Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

MVE - 2

Příručka

Milan Frank a kol.
7.2.2005 (Verze: alfa-3)

1. Úvod

Systém MVE-II je modulární prostředí založené na datovém toku. Poskytuje univerzální, snadno pochopitelná rozhraní, jak pro tvorbu modulů (výkonného kódu) tak pro vytváření datových struktur které jsou mezi moduly sdíleny. Součástí systému je běhové prostředí (Runtime) se zajímavými vlastnostmi, které poskytují dobré vyjadřovací schopnosti při propojování modulů. Dvě nejvýraznější vlastnosti jsou: možnost cyklického propojení modulů a modulem řízené spouštění části sítě.

Systém v současnosti poskytuje dva způsoby jak definovat propojení modulů. Uživatelsky příjemnější je použít program MapEditor, kde je možné graficky, intuitivním způsobem, vytvářet a modifikovat různá propojení modulů. Druhou možností je přímo definovat uložení mapy modulů v textovém XML souboru. Tento formát je používán programem MapEditor pro ukládání vytvářené mapy. Vlastní spouštění mapy modulů je pak možné buď v samotném MapEditoru nebo z příkazové řádky pomocí programu RunMap.

S původním MVE z roku 1996 nemá nové prostředí mnoho společného. Zpětná kompatibilita je do budoucna plánována pomocí „proxy“ modulů, které zpřístupní funkcionalitu modulů starého MVE.

Velký rozdíl od předchozí verze leží v obecnosti návrhu jádra. Původní systém definoval datové struktury pro vizualizaci ve svém jádře a přidání nové datové množiny vyžadovalo rekompilaci celého systému. Nové prostředí naproti tomu neposkytuje ve svém jádře žádné datové struktury. Poskytuje pouze jakýsi standard co musí datová množina splňovat, aby ji bylo možné sdílet mezi moduly. Každá knihovna modulů pak může definovat vlastní datové struktury.

Obecnost návrhu vede k myšlence, že jádro systému není ve své podstatě prostředí pro vizualizaci, ale jedná se o obecné modulární prostředí využitelné v libovolné aplikační oblasti. Záleží jen na vytvořených datových množinách a modulech nad těmito množinami pracujícími, může tak snadno vzniknout například modulární prostředí pro zpracování textu.

Pokud mají začít vznikat navzájem kompatibilní moduly, pak musí být nejprve ustanoveno s jakými datovými množinami budou tyto moduly pracovat. Jinak by byl každý autor modulu nucen si vytvořit vlastní datovou množinu a moduly různých autorů by poté bylo jen velmi obtížné přimět ke spolupráci. Proto je s jádrem systému dodávána knihovna modulů Vizualization, která obsahuje množinu datových struktur, která by měli vyhovovat širokému spektru aplikací z počítačové grafiky a vizualizace dat.

Filosofie těchto datových struktur je velmi podobná systému VTK (Vizualization toolkit) od společnosti Kitware. Byly však provedeny nemalé úpravy pro lepší využitelnost, pochopitelnost a rozšiřitelnost. Celkově je návrh pro MVE-II „více objektový“. Dále se výrazně projevila snaha o podporu širokého spektra souřadných systémů a různých dimenzí.

Výzva: Pokud některý z uživatelů zjistí, že v oblasti vizualizace dat a počítačové grafiky mu chybí nějaká datové množina, nechť laskavě kontaktuje vývojový tým MVE II aby jím požadovaná datová struktura byla „standardizována“ a zahrnuta do knihovny Vizualization.

2. Spouštěcí mechanismus

Mapa modulů je multigrafem. Skládá se z množiny modulů (uzly) a orientovaných propojení mezi moduly (hrany). Pokud do nějakého modulu vedou hrany a žádné z něho nevychází jedná se o

terminální modul. Úkolem „spuštění mapy" je korektně spustit terminální moduly. Korektnost spočívá v připravení dat, která má terminální modul na vstupu před vlastním spuštěním. Musejí být tedy spuštěny všechny moduly ze kterých vedou spojení do terminálního modulu. Takto nám vzniká rekurzivní podmínka spuštění mapy modulů až ke zdrojovému modulu.

Místem propojení mezi moduly jsou porty. Máme porty vstupní a výstupní. Port nese informaci o datovém typu který akceptuje (vstupní) nebo který poskytuje (výstupní). Díky tomu mohou být prováděny kontroly zda jsou propojena kompatibilní místa.

Specifický význam pro spouštění a přenos dat má takzvaný zdržovací modul („delay module"), který na výstup poskytuje data z N-1 kroku a má k dispozici počáteční hodnotu. Díky tomu je možné aby graf propojení obsahoval cykly. Cykly však musí obsahovat zdržovací moduly, tak aby při odstranění jejich vstupu „Input" se graf stal stromem. Kontrola na korektní použití cyklů a zdržovacích modulů je prováděna před vlastním spuštěním mapy a uživatel je na takovou nekonzistenci upozorněn.

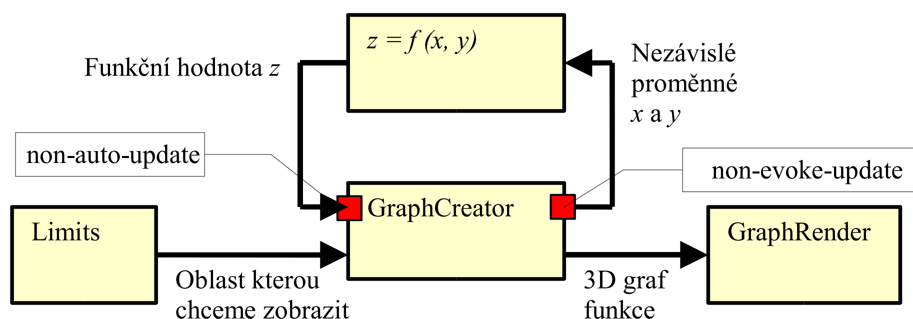
Další možností jak vytvořit korektní cyklus je místo zdržovacího modulu použít modul, který obsahuje výstupní port se speciální vlastností (non-evoke-update), ale k té až později.

Vstupní porty mohou mít u některých modulů speciální příznak (non-auto-update). Speciální port s tímto příznakem se explicitně vzdává přípravy vstupních dat před vlastním spuštěním modulu. Data jsou připravena až v okamžiku, kdy o ně modul za běhu explicitně požádá. Požadavek na přípravu dat může učinit libovolněkrát v průběhu jednoho běhu. Nazvěme to „aktualizace vstupu řízená modulem".

Podobně výstupní port může mít speciální příznak, o kterém již byla zmínka dříve. Do značné míry se jedná o analogii k variantě non-auto-update portu vstupního. Pokud má port nastaven příznak EvokeUpdate na false (non-evoke-update), pak požadavek na data z tohoto portu nezpůsobí spuštění modulu. Modul zaručuje, že na takovém portu vždy budou vystavena korektní data.

Primárním důvodem k zavedení non-evoke-update portu je, že modul může na takovýto výstupní port vystavit nějakou hodnotu a pak způsobit aktualizaci větve (části sítě), která vychází z jeho non-auto-update portu a končí v jeho non-evoke-update portu. Tímto způsobem si modul sám řídí běh celé větve, která je na něho připojena koncem i počátkem.

Prohlédněme si následující příklad, kde je nakresleno typické použití obou těchto speciálních portů.



Cílem výše uvedené mapy je nakreslit graf funkce počítané modulem $z=f(x,y)$ v oblasti dané modulem *Limits*. Celý graf funkce je vykreslen při jednom spuštění celé sítě. Přitom modul $z=f(x,y)$ může běžet libovolně krát. Modul *GraphCreator* vystaví na svém výstupním (non-evoke-update) portu nezávislé proměnné x, y a pak požádá přes vstupní (non-auto-update) port o aktualizaci větve. Tím dojde ke spuštění připojeného modulu a výpočtu funkční hodnoty, která je následně k dispozici modulu *GraphCreator*. Takto si může modul napočítat funkční hodnoty

v libovolných bodech a z nich pak vytvořit 3D graf. Ten po skončení výpočtu vystaví na normální výstupní port, ze kterého si jej *GraphRenderer* přečte a zobrazí.

Díky různým typům portů dostávají tvůrci modulů poměrně silný vyjadřovací prostředek. Pro autora mapy modulů (uživatele modulů) je pak řízení běhu sítě transparentní. Z určitého pohledu můžeme říci, že část sítě připojená k modulu přes not-auto-update port je podsítí tohoto modulu a stává se součástí běhu tohoto modulu, který tak přebírá zodpovědnost za to, kdy a kolikrát bude spuštěna.

Je zřejmé, že podsíť může opět obsahovat modul s non-auto-update portem a tak může vzniknout podsíť podsítě. Proto je zaveden pojem „úroveň sítě". Úrovně nula jsou všechny terminální moduly. Od nich se určuje úroveň zbytku mapy. Existují pravidla, která zaručují korektní chování mapy:

- Zdrojový modul může být nižší úrovně než modul cílový pouze v případě, že vstupní port je typu non-auto-update.
- Zdrojový modul může být vyšší úrovně než modul cílový pouze v případě, že cílový modul je *DelayModule* nebo zdrojový port je non-evoke-update.
- Ostatní propojení musí vést z a do stejné úrovně.

Tato pravidla jsou kontrolována před spuštěním mapy a v případě nekonzistence je uživatel náležitě upozorněn. Úroveň sítě je možné v aktuální verzi MVE zobrazit

Spouštěcí mechanismus je vzhledem k povoleným cyklům, zdržovacím modulům, *non-auto-update* a *non-evoke-update* portům poměrně komplikovanou záležitostí. Možnosti propojení modulů jsou obrovské. Současné běhové prostředí by mělo bez problémů zvládat prakticky všechny kombinace. Pokud uživatel i přes to při svém používání systému vytvořily mapu, která by měla být spustitelná, ale systém ji nezvládne tak nechť laskavě; kontaktujte vývojový tým MVE-2.

3. Tvorba Modulu

V MVE-2 jsou moduly organizovány do tzv. Assembly (pojem z .NET). Zjednodušeně řečeno se jedná o „dotnetovskou dll knihovnu", která může obsahovat libovolné množství modulů, datových objektů a libovolných dalších entit dle možností .NET. V současné verzi je možné jmenný prostor pojmenovávat libovolně. MapEditor zobrazuje moduly ve stromové hierarchii podle jmenných prostorů. Z využití jmenných prostorů také vyplývá existence plných a zkrácených jmen modulů. Například Examples.Sumator je plně kvalifikované jméno modulu Sumator ze jmenného prostoru Examples uložené v souboru Examples.dll. Tímto způsobem je možné odlišit moduly se shodným pojmenováním.

3.1. Rychlý úvod

Ukážeme si jak vytvořit minimalistický modul, který bude tisknout na standardní výstup řetězec získaný ze vstupních dat metodou ToString().

Každý modul je potomkem třídy MVECore.Module a minimálně musí implementovat dvě metody. Konstruktor, kde definuje své počáteční vlastnosti a metodu *Execute*, která je implementací vlastní činnosti modulu. V konstruktoru vytvoříme vstupní port s identifikačním řetězcem „Input". V metodě *Execute* získáme vstupní data z tohoto portu a vytiskneme jejich řetězcovou reprezentaci na standardní výstup.

```
public class ConsolePrinter : Zcu.Mve.Core.Module
{
  public ConsolePrinter()
  {
    this.AddInPort("input", "Zcu.Mve.Core.IDataObject");
  }

  public override void Execute()
  {
    DataObject data = GetInput("input");
    Console.WriteLine(data);
  }
}
```

Nyní se podívejme na výše uvedený příklad podrobněji. Třída ConsolePrinter je potomkem abstraktní třídy Module definované v jádře systému. Tím se stává modul srozumitelný pro jádro, které pracuje se všemi moduly jako s instancemi třídy Module. Jedinou výjimkou je třída jádra DealyModule, ale k té až později.

Standardní místo, kde se vytvářejí porty je konstruktor. Vlastní vytvoření je realizováno voláním metody AddInPort resp. AddOutPort. Prvním parametrem těchto metod je identifikační řetězec portu (jméno portu), který by měl pokud možno vystihovat jeho funkci. Jméno musí být unikátní v rámci modulu. Druhým parametrem je plně kvalifikované jméno datového typu jenž bude port akceptovat, resp. poskytovat.

Při použití modulu pak systém nedovolí propojení portů s nekompatibilními datovými typy (hrušek na jablka). V probíraném příkladě si autor modulu neklade žádné upřesňující požadavky. Jeho vstupní port je typován na základní třídu a tak akceptuje vše. Potomky základní třídy DataObject jsou všechny datové objekty.

Metoda Execute implementuje vlastní činnost modulu. V prvním řádku implementace vidíme volání metody GetInput. Tím je do metody Execute předána reference na data, která byla vystavena na výstupní port jež je připojen na „náš“ vstupní port. Pokud by byl vstupní port typován například na ScalarNumber, pak bychom mohli bez rizika přetypovat vrácenou referenci na tento typ a tak jednoduše získat data se kterými je možné dále pracovat. Následuje příklad typického získání dat ze vstupního portu typovaného na SclarNumber.

```
ScalarNumber num = (ScalarNumber)GetInput("input");
```

Vystavení dat na výstupní port se provádí metodou SetOuput. Voláním této metody je systému oznámeno, že tato data jsou připravena pro čtení ostatními moduly a do dalšího spouštění metody Execute nebudou měněna. Pokud modul nezavolá metodu SetOutput, tak to běhový systém považuje za signál, že výstupní data nebyla změněna a oznámí to připojeným modulům. Následně může běhový systém provést optimalizace spuštění připojených modulů. Dobrý příklad pro použití vstupních a výstupních portů je modul Sumator v knihovně Examples

3.2. Kompletní reference tvorby modulu

Nový modul se vytváří děděním od základního objektu Zcu.Mve.Core.Module. Ve svém potomkovi pak definuje svůj vlastní konstruktor, ve kterém typicky vytváří porty a provádí další potřebné činnosti. Tento konstruktor je volán při načtení mapy do operační paměti. Objekt tedy existuje přes

nedefinované množství spuštění mapy. Kromě toho tvůrce modulu definuje metodu Execute, která implementuje vlastní činnost modulu a je v případě potřeby volána runtimem.

Autor modulu má k dále dispozici celou řadu dalších metod pro přepsání i volání. Objekt Module také definuje řadu událostí. Využitím těchto dodatečných prostředků je možné tvořit moduly s pokročilými vlastnostmi.

Povinné překrytí

Execute

hlavní metoda implementující vlastní činnost modulu

Konstruktor

bezparametrický konstruktor, ve kterém je se vytvářejí porty. Dále zde autor modulu může provézt ostatní inicializační kroky. Zvažte také použití událostí *SimulationStart* či *ModuleCreate*.

Nepovinné překrytí

InvokeSetup

metoda je volána z GUI. Typicky pro vyvolání dialogu, který mění konfigurační data modulu.

WriteConfig(XmlElment config)

metoda plní XML element s názvem „config“ svými konfigurační data. Tato data jsou součástí souboru obsahující popis mapy modulů, proto by měla být malé velikosti.

Vhodné např. pro jméno souboru zdrojového modulu či prahovou hodnotu extrakce vlnoplochy.

Nevhodné např. pro objemová data, či počáteční podmínky výpočtu.

ReadConfig(XmlElment config)

parametr metody je XML element ze kterého si modul načte konfigurační data

Nutné volat

Povinnost použití následující sady funkcí je relativní a závisí na typu modulu (filtr, zdroj, stok):

AddInputPort(string type, string name, [bool required, bool autoUpdate])

Přidání vstupního portu. Zadává se požadovaný datový typ a jméno portu, které musí být v rámci modulu unikátní. Dále je možné definovat, zda má port povinný vstup (implicitně je povinný). Poslední parametr udává, zda daný port bude updatován až na žádost modulu v rámci přerušení výpočtu vlastního modulu. (Implicitně je to zakázáno)

AddOutputPort(string type, string name, [bool evokeUpdate])

Přidání výstupního portu. Zadává se požadovaný datový typ a jméno portu, které musí být v rámci modulu unikátní. Pokud je poslední parametr nastaven na *false* (implicitně *true*), pak žádost o data z tohoto portu nezpůsobí spuštění vlastního modulu. Je však nutné zajistit, aby v kterémkoliv okamžiku byla na portu vystavena nějaké korektní hodnota. Jedná se v podstatě o port s implicitní hodnotou.

GetInput(string name)

Čtení obsahu vstupního portu. POZOR! Obsah takto získaných dat nesmí být změněn!

SetOutput(string name, DataObject data, [bool dataSame])

Vystavení dat na výstup.

dataSame je nepovinný příznak předávaný modulům, které čtou z příslušného portu.

Možné volat

UpdateInput (string portName)

Způsobí okamžitý update větve, která je napojená na daný vstupní port. Ten proběhne mimo normální spouštěcí sekvenci. Možná (a nutné) volat pouze nad porty s příznakem *non-auto-update*. Řešení „podsítě“.

RemoveInPort/RemoveOutPort()

Odebrání portu v průběhu existence mapy. Nevolat při běhu mapy.

IsDataSame(string portName)

Vrací příznak zda jsou data totožná jako v předchozím volání Execute.

Události

Množina událostí, které je možné zaregistrovat.

SimulationStart/SimulationEnd

Začátek a konec výpočtu.

Prototyp:

```
void SimulationStartCallBack(object source);
void SimulationEndCallBack(object source);
```

PortsChanged

Jakákoliv změna konfigurace portu. (přidání, ubrání, změna datového typu)

Prototyp:

```
void PortsChangedCallBack(object source, Port port, PortChangeType
change);
```

StateChanged

Změna stavu modulu (waiting/updating/running)

Prototyp:

```
void StateChangedCallBack(object source, ModuleState newState);
```

Connect(portFrom, portTo, allowed) / Disconnect(port)

Událost nastává před pokusem připojení něčeho na port modulu. Možno zakázat propojení nastavením *allowed=false*.

Prototyp:

```
void ConnectCallBack(OutPort portFrom, InPort portTo, out bool allowed);
void DisconnectCallBack(InPort portTo, out bool allowed);
```

DataReady

Událost nastane při vystavení dat na připojený port

Prototyp:

```
void DataReadyCallBack(InPort inPort);
```

ModuleCreate

Událost je volána bezprostředně po přidání modulu do mapy modulů. Je zde tedy možné volat funkce, které nejsou v konstruktoru dostupné.

Je možné zabránit vytvoření modulu vyhozením výjimky.

Prototyp:

```
void ModuleCreateCallBack();
```

3.3. Příklady použití pokročilejších funkcí API

V následujících odstavcích budou ukázány typické postupy při tvorbě modulů a datových typů. Jedná se o ukázky jakým způsobem byly funkce API zamýšleny.

3.3.1. Proměnné porty

Typickým použitím reakce na *OnConnect* událost je přizpůsobení modulu aktuálnímu typu právě připojeného portu. Představme si případ že vytváříme modul, který násobí určitou množinu datových typů (Skalár, Vektor2D, Vektor3D) konstantou. Naší snahou by mělo být poskytnout na výstupu stejný typ jako máme na vstupu.

Typický postup v tomto případě je, že v konstruktoru vytvoříme pouze vstupní port typovaný na *IDataObject* a zaregistrujeme událost *OnConnect*. V reakci na tuto událost pak vytvoříme výstupní port stejného typu jako aktuální připojený port. V případě, že je připojený port nevhodného typu, pak jednoduše nastavíme výstupní parametr *allowed* na *false* a uživateli tím zamezíme připojení. Symetricky k této události je velmi vhodné registrovat také událost *OnDisconnect* ve které provedeme zrušení výstupního portu.

Následuje zdrojový kód, který je příkladem obsluhy události *OnConnect* a implementuje výše popsanou činnost.

```
void Connect(object source, OutPort portFrom, InPort portTo, out bool allowed)
{
  if (portFrom.PortType.FullName == typeof(Scalar).FullName ||
      portFrom.PortType.FullName == typeof(Vector2D).FullName ||
      portFrom.PortType.FullName == typeof(Vector3D).FullName)
  {
    this.AddOutPort("Output",  portFrom.PortType);
    allowed = true;
  }
  else
  {
    allowed = false;
  }
}
```

3.3.2. Ukládání konfigurace modulu

MVE-2 poskytuje mechanismus pro uložení důležitých konfiguračních dat modulu jakou součást mapy. Mapa modulů je ukládána ve formátu XML a konfigurace konkrétního modulu je ukládána v

uzlu *config*. Pro lepší názornost následuje příklad, který obsahuje část XML reprezentace mapy. Modul typu Examples.NumberSource je pojmenován NumberSource2 a obsahuje konfigurační data v jednoduché formě řetězce.

```
<mod name="NumberSource2" type="Examples.NumberSource">
  <config>1</config>
</mod>
```

Uzel *config* však může obsahovat libovolné atributy či další potomky. Je zřejmé, že není vhodné tímto způsobem ukládat data většího rozsahu.

Práce tvůrce modulu spočívá v přidávání či čtení dat do/z uzlu *config*, který dostane jako parametr metody *WriteConfig* resp. *ReadConfig*. Následuje smyšlený příklad ukládání dat do uzlu *config*.

```
public override void WriteConfig(XmlElement config)
{
  config.SetAttribute("value", val.ToString());
  XmlElement child = config.OwnerDocument.CreateElement("mychild");
  child.InnerText = "My child config data.";
  config.AppendChild(child);
}
```

Na první řádce je přidání atributu s názvem *value* a hodnotou *val* převedenou na řetězec. Na dalším řádku je vytvoření nového XML uzlu. Třetí řádek obsahuje zadání hodnoty uzlu. Poslední řádek implementuje přidání elementu do nadřazeného elementu config. Výsledný XML zápis pak může vypadat následujícím způsobem.

```
<mod name="NumberSource2" type="Examples.NumberSource">
  <config value="10">
    <mychild>
      My child config data.
    </mychild>
  </config>
</mod>
```

Čtení probíhá podobným způsobem. Dejme si však pozor na jednu vlastnost. Při čtení hodnoty uzlu *child* nedostaneme řetězec `"My child config data."`, ale `"\r\nMy child config data.\r\n"`. Znaky odřádkování činí potíže především při převodu řetězce na číslo. Proto doporučuji ořezávat řetězec metodou *Trim()*.

4. Tvorba datové třídy

Před vytvoření nového datové typu by měl předcházet důkladný průzkum, zda již neexistuje nějaký podobný, který by bylo možné použít a pro který již zcela jistě existuje množina použitelných modulů (např.: načítání, ukládání, rendering, …). Vznik duplicitních datových typů by po čase vedl k existenci navzájem nekompatibilních modulů.

Rozhraní datové třídy je jednodušší než rozhraní modulu. Deklaruje sice více metod, které musí tvůrce datové struktury implementovat, ale celkový počet je podstatně nižší. Každý datový objekt, který může být sdílen mezi moduly musí být potomkem třídy *Zcu.Mve.Core.DataObject*.

Jako jednoduchý příklad dobře poslouží třída *ScalarNumber* z knihovny *Examples*. Jedná se o reprezentaci jednoduché skalární hodnoty. Následuje popis významu jednotlivých částí.

Soukromá členská proměnná *val* nese vlastní data, která třída *ScalarNumber* obaluje. Výchozí hodnota je nastavena v bezparametrickém konstruktoru a přístup k ní je umožněn pomocí vlastnosti (property) *Val*. Přes tuto vlastnost k datovému objektu přistupují moduly. Výše popsané prvky jsou nepovinné, nicméně určují „užitečné“ vlastnosti datového objektu.

Dále následují metody, které musí být vždy implementovány. Velice důležité a implementačně poměrně náročné jsou metody *ReadData* a *WriteData*. Tyto metody zprostředkovávají zápis a čtení dat obsažených v datovém objektu do a z XML souboru. Implementací těchto metod při návrhu datové struktury je autor nucen rozmýšlet, jak svá data efektivně ukládat na disk v čitelné formě.

Při implementaci metody *ReadData* není dobré zcela spoléhat, že XML soubor který dostane metoda *ReadData* je vždy vytvořen metodou *WriteData*. Uživatel dat bezpochyby rád přímo upraví XML reprezentaci svých dat tak, jak by to metoda *WriteData* nikdy neudělala. Proto by metoda *ReadData* měla být schopna zpracovat (dle možností) libovolný rozumný zápis. To je mnoho práce navíc, ale určitě se vyplatí, jelikož tyto metody jsou psány pouze jednou pro jeden datový typ a poté jsou široce používány.

Další metodou je *DeepCopy*. Jedná se o metodu jež musí (pokud má systém správně pracovat) alokovat nový paměťový prostor a provést zkopírování veškerých dat. To musí být provedeno tak aby, se změna datového obsahu v originálu neprojevila v kopii a obráceně (podstata hluboké kopie datové struktury …). Tato metoda je používána například pro kopírování obsahu vstupu do *DelayModulů*, aby v dalším kroku mohli poskytnout data, jež měla na vstupu v kroku minulém.

Metoda *CheckConsistence* je důležitá především pro komplikované datové struktury, kde může docházet k nekonzistencím. (např.: trojúhelník, jehož jeden z indexů překračuje počet bodů, normála jejíž velikost je různá od jedné +/- epsilon, atd.) V takových případech má metoda vrátit hodnotu *false* a tak uživatele datové struktury upozornit na možné problémy při dalším zpracování. V případě že metoda vrací *false* je velmi vhodné popsat na standardní výstup, pomocí *Console.WriteLine()*, příčinu nekonzistence. Dále je možné vypisovat určitá varování, i když je vrácena hodnotu *true*. Tedy i v případě, že jsou data uznána za konsistentní, ale přesto něco není zcela běžné.

Dále je možné přepsat metodu *ToString* a rozumným způsobem tak převést data na čitelný řetězec, který například může být tisknut na konsoli pomocí modulu *ConsolePrinter*.