obsahovat dvě části. Krátký přehled vzorů v katalogu obsažených a jejich použití u datových struktur, jež uvádím níže:

A. Objektové vzory:

- Iterátor:** poskytuje možnost sekvenčního přístupu k objektům kontejneru, aniž by odkrýval jeho implementaci.
- Pozice:** umožňuje skrytí vnitřní implementaci kontejneru před uživatelem; uživatel například nemá vědět, zdal-li je kontejner implementovaný jednosměrným seznamem nebo obousměrným. Stejně tak uživatel nesmí měnit datovou strukturu kontejneru z jeho vnějšku.
- Lokátor:** uchovává referenci na pozici objektu v kontejneru. Díky tomu máme přímý přístup k tomuto objektu tj. nemusíme ho vyhledávat.

B. Třídní vzory:

Adaptér: umožní vytvářet typově zaměřené kontejnery, které by uměly ukládat pouze objekty určitého typu. Navíc umožní použití univerzálnějších kontejnerů k implementaci kontejnerů, které mají konkrétní účel. Například můžeme použít oboustrannou frontu k implementaci metod zásobníku.

Komparátor: poskytuje rozhraní Comparator, které obsahuje metody pro porovnávání objektů v uspořádaných kontejnerech. Vyhneme se tak nutnosti porovnání objektů v rámci implementace kontejneru, což by bylo neefektivní, protože bychom museli pro ten samý kontejner napsat několik implementací, které by se lišily pouze metodami pro porovnávání objektů kontejneru. Každý typ objektů má totiž svoje specifická pravidla (tudíž vlastní metody) pro porovnávání. Například jinak porovnáváme slova (abecedně) a jinak datумы. Místo toho pro každý typ objektů napíšeme vlastní porovnávací třídu, která implementuje Comparator a tu předáme jako argument konstruktoru kontejneru.

Dekorátor: využívá se pro změnu nebo přidávání datových prvků třídám. V tomto materiálu je uveden jako příklad uzel binárního vyhledávacího stromu, který díky dekorátoru může jednou sloužit jako uzel AVL stromu a jednou jako uzel červeno černého stromu.

Šablonová metoda: poskytuje třídu, která implementuje kostru algoritmu. Detaily implementace, které se mohou lišit u různých aplikací nechává na jejich podtřídě. Například spousta algoritmů je v podstatě procházení stromem, ale liší se akcí, která je vykonávána u jednotlivých uzlů těchto stromů.

KATALOG NÁVRHOVÝCH VZORŮ

Objektové vzory

1. Iterátor

Název vzoru: *Iterátor – Iterator.*

Problém:

Potřebujeme způsob, jak procházet prvky (objekty) kontejneru, samozřejmě za účelem vyhledávání a přitom se vyhnout tomu, aby se odkryly detaily implementace kontejneru.

Řešení:

Iterátor je objekt, který se skládá z lineární datové struktury (například sekvence nebo pole), která obsahuje prvky procházeného kontejneru, z aktuální pozice v této struktuře a současně obsahuje metodu, jak se dostat na další pozici a učinit ji aktuální. Iterátor vytvoříme pomocí metody kontejneru, takže iterátor neobsahuje nic jiného než kopii prvků v kontejneru a tím, že procházíme prvky mimo kontejner, tak nemáme možnost zjistit, jak je implementován. To chrání kontejnery před nepovoleným zásahem zvenčí.

Návrh vzoru se skládá z rozhraní *Iterator* a ze třídy, která reprezentuje konkrétní implementaci iterátoru. V následujícím příkladě si ukážeme implementaci iterátoru pomocí pole a jak lze ho využít k procházení prvků kontejnerů.

Implementace iterátorů pomocí pole

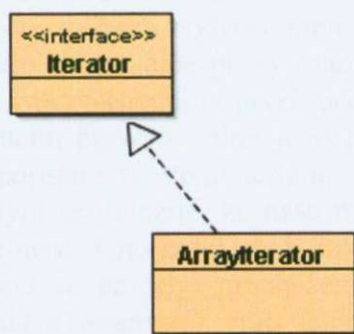


diagram tříd návrhového vzoru Iterátor

Jak vidíte vlevo budeme implementovat iterátor pomocí třídy *ArrayIterator*.

Rozhraní iterátoru obsahuje následující metody:

- *first()* – vrací první objekt uložený v iterátoru
Input: None Output: Object
- *next()* – vrací následující objekt uložený v iterátoru
Input: None Output: Object
- *current()* – vrací aktuální objekt iterátoru.
Input: None Output: Object
- *hasNext()* – testuje, jestli jsou v iterátoru ještě prvky k procházení.
Input: None Output: Boolean

Konstruktor *ArrayIterator*u má následující kód:

```
public ArrayIterator(Object[] pole) {
    polekProchazeni = new Object[pole.length];
    System.arraycopy(pole,0,polekProchazeni,0,pole.length);
    aktualniPozice = -1;
}
```

Kopie kontejneru se uloží do instanční proměnné *polekProchazeni*. Vlastní uložení kopie provedeme příkazem

```
system.arraycopy(pole,0,polekProchazeni,0,pole.length);
```

Instanční proměnná `aktualniPozice` uchovává aktuální pozici iterátoru. Při vzniku objektu iterátoru je záporná, protože jsme ještě nezačali s procházením prvků kontejneru (přesněji kopie kontejneru), tudíž jsme se nedostali ani na první prvek kontejneru.

Iterátor kontejneru vytvoříme metodou `elements()`. Tu má každý kontejner. Tato metoda vytváří lokální pole obsahující prvky kontejneru a předá ho jako argument nově vytvořenému iterátoru. Lokální pole je proměnná `Object[]` pole v kódu uvedeném výše. V následujícím příkladě se podíváme na konkrétní implementaci metody `elements()` kontejneru `List` (seznam). `List` je rozhraní pro implementaci jednosměrných nebo obousměrných seznamů³ a metoda `elements()` je jednou z metod třídy `LinkedList`, která implementuje toto rozhraní. Kód této metody vypadá takto:

```
public Iterator elements(){
    Object[] elementy = new Object[size()];
    DLNode node = (DLNode)first();
    Object e = node.getElement();
    for(int i=0; i!=size(); ++i){
        elementy[i] = e;
        if(i<size()-1){
            node = (DLNode) after(node);
            e = node.getElement();
        }
    }
    return new ArrayIterator(elementy);
}
```

Všimněte si posledního řádku metody: ten vytvoří novou instanci iterátoru, které předá pole s prvky kontejneru. Zároveň ji vrátí jako proměnnou typu rozhraní `Iterator`. Uživatel ji pak může použít k procházení prvků kontejneru a provádění určitých akcí s nimi. Zdůrazňuji, že iterátor můžeme vytvořit i jinak než pomocí pole: můžeme ho implementovat sekvencí nebo dokonce frontou, pokud nám záleží na pořadí vložených prvků. Tak místo instance `ArrayIterator` vrátíme instanci `SequenceIterator` nebo `QueueIterator`.

Na začátku vytvoří metoda pole objektů s prvky kontejnerů. Je to pole `elementy`. V cyklu naplňujeme prvky tohoto pole opakovaným voláním metody `after(Position p)`, která vrací následující prvek tak dlouho, až dojde na konec kontejneru. Zatím se nezabývejte matoucím přetypováním: jeho podstatu vysvětlíme na návrhovém vzoru **Pozice**, který bude dále popsán v tomto materiálu.

Nyní je důležité, že naše metoda `elements()` vytvoří pole `elementy` obsahující prvky kontejneru. Tuto kopii předá instanci třídy `ArrayIterator`, která ho uloží do vlastního pole a připraví se na jeho procházení. To jsme viděli v konstruktoru popsaném výše. Nakonec metoda `elements()` vrátí tento iterátor k dalšímu použití. Uživatel ho nyní může použít k vyhledávání v kontejneru. Jak na to, se podíváme dále.

Na stránce 9 jsme uvedli, jak zpracovat seznam pozorovatelů. Provedli jsme to právě přes iterátor. Nyní si tento kód položíme opět před sebe:

```
public void notifyObservers(Object arg) {
    Iterator it = observers.iterator();
    while(it.hasNext()) {
        Observer obs = (Observer)it.next();
        obs.sendNotify(arg);
    }
}
```

Proměnná `observers` je seznam, do kterého jsme ukládali objekty typu `Pozorovatel`. Jsou to grafické komponenty, které vždy přizpůsobují svůj vzhled (stav) aktuálnímu stavu výpočetní

³ podívejte se do balíku hierarchie, který obsahuje hierarchii rozhraní kontejnerů. Stejně tak i do balíku `list`, kde je konkrétní implementace obousměrného seznamu – třída `LinkedList`.

třídy. Vždy, když výpočetní třída mění svůj stav, tak si vyžádá iterátor k procházení pozorovatelů v seznamu a přes něho volá jejich metody `sendNotify(arg)`, které aktualizují vzhled komponenty (pozorovatele). Tady vidíme, že metoda vrací iterátor seznamu `observers iterator()`.

Tady jsem použil hotovou metodu a třídu z knihoven tříd Java⁴. Jde o třídu `ArrayList` jejíž instancí je `observers` a jejíž metodou je `iterator()`, která vrací požadovaný iterátor. Tato metoda funguje na stejném principu jako `elements()`, tj. také vytvoří iterátor a vrátí ho jako návratovou hodnotu. Je tu další kosmetická změna: tou je, že `iterator()` vrací iterátor, který je instancí rozhraní `java.util.Iterator`. Stručně řečeno metodu `notifyObservers(Object arg)` můžeme implementovat se stejným účinkem následovně:

```
public void notifyObservers(Object arg) {
    Iterator it = observers.elements();
    while(it.hasNext()) {
        Observer obs = (Observer)it.next();
        obs.sendNotify(arg);
    }
}
```

kde `observers`, by mohlo být instancí třídy `LinkedList` z našeho balíku `list`. Po vrácení iterátoru a jeho uložení do proměnné `it` můžeme začít procházet pozorovatele a volat jejich metody `sendNotify(arg)`. Po testování podmínky jejíž účel byl popsán na začátku této podkapitoly (viz popis rozhraní iterátorů), procházíme prvky kontejneru voláním metody `next()`. Ta při prvním volání vrátí první prvek iterátoru. Nyní uvedeme celý kód třídy `ArrayIterator` a objasníme si na něm, jak fungují metody iterátoru. Zdůrazňuji, že mluvím o implementaci `ArrayIteratoru`, nikoli `Iteratoru`, protože iterátor je pouze rozhraní, které obsahuje referenci na objekt `ArrayIterator` vrácený metodou `elements()`.

```
package iterator;
import java.util.*;

/**implementuje iterátor pomocí pole*/
public class ArrayIterator implements Iterator
{
    private Object[] poleKProchazeni;
    private int aktualniPozice;

    public ArrayIterator(Object[] pole){
        poleKProchazeni = new Object[pole.length];
        System.arraycopy(pole,0,poleKProchazeni,0,pole.length);
        aktualniPozice = -1;
    }

    public Object first() {
        aktualniPozice=0;
        return poleKProchazeni[aktualniPozice];
    }

    public Object next() throws NoSuchElementException{
        Object soucasny;
        if (hasNext()) {
            ++aktualniPozice;
            soucasny = poleKProchazeni[aktualniPozice];
        } else
            throw new NoSuchElementException("Iterátor se nachází
            za koncem kontejneru nebo kontejner je prázdný.");

        return soucasny;
    }
}
```

⁴ cílem práce je ukázat, jak se mohou návrhové vzory použít pro implementaci datových struktur, nebo chcete-li kontejnerů. Pro tento účel jsem se musel obejít bez použití hotových kontejnerů z knihovny tříd a použít vlastní. Proto se nenechte zmást tím, že například v balíku `list` je třída `LinkedList`, která se jmenuje stejně jako hotová třída `LinkedList`, která se nachází v knihovně tříd v balíku `java.util` a která implementuje stejnojmenné rozhraní `List`. Jde o třídy s totožnou funkcí s tím rozdílem, že jejich metody se jmenují jinak. Třída `LinkedList`, která je v balíku programů práce je zjednodušenou verzí té z knihovny tříd.

```

public boolean hasNext() {
    if ((aktualniPozice+1) != poleKProchazeni.length)
        return true;
    else
        return false;
}
public Object current() throws NoSuchElementException {
    if(aktualniPozice!= -1)
        return poleKProchazeni[aktualniPozice];
    else
        throw new NoSuchElementException("Iterátor se nachází
            před začátkem kontejneru nebo kontejner je
            prázdný.");
}
}

```

Řekli jsme, že po vytvoření objektu iterátoru, budeme mít pole s prvky kontejneru a aktuální pozici nastavenou na zápornou hodnotu. Po prvním volání `next()` dostaneme první prvek kontejneru, neboť proměnná `aktualniPozice` poskočí na první prvek pole (`poleKProchazeni[0]`). Stejný účinek by mělo volání metody `first()`, neboť by vrátila stejnou hodnotu (tj. `poleKProchazeni[0]`). Po každém volání `next()` poskočíme na další prvek pole a nastavíme ho jako aktuální hodnotu. Tu by nám vracela metoda `current()`.

Vidíme, že iterátor je poměrně jednoduchý mechanismus, jak získat přístup ke všem prvkům kontejneru tím, že je procházíme jeden po druhém. Je to návrhový vzor, který je často používaný v programech a je prostoupen napříč knihovnamí tříd *Java*. V programech, které jsou součástí této práce ho také používám na několika místech.

V následující kapitole věnované návrhovému vzoru **Pozice** se ještě podíváme na konkrétní typ iterátorů (z hlediska třídy objektů jež prochází), tím je `PositionIterator`. Je to iterátor pozic kontejneru neboli iterátor objektů typu `Position`. Co je to za typ a jak souvisí s návrhovým vzorem **Pozice** uvidíme dále.

Cvičení

1. Naimplementujte iterátor využitím spojového seznamu.
2. Napište metodu `remove()`, která by odstranila prvek z iterátoru po projetí tohoto prvku.

2. Pozice

Název vzoru: *Pozice – Position.*

Problém:

Potřebujeme mechanismus přístupu k **uzlům** (*nodes*) spojového seznamu nebo stromu nebo jiného kontejneru obsahujícímu uzly bez toho, abychom se vystavili riziku změny vnitřní struktury kontejneru z jeho vnějšku. To je možné díky tomu, že tyto kontejnery obsahují vesměs metody, které vrací jako výsledek operace uzel kontejneru nebo přijímají jako parametr uzel kontejneru. Zde jsou příklady:

- Obousměrný spojový seznam `LinkedList` má například metodu `remove(DLNode node)`, která, jak vidíme, přijímá jako argument uzel, jež se má odstranit i s elementem (*data*), který je v něm uložen. Jak vidíme uživatel kontejneru `LinkedList` má odkaz na uzel tohoto kontejneru (*node*), který se nachází mimo vlastní kontejner. Díky tomu uživatel může použít tento odkaz ke změně vnitřní struktury tak, že dejme tomu použije metodu uzlu `node setNext(DLNode newNext)` k jeho spojení s uzlem, který leží mimo náš kontejner. Tím dokáže náš kontejner „rozbít“. Jak vidíme z hlediska bezpečnosti, to není dobré.

- Stejně tak spojový binární strom `LinkedBinaryTree`⁵ má velmi důležitou metodu

```
public BTNode root();
```

která vrací kořen stromu nebo přesněji odkaz na jeho kořen. Uznejte, že by to byl problém, kdyby volající třída použila tento odkaz k tomu, aby manipulovala s odkazy na potomky stromu (*left* a *right*), což v nejhorším případě může způsobit smazání stromu z paměti.

Stejně tak nemáme zájem na tom, aby uživatel věděl, zda-li naše třída je implementována obousměrnou datovou strukturou či jednosměrnou⁶. To snadno pozná, když uvidí, že třída přijímá argument typu `Node` (*Single Node*), což je uzel jednosměrného seznamu nebo argument typu `DLNode` (*Double Node*), což je uzel obousměrného seznamu (viz [1] strana 194).

Řešení:

Řešením obou problémů je přetypovat vrácený uzel na typ, který by obsahoval pouze *data* uzlu tzn., že nebude obsahovat odkazy na další uzly v kontejneru a zároveň by neprozrazoval, o jakou datovou strukturu jde (jestli je to pole nebo spojová struktura, nebo jestli je to obousměrná nebo jednosměrná spojová struktura). Tento typ nám nabízí právě vzor **Pozice**.

Někdo by mohl namítnout, že existuje ještě jedno řešení: tím je použít k implementaci seznamů nebo stromů na místo spojové struktury pole. Ano, bylo by to možné, ale není to ideální řešení, protože tak přijdeme o výhodu, kterou nám nabízejí spojové struktury. Tou je **rychlost**. Zatímco kontejner se spojovou strukturou provede odstranění objektu kontejneru v čase $O(1)$ (stačí změnit odkazy předchozího a následujícího uzlu po odstraněném uzlu), kontejner implementovaný polem to samé provede v čase $O(n)$, neboť odstranění prvku by si vyžádalo posunutí všech prvků doleva na místo odstraněného. Podívejte se na obrázek **Mazání prvku pole** na následující stránce.

Typ, který nám poskytuje návrhový vzor **Pozice** se jmenuje `Position`. Je to abstrakce intuitivní představy o umístění neboli pozici prvku v kontejneru. Každý uzel má určitou pozici v kontejneru; tudíž implementuje rozhraní `Position`, které má jedinou metodu

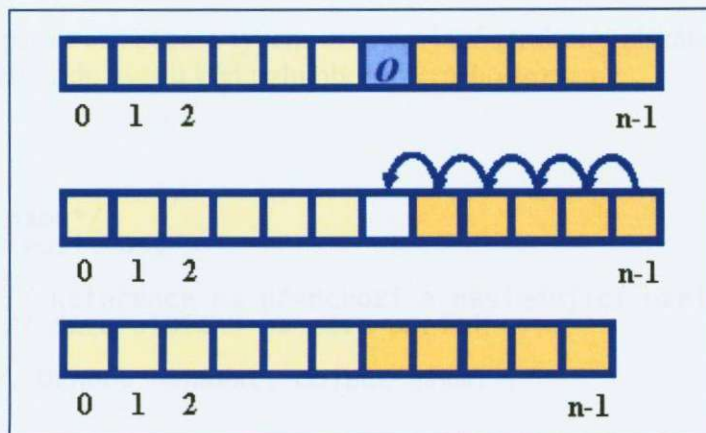
⁵ balík `trees`.

⁶ Třída `LinkedList` se nejmenuje takto zbytečně. Kdyby se jmenovala `DoublyLinkedList` tak by každý poznal, že je implementována obousměrným seznamem.

```
public Object getElement();
```

Tato metoda vrací element uložený na dané pozici v kontejneru, resp. v uzlu ležícím na této pozici.

Takže řešení prvního problému spočívá v tom, že vrátíme na místo instance typu `XXNode` instanci `Position`; **instanci `XXNode` přetypujeme na instanci rozhraní `Position`**: využíváme přitom faktu, že při přetypování instance třídy (`XXNode`) na instanci rozhraní (`Position`) je možné potom využívat z metod



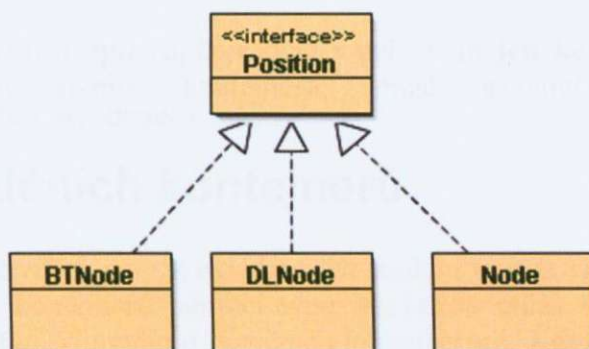
Mazání prvku pole

třídy pouze ty, které jsou deklarovány rozhraním. V našem případě je to metoda `getElement()`. Ta vrací data uložená v uzlu. Metody třídy `DLNode` `setNext(DLNode newNext)` a `setPrev(DLNode newPrev)` nebo metody třídy `BTNode` `setLeft(BTNode left)` a `setRight(BTNode right)` jsou z instance rozhraní nepřístupné, tudíž uživatel nemá možnost měnit odkazy na sousední uzly⁷. Tak uživatel získá z každé metody spojového kontejneru jen data aktuálního uzlu, tedy jen informace, které skutečně potřebuje. Použitím rozhraní `Position` dosáhneme efektu zapouzdření: my můžeme místo uzlů jako takových předat metodám, jako parametry, instance typu rozhraní `Position`. Například místo, aby metoda pro odstranění uzlu měla hlavičku `remove(DLNode node)`, tak bude mít hlavičku

```
remove(Position p)
```

Tím skryjeme detail implementace zda-li jde o obousměrný kontejner nebo jednosměrný.

Návrh vzoru Pozice se bude skládat z těchto tříd:



⁷ Pozornější čtenář by mohl namítnout, že to není pravda, protože uživatel třídy může rozhraní přetypovat zpátky na instanci uzlu a tudíž opět využívat metody uzlu. Je to logický postřeh, ale když se podíváte do dokumentace *Javy*, tak zjistíte, že nejsou přístupné informace, na kterou třídu se dá vrácená instance rozhraní přetypovat zpět. Například v dokumentaci k metodám `iterator()` se sice dozvíte, že vrací instanci typu `iterátor` (je to instance rozhraní!), ale už se nedozvíte, co to je za `iterátor`: jestli je to sekvenční `iterátor`, `iterátor` pole nebo jiný typ.

Jasnější představu o rozdílu mezi typem `Position` a typem uzlu spojové struktury získáte, když se podíváte na implementace uzlů. Zde uvedu kód uzlu obousměrného seznamu

```
package list;
import position.*;
import vyjimky.*;
/**třída uzlů obousměrných seznamů*/
public class DLNode implements Position{

    private DLNode prev, next; // Reference na předchozí a následující uzel
    private Object element;    // data uložená na této pozici
    // Konstruktor
    public DLNode(DLNode newPrev, DLNode newNext, Object elem) {
        prev = newPrev;
        next = newNext;
        element = elem;
    }
    // Metoda z rozhraní Position
    public Object getElement() throws InvalidPositionException {
        if ((prev == null) && (next == null))
            throw new InvalidPositionException("Pozice se nenachází v žádném seznamu.");
        return element;
    }
    //Přístupové metody
    public DLNode getNext() { return next; }
    public DLNode getPrev() { return prev; }
    //Aktualizační metody
    public void setNext(DLNode newNext) { next = newNext; }
    public void setPrev(DLNode newPrev) { prev = newPrev; }
    public void setElement(Object newElement) { element = newElement; }
}
```

Vidíme, že třída `DLNode` poskytuje sadu metod, které dokáží manipulovat s uzlem a se sousedními uzly. Právě proto, by se měla použít tato třída pouze pro vnitřní potřeby seznamu, kdy potřebujeme vkládat nebo odstranit uzly a přitom zachovat korektnost odkazů uvnitř seznamu. Když se podíváme na kód rozhraní, zjistíme, jak velké redukci informací přístupným uživateli došlo:

```
package position;
/**abstrakce pozice nebo umístění prvku v kontejneru*/
public interface Position{
    public Object getElement();
}
```

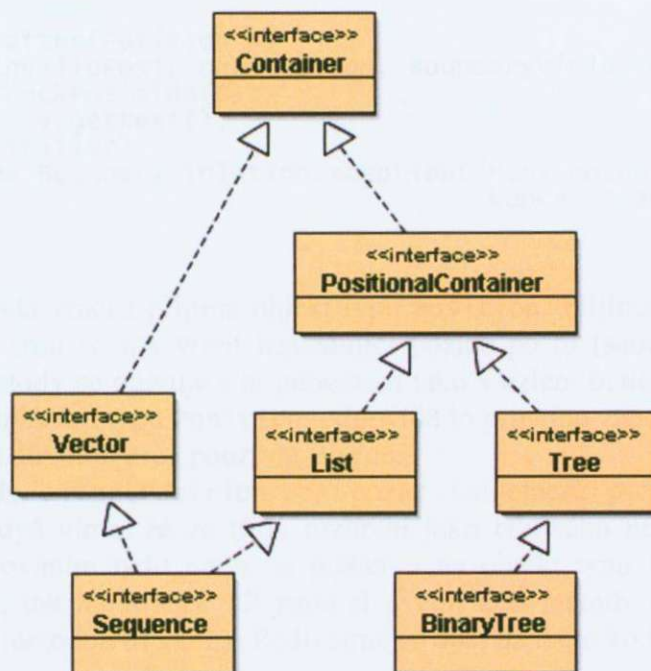
Jak jsme řekli, tak `Position` zpřístupňuje data v uzlu a **to jen ke čtení**. Pokud uživatel potřebuje měnit data uvnitř kontejneru, musí sáhnout k veřejné metodě `replaceElement(Position v, Object o)`.

Hierarchie pozičních kontejnerů

Všechny kontejnery používající uzly k ukládání dat mohou použít vzor `Pozice`. Díky tomu implementace spojových kontejnerů pomocí typu `Position` může být prostoupena napříč hierarchií datových struktur. Mluvíme o pozičních kontejnerech. Když se podíváte do balíku `hierarchie`, zjistíte, že podskupinou kontejnerů je skupina pozičních kontejnerů. Všechny tyto kontejnery dědí od rozhraní `PositionalContainer`, které deklaruje základní metody pro práci s pozicemi.

Kromě stromů a seznamů to jsou sekvence. Sekvence sice není čistě poziční kontejner, ale díky tomu, že dědí také od rozhraní `List`, tak používá metody pozičního kontejneru. Viz obrázek **Hierarchie pozičních kontejnerů** na následující stránce.

Základními metodami každého pozičního kontejneru jsou metody uvedené v následujícím fragmentu kódu:



Hierarchie pozičních kontejnerů: vidíme, že sekvence dědí metody od vektoru a seznamu. Zatímco vektor není poziční kontejner, tak seznam jím je. Díky tomu sekvence využívá jak metody pozičních kontejnerů, tak i metody vektoru.

```

package hierarchy;
import position.*;
import iterator.*;

/**metody pozičního kontejneru*/
public interface PositionalContainer extends Container{
    public PositionIterator positions();
    public void swapElements(Position v, Position w);
    public Object replaceElement(Position v, Object o);
}
  
```

Každý poziční kontejner potřebuje tedy metodu pro aktualizaci dat v uzlech. Tou metodou je `replaceElement(Position v, Object o)`, o které jsem se zmínil výše. Ta funguje tak, že vezme objekt `o` a nahradí jím objekt na pozici `v`. Metoda `swapElements(Position v, Position w)` přehodí data na různých dvou pozicích v pozičním kontejneru a nakonec metoda `positions()`, která vrací iterátor přes pozice. Posledně jmenovanou metodu probereme zvlášť v následující podkapitole o pozičních iterátorech.

Jak vidíme tyto metody jsou zapouzdřeny v rozhraní obecného pozičního kontejneru. Hierarchie kontejnerů se dále větví, protože rozlišujeme dva základní druhy pozičních kontejnerů: **seznamy** a **stromy**. Každá skupina implementuje metody deklarované výše jinak a potom obsahuje vlastní poziční metody, které druhá skupina nemá. Toto vyplývá z opravdu rozdílné povahy těchto kontejnerů. My se dále soustředíme na seznamy; protože jsou pro vysvětlování návrhového vzoru Pozice jednodušší; ale podstatu pozičních iterátorů si vysvětlíme na pozičním iterátoru přes uzly stromu.

V balíku `list` se nachází třída `LinkedList`, která implementuje rozhraní `List` uvedené výše pomocí obousměrného seznamu. Tato třída obsahuje sadu metod pracujících s kontejnerem typu seznam a využívajících typ `Position`. Podívejme se například na metodu `after(Position p)`, která vrací následující objekt kontejneru:

```

public Position after(Position p)
    throws InvalidPositionException, BoundaryViolationException {
    DLNode v = checkPosition(p);
    DLNode next = v.getNext();
    if (next == trailer)
        throw new BoundaryViolationException("Není možné pokročit za
                                            koncem seznamu.");
    return next;
}

```

Vidíme, že tato metoda vrací i přijímá objekt typu `Position`. Přijímá jako argument pozici kontejneru `p`, pro kterou se má vrátit následující pozice po ní (samozřejmě i s daty v ní obsažené). Uvnitř metody se operuje s argumentem jako s uzlem `DLNode` zatímco venku se s ním operuje jako s objektem typu `Position`: odpovídá to principu zapouzdření, což je, jak už víte, jeden z hlavních důvodů, proč používat `Pozice`.

Kde bere metoda `after(Position p)` pozici kontejneru `p`, pro kterou má najít následující pozici, když víme, že ze třídy rozhraní jako takového nelze vytvářet instance? Odpověď zní přetypováním uzlu `node` na pozici `p` na objekt typu `Position`. Když jsme mluvili o iterátorech, tak na stránce 12 jsme si uvedli kód metody seznamu `elements()`, která právě pracuje s metodou `after()`. Podívejme se opět na tento kód:

```

public Iterator elements(){
    Object[] elementy = new Object[size()];
    DLNode node = (DLNode)first();
    Object e = node.getElement();
    for(int i=0; i!=size(); ++i){
        elementy[i] = e;
        if(i<size()-1){
            node = (DLNode) after(node);
            e = node.getElement();
        }
    }
    return new ArrayIterator(elementy);
}

```

Řekli jsme, že tato metoda vrací iterátor prvků seznamu a proto prochází prvky seznamu v cyklu. Aby mohla vůbec získávat data na pozicích seznamu, tak musí tyto pozice nejdříve přetypovat na uzly a pak prostřednictvím metody uzlu `getElement()`⁸ získat data uzlu. Tu samou instanci uzlu pak použije jako argument `after(node)`, aby mohla přistoupit k následujícímu uzlu.

Mechanismus používaný výše (přetypování uzlu na pozici a zpět) se používá pouze uvnitř metod **pozičního** kontejneru. Uživatel pozičního kontejneru nemá možnost tento mechanismus používat. Když bude chtít dodat nějaké metodě tohoto kontejneru argument typu `Position`, tak ho získá jedině použitím metody, která vrací hotový objekt `Position`⁹. Podívejte se na následující fragment kódu:

```

public Object removeFirst() throws DequeEmptyException{
    try {
        return L.remove(L.first());
    }catch(ContainerEmptyException err) {
        throw new DequeEmptyException("Oboustranná fronta je rázdná.");
    }
}

```

Toto je jedna z metod oboustranné fronty¹⁰, která používá metodu `remove(Position p)` z rozhraní `List` (`L` je jeho instanční proměnná)¹¹. Tato metoda má odstranit první prvek fronty,

⁸ Podívejte se na kód třídy `DLNode` na stránce 17.

⁹ Netřeba dodat, že ten získáme jako návratovou hodnotu metody typu uzel přetypovanou na pozici. Podívejte se znovu na metodu `after(Position p)`.

¹⁰ Viz balík `queues`, třída `LinkedDeque`.

¹¹ Nenechte se zmást velikostí písmena. Není to třída, je to opravdu referenční proměnná.

proto předá `remove()` první pozici seznamu. Tu fronta získá **jedině** z metody `Position first()` seznamu.

Tak máte představu o pozičních kontejnerech, jejich hierarchii, použití jejich metod z pohledu návrhového vzoru Pozice a to jak v rámci pozičního kontejneru tak i mimo něj. Pro rozšíření vědomosti podívejte na implementace tříd prezentující poziční kontejnery.

Poziční iterátory

V minulé kapitole jsem se zmínil o pozičních iterátorech. Zmínil jsem se také o tom, že si je blíže vysvětlíme a tak učiníme:

Poziční iterátor, jak z názvu vyplývá, je iterátor přes pozice kontejneru. Přesněji řečeno je to iterátor objektů typu `Position`. Proto má speciálnější implementaci v tom smyslu, že bude uchovávat pole objektů typu `Position` (uvažujeme opět o implementaci iterátoru pomocí pole) a bude procházet toto pole speciální metodou `nextPosition()`, která vrací objekt typu `Position`, tzn. že nemusíme ho přetypovávat¹². Implementace pozičního iterátoru vypadá takto:

```
public class ArrayPositionIterator implements PositionIterator{
    private ArrayIterator iterator;

    public ArrayPositionIterator(Position[] pozice) {
        iterator= new ArrayIterator(pozice);
    }

    public Position firstPosition() {
        return (Position) iterator.first();
    }

    public Position nextPosition() throws InvalidPositionException{
        try {
            return (Position) iterator.next();
        } catch (NoSuchElementException e){
            throw new InvalidPositionException("Iterátor se nachází za
                                                poslední pozicí.");
        }
    }

    public boolean hasNext() {
        return iterator.hasNext();
    }

    public Position currentPosition() {
        try {
            return (Position) iterator.current();
        } catch (NoSuchElementException e){
            throw new InvalidPositionException("Iterátor se nachází před
                                                první pozicí.");
        }
    }
}
```

Vidíme, že tato třída využívá již hotovou třídu `ArrayIterator`. Je to naprosto korektní používat již hotové metody k implementaci jiných metod. Tak pracuje návrhový vzor **Adaptér**, jak uvidíme později.

Každý poziční kontejner musí mít kromě metody `elements()`¹³ (která vrací obecný

¹² Obecný iterátor má tu nevýhodu, že po uložení objektu do něj dojde ke ztrátě typové informace. Tzn., že po vyjmutí objektu ho musíme přetypovat na původní typ.

¹³ Nezapomínejte, že poziční kontejner je kontejner, tj. rozhraní `PositionalContainer` dědí `Container`, tudíž dědí také jeho metodu `elements()`. Viz balík hierarchy.

iterátor) také metodu `positions()`. Ta vrací poziční iterátor přes pozice kontejneru. U stromů tyto metody vrací rozdílné výsledky: zatímco metoda `elements()` vrací iterátor dat stromu, tak `positions()` vrací iterátor pozic. To znamená, že iterátor pozic bude obsahovat jak externí uzly, které žádná data nemají¹⁴, tak i interní uzly. Právě oni uchovávají data stromu. Metoda `elements()` uloží do iterátoru pouze data uložená v interních uzlech.

Metoda `positions()` binárního stromu¹⁵ má následující kód:

```
public PositionIterator positions() {
    LinkedList listOfPositions = new LinkedList();
    inorderPositions(root(), listOfPositions);
    Position[] positions = toPositionArray(listOfPositions);
    return new ArrayPositionIterator(positions);
}
```

Metoda `inorderPositions(Position p, List listOfPositions)` ukládá do seznamu `listOfPositions` **všechny** pozice stromu tím, že prochází strom inorder průchodem počínaje kořenem stromu. Metoda `toPositionArray(List list)` převede tento seznam na pole pozic, které pak předá jako argument konstruktoru pozičního iterátoru. Jde o stejný mechanismus tvorby iterátoru jako v metodě `elements()`.

Podobně jako u obecného iterátoru si ukážeme příklad, jak využít poziční iterátor k procházení prvků stromu a provedení akce s nimi. Půjde konkrétně o uložení **dat** stromu (tj. elementy uložené v interních uzlech stromu) do textového souboru (budeme předpokládat, že jsou to textová data). Zde je kód příkladu:

```
if(!jmena.isEmpty()){
    try{
        FileWriter vytvorSoubor = new FileWriter("data.txt");
        PrintWriter formOut = new PrintWriter( vytvorSoubor, false);
        PositionIterator posIt = jmena.positions();
        while(posIt.hasNext()){
            Position pos = posIt.nextPosition();
            Object data = pos.getElement();
            if(data!=null){
                String jmeno = (String) data;
                formOut.println(jmeno);
            }
        }
        formOut.close();
    }catch(IOException ioe){
        System.out.println("Neznámá chyba.");
    }
}
```

Půjde o binární strom `jmena` uchovávající jména lidí. Po vytvoření objektů potřebných k práci s textovým souborem zavoláme metodu `positions()`, která vrátí poziční iterátor `posIt`. Pozice v něm obsažené procházíme metodou `nextPosition()` a ověřujeme, zda se na nich nachází data, tj. jestli uzel na dané pozici je externí nebo interní. Pokud proměnná data se bude rovnat `null`, tak to znamená, že daný uzel je externí; v opačném případě se data (jméno) uloží do textového souboru. Po projetí všech pozic se soubor uzavře.

Toto byla náročná kapitola. Její náročnost vidím v rozličnosti datových struktur pracujících s pozicemi a ve velkých rozdílech ve způsobu práce s pozicemi mezi jednotlivými druhy pozičních struktur, tj. stromy a seznamy.

¹⁴ Budeme tedy předpokládat, že externí uzly jakéhokoliv stromu nebudou obsahovat data. Z tohoto předpokladu vychází implementace stromových struktur v balíku `trees`.

¹⁵ Viz balík `trees`, třída `LinkedBinaryTree`.

Cvičení

1. Prostudujte využití návrhového vzoru Pozice při implementaci stromů.
2. Napište jednoduchý textový editor, který ukládá řetězec znaků použitím spojového seznamu. Písmeno, pod kterým bude blikat kurzor bude vždy zvýrazněné. Editor by měl podporovat následující operace a zaktualizovat text (včetně zvýrazněného písmena), kdykoli se volá jedna z nich:
 - `left`: pohne kurzorem doleva o jeden znak (nebo nedělá nic, když kurzor je na konci textu).
 - `right`: pohne kurzorem doprava o jeden znak (nebo nedělá nic, když kurzor je na konci textu).
 - `delete`: smaže znak napravo od kurzoru (nebo nedělá nic, když kurzor je na konci textu).
 - `insert c`: vloží znak `c` po kurzoru.

3.Lokátor

Název vzoru: Lokátor –Locator.

Problém:

Návrhový vzor Pozice nám kromě jiného umožňoval přistupovat, odstraňovat a vkládat data do pozičního kontejneru v konstantním čase ($O(1)$). Jsou však situace, kdy i s využitím vzoru Pozice to není možné. Předpokládejme, že jsme vložili do kontejneru prvek **e**. Pak jsme modifikovali kontejner sérií operací `swapElements(p,v)`, která vyjme prvek **e** z pozice **p** a vloží ho na pozici **v** a naopak; tj. prvek na pozici **v** je vložen do uzlu na pozici **p**. Když nyní zavoláme metodu `remove(p)` s účelem odstranění prvku **e** z pozice **p**, tak neuspějeme, protože na pozici **p** je už jiný prvek než **e**. Jak tedy máme zachovat přímý přístup k elementům pozičního kontejneru i přes změnu pozic elementů?

Stejně tak existují kontejnery, které nepodporují mechanismus pozic (nejsou spojové, tj. nemají uzly), ale přesto mění pořadí prvků po vložení nového prvku a přitom bychom též potřebovali mechanismus umožňující přímý přístup k prvkům těchto kontejnerů. Jak jsme řekli, jsou to kontejnery, které nepodporují uzly. Jde například o **prioritní fronty a slovníky**, které neobsahují ani uzly, ani místa v poli, ale tzv. **položky (items)**, které se porovnávají na základě klíčů (*keys*) v nich obsažené. Jsou to tedy uspořádané kontejnery, které se hodí například k implementaci jazykového slovníku, protože ten obsahuje položky, kterými jsou dvojice klíče (slovo jehož význam hledáme) a elementu neboli *element* (význam slova). Jak tedy přistupovat k těmto položkám přímo? Odpovědí na tyto otázky je návrhový vzor Lokátor.

Řešení:

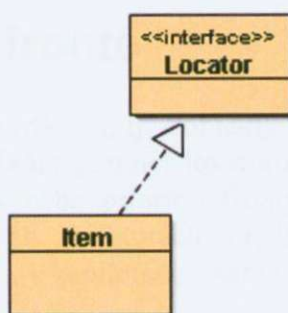
Potřebujeme abstrakci, která nám popíše **umístění (location)** prvku v kontejneru. Tato abstrakce musí zachovat odkaz na prvek, ať je umístěný na jakékoliv pozici nebo místě v kontejneru. Jedině tak budeme mít záruku přímého přístupu k prvků. Tuto abstrakci nám nabízí návrhový vzor Lokátor.

Lokátor je mechanismus, který udržuje vazbu mezi daty a jejich aktuální pozicí v kontejneru. Lokátor je „připíchnutý“ ke konkrétnímu elementu (datům) i když ten mění svou pozici uvnitř kontejneru. Lokátor je jako lístek od šatny (viz [1] strana 320); když dáme náš kabát šatnářce k úschově, šatnářka nám dá lístek. Tento lístek pak vrátíme šatnářce a ona podle něj pozná, **kde** je náš kabát. Tento lístek je tedy **lokátorem** našeho kabátu. Pozice našeho kabátu vzhledem k ostatním kabátům se může měnit, když šatnářka dává do šatny další kabáty nebo je vrací odcházejícím lidem, ale náš lístek nám pořád zaručuje, že náš kabát dostaneme zpátky. Je důležité si zapamatovat, že lokátor následuje svůj element i když ten mění svoje místo nebo pozici.

Nyní si můžeme představovat, že když vkládáme element do kontejneru, dostaneme zpátky něco jako lístek od kabátu uloženého v šatně – dostaneme zpátky lokátor elementu. Tento lokátor můžeme později použít k vyzvednutí elementu z kontejneru nebo provedení jiné operace, protože na něj uchovává odkaz. Můžeme tedy rozšířit repertoár operací pozičního kontejneru, jako je sekvence nebo binární strom, o metody, které podporují vzor Lokátor. Například můžeme definovat metody vkládání prvků kontejneru, které vrací lokátor prvků, a operace odstraňování prvků, které mají jako argument lokátor odstraněného prvku.

Jak jsme řekli, výše jsou kontejnery, které nepracují s uzly. I u nich můžeme použít lokátory, které by odkazovaly na jednotlivé **položky**. Vskutku, návrhový vzor lokátor nabízí mechanismus pro přístup k elementům, nebo dvojicím klíč–element, uložené v kontejneru způsobem, který je nezávislý na konkrétní implementaci kontejneru (zda je to poziční kontejner nebo kontejner pracující s klíči).

Návrhový vzor Lokátor se bude skládat z těchto tříd:



kde Locator je odkaz (lístek od kabátu) na položku Item v kontejneru a Item je položka kontejneru, na kterou odkazuje objekt typu Locator. Přes instanci rozhraní Locator je možné získat informace o klíči a jeho hodnotě (elementu) obsažených v položce kontejneru, na kterou odkazuje stejně jako lístek ke kabátu má na sobě číslo, které je stejné jako číslo na kabátě v šatně. To je vidět z kódu rozhraní Locator:

```

package locator;

/**Rozhraní návrhového vzoru Lokátor*/
public interface Locator{
    public Object key();
    public Object element();
}
    
```

Třída Item implementuje rozhraní Locator, právě proto, aby proměnná typu Locator mohla odkazovat na odpovídající položku v kontejneru. Funguje to tak, že po vložení položky (item) do kontejneru se vrátí odkaz na vloženou položku přetypovaný na instanci typu Locator. Tím získáme odkaz na položku, který je typu Locator. Funguje to stejně jako u návrhového vzoru **Pozice**, kde vracíme uzel jako pozici: tady vracíme položku jako lokátor. Podívejme se na kód třídy Item:

```

package queues;
import position.*;
import locator.*;

public class Item implements Locator {
    private Object key, elem;
    private Position pos;
    public Item (Object k, Object e, Position p) {
        key = k;
        elem = e;
        pos = p;
    }
    public Position position() {return pos; }
    public void setPosition(Position p) { pos = p; }
    public Object key() { return key; }
    public Object element() { return elem; }
    public void setKey(Object k) { key = k; }
    public void setElement(Object e) { elem = e; }
}
    
```

Vidíme, že kromě informací o klíči a jeho hodnotě v položce, objekt třídy Item obsahuje informaci o jeho pozici v kontejneru. Tuto informaci musí aktualizovat vždy, když se mění jeho pozice a to přes metodu setPosition (Position p). Tak lokátor neboli proměnná typu Locator uchová odkaz na položku, ať je kdekoliv v kontejneru. A to je právě náš cíl. To je to co potřebujeme, abychom přistupovali k prvku kontejneru v konstantním čase.

V následujícím příkladě uvádím první program mé práce pracující s datovými strukturami, tady konkrétně program, který přestupuje pomocí lokátorů k datům uložených v prioritní frontě.

Lokátory v prioritní frontě

Je řada kontejnerů pracujících s položkami, tj. s objekty typu `Items`. Všechny tyto kontejnery mohou přistupovat ke svým položkám pomocí lokátorů, pokud obsahují metody, které toto podporují, ať se jedná o slovníky nebo prioritní fronty. Vybral jsem prioritní frontu jako příklad použití lokátorů v datových strukturách z jednoduchého důvodu: o slovnících a návrhových vzorech pojednáme v kapitolách zabývajících se vzory **Komparátor** a **Dekorátor**.

Zdůrazňuji, že návrhový vzor není omezený na používání jen s jednou konkrétní datovou strukturou. Viděli jsme, že iterátory se používají ve všech kontejnerech, pozice v podskupině kontejnerů—tzv. poziční kontejnery. Na druhé straně jsou vzory, které se hodí jen pro některé typy kontejnerů: například lokátor se nejpřirozeněji používá právě u prioritních front. Komparátory se zase hodí v kontejnerech používající klíče, takže bude o něm zmínka už tady.

Prioritní fronta je fronta, která na rozdíl od obyčejné fronty nevkládá prvky do sebe v pořadí jejich příchodu (jako fronta před pokladnou), ale podle priority. To znamená, že položka s největší prioritou je ve frontě první (odebere se jako první) a položka s nejnižší prioritou se zařadí na konec fronty. Prioritu položky ve frontě určuje její klíč. Platí, že klíč s nejmenší hodnotou má nejvyšší prioritu. Ze fronty se odebírají prvky z její přední části, ale uvidíme, že existují praktické důvody, proč odebírat prvky ze fronty kdekoliv.

Kdybychom nepoužili lokátory, tak vyhledání položky v prioritní frontě by v průměru zabíralo čas $O(n)$. Díky nim to můžeme provést v konstantním čase, což je rapidní zrychlení. Jak na to?

Řekli jsme, že to provedeme tak, že pro každý vložený prvek získáme jeho lokátor a ten pak použijeme k další operaci s ním. Jenže logicky ho musíme někam uložit než ho znova použijeme, ale kam? Odpověď zní: záleží na konkrétní aplikaci, která používá lokátory k přístupu k nějaké datové struktuře v ní obsažené. My si ukážeme aplikaci, kterou je diář sloužící k zaznamenání termínů úkolů neboli povinností. Tento diář bude ukládat úkoly a jejich termíny do prioritní fronty, přičemž nejvyšší prioritu bude mít nejbližší termín. Nejprve si ukážeme, co umí prioritní fronta používající lokátory:

- `insert(Object k, Object e)`—vloží do fronty položku s klíčem `k` a hodnotou `e`. Vloží ho v pořadí jeho priority. Vrací lokátor této položky.
Input: `Object(2x)` Output: `Locator`
- `removeMin()`—odstraní ze fronty položku s nejvyšší prioritou.
Input: `None` Output: `Object`
- `min()`—vrací lokátor na položku s nejmenším klíčem (nejvyšší prioritou). Neodstraňuje ho.
Input: `None` Output: `Locator`
- `remove(Locator loc)`—odstraní jakýkoliv objekt fronty, pokud dodáme odkaz na něj.
Input: `Locator` Output: `None`
- `replaceKey(Locator loc, Object newKey)`—vymění klíč na který odkazuje lokátor `loc`, za klíč `newKey`. Nový klíč správně zařadí do fronty a do `loc` uloží odkaz na jeho pozici. Starý klíč je odstraněn.
Input: `Locator and Object` Output: `Object`
- `replaceElement (Locator loc, Object newElement)`—hodnota na kterou odkazuje lokátor `loc` je nahrazena hodnotou `newElement`. `loc` odkazuje i po konci operaci na stejnou pozici ve frontě.

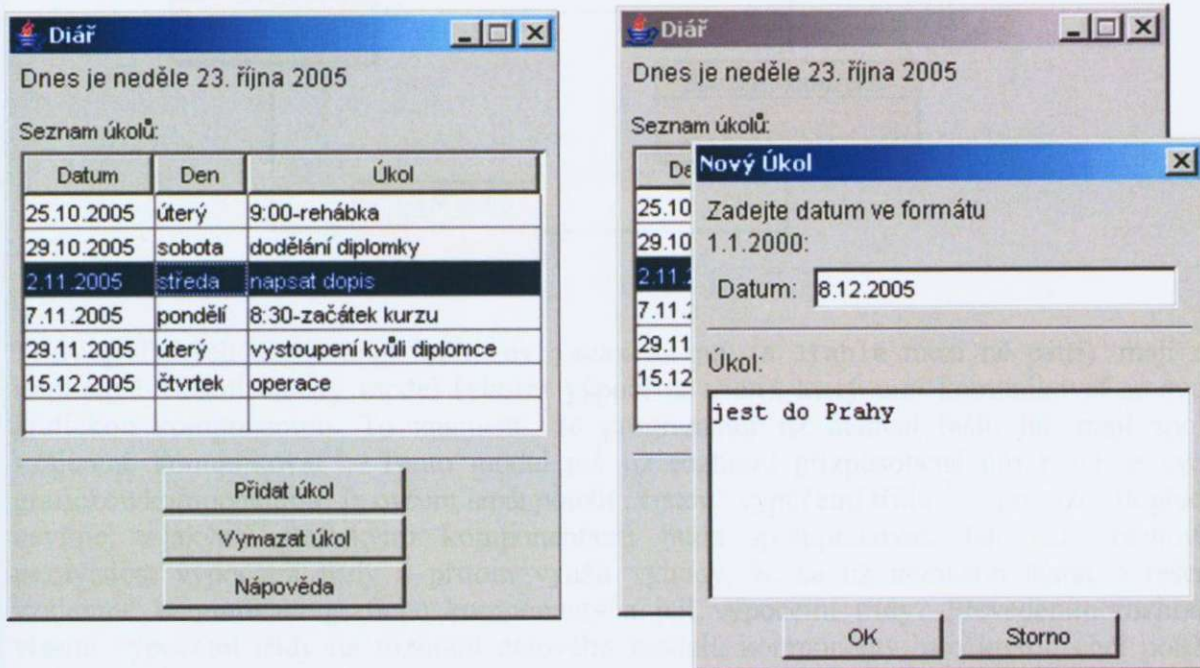
Tato definice dovedností prioritní fronty vybočuje z klasického pojetí prioritní fronty. V klasické frontě můžeme odebírat položky jenom z přední části, ale tady mluvíme o metodě `remove(Locator loc)`, která umí odstranit **jakýkoliv** objekt fronty. Nahoře jsme se zmínili o určitých praktických důvodech, proč tomu tak je. Nyní si tyto důvody rozebereme:

Může se stát, že z fronty odeberme nejen úkoly s nejvyšší prioritou, které jsme už splnili, ale i úkoly, které prostě už není třeba plnit. Takové úkoly můžeme mít samozřejmě naplánované na kdykoliv.

Stejně tak si můžeme představit aplikaci pracující s frontou cestujících na letišti, někteří z nich se rozhodnou odejít před přistávací dobou, proto na ně použijeme metodu `remove()`. (viz [1] strana 322).

Aplikace Diář

Aplikace umí ukládat úkoly, které je třeba splnit. Řadí je podle datumů termínů splnění. Aplikace umí tyto úkoly mazat, měnit datumy plnění (priority plnění) a počítá s tím, že uživatel může změnit názor a může chtít v konkrétní den vyřídit jiné věci než které měl původně naplánované. Funkce aplikace tedy kopírují všechny podstatné dovednosti prioritní fronty. Uživatelské rozhraní aplikace vypadá takto:



Vlevo: hlavní okno aplikace

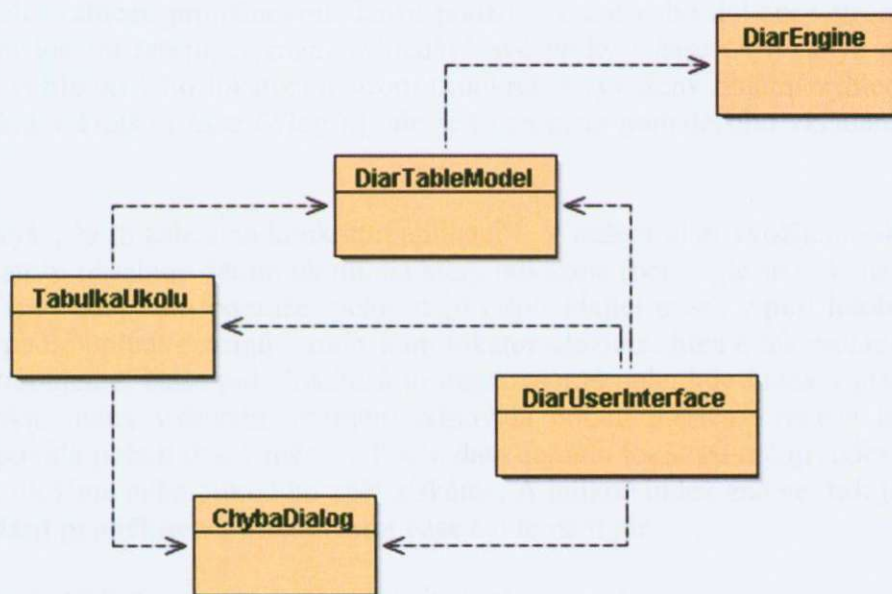
vpravo: dialogové okno pro přidávání úkolů

Data aplikace jsou uložena v prioritní frontě. Klíče fronty jsou datumy a hodnoty jsou úkoly. Úkol s nejbližším termínem má největší prioritu. Protože pro prezentaci dat používáme grafickou komponentu typu `Jtable` (tabulka), uložíme kopie dat fronty do dvourozměrného pole, kde úkol s největší prioritou je uložen na pozici pole s indexem 0.

Práci s prioritní frontou má na starost výpočetní třída aplikace `DiarEngine`, jejíž vazbu na ostatní komponenty (třídy) aplikace znázorňuje diagram tříd na následující stránce.

Na popis této třídy se soustředíme, protože ta přímo souvisí s naším tématem, ale abychom nebyli vytrženi z kontextu, hned si něco povíme o všech třídách:

- **DiarEngine**: nejdůležitější třída aplikace. Je to její výpočetní třída. Ukládá data do paměti (prioritní fronty) a zpracovává je.
- **DiarUserInterface**: reprezentuje grafické uživatelské rozhraní aplikace a zpracovává události programu.
- **TabulkaUkolu**: reprezentuje tabulku programu. Dědí třídu `JTable`. Tabulka je složitá komponenta z hlediska implementace, proto si zaslouží samostatnou třídu.
- **ChybaDialog**: pomocná třída pro zobrazení chybových hlášení programu.
- **DiarTableModel**: druhá výpočetní třída programu. Právem se ptáte, na co je třeba druhá třída, když už tu máme třídu **DiarEngine**. Vysvětlím to v následujícím odstavci.



Třídy grafických komponent knihovny `javax.swing` (a `JTable` mezi ně patří) mají už implicitně vlastní datový model (vlastní výpočetní třídu), který umí komunikovat se svou grafickou komponentou. To znamená, že programátor už nemusí řešit, jak mají spolu vzájemně komunikovat¹⁶. Tento model má již rozhraní přizpůsobené pro práci se svou grafickou komponentou. Je ovšem lepší použít „čistou“ výpočetní třídu¹⁷, protože dopředu nevíme, s jakými grafickými komponentami bude spolupracovat. Jak tedy zachovat nezávislost výpočetní třídy a přitom využít výhody, že se už nemusím starat o řešení vzájemné komunikaci grafické komponenty a její výpočetní třídy? Převedením rozhraní vlastní výpočetní třídy na rozhraní datového modelu komponenty, pro kterou chci použít moji výpočetní třídu. V našem případě to znamená, že přizpůsobíme metody třídy **DiarEngine** pro implementaci metod třídy **DiarTableModel**¹⁸.

Vraťme se nyní ke třídě **DiarEngine**. Ta obsahuje metody pro práci s diářem¹⁹. Jedna z instančních proměnných je prioritní fronta úkoly, která je při inicializaci třídy prázdná²⁰. Po každém vložení nového úkolu metodou `pridatUkol(GregorianCalendar datum, String ukol)`, dostaneme lokátor nového úkolu, který vrací volání metody:

```
Locator loc = ukoly.insert(datum,ukol);
```

¹⁶ To znamená, že nemusíme použít návrhový vzor **Pozorovatel** popsany na stránkách 2-9. Jinak v ostatních aplikacích této práce tento vzor používáme.

¹⁷ Tj. bez jakékoliv vazby na jakoukoliv grafickou komponentu.

¹⁸ Nejedná se o nic jiného než o použití návrhového vzoru **Adaptér**.

¹⁹ Implementace třídy a celé aplikace najdete v adresáři **Diář**.

²⁰ Ale nemusí být, pokud si natáhne data ze souboru. Viz implementace.

Ta se volá logicky uvnitř implementace metody `pridatUkol()`.

Lokátor `loc` se použije později například pro odstranění úkolu na který odkazuje. Voláním metody `vymazatUkol(GregorianCalendar datum)`, které předáme datum odstraňovaného úkolu, předáme metodě `remove(Locator loc)` objekt `loc`, díky němuž najdeme odstraňovaný úkol v konstantním čase a odstraníme ho.

Jenže kam tedy uložit `loc`, abychom ho mohli využít? Nejspíše to bude nějaká datová struktura, ale musíme se vyvarovat použití struktury, která by vynulovala výhodu lokátorů – vyhledávání prvku kontejneru v konstantním čase. Konkrétně, do prioritní fronty úkoly (do jakékoliv fronty) můžeme vložit neomezený počet úkolů²¹. Pro každý úkol tedy dostaneme lokátor. Těžko ho uložíme do zvláštní proměnné, protože by náš program (přesněji jeho kód) byl zanedlouho zahlcen proměnnými. Jenže použít seznam nebo dokonce strom pro uložení lokátorů není ideální řešení. Seznam vyhledává své prvky v čase $O(n)$ (když mažeme úkol, tak musíme **vyhledat** jeho lokátor) a strom (konkrétně vyvážený binární vyhledávací strom) sice vyhledává v kratším čase $O(\log n)$, ale je to za cenu pomalejšího vkládání. Jaké řešení tedy zvolit?

Řekli jsme výše, že to záleží na konkrétní aplikaci²². V našem diáři využijeme skutečnosti, že instance lokátoru obsahuje datum úkolu, na který odkazuje (podívejte se výše na kód rozhraní lokátoru). Tento údaj nám pomůže rychle najít odpovídající místo v **poli lokátorů**, které je uspořádané podle pořadí datumů v roce, kam lokátor uložíme. Stejně tak rychle najít lokátor, když ho potřebujeme. Naše pole lokátorů je trojrozměrné pole, kde index v prvním rozměru odpovídá roku, index v druhém rozměru odpovídá pořadí měsíce v roce a index v třetím rozměru odpovídá pořadí dne v měsíci. Takže data datumu lokátoru určují index místa v poli, kam lokátor uložíme nebo odkud ho zpět získáme. A jelikož index známe, **tak jak ukládání, tak i vyhledání proběhne v konstantním čase** a o to nám jde.

Konkrétně, naše třída `DiarEngine` obsahuje instanční proměnnou:

```
private Locator[][][] dniVRoce;
```

Ta je úložištěm lokátorů, které vrací metoda `insert(Object k, Object e)` fronty. Tato proměnná se inicializuje po startu aplikace voláním metody `inicialDnuVRoce()`:

```
private void inicialDnuVRoce(){
    dniVRoce = new Locator[2][12][];
    //předpokládám, že diář nebude obsahovat úkoly pro více než dva
    //roky za sebou

    for(int i=0; i!=dniVRoce.length;++i){
        for(int j=Calendar.JANUARY; j!=dniVRoce[0].length;++j){
            if(j==Calendar.JANUARY || j==Calendar.MARCH ||
               j==Calendar.MAY || j==Calendar.JULY ||
               j==Calendar.AUGUST || j==Calendar.OCTOBER ||
               j==Calendar.DECEMBER)
                dniVRoce[i][j] = new Locator[31];
            else if (j==Calendar.FEBRUARY)
                dniVRoce[i][j] = new Locator[29];
            else
                dniVRoce[i][j] = new Locator[30];
        }
    }
}
```

Jak vidíme, druhý rozměr odpovídá počtu měsíců v roce. První rozměr má pouze dva prvky (roky), protože aplikace pracuje pouze s aktuálním rokem a následujícím (málokdy budeme

²¹ Přesněji řečeno počet je omezený pamětí.

²² A na důmyslnosti programátora:-)

plánovat povinnosti už na rok 2011). Třetí rozměr závisí na konkrétním měsíci (některé měsíce mají 31 dní, některé 30) a alokuje se pro něj paměť v následujícím kódu.

Když si naplánujeme cestu do Prahy na 8. prosince 2005 (viz obrázek **Vlevo: hlavní okno...-vpravo**), vrácený lokátor se uloží do pole metodou `saveLocator(Locator loc)`:

```
private void saveLocator(Locator loc){
    Calendar date = (Calendar)loc.key();
    nastavDenMesicARok(date);
    dniVRoce[indexRoku(rok)][mesic][den-1] = loc;
}
```

Nejdříve získáme datum úkolu voláním metody `key()`. Z něj musíme vyextrahovat index místa v poli, kam se má lokátor uložit. To uděláme tak, že zavoláme příslušné metody objektu `date`, které do celočíselných proměnných `rok`, `mesic` a `den` uloží složky datumu. Obsah proměnných bude tento:

```
rok = 2005, mesic = 1123, den = 8
```

potom je použijeme jako indexy příslušných rozměrů pole, které tvoří konkrétní místo v něm, kam uložíme lokátor s daným datumem. V případě proměnné `rok`, převedeme její obsah číslo 2005 na index 0, protože je to aktuální rok (následující rok má index 1). Takže příkaz pro uložení lokátoru do pole bude vypadat takto:

```
dniVRoce[0][11][7] = loc;
```

Díky znalosti indexu místa kam náš lokátor uložíme, ukládáme ho v čase $O(1)$.

Stejně tak, když tento úkol budeme chtít odstranit, tak díky znalosti jeho datumu, můžeme najít jeho lokátor v poli lokátorů a ten předat metodě `remove(Locator loc)`, která podle něj najde úkol ve frontě a odstraní ho:

```
private Locator getLocator(GregorianCalendar date){
    nastavDenMesicARok(date);
    Locator loc = dniVRoce[indexRoku(rok)][mesic][den-1];
    return loc;
}
```

Vidíme, že po převedení složky datumu na příslušné indexy pole najdeme podle nich lokátor a uložíme ho do proměnné `loc`. Tu pak předáme metodě `remove()`.

Po odstranění úkolu ze fronty musíme ještě odstranit lokátor úkolu z pole. To učiní metoda `destroyLocator(GregorianCalendar date)`. Ta to udělá tak, že nastaví jeho index na `null`:

```
private void destroyLocator(GregorianCalendar date){
    nastavDenMesicARok(date);
    dniVRoce[indexRoku(rok)][mesic][den-1]=null;
}
```

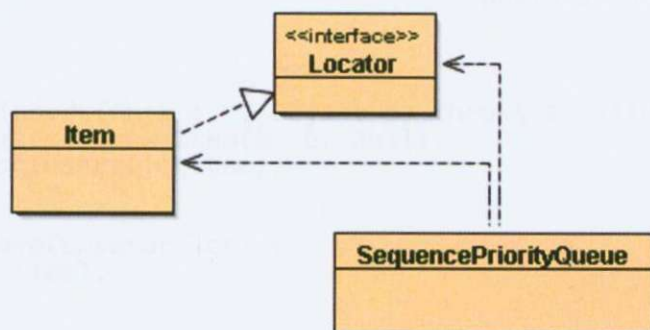
Obě operace vyhledání lokátoru a odstranění úkolu s ním spojený proběhnou v konstantním čase, tudíž použitím návrhového vzoru Lokátor jsme dosáhli požadovaného efektu a to je zrychlení práce s prioritní frontou a tudíž zrychlení běhu celé aplikace.

Implementace prioritní fronty s využitím lokátorů

Nyní, když už víme jak lokátory můžou přispět k urychlení práce s prioritní frontou, přistoupíme k prioritní frontě jako takové a ukážeme si jak má vypadat její implementace, aby podporovala návrhový vzor Lokátor.

²³ metoda objektu `date` `get(Calendar.MONTH)` vrací pořadí měsíce v roce snížené o 1.

Diagram stavebních prvků fronty pracující s lokátory má tuto podobu:



Z tohoto diagramu vidíme, že prioritní fronta je kontejner, který pracuje s prvky typu **položka (Item)** a protože se jedná o frontu, která podporuje vzor Lokátor, tak má napojení na jeho prvky (viz strana 23).

Kód fronty vypadá takto:

```

public class SequencePriorityQueue implements PriorityQueue {
    private Sequence S = new ListSequence();
    private Comparator comp;

    public SequencePriorityQueue (Comparator c) { comp = c; }

    private Object key (Position pos) {
        return ((Item) pos.getElement()).key();
    }
    private Object elem (Position pos) {
        return ((Item) pos.getElement()).element();
    }
    private Object elem (Object kep) {
        return ((Item) kep).element();
    }
    private Locator locInsert(Item locitem) throws InvalidKeyException{
        Position p, curr;
        Object k = locitem.key();
        if (!comp.isComparable(k))
            throw new InvalidKeyException("Neplatný klíč.");

        if (S.isEmpty())
            p = S.insertFirst(locitem);
        else if (comp.isGreaterThan(k, key(S.last())))
            p = S.insertAfter(S.last(), locitem);
        else {
            curr = S.first();
            while (comp.isGreaterThan(k, key(curr)))
                curr = S.after(curr);
            p = S.insertBefore(curr, locitem);
        }
        locitem.setPosition(p);
        return (Locator) locitem;
    }

    private Item locRemove(Locator loc) {
        S.remove(((Item)loc).position());
        return (Item) loc;
    }

    public int size () {return S.size(); }
    public boolean isEmpty () { return S.isEmpty(); }
    public Iterator elements(){
        return S.elements();
    }
    public Locator min() throws PriorityQueueEmptyException {
        try{
            return (Locator) S.first().getElement();
        }
    }
}

```

```

        catch(ContainerEmptyException e){
            throw new PriorityQueueEmptyException("Prioritní fronta je
                                                    prázdná.");
        }
    }

    public Locator insert(Object k, Object e) throws InvalidKeyException{
        Item locitem = new Item(k, e, null);
        return locInsert(locitem);
    }

    public void remove(Locator loc) {
        locRemove(loc);
    }

    public Object removeMin() throws PriorityQueueEmptyException {
        Object toReturn = ((Item) min()).element();
        remove(min());
        return toReturn;
    }

    public Object replaceElement (Locator loc, Object newElement) {
        Object oldElement = ((Item) loc).element();
        ((Item) loc).setElement(newElement);
        return oldElement;
    }

    public Object replaceKey(Locator loc, Object newKey)
                               throws InvalidKeyException{
        Item locitem = locRemove(loc);
        Object oldKey = ((Item) loc).key();
        locitem.setKey(newKey);
        locInsert(locitem);
        return oldKey;
    }
}

```

Z kódu metody `locInsert(Item locitem)` je vidět, jak získat odkaz na položku fronty v podobě lokátoru. Metoda `locInsert()` je sice formálně pomocná metoda využívaná metodou `insert(Object k, Object e)`, ale je klíčová pro pochopení její fungování.

Metoda `insert()` tedy funguje tak, že nejdříve ze všeho vytvoří novou položku kontejneru obsahující nový klíč a jeho hodnotu příkazem

```
Item locitem = new Item(k, e, null);
```

Položka je vytvořena, ale ještě není vložena do kontejneru. O to se `insert()` (pomocí pomocné metody `locInsert()`) pokusí v následujícím kroku, kdy na začátku ověří, jestli vložený klíč je platný klíč, tj. jestli se dá porovnávat s ostatními klíči v kontejneru. O tom rozhoduje objekt `comp` typu `Comparator`.

Je to objekt, který umí porovnávat klíče (z hlediska relace $\leq$) v položkách kontejneru. Využívá k tomu sadu metod jako například zde uvedenou metodu `isGreaterThan(Object keyOne, Object keyTwo)`, která rozhoduje o tom, zda-li klíč `keyOne` je větší než klíč `keyTwo` nebo metodu `isEqualTo(Object keyOne, Object keyTwo)`, která rozhoduje o rovnosti jejich argumentů²⁴. Ale dříve než objekt `comp` přistoupí k porovnávání nějakého klíče s jiným, rozhodne o tom, zda je tento klíč vůbec porovnatelný metodou

```
comp.isComparable(Object key);
```

Pokud tedy vložený klíč je platný, tak se vloží položka na odpovídající místo neboli pozici ve frontě. Jaké je to umístění rozhodují metody objektu `comp`, které vlastně určují prioritu nové položky. Pak se uloží informace o pozici nové položky do samotné položky. Tuto informaci pak musíme aktualizovat, kdykoliv položka mění pozici a to stejnou metodou

²⁴ Podrobněji o objekty typu `Comparator` budeme mluvit v kapitole o stejnojmenném návrhovém vzoru.

setPosition(Position p). **Tuto informaci později využijeme k přímému přístupu k položce. Takže nebudeme muset ji vyhledat.** Po vložení položky a uložení informace o její pozici vytvoříme a vrátíme lokátor této položky. Zajistíme to jednoduše příkazem

```
return (Locator) locitem;
```

Ten přetypuje novou položku locitem na objekt typu Lokátor a vrátí ho k pozdějšímu využití.

Když budeme nyní chtít použít nově vytvořený lokátor k odstranění poslední vložené položky, tak stačí ho dodat jako argument metodě remove(Locator loc). Ta vykoná tento úkon provedením pomocné metody locRemove(Locator loc). V jejím kódu je vidět celá podstata vzoru Lokátor, a to jak z hlediska jeho filozofie, tak i z hlediska implementace:

```
private Item locRemove(Locator loc) {
    S.remove(((Item)loc).position());
    return (Item) loc;
}
```

Vraťme se k přirovnání lokátoru k lístku od kabátu. Můžeme říct, že použijeme lístek loc k tomu, abychom našli jeho kabát. Najdeme ho podle čísla na lístku, které odpovídá číslu připojeném ke kabátu (dvojice dat key() a element() lokátoru loc odpovídají dvojicím dat key() a element() na položce lokátoru (Item) loc). Kabát s tímto číslem je na nějakém místě v šatně v zadní části. Najdeme ho tam (((Item)loc).position()) a vyzvedneme ho:

```
S.remove(((Item)loc).position());
```

Jinými slovy lokátor loc přetypujeme zpátky na položku na kterou ukazuje, najdeme pozici této položky (voláním její metody position()) a dodáme ji jako argument sekvenci, která už pozná, co je za položku na této pozici a odstraní ji.

Poslední dvě základní metody prioritní fronty jsou replaceKey(Locator loc, Object newKey) a replaceElement (Locator loc, Object newElement). Ta první mění prioritu položek a ta druhá mění hodnoty klíčů ve frontě.

Obě metody využíváme v diáři. Tím, že měníme datum plnění úkolu, měníme jeho prioritu. Opět dodáváme lokátor úkolu, jehož datum je třeba změnit na novou hodnotu newKey. Mluvíme nyní o metodě replaceKey(). Z jejího kódu poznáme, jak si položka aktualizuje svoji pozici (mění se totiž její priorita).

Metodu replaceElement() využíváme v diáři ke změně zadání úkolu pro určitý datum. Například když zítra místo práce na zahradě musíme jít nakoupit potřeby do domácnosti. Je to jednoduchá metoda, neboť nemusíme pracně aktualizovat pozici položky, protože zůstává na svém místě.

Vidíme, že návrhový vzor Lokátor je šikovný mechanismus pro urychlení práci s prvky kontejneru. Je to také poslední objektový vzor, které analyzujeme. V příští kapitole se začneme věnovat třídním návrhovým vzorům, neboli vzorům, které přímo mění strukturu tříd na které se aplikují.

Cvičení

1. Napište nové metody slovníku (např. binárního vyhledávacího stromu), které by podporovaly návrhový vzor Lokátor.
2. V příloženém CD je aplikace **Telefonní seznam** se zdrojovými kódy. Zkuste ji modifikovat tak, aby řadila jména nejen abecedně, ale i podle toho komu voláme nejčastěji. Využijte k tomu prioritní frontu.

4. Adapter

Návrhový vzor - Adapter

Ukázka

Ukázka aplikace, která využívá návrhový vzor Adapter. V tomto případě se jedná o aplikaci, která umožňuje komunikaci mezi různými typy zařízení (například mezi staršími a novějšími verzemi hardwaru) prostřednictvím standardizovaného rozhraní.

Ukázka kódu, který implementuje návrhový vzor Adapter. V tomto případě se jedná o kód, který umožňuje komunikaci mezi různými typy zařízení (například mezi staršími a novějšími verzemi hardwaru) prostřednictvím standardizovaného rozhraní.

Ukázka

Ukázka aplikace, která využívá návrhový vzor Adapter. V tomto případě se jedná o aplikaci, která umožňuje komunikaci mezi různými typy zařízení (například mezi staršími a novějšími verzemi hardwaru) prostřednictvím standardizovaného rozhraní.

Ukázka kódu, který implementuje návrhový vzor Adapter. V tomto případě se jedná o kód, který umožňuje komunikaci mezi různými typy zařízení (například mezi staršími a novějšími verzemi hardwaru) prostřednictvím standardizovaného rozhraní.

Ukázka aplikace, která využívá návrhový vzor Adapter. V tomto případě se jedná o aplikaci, která umožňuje komunikaci mezi různými typy zařízení (například mezi staršími a novějšími verzemi hardwaru) prostřednictvím standardizovaného rozhraní.

Třídní vzory

V předchozí části katalogu jsme pracovali s objektovými vzory neboli se vzory, které se aplikovaly primárně na objekty: Iterátory se používaly k projetí **objektů** kontejneru. Typ **Position** je abstrakcí pozice **objektu** v kontejneru a konečně lokátor definuje umístění (*location*) **objektu** v kontejneru.

V této části katalogu budeme pracovat, jak z jejího názvu vyplývá, se třídními vzory. Jsou to vzory, které zasahují do struktur tříd. Činí je pružnější, elegantnější a hlavně **znovupoužitelné**. O těchto vzorech jsme se okrajově zmiňovali v různých místech předchozí části práce. Asi už tušíme k čemu používat **Adaptér** nebo **Komparátor**. Nyní se na to soustředíme a řekneme si všechny důležité podrobnosti nutné pro pochopení těchto vzorů.

4. Adaptér

Název vzoru: *Adaptér – Adapter.*

Problém:

Jsou v praxi situace, kdy potřebujeme dodat do aplikace hotovou třídu, která by rozšířila její funkčnost, ale nemůžeme tak učinit, protože rozhraní hotové třídy neodpovídá rozhraní aplikace. Jde o problém popisovaný na stránce 10.

Z hlediska datových struktur potřebujeme vytvářet typově zaměřené kontejnery, které by uměly ukládat pouze objekty určitého typu a vyjímát je bez nutnosti přetypování na původní typ: nevýhodou používání kontejnerů v jazyce *Java* je totiž ztráta typové informace po uložení objektu do kontejneru.

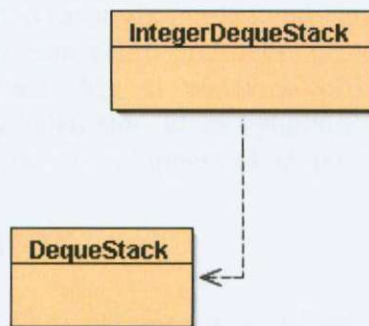
Řešení:

Oba tyto problémy lze vyřešit převedením rozhraní jedné třídy na rozhraní jiné třídy. V prvním případě to znamená převést rozhraní hotové třídy na rozhraní aplikace. V druhém případě to znamená použít rozhraní obecného kontejneru (kontejner, který ukládá heterogenní objekty tj. objekty různého druhu) pro tvorbu rozhraní typově zaměřeného kontejneru. Jak to udělat, to nám řekne návrhový vzor **Adaptér**:

Podstata provedení spočívá v tom, že upravíme (adaptujeme, přizpůsobíme) metody třídy, jejíž rozhraní nám nevyhovuje tak, aby se daly použít pro implementaci metod třídy s požadovaným rozhraním. Třidu s požadovaným rozhraním nazýváme **adaptér**. Adaptace by měla být jednoduchá. Tzn., že implementace metod adaptéru mají mít podobu jednořádkových volání metod přizpůsobující se třídy.

Nyní konkrétně: zaměříme se na adaptaci obecného kontejneru pro implementaci typově zaměřeného kontejneru. Půjde, konkrétně, o vytvoření zásobníku uchovávající pouze primitivní proměnné `int`²⁵ pomocí obecného zásobníku. Diagram tříd návrhového vzoru **Adaptér**, v tomto případě, bude mít tuto podobu:

²⁵ Ve skutečnosti bude uchovávat objekty typu `Integer`, ale navenek se bude tvářit, že uchovává primitivní proměnné typu `int`.



IntegerDequeStack je **adaptér** třídy DequeStack, neboť **využívá** její metody pro implementaci vlastních typově zaměřených metod. Implementace adaptéru vypadá takto:

```

package stack;
import vyjimky.*;
import hierarchy.*;

/**implementace adaptéru IntegerDequeStack*/
public class IntegerDequeStack{
    private Stack S;
    private Integer celeCislo;
    public IntegerDequeStack(){
        S = new DequeStack();
    }

    public int size(){
        return S.size();
    }
    public boolean isEmpty(){
        return S.isEmpty();
    }
    public int top()
        throws StackEmptyException{
        celeCislo = (Integer) S.top();
        return celeCislo.intValue();
    }
    public void push(int i){
        S.push(new Integer(i));
    }
    public int pop()
        throws StackEmptyException{
        celeCislo = (Integer) S.pop();
        return celeCislo.intValue();
    }
}

```

Vidíme, že metody zásobníku celých čísel implementujeme jednoduše tak, že voláme metody obecného zásobníku se stejnou funkcí s tím, že návratovou hodnotu přetypujeme na požadovaný typ `Integer`. Například metoda `pop()`, která z vrcholu zásobníku odstraňuje celá čísla, volá stejně funkční metodu `pop()` obecného zásobníku s tím, že její návratovou hodnotu převede na požadovaný typ `int`.

Všechny ostatní metody jsou implementovány ve stejném duchu, takže máme opravdu typově zaměřený zásobník, který pracuje s proměnnými typu `int`. Podobně můžeme naimplementovat zásobníky, operující s proměnnými typu `double` nebo typu `char`. Půjde o typově zaměřené zásobníky `DoubleDequeStack` nebo `CharDequeStack`.

Abychom mohli použít přizpůsobující se třídu (tady `DequeStack`) k implementaci adaptéru, tak adaptér musí vytvořit instanci (odkaz na) třídy `DequeStack`. Tuto instanci pak použije pro volání metod `DequeStack`. Všimněte si, že jde o jednoduchá jednořádková volání.

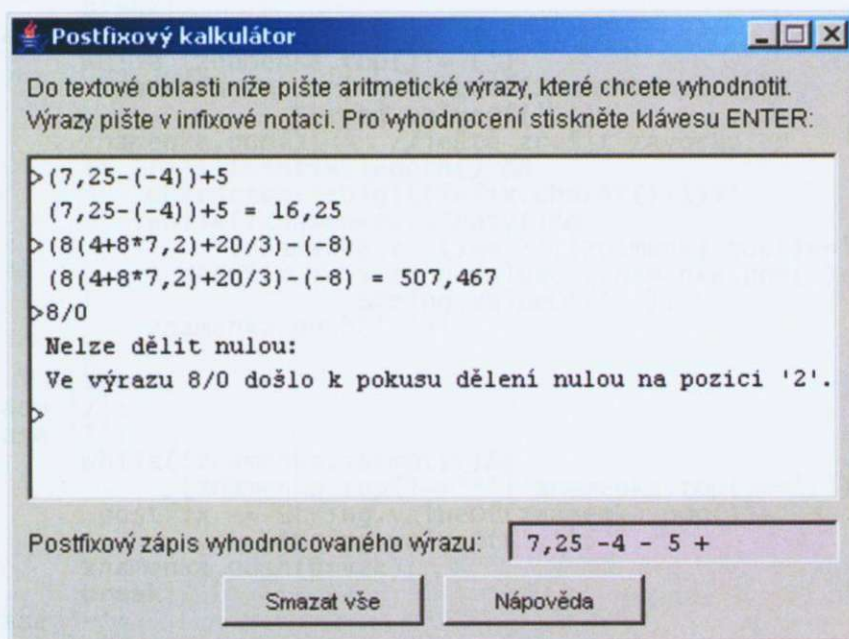
Objekt s je typu DequeStack. Je to zásobník implementovaný pomocí oboustranné fronty.

Typově zaměřené kontejnery jsou velmi praktické, protože díky jim můžeme se vyhnout nutnosti neustálého přetypování. My si ukážeme příklad použití typově zaměřených zásobníků, v aplikaci **Postfixový kalkulátor**, která vyhodnocuje aritmetické výrazy s využitím jejich postfixové notace a vypisuje, ve výstupním okně, postfixovou notaci výrazu.

Postfixový kalkulátor

Zásobníky se používají (mimo jiné) při zpracovávání aritmetických výrazů. Když chceme aritmetický výraz vyhodnotit, předtím musíme ho převést z infixové podoby, zadané uživatelem, na postfixový zápis, který je pro vyhodnocování či pro jiné další zpracování výrazu²⁶ nejvýhodnější. Pro tento účel využíváme pomocný zásobník na odkládání znamének, tj. operátorů a závorek. Potom když výraz je už v postfixové notaci snadno ho vyhodnotíme. Pro samotné vyhodnocování opět využijeme zásobník k ukládání číselných hodnot–mezivýsledků.

Naše aplikace **Postfixový kalkulátor**²⁷ umí vyhodnocovat aritmetické výrazy, které obsahují pouze číselné konstanty, konkrétně reálná čísla. Tzn., že neumí pracovat s proměnnými nebo s výrazy typu $\cos 45^\circ$ nebo $\log 20$. S operandy provádí jen základní aritmetické operace, tj. sčítání, odečítání, násobení a dělení. Tyto funkce stačí pro demonstraci parametrizovaných zásobníků. Grafické uživatelské rozhraní aplikace máte na následujícím obrázku:



GUI postfixového kalkulátoru: v textovém poli dole je postfixová notace výrazu $(7.25-(-4))+5$.

Uživatel zadává aritmetické výrazy na příkazovém řádku za ostrou hranatou závorkou (>). Po každém potvrzení příkazu, klávesou ENTER, objeví se, v textovém poli v pravém dolním

²⁶ Z matematiky jsme všichni zvyklí na běžný zápis aritmetických výrazů, který bývá někdy označován jako infixová notace. Vedle běžně používané infixové notace se zejména v programování objevují ještě další formy zápisu aritmetických výrazů, jedna z nich je postfixový zápis.

Postfixový zápis spočívá v tom, že každý operátor v zápisu výrazu následuje bezprostředně za oběma operandy, k nimž se vztahuje. Přitom každým z jeho operandů může být buď přímo číselná konstanta, nebo nějaký podvýraz stejné postfixové struktury. Například infixový výraz $(65 - 3*5)/(2 + 3)$ má v postfixové notaci podobu $65\ 3\ 5\ *\ -\ 2\ 3\ +\ /\$.

²⁷ Najdete ji i se zdrojovými kódy v adresáři Postfixový kalkulátor.

rohu aplikace, postfixová notace aktuálně vyhodnocovaného výrazu.

Výrazy zadané uživatelem program zpracovává jako řetězce, které ukládá do proměnné infix typu String. Je to instanční proměnná výpočetní třídy aplikace kalkulatorEngine. Právě ta má vyhodnocování výrazu na starost. Její metoda infixNaPostfix() převede řetězec infix na postfixovou podobu:

```
/**
 * převede infixový výraz na postfixovou notaci.
 * @return vrací postfixový zápis výrazu.
 */
public String infixNaPostfix() throws SyntaxError{
    try {
        syntax();
        postfix="";
        CharDequeStack znamenka = new CharDequeStack();
        for(int i=0;i!=infix.length();++i){
            prvek=infix.charAt(i);
            switch (prvek){
                case '(':
                    if ((i!=0)&&(Character.isDigit(infix.charAt(i-1))||
                        (infix.charAt(i-1)==' ')||((infix.charAt(i-1)=='-'&&
                        jeUnarni('-',i-1)))) {
                        while(!znamenka.isEmpty()&&(znamenka.top()=='*'||
                            znamenka.top()=='/'))
                            postfix += String.valueOf(znamenka.pop())+
                                String.valueOf(' ');
                        znamenka.push('*');
                    }
                    znamenka.push(prvek);
                    break;
                case ')':
                    while (znamenka.top()!='(')
                        postfix += String.valueOf(znamenka.pop())+
                            String.valueOf(' ');
                    znamenka.pop(); //ještě zrušit závorku.
                    if ((i+1)!=infix.length() &&
                        Character.isDigit(infix.charAt(i+1))) {
                        while(!znamenka.isEmpty()&&
                            (znamenka.top()=='*'||znamenka.top()=='/'))
                            postfix += String.valueOf(znamenka.pop())+
                                String.valueOf(' ');
                        znamenka.push('*');
                    }
                    break;
                case '/':
                case '*':
                    while(!znamenka.isEmpty()&&
                        (znamenka.top()=='*'||znamenka.top()=='/'))
                        postfix += String.valueOf(znamenka.pop())+
                            String.valueOf(' ');
                    znamenka.push(prvek);
                    break;
                case '-':
                case '+':
                    if (jeUnarni(prvek,i)) {
                        if (prvek == '-')
                            if (infix.charAt(i+1)=='(') postfix += "-1 ";
                        else postfix += String.valueOf(prvek);
                    } else {
                        while(!znamenka.isEmpty()
                            &&(znamenka.top()=='+'||znamenka.top()=='-'||
                                znamenka.top()=='*'||znamenka.top()=='/'))
                            postfix += String.valueOf(znamenka.pop())+
                                String.valueOf(' ');
                        znamenka.push(prvek);
                    }
                    break;
                default:
                    postfix += String.valueOf(prvek);
                    if (((i+1)==infix.length())||((infix.charAt(i+1)!=' ')&&
                        (!Character.isDigit(infix.charAt(i+1)))))

```

```

        postfix += String.valueOf(' ');
    }
    while(!znamenka.isEmpty())
        postfix += String.valueOf(znamenka.pop())+String.valueOf(' ');
    return postfix;
} catch (SyntaxError e) {
    throw e;
}
}

```

Metoda funguje tak, že čte infixový zápis po prvcích zleva doprava (viz [4] strana 232). Jedním prvkem rozumíme nejen číselnou konstantu nebo binární operátor, ale také levou či pravou závorku. Jak jsme se již zmínili, k výpočtu se využívá pracovní zásobník `znamenka` typu `CharDequeStack` na odkládání znamének, tj. operátorů a závorek. V postfixovém zápisu výrazu mají všechny operandy stejné vzájemné pořadí jako v zápisu infixovém. Operátory se ovšem oproti infixu musí pozdržet, aby následovaly až za „svými“ operandy. Závorky určují způsob pozdržování jednotlivých operátorů, do postfixového zápisu se ale již neukládají. Při převádění tedy budeme postupovat následovně. Kdykoli přečteme z infixového zápisu číselnou konstantu, přímo ji zapíšeme do vytvářeného postfixového zápisu uloženého do instanční proměnné `postfix`. Jestliže přečteme aritmetický operátor, vložíme ho do zásobníku. Předtím ale ještě ze zásobníku vyjmeme a zapíšeme do `postfix` všechny operátory stejné nebo vyšší priority, které tam na vrchu jsou. Výraz je totiž vyhodnocován podle priorit a operátory stejné priority zleva doprava, takže žádný operátor nesmí v postfixovém zápisu „předběhnout“ jiný operátor stejné nebo vyšší priority. Pokud přečteme ze vstupu levou závorku, uložíme ji vždy do zásobníku. Až později přečteme závorku pravou, vyjmeme postupně ze zásobníku a zapíšeme do `postfix` všechny operátory ležící nad zaznamenanou levou závorkou. Tuto závorku ze zásobníku také vyzvedneme, ale do výstupního postfixového zápisu ji neukládáme.

Metoda před vlastním převedením zavolá metodu `syntax()`, aby zkontrolovala správnost zápisu infixového výrazu zadaného uživatelem.

Řekli jsme, že metoda `infixNaPostfix()` ke své práci používá zásobník typu `CharDequeStack`. Je to typově zaměřený zásobník, který pracuje se znaky (operátory a závorky nejsou nic jiného než znaky). Vidíme jaké je to velké zjednodušení, že můžeme vkládat prvky do zásobníku nebo vyjímát je bez nutnosti externího přetypování, tj. přímo.

Podobně využíváme zásobník operandů typu `DoubleDequeStack()` v metodě `kalkulacka()`, která provádí vlastní vyhodnocování. Ten se používá pro ukládání mezivýsledků vyhodnocování, pracuje tedy s proměnnými typu `double`. Nebudu tu vysvětlovat algoritmus této metody. Zájemce se mohou podívat do zdrojového kódu v adresáři **Postfixový kalkulátor** a do knihy programovací techniky Pavla Töpfera na straně 220.

Pro úplnost zmíním se ještě o `IntegerDequeStack`, který se využívá, ve třídě `kalkulatorEngine`, pro kontrolu správnosti zápidu zadaného výrazu (tj. v metodě `syntax()`).

Metoda `syntax()` čte výraz zleva doprava a kontroluje jeho správnost. Zásobník se využívá pro kontrolu závorek, tj. jestli nepřebývá nějaká závorka. Funguje to tak, že po přečtení jakékoliv levé závorky její index se ukládá do zásobníku. Později se index využívá k formulaci hlášení na jakém místě (jaký je index závorky v řetězci) přebývá závorka. Když metoda narazí na pravou závorku, tak vyzvedne ze zásobníku posledně uloženou levou závorku a kontroluje tak, zda každá levá závorka má „svoji“ pravou polovičku. Pokud metoda projde celý výraz a v zásobníku zbude levá závorka, nebo během procházení výrazu narazí na pravou závorku a zásobník je prázdný, vyhodí výjimku. Kód metody `syntax()` vypadá takto:

```

/**
 * Zkontroluje správnost zápisu aritmetického výrazu.
 * @throws SyntaxError pokud je chyba v zápisu aritmetického výrazu.
 */
public void syntax() throws SyntaxError{
    String operace; //Dělíme, násobíme, sčítáme, nebo odečítáme?
    int pocetCerek = 0; //Počet desetinných čárek v čísle
    boolean jeCislice;
    String chyba = " Chyba ve výrazu "+this.vyraz+" :\n";

    IntegerDequeStack indexyLevychZavorek = new IntegerDequeStack();

    for(int i=0;i!=vyraz.length();++i){
        prvek=vyraz.charAt(i); //i-tý znak ve výrazu
        if(!jeCislice(prvek)) //je to číslice?
            pocetCerek = 0; //Pokud, už je konec čísla tak vynuluj počet
                             //desetinných čárek.
        switch (prvek){ //Co je to za znak?
            case '(':
                indexyLevychZavorek.push(i);
                break;
            case ')':
                /** pokud nemá svoji levou závorku, nebo bezprostředně předchozí
                 * znak je levá závorka*/
                if ((indexyLevychZavorek.isEmpty())||
                    (vyraz.charAt(i-1)=='('))
                    throw new SyntaxError(chyba+" Neplatná pravá závorka
                                         na pozici '"+(i+1)+"'."");
                indexyLevychZavorek.pop();
                break;
            case '/':
            case '*':
                /** operátory *,/ jsou neplatné v následujících případech:
                 * -nachází se na začátku, nebo na konci výrazu.
                 * -předchází jim levá nebo následuje po nich pravá závorka.
                 * -následují po sobě dva operátory násobení nebo dělení.
                 */
                if (prvek=='/') operace="dělení";
                else operace="násobení";
                if ((i==0)||(i==vyraz.length()-1))
                    throw new SyntaxError(chyba+" Operátor "+operace+
                                           " se nachází na začátku nebo na konci výrazu.");
                if((vyraz.charAt(i-1)=='(')||(vyraz.charAt(i+1)=='('))
                    throw new SyntaxError(chyba+" Operátoru "+operace+
                                           " na pozici '"+(i+1)+"' předchází nebo nadchází závorka.");
                if((vyraz.charAt(i+1)=='/')||(vyraz.charAt(i+1)=='*'))
                    throw new SyntaxError(chyba+" Dva po sobě jedoucí
                                           operátory dělení nebo násobení na pozici '"+(i+1)+"'."");
                break;
            case '-':
            case '+':
                /** operátory -,+ jsou neplatné v následujících případech:
                 * -nachází se na konci výrazu.
                 * -následuje po nich pravá závorka, nebo jakýkoliv jiný
                 * operátor(*,/,+,-).
                 */
                if (prvek=='-') operace="rozdíl";
                else operace="součet";
                if (i==vyraz.length()-1)
                    throw new SyntaxError(chyba+" Operátor "+operace+" se
                                           nachází na konci výrazu.");
                if (vyraz.charAt(i+1)=='(')
                    throw new SyntaxError(chyba+" Po operátoru "
                                           +operace+" následuje pravá závorka na pozici '"+(i+2)+"'."");
                if((vyraz.charAt(i+1)=='/')||(vyraz.charAt(i+1)=='*')||
                    (vyraz.charAt(i+1)=='+')||(vyraz.charAt(i+1)=='-'))
                    throw new SyntaxError(" Neplatná posloupnost
                                           operátorů na pozici '"+(i+1)+"'."");
                break;
        }
    }
}

```

```

        case ',':
/** desetinná čárka je neplatná pokud:
 * -se nachází na začátku nebo konci výrazu.
 * -se vyskytuje v čísle více desetinných čárek než jedna a to i
 * bezprostředně po sobě.
 */
        ++pocetCarek; //Počet desetinných čárek
        if((i==0)|| (i==vyraz.length()-1)|| (pocetCarek>1))
            throw new SyntaxError(chyba+" Neplatná desetinná čárka na
                                pozici '"+(i+1)+"'");
        break;
    default:
        //Pokud je to jiný znak než předchozí platné znaky
        if (!jeCislice(prvek))
            throw new SyntaxError(chyba+" Neplatný znak '"
                                +String.valueOf(prvek)+"' na pozici '"+(i+1)+"'");
    } //endswitch
} //endfor
//pokud je ve výrazu přebytečný počet levých závorek
if(!indexyLevychZavorek.isEmpty()) {
    int index = indexyLevychZavorek.pop(); //index poslední neplatné
                                           //závorky
    while(!indexyLevychZavorek.isEmpty()) //Dokud zásobník není prázdný
        indexyLevychZavorek.pop();
    throw new SyntaxError(chyba+" Neplatná levá závorka na pozici '"
                        +(index+1)+"'");
}
//Pokud jsme došli až sem, tak výraz je syntakticky správný
infix=vyraz; //....a máme nový infixový zápis
}

```

Poziční iterátor jako adaptér

Když se znova podíváte na stranu 20 na implementaci pozičního iterátoru, tak si uvědomíte, že poziční iterátor je adaptér obecného iterátoru. Je to typově zaměřený iterátor, který pracuje s typem `Position` neboli s pozicemi v kontejneru. Poziční iterátor využívá obecný iterátor tak, že přizpůsobuje (adaptuje) jeho metody pro implementaci vlastních metod.

DequeStack

Adaptér se využívá také tam, kde chceme univerzálnější kontejner uplatnit pro implementaci kontejneru, který má konkrétní účel. To je právě případ třídy `DequeStack`. Je to zásobník, který implementuje svoje metody pomocí oboustranné fronty (*deque*). Zatímco oboustranná fronta se málo používá, tak zásobník je jeden z nejdůležitějších datových struktur. Zásobník je implementován v instrukční sadě mikroprocesoru. Využívá ho Java Virtual Machine. Každý proces běžící v programu, vytvořeným v Javě, má svůj vlastní zásobník metod. Zásobník se používá při řešení úloh metodou „rozděl a panuj“ (viz [4] strana 90) k odložení informací o tom, které dílčí úkoly je ještě třeba vykonat. Konec konců zásobník se využívá pro zpracování aritmetických výrazů. Proč tedy neuplatnit oboustrannou frontu v implementaci zásobníku?

Toto nám umožňuje návrhový vzor Adaptér. Ten použije rozhraní oboustranné fronty pro implementaci metod zásobníku. Při převodu rozhraní univerzálního kontejneru na rozhraní speciálního kontejneru se může stát, že některé metody univerzálního kontejneru nevyužijeme vůbec. To je i náš případ:

```

/**Implementace zásobníku pomocí oboustranné fronty*/
public class DequeStack implements Stack{
    private Deque D;

```

```

public DequeStack(){
    D = new LinkedDeque();
}

public int size(){
    return D.size();
}
public boolean isEmpty(){
    return D.isEmpty();
}
public Iterator elements(){
    return D.elements();
}
public Object top()
    throws StackEmptyException{
    try {return D.last();}
    catch (DequeEmptyException err){
        throw new StackEmptyException("Zásobník je prázdný.");
    }
}
public void push(Object o){
    D.insertLast(o);
}
public Object pop()
    throws StackEmptyException{
    try {return D.removeLast();}
    catch (DequeEmptyException err) {
        throw new StackEmptyException("Zásobník je prázdný.");
    }
}
}

```

Vidíme, že při adaptaci metod fronty pro použití v zásobníku vůbec nevyužíváme metody fronty `insertFirst(Object e)` a `removeFirst()`. Počátek fronty, v této implementaci, odpovídá dnu zásobníku a ke dnu zásobníku nemáme vůbec přístup, tudíž v zásobníku tyto metody jsou nepoužitelné.

V kódu opět vidíme, že musíme deklarovat referenci na frontu, abychom mohli využít její metody. Vidíme též, že když vložíme objekt na vrchol zásobníku, tak tento objekt vlastně vkládáme na konec fronty (voláním metody `insertLast(Object e)`). Stejně tak, když ho odebíráme z vrcholu zásobníku, tak ho odebíráme z konce fronty.

Je řada jiných příkladů, kdy využíváme jeden kontejner k implementaci metod jiného. Například oboustrannou frontu můžeme naimplementovat pomocí seznamu. Podívejte se na její implementaci v balíku `queues`.

Návrhový vzor Adaptér je mocný nástroj pro navrhování aplikací. O tom se můžete přesvědčit, když se znova podíváte na návrh tříd aplikace **Diář**. Chtěl jsem zachovat nezávislost výpočetní třídy `DiarEngine` (viz strana 26), ale přitom těžit z výhod jazykových konstrukcí programovacího jazyka Java. Konkrétně jsem chtěl využít výhodu, že třída `JTable` má svůj vlastní datový model, který je implicitně uzpůsobený pro komunikaci s grafickými komponentami typu tabulka. Toho jsem dosáhl právě použitím vzoru Adaptér (převodl jsem rozhraní třídy `DiarEngine` na rozhraní `DiarTableModel`). O tom se můžete přesvědčit, když se podíváte na implementaci třídy `DiarTableModel`.

Můžeme říct z jiného úhlu pohledu, že jsme použili **obecnou** třídu `DiarEngine` (je totiž nezávislá na jakékoliv grafické komponentě) k implementaci metod „typově zaměřené“²⁸ výpočetní třídy `diáře` (pracuje jen s tabulkami).

²⁸ Je to vědomá nadsázka:-)

Cvičení

1. Použijte návrhový vzor Adaptér k implementaci zásobníku pomocí dvou front.
2. Naimplementujte metody `insertLast` a `insertBefore` spojového seznamu použitím jen kombinaci metod `isEmpty`, `checkPosition`, `insertAfter` a `insertFirst`.

5. Komparátor

Název vzoru: *Komparátor – Comparator.*

Problém:

Potřebujeme způsob, jak porovnávat v kontejneru objekty (podle relace $\leq$) bez toho abychom pro každý typ objektů napsali zvláštní implementaci kontejneru. Každý typ objektů totiž má svoje specifická pravidla pro porovnávání. Například jinak porovnáváme slova (abecedně) a jinak datumy. Tudíž museli bysme napsat zvláštní implementaci slovníku, která by porovnávala slova (obsahovala by speciální metody pro porovnávání slov), a jinou pro porovnávání datumů²⁹ (obsahovala by úplně jiné metody pro porovnávání těchto datumů). Tyto dvě implementace slovníku by se lišily akorát metodami pro porovnávání svých objektů, jinak by byly naprosto stejné. To je neefektivní, protože to přenáší spoustu duplicitního kódu. Je to ztráta času napsat pro každý typ porovnatelných objektů, kterých je nespočet, zvláštní implementaci kontejneru.

Řešení:

Můžeme přenést odpovědnost porovnávání na samotné porovnávané objekty. Tak každý objekt by byl schopen porovnat sebe s ostatními objekty v kontejneru. Napsali bysme tak jednu jedinou implementaci kontejneru, protože ten by se už nestaral o porovnávání svých objektů. Tyto objekty totiž jsou typu třídy, která implementuje nějaké rozhraní typu *Comparable* a zapouzdřuje metody nutné pro porovnávání těchto objektů. Toto řešení je jistý pokrok, protože nám dovolí napsat obecnou implementaci kontejneru, který by se uměl poradit se spousty typů klíčů.

Jsou ale jisté případy, kdy toto řešení je nevyhovující. Existují objekty, které neví, jak se mají navzájem porovnávat. Podívejme se na následující příklad.

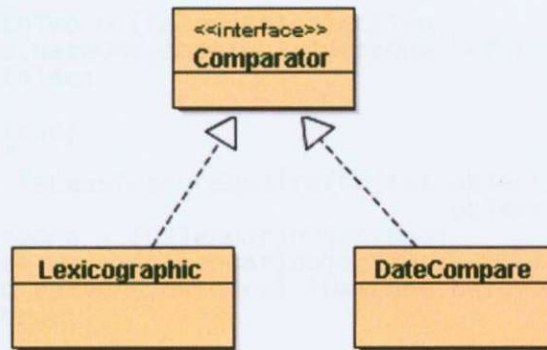
Geometrický algoritmus může porovnávat v rovině body A a B (viz [1] strana 293) podle jejich x-ové souřadnice (tj. $A \leq B$, jestliže $x(A) \leq x(B)$) pokud by chtěl je uspořádat zleva doprava, zatímco jiný algoritmus by je mohl porovnávat podle jejich y-ové souřadnice (tj. $A \leq B$, jestliže $y(A) \leq y(B)$) pro jejich uspořádání zezdola nahoru. Neexistuje žádné pravidlo, které by nám jednoznačně řekl, jestli porovnávat body v rovině podle x-ové souřadnice nebo podle y-ové souřadnice. Jsou totiž ještě jiné způsoby, jak porovnávat body (například můžeme porovnávat vzdálenosti A a B od počátku soustavy souřadnic).

Neměli bychom tedy spoléhat na samotné objekty, že se umí navzájem porovnávat. Místo toho použijeme speciální **porovnávací** objekty, které se nachází mimo implementaci porovnávaných objektů. Ty by porovnávaly tyto objekty. Tento porovnávací objekt nazýváme **komparátorem**. **Komparátor je objekt, který porovná dva porovnatelné objekty.**

Když budeme chtít napsat implementaci uspořádaného kontejneru, tak vždycky to uděláme tak, že dodáme jeho konstruktoru konkrétní typ komparátoru, který by uměl porovnávat objekty kontejneru. Komparátorů je celá řada. Každý typ objektů může mít svůj komparátor, tedy „porovnávač“, který umí objekty tohoto typu porovnávat. Například slovník pracující se jmény v telefonním seznamu použije komparátor *Lexicographic*, který umí porovnávat klíče (objekty) typu *String*. Prioritní fronta pracující se klíči typu *GregorianCalendar* tj. datumy (viz aplikace **Diář**) použije komparátor *DateCompare*.

Návrhový vzor Komparátor se bude skládat z rozhraní porovnávacích objektů, které je jednotné a z konkrétních komparátorů. My v následujícím diagramu uvádíme dva typy komparátorů zmíněné výše.

²⁹ jak datumy tak i slova by byly klíče slovníku.



rozhraní Comparator má následující operace:

- `isLessThan(a,b)`: je to pravda jen a jen tehdy, když `a` je menší než `b`.
Input: Object(2x) Output: Boolean
- `isLessThanOrEqualTo(a,b)`: je to pravda jen a jen tehdy, když `a` je menší než nebo rovno `b`.
Input: Object(2x) Output: Boolean
- `isEqualTo(a,b)`: je to pravda jen a jen tehdy, když `a` je rovno `b`.
Input: Object(2x) Output: Boolean
- `isGreaterThan(a,b)`: je to pravda jen a jen tehdy, když `a` je větší než `b`.
Input: Object(2x) Output: Boolean
- `isGreaterThanOrEqualTo(a,b)`: je to pravda jen a jen tehdy, když `a` je větší než nebo rovno `b`.
Input: Object(2x) Output: Boolean
- `isComparable(a)`: je to pravda jen a jen tehdy, když `a` je porovnatelné.
Input: Object Output: Boolean

Každý komparátor má vlastní implementaci těchto metod. Záleží na povaze porovnávaných objektů. My si ukážeme implementace `DateCompare`, které porovnává objekty typu datum:

```

package comparator;
import java.util.*;

/**Implementace komparátoru, který porovnává objekty typu Calendar*/
public class DateCompare implements Comparator{
    public boolean isComparable(Object o){
        if(o instanceof Calendar)
            return true;
        else
            return false;
    }
    public boolean isEqualTo(Object objectOne, Object objectTwo){
        Calendar dateOne = (Calendar)objectOne;
        Calendar dateTwo = (Calendar)objectTwo;
        if((dateOne.get(Calendar.YEAR)==dateTwo.get(Calendar.YEAR))&&
            (dateOne.get(Calendar.MONTH)==dateTwo.get(Calendar.MONTH))&&
            (dateOne.get(Calendar.DAY_OF_MONTH)==
            dateTwo.get(Calendar.DAY_OF_MONTH)))
            return true;
        else return false;
    }
    public boolean isLessThan(Object objectOne, Object objectTwo){
        Calendar dateOne = (Calendar)objectOne;
        Calendar dateTwo = (Calendar)objectTwo;
        if(isEqualTo(dateOne,dateTwo)||dateOne.after(dateTwo))
            return false;
        else return true;
    }
    public boolean isGreaterThan(Object objectOne, Object objectTwo){
        Calendar dateOne = (Calendar)objectOne;

```

```

        Calendar dateTwo = (Calendar)objectTwo;
        if(isEqualTo(dateOne,dateTwo)||dateOne.before(dateTwo))
            return false;
        else
            return true;
    }
    public boolean isLessThanOrEqualTo(Object objectOne, Object
                                   objectTwo){
        Calendar dateOne = (Calendar)objectOne;
        Calendar dateTwo = (Calendar)objectTwo;
        if(isEqualTo(dateOne,dateTwo)||dateOne.before(dateTwo))
            return true;
        else
            return false;
    }
    public boolean isGreaterThanOrEqualTo(Object objectOne, Object
                                   objectTwo){
        Calendar dateOne = (Calendar)objectOne;
        Calendar dateTwo = (Calendar)objectTwo;
        if(isEqualTo(dateOne,dateTwo)||dateOne.after(dateTwo))
            return true;
        else
            return false;
    }
}

```

Všimněte si, že komparátor nemá vlastní data (instanční proměnné). **Je to logické, neboť komparátor je objekt, který je vytvořený speciálně proto, aby porovnával objekty své třídy. Jeho existence jiný smysl nemá.** Objekty, které má porovnávat, jsou mu dodávány zvenčí z kontejneru, který tyto objekty ukládá a chce vědět kam je zařadit. Kontejner rozhoduje o zařazení na základě hodnot vrácených komparátorem.

Algoritmus porovnávání datumů je jednoduchý. Komparátor porovnává jednotlivé složky dodaných datumů a na jejich základě rozhodne, které datum je větší, menší nebo jestli jsou si rovny.

Metody rozhraní Comparatoru mají dlouhá jména v zájmu čitelnosti. Pro zajímavost balík java.util také obsahuje rozhraní pro návrhový vzor Komparátor. Toto rozhraní obsahuje pouze jedinou metodu:

compare(a,b): vrací celé číslo i , takové že $i < 0$ jestliže $a < b$; $i = 0$ jestliže $a = b$ a $i > 0$ jestliže $a > b$.

Input: Object(2x) Output: Integer

Nezáleží na tom, jestli používáme dlouhá nebo krátká jména metod. Důležité je, že návrhový vzor Komparátor nabízí univerzální, dynamický a znovupoužitelný způsob, jak porovnávat objekty.

Porovnávání datumů v diáři

Komparátory se používají všude, kde je třeba uspořádat objekty. Tj. ve všech uspořádaných kontejnerech jako jsou například slovníky nebo prioritní fronty, které uspořádají své prvky podle klíče.

Vraťme se nyní k aplikaci **Diář**, která používá prioritní frontu k ukládání úkolů. Je to instanční proměnná třídy DiarEngine (proměnná ukoľy):

```

/**výpočetní třída aplikace Diář*/
public class DiarEngine{
    .....
    private PriorityQueue ukoľy;
    private Comparator dateCompare;
    .....
}

```

```

public DiarEngine(){
    inicialDnuVRoce();
    formatter = new SimpleDateFormat("", new Locale("cs", "CZ"));
    dnesniDatum = new GregorianCalendar();
    dateCompare = new DateCompare();
    ukoly = new SequencePriorityQueue(dateCompare);
}

```

Vidíme, že při inicializaci fronty `ukoly` dodáme jejímu konstruktoru komparátor typu `DateCompare`. Ten bude porovnávat vložené úkoly do fronty podle jejich klíče (datum úkolu). Už víme, že komparátor `DateCompare` toto umí. Je to jeho specialita.

Kdykoli uživatel vloží do fronty úkol, zavolá se metoda `pridatUkol(GregorianCalendar datum, String ukol)`, která předá argumenty `datum` a `ukol` metodě fronty `insert(Object key, Object element)`, která vzápětí předá (přesněji její pomocná metoda `locInsert()` viz strana 29) klíč `datum` objektu `dateCompare`. Ten ho porovná s posledním prvkem fronty (prvek s nejnižší prioritou³⁰) a na základě jeho rozhodnutí (zda-li je porovnávaný prvek menší nebo větší než poslední prvek fronty), fronta už ví kam ho zařadit. Komparátor `dateCompare` nejdříve rozhodne, jestli je to platný klíč, tj. jestli je to datum. Vidíme, že komparátor přijímá prvky, které má porovnávat, zvenčí.

Když se podíváte pozorněji na implementaci fronty, tak si uvědomíte, že v jejím kódu není žádná zmínka o tom, jakého typu je její komparátor. Obrazně řečeno fronta neví jakého typu je její komparátor a ani jí to nemusí zajímat. Ať je jakýkoliv, vždycky se fronta řídí výsledky volání jeho metod. Fronta řekne: podívej se mi ty, nějaký komparátore, na tyto dva objekty a řekni mi, který je větší (metoda `isGreaterThan(a,b)`) a na základě výsledku volání metody fronta rozhodne o zařazení toho prvku. Jinými slovy frontě prostě stačí k rozhodování vědět, který prvek je větší, ale už jí nemusí zajímat proč (to zařídí komparátor). Z toho vyplývá, **že ať je dodaný komparátor jakýkoliv kód fronty zůstává stejný.**

To znamená, že když budeme chtít vytvořit prioritní frontu uchazečů o zaměstnání, v ní by prioritou uchazeče byla určena délkou jeho praxe, tak stačí vytvořit komparátor objektů typu `Uchazeč`, který by uměl porovnávat jejich praxe a ten dodáme našemu objektu `ukoly`. Náš objekt „nejednou“ umí řadit uchazeče³¹. Nemusíme tedy vytvářet zvláštní frontu pro uchazeče o zaměstnání. A to je náš cíl.

³⁰ Z hlediska komparátoru `DateCompare` největší prvek ve frontě. Nejmenší prvek fronty má naopak největší prioritu.

³¹ Samozřejmě bylo by vhodné přejmenovat proměnnou `ukoly` na `uchazeci`.

Cvičení

1. Naimplementujte objekt typu `Comparator`, který by uměl porovnat česká slova. komparátor `Lexicographic` umí porovnat jen řetězce bez diakritiky. (Návod: použijte knihovni třídu `Javy Collator`).
2. modifikujte aplikaci Telefonní seznam tak, aby uměla ukládat a řadit i jména s diakritikou.

6. Dekorátor

Název vzoru: *Dekorátor – Decorator.*

Problém:

Návrhový vzor Dekorátor nabízí nám způsob, jak upravit chování jednotlivých objektů třídy bez nutnosti vytvořit k tomu účelu novou podtřídu této třídy. Předpokládejme, že máme program, který používá osm objektů, ale tři z nich potřebují nové funkce. Můžete sice vytvořit podtřídu pro každý z těchto objektů a v mnoha případech je to ideální řešení. Nicméně, jestliže každý z těchto tří objektů vyžaduje *odlišné* funkce, musíte vytvořit tři odvozené třídy. Dále, jestliže každá z těchto tříd má vlastnosti ostatních *dvou* tříd, začnete vytvářet zbytečné a matoucí komplikace.

Například předpokládejme, že chceme vytvořit speciální ohraničení kolem některých tlačítek na panelu nástrojů (toolbar) programu (viz [3] strana 103). Jestliže vytvoříme novou podtřídu tlačítka, tak by to znamenalo, že všechna tlačítka v této nové třídě budou mít vždy tohle nové ohraničení. A toto nemusí být náš záměr.

Místo toho vytvoříme dekorační třídu, která by „vyzdobila“ (*decorate*) tlačítka. Potom můžeme odvodit libovolný počet dekorátorů (dekoračních objektů) z hlavní dekorační třídy a každý z nich by nám nabídnul určitou dekoraci.

Když přečtete předchozí odstavce, tak se právem můžete ptát: ano je to zajímavý mechanismus, ale mě mate, že se vzor používá na objekty jako takové a ne na třídy. Proč je tedy řeč o vzoru až tady? Je zde, protože konkrétně u datových struktur návrhový vzor Dekorátor se aplikuje na třídy, tedy ne na jejich jednotlivé objekty. Ale povaha problémů, pro které se tento vzor používá je úplně stejná.

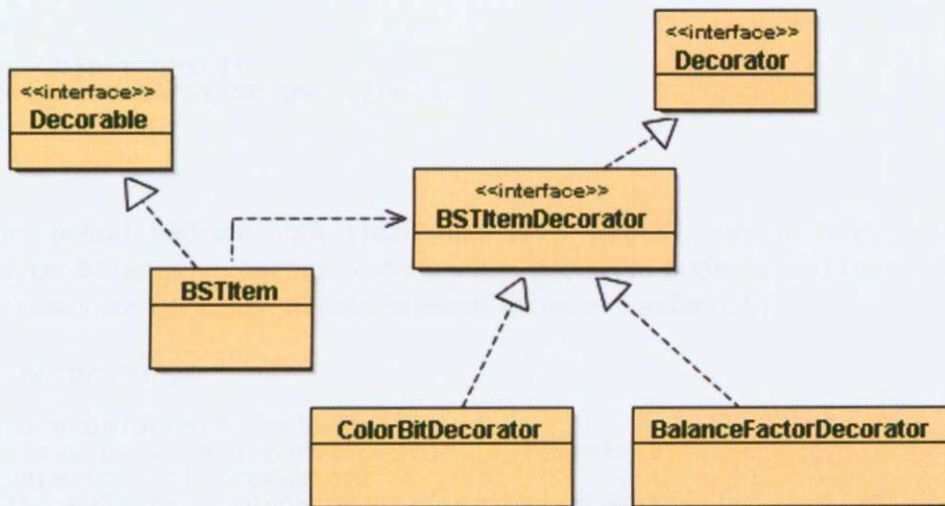
Někdy se totiž stane, že chcete datové struktury dodat určité vlastnosti, pro které by bylo zbytečné vytvořit nové podtřídy (z důvodů, které byly řečeny výše). Vemte si například uzel vyváženého binárního vyhledávacího stromu³². Ten můžeme chápat, jak jako uzel AVL stromu, tak i jako uzel červeno černého stromu. Z implementačního hlediska uzel AVL stromu a uzel červeno černého stromu mohou dědit od společné nadtřídy reprezentující uzel binárního vyhledávacího stromu (přesněji řečeno vyváženého). Jenže je to zbytečné, protože takto vytvořené podtřídy mají naprosto stejné vlastnosti (uchovávají a nastavují klíče a jejich hodnoty), až na jednu jedinou a tou je údaj, který nám řekne jestli daný uzel je vyvážený (v případě AVL stromu je to balanční faktor a v případě červeno černého stromu je to barevný bit (*color bit*)). Jiný způsob implementace nám nabídne právě návrhový vzor Dekorátor.

Řešení:

Ten spočívá v tom, že dodáme barevný bit nebo balanční faktor jako tzv. *dekoraci* uzlu vyváženého binárního vyhledávacího stromu (dále jen BVS). To provedeme tak, že přidáme datový prvek třídě uzlu BVS, který bude reprezentovat danou dekoraci (tzn. že může to být jak balanční faktor, pokud chceme zacházet s uzlem jako s uzlem AVL stromu, nebo barevný bit, který se používá pro vnitřní účely červeno černého stromu). Potřebujeme taky univerzální způsob jak měnit hodnotu této dekoraci, proto musí náš uzel implementovat rozhraní *Decorable*. Počáteční hodnotu této dekoraci určuje tzv. *dekorátor* „aranžér“. Dekorátorů je celá řada, co typ dekorace to dekorátor.

Struktura vzoru Dekorátor vypadá takto:

³² Přesněji řečeno položka (*item*) binárního vyhledávacího stromu. My tu totiž chápeme binární vyhledávací strom chápeme jako slovník, tudíž jeho uzly jsou položky slovníku (dvojice klíč-hodnota). Je ale pravda, že binární vyhledávací strom používá uzly instance třídy `LinkedBinaryTree` jako úložiště svých položek. Viz balík `trees`. My tu pro srozumitelnost budeme držet termínu *uzel*.



Kód uzlu BVS (třída BSTItem) vypadá takto:

```

package trees;
import decorator.*;

public class BSTItem implements Decorable{
    private Object key, elem;
    private BSTItemDecorator decoration;
    public BSTItem (Object k, Object e) {
        key = k;
        elem = e;
        decoration = null;
    }

    public BSTItem (Object k, Object e, BSTItemDecorator decoration){
        this(k,e);
        this.decoration = decoration;
    }

    public Object key() {return key; }
    public Object element() { return elem; }
    public void setKey(Object k) { key = k; }
    public void setElement(Object e) { elem = e; }

    //metody rozhraní Decorable
    public void create(Object decoration, Object value){
        this.decoration = (BSTItemDecorator) decoration;
        this.decoration.setValue(value);
    }

    public Object destroy(){
        Object toReturn = decoration.getValue();
        decoration = null;
        return toReturn;
    }
    public boolean has(){
        if(decoration != null)
            return true;
        else
            return false;
    }
    public void set(Object value){
        decoration.setValue(value);
    }
}
  
```

```

    public Object get(){
        return decoration.getValue();
    }
}

```

Jak vidíme, pokud budeme chtít vložit uzel BVS do AVL stromu, tak stačí mu dodat dekorátor typu `BalanceFactorDecorator`, jehož počáteční hodnota je 0 (nově vložený uzel je vždycky dokonale vyvážený, tj. jeho balanční faktor má hodnotu 0):

```

package decorator;

/**Dekorace balančním faktorem*/
public class BalanceFactorDecorator implements BSTItemDecorator{
    private int balanceFaktor;
    public BalanceFactorDecorator(int balanceFaktor){
        setValue(new Integer(balanceFaktor));
    }

    public void setValue(Object value){
        balanceFaktor = ((Integer)value).intValue();
    }
    public Object getValue(){
        return new Integer(balanceFaktor);
    }
}

```

Pokud budeme chtít změnit hodnotu balančního faktoru, zavoláme metodu `set(Object value)`. V ní hodnotu dekoraci nastavíme na novou hodnotu obsaženou v objektu `value`. Analogicky pokud budeme chtít zacházet s uzlem BVS jako s uzlem červeno černého stromu, tak dodáme jeho konstruktoru objekt typu `ColorBitDecorator`, který bude spravovat barevný bit uzlu.

Ostatní metody rozhraní `Decorable` mají tento význam:

- `create(Object decoration, Object value)`: vytvoří novou dekoraci a nastaví její počáteční hodnotu na `value`. Tato metoda má význam, pokud budeme chtít dynamicky měnit dekoraci uzlu BVS. To znamená, že můžeme použít objekt `BSTItem` jednou jako uzel AVL stromu a podruhé jako uzel červeno černého stromu. To potřebujeme málo kdy, ale obecně se hodí měnit datové prvky objektu dynamicky při běhu programu.
- `destroy()`: odstraní dekoraci. Jsou případy, kdy jednou objekt může mít dekoraci a jindy žádnou. U uzlu BVS nemá význam, protože ten vždy použijeme v nějakém konkrétním typu vyvážených binárních stromů. To znamená, že uzel BVS bude mít vždy nějakou dekoraci.
- `has()`: testuje, jestli objekt má nějakou dekoraci.
- `get()`: vrací hodnotu dekoraci.

Použití BVS uzlu podporujícího vzor Dekorátor v AVL stromech

Implementace AVL stromu (viz balík `trees`) je založena na myšlence, že položky stromu se uchovávají v uzlech binárního stromu `LinkedBinaryTree` jako jejich elementy.. Tudíž po vložení nové položky (dvojice klíč-hodnota) do stromu zavolá se metoda

```

T.replaceElement(zPos, new BSTItem(key, element,
    new BalanceFactorDecorator(0)));

```

Instance třídy `LinkedBinaryTree` `T` nahradí nově vloženou položku uloženou **na pozici** `zPos` (viz návrhový vzor `Pozice` na straně 15) novým uzlem, který již bude mít v sobě, který

je mu dodán jako dekorace. Hodnota balančního faktoru uzlu je nastavena na 0, neboť uzel se vkládá na konec stromu, tudíž je dokonale vyvážený.

Když nyní budeme chtít znovu vyvážit náš AVL strom, musíme nejdříve zaktualizovat hodnoty balančních faktorů v jednotlivých uzlech po cestě od nově vloženého uzlu až po kořeni (viz metoda AVL stromu `rebalance(Position zPos)`), abychom poznali, který uzel je nevyvážený a následně, kdyby tomu tak bylo, restrukturalizovali jeho podstrom. **Tudíž budeme aktualizovat hodnoty dekorací.** Tudíž metoda

```
private void setBalanceFactor(Position p) {
    int bf = T.height(T.leftChild(p)) - T.height(T.rightChild(p));
    BSTItem polozka = (BSTItem) p.getElement();
    polozka.set(new Integer(bf));
}
```

na posledním řádku volá metodu uzlu `BSTItem set(Object value)`, aby nastavila hodnotu jeho dekoraci (tj. hodnotu balančního faktoru) na novou hodnotu.

Vidíme, že ostatní metody rozhraní `Decorable` jsme nevyužili. Nepotřebovali jsme je: kromě nastavení počáteční hodnoty balančního faktoru a její průběžné aktualizace (metodou `set()`) jsme s balančním faktorem nedělali nic jiného. Nemůžeme například využít jeho hodnotu (třeba metodou `get()`) pro výpočet hodnot balančních faktorů jiných uzlů. Jejich hodnoty vždy počítáme na základě výšek dětí uzlů. Tady je na pováženou jaký je smysl využití dekorátorů u uzlů BVS. Možná si říkáte, že zrovna v tomto případě to není problém dědit od společného uzlu BVS. Nuže programovací jazyk je jako „lidský“ jazyk. Záleží na každém jaké vyjadřovací prostředky použije a které mu vyhovují.

Každopádně návrhový vzor má nepopiratelnou roli při implementaci grafů. Podívejte se do knihy *Data Structures and Algorithms in Java* od autorů Michael T. Goodrich a Roberto Tamassia strana 561.

Cvičení

1. Naimplementujte červenočerný strom, který bude využívat k implementaci uzlů třídu `BSTItem`, která bude dekorována barevným bitem.
2. Zkuste naimplementovat algoritmus depth-first search v grafu použitím dekorátoru.

7. Šablonová metoda

Název vzoru: Šablonová metoda – *Template Method*.

Problém:

Často potřebujeme zobecnit řešení (přesněji algoritmus řešení) konkrétního problému tak, aby se dalo použít na řešení budoucích problémů stejné podstaty. Potřebujeme tedy **znovupoužitelnou šablonu** pro řešení problémů určitého okruhu.

S potřebou zobecňování se setkáváme na každém rohu. Od návrhu grafických komponent, algoritmů pro modelování dat až po návrhu datových struktur. Například podstata implementace hranice komponent je stejná, ale liší se v detailech od hranici s titulkem až po obyčejné hranici v podobě rámu. Stejně tak, datové modely grafické komponenty tabulka mohou být různé, ale podstata je stejná: každý takový datový model musí umět jednu velice podstatnou věc: umět komunikovat s tabulkou.

Známe různé metody průchodu binárním stromem do hloubky. Jsou to inorder, postorder a preorder³³. Neexistuje způsob jak zobecnit tyto metody a tím vytvořit **jednu jedinou metodu**, která by reprezentovala obecný průchod binárním stromem do hloubky? Výhoda tohoto průchodu je v tom, že nám umožní vyjádřit mnoho obecných algoritmů jednoduchým způsobem.

Řešení:

Řešení jednoduše spočívá v tom, že musíme mít programové prostředky, které by nám toto zobecňování umožnily. Tyto programové prostředky jsou obsaženy v samotném jádře objektově orientovaných technologií. Kdykoli píšete rodičovskou třídu a necháváte implementaci jedné nebo více metod na odvozené třídě, tak vězte, že nevědomky používáte šablonovou metodu. Tato metoda neboli tento vzor zobstrakňuje myšlenku definice obecné kostry algoritmu (v podobě třídy) s tím, že detaily některých částí algoritmu nechává na starost odvozeným třídám. Jinými slovy, jestliže vaše rodičovská třída je abstraktní třída, která nechává těla některých metod prázdné s tím, že jejich implementaci nechává na podtřídy, tak se jedná o použití šablonové metody.

Java oplývá vzory typu šablonová metoda. Velký počet tříd mají počátek v hierarchii dědičnosti u abstraktní třídy, která má prázdné (abstraktní) metody, jejich implementace je ponechávána na podtřídy. Například třída `AbstractTableModel` definuje všechny potřebné metody potřebné pro implementaci datového modelu třídy `JTable`. `AbstractTableModel` implementuje všechny metody až na tři. Když vytváříte „datovou“ třídu `JTable`, tak musíte naimplementovat metody `getRowCount()`, `getColumnCount()` a `getValueAt(int row, int column)`. Podívejte se na implementaci aplikace **Diář**.

Nás, ale bude především zajímat použití šablonové metody v návrhu datových struktur. Konkrétně odpověď na otázku, jak zobecnit průchod do hloubky binárním stromem použitím šablonové metody.

Jak jsme již řekli, my můžeme použít tři různé metody průchodu stromem: inorder, preorder a postorder. Tyto průchody můžeme skloubit do **jednoho jediného průchodu** a to následujícím způsobem:

³³ **Preorder:** nejdříve navštívíme kořen v stromu, a potom podstromy s kořeny v dětech uzlu v procházíme zleva doprava. **Inorder:** navštívujeme uzel mezi rekurzivními voláními obou jeho dvou postromů. **Postorder:** opak preorderu: rekurzivně prochází (zleva doprava) nejdříve podstromy s kořeny v dětech hlavního kořene a pak samotný kořen. Zatímco preorder a postorder jsou průchody každým stromem, tak inorder je specialita binárního stromu.

- „Nalevo“– preorder
- „Zezdola“– inorder.
- „Napravo“–postorder.

```

graph TD
    Root((-)) --> Div(/)
    Root --> Plus1(+)
    Div --> Mult1(×)
    Div --> Plus2(+)
    Mult1 --> Plus3(+)
    Mult1 --> Leaf3[3]
    Plus3 --> Leaf3_1[3]
    Plus3 --> Leaf3_2[1]
    Plus2 --> Minus1(-)
    Plus2 --> Leaf2[2]
    Minus1 --> Leaf9[9]
    Minus1 --> Leaf5[5]
    Plus1 --> Mult2(×)
    Plus1 --> Leaf6[6]
    Mult2 --> Leaf3_3[3]
    Mult2 --> Minus2(-)
    Minus2 --> Leaf7[7]
    Minus2 --> Leaf4[4]
  
```

Toto je průchod o kterém jsme mluvili. Když se na ně podíváte pozorně, tak zpozorujete, že opravdu v sobě slučuje všechny průchody: preorder (cesta podél stromu vlevo), inorder (návštěva každého uzlu zezdola, tj. na rozhraní dvou podstromů) a postorder (cesta podél stromu vpravo). Tento průchod se nazývá **Eulerův průchod**, přičemž ostatní průchody jsou jeho speciální případy. Eulerův průchod si můžeme také představit jako „chůze“ kolem stromu. Chůze, která začíná u kořene stromu a končí u něj.

```

vykonej akci při návštěvě uzlu  $v$  nalevo
if  $v$  je interní uzel then
    rekurzivně procházej levý podstrom uzlu  $v$  voláním eulerTour( $T$ ,  $T.leftChild(v)$ )
vykonej akci při návštěvě uzlu  $v$  zezdola
if  $v$  je interní uzel then
    rekurzivně procházej pravý podstrom uzlu  $v$  voláním eulerTour( $T$ ,  $T.rightChild(v)$ )
vykonej akci při návštěvě uzlu  $v$  napravo

```

56

takového stromu vidíte na horním obrázku³⁴ (viz tamtéž na straně 232). Algoritmus `printExpression` to dělá následujícími kroky použitím Eulerova průchodu:

- akce „nalevo“: jestliže tento uzel je interní, vytiskni levou závorku „(“.
- akce „zezdola“: vytiskni hodnotu nebo operátor uložený v tomto uzlu.
- akce „napravo“: jestliže tento uzel je interní, vytiskni pravou závorku „)“.

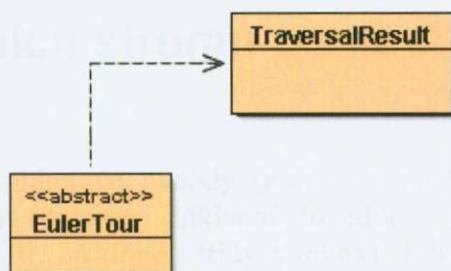
Eulerův průchod není nic jiného než aplikace návrhového vzoru **Šablonová metoda**, kde tento průchod je **obecnou šablonou** pro jeho konkrétní podoby v podtřídách, ve kterých můžeme naimplementovat například speciálně inorder nebo postorder průchod nebo kombinace těchto průchodů. Tyto „budoucí“ algoritmy se mohou samozřejmě také lišit akcí vykonanou u jednotlivých uzlů.

My navrhujeme algoritmus `eulerTour()` (zapouzdřený ve třídě stejného názvu), který bude implementovat obecný Eulerův průchod binárním stromem. Když voláme tuto metodu na uzel `v`, metoda `eulerTour()` volá pomocnou metodu `visitExternal()`, jinak (jestliže `v` je interní uzel) `eulerTour()` provede následující kroky:

- Volá pomocnou metodu `visitLeft()`, která provede výpočet spojený s návštěvou uzlu nalevo.
- Rekurzivně volá sebe na levé dítě.
- Volá pomocnou metodu `visitBelow()`, která provede výpočet spojený s návštěvou uzlu zezdola.
- Rekurzivně volá sebe na pravý podstrom.
- Volá pomocnou metodu `visitRight()`, která provede výpočet spojený s návštěvou uzlu napravo.

Nakonec `eulerTour()` vrací výsledek výpočtu voláním metody `result(TraversalResult r)`. Těla metod uvedených výše jsou prázdné, protože jejich povaha, tedy implementace bude záviset na podtřídě (konkrétní aplikaci) implementující náš průchod. Tyto metody jsou tedy abstraktní.

Návrhový vzor Šablonová metoda v případě stromů se tedy skládá z těchto prvků:



Třída `TraversalResult` reprezentuje výsledek výpočtu průchodu.

Tady je konečně implementace třídy Eulerova průchodu, jehož detaily implementace (implementace metod `visitLeft()`, `visitBelow()`, `visitRight()` a `visitExternal()`) budou záviset na konkrétní implementaci:

```
package trees.euler;
import hierarchy.*;
import position.*;
```

³⁴ Strom na obrázku **Eulerův průchod** reprezentuje výraz $(((((3+1)3)/((9-5)+2))-((3(7-4))+6)))$

```

/**
 * Šablona pro algoritmy procházející binární strom použitím Eulerova
 * průchodu. Podtřídy této třídy budou implementovat některé
 * metody této třídy pro vytvoření určitého průchodu.
 */
public abstract class EulerTour {
    protected BinaryTree tree;
    public Object execute(BinaryTree T) {
        tree = T;
        return null; // nic zajímavého ke vracení
    }

    protected Object eulerTour(Position p) {
        TraversalResult r = initResult();
        if (tree.isExternal(p)) {
            visitExternal(p, r);
        } else {
            visitLeft(p, r);
            r.leftResult = eulerTour(tree.leftChild(p));
            visitBelow(p, r);
            r.rightResult = eulerTour(tree.rightChild(p));
            visitRight(p, r);
        }
        return result(r);
    }
    //metody, které mohou být naimplementovány podtřídami
    protected void visitExternal(Position p, TraversalResult r) {}
    protected void visitLeft(Position p, TraversalResult r) {}
    protected void visitBelow(Position p, TraversalResult r) {}
    protected void visitRight(Position p, TraversalResult r) {}
    protected TraversalResult initResult() { return new TraversalResult(); }
    protected Object result(TraversalResult r) { return r.finalResult; }
}

```

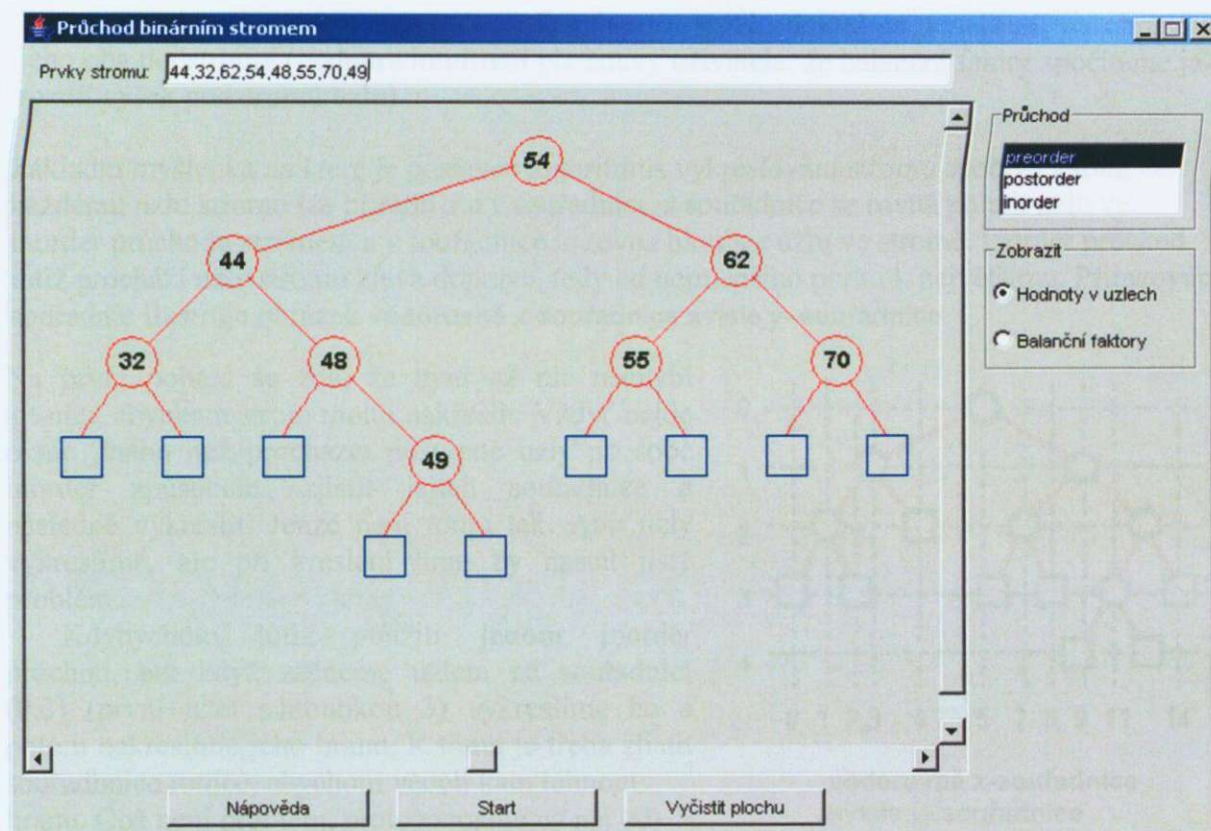
Výsledek výpočtu průchodu se průběžně ukládá do objektů, které reprezentují výsledek po návštěvě pravého podstromu (`r.rightResult`), návštěvě levého podstromu (`r.leftResult`) a finální výsledek (`r.finalResult`).

My si nyní ukážeme konkrétní implementaci Eulerova průchodu.

Kreslení binárních stromů pomocí Eulerova průchodu

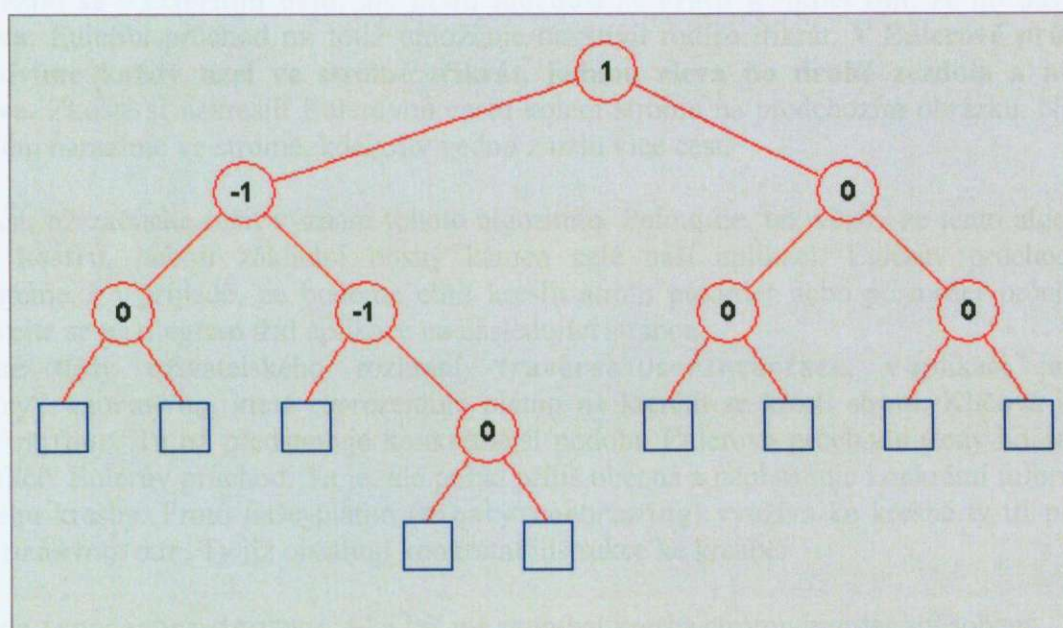
My si ukážeme využití šablonové metody (tedy konkrétní využití Eulerova průchodu) v aplikaci **Průchod binárním stromem**. Základní dovedností této aplikace je, že umí kreslit binární stromy na plátně (tj. na objekt třídy `Canvas`). Přičemž vstupními daty pro tvorbu těchto stromů jsou čísla, které se ukládají do stromu uspořádaným způsobem a výsledný strom je vždy vyvážený. Bude se tedy jednat o vyvážené vyhledávací stromy, konkrétně AVL stromy. Aplikace má především vzdělávací účel. Její hlavní účel je totiž demonstrovat na plátně různé průchody binárním stromem. Eulerův průchod ne. Ten se využívá pro potřeby implementace aplikace. Aplikace umí navíc zobrazit balanční faktory uzlů a může sloužit jako simulace vkládání a odebírání uzlu z AVL stromu. Grafické uživatelské rozhraní aplikace je na následujícím obrázku **GUI aplikace Průchod binárním stromem**.

Jak vidíte strom ukládá data, které uživatel zadá do vstupního textového pole nahoře. Přičemž uživatel si zvolí v levé části okna aplikace, že chce nakreslit strom preorder průchodem a že chce zobrazit číselná data uložená v uzlech. Aplikace skutečně (vzhledem k volbě preorder) kreslí strom zezhora a to dostatečně pomalu, aby uživatel viděl jak vypadá preorder průchod stromem. Kdyby zadal průchod inorder, tak aplikace začne kreslit strom



GUI aplikace Průchod binárním stromem

zezdola postupně uzel po uzlu, hranu po hraně, takže uživatel dobře vidí, jak se nejdříve vykreslí levý externí uzel, pak jeho rodič potom pravý potomek rodiče. Pokud by uživatel zaškrtnl checkbox „**Balanční faktory**“, tak aplikace pro ty samé vstupní data zobrazí v uzlech jejich balanční faktory:



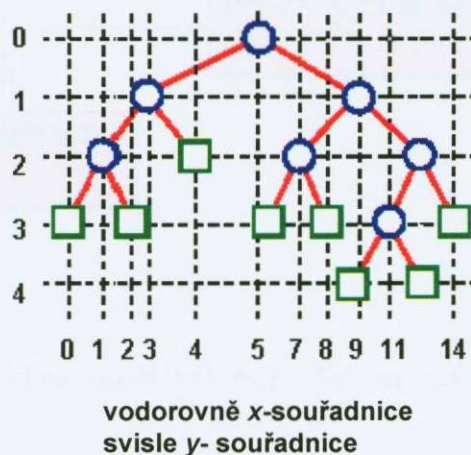
Balanční faktory

Přičemž se nezobrazí hodnota balančního faktoru v uzlu dokud se nenakreslí na obrazovce jeho oba podstromy (podpora intuitivní představy uživatele, že balanční faktor spočítáme jako rozdíl výšek podstromů uzlu).

Základní myšlenka na které je postaven algoritmus vykreslování stromu spočívá v tom, že každému uzlu stromu lze přiřadit x a y souřadnici. x souřadnice se rovná pořadí uzlu ve inorder průchodu stromem a y souřadnice je rovna hloubce uzlu ve stromě. Inorder průchod totiž prochází uzly stromu zleva doprava, tedy od nejmenšího prvku k největšímu. Přiřazování souřadnic ilustruje obrázek **vodorovně x-souřadnice svisle y- souřadnice**

Na první pohled se zdá, že nyní už nic nechybí k tomu, abychom strom mohli nakreslit. Vždyť nejde o nic jiného než procházet postupně uzly po sobě inorder způsobem, zjistit jejich souřadnice a následně vykreslit. Jenže není tomu tak. Ano uzly vykreslíme, ale při kreslení hran by nastal jistý problém.

Kdybychom totiž použili **jenom** inorder průchod, tak když začneme uzlem na souřadnici (0,3) (první uzel s hloubkou 3) vykreslíme ho a potom nakreslíme jeho hranu. K tomu je třeba zjistit souřadnice rodiče, abychom věděli kam táhnout hranu. Což není problém, protože máme na něj od-



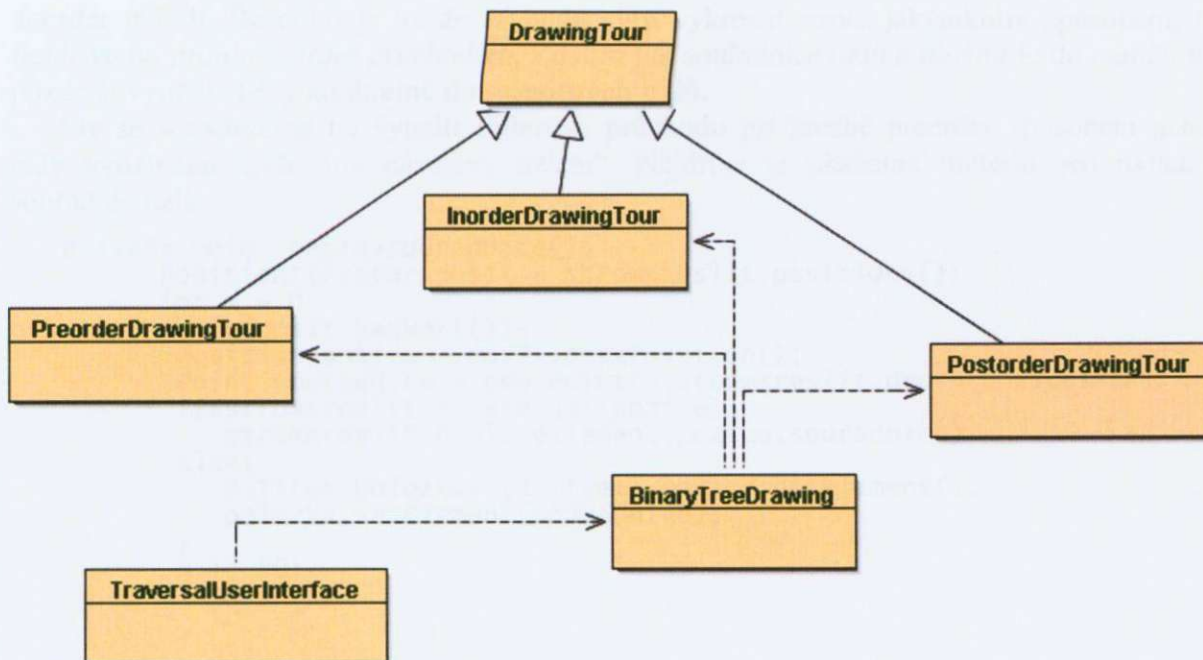
kaz (jedná se o obousměrnou datovou strukturu). Nyní navštívíme rodiče na pozici (1,2), vykreslíme ho, pak hranu na následující externí uzel. Ten též vykreslíme. Jenže co dál? Měli bychom vykreslit hranu do uzlu na pozici (3,1), jenže přímý odkaz z externího uzlu ve kterém se nyní nacházíme do uzlu (3,1) nemáme. Takže bychom se potřebovali vrátit do uzlu od kterého jsme přišli (tj. na pozici (2,1)), **jenže inorder průchod to nám neumožňuje**. Jistě existují různé „klíčky“ jak to obejít, ale bylo by to složité, nepřehledné a neefektivní.

Tady přichází ke slovu Eulerův průchod. Ten nám nabídne elegantní řešení: **Ano nacházím se v externím uzlu, ale mám možnost se vrátit k rodiči tím, že ho navštívím zprava**. Eulerův průchod mi totiž umožňuje navštívit rodiče třikrát. **V Eulerově průchodu navštívíme každý uzel ve stromě třikrát, jednou zleva po druhé zezdola a po třetí zprava**. Zkuste si nakreslit Eulerovou cestu kolem stromu na předchozím obrázku. Na tento problém narazíme ve stromě, kdekoli vedou z uzlu více cest.

Asi, už začínáte tušit význam tohoto algoritmu. Pokud ne, tak vezte, že tento algoritmus tvoří **kostru**, neboli základní nosný kámen celé naší aplikace. Eulerův průchod totiž použijeme, i v případě, že budeme chtít kreslit strom preorder nebo postorder průchodem. Podívejte se na diagram tříd aplikace na následující stránce.

Kromě třídy uživatelského rozhraní `TraversalUserInterface`, v aplikaci je třída `BinaryTreeDrawing`, která reprezentuje plátno na kterém se kreslí strom. Klíčová je třída `DrawingTour`. Ta už představuje konkrétnější podobu Eulerova průchodu (tedy ho dědí), tj. „kreslící“ Eulerův průchod. Ta je, ale pořád příliš obecná a neobsahuje konkrétní informace o průběhu kresby. Proto naše plátno (`BinaryTreeDrawing`) využívá ke kresbě ty tři podtřídy třídy `DrawingTour`. Ty již obsahují konkrétní instrukce ke kresbě:

- Třída `InorderDrawingTour`, říká jak má probíhat kresba stromu inorder způsobem.
- Třída `PreorderDrawingTour`, říká jak má probíhat kresba stromu preorder způsobem.
- Třída `PostorderDrawingTour`, říká jak má probíhat kresba stromu postorder způsobem.



Když uživatel vybere ze seznamu, průchod kterým chce strom vykreslit. Tak se zavolá metoda plátna:

```

public void paint(Graphics g){
    if(!stromKreslitStr.equals("")){
        switch(traversal){
            case PREORDER:
                new PreorderDrawingTour(stromKreslit,naplnituzly,
                    posunX,pauzaUsecka,pauzaBalance,g);
                break;
            case INORDER:
                new InorderDrawingTour(stromKreslit,naplnituzly,
                    posunX,pauzaUsecka,pauzaBalance,g);
                break;
            case POSTORDER:
                new PostorderDrawingTour(stromKreslit,naplnituzly,
                    posunX,pauzaUsecka,pauzaBalance,g);
                break;
        }
        pauzaUsecka = 0;
        pauzaBalance = 0;
    }
}

```

Po výběru způsobu procházení se uloží do proměnné `traversal` celočíselná hodnota reprezentující způsob jakým se má vykreslit strom. Ta se testuje v příkazu `switch` a v závislosti na její hodnotě plátno spustí příslušný algoritmus kresby.

Každý algoritmus tedy spustí kresbu, využívající konkrétní způsob procházení stromu (inorder, postorder nebo preorder). Všechny tyto algoritmy využívají ke kreslení Eulerův průchod, který u každého algoritmu vypadá trochu jinak, ale jeho myšlenka je stejná (viz předchozí strana).

Již jsme naznačili jak by vypadala kresba využitím Eulerova průchodu a nyní to popíšeme v detailech. Ale nejdříve jedna poznámka:

Způsob vykreslování stromu inorder průchodem je poměrně jednoduchý neboť, okamžitě po zjištění souřadnice uzlu nakreslíme. Ale co když budu procházet strom preorder způsobem? Souřadnice uzlů zjišťujeme inorder způsobem, tedy když začnu procházet strom preorderem počínaje kořenem, tak neznám jeho souřadnici, nutnou pro vykreslování, protože neznám jeho

inorder pořadí. Řešením je to, že ať budu chtít vykreslit strom jakýmkoliv způsobem, tak nejdříve ho projdu inorder průchodem. Zjistím tak souřadnice uzlů a uložíím je do paměti pro pozdější využití. Tedy konkrétně **do samotných uzlů**.

My se soustředíme na využití Eulerova průchodu při kresbě preorder způsobem neboť tady využíváme „všechny návštěvy uzlem“. Nejdříve si ukážeme metodu pro ukládání souřadnic uzlů:

```
private void nastavSouadnice(){
    PositionIterator posIt = stromKreslit.positions();
    int x = 0;
    while(posIt.hasNext()){
        Position pozice = posIt.nextPosition();
        Point souradnice = new Point(x,stromKreslit.depth(pozice)*60);
        if(stromKreslit.isExternal(pozice))
            stromKreslit.replaceElement(pozice,souradnice);
        else{
            BSTItem polozka= (BSTItem) pozice.getElement();
            polozka.setElement(souradnice);
        }
        x += 30;
    }
}
```

Tuto metodu volá samotné plátno, které tak nastavuje souřadnice budoucího nakresleného stromu. Vracený iterátor je výsledkem inorder průchodu stromem, tudíž řadí prvky stromu od nejmenšího k největšímu. Po získání iterátoru již probíhá proces nastavování souřadnic. Všimněte si, že souřadnice ukládám i do externích uzlů.

Po nastavování souřadnic již se volá metoda paint(Graphics g), která vytvoří objekt typu PreorderDrawingTour, kód této třídy vypadá takto:

```
public class PreorderDrawingTour extends DrawingTour{
    public PreorderDrawingTour(BinaryTree strom,String napln,int x,long
                                pauzaU,long pauzaB,Graphics g){
        stromKreslit = strom;
        data = napln;
        posunX = x;
        pauzaUsecka = pauzaU;
        pauzaBalance = pauzaB;
        grafickyObjekt = (Graphics2D) g;
        fontDat = new Font("SansSerif",Font.BOLD,14);
        grafickyObjekt.setFont(fontDat);
        execute(stromKreslit);
    }

    public Object execute(BinaryTree T) {
        super.execute(T); // calls method of superclass
        eulerTour(tree.root());
        return null; // nothing interesting to return
    }

    protected void visitExternal(Position p, TraversalResult r) {
        Point souradnice = (Point) p.getElement();
        grafickyObjekt.setColor(Color.BLUE);
        grafickyObjekt.draw(new Rectangle2D.Double(souradnice.x+posunX-
            12,souradnice.y+20,24,24));
    }

    protected void visitLeft(Position p, TraversalResult r) {
        BSTItem item = (BSTItem) p.getElement();
        Point stredUzlu = (Point) item.element();
        grafickyObjekt.setColor(Color.RED);
        grafickyObjekt.draw(new Ellipse2D.Double(stredUzlu.x+posunX-
            15,stredUzlu.y+20,30,30));
        if(data.equals("Hodnoty v uzlech"))
            naplnUzel(item.key().toString(),stredUzlu);

        Position levy = tree.leftChild(p);
        boolean external; Point stredNaslednika; int y;
        if(tree.isExternal(levy)){
            external = true;
            stredNaslednika = (Point)levy.getElement();
        }
    }
}
```

```

        y = stredNaslednika.y+20;
    }else{
        external = false;
        item = (BSTItem) levy.getElement();
        stredNaslednika = (Point) item.element();
        y = stredNaslednika.y+35;
    }
    spojUzly(external,new Point(stredUzlu.x+posunX,stredUzlu.y+35),
        new Point(stredNaslednika.x+posunX,y));
}

protected void visitBelow(Position p, TraversalResult r) {
    BSTItem item = (BSTItem) p.getElement();
    Point stredUzlu = (Point) item.element();
    Position pravy = tree.rightChild(p);
    boolean external; Point stredNaslednika; int y;
    if(tree.isExternal(pravy)){
        external = true;
        stredNaslednika = (Point)pravy.getElement();
        y = stredNaslednika.y+20;
    }else{
        external = false;
        item = (BSTItem) pravy.getElement();
        stredNaslednika = (Point) item.element();
        y = stredNaslednika.y+35;
    }
    spojUzly(external,new Point(stredUzlu.x+posunX,stredUzlu.y+35),
        new Point(stredNaslednika.x+posunX,y));
}

protected void visitRight(Position p, TraversalResult r) {
    if(data.equals("Balanční faktory")){
        BSTItem item = (BSTItem) p.getElement();
        Point stredUzlu = (Point) item.element();
        stopka(pauzaBalance);
        naplnUzel(item.get().toString(),stredUzlu);
    }
}
}

```

Vidíme, že tento algoritmus skutečně využívá Eulerův průchod. Implementuje jeho metody, což znamená, že určuje co se má dít po každé návštěvě uzlu a to z jeho tří stran. Při Eulerově průchodu, tj. cestě kolem stromu, začne algoritmus navštěvovat zleva (preorder), tudíž protože chceme vykreslit strom tímto způsobem tak začne tyto uzly okamžitě vykreslovat i s jejich prvky a i hrany, které kopírují cestu. Když navštíví uzel zezdola (inorder) tak pokud to není externí uzel tak vykreslí jeho druhou hranu, kterou předtím nenakreslil.

A konečně pokud uživatel bude chtít zobrazit balanční faktory, tak se využije návštěva zprava (postorder) k vepsání hodnot balančních faktorů. Je přirozené, že hodnotu balančního faktoru vepíšu až tehdy, když „už znám“ výšku dětí. A toto je přesně co postorder průchod umožňuje: ten nejdříve navštěvuje děti a pak jejich rodiče. Tak vidíme, že díky Eulerově průchodu dokážeme vykreslit strom a přitom zobrazit jeho údaje věrohodně. To znamená, že ať procházím strom inorder nebo preorder způsobem, tak vždycky můžu s tím **současně** zobrazit hodnoty balančních faktorů **až po** zobrazení dětí, což mi umožňuje jedinečně postorder průchod. A právě jedinečně Eulerův průchod mi umožní uplatnit tento průchod s jinými průchody najednou. Takže jeho hlavní zásluha spočívá v tom, že se mohou vykonat více akcí společně, které zdánlivě nemohou fungovat současně.

Cvičení

1. Jak by se dal použít Eulerův průchod pro tisk aritmetického výrazu? Napište program, který by pracoval s aritmetickými výrazy a využíval by k tomu Eulerův průchod. Aritmetický výraz je uložen tedy v binárním stromu.
2. Zkuste naprogramovat aplikaci (**Průchod binárním stromem**) použitím vzoru Strategie (Strategy).

PŘÍLOHY

Příloha CD

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

Priloha CD je součástí knihy a obsahuje:

PŘÍLOHY

Přiložené CD

- **Aplikace:** obsahuje spustitelné soubory aplikací.
Práce obsahuje tyto aplikace:
 - ❑ **Postfixový kalkulátor:** počítá aritmetické výrazy zadané ze vstupu v infixové notaci. Vypíše jejich postfixovou notaci. Aplikace používá návrhový vzor **Adaptér**.
 - ❑ **Telefonní seznam:** slouží pro jednoduchou správu telefonních kontaktů. Používá vzor **Dekorátor**.
 - ❑ **Diář:** jednoduchý organizér pro plánování termínů. Používá vzor **Lokátor**.
 - ❑ **Průchod binárním stromem:** výuková aplikace, která demonstruje všechny tři způsoby průchodu binárním stromem do hloubky (inorder, postorder a preorder). Používá vzor **Šablonová metoda**.
- **Zdrojáky:** obsahuje zdrojové kódy aplikací.
- **Datové struktury:** obsahuje zdrojové kódy všech datových struktur a návrhových vzorů použitých v práci.
- **Pomocné:** obsahuje pomocné třídy použité pro implementace aplikací uvedených výše. Například třída `JxFrame` v adresáři `Look&Feel` definuje vzhled aplikací. Přizpůsobuje ho operačnímu systému instalovaného na počítači ve kterém se použít aplikace.

Literatura

- [1] MICHAEL T. GOODRICH, ROBERTO TAMASSIA: Data Structures and Algorithms in Java™ second edition, John Wiley & Sons Inc., 2001
- [2] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES: Návrh programů pomocí vzorů Stavební kameny objektově orientovaných programů, Grada, 2003
- [3] JAMES W. COOPER: Java™ Design patterns A tutorial, Addison – Wesley Inc., 2000
- [4] PAVEL TÖPFER: Algoritmy a programovací techniky, Prometheus, 1995