# IMPLEMENTACE INFORMAČNÍHO SYSTÉMU VIDIUM V JAZYCE JAVA

**Michal Brožek, Dominik Svěch, Jaroslav Štefaník**

MediumSoft a.s., Cihelní 14, 702 00 Ostrava, ČR

## Abstrakt

Autoři v článku popisují postupy použité při implementaci vizualizačního systému v návaznosti na objektově orientovaný jazyk JAVA. Zdůvodňují volbu tohoto jazyka a na praktických ukázkách fragmentů kódu ukazují výhody řešení a specifické vlastnosti jazyka JAVA. V hlubším pohledu bude proveden rozbor jednotlivých částí systému a jejich vzájemné vazby. Obsahem popisu bude pohled jazyka JAVA na zpracování dat a událostí, komunikace pomocí TCT/IP i využití a práce s vlákny.

**Klíčová slova:** JAVA, VIDIUM, komunikace, vlákna.

## 1. Java

Sun charakterizuje jazyk Java následujícím způsobem: "Jednoduchý, objektově orientovaný, distribuovaný, interpretovaný, robustní, bezpečný, nezávislý na architektuře, přenosný, vysoce výkonný, víceprocesní a dynamický jazyk [1]. Jednotlivě jsou tyto formální pojmy jsou rozebrány v uvedené literatuře nebo také v [2]. Protože systém VIDIUM je založen hlavně na internet/intranet technologii, je Java zajímavá možností vytváření appletů, tedy "programů", které jsou prováděny přímo prohlížečem ze zvolené HTML stránky. Proto byla JAVA zvolena jako implementační jazyk.

## 2. Vidium

Systém VIDIUM slouží pro prezentaci měřených a agregovaných technologických údajů, jež jsou zpracovávány a uchovávány v datového skladu. Obecně jde tedy o komplexní informační systém, který zahrnuje sběr uchování, analýzu a vizualizaci dat. Technicky je tento systém založen na internetovských (intranetovských) technologiích s podpůrnými programy pro komunikaci, sledování a řízení systému. Nejedná se o řízení vlastních technologií. Základní myšlenkou je co největší využití možností globalizace a standardizace, jež se v oblasti vývoje software nabízí. Podporována je myšlenka použití takových programů a programových nástrojů, jež zákazník má standardně k dispozici již z dřívějších nákupů softwarových balíků, případně zdarma jako součást operačního systému. Ve výsledku to znamená omezení nákupů licencí na co největší možnou míru případně vůbec. Z tohoto hlediska se pak jeví jako výhodné, má-li již nyní zákazník prohlížeč Microsoft Internet Explorer (MSIE) a pro editaci stránek HTML již existují další nástroje, jako FrontPage nebo Word97. Tento způsob řešení omezuje množství programátorských nástrojů věnovaných na nestandardní řešení a zároveň dává zákazníkovi možnosti pro pozdější ovlivňování vzhledu prezentace údajů dle aktuálních požadavků. Není třeba pro každý nový údaj programovat zvlášť nějaký zobrazovací panel, stačí mít MSIE, umět vytvořit HTML stránku a dodržet určitá programátorská pravidla pro připojení k datovému serveru a pro přenos dat.

Konečná vizualizace systému pro uživatele probíhá v prostředí MSIE. To znamená, že celý systém je rozdělen na příslušné HTML stránky s určitou hierarchií. Statické informace HTML stránek jsou podporovány Java applety. Java applet zajišťuje komunikaci s procesy na datovém skladu, komunikaci s uživatelem a tím i oživení vlastní HTML stránky [3].

## 3. Komunikace

Komunikace s modulem prezentační části skladu je prováděna pomocí TCP/IP komunikace. Klientská část (applet) nejprve vytvoří spojení na databázový modul a zajistí ověření práv a přístupu na základě uživatelského jména a hesla. Po navázání spojení s modulem skladu jsou vytvořeny ”streamy” pro komunikaci. Prvním je písař (writer). Tento stream je zapouzdřen v objektu *StreamWriter* a poskytuje metody pro formátování a odesílání dat nebo požadavků. Druhý stream je čtenář (listener). Ten má za úkol příjem příchozích dat a jejich dekódování. Tento je prezentován objektem *StreamListener*. Aby bylo zaručeno, že všechny příchozí zprávy budou zpracovány, je objekt *StreamListener* obsluhován nezávislým vláknem, které má nastavenu vyšší prioritu. Pokud klient požaduje data, pak zadá požadavek třídě *spojení*, která využívá služeb v třídě *StreamListener*. Tyto požadavky jsou zasílány většinou synchronně. To znamená, že vykonávání hlavního klientského vlákna je zastaveno po dobu než je požadavek odeslán do databázového modulu. Aby databázový modul rozpoznal o jaká data se jedná, je stanoven pro každý požadavek kód metody, která požadavek zasílá. Množina těchto metod je omezena a plně pokrývá spektrum požadavků. Databázový modul tedy ”ví”, která metoda požadavek vyslala a pro každou takovou metodu existuje protějšek ve formě procedur nebo SQL dotazů. V databázovém modulu je požadavek dekódován, splněn a výsledek odeslán zpět klientské části (appletu). Zde zpracování pokračuje podle toho, o jaký požadavek se jednalo. Existují dvě kategorie požadavků:

1) synchronní – metoda čeká na odpověď, jedná se výlučně o tzv. konfigurační metody. Tyto metody mají za úkol poskytovat ostatním objektům data potřebná pro jejich konfiguraci, vytváření, nebo jsou to data nutná pro další korektní funkci těchto objektů,
2) asynchronní – jsou výlučně ty požadavky, jejichž splnění není vázáno na vytváření objektu nebo jeho konfiguraci. Typickým příkladem jsou například prezentované agregace, respektive hodnoty těchto agregací ve zvoleném časovém rozmezí. Odpověď není v tomto případě zasílána zpět volající metodě, ale jsou zasílána objektu *DataStore*, kde jsou uchována pro pozdější použití. (pozn. Volající metoda, je po odeslání požadavku opuštěna a k přerušení hlavního klientského vlákna nedojde).

Proč toto rozdělení? Na následujícím příkladu si to vysvětlíme. Pokud vytvářím graf z hodnot agregací, pak potřebuji vědět jak má tento graf vypadat, tj. potřebuji znát rozměry, barvy, popisy a umístění os, značky os, kategorie agregací nakonec i jméno grafu a další. Takže tyto údaje potřebuji, aby bylo možné graf sestrojit. Je tedy logické, že pokud klient zašle požadavek na tyto data, pak je potřebuje ještě před tím, než začne objekt graf vytvářet. Zatímco graf je konstruován potřebuji i hodnoty, které budou v grafu nakonec vyneseny. Protože těchto hodnot je mnoho a jejich zpracování modulem skladu může být zdlouhavé, je zaslán požadavek na zvolené hodnoty již na začátku konfigurace, či vytváření objektu. Tento požadavek je asynchronní, takže se nečeká na návrat dat z databázového modulu, ale pokračuje se normálně v činnosti až do fáze, kdy jsou tato data vyžadována – zde konkrétně pro vykreslení hodnot grafu. Jakmile jsou požadována data, provádí se dotaz na *DataStore*, případně sám *DataStore* oznámí přítomnost nových dat všem zájemcům. V systému VIDIUM je upřednostňován spíše způsob oznámení nových dat. (pozn. Databázový modul je umístěn na serveru databázového skladu a je tvořen několika vlákny, která zpracovávají požadavky jednotlivých klientů)

Následující fragmenty ukazují, jak je komunikace implementována a jak jsou zadávány požadavky. V souvislosti s *DataStore* je nutné zmínit rozhranní *DataAcceptor*, které musí každý objekt požadující asynchronní data implementovat. Rozhranní *DataAcceptor* obsahuje dvě metody. První metoda slouží pro identifikaci příslušného objektu a tím je určena i identifikace jemu náležejících dat. Druhá metoda, kterou je nutné implementovat, je *updateData*. Tato metoda je používána pro předávání nových dat k jejich dalšímu zpracování. Uvedené rozhranní je vypsáno v Př. 1.

```
public interface DataAcceptor {
  public int getId();
  public void updateData(Data buff);
}
```

Př. 1 - DataAcceptor

Klient pro vytvoření spojení na databázový server používá třídu *Socket*. Jeden z konstruktorů této třídy je využíván i v následujícím příkladu (Př. 2), kde je zadáno jméno počítače a port, na který se má klient připojit. Protože bezpečnostní politika appletu nedovoluje připojit se na libovolný server, ale pouze na ten, ze kterého byl

applet spuštěn, je adresa zjištěna z bázové adresy appletu a tato je pak převedena na jméno počítače. Port na kterém databázový modul poslouchá musí být samozřejmě známý předem.

```
Public EstablishConnection(int port) {
  iPort= port;
  notEstablished= true;
  int pocet = 5;
  while(notEstablished) {
    try {
      notEstablished=false;
      socket = new Socket( myApplet.getCodeBase().getHost(), iPort);
    }
    catch (IOException e) {
      notEstablished = true;
      if (pocet<0) break;
      try {Thread.sleep(2000); } catch (InterruptedException ie) {; }
    }
    catch (SecurityException ex) {
      System.out.println( "Bezpecnostni problem. "+ex.printStackTrace());
      NotEstablished = true;
    }
  }
  if (notEstablished) {
    Globals.printerr("Nepodarilo se pripojit na databazovy modul");
    Return;
  }
  localPort = socket.getLocalPort();

  //urcim streamy
  try {
    in = socket.getInputStream();
    out = socket.getOutputStream();
  }
  catch (IOException e) {
    System.out.println("Chyba getStream. "+ e.printStackTrace());
  }
}
```

Př. 2 – Připojení na databázový modul

V tomto příkladu (Př. 2) je rovněž ukázáno použití dvou metod pro vytvoření streamů, které jsou určené pro přenos dat. Tyto streamy jsou objekty, které poskytují pouze základní operace s daty, tj. čtení resp. zápis dat.

V následujícím příkladu (Př. 3) je ukázáno jak je nově vytvořený objekt *EstablishConnection* využit pro inicializaci již zmiňovaných objektů *StreamListener* a *StreamWriter*. Oba objekty mají za úkol data kódovat pro přenos a odeslat resp. přijmout a dekódovat.

```
Public Connection(int port) {
  //pripojeni na server
  conn = new EstablishConnection(port);

  if ( !conn.spojeno()) {
    Globals.showStatus("Nepripojeno na server");
    Dialogs d= new Dialogs(new DialogAdapter(), new Frame(), false, "Chyba!",
                                  "Nepodařilo se připojit na server!",
                                  Dialogs.ICON_CRITICAL,                    Buttons.BTN_OK+
Buttons.BTN_HELP);
    Return;
  }
  //napojeni vsech akceptoru na stream listenera a jeho vytvoreni
  dataStore = new DataStore(this);

  streamListener = new StreamListener(conn.in(),dataStore);
  streamWriter = new StreamWriter(conn.out());
  spojeni = new Spojeni(this);
}
```

Př. 3 – vytvoření streamu

Pokud jsou veškeré inicializace v pořádku, pak můžu zadávat požadavky na databázový modul. Typickým příkladem je použití objektu spojení. Tento objekt poskytuje základní metody pro dotazy a požadavky na databázový modul. Jedním z požadavků (Př. 4) je seznam agregací (tj. seznam jmen agregací). Vždy při sestavování požadavku na databázový modul je nezbytné vytvořit i objekt *DBError*, který je použit pro následnou analýzu vykonaného požadavku. V tomto příkladu je použit implicitní konstruktor. Ihned po dotazu následuje test úspěšnosti a na základě tohoto testu lze provést analýzu stavu a případné opakování dotazu popř. je zvolen jiný postup. (pozn. v tomto příkladu je vypsáno pouze chybové hlášení)

```
Public void initAgr() {
  Connection = new Connection();
  if ( !connection.spojeno()) {
    return;
  }

  DBError dbe= new DBError();
  ai= connection.spojeni().getListAgregaci( dbe, 0, 20000);

  if (!dbe.getStatus()) {
    Dialogs d= new Dialogs(new DialogAdapter(), new Frame(), false, "Chyba!",
                                  "Nepodařilo se načíst data! \n"+ dbe.sGetError(),
                                  Dialogs.ICON_CRITICAL,                    Buttons.BTN_OK+
Buttons.BTN_HELP);
  return;
```

```
  // Další inicializace

}
```

Př. 4 – dotaz na databázový modul

## 4. Vlákna

Java poskytuje dvě možnosti, jak vytvořit *thread*. Nový thread je vždy vytvářen jako objekt. Podmínkou je, aby třída ze které je thread vytvářen, byla buďto dědic třídy *Thread* nebo aby třída implementovala rozhranní *Runnable*. Rozdíl mezi oběma způsoby si vysvětlíme na následující ukázce (Tab. 1).

| **Thread** | **Runnable** |
|---|---|
| Class mujThread extends Thread {<br>  Private long test;<br>  MujThread( long test) {<br>  This.test= test;<br>  }<br>  public void run() {<br>  // zde se nachází nějaký<br>  // užitečný kód vlákna<br>  }<br>}<br><br>spuštění:<br><br>mujThread p = new mujThread(123);<br>p.start(); | Class mujRun implements Runnable {<br>  Private long test;<br>  MujRun( long test) {<br>  This.test= test;<br>  }<br>  public void run() {<br>  // zde se nachází nějaký<br>  // užitečný kód vlákna<br>  }<br>}<br><br>spuštění:<br><br>mujRun p = new mujRun(123);<br>new Thread(p).start(); |

Tab. 1 – srovnání Thread a Runnable

Protože Java nedovoluje dědičnost z více než z jedné třídy najednou, je první výhoda rozhranní *Runnable* jasná. Tuto variantu vytváření vlákna použijeme vždy, pokud vyžadujeme rozšíření nějaké třídy za současného vytvoření vlákna. Implementovat rozhranní Runnable je také výhodnější vždy, pokud nevyžadujeme současnou redefinici metod *start* a *stop*, které jsou součástí třídy *Thread*. Třída *Thread* je navíc zajímavá tím, že ona samotná už je implementací zmiňovaného rozhranní *Runnable*.

Postup použití *Runnable* je použitý právě u již dříve zmiňované třídy *StreamListener*. Tato třída má za úkol přijatá data zpracovat a poskytnout příslušným metodám, resp. objektu *DataStore*. Metoda *run* je implementována jako nekonečná smyčka, ve které je prováděn test na přijatá data, jejich zpracování a následné exportování pro další zpracování.

Vlákno lze v Javě ukončit dvěmi způsoby. První znamená volání metody stop, která přeruší v blíže nedefinovaném stavu provádění metody *run*. Druhý způsob je opuštění metody *run* a tím v podstatě ukončení vykonávání vlákna. Druhý způsob je korektnější, protože lze předem určit místo, kde dojde k přerušení vlákna a tím

je zaručena i správnost jeho ukončení, na rozdíl od prvního způsobu. Vlastní určení místa ukončení vlákna je bezpečnější už proto, že je zaručen návrat ze všech používaných objektů, ovšem to vše za cenu toho, že vlákno nemusí být ukončeno okamžitě. Lze sice použít triku spuštění vlákna jako konkrétního *Threadu* a tím mít možnost použití metody stop, ale jak již bylo řečeno je bezpečnější použít vlastní mechanizmus ukončení vlákna. Pro práci s vlákny lze také využít možnosti použití seskupování vláken. Tím lze ovládat více vláken najednou, lze zjišťovat informace o vláknech skupiny a samozřejmě je i přerušit nebo zrušit celou skupinu najednou. Navíc lze seskupit i skupiny vláken. Toto pokročile seskupování má několik pravidel i výhod viz. literatura [2].

## 5. Závěrem

V systému VIDIUM je upřednostňováno používání rozhranní a abstraktních tříd před dědičností, s výhodou je využíváno kompozice. Samozřejmě jsou případy, kdy je dědičnost nevyhnutelná. Autoři se však domnívají, že v mnoha případech ji lze efektivně nahradit například rozhranním a tím dosáhnout i lepšího řešení větší volnosti.

Systém VIDIUM se stále vyvíjí a autoři si jsou vědomi, že pro tak rozsáhlý projekt je popsání všech specifik na několika stranách nemožné. Záměrně byly vybrány "neviditelné" části systému, tak aby s nimi mohli být seznámeni i čtenáři s minimálními znalostmi jazyka JAVA.

Veškeré připomínky jsou vítány. Další informace o projektu i zkušenostech s jeho implementací lze nalézt na internetové adrese [4].

## Literatura


[1] The Java Language: A White Paper, http://java.sun.com
[2] FLANAGAN, D., Programování v jazyce Java, 1996, O`Reilly, ISBN 80-85896-78-8
[3] ŠTEFANÍK, J., Podklady pro dokumentaci projektu VIDIUM, 1998, firemní dokumentace Mediumsoft a.s., Ostrava
[4] BROŽEK, M., SVĚCH, D., ŠTEFANÍK, J., ASŘTP - VIDIUM, http://www.mediumsoft.cz/PRODUKT/ASRTP/vidium.htm