Přírodovědecká fakulta Univerzity Palackého v Olomouci
katedra matematické informatiky

DIPLOMOVÁ PRÁCE

Abstraktní vzhledy dat

1996 Miloslav Bešta

Katedra matematické informatiky PřF UP Olomouc

Předpis pro vypracování diplomové práce

I.

Při zpracování diplomové práce student usiluje o co nejvýstižnější a nejpříznivější obraz o svých schopnostech, úrovni svých znalostí a osvědčuje, jak si osvojil nezbytné návyky odborného a technického způsobu vyjadřování, jaké má znalosti odborné literatury a jak jí umí používat.

Diplomová práce se v tomto smyslu hodnotí jako celek.

II.

Student se musí ve své práci bez újmy na úplnosti vyjadřovat stručně, odborně, slohově i gramaticky správně. Text (včetně popisků příloh) diplomové práce musí být před odevzdáním pečlivě prohlédnut. Písařské chyby, chyby v tisku apod., musí být opraveny.

Nedostatky diplomové práce v tomto směru snižují klasifikaci i práce jinak obsahově dobré.

III.

Text diplomové práce musí být zpracován programem LaTeX s použitím makra katedry matematické informatiky a vytištěn po jedné straně papíru formátu A4. Diplomová práce se odevzdává takto:

- 1x originál v knihařské vazbě
- 1x kopie spojená v polotuhých deskách
- 1x záznam textu diplomové práce pořízený textovým editorem na disketě, kterou si student před odevzdáním vyzvedne na sekretariátě katedry.

IV.

V originále se na stránku s místopřísežným prohlášením připojí vlastnoruční podpis.

Došlo-li v průběhu zpracování diplomové práce k významným odchylkám od zadání diplomového úkolu, které na základě žádosti studenta schválil vedoucí

katedry, připojí se další list, na němž budou tyto skutečnosti uvedeny. Způsob uvedení se stanoví individuálně a v originále bude k záznamu připojen podpis vedoucího katedry. Nevýznamné odchylky uvádí student v anotaci.

V.

Uspořádání odevzdaných vyhotovení diplomové práce se předepisuje následovně:

- předsádkový list
- zadání diplomové práce
- předpis pro vypracování diplomové práce
- místopřísežné prohlášení o samostatném vypracování diplomové práce
- (list s vyjádřením k odchylkám od zadání, pokud byly schváleny)
- anotace diplomové práce
- obsah diplomové práce s odkazy na odpovídající stránky nebo čísla příloh, seznam tabulek a obrázků
- rozvedení diplomového úkolu podle obsahu s dílčími závěry na konci každé kapitoly
- celkový závěr diplomové práce
- cizojazyčné resumé (cca 15 řádků)
- seznam použité literatury v abecedním autorském uspořádání – tabulky, nákresy, přílohy (doklady, prospekty, elaboráty apod.)

Přední deska originálu bude potištěna stejně jako předsádkový list, je možno vypustit znak univerzity.

Diplomová práce, která nebude vyhovovat tomuto uspořádání, nemůže být přijata.

VI.

Použitá literatura, předlohy apod., použité při zpracování musí být na příslušných místech v diplomové práci označeny odkazem na průběžné číslo ze seznamu použité literatury, příslušná stránka se uvede v hranaté závorce. Jde-li o citát, uvedou se čísla prvního a posledního řádku citátu.

Součástí řešení diplomového úkolu je zdrojový text programu. Způsobilost provozu produktu osvědčuje student při obhajobě diplomové práce.

VII.

Všechny propočty nebo výpočty musí být podrobně a přehledně uspořádány tak, aby každý odborník byl schopen jejich správnost přezkoušet. U použitých vzorců, součinitelů nebo hodnot z praxe musí být uveden původ. Jsou-li uváděny údaje, které mohou tvořit předmět hospodářského nebo státního tajemství, je třeba tuto okolnost uvést při odevzdání diplomové práce. V takovém případě se pro přístup k diplomové práci předepíše zvláštní režim (závazný i pro oponenty).

V Olomouci dne 13. března 1996

Doc. RNDr. Josef Hos
vedoucí katedry
matematické informatiky

Za správnost: Zd. Nesvadbová

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval(a) samostatně.

25. března 1996 Miloslav Bešta

Anotace

Během posledních několika málo let došlo v oblasti softwarového inženýrství k velmi bouřlivému vývoji. K tomuto vývoji přispěl velkou měrou také obrovský pokrok v návrhu znovupoužitelných softwarových komponent. Cílem této diplomové práce je představení nového konceptu pro návrh interaktivních aplikací zvaného Abstraktní Datový Pohled. Tento koncept je založen na oddělení uživatelského rozhraní aplikace od vlastního výkonného jádra aplikace a to jak na úrovni návrhu, tak na úrovni implementace. Pro popis abstraktního datového pohledu je použito objektově orientované modifikace specifikace VDM.

Na tomto místě bych chtěl poděkovat vedoucímu diplomové práce, RNDr. Vladimíru Sklenářovi, za jeho odborné vedení a věcné rady a připomínky, které mi poskytl při zpracování této diplomové práce.

Obsah

Předmluva **1**

1 Úvod **2**
1.1 Historie programování 2
1.2 Nové přístupy k návrhu aplikace 2
1.2.1 Objektově orientovaný model 2
1.2.2 Oddělené uživatelské rozhraní 3
1.2.3 Návrhové vzory 4
1.2.4 Formální specifikace 5

2 Abstraktní datové pohledy **6**
2.1 Koncept abstraktního datového pohledu 6
2.1.1 Charakteristika ADP 6
2.1.2 Formální model ADP 8
2.1.3 Zobecnění ADP — prezentace po síti 10
2.2 Klasifikace modelů pro návrh uživatelského rozhraní 12
2.2.1 Monolitický model 13
2.2.2 Model klient – server 14
2.2.3 Model MVC 14
2.2.4 Model ADP 14

3 VDM — formální metoda pro vývoj aplikací **16**
3.1 Základní konstrukce VDM 16
3.1.1 Úvod do matematické notace 16
3.1.2 Modelování typů a operací 17
3.1.3 Modelování množinových typů 19
3.1.4 Modelování sekvenčních typů 20
3.1.5 Specifikace implicitních funkcí 21
3.1.6 Specifikace explicitních funkcí 22
3.1.7 Modelování strukturovaných typů 25
3.1.8 Modelování asociovaných typů 27
3.2 Koncept ADP a jeho specifikace 28
3.2.1 Specifikace objektových typů 28
3.2.2 Specifikace abstraktních datových pohledů 31
3.2.3 Zachytávání a obsluha událostí systému 33
3.3 Příklad specifikace jednoduchého ADP 34

4 Principy činnosti knihovny ADP **37**
4.1 Stanovení cílů 37
4.2 Hierarchie tříd 38
4.3 Principy spolupráce mezi třídami 40

4.4 Základní metody tříd ADP 43
4.5 Implementace jednoduchého komunikačního ADP 44

5 Postup při programování metodou ADP 47
5.1 Transformace specifikace VDM do jazyka C++ 47
5.2 Postup při kódování aplikace 50
5.2.1 Metody třídy TBaseADV 50
5.2.2 Metody pro obsluhu událostí 51
5.2.3 Metody třídy TVisualADV 52
5.2.4 Metody třídy TConnectADV 52
5.2.5 Zbývající požadavky na tvorbu aplikace 53

6 Závěr 54

A Popis událostí systému 58

B Popis metod některých tříd 61
B.1 Třída TBaseADV 61
B.2 Třída TVisualADV 63
B.3 Třída TConnectADV 64
B.4 Třída TADVContext 65

Seznam obrázků

1 Vztahy mezi instancemi ADP a ADO 7
2 Formální model abstraktního datového pohledu 8
3 Popis typů formálního modelu ADP 9
4 Podpora tvorby jednosměrného rozhraní mezi dvěma ADO 10
5 Podpora tvorby obousměrného rozhraní mezi dvěma ADO 10
6 Implementace modelu klient–server pomocí ADP 11
7 Formální model rozšířeného abstraktního datového pohledu 12
8 Srovnání architektur pro návrh aplikací 13
9 Schéma deklarace objektového datového typu 29
10 Schéma deklarace potomka objektového datového typu 30
11 Schéma deklarace abstraktního datového pohledu 31
12 Mapování ADP na ADO pomocí proměnné *Owner* 32
13 Schéma deklarace operací pro obsluhu událostí 33
14 Specifikace ADP zobrazujícího celé číslo na stupnici 35
15 Hierarchie tříd abstraktních datových pohledů 39
16 Schéma činnosti návrhového vzoru abstraktní továrna 41
17 Schéma činnosti návrhového vzoru adaptér 42
18 Deklarace třídy pro převod stavu trenažéru 45
19 Implementace metod třídy pro převod stavu trenažeru 46
20 Přechod od VDM ke konkrétní implementaci 48
21 Implementace datových typů VDM 49

Předmluva

Během posledních několika málo let došlo v oblasti softwarového inženýrství k velmi bouřlivému vývoji. K tomuto vývoji přispěl velkou měrou také obrovský pokrok v návrhu znovupoužitelných softwarových komponent, díky kterým se podařilo značně urychlit fáze návrhu a implementace nové aplikace.

Cílem této diplomové práce je představení nového konceptu pro návrh interaktivních aplikací, který se nazývá Abstraktní Datový Pohled. Tento koncept je založen na oddělení uživatelského rozhraní aplikace od vlastního výkonného jádra aplikace a to jak na úrovni návrhu, tak na úrovni implementace. Pro popis abstraktního datového pohledu je použito objektově orientované modifikace specifikace VDM. Cílem práce není do detailů vysvětlit specifikaci VDM či naopak popsat všechny implementační detaily. Spíše se jedná o snahu představit tento nový koncept pro návrh aplikací tak, aby byly zřejmé jeho přednosti i nedostatky a bylo tak čtenáři umožněno srovnávání s jinými možnými modely.

Vlastní text této práce je rozdělen celkem do šesti kapitol, ve kterých je postupně rozebrána celá problematika abstraktních datových pohledů.

V úvodní kapitole je v krátkosti popsán dosavadní vývoj v oblasti návrhu a vývoje aplikací. Jsou zde také zmíněny některé relativně nové postupy, které výrazně urychlily a zjednodušily tvorbu nových aplikací.

Ve druhé kapitole je podrobně rozebrán pojem „abstraktní datový pohled“. Důraz je zde kladen především na obecnou charakteristiku abstraktního datového pohledu, jeho vlastnosti, chování a jeho provázanost na okolní prostředí uvnitř aplikace. Dále je v této kapitole provedena klasifikace zkoumaného modelu architektury aplikace a jeho srovnání s jinými, v praxi běžně používanými modely.

Třetí kapitola se zabývá popisem formální specifikace VDM. Jsou zde vysvětleny základní rysy této specifikace, její syntaxe a použití. Dále jsou zde popsána rozšíření, která byla použita k popisu abstraktního datového pohledu.

Ve čtvrté kapitole je popsána vlastní implementace abstraktních datových pohledů. Jsou zde vysvětleny principy činnosti některých základních metod, je zde popsána použitá objektová hierarchie a také vzájemné vztahy mezi datovými strukturami.

Pátá kapitola popisuje metodiku přechodu od formální specifikace VDM k implementaci abstraktních datových pohledů, která je popisována ve čtvrté kapitole. Dále je zde popsán doporučený postup, jak vytvářet nové datové pohledy a jakým způsobem navrhovat aplikace, které budou modelu abstraktních datových pohledů používat.

Konečně v šesté kapitole jsou zhodnoceny výsledky, kterých bylo dosaženo v této diplomové práci, a je zde proveden rozbor použitelnosti těchto výsledků v praxi.

1 Úvod

1.1 Historie programování

Programování, tak jako mnoho dalších oborů lidské činnosti, vzniklo v relativně nedávné minulosti a od té doby prošlo bouřlivým vývojem. Vše vlastně začalo někdy ve čtyřicátých letech našeho století, kdy došlo k vynálezu prvního počítače. Od tohoto okamžiku se tento obor stal jedním z nejprogresivněji se rozvíjejících oborů lidského bádání.

Okamžitě po vynálezu počítače začal také rozvoj disciplín s tímto objevem souvisejících. Mezi nimi se v první řadě jednalo o softwarové inženýrství a vývojářství. Z počátku byla efektivita vývoje aplikací velmi nízká. Pak však došlo k výraznému vývojovému skoku, který byl způsoben prvním použitím von Neumannova návrhu architektury počítače. Od tohoto okamžiku začal také bouřlivý rozvoj nových metod návrhu aplikací.

Nejprve došlo k rychlému vývoji programovacích jazyků, pak následovaly změny v koncepci těchto jazyků. Zde je třeba se zmínit o významu procedurálních a modulárních jazyků, které značným způsobem změnily a urychlily vývoj aplikací. Na konci šedesátých let pak došlo k průlomu vytvořením prvního objektově orientovaného jazyka. Následné využití principů objektově orientovaného programování v jazyku Smalltalk předznamenalo éru moderních programovacích jazyků a nových velmi efektivních postupů při návrhu aplikací.

1.2 Nové přístupy k návrhu aplikace

1.2.1 Objektově orientovaný model

Základem dnešního moderního programování je v každém případě objektově orientované programování (OOP). Ovšem pod pojmem OOP si nelze představit jen syntaxi nějakého konkrétního programovacího jazyka. OOP představuje celý soubor pravidel a technologií, jak navrhovat a psát nové aplikace. Současně s pojmem OOP je proto třeba se také zmínit o dalších pojmech, jako je objektově orientovaná analýza (OOA) a objektově orientovaný návrh (OOD). Samotné objektově orientované programování je založeno na třech základních principech.

Za prvé je to princip zapouzdření. Tento princip spočívá v tom, že data a funkce, které s těmito daty pracují, jsou společně svázány v popisu, který se nazývá třída. Příslušné datové členy třídy jsou pak z vnějšku přístupné zpravidla jen voláním funkcí, které jsou s danou třídou svázány. Tyto funkce bývá zvykem nazývat metodami třídy a volání metod je pak možno považovat za zasílání zpráv. Od každé třídy se vždy vytvářejí tzv. instance třídy neboli objekty, které zabezpečují vlastní uložení informací a vykonávání transformačních algoritmů nad požadovanými daty.

Druhým principem OOP je dědičnost. Tento princip umožňuje mnohonásobné

použití již jednou vytvořeného kódu a také snadnější údržbu a úpravu již hotového programového systému. Dědičnost je založena na tom, že každá třída, která je potomkem jiné třídy dědí všechny datové členy a také všechny metody třídy předka. Navíc je možné do takto zděděné třídy přidat nové datové členy a nové metody případně přepsat kód již existujících metod. Díky tomu lze vytvořit novou třídu a v této nové třídě využít již napsaného původního kódu metod rodičovské třídy.

Třetím, snad nejdůležitějším principem OOP, je polymorfismus. Tento princip totiž umožňuje využití abstrakce při návrhu programového díla. Polymorfismus v objektovém modelu znamená, že ta samá zpráva může být zaslána rozličným objektům — dokonce instancím různých tříd — a přitom v době kódování této zprávy nemusí a zpravidla také není znám ani přijímající objekt ani implementace jeho metod. Každý objekt přitom může reagovat na poslanou zprávu jiným, sobě vlastním způsobem. Takto je umožněno, aby byly psány aplikace, které jsou velmi snadno modifikovatelné a rozšiřitelné, a to dokonce i v těch případech, kdy nejsou k dispozici zdrojové kódy původních tříd.

Objektově orientovaný návrh klade důraz na správnou strukturu aplikace, která by měla být založena na objektově orientované dekompozici systému. Návrh a tvorba programového systému podle zásad OOD má charakter analýzy vztahů, která vede ke klasifikaci objektů a jejich tříd. Výsledný návrh systému je pak postupným zlepšováním a zpřesňováním založeným na lepším porozumění datovým strukturám. Fáze specifikace toho, co má programový celek plnit, by pak měla být odložena na co nejpozdější etapu vývoje. Jen tento přístup k návrhu zabezpečí vysokou úroveň znovupoužitelnosti a snadné modifikovatelnosti systému.

Konečně cílem objektově orientované analýzy systému je vytvoření správného modelu problému na základě objektově orientovaného přístupu. Tento model by měl vyjadřovat všechny požadované vztahy a zahrnovat veškeré požadované chování aplikace. Přitom by ale měl dbát na to, aby objekty a třídy, které budou tento model reprezentovat, vyhovovaly všem požadavkům objektově orientované dekompozice. OOA tedy zkoumá požadavky na systém a snaží se objevit třídy a objekty, které se v systému vyskytují.

1.2.2 Oddělené uživatelské rozhraní

Objektově orientované programování znamenalo relativně velký skok v efektivitě programování. Ovšem vývoj se v tomto směru nezastavil. Dalším krokem, který vylepšil možnost znovupoužití již jednou napsaného kódu aplikace, bylo oddělení uživatelského rozhraní aplikace od vlastního algoritmického jádra aplikace. Toto oddělení rozhraní se projevilo nejen na úrovni vlastní aplikace, ale také na úrovni jednotlivých tříd uvnitř aplikace. To vedlo k tomu, že začalo vznikat několik skupin specializovaných tříd. Jedna skupina tříd například zapouzdřuje základní datové struktury, které jsou aplikacemi používány, druhá skupina tříd zajišťuje vykonávání určitých algoritmů nad těmito datovými strukturami a třetí skupina

pak umožňuje prezentaci těchto datových struktur směrem k uživateli.

Výhodou modelu, kdy jsou vzájemně odděleny datové struktury aplikace od jejich prezentační či manipulační vrstvy, je snadná záměna objektů těchto vrstev bez toho, že by bylo třeba cokoliv měnit na zbývajících vrstvách. Jedinou podmínkou pro zachování funkčnosti tohoto modelu je dodržení pevně definovaného rozhraní mezi vrstvami. Důsledkem takto navrženého modelu je velmi vysoká míra flexibility aplikace a také značné rozšíření možností znovupoužití tímto způsobem vytvořených systémů.

Abstraktní datové pohledy představují jednu z možných interpretací tohoto modelu vhodnou zvláště pro návrh vysoce interaktivních aplikací. Abstraktní datové pohledy totiž důsledně oddělují vrstvu datových struktur a vrstvu prezentační v dané aplikaci. Díky tomu je pak možné nahradit například pohled, který zobrazuje pole jako řadu čísel, jiným pohledem, který totéž pole zobrazí jako graf, beze změny výkonného jádra aplikace. Podrobnosti o tomto modelu však budou diskutovány až v kapitole 2 a následujících.

1.2.3 Návrhové vzory

Navrhování objektově orientovaného systému není jednoduché a navrhování znovupoužitelného objektově orientovaného systému je možná ještě těžší. Snad každý zkušený návrhář jistě potvrdí, že vytvořit „správný“ návrh systému na první pokus je téměř nemožné. Avšak zkušenost také říká, že při návrhu různých systémů jsou často využívány stejné myšlenky a stejné postupy. Návrhové vzory slouží k uchování těchto postupů a myšlenek tak, aby jich mohli využívat všichni, kdo je potřebují, a nebylo třeba tzv. znovu vynalézat kolo. Návrhové vzory tak umožňují místo znovupoužitelnosti kódu znovu použít myšlenky.

Co to vlastně návrhový vzor je? Lze říci, že návrhový vzor je slovní popis myšlenky či postupu, který má obvykle čtyři následující části:

- Jméno vzoru, které dokáže v několika málo slovech vyjádřit podstatu problému.
- Popis problému, při řešení kterého je vhodné daný vzor řešení použít.
- Řešení problému, které popisuje jednotlivé prvky návrhu, jejich vzájemné vztahy, komunikaci a spolupráci.
- Výsledky a důsledky řešení problému tímto způsobem.

V tomto popisu jsou obsaženy všechny nezbytné údaje pro to, aby návrhář mohl po definování problému najít vzor, který je vhodný pro jeho řešení. Na základě tohoto vzoru pak může navrhnout systém tříd a objektů, který bude vyhovovat všem požadavkům na objektovou orientovanost a znovupoužitelnost. Návrh aplikace tímto způsobem je tedy rychlý a přitom se v něm vyskytuje minimum chyb.

1.2.4 Formální specifikace

V mnoha případech je třeba, aby návrh systému byl velmi přesný, konzistentní a úplný, ale aby přitom bylo pominuto všech nepodstatných detailů. Dále je nezbytné, aby zápis tohoto návrhu byl dobře srozumitelný a také lehce implementovatelný v libovolném programovacím jazyce. Všem těmto požadavkům vyhovuje specifikace návrhu pomocí některé z formálních metod.

Formální metody využívají matematické notace a matematického odvozování pro řešení a zápis požadovaného problému. Výsledkem návrhu systému je model dat a operací nad nimi, který může být dále zpřesňován a upravován, kontrolován a následně implementován. Takto vytvořený systém je pak velmi dobře znovupoužitelný a přenositelný do libovolného softwarového prostředí. Správnost a úplnost návrhu systému lze navíc ověřit pomocí matematického aparátu specifikace.

V této diplomové práci je pro zápis abstraktních datových pohledů použito notace VDM, která je doplněna o objektově orientované rysy tak, aby vyhovovala nezbytným požadavkům pro zápis abstraktních datových pohledů. Podrobnější popis této specifikace lze nalézt v kapitolách 3 a 5.1. Detailní charakteristika a popis notace VDM se pak nachází v [8, 14].

2 Abstraktní datové pohledy

V poslední době se stále častěji objevují nové postupy pro návrh a tvorbu aplikací, které se snaží vytvářet uživatelské rozhraní aplikace pomocí integrace jednotlivých vizuálních komponent do sebe. Přitom existuje zřejmá snaha oddělit vlastní návrh i implementaci uživatelského rozhraní od návrhu a implementace samotného jádra aplikace. V této kapitole se pokusím popsat a následně zobecnit jeden z modelů, který splňuje výše uvedené požadavky. Tento model, který se nazývá abstraktní datový pohled (ADP), byl poprvé navržen na katedře informatiky Univerzity ve Waterloo v Kanadě. Specifikace tohoto modelu je popsána například v [2, 3, 4, 12].

2.1 Koncept abstraktního datového pohledu

2.1.1 Charakteristika ADP

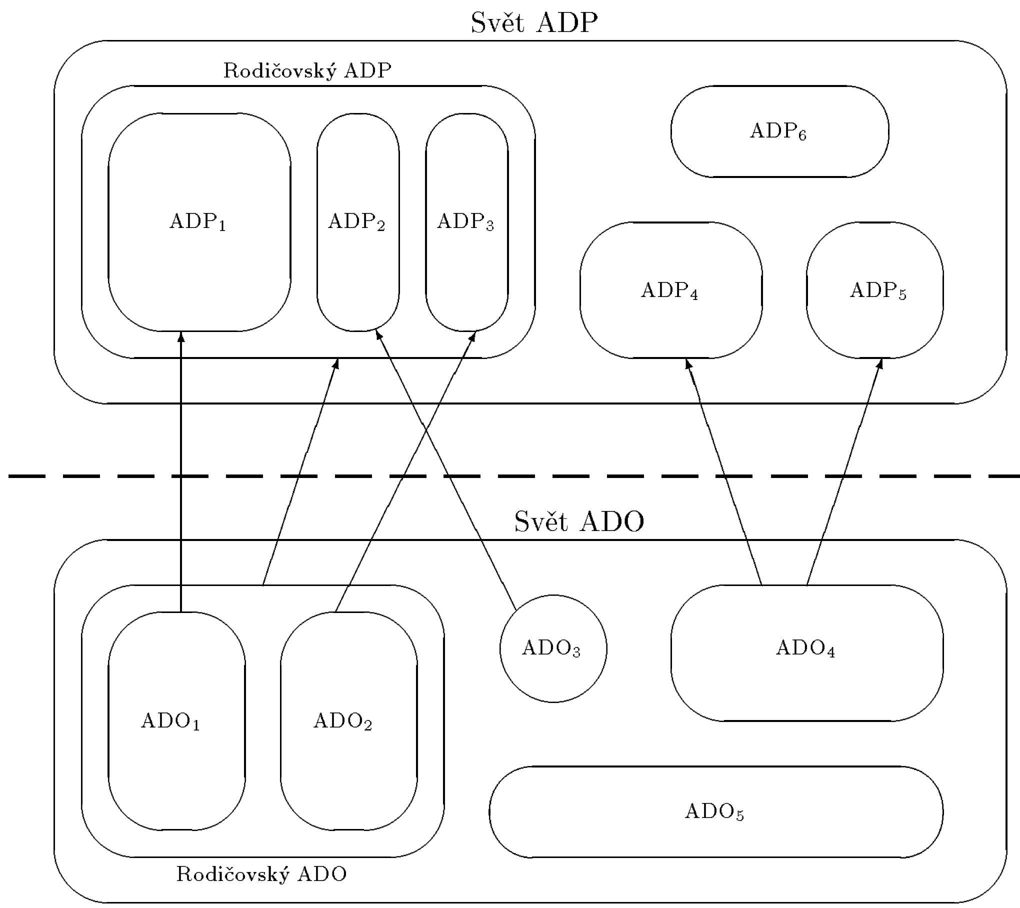
Jak už bylo řečeno v předchozím odstavci, abstraktní datové pohledy by měly vyhovovat především požadavkům na oddělení prezentační části aplikace od výkonné části a měly by také umožňovat vytváření nových pohledů skládáním pohledů do větších celků.

Co to tedy je abstraktní datový pohled? V první fázi si definujme ADP jako objekt, který je schopen grafické prezentace jiného (datového) objektu například v okně a který je také schopen komunikace s uživatelem pomocí vstupních zařízení, jako jsou například myš a klávesnice.

Objekt, který je prezentován pomocí ADP, budeme nazývat abstraktní datový objekt (ADO). Tento ADO bývá zpravidla součástí výkonného jádra aplikace a nemá žádnou možnost přímé komunikace s uživatelem. Množina všech těchto datových objektů tvoří jistý stav aplikace. Tento stav aplikace je v průběhu života aplikace měněn a tyto změny jsou následně prezentovány uživateli právě pomocí ADP. Je tedy zřejmé, že ADP a ADO tvoří navzájem jisté páry, kde každý ADP prezentuje jeden ADO, avšak jeden ADO může být prezentován několika ADP současně.

Na obrázku č. 1 je ukázka možných vazeb mezi abstraktními datovými pohledy a abstraktními datovými objekty. Je zde velmi dobře vidět, že jeden datový objekt může být prezentován i pomocí více než jednoho pohledu, avšak je také možné, že některé datové objekty nemají žádnou reprezentaci vzhledem k uživateli. To je zpravidla případ pomocných datových struktur. Také mohou existovat datové pohledy, které nejsou svázané s žádným datovým objektem. V tomto případě se zpravidla jedná o tzv. dekorační pohledy. Konečně z obrázku je zřejmá ještě další vlastnost ADP a to je možnost kompozice abstraktních datových pohledů do větších celků.

Kompozice je schopnost abstraktních datových pohledů být navzájem do sebe vloženy. Tato schopnost umožňuje, aby dětský pohled byl vykreslován uvnitř



Obr. 1: Vztahy mezi instancemi ADP a ADO

$Prezentace$:$(Stav_ADP \times Stav_ADO) \rightarrow Stav_Okna$
$Chování_ADO$:$(Stav_ADO \times Operace_ADO) \rightarrow Stav_ADO$
$Změna_ADP$:$(Stav_ADP \times Stav_ADO \times Sys_Událost) \rightarrow Stav_ADP$
$Přenos_Události_ADP$:$(Stav_ADP \times Stav_ADO \times Sys_Událost) \rightarrow Operace_ADO$
$Odkazování_ADP$:$(Stav_ADP \times Stav_ADO \times Sys_Událost)$
$\rightarrow Sys_Událost \times Seznam_ADP$
$Invarianty$:$(Stav_ADP \times Stav_ADO) \rightarrow Boolean$

Obr. 2: Formální model abstraktního datového pohledu

okna rodičovského pohledu. Díky této vlastnosti je pak možné vytvářet složitější pohledy skládáním z jednodušších pohledů.

Na okno pohledu nejsou také kladeny žádné požadavky co se týče jeho tvaru. Proto je možné navrhnout ADP jehož okno bude mít tvar obdélníku, ale také elipsy či polygonu. Vše je ponecháno až na implementaci ADP a výsledkem tohoto postupu je vysoká flexibilita použití.

Abstraktní datové pohledy dále umožňují dědění vlastností a chování. Jen tak je totiž možné zajistit, aby uživatel mohl bez problémů vytvářet nové specializované pohledy bez toho, aby musel mít k dispozici původní zdrojový kód. Tvorba nového ADP je tím zjednodušena na modifikaci v ideálním případě jen jediné části kódu, která se zabývá vlastním zobrazováním pohledu.

Zřejmě každý datový pohled musí obsahovat vlastní privátní datové struktury, které definují jeho aktuální stav. Tato privátní data hrají zpravidla pomocnou funkci při prezentaci vlastního pohledu, avšak mohou hrát také roli ADO a být tedy prezentována dalšími, zpravidla dětskými ADP. Tím je opět umožněno snadnější vytváření složitějších pohledů. Například uvnitř pohledu, který prezentuje textový soubor může být dětský pohled, který je svázán s číslem aktuálně zobrazovaného řádku, a tento dětský pohled se uživateli prezentuje jako posuvný jezdec na pravítku.

2.1.2 Formální model ADP

V této kapitole se pokusím stručně popsat formální model abstraktního datového pohledu. Tento model pak bude využíván v následujících kapitolách při formálním zápisu pohledů (viz. kapitola 3 a zvláště pak 3.2).

Formální model ADP lze popsat vztahy, které jsou uvedeny na obrázku č. 2. Každá definice zde představuje zobecněnou funkci, která určuje vztah mezi hodnotami definičního oboru funkce a oboru hodnot. Avšak ještě dříve, než budeme pokračovat v důkladnějším popisu tohoto formálního modelu, je třeba se pozastavit u obrázku č. 3 a podrobněji si vysvětlit jednotlivé typy hodnot použitých v těchto vztazích.

Typy *Stav_ADP* a *Stav_ADO* jsou definovány jako množiny všech hodnot, které jsou přípustné pro instance abstraktního datového pohledu a abstraktní-

Stav_ADP	stav abstraktního datového pohledu (ADP)
Seznam_ADP	seznam všech instancí ADP
Stav_ADO	stav abstraktního datového objektu
Stav_Okna	stav okna se zobrazeným ADP
Operace_ADO	přípustné operace s abstraktním datovým objektem (ADO)
Sys_Událost	přípustné události generované vstupním zařízením

Obr. 3: Popis typů formálního modelu ADP

ho datového objektu. Další typ, *Seznam_ADP*, představuje typ seznamu všech instancí abstraktních datových pohledů, které aktuálně v aplikaci existují. Typ *Stav_Okna* je určen rozsahem možných hodnot nebo vzhledem části obrazovky monitoru, která je právě aktivní a zobrazuje stav ADP. *Operace_ADO* zahrnuje množinu všech přípustných operací, které mohou být vykonávány nad ADO. Konečně typ *Sys_Událost* je definován jako množina všech vstupních událostí, které mohou nastat během interakce uživatele s datovým pohledem.

Nyní se pokusme podrobněji vysvětlit vztahy formálního modelu, jak jsou definovány na obrázku č. 2. První vztah definice, který je nazván *Prezentace*, vyjadřuje skutečnost, že vzhled aktivního okna na obrazovce monitoru je důsledkem jak stavu datového pohledu (ADP), tak stavu datového objektu (ADO).

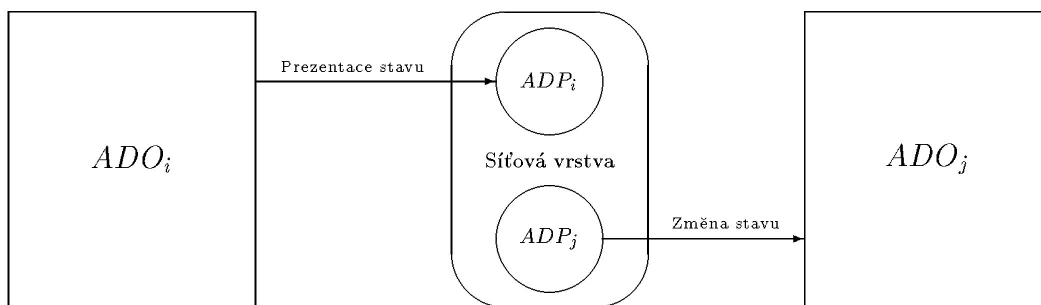
Vztah nazvaný *Chování_ADO* vyjadřuje zcela zřejmou skutečnost, že výsledkem aktuálního stavu datového objektu a operace, která je na tento objekt aplikována, je nový stav daného datového objektu.

Vztah *Změna_ADP* říká, že nový stav datového pohledu je ovlivňován jeho předchozím stavem, stavem datového objektu a případně i událostí, která byla vygenerována uživatelem.

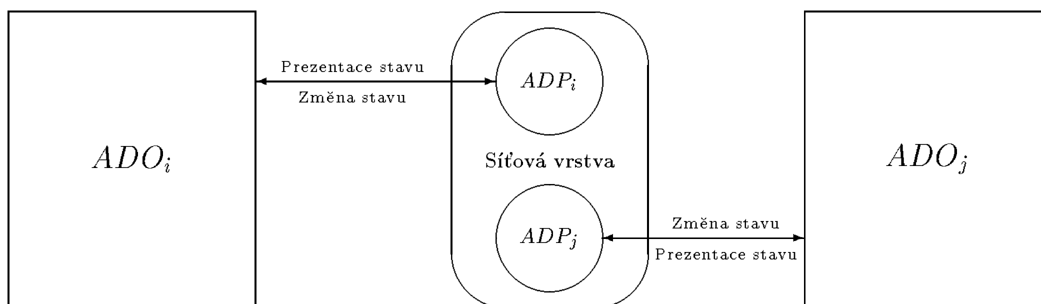
Přenos_Události_ADP vyjadřuje následující chování datových pohledů: Událost vygenerovaná uživatelem je zachycena aktivním abstraktním datovým pohledem a v závislosti na aktuálním stavu tohoto pohledu a stavu datového objektu, který je s pohledem asociován, se provede transformace události na odpovídající operaci manipulující s ADO.

Poslední vztah definice, *Odkazování_ADP*, říká, že vstupní událost v závislosti na stavu ADP a ADO může generovat novou událost pro ten samý datový pohled nebo tato událost může být transferována do jiného ADP nebo může být tato událost ignorována. Příkladem může být například stav, kdy událost „uzavři ADP“ zaslaná rodičovskému ADP je tímto ADP rozeslána i všem dětským datovým pohledům.

Konečně řádek nazvaný *Invarianty* zdůrazňuje skutečnost, že ne všechny kombinace stavů ADP a ADO jsou platné. Vlastní vztah pro určení platnosti resp. neplatnosti dané kombinace stavů závisí na implementaci modelu.



Obr. 4: Podpora tvorby jednosměrného rozhraní mezi dvěma ADO



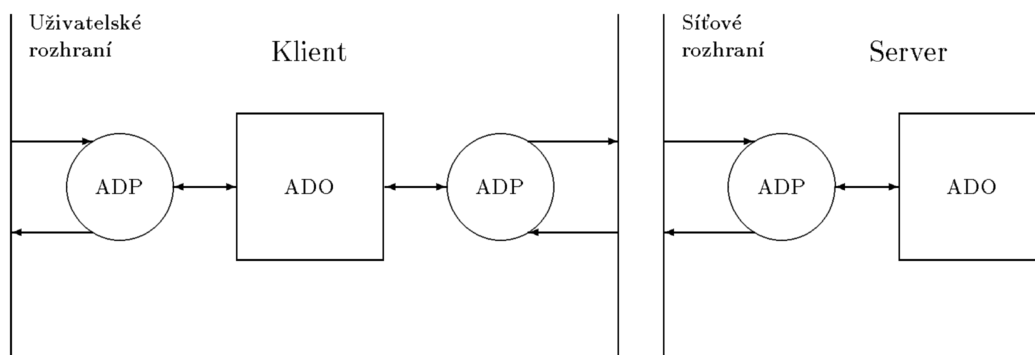
Obr. 5: Podpora tvorby obousměrného rozhraní mezi dvěma ADO

2.1.3 Zobecnění ADP — prezentace po síti

Původně byl model abstraktního datového pohledu určen pouze pro vytváření uživatelského rozhraní aplikace zcela odděleného od jejího jádra, kde ADP sloužil ke grafické prezentaci ADO na monitoru. V této kapitole bude provedeno rozšíření tohoto modelu tak, aby byla umožněna prezentace datových objektů navzájem mezi sebou a to i v různých výpočetních prostředích.

Již při specifikaci grafického abstraktního datového pohledu bylo možné si povšimnout, že ADP slouží jako jistý transformátor, který transformuje údaje získané od uživatele na data resp. operace srozumitelné pro ADO a naopak. Za této situace vznikla myšlenka, proč by nemohl být uživatel nahrazen někým či spíše něčím jiným. Vždyť velmi často nastává případ, kdy mezi sebou komunikují dvě aplikace, a konečně nejsou vzácné ani případy, kdy dochází k vzájemné komunikaci mezi datovými objekty uvnitř jedné aplikace. Pokusil jsem se tedy rozšířit možnosti použití modelu ADP i na tyto případy.

Hlavním požadavkem kladeným na tento typ ADP tedy bude schopnost komunikace mezi dvěma ADP. Každý abstraktní datový pohled bude asociován se svým datovým objektem a bude zajišťovat jeho prezentaci vzhledem k okolnímu světu. Tento svět je pak představován druhým ADP. Vzájemná komunikace mezi



Obr. 6: Implementace modelu klient–server pomocí ADP

pohledy, která se odehrává v nějakém společně dohodnutém protokolu, následně zajišťuje vzájemnou transformaci a synchronizaci stavu datových objektů.

Typ komunikace respektive prezentace mezi ADP může být rozdělen do dvou různých kategorií. Buď se může jednat o komunikaci jednosměrnou, nebo může jít o komunikaci obousměrnou. Obě dvě kategorie mají v jistých situacích své opodstatnění, podobně jako grafické pohledy, které mohou buď ADO pouze prezentovat, nebo mohou také přijímat informace od uživatele.

Příklad jednosměrné komunikace mezi abstraktními datovými objekty je znázorněn na obrázku č. 4. Příklad obousměrné komunikace, která je v síťovém prostředí velmi podobná jednosměrné komunikaci, je představen na obrázku č. 5. V obou případech je abstraktní datový objekt svázán se svým pohledem, který jej prezentuje navenek nebo naopak přijímá podněty z vnějšku, které mají vliv na jeho stav.

Jak už bylo výše zmíněno, pro komunikaci mezi dvojicí ADP je použito nějakého dohodnutého protokolu. Tento protokol ovšem není předem určen a nejsou na něj kladena žádná předběžná omezení. Všechna tato rozhodnutí jsou ponechána až na vlastní implementaci ADP. Podobně až implementace určí, zda komunikace mezi pohledy bude probíhat pouze uvnitř jedné aplikace nebo na lokálním stroji, lokální síti nebo dokonce v nějakém ještě rozsáhlejším síťovém prostředí.

Díky takto pojatému návrhu rozšíření ADP je možné, aby mezi sebou komunikovaly nejen objekty uvnitř jediné aplikace, ale také, aby mohlo docházet k vzájemné komunikaci mezi více aplikacemi a to případně i na různých strojích navzájem propojených síťovým rozhraním. Přitom vzhledem k jádru aplikace je tato komunikace zcela transparentní.

Příkladem může být schéma, které se nachází na obrázku č. 6 a zobrazuje, jakým způsobem by pomocí abstraktních datových pohledů mohlo být implementováno paradigma klient–server. Z obrázku je zřejmé, že existují dvě aplikace — klient a server. Aplikace klienta má možnost komunikace přes různá ADP jak s uživatelem tak se serverem. Aplikace serveru je pak připravena s klientem spolupracovat a okamžitě po přijetí požadavku jej vykonat a výsledek předat

$Prezentace$:$(Stav_ADP \times Stav_ADO) \rightarrow Stav_Příjemce$
$Chování_ADO$:$(Stav_ADO \times Operace_ADO) \rightarrow Stav_ADO$
$Změna_ADP$:$(Stav_ADP \times Stav_ADO \times Sys_Událost) \rightarrow Stav_ADP$
$Přenos_Události_ADP$:$(Stav_ADP \times Stav_ADO \times Sys_Událost) \rightarrow Operace_ADO$
$Odkazování_ADP$:$(Stav_ADP \times Stav_ADO \times Sys_Událost)$
$\rightarrow Sys_Událost \times Seznam_ADP$
$Invarianty$:$(Stav_ADP \times Stav_ADO) \rightarrow Boolean$

Obr. 7: Formální model rozšířeného abstraktního datového pohledu

zpět.

Výhoda takto navrženého rozhraní je tedy zřejmá. Jednotlivé datové objekty, které jsou spolu propojeny, nemají na sebe žádné přímé odkazy a navzájem neví nic ani o svém umístění ani o své existenci. Tato skutečnost následně dovoluje použít ADO v různých prostředích, což opět zvyšuje znovupoužitelnost.

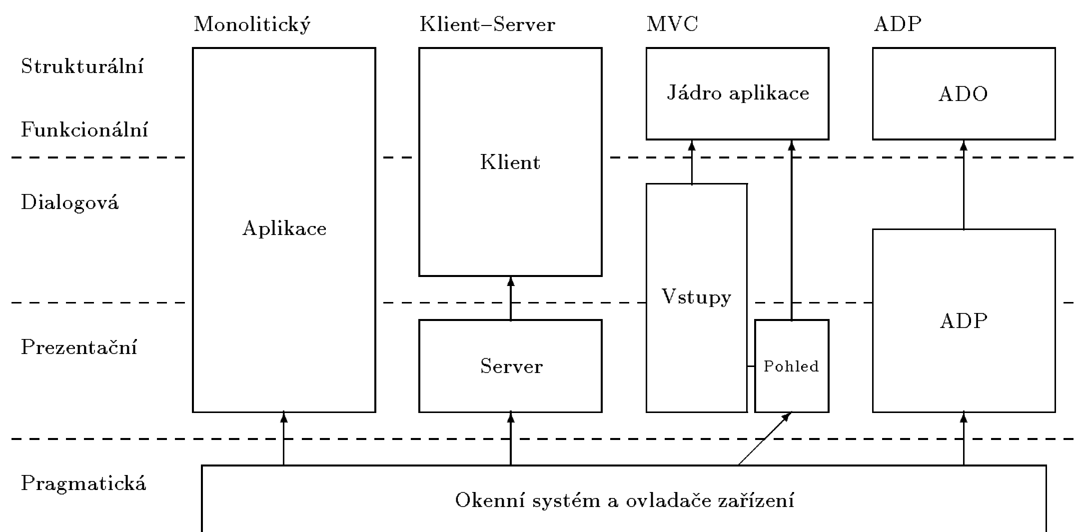
Formální specifikace upraveného modelu ADP, která je uvedena na obrázku č. 7, nedoznává vzhledem ke specifikaci uvedené na obrázku č. 2 žádných velkých změn. Snad jediný podstatnější rozdíl nastává u vztahu *Prezentace*, který nyní vyjadřuje skutečnost, že stav přijímajícího ADP je důsledkem jak stavu datového vysílajícího pohledu, tak stavu datového objektu, který je s tímto pohledem asociován. Důsledkem změny stavu přijímajícího pohledu pak je, že dojde na základě definice *Změna_ADP* i ke změně stavu ADO asociovaného s tímto příjemcem. Ostatní vztahy v modelu zůstávají zachovány beze změn.

2.2 Klasifikace modelů pro návrh uživatelského rozhraní

V literatuře je popsáno několik schémat pro klasifikaci modelů pro návrh uživatelského rozhraní aplikací. V této kapitole se zaměřím na klasifikaci podle schématu, které je použito například v [1] a [9]. Toto schéma rozděluje rozhodnutí, která mají být vykonána během návrhu uživatelského rozhraní, do následujících pěti tříd:

1. Strukturální
2. Funkcionální
3. Dialogová
4. Prezentační
5. Pragmatická

Třída strukturální zahrnuje rozhodnutí, která určují strukturu objektů, se kterými bude manipulováno přes uživatelské rozhraní, a jejich vzájemné vztahy.



Obr. 8: Srovnání architektur pro návrh aplikací

Funkcionální třída určuje funkce, které budou použity při manipulaci s objekty navrženými v předchozí třídě, jejich syntaxi a sémantiku. Proto by u každé funkce měl být uveden popis vstupů, výstupů a případně chybových stavů.

Rozhodnutí dialogové třídy určují obsah a posloupnost zpráv respektive informací, které budou zabezpečovat komunikaci mezi uživatelem a systémem. Tyto informace lze rozdělit do dvou podkategorií a to na informace vyměňované mezi uživatelem a uživatelským rozhraním aplikace a na informace vyměňované mezi uživatelským rozhraním a výkonným jádrem aplikace.

Prezentační třída specifikuje interaktivní objekty, které budou tvořit uživatelské rozhraní. Tato specifikace zahrnuje také algoritmy, které implementují zobrazování uživatelského rozhraní a algoritmy zpracovávající uživatelský vstup.

Pragmatická třída pak obsahuje všechna rozhodnutí, která jsou závislá na hardware případně na specifických vlastnostech operačního systému.

V následujících kapitolách se zaměřím na charakteristiku čtyř různých modelů pro návrh aplikací. Každý model má své specifické rysy a je vhodný pro jinou oblast použití. Na obrázku č. 8 jsou graficky znázorněny jednotlivé modely včetně rozdělení jejich komponent do jednotlivých tříd rozhodování při návrhu.

2.2.1 Monolitický model

Monolitický model vůbec nerozlišuje mezi třídami jednotlivých rozhodnutí při návrhu uživatelského rozhraní aplikace. Vše od strukturální po prezentační třídu je navrženo a implementováno jako jeden kompaktní celek. Tento model architektury systému není doporučován pro návrh interaktivních systémů ani systémů jiných. V tomto případě totiž dochází k tomu, že navržená struktura aplikace

je velmi obtížně modifikovatelná a znovupoužitelná, protože zásah do libovolné části aplikace může vést k následným zásahům do všech zbývajících částí.

2.2.2 Model klient – server

Model klient – server patří k těm častěji používaným modelům. Tento model rozděluje jedinou komponentu monolitického modelu na komponenty dvě: klienta a server. Hranice mezi tím, co má být součástí klienta a co součástí serveru, není přesná a její určení při návrhu vyžaduje jistých zkušeností. Tento model je nejčastěji používán pro řešení komunikačních a síťových problémů. Jeho použití pro návrh interaktivních aplikací již není tak časté.

2.2.3 Model MVC

Toto schéma (Model View Controller) rozděluje systém do tří komponent a to modelu datových struktur, objektu řízení vstupů a zobrazovacích pohledů. Zobrazovací pohledy podporují pouze výstup informací směrem k uživateli a proto jsou součástí pouze prezentační třídy. Komponenta řízení vstupů zajišťuje příjem uživatelských příkazů a koordinuje přenos těchto informací mezi uživatelem a datovým jádrem aplikace. Z tohoto důvodu jsou při návrhu řízení vstupů použita rozhodnutí jak z dialogové, tak z prezentační třídy. Struktury z jádra aplikace komunikují s obrazovkou (zobrazovacími pohledy) i se vstupními zařízeními (komponentami řízení vstupů). Běh aplikace je pak určen tím, že uživatel vykoná nějakou vstupní akci a ta je zachycena na vstupu. Následuje změna stavu datových struktur jádra a informace o těchto změnách jsou rozeslány všem pohledům. Pohledy tedy nemusí monitorovat data, stačí jen čekat na zprávu o změně. Tento koncept je vhodný a také často používaný pro návrh interaktivních aplikací.

2.2.4 Model ADP

Model ADP striktně odděluje vlastní jádro aplikace od uživatelského rozhraní. Typický systém, který je založen na modelu abstraktních datových pohledů, je složen z množiny abstraktních datových objektů (ADO), které zajišťují uchování stavu aplikace během jejího vykonávání, a seznamu pohledů, které zajišťují prezentaci stavu aplikace, zachytávání událostí a zajišťují také jejich vykonání, případně transformaci na jiné akce. Abstraktní datové objekty tedy implementují rozhodnutí zařazená do třídy funkcionální a strukturální, naproti tomu ADP implementují činnosti z třídy prezentační a částečně také dialogové. Podrobnosti o chování tohoto modelu lze nalézt v kapitole 2.1 nebo v [1].

Důsledkem takto pojatého návrhu aplikace je, že ADO je zcela nezávislý na uživatelském rozhraní aplikace a nepotřebuje k tomuto rozhraní žádný přístup. Naopak ADP zajišťuje kontrolu veškerých vstupně–výstupních operací a následně takto plně kontroluje asociovaný ADO.

Při srovnání modelu ADP s předchozími modely lze konstatovat, že komunikace mezi komponentami modelu ADP nastává v okamžiku, kdy vrstva pohledů synchronně volá funkce z vrstvy datových objektů. V tomto modelu tedy ADO nikdy negenerují události, které by následně musely být asynchronně zachyceny a zpracovány některými ADP. Naopak v ostatních srovnávaných modelech často dochází k tomu, že jednotlivé komponenty musejí zpracovávat i asynchronní události, což je implementačně dost náročné.

Důvodem takového chování je, že ve většině aplikací je centrum ovládání aplikace umístěno ve vrstvě strukturální, zatímco u modelu ADP je centrum ovládání ve vrstvě prezentační. V zásadě lze říci, že zatímco většina aplikací je psána způsobem, kdy uživatel čeká na akci aplikace, v modelu ADP naopak aplikace čeká na akci uživatele. Proto je použití tohoto konceptu obzvláště vhodné při návrhu vysoce interaktivních aplikací.

3 VDM — formální metoda pro vývoj aplikací

VDM (Vienna Development Method) je jednou z metod pro formální návrh a vývoj nového software. Specifikace VDM není ekvivalentem programovacího jazyka a jako taková není ani určena k přímé interpretaci na počítači. Dává však vývojáři možnost snadného použití predikátové logiky při popisu návrhu aplikace. Vzhledem k tomu, že VDM používá pro specifikaci návrhu aplikace matematické notace, je tento návrh univerzální a lehce přenositelný mezi různými programovými platformami.

Vývoj software pomocí metody VDM sestává z následujících fází:

- analýza požadavků ze zadání a následná tvorba datového modelu a operací
- úprava datového modelu tak, aby splňoval i konkrétní podrobnosti zadání
- úprava operací tak, aby přesně vyhovovaly konkrétnímu datovému modelu
- transformace konkrétního datového modelu na datové typy programovacího jazyka
- transformace konkrétních operací na programový kód

3.1 Základní konstrukce VDM

V následujících podkapitolách budou stručně popsány základní konstrukce notace VDM a uvedeno i jejich případné použití. Tento popis si neklade za cíl vysvětlovat detaily. Spíše má sloužit jako stručná referenční příručka notace.

3.1.1 Úvod do matematické notace

Jak už bylo v úvodu řečeno, při zápisu notace VDM se hojně využívá predikátové logiky a teorie množin. Predikátová logika tvoří základ každé formální metody. Umožňuje totiž odvozování a dokazování vztahů. Nejprve bych se však zmínil o tom, co to je množina a jakým způsobem se v této notaci množiny a operace nad nimi zapisují.

Množina je neuspořádaná kolekce prvků ve které se nevyskytují dva stejné prvky. Zápis množiny může být buď explicitní, např. $\{1, 3, 5\}$, nebo implicitní, jako např. $\{x \in \mathbf{N} \mid x < 100\}$. Pro zápis operací nad množinami se používá standardních množinových operátorů, jako jsou $\in$, $\notin$, $\subset$, $\subseteq$, $\cup$, $\cap$, $-$, $\times$ a card, které v tomto pořadí značí operace pro náležitost resp. nenáležitost prvku do množiny, ostrou resp. neostrou inkluzi, sjednocení, průnik a rozdíl množin, kartézský součin množin a kardinalitu (mohutnost) množiny.

V notaci VDM existuje několik množin, které mají svůj název již předdefinován. Jsou to:

- množina přirozených čísel $\mathbf{N} = \{1, 2, 3, \dots\}$
- množina přirozených čísel s nulou $\mathbf{N}_0 = \{0, 1, 2, \dots\}$
- množina celých čísel $\mathbf{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- množina racionálních čísel $\mathbf{Q}$
- množina reálných čísel $\mathbf{R}$
- množina logických hodnot $\mathbf{B} = \{\textsf{false}, \textsf{true}\}$
- množina znaků $Char$; obsahuje velká a malá písmena abecedy, číslice a další znaky

Relace mezi dvěma množinami A a B je definována jako množina uspořádaných dvojic, ve kterých je první prvek dvojice prvkem množiny A a druhý prvkem množiny B. Lze tedy říci, že relace je podmnožinou kartézského součinu $A \times B$. Speciálním případem relace je pak funkce nebo operace.

Ve specifikaci jsou také předdefinovány všechny běžné aritmetické a logické operátory. Tradičně bývá výsledkem logických operátorů hodnota true nebo false. V notaci VDM přistupuje k těmto dvěma hodnotám někdy ještě třetí hodnota nazvaná undefined. Této hodnoty se používá během dokazování správnosti zápisu specifikace, kdy ne všechny výrazy musí být na počítači za všech okolností řešitelné.

3.1.2 Modelování typů a operací

Základem metody VDM je pojem **datový typ**. Ten je definován jako množina hodnot spolu s množinou operací, které jsou nad těmito hodnotami definovány. Například datový typ *integer*, který je znám téměř ze všech programovacích jazyků, je definován jako množina celých čísel $\{-32768, \dots, 32767\}$ spolu s operacemi, jako jsou sčítání, odčítání, násobení, dělení atd. Datové typy, které jsou ve VDM již předdefinovány, se nazývají **primitivní datové typy**. Primitivními datovými typy jsou množiny $\mathbf{N}$, $\mathbf{N}_0$, $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{B}$ a $Char$ spolu s příslušnými aritmetickými a logickými operátory.

Pro potřeby definování nového datového typu, existuje ve VDM jednoduchá syntaktická konstrukce. Pokud bychom chtěli například vytvořit datový typ, jehož prvky jsou jména dnů v týdnu, zápis by vypadal následovně:

types

$$Dny\text{-}Týdne = \{\textsc{Pondělí}, \textsc{Úterý}, \textsc{Středa}, \textsc{Čtvrtek}, \textsc{Pátek}, \textsc{Sobota}, \textsc{Neděle}\}$$

V současnosti je většina v praxi používaných programovacích jazyků imperativního typu. Cílem imperativního programu je transformace dat uložených v proměnných a algoritmus je pak popisem průběhu této transformace dat v čase. Tuto skutečnost musí vzít na vědomí i notace VDM. Proto je uložení dat ve VDM modelováno pomocí takzvaných externích proměnných. **Externí proměnná** je globální proměnná, která může být modifikována jen operacemi VDM. Systém všech externích proměnných se pak nazývá **stav aplikace**.

Operace ve VDM jsou velmi podobné procedurám nebo funkcím v klasických programovacích jazycích. Mohou pracovat s externími proměnnými, mohou mít vstupní a případně i jeden výstupní parametr. Typ přístupu operace k externím proměnným musí být v notaci VDM explicitně vyjádřen. Existují dva typy přístupu:

- externí proměnná je určená pouze ke čtení (rd)
- externí proměnná je určená ke čtení i modifikaci (wr)

Každá externí proměnná, která má být uvnitř operace přístupná, musí být deklarovaná pomocí klíčového slova ext, musí být pro ni určen typ přístupu a musí být také určen datový typ této proměnné.

Deklarace operace v notaci VDM se skládá ze čtyř následujících částí:

1. Hlavička operace — obsahuje jméno operace, seznam vstupních parametrů v závorkách a případný výstupní parametr.
2. Deklarace externích proměnných — nepovinná část deklarace, která je složena z klíčového slova ext následovaného seznamem externích proměnných s určením datového typu a typu přístupu.
3. Vstupní predikát — nepovinná část deklarace, která obsahuje klíčové slovo pre a predikát, který musí být pravdivý, aby bylo možné vyhodnotit výstupní predikát.
4. Výstupní predikát — po klíčovém slově post následuje predikát, který určuje nové hodnoty externích proměnných a výstupního parametru v případě, kdy je pravdivý.

Obecný tvar operace se vstupními i výstupním parametry má následující tvar:

Jméno(*parametry*: *Typ*) *výsledek*: *Typ*
ext wr *modifikovatelné proměnné*: *Typ*
rd *proměnné jen pro čtení*: *Typ*
pre *vstupní predikát*
post *výstupní predikát*

Některé části tohoto zápisu mohou být vynechány. To se týká již zmiňovaných sekcí **ext** a **pre**, ale také třeba zbytečných parametrů v hlavičce operace. Zde může být zcela bez náhrady vynechán výstupní parametr, avšak pokud operace nemá žádný vstupní parametr, závorky za názvem operace zůstávají.

Uvažme případ operace, která násobí hodnotu externí proměnné hodnotou argumentu operace. Zápis této operace by mohl vypadat takto:

$Násob(koef\colon \mathbf{R})$
ext   wr x: $\mathbf{R}$
pre $x < 16384$
post $x = koef \times \overleftharpoon{x}$

V tomto příkladě klauzule **ext** říká, že stav proměnné x je možné modifikovat, klauzule **pre** zaručuje, že operaci bude možno vykonat pouze v případě, kdy platí $x < 16384$, a v klauzuli **post** se vyhodnocuje nový stav proměnné x. Vzhledem k tomu, že v klauzuli **post** se může pracovat a také pracuje jak s původní hodnotou x, tak s nově počítanou hodnotou x, existuje konvence, že odkaz na původní hodnotu proměnné bude naznačen vodorovnou čárkou s hrotem nad jménem této proměnné ($\overleftharpoon{x}$). V klauzuli **pre** se vždy pracuje s původní hodnotou parametrů a externích proměnných a proto není třeba této konvence používat.

3.1.3 Modelování množinových typů

Množinový typ je datový typ, jehož prvky jsou množiny. Pro vytváření nových datových typů existuje ve specifikaci VDM vždy nějaký konstruktor. Tento konstruktor definuje daný datový typ pomocí jiných typů či hodnot.

Nechť tedy T je již definovaný datový typ. Potom lze vytvořit datový typ množina, jehož prvky jsou všechny konečné množiny prvků typu T, pomocí následující syntaktické konstrukce:

T-set

Nad prvky tohoto množinového datového typu lze provádět všechny běžné množinové operace tak, jak už byly jmenovány dříve v kapitole 3.1.1.

Příkladem použití může být například definice množiny všech konečných podmnožin celých čísel, která má tento tvar:

types
  $Intset = \mathbf{Z}$-set

Omezení prvků datového typu množina jen na konečné podmnožiny je způsobeno tím, že na počítači je možno pracovat vždy jen s hodnotami konečné velikosti. Protože se předpokládá, že vše, co bude specifikováno pomocí VDM bude následně implementováno na počítači, je toto omezení zabudováno již do samotné specifikace.

3.1.4 Modelování sekvenčních typů

Sekvence je konečná, uspořádaná kolekce s žádným, jedním nebo více prvky. Je to tedy obdobný typ jako množina, ale na rozdíl od množin záleží u sekvencí na pořadí prvků a také se uvnitř sekvence může vyskytovat jeden prvek vícekrát.

Pokud je třeba použít explicitní zápis sekvence, uzavírají se prvky sekvence do hranatých závorek. Například

['a', 'b', 'c']

představuje tříprvkovou sekvenci znaků. Tak jako u množinového typu, existuje i u sekvenčního typu konstruktor. Ten má následující syntaxi:

T^*

kde T je tzv. **bázový typ**. Prvky takového sekvenčního datového typu jsou všechny konečné sekvence prvků typu T.

Speciálním případem sekvencí jsou sekvence znaků. Tyto sekvence se nazývají **řetězce** a pro jejich explicitní zápis je možné použít speciální konstrukce, kde sekvenci ['A', 'H', 'O', 'J'] lze zapsat jako "AHOJ".

Pro práci se sekvencemi je ve VDM definováno několik funkcí:

Délka sekvence: Pro zjištění počtu prvků v sekvenci slouží funkce len. Například len [25, 30] = 2.

Indexování sekvence: Když s je sekvence, pak i-tý prvek sekvence s lze získat pomocí zápisu $s(i)$. Prvky sekvence jsou indexovány 1, 2, ..., len s. Pokud je index mimo tento rozsah, je výsledek nedefinován. Například výsledkem výrazu [1, 5, 10](2) je číslo 10.

Subsekvence: Pokud s je sekvence, i a j jsou platné indexy této sekvence a platí $i \leq j$, pak subsekvenci od i-tého do j-tého prvku sekvence s lze získat zápisem $s(i, \ldots, j)$. Pokud $i > j$ je výsledná subsekvence prázdná [ ]. Pokud některý z indexů i a j není platný, je výsledek nedefinován. Například výsledkem výrazu "počítač"(3, ..., 7) je slovo "čítač".

Slučování: Když s a t jsou dvě sekvence, pak $s \frown t$ je sekvence sloučená. Například "Pokus"$\frown$['y'] se vyhodnotí na sekvenci "Pokusy".

Přepisování: Operátor † může být použit pro přepsání hodnoty některého elementu sekvence. Je to binární operátor; levým argumentem je indexovaná sekvence a pravým argumentem je hodnota, která má přepsat původní hodnotu. Například výsledkem [1, 5, 8](2) † 10 je sekvence [2, 10, 8].

První prvek sekvence: Každá neprázdná sekvence se skládá z prvního prvku sekvence a zbytku sekvence bez prvního prvku. Proto ve VDM existují

funkce, které umožňují přístup k těmto částem sekvence. První z nich je funkce hd, která má jeden argument typu sekvence a jejímž výsledkem je první prvek sekvence. Například hd ['a', 'b', 'c'] se vyhodnotí na prvek 'a'. Pro prázdnou sekvenci [ ] není výsledek funkce definován.

Zbytek sekvence: Opakem funkce hd je funkce tl. Ta má také jako argument sekvenci, ale výsledkem funkce je zbytek sekvence bez prvního prvku. Například tl ['a', 'b', 'c'] se vyhodnotí na ['b', 'c']. Pro prázdnou sekvenci není výsledek opět definován.

Přidání prvku na začátek sekvence: Podobně jako funkce hd a tl rozdělují sekvenci na první prvek a zbytek sekvence, existuje funkce cons, která naopak z prvního prvku a zbytku vytvoří původní sekvenci. Například cons(1, [2, 3]) se vyhodnotí na sekvenci [1, 2, 3].

Množina indexů: Často je třeba pracovat s množinou všech indexů nějaké sekvence. K tomuto účelu je ve VDM definována funkce inds. Nechť s je sekvence, pak funkce inds je definována následovně:

$$\mathsf{inds}\ s = \{i \in \mathbf{N} \mid 1 \leq i \leq \mathsf{len}\ s\}$$

Hodnota výrazu inds [ ] není definována. Například inds ['a', 'b', 'c'] se vyhodnotí na množinu $\{1, 2, 3\}$.

Množina prvků: Pro případy, kdy je třeba získat množinu všech prvků, které se v nějaké sekvenci vyskytují slouží funkce elems. Když s je sekvence, pak funkce elems je definována následovně:

$$\mathsf{elems}\ s = \{s(i) \in T \mid i \in \mathsf{inds}\ s\}$$

Výsledkem výrazu elems [ ] je prázdná množina { }. Například výraz elems "anakonda" se vyhodnotí na množinu {'a', 'n', 'k', 'o', 'd'}.

3.1.5 Specifikace implicitních funkcí

Implicitní funkce v notaci VDM jsou určeny k tomu, aby matematicky popsaly relaci mezi definičním oborem a oborem hodnot funkce. Výhodou implicitních funkcí je, že jejich matematický popis blíže nespecifikuje algoritmus, který by měl být použit při výpočtu, ale soustředí se pouze na popis toho, co je třeba vypočítat.

Specifikace implicitní funkce má následující komponenty:

1. hlavičku, která je složena z:
   - jména funkce
   - uzávorkovaného seznamu parametrů s určením jejich typů

- výsledkového parametru s určením typu

2. vstupní predikát, který je volitelný a je uvozen klíčovým slovem pre
3. výstupní predikát uvozený klíčovým slovem post

Obecný tvar specifikace implicitní funkce je pak následující:

Jméno_funkce($parametr_1$: Typ_1, $parametr_2$: Typ_2, ...) *výsledek*: *Typ*
pre *vstupní predikát*
post *výstupní predikát*

Formát specifikace implicitní funkce se částečně podobá specifikaci operace, jak byla uvedena v kapitole 3.1.2. Jedná se vlastně o speciální případ operace. Na rozdíl od operace však implicitní funkce nemá klauzuli ext pro práci s externími proměnnými a výstupní parametr funkce je zde povinný.

Postup při vyhodnocování funkce je následující: Nejprve se vyhodnotí vstupní predikát. Pokud je pravdivý, pokračuje se s vyhodnocováním výstupního predikátu. Pokud pravdivý není, hodnota výstupního parametru funkce není definována. Když vstupní predikát nebyl specifikován, předpokládá se, že je pravdivý. Výpočet výstupního parametru probíhá způsobem, že se hledá hodnota tohoto parametru taková, aby výstupní predikát byl pravdivý.

Následující implicitní funkce například vypočítá hodnotu maximálního prvku z nějaké množiny celých čísel:

Maximum_množiny(s: $\mathbf{N}$-set) r: $\mathbf{N}$
pre $s \neq \{\}$
post $r \in s \ \wedge (\forall n \in s \ \bullet n \leq r)$

Vstupní predikát zde říká, že funkce je definována jen pro neprázdné množiny, a výstupní predikát určuje, jakou podmínku musí splňovat výstupní parametr.

Při návrhu funkce nebo operace je třeba dbát na to, aby podmínka v klauzuli pre byla co nejvíce restriktivní, pokud možno s co nejvíce konjunkcemi. Naopak podmínka v klauzuli post má být co nejslabší. Při dodržení těchto doporučení lze lépe provádět dokazování správnosti funkce.

Operace nebo funkce se nazývá implementovatelná, jestliže pro každou platnou množinu vstupních dat existuje platný výstup. Při specifikaci operací a funkcí je třeba dbát na to, aby byly implementovatelné a současně aby zachovávaly invarianty datových typů.

3.1.6 Specifikace explicitních funkcí

V předchozí kapitole byl popsán způsob specifikace implicitních funkcí. Nyní se soustředíme na popis explicitních funkcí. Zápis těchto funkcí je mnohem více algoritmický, nevyskytuje se zde ani vstupní ani výstupní predikát, ale místo nich je

tělo funkce složeno z jistých programových konstrukcí, které popisují algoritmus, kterým by mohl, ale nemusel být vypočítán výsledek funkce.

Specifikace explicitně definované funkce se skládá z následujících částí:

1. signatury, která je složena z:
    - jména funkce
    - dvojtečky
    - určení typů vstupních parametrů funkce ve tvaru kartézského součinu, tj. navzájem oddělených znakem $\times$
    - symbolu $\rightarrow$
    - určení typu výstupního parametru
2. hlavičky složené z:
    - jména funkce
    - uzávorkovaného seznamu jmen vstupních parametrů
3. symbolu $\triangleq$, který znamená „je definována jako“
4. výrazu popisujícího algoritmus

Obecný formát explicitní definice funkce má tvar:

jméno_funkce: *typy_definičních_oborů* $\rightarrow$ *typ_výsledku*
jméno_funkce(*seznam_argumentů*) $\triangleq$
    algoritmický_zápis

Příkladem zápisu explicitní funkce může být následující funkce pro součet dvou reálných čísel:

součet: $\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$
součet$(x, y) \triangleq$
    $x + y$

Jak už bylo řečeno, u explicitně definovaných funkcí se používá pro popis algoritmu pro výpočet výsledné hodnoty několika syntaktických konstrukcí.

Výraz if–then–else: Tento výraz umožňuje větvení algoritmu. Jeho syntaktický zápis je následující:

if *logický_výraz* then $výraz_1$ else $výraz_2$

V případě, že *logický_výraz* platí, výsledkem podmíněného výrazu je hodnota výrazu $výraz_1$, jinak je to hodnota výrazu $výraz_2$.

Výraz cases: Tento výraz umožňuje nahradit některé případy opakovaného použití výrazu if–then–else. Jeho obecný formát je následující:

$$
\begin{array}{l}
\mathsf{cases}\ index: \\
\quad \left(\begin{array}{lcl}
hodnota_1, hodnota_2 & \rightarrow & výsledek_1, \\
hodnota_3, hodnota_4, hodnota_5 & \rightarrow & výsledek_2, \\
\vdots & & \\
hodnota_n & \rightarrow & výsledek_m
\end{array} \right.\\
\quad)
\end{array}
$$

V případě, že hodnota *index* je rovna některé hodnotě vyjmenované uvnitř výrazu, pak výsledkem výrazu je hodnota uvedená za šipkou. Jinak je výsledná hodnota nedefinovaná.

Výraz let–in: Velmi často se stává, že výrazy ve specifikaci VDM jsou velmi dlouhé a komplexní. Z tohoto důvodu byla zavedena syntaktická konstrukce let–in, která zápis těchto výrazů zjednodušuje. Zápis této konstrukce má následující tvar:

$\mathsf{let}\ identifikátor = výraz_1\ \mathsf{in}\ výraz_2$

Vyhodnocování tohoto výrazu probíhá následovně:

1. Vyhodnotí se $výraz_1$ a výsledek vyhodnocení se naváže na *identifikátor*.
2. Ve výrazu $výraz_2$ se nahradí všechny výskyty identifikátoru *identifikátor* hodnotou, která na něj byla navázána v předchozím kroku.
3. Výsledkem výrazu len–in je pak hodnota upraveného výrazu $výraz_2$.

Konstrukci let–in je možné také použít pro výběr hodnoty z množiny a její následnou aplikaci na výraz. K tomu slouží upravená syntaktická konstrukce

$\mathsf{len}\ ident \in množinový_výraz\ \mathsf{in}\ výraz$

nebo konstrukce

$\mathsf{len}\ ident \in množinový_výraz\ \mathsf{s.t.}\ logický_výraz\ \mathsf{in}\ výraz$

Obě tyto konstrukce mají své opodstatnění především při rekurzivním volání funkce, kdy s identifikátorem je postupně asociována ještě nepoužitá hodnota z dané množiny. Druhá konstrukce navíc ještě omezuje výběr jen na hodnoty, které splňují zadaný logický výraz.

Výraz for–to–do: Tento výraz umožňuje snadné opakování části algoritmu. Jeho syntaktický zápis je následující:

for *identifikátor* $\leftarrow$ *výraz*$_1$ to *výraz*$_2$
do *predikát*
end

Nejprve se vyhodnotí *výraz*$_1$ a výsledek výrazu se naváže na *identifikátor*. Potom se vyhodnotí *výraz*$_2$ a pokud je pravdivý, provede se vyhodnocení predikátu. Následuje opětovná kontrola pravdivosti výrazu a případné následné vyhodnocení predikátu. Výsledkem výrazu for–to–do je hodnota posledního vyhodnoceného predikátu.

Ukázkou deklarace explicitní funkce může být například následující funkce pro výpočet absolutní hodnoty z celého čísla:

$abs \colon \mathbf{Z} \to \mathbf{N}$
$abs(z) \triangleq$
if $z < 0$ then $-z$ else z

Pro srovnání nyní uveďme příklad stejné funkce definované implicitně:

$abs(z \colon \mathbf{Z})\ r \colon \mathbf{N}$
pre true
post $(z < 0 \wedge r = -z) \vee (z \geq 0 \wedge r = z)$

Z příkladu je zřejmé, že zápis implicitní funkce je mnohem více univerzální, ale někdy hůře čitelný. Proto by měl být používán pokud možno co nejčastěji v případech, kdy to neuškodí čitelnosti. Explicitní zápis by měl být naopak používán jen v případech, kdy jeho použití výrazně zjednoduší a zprůhlední definici.

3.1.7 Modelování strukturovaných typů

Strukturované typy slouží pro sdružování proměnných různých typů do jednoho kompaktního celku. Specifikace složeného typu má následující tvar:

Složený_typ ::
položka$_1$: *Typ*$_1$
položka$_2$: *Typ*$_2$
...

a je tedy velmi podobná například typu **struct** z jazyka C++.

Specifikace typu *Datum* by pak mohla vypadat následovně:

Datum ::
den: $\{d \in \mathbf{N} \mid d \leq 31\}$
měsíc: $\{m \in \mathbf{N} \mid m \leq 12\}$
rok: $\{r \in \mathbf{Z}\}$

Přístup k jednotlivým položkám strukturovaného typu se provádí pomocí operátoru tečka. Pokud je například d proměnná typu *Datum*, pak k její položce *den* lze přistoupit pomocí zápisu $d.den$.

V praxi je často třeba explicitně vyjadřovat hodnoty daného typu. Protože pro strukturovaný typ neexistuje žádný zápis pomocí složených nebo hranatých závorek, jak tomu bylo u množin a sekvencí, je pro každý strukturovaný typ standardně definována tzv. konstrukční funkce.

Když *Jméno* je název strukturovaného typu, pak konstrukční funkce se jmenuje *mk-Jméno*. Tato funkce má právě tolik argumentů, kolik je složek strukturovaného typu. Pořadí a typy argumentů odpovídají pořadí a typům složek daného strukturovaného datového typu. Například konstantu 1.1.1996 typu *Datum* lze psát *mk-Datum*(1, 1, 1996).

U některých, především strukturovaných datových typů není specifikace zcela dostatečná. Například definice typu *Datum* umožňuje pracovat i s datem 30.2.1996 a přitom je zřejmé, že toto datum není platným prvkem množiny skutečných kalendářních dat. Z tohoto důvodu byl zaveden dodatečný predikát nazývaný **invariant**, který může zabezpečit ošetření i takových případů.

Když T je nějaký datový typ, pak invariant tohoto typu má následující zápis:

$$\textit{inv-T}\colon T \to \mathbf{B}$$
$$\textit{inv-T}(\mathrm{t}) \triangleq \ldots$$

Prvkem datového typu T pak mohou být jen ty hodnoty, které splňují invariant datového typu.

Velmi často nastává případ, kdy je třeba vytvořit kopii proměnné strukturovaného typu a přitom modifikovat některé její složky. Pro tento účel byla ve specifikaci VDM zavedena funkce μ. Tato funkce má jako první argument hodnotu, ze které má být vytvořena kopie a další argumenty určují, jak se mají jednotlivé složky kopie modifikovat. Příklad použití této funkce vypadá následovně:

$$d = \textit{mk-Datum}(1,\ 1,\ 1996)$$
$$\mu(d,\ \textit{den} \mapsto 2,\ \textit{měsíc} \mapsto 5)$$

V tomto případě je výsledkem volání funkce μ datum 2.5.1996.

Při používání strukturovaných typů je často třeba pracovat s lokálními proměnnými, kterým jsou přiřazeny hodnoty složek strukturované proměnné. Pro zjednodušení tohoto přiřazení existuje ve VDM speciální možnost zápisu. Zápis

$$\mathsf{let}\ \textit{datum_narození} = \textit{mk-Datum}(d,\ m,\ r)\ \mathsf{in}\ \ldots$$

je ekvivalentní složitějšímu

$$\mathsf{len}\ \textit{date.den} = d,\ \textit{date.měsíc} = m,\ \textit{date.rok} = r\ \mathsf{in}\ \ldots$$

3.1.8 Modelování asociovaných typů

Nechť A a B jsou dvě množiny a $A \times B$ je kartézský součin těchto množin. Potom mapování m z A do B je konečná podmnožina kartézského součinu $A \times B$, pro kterou platí, že žádné dvě různé uspořádané dvojice v této podmnožině nemají stejný první prvek. Například

$$\{a_1 \mapsto b_1,\ a_2 \mapsto b_2,\ a_3 \mapsto b_1\}$$

kde $a_i \in A$, $b_j \in B$, je explicitní zápis mapování z A do B.

Asociovaný datový typ množin A a B v tomto pořadí je množina všech mapování z A do B. Pro zápis asociovaného datového typu se používá syntaktické konstrukce $A \xrightarrow{m} B$. Prvky asociovaného typu $\mathbf{N} \xrightarrow{m} \mathbf{N}$ jsou například:

$$\{\} \qquad \{1 \mapsto 15\}$$

Prvek asociovaného typu lze také určit implicitním zápisem pomocí predikátu. Například pro asociovaný typ $\mathbf{N} \xrightarrow{m} \mathbf{N}$ prvek

$$\{x \mapsto y \mid y = x^2 \wedge x < 100\}$$

určuje mapování, které každému přirozeném číslu menšímu než sto přiřadí jeho druhou mocninu.

Pro práci s asociovanými typy je ve VDM definováno několik funkcí:

Funkce dom: Toto je funkce, která pro mapování m vrací množinu prvků, které se nachází na prvním místě v uspořádaných dvojicích. Lze ji definovat také takto:

$$\mathsf{dom}\colon A \xrightarrow{m} B \to A\text{-}\mathsf{set}$$
$$\mathsf{dom}(m) \triangleq \{a \in A \mid \exists b \in B \bullet ((a \mapsto b) \in m)\}$$

Pokud je $a \in \mathsf{dom}(m)$, pak asociovaný prvek $b \in B$ lze vyjádřit zápisem $m(a)$ a tehdy řekneme, že mapování m bylo aplikováno na hodnotu a. Pokud $a \in A$ a přitom $a \notin \mathsf{dom}(A)$ je výsledek $m(a)$ nedefinován.

Funkce rng: Tato funkce vrací množinu prvků, které se nachází na druhém místě v uspořádaných dvojicích mapování. Lze ji definovat takto:

$$\mathsf{rng}\colon A \xrightarrow{m} B \to B\text{-}\mathsf{set}$$
$$\mathsf{rng}(m) \triangleq \{b \in B \mid \exists a \in A \bullet ((a \mapsto b) \in m)\}$$

Operátor přepisování: Operátor † slouží k modifikaci některých prvků mapování. Když m a n jsou mapování stejného typu, potom výsledkem $m \dagger n$ je kopie mapování m, ve které se všechny uspořádané dvojice, které mají stejný první prvek jako nějaká jiná uspořádaná dvojice z mapování n, nahradí touto dvojicí. Prvky mapování n, které nebyly využity jsou pak přidány k novému mapování. Například

$$\{2 \mapsto 4, 1 \mapsto 3\} \dagger \{3 \mapsto 5, 1 \mapsto 2\}$$

se vyhodnotí na mapování

$$\{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 5\}$$

Operátor restrikce mapování: Tento operátor vrací jako výsledek mapování, ve kterém se nacházejí jen uspořádané dvojice, které mají první prvek uspořádané dvojice z množiny, podle které se restrikce provádí. Pro operátor restrikce se užívá znaku $\lhd$. Formálně lze tento operátor definovat takto:

$$\lhd\colon A\text{-}\mathsf{set} \times A \xrightarrow{m} B \to A \xrightarrow{m} B$$
$$s \lhd m \triangleq \{a \mapsto m(a) \mid a \in (\mathsf{dom}(m) \cap s)\}$$

Operátor rušení mapování: Tento operátor vrací jako výsledek mapování, ve kterém jsou zrušeny prvky, které mají první prvek uspořádané dvojice z množiny, podle které se rušení provádí. Pro operátor rušení se užívá znaku $⩤$. Formálně lze tento operátor definovat takto:

$$⩤\colon A\text{-}\mathsf{set} \times A \xrightarrow{m} B \to A \xrightarrow{m} B$$
$$s ⩤ m \triangleq \{a \mapsto m(a) \mid a \in (\mathsf{dom}(m) - s)\}$$

Za některých okolností je třeba přiřadit proměnné typu T hodnotu, která říká, že hodnota dané proměnné ještě nebyla určena. K tomuto účelu slouží konstanta nil. Pokud T je nějaký datový typ, pak zápis $[T]$ označuje datový typ, který se od typu T liší jen v tom, že jeho prvkem je také konstanta nil.

3.2 Koncept ADP a jeho specifikace

V této kapitole bude popsáno rozšíření specifikace VDM o objektově orientované rysy. Dále pak bude následovat úprava notace tak, aby ji bylo možno jednoduše aplikovat pro popis abstraktních datových pohledů.

3.2.1 Specifikace objektových typů

V současnosti je většina nových aplikací implementována v nějakém objektově orientovaném programovacím jazyku, jako jsou například ANSI C++, Smalltalk nebo i Pascal. Z tohoto důvodu by bylo vhodné zavést do notace VDM také nějakou možnost specifikace objektových datových typů. V této části se pokusím ukázat jeden z možných způsobů řešení tohoto problému.

V kapitole 3.1.7 je popsán strukturovaný datový typ notace VDM a z tohoto popisu se bude vycházet i při návrhu objektového datového typu. Nejprve je však třeba určit požadavky, které tento nový datový typ musí splňovat. Zřejmě se bude jednat především o požadavky na schopnost zapouzdření složek datového typu a

Specification *Objektový_typ*

declaration *proměnná*$_1$: *Typ*$_1$
proměnná$_2$, *proměnná*$_3$: *Typ*$_2$
...

inv-Objektový_datový_typ $\triangle$ *výraz*

Constructor *Jméno_konstruktoru*(*parametry*)
ext *deklarace externích proměnných*
pre *vstupní predikát*
post *výstupní predikát*

Destructor *Jméno_destruktoru*()
ext *deklarace externích proměnných*
pre *vstupní predikát*
post *výstupní predikát*

Operation *Jméno_operace*(*parametry*) *výsledek*
ext *deklarace externích proměnných*
pre *vstupní predikát*
post *výstupní predikát*
⋮

End *Objektový_typ*

Obr. 9: Schéma deklarace objektového datového typu

Specification *Objektový_typ*
  Subtype of *Rodičovský_typ*

  declaration $proměnná_1$: Typ_1
  ...

  inv-Objektový_datový_typ $\triangle$ *výraz*

  Constructor *Jméno_konstruktoru*(*parametry*)
  ⋮

End *Objektový_typ*

Obr. 10: Schéma deklarace potomka objektového datového typu

metod, které s těmito složkami budou pracovat, a schopnost dědit vlastnosti a chování od předka objektového datového typu.

Na obrázku č. 9 se nachází schéma deklarace objektového datového typu, který splňuje první požadavek na tento typ a to je zapouzdřenost. Nyní se pozastavme u popisu tohoto schématu.

Klíčové slovo **Specification** deklaruje, že nyní bude následovat specifikace objektového datového typu. V okamžiku zápisu tohoto klíčového slova je jako první datový člen datového typu deklarována proměnná *self*. Tato proměnná slouží k jednoznačné identifikaci instance datového typu, podobně jako je tomu u klasických objektově orientovaných jazyků.

Formální přechod od proměnné *self* k instanci objektového datového typu je prováděn pomocí mapování, které má následující schéma:

Objektový_typ ::
  self: **N**
  deklarace dalších složek typu
Objektové_mapování: $\mathbf{N} \xrightarrow{m}$ *Objektový_typ*

kde *Objektové_mapování* mapuje hodnotu proměnné *self* na konkrétní instanci objektového typu.

Následuje sekce deklarace proměnných, které budou sloužit podobně jako složky strukturovaného datového typu. Hned po deklaraci těchto proměnných následuje volitelná deklarace invariantu, který zabezpečuje platnost dat uložených v proměnné tohoto objektového typu.

Dále následuje deklarace jednoho nebo více operací konstruktoru, které jsou uvozeny klíčovým slovem **Constructor**. Samotné tělo konstruktoru má strukturu, která odpovídá struktuře operace tak, jak byla popsána v kapitole 3.1.2. Rozdíl vzhledem k operaci je v tom, že návratovou hodnotou, kterou není třeba

ADV *Jméno* **For Type** *Prezentovaný_typ*

declaration *proměnná*$_1$: *Typ*$_1$
...

inv-Jméno $\triangleq$ *výraz*

Constructor *Jméno_konstruktoru*(*parametry*)
⋮

End *Jméno*

Obr. 11: Schéma deklarace abstraktního datového pohledu

v deklaraci uvádět, je vždy vznikající instance tohoto objektového datového typu, podobně jako u konstrukčních funkcí strukturovaného datového typu.

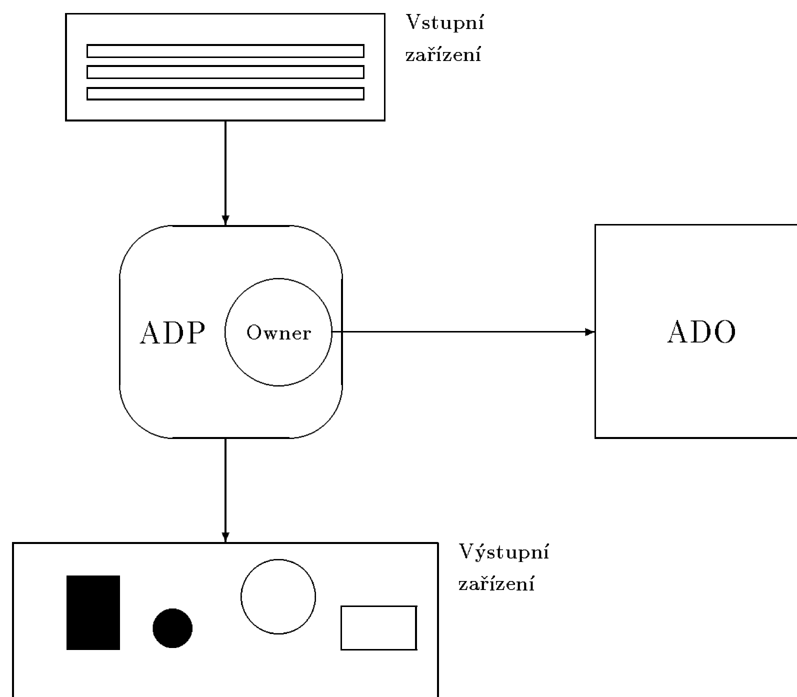
Destruktor je operace, která je automatiky volána při zániku instance objektového datového typu a nemůže být volána nikdy přímo. Operaci destruktoru je také možné vynechat. V tomto případě se použije pro rušení instancí klasického postupu strukturovaného datového typu.

Třetím novým klíčovým slovem specifikace je slovo **Operation**, kterým se uvozují metody objektového typu. Metody se liší od klasických operací specifikace VDM je v jediném bodě. Ve všech metodách, konstruktorech a destruktoru je navíc přístupný skrytý parametr metody a to datový člen *self*. Tento datový člen je pak používán pro přímý přístup k složkám objektového typu. Z tohoto důvodu nemusí být v sekcích ext, pre a post jednotlivých operací přímo uváděn přístup ke konkrétní externí proměnné objektového typu a naopak. Proto se metody objektového datového typu uvozují speciálními klíčovými slovy.

Druhou požadovanou vlastností objektových typů je dědičnost. Tato vlastnost může být řešena způsobem, který je uveden na obrázku č. 10. Zde se hned za hlavičkou specifikace nachází sekce, která určuje, od jakých objektových datových typů nově vytvářený typ dědí. Vlastní zdědění datových členů a operací by bylo možné vyřešit pomocí mechanického přepsaní specifikace těchto datových členů a operací z popisu typu předka do popisu typu potomka. Vzhledem k tomu, že se jedná o rutinní záležitost, byla nakonec zvolena jednodušší možnost a to použití konstrukce **Subtype of**, která vše výše uvedené vyřeší.

3.2.2 Specifikace abstraktních datových pohledů

V případě abstraktních datových pohledů by bylo zbytečné neustále používat specifikace objektového datového typu a v ní určovat jako třídu předka typ abstraktního datového pohledu. Z tohoto důvodu je zavedena upravená specifikace, která je určena přímo pro popis ADP. Ukázka použití této specifikace se nachází



Obr. 12: Mapování ADP na ADO pomocí proměnné *Owner*

na obrázku č. 11.

V této upravené specifikaci je konstrukce **Subtype of** nahrazena konstrukcí **ADV**. Tento upravený zápis je ekvivalentní zápisu

Specification *Jméno*
  Subtype of *ADV*
  ⋮

Výhodou je snadnější porozumění významu deklarace.

Novinkou vzhledem k objektovému datovému typu je další konstrukce a to **For Type**. Jak už bylo popsáno v kapitole 2.1, abstraktní datový pohled je určen převážně pro prezentování stavu nějakého abstraktního datového objektu. Ovšem při implementaci ADP je třeba mít k tomuto datovému objektu přístup a ten je zprostředkován právě pomocí syntaktické konstrukce **For Type**.

Uvedením této konstrukce v deklaraci abstraktního datového pohledu dojde k vytvoření nové proměnné, složky objektového datového typu, jejíž název je *Owner* a která je typu *Prezentovaný_typ*. Pomocí této proměnné je pak zprostředkován přístup k ADO, který má být prezentován, i uvnitř operací, které vlastní prezentaci vykonávají. Samozřejmě existuji i tzv. dekorační pohledy, jejichž prezentace nezávisí na stavu ADO. V tomto případě může být konstrukce

ADV *Jméno* **For Type** *Prezentovaný_typ*

declaration $proměnná_1$: Typ_1
$proměnná_2$, $proměnná_3$: Typ_2
...

inv-Jméno $\triangle$ *výraz*

Constructor *Jméno_konstruktoru*(*parametry*)
⋮

Event *Present*()
ext *externí proměnné*
post *výstupní predikát*

Event *MouseMove*(*point*: *TADVPoint*)
⋮

End *Jméno*

Obr. 13: Schéma deklarace operací pro obsluhu událostí

For Type vynechána. Ukázka toho, jaký význam má proměnná *Owner*, se nachází na obrázku č. 12.

3.2.3 Zachytávání a obsluha událostí systému

Nedílnou součást funkčnosti abstraktních datových pohledů tvoří zachytávání a obsluha událostí, které nastávají během práce systému. Z tohoto důvodu existuje i syntaktická konstrukce, která dokáže specifikovat operace, které mají být provedeny jako reakce na nějakou událost.

Na obrázku č. 13 se nachází schéma zápisu operací, které budou volány jako odpověď na událost. Syntaktický zápis těchto operací je velmi jednoduchý; skládá se z klíčového slova **Event** následovaného jménem operace s parametry, které jsou získány při akceptování zprávy o této události. Operace pro obsluhu událostí zpravidla nemají žádný výstupní parametr a také použití sekce pre se vstupním predikátem je řídké, protože na událost by neměly být kladeny žádné předběžné požadavky co se týče obsahu parametrů.

V systému existují již předdefinované typy událostí, jako je například událost **Event** *Change*() nebo **Event** *Present*(), které sdělují cílovému ADP, že je třeba provést novou prezentaci nebo naopak akceptovat vstup a změnit stav asociovaného ADO. Samozřejmě existuje ještě mnoho dalších typů událostí, jako například události myši, klávesnice, operačního systému, vzájemné koordinace pohledů a

podobně. Podrobnosti lze nalézt v popisu implementace systému.

3.3 Příklad specifikace jednoduchého ADP

V této kapitole si na jednoduchém příkladu ukážeme použití objektově orientované modifikace notace VDM v praxi. Tento řešený příklad by měl ozřejmit případné nejasnosti a měl by také ukázat relativní jednoduchost použití notace.

V reálném světě se často vyskytují požadavky na grafické zobrazení hodnot proměnných. Jedním z typických grafických výstupů je zobrazení hodnoty jako nějakého ukazatele na stupnici. Na tomto místě se pokusím nastínit řešení právě tohoto případu.

Nechť tedy existuje nějaká externí proměnná například celočíselného typu. Úkolem je specifikovat abstraktní datový pohled, který by dokázal stav této proměnné zobrazovat na stupnici. Náčrt možné specifikace tohoto ADP se nachází na obrázku č. 14.

V sekci deklarací lokálních proměnných objektového datového typu *Gauge* se nachází několik proměnných. Proměnné *Min* a *Max* obsahují minimální resp. maximální možnou hodnotu, která může být na stupnici zobrazena. Proměnná *Step* určuje velikost jednoho kroku posunutí jezdce na stupnici. Zbývající proměnné mají jen pomocnou funkci. Proměnná *Pressed* informuje ADP o tom, zda je stisknuté tlačítko myši a tak dochází při pohybu myší i k pohybu jezdce na stupnici. Proměnná *OldPos* pak udržuje poslední použitou pozici jezdce.

Následující část zápisu obsahuje specifikaci metod objektového typu. Nejprve se zde vyskytuje deklarace invariantu, který říká, že ADP může zobrazovat jen hodnoty v rozsahu *Min–Max* a aby to bylo možné, musí platit $Min < Max$.

Dále následuje deklarace konstruktoru, který inicializuje lokální proměnné. V sekci **pre** se kontrolují vstupní argumenty, v sekci **post** pak probíhá vlastní inicializace.

Operace *SetupDraw*() zabezpečuje přípravu datových členů na zobrazování. Jde o to, že v případě neplatného rozsahu proměnné *Owner* se stav této proměnné patřičně upraví.

V operaci *Draw*(), která zabezpečuje vlastní vykreslení stavu proměnné *Owner*, se nachází jen slovní popis toho, jakým způsobem se má daná hodnota prezentovat. Specifikace totiž nemusí za všech okolností vyjadřovat chování jen pomocí matematické notace. V případě grafického výstupu by to bylo celkem obtížné a navíc by bylo třeba znát specifikace operací, které tento výstup dokáží realizovat.

Následuje specifikace metod pro obsluhu událostí od myši. První metoda, *LButtonDown*(), je volána pro obsluhu stisku levého tlačítka myši v okamžiku, kdy se myš nachází nad tímto abstraktním datovým pohledem. Na základě polohy myši uvnitř ADP se rozhodne, zda dojde k zvýšení nebo snížení hodnoty proměnné *Owner* o hodnotu *Step*, nebo se tato událost zaznamená jako uchopení jezdce

types
  $Point$ ::
    x, y: $\mathbf{Z}$

ADV *Gauge* **For Type Z**

declaration Min, Max: $\mathbf{Z}$
  $OldPos$: $\mathbf{Z}$
  $Step$: $\mathbf{N}$
  $Pressed$: $\mathbf{B}$

$inv\text{-}Gauge \triangleq Min < Max$

Constructor $CreateGauge(min, max, step\colon \mathbf{Z})$
pre $min < max \wedge step < max - min$
post $Min = min \wedge Max = max \wedge Step = step \wedge Pressed = \mathsf{false}$

Operation $SetupDraw()$
post $Owner = Min \wedge \overleftarrow{Owner} < Min \vee Owner = Max \wedge \overleftarrow{Owner} > Max$

Event $Draw()$
post „Vykresli stupnici s jezdcem umístěným na pozici,
  která odpovídá hodnotě *Owner*"

Event $LButtonDown(p\colon Point)$
post $(OnLeftButton(p) \wedge Owner = \overleftarrow{Owner} - Step \vee$
  $OnRightButton(p) \wedge Owner = \overleftarrow{Owner} + Step) \wedge$
  $Pressed = OnRider(p) \wedge OldPos = p.x$

Event $LButtonUp(p\colon Point)$
post $Pressed = \mathsf{false}$

Event $MouseMove(p\colon Point)$
post $Pressed \wedge Owner = \overleftarrow{Owner} + Step * (\overleftarrow{OldPos} - p.x) \wedge OldPos = p.x$

End *Gauge*

Obr. 14: Specifikace ADP zobrazujícího celé číslo na stupnici

na stupnici. Ve specifikaci jsou použity funkce *OnLeftButton*(), *OnRightButton*() a *OnRider*(), které určují, kde uvnitř ADP se právě myš nachází.

Zbývající dvě metody pro obsluhu zpráv jsou velmi jednoduché. Metoda *LButtonUp*() jen odznačí příznak stisku myši na jezdci a metoda *MouseMove*() zabezpečí v případě pohybu jezdcem přenesení tohoto pohybu na změnu stavu proměnné *Owner*.

Zápis algoritmů pomocí specifikace VDM není všelékem a jak už bylo řečeno, existuje množství oblastí návrhu aplikací, kde použití VDM nepřinese oproti klasickým metodám žádné výhody. To je případ především aplikací s minimem algoritmů, ale s velkým množstvím rutinních rozsáhlých transformací. Avšak v případě, kdy je třeba výstižně popsat chování jádra aplikace nebo jeho části, přináší použití VDM velké výhody.

4 Principy činnosti knihovny ADP

Tato kapitola je věnována popisu postupů při implementaci abstraktních datových pohledů. Jsou zde popsány základní principy implementace i použité mechanismy a algoritmy. Pro vlastní implementaci datových struktur bylo použito jazyka ANSI C++ a zdrojové kódy produktu byly odladěny v prostředí Borland C++ 4.5.

4.1 Stanovení cílů

V případě návrhu nějakého rozsáhlejšího programového systému je třeba vždy nejprve stanovit cíle. Jen tak je možné zaručit, že výsledkem práce bude to, co je požadováno. Východiskem pro tento programový systém jsou požadavky na abstraktní datový pohled tak, jak jsou uvedeny v kapitolách 2.1.1 a 2.1.3. Tyto požadavky byly ještě dále upraveny a rozšířeny o některé nové rysy.

Nejprve tedy ve stručnosti shrnutí požadavků tak, jak byly definovány ve výše uvedených kapitolách. Zcela základní požadavek na abstraktní datový pohled představuje oddělení uživatelského rozhraní od datových objektů v jádru aplikace. Jak už je známo, datové pohledy se nachází ve vrstvě uživatelského rozhraní a odsud má každý datový pohled přístup ke svému asociovanému datovému objektu. Dalším neméně důležitým požadavkem na ADP je schopnost ADP vystupovat jako subpohled při tvorbě větších komponovaných pohledů. Jak už bylo také řečeno, model ADP neklade žádné bližší požadavky na tvar a způsob zobrazování asociovaného datového objektu. Proto by ani vytvářený systém neměl budoucí uživatele v tomto směru omezovat. Poslední větší požadavek na ADP představuje schopnost ADP zajišťovat prezentaci stavu ADO nejen v nějakém okně, ale také, v případě potřeby, směrem k jinému ADP a to i ve vzdáleném výpočetním systému.

Požadavky uvedené v předchozím odstavci představují jistý základ, na kterém lze začít stavět programový systém. Ovšem přitom je třeba některé požadavky zpřesnit a upravit, případně ještě nové požadavky nebo omezení doplnit. Tyto změny budou předmětem následujících odstavců.

Prvním cílem, který jsem si vytyčil na počátku návrhu systému abstraktních datových pohledů, bylo vytvoření takového systému, který by byl maximálně přenositelný a znovupoužitelný. Tohoto cíle jsem chtěl dosáhnout využíváním maximální míry abstrakce při návrhu tříd a naprostým oddělením systému abstraktních datových pohledů od operačního systému. Mimoto musela být respektována zásada, že abstraktní datové objekty, které mohou být asociovány a následně prezentovány pomocí ADP, mohou být zcela libovolného typu a není je třeba žádným způsobem upravovat pro spolupráci s ADP. Důsledkem tohoto požadavku je, že ADP nesmí předpokládat nic bližšího o typu, chování a operacích ADO.

Dalším cílem bylo navrhnou systém tak, aby byl snadno rozšiřitelný o nové datové pohledy. Důležité v tomto případě je, aby návrh tříd byl přehledný a me-

tody tříd měly přesně určenou jednoznačnou úlohu. Přitom je nutné, aby každá metoda vykonávala jen jednu určitou činnost, aby více metod nevykonávalo stejnou činnost, ale na druhé straně, aby metod, které je třeba při návrhu nové třídy abstraktního datového pohledu modifikovat bylo co nejméně.

Významnou roli při vytváření tohoto systému hrál také požadavek, aby do hierarchie tříd byly zahrnuty také typy datových pohledů, které jsou schopné vzájemné komunikace a výměny informací například v síťovém prostředí. Přitom by ovšem rozdíl mezi graficky orientovanými pohledy a síťově orientovanými pohledy neměl být z hlediska uživatele systému vůbec podstatný.

Výsledným produktem by měla být knihovna tříd abstraktních datových pohledů, kterou bude možno distribuovat ne pouze s jednou aplikací, ale se všemi aplikacemi, které používají paradigma ADP, bez rozdílu na typ používaného operačního systému, síťového systému nebo hardware. Cíl je to sice ambiciózní, ale s vynaložením dostatečného úsilí splnitelný.

4.2 Hierarchie tříd

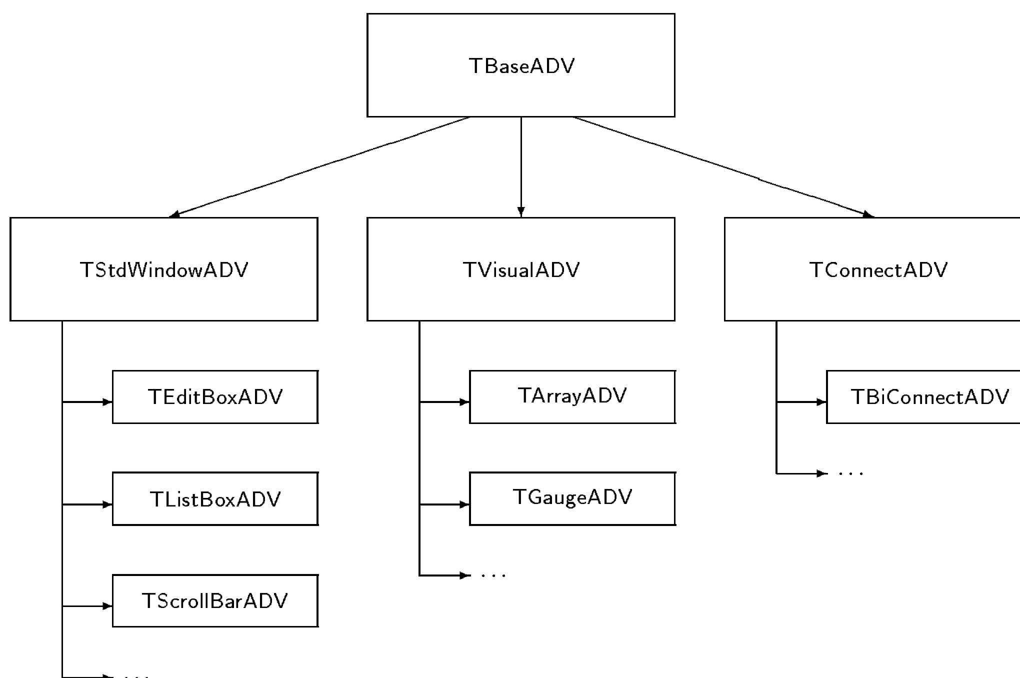
Po stanovení požadavků na implementaci systému nastává nejobtížnější fáze návrhu a to je stanovení hierarchie tříd, která bude dané požadavky splňovat. Přitom snad ještě důležitější úlohu než vlastní určení vztahů mezi třídami hraje návrh metod těchto tříd. V této kapitole se ale prvotně zaměřím na popis navržené hierarchie.

Základní část stromu objektové hierarchie je možné si prohlédnou na obrázku č. 15. Bázovou třídou celé hierarchie abstraktních datových pohledů je třída TBaseADV. Tato třída zapouzdřuje všechny základní prvky chování ADP. Zejména se jedná o následující:

- schopnost skládání ADP do větších celků
- schopnost prezentace vzhledem k okolí
- schopnost přijímat podněty z okolí
- schopnost zachytávat a obsluhovat události
- schopnost přenosu dat do dětských ADP
- schopnost přístupu k asociovanému abstraktnímu datovému objektu

V této třídě je také definováno mnoho virtuálních metod, které budou v dětských třídách předefinovány. Na úrovni třídy TBaseADV se ještě nerozlišuje mezi grafickými a síťovými datovými pohledy. K tomuto rozlišení dojde až později.

Prvním potomkem třídy TBaseADV je třída TVisualADV. Tato třída představuje hlavní směr implementace ADP. Slouží jako předek, ve kterém se implementují všechny vlastnosti datových pohledů společné grafickým pohledům. Mezi tyto vlastnosti patří také:



Obr. 15: Hierarchie tříd abstraktních datových pohledů

- schopnost prezentace v okně nezávisle na operačním systému
- schopnost ošetření speciálních akcí uživatele, jako je například práce s myší a klávesnicí
- schopnost spolupráce s dětskými grafickými pohledy

Už při použití této třídy se projevují první výhody, které plynou z vhodného návrhu rodičovské třídy. Díky vhodně navrženému rozhraní je třeba ve třídě TVisualADV překrýt jen minimum metod rodičovské třídy.

Dalším potomkem třídy TBaseADV je třída TConnectADV, která představuje bázovou třídu pro všechny abstraktní datové pohledy, které neprezentují svůj stav v okně, ale v síti. I tato třída s výhodou využívá množství metod rodičovské třídy a přidává jen několik nových, které zajišťují mechanismus síťové komunikace. Pod pojmem síťová komunikace si však lze představit téměř cokoliv. Zdaleka se nemusí jednat jen o síťovou komunikaci v pravém slova smyslu. Může také jít například o meziprocesovou komunikaci mezi dvěma aplikacemi nebo thready na jediném lokálním počítači, ba co víc, dokonce se může jednat i o přímou komunikaci mezi objekty toho samého procesu. Vše záleží na tom, jak bude systém ADP implementován a implementace se na této úrovni ještě neřeší.

Třetím a zatím posledním potomkem třídy TBaseADV je třída TStdWindowADV. Tato třída má sloužit jako bázová třída pro jisté grafické abstraktní

datové pohledy, které využívají prvků grafického rozhraní částečně závislého na operačním systému. Důvodem vzniku této větve objektové hierarchie je snaha zjednodušit použití operačním systémem definovaných tříd grafických objektů tak, aby uživatel nemusel pro tyto již existující prvky psát nové datové pohledy. Vzhledem k tomu, že se jedná o částečnou závislost ADP na operačním systému, bylo třeba mírně omezit některé požadavky kladené na abstraktní datové pohledy. Například ADP tohoto typu mohou být pouze obdélníkového tvaru a nemohou zpracovávat větší část vstupních událostí.

Každá z těchto tří větví objektové hierarchie má nebo může mít ještě množství potomků, které zde ale nebudu uvádět. V případě potřeby lze podrobnosti nalézt v referenční příručce k programovému systému. Přesto se na tomto místě zmíním ještě o jedné velmi důležité třídě pro chod systému.

Tímto důležitým prvkem systému je třída TADVFactory. Tato třída představuje základní kámen nezávislosti programového systému na operačním systému. Samotná třída TADVFactory má velmi jednoduchou strukturu s minimem implementovaných funkcí. Většina funkcí je totiž čistě virtuálních. Instance jistého potomka této třídy slouží jako továrna na auta (objekty uživatelského rozhraní), ve které na základě objednávky (od ADP) vyrobí automobil (objekt) požadovaného typu, avšak s řízením přizpůsobeným cílové zemi prodeje (operačnímu systému). To, pro jaký operační systém budou objekty vyráběny, se určí až během fáze linkování aplikace.

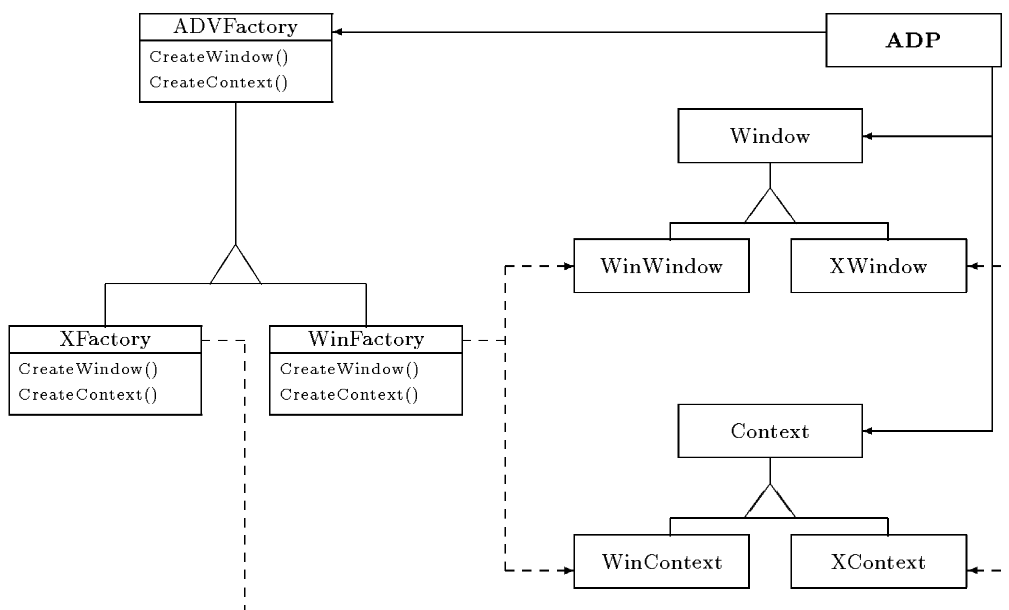
4.3 Principy spolupráce mezi třídami

V předchozí kapitole byly charakterizovány základní třídy systému pro podporu abstraktních datových pohledů. Cílem této kapitoly je charakterizovat principy činnosti systému a popsat vzájemné vazby mezi výše uvedenými třídami. Při vysvětlování vztahů mezi některými třídami bude často používáno návrhových vzorů, jak jsou popsány v [7].

Jak už bylo dříve řečeno, základem celé objektové hierarchie je třída TBaseADV. V této třídě se nachází několik důležitých datových členů. Je zřejmé, že každá instance abstraktního datového pohledu musí mít přiřazen nějaký jednoznačný identifikátor. Tento identifikátor se používá především pro určení cílového ADP při zasílání zpráv. Dále se ho používá při vytváření složených pohledů. Datový člen, který identifikátor pohledu obsahuje, se jmenuje *AdvId*. Podobně datový člen *ParentId* obsahuje identifikátor rodičovského pohledu nebo okna, ve kterém má instance pracovat.

Jak je tedy z popisu zřejmé, abstraktní datové pohledy nepoužívají pro vzájemnou identifikaci klasických ukazatelů, ale místo toho používají jednoznačných identifikátorů. Z toho plynou následující výhody:

- Odstranění typové kontroly při komunikaci mezi pohledy.



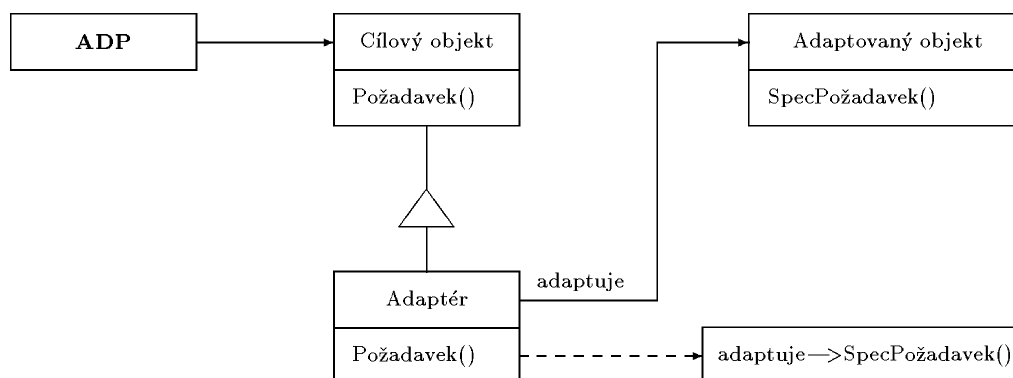
Obr. 16: Schéma činnosti návrhového vzoru abstraktní továrna

- Možnost propojení dětského ADP na rodičovský i v případě, kdy rodičovský ADP ještě fyzicky neexistuje.
- Možnost jednoznačné identifikace a komunikace mezi ADP po síti.
- Transparentní začlenění abstraktních datových pohledů do prostředí konkrétního operačního systému.

Potomkem třídy TBaseADV je třída TVisualADV. Návrhu a implementaci této třídy bylo věnováno největší úsilí. Zřejmě je třeba, aby instance této třídy měly schopnost prezentovat se v okně. V tomto okamžiku ale nastávají problémy s tím, jak zabezpečit nezávislost této třídy na okolním prostředí. Nakonec bylo po důkladné analýze zvoleno řešení založené na upraveném návrhovém vzoru abstraktní továrna (Abstract Factory). Schéma činnosti tohoto vzoru se nachází na obrázku č. 16, podrobný popis pak lze nalézt v [7], str. 87–96.

Jak je z obrázku patrné, tento vzor se skládá z několika výkonných částí. Základ tvoří třída abstraktní továrny TADVFactory. Tato třída definuje rozhraní, které bude používáno pro získávání objektů požadovaných jednotlivými instancemi abstraktních datových pohledů. Potomkové této třídy pak implementují tyto metody tak, aby výsledná hodnota byla instance typu, který je již svázán s konkrétním prostředím operačního systému.

Návratové hodnoty vytvářecích funkcí jsou ukazatele typované na nějaké abstraktní třídy. Implementace továrny v každém operačním systému vždy vytvoří potomky těchto tříd, které už budou používat implementačně závislé datové



Obr. 17: Schéma činnosti návrhového vzoru adaptér

struktury a funkce. Použití těchto funkcí však bude uživateli skryto právě za abstraktním rozhraním bázových tříd.

Při implementaci abstraktních datových pohledů došlo ještě k dalšímu rozšíření návrhového vzoru abstraktní továrna tak, že tato továrna poskytuje pohledům nejen objekty, které dokáží spolupracovat s operačním systémem, ale navíc poskytuje také abstraktní přístup ke službám tohoto operačního systému, jako je například posílání zpráv. Díky tomu je možné používat abstraktních metod pro komunikaci mezi ADP a až konkrétní implementace abstraktní továrny rozhodne, jakým způsobem se bude vlastní zasílání zpráv provádět. Například může tato továrna implementovat síťovou komunikaci pomocí protokolu TCP/IP nebo naopak může síťovou komunikaci úplně potlačit.

Při vytváření konkrétních implementací návrhového vzoru je často třeba přizpůsobovat rozhraní již existujících objektů operačního systému rozhraní, které je definováno pro bázovou třídu továrny. V tomto případě je s výhodou použito návrhového vzoru adaptér. Podrobný popis tohoto vzoru je opět možné nalézt v [7], str. 139–150.

Návrhový vzor adaptér (Adapter), jehož schéma implementace se nachází na obrázku č. 17, je založen na relativně jednoduchém principu. Existuje třída cílového objektu a třída adaptovaného objektu. Tuto třídu je třeba přetransformovat na typ třídy, který je požadován. Lze to provést několika způsoby, z nichž nejjednodušší je, že se vytvoří nová třída, která je potomkem třídy cílového objektu, v této třídě bude deklarován nový datový člen typu ukazatel na instanci adaptované třídy a překryté virtuální metody nové třídy budou postupně volat přes tento datový člen metody původní adaptované třídy. Tak je zabezpečeno, že vzhledem k abstraktnímu datovému pohledu se chová objekt operačního systému tak, jak se očekává.

Při návrhu systému, který by byl zcela nezávislý na okolním prostředí bylo použito i několika dalších návrhových vzorů, jako například stavitel (Build-

er), tovární metoda (Factory Method), kompozice (Composite), řetězec odpovědí (Chain of Responsibility) a některé další. Vzhledem k tomu, že tyto návrhové vzory nehrají tak významnou roli pro pochopení činnosti systému ADP, nebudou zde popisovány. V případě potřeby lze podrobnosti nalézt opět v [7].

Co se týče třídy TConnectADV, nejsou při její implementaci použity žádné zvláštní postupy. Jediný datový člen, který je v ní doplněn, slouží jako identifikátor abstraktního datového pohledu, se kterým má být navázáno spojení v případě síťové komunikace. Pokud se jedná o jednosměrnou síťovou komunikaci, není tento datový člen v instanci příjemce vůbec používán.

Podobně třída TStdWindowADV nemá příliš mnoho nových funkcí vzhledem k třídě předka. Základ funkčnosti této třídy tvoří datový člen, který odkazuje na abstraktní prvek systému, který je získán od továrny tak, jak bylo popsáno o několik odstavců výše.

Poslední třídou, o které bych se chtěl na tomto místě zmínit, je třída TMessage, která představuje univerzální způsob přenosu zpráv mezi jednotlivými abstraktními datovými pohledy. Datové členy této třídy nesou informaci o tom, komu je zpráva určena, od koho pochází, jaké má příznaky a pak se zde nachází také datová oblast libovolné délky, ve které jsou uchovány dodatečné informace pro daný typ zprávy.

4.4 Základní metody tříd ADP

V předchozí kapitole byly popsány vztahy mezi jednotlivými třídami systému ADP. Tato kapitola se naopak zaměřuje na popis metod tříd, které implementují základní funkčnost systému.

Opět je třeba začít s popisem třídy TBaseADV. V této třídě je implementováno funkční jádro systému. Zde se však zmíním jen o některých metodách.

Patrně nejdůležitější roli pro funkčnost modelu hrají metody *PresentState*() a *ChangeState*(). Uvnitř první z uvedených metod se odehrává prezentace stavu abstraktního datového objektu vzhledem k okolnímu světu. Na úrovni bázové třídy ještě není vůbec nic známo o tom, jakým způsobem bude docházet k vlastní prezentaci stavu. Může se jednat jak o grafickou prezentaci v okně, tak o síťovou prezentaci. Konkrétní způsob prezentace je určen až v překrývajících metodách tříd TVisualADV a TConnectADV. Druhá metoda slouží naopak pro příjem informací, které mají vliv na změnu stavu ADO. Také tato metoda není na úrovni třídy TBaseADV ještě implementována a její implementace se ponechává na třídách potomků.

Další důležitou součást třídy TBaseADV představuje metoda *SetupADV*(). Jedná se o virtuální metodu, která je volána jako první metoda okamžitě po tom, co je vytvořeno fyzické rozhraní ADP směrem k operačnímu systému. Obdobnou roli hraje metoda *CleanupADV*(), která je naopak volána jako poslední těsně před zánikem fyzické reprezentace ADP.

Ve třídě TBaseADV existuje ještě několik desítek dalších metod. Tyto metody však mají převážně pomocnou a doplňující funkci a nehrají podstatnější roli ve vlastním modelu ADP. Jejich popis lze nalézt v příloze.

Nyní se ještě krátce zmíním o metodách některých dalších tříd. Nejprve se zastavme u třídy TVisualADV. Tato třída zásadním způsobem upravuje chování metody *PresentState*() a to tak, že zabezpečuje grafickou prezentaci stavu ADO uvnitř nějakého okna. Z tohoto důvodu byla v této třídě deklarována nová virtuální metoda se jménem *Draw*(), která jako jediná určuje způsob vykreslení stavu datového objektu uvnitř kreslicí oblasti abstraktního datového pohledu. V ideálním případě je třeba při návrhu nového typu ADP překrýt ve třídě potomka jen tuto metodu. S touto metodou má také úzkou souvislost jiná metoda a to *SetupDraw*(). Tato metoda je volána ještě před vlastním započetím prezentace stavu v okně a tak je možné změnit některé atributy vykreslovaného pohledu, jako je jeho tvar či velikost, ještě před tím, než dojde k vlastnímu vykreslení pohledu.

Také třída TConnectADV překrývá metody *PresentState*() a *ChangeState*() a to takovým způsobem, aby bylo možné jednoduše používat ADP tohoto typu pro datovou prezentaci po síti. Z tohoto důvodu jsou v této třídě deklarovány dvě nové virtuální metody, které jsou volány uvnitř *PresentState*() a *ChangeState*(). Tyto metody se nazývají *InputState*() a *OutputState*() a slouží jako transformátory pro převod stavu ADO z nebo do protokolu, který je používán pro síťovou prezentaci.

Konečně ve třídě TStdWindowADV jsou deklarovány metody *ReadWindowState*() a *WriteWindowState*(), které slouží k podobným účelům, jako *InputState*() a *OutputState*() jen s tím rozdílem, že se nepracuje s nějakým protokolem, ale s abstraktními objekty rozhraní operačního systému.

4.5 Implementace jednoduchého komunikačního ADP

Uvažme následující úlohu. Na jistém počítači existuje trenažer pro řízení automobilu. Na tomto trenažeru je možné ovládat prvky řízení automobilu včetně řazení rychlostních stupňů a přidávání či ubírání plynu. Stav těchto prvků řízení je uložen v nějaké datové struktuře, například TAuto. Na jiném místě existuje řídící pult, na kterém vyučující kontroluje, zda žák na trenažeru nepřekročí maximální povolenou rychlost.

Jedním z možných způsobů řešení tohoto příkladu je použití modelu ADP pro komunikaci mezi trenažerem a řídícím pultem. Na obrázku č. 18 se nachází hlavičkový soubor, který obsahuje deklaraci nové třídy abstraktního datového pohledu, která bude sloužit pro transformaci stavu řízení automobilu na rychlost. Vzhledem k tomu, že se jedná o použití ADP pro síťovou komunikaci, je nová třída TPrevodovkaADV potomkem třídy TConnectADV. Protože se jedná o jednosměrnou komunikaci od trenažeru směrem k řídícímu pultu, bude překryta jen metoda *OutputState*(), ve které bude implementována požadovaná datová transformace. Zbývající dvě metody třídy mají jen pomocnou funkci.

```
struct TAuto
{ int Plyn;        // 0 - 100
  int Stupen;      // 0 - 5
};

class TPrevodovkaADV : public TConnectADV
{ protected:
    TAuto &Data;
    virtual BOOL ContainsAddress(void *address);

  public:
    TPrevodovkaADV(const TId &parent, const TId &map, TAuto *owner);

    virtual void  OutputState(void *data);
    virtual int   StateSize() {return sizeof(int);}
};
```

Obr. 18: Deklarace třídy pro převod stavu trenažéru

Vlastní implementaci metod třídy TPrevodovkaADV je možné si prohlédnout na obrázku č. 19. Činnost metod je tak jednoduchá, že snad nepotřebuje žádných dalších komentářů. Přesto se pozastavím u metody *ContainsAddress*(). Tato metoda slouží k rozpoznání toho, které adresy v paměti jsou obsazeny datovým objektem, který je tímto ADP prezentován. Na základě volání této metody je už třída TBaseADV schopná rozpoznat, že došlo k modifikaci stavu právě tohoto asociovaného ADO a je třeba tedy provést novou prezentaci ADO vzhledem k okolí.

```
BOOL TPrevodovkaADV::ContainsAddress(void *address)
{ return address == Owner || (int *)address == &Data.Plyn ||
                             (int *)address == &Data.Stupen;
}


TPrevodovkaADV::TPrevodovkaADV(const TId &parent, const TId &map,
                               TAuto *owner)
  : TConnectADV(parent, map, owner), Data(*owner)
{
}


void TPrevodovkaADV::OutputState(void *data)
{ int rychlost;
  if (data != NULL)
  { switch (Data.Stupen)
    { case 0: rychlost = 0;
              break;
      case 1: rychlost = (Data.Plyn * 3) / 10;
              break;
      case 2: rychlost = (Data.Plyn * 4) / 10 + 10;
              break;
      case 3: rychlost = (Data.Plyn * 6) / 10 + 20;
              break;
      case 4: rychlost = (Data.Plyn * 8) / 10 + 40;
              break;
      case 5: rychlost = (Data.Plyn * 11) / 10 + 60;
              break;
    }
    *(int *)data = rychlost;
  }
}
```

Obr. 19: Implementace metod třídy pro převod stavu trenažeru

5 Postup při programování metodou ADP

Při použití postupů popsaných v předchozích kapitolách v praxi může vyvstat několik otázek, které zatím ještě nebyly zodpovězeny. Jedná se například o otázku, jakým způsobem transformovat specifikaci abstraktních datových pohledů na konkrétní datové struktury, nebo jakým způsobem by mělo být postupováno při vlastním kódování aplikace, která konceptu ADP využívá. Vysvětlení těchto otázek jsou věnovány následující kapitoly.

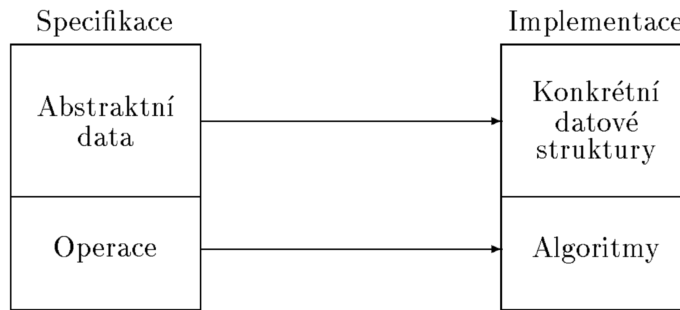
5.1 Transformace specifikace VDM do jazyka C++

Specifikace VDM ani její modifikace nejsou v žádném případě obdobou programovacího jazyka. Není tedy ani možné, aby systém popsaný pomocí notace VDM byl přímo implementován na počítači nebo zcela mechanicky přenesen do nějakého programovacího jazyka. Z tohoto důvodu je třeba se zmínit o postupu, který vede k transformaci specifikovaného systému na konkrétní kód aplikace v některém z programovacích jazyků. Vzhledem k tomu, že pro implementaci knihovny ADP byl použit programovací jazyk ANSI C++, bude v této kapitole popsán přechod od specifikace VDM právě do tohoto jazyka.

Jak už bylo dříve v kapitole 3 řečeno, vlastní specifikace programového systému v notaci VDM se skládá z dvou hlavních částí. Je to popis abstraktních datových typů a popis operací, které pracují s množinou proměnných, takzvaným stavem aplikace, a popisují požadované transformace a chování systému. Každé části popisu systému v notaci VDM odpovídá jiná část konkrétního programového kódu aplikace. Části popisující abstraktní datové typy odpovídá v jazyce ANSI C++ část deklarací datových typů a datových struktur, naopak části specifikace popisující operace odpovídá kód funkcí a metod. Kód těchto metod jistým způsobem popisuje algoritmus, který má být vykonán, aby došlo k splnění požadavků specifikovaných v operacích. Schéma přechodu od specifikace VDM ke konkrétnímu programovacímu jazyku se nachází na obrázku č. 20.

Nejprve se soustředíme na popis transformace datových typů. Nejzákladnějšími datovými typy ve specifikaci jsou již předdefinované typy $\mathbf{N}$, $\mathbf{N}_0$, $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{B}$ a *Char*. K těmto datovým typům zpravidla existuje odpovídající skalární typ proměnných v jazyce C++. Například datové typy $\mathbf{N}$ a $\mathbf{N}_0$ mohou být reprezentovány v C++ pomocí typu **unsigned int**, typ $\mathbf{Q}$ může být reprezentován typem **double**.

Výše jmenované datové typy jsou speciálním případem obecného datového typu množina. Množinový typ specifikace VDM je omezen tak, že prvkem tohoto datového typu mohou být vždy jen konečné podmnožiny prvků. V závislosti na počtu a typu prvků množiny je pak třeba použít konkrétní reprezentace hodnot tohoto typu v programovacím jazyku. V případě, kdy množiny mohou obsahovat jen malý počet prvků, lze použít prezentace například pomocí typu **enum**. V případě rozsáhlejších množin je třeba použít pole bitů nebo nějaké dynamic-



Obr. 20: Přechod od VDM ke konkrétní implementaci

ké datové struktury. Vše závisí na konkrétních okolnostech a neexistuje předpis, který by dokázal mechanicky určit nejvhodnější datovou reprezentaci.

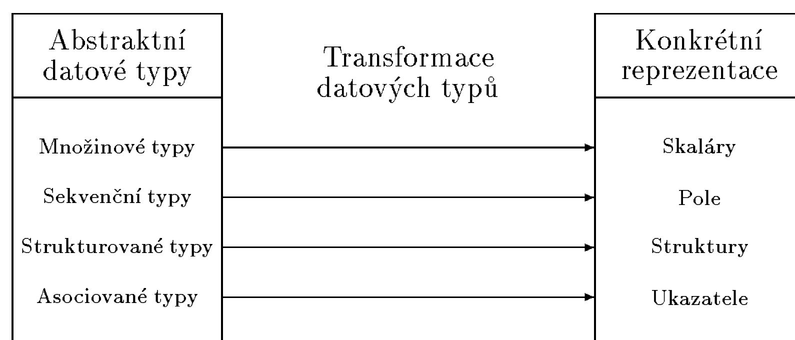
Dalším datovým typem specifikace VDM jsou sekvence. Z definice sekvence opět plyne konečnost prvků tohoto datového typu. Pro reprezentaci sekvence je možné použít několika konkrétních datových struktur. Velmi vhodné se jeví v případě sekvencí s pevnou maximální délkou použití polí jazyka C++. V případě sekvencí s blíže neurčeným rozsahem hodnot je pak možné použít dynamického seznamu.

Strukturovaný datový typ specifikace VDM přímo odpovídá typu **struct** programovacího jazyka C++ a tato reprezentace bude ve většině případů zcela postačující. Za jistých okolností však může být použito i jiných konkrétních datových reprezentací, zpravidla dynamických struktur.

Posledním standardním datovým typem specifikace je asociovaný typ. Tento typ slouží k vzájemnému provázání hodnot dvou datových typů. Například se může jednat o provázání strukturovaného datového typu, který představuje evidenční kartu zaměstnance s jiným strukturovaným typem, který představuje kartu s čerpáním dovolené. Už z tohoto příkladu vyplývá, že konkrétní datovou reprezentací může být například typ **struct**. Nevýhodou této konkrétní reprezentace je však statická deklarace typu a provázanost hodnot obou množin v poměru 1 : 1. Proto se jeví jako mnohem vhodnější použití dynamických datových struktur a ukazatelů. V tomto případě lze modelovat asociovaný typ bez omezení.

Na obrázku č. 21 se nachází schéma přechodu od abstraktních datových typů specifikace VDM ke konkrétním typům implementace v nějakém programovacím jazyce. Je zde ve zkratce naznačeno to, co již bylo o převodu typů řečeno v předchozích odstavcích.

Dále je třeba se zmínit o tom, jakým způsobem transformovat operace a funkce zapsané v notaci VDM na funkce programovacího jazyka. V případě explicitně deklarovaných funkcí se jedná o relativně jednoduchou záležitost. Jak je už známo, deklarace takových funkcí obsahují popis algoritmu, který by měl být konkrétními funkcemi programovacího jazyka vykonán. Nezbývá proto nic jiného,



Obr. 21: Implementace datových typů VDM

než tento algoritmus převést do syntaktických konstrukcí konkrétního programovacího jazyka, v našem případě jazyka C++.

V případě funkcí a operací deklarovaných implicitně je situace o něco složitější. V tomto případě totiž specifikace obsahuje matematický popis činnosti funkce a už zde není nic řečeno o tom, jakým způsobem by se mělo pokračovat ve vlastní implementaci. To je na jedné straně výhodné, protože tak je umožněn univerzální popis funkčnosti systému nezávislý na schopnostech programovacího jazyka, ale na druhé straně je třeba vynaložit větší úsilí při vlastním přechodu od specifikace k implementaci. Nezbývá totiž nic jiného, než pochopit význam matematického popisu a ten pak podle okolností vyjádřit konkrétním algoritmem v daném programovacím jazyce.

Specifikace VDM je v této diplomové práci rozšířena o objektově orientované rysy, které umožňují snadný popis abstraktních datových pohledů. Popišme si proto, jakým způsobem postupovat při transformaci tohoto abstraktního datového typu do konkrétních programových konstrukcí programovacího jazyka C++.

Už na první pohled je zřejmé, že samotná specifikace objektového datového typu je svou strukturou velmi podobná deklaraci třídy v jazyce C++. Deklarace **Specification** odpovídá programové konstrukci **class**, deklarace **Subtype of** odpovídá zavedení dědičnosti tříd a pro deklarace konstruktorů, destruktorů i operací včetně proměnné *self* existují v jazyce C++ také odpovídající syntaktické konstrukce. Jediným rozdílem je deklarace invariantu ve specifikaci objektového datového typu, která v C++ neexistuje. Ovšem v tomto případě se nejedná o nic jiného než o popis požadavků kladených na datové složky objektu, které musí být splněny, aby se objekt nacházel v korektním stavu. Vzhledem k tomu, že k datovým členům objektu se vždy přistupuje pomocí metod, není těžké zabudovat kontrolu dodržování tohoto invariantu do implementace metod třídy.

Speciálním případem specifikace objektového datového typu je specifikace abstraktního datového pohledu. Jak už bylo řečeno v kapitole 3.2.2, ve skutečnosti se nejedná o nic jiného, než o náhradu složitější konstrukce jednodušší syntaktickou zkratkou. Z tohoto hlediska se popis ADP od popisu objektového typu liší

jen v tom, že třída vytvářeného abstraktního datového pohledu má vždy pevně daného předka. Tímto předkem je pro grafický pohled třída TVisualADV a pro síťový komunikační pohled třída TConnectADV. Podrobnosti lze nalézt v následující kapitole a v přílohách.

5.2 Postup při kódování aplikace

Tato kapitola je zaměřena na popis postupu, jakým by měly být vytvářeny nové aplikace využívající abstraktních datových pohledů v jazyce C++. Jsou zde také popsány základní metody tříd, které je třeba překrýt, aby byla zaručena funkčnost aplikace. V každé třídě, která implementuje abstraktní datové pohledy se nachází několik desítek metod. Většinu z nich běžný uživatel nebude nikdy překrývat. Funkčnost těchto metod je popsána až v přílohách. Zde se zaměřím na popis jen těch nejdůležitějších metod, které je třeba vždy nebo alespoň ve většině případů překrýt.

5.2.1 Metody třídy TBaseADV

Nejprve se budu věnovat třídě TBaseADV. Této třídy se zpravidla nepoužívá pro přímé odvozování nových abstraktních datových pohledů. Slouží však jako rodičovská třída jiných tříd, které se už pro konstrukci nových pohledů používají. Nejdůležitější roli z hlediska uživatele hrají následující metody třídy:

virtual void	PresentState();
virtual BOOL	ChangeState(**void** *data);
virtual void	SetupChildOwners();
virtual BOOL	ContainsAddress(**void** *address);
virtual int	Transfer(**void** *buffer, TDirection direction);

Největší význam pro činnost abstraktního datového pohledu mají první dvě funkce. Funkce *PresentState*() je volána vždy, když je třeba prezentovat stav navázaného abstraktního datového objektu, funkce *ChangeState*() je volána vždy, když je třeba z vnějšku změnit stav asociovaného ADO. Překrytím těchto metod lze dosáhnout zcela nového způsobu prezentace. Příkladem může být například překrytí obou metod ve třídách TVisualADV a TConnectADV. Překryté metody zde jednou prezentují asociovaný ADO vizuálně v okně, podruhé komunikačně jako přenos transformovaných dat například po síti.

Metoda *SetupChildOwners*() je volána před vlastní prezentací datového pohledu a slouží k nastavení vlastníka dětských pohledů. Standardně tato metoda neprovádí žádné akce. Občas však může být nutné ji překrýt tak, aby dětské pohledy zobrazovaly data z nějaké konkrétní množiny vlastníků. Například abstraktní datový pohled pro zobrazování polí dokáže současně zobrazit jen omezený počet prvků velkého pole. Na základě změny indexu prvního prvku pole, který

má být zobrazován, nastaví tato metoda hodnoty proměnných *Owner* v dětských pohledech na ty prvky pole, které mají být právě zobrazovány.

Metoda *ContainsAddress*() se používá při automatickém zjišťování změny stavu abstraktních datových objektů. V případě, že nastane změna stavu asociovaného ADO, dojde automaticky k nové prezentaci ADP. Standardně tato metoda zjišťuje, zda adresa, která je zadána jako parametr metody, neodpovídá adrese asociovaného ADO. V případě složitých datových objektů je však možné, že dojde ke změně jen některé složky dat. V tomto případě je třeba metodu překrýt tak, aby byly rozpoznány změny i na jiných adresách, než na adrese celého objektu.

Poslední metodu *Transfer*() je třeba překrýt v případě, že nově vytvářený abstraktní datový pohled bude využívat mechanismu transferu dat do dětských datových pohledů.

5.2.2 Metody pro obsluhu událostí

Nedílnou součástí třídy TBaseADV a všech jejích potomků je schopnost obsluhovat přicházející zprávy. Z tohoto důvodu bylo definováno několik maker, která tuto činnost velmi usnadňují.

V případě, kdy se požaduje, aby abstraktní datový pohled obsluhoval nějakou událost, je třeba v deklaraci třídy uvést makro

DECLARE_HANDLER(*jméno_třídy*)

kde za jméno třídy se dosadí skutečné jméno nově vytvářené třídy. Dále je třeba někde ve zdrojovém textu deklarovat

IMPLEMENT_HANDLER(*jméno_třídy*)
EV_xxxx
⋮
END_HANDLER1(*jméno_třídy_předka*);

kde *jméno_třídy* se nahradí stejným identifikátorem, který byl použit v makru DECLARE_HANDLER a za *jméno_třídy_předka* se dosadí název třídy předka, například TVisualADV. V případě, kdy by nově vytvářená třída abstraktního datového pohledu měla více předků typu abstraktního datového pohledu, lze použít upraveného makra END_HANDLER2 resp. END_HANDLER3 pro dva nebo tři předky. V tomto případě jsou jména tříd předků oddělených čárkami uvedena za makrem v kulatých závorkách.

Mezi makry IMPLEMENT_HANDLER a END_HANDLER se uvádí seznam maker, která určují typy zpráv, které budou zpracovávány metodami nové třídy. Každému konkrétnímu makru odpovídá metoda s předem definovaným jménem a argumenty. Tato metoda bude volána v okamžiku doručení zprávy do instance třídy. Podrobnosti o typech zpráv a jejich argumentech lze nalézt v příloze A.

5.2.3 Metody třídy TVisualADV

Nyní se zmíníme o třídě TVisualADV. V této třídě, která je potomkem třídy TBaseADV, je možné kromě již dříve uvedených metod předka překrývat následující dvě metody:

```
virtual void SetupDraw();
virtual void Draw(TADVContext *context);
```

Překrytí metody *SetupDraw*() není příliš časté. Tato metoda je volána ještě před započetím vlastní grafické prezentace pohledu a je vhodné ji překrýt například v případě, kdy je třeba v závislosti na změně stavu ADP nebo ADO změnit hodnoty některých datových členů.

Zcela zásadní význam pro grafickou prezentaci pohledu má však metoda *Draw*(). Tato metoda, která zabezpečuje vlastní vykreslení okna abstraktního datového pohledu, musí být překryta v každé třídě. Uvnitř metody se nachází posloupnost volání metod grafického kontextu okna pohledu. K tomuto kontextu je umožněn přístup pomocí parametru *context*. Podrobnosti o kontextu lze nalézt v příloze B.4.

V potomcích třídy TVisualADV by naopak neměla být překrývána metoda *PresentState*(). Tato metoda je již překryta takovým způsobem, který zabezpečuje správné nastavení ořezávací oblasti pro kreslení datového pohledu a uvnitř metody je následně volána metoda *Draw*() pro vykreslení okna pohledu.

Konečně je zřejmé, že každá nová třída musí mít deklarován svůj konstruktor. V tomto konstruktoru je třeba vždy určit tvar pohledu přiřazením patřičného typu regionu do proměnné *Region*.

5.2.4 Metody třídy TConnectADV

Další větev hierarchie tříd tvoří třída TConnectADV. Tato třída specializuje obecný typ TBaseADV tak, aby byla umožněna jednoduchá síťová komunikace a prezentace abstraktních datových objektů. Tato specializace je podobně jako ve třídě TVisualADV provedena pomocí překrytí metod *PresentState*() a *ChangeState*() a proto by tyto metody neměly být nadále překrývány. V této třídě se nacházejí následující metody, které je třeba v potomcích překrývat:

```
virtual void  InputState(void *data);
virtual void  OutputState(void *data);
virtual int   StateSize();
```

První metoda je volána v okamžiku příjmu události o změně stavu abstraktního datového objektu, který je vlastníkem síťového ADP se kterým má přijímající pohled navázáno spojení. Uvnitř této metody je třeba provést transformaci dat z vyrovnávacího bufferu *data* na nový stav abstraktního datového objektu

vlastníka pohledu. Tuto metodu není třeba překrývat jen v případě jednosměrné komunikace, kdy instance vytvářené třídy budou hrát roli přijímajícího pohledu.

Druhá metoda je naopak volána v okamžiku, kdy došlo ke změně stavu asociovaného ADO a má tak dojít k prezentaci tohoto stavu vzhledem k okolnímu světu. V tomto okamžiku je třeba do vyrovnávacího bufferu naskládat informace o datové prezentaci, které budou zaslány napojenému ADP, který tyto informace naopak dekóduje uvnitř metody *InputState*(). Metodu není třeba překrývat, pokud instance této třídy budou umožňovat jen jednosměrnou komunikaci a vždy budou hrát roli vysílajícího pohledu.

Třetí metoda by měla být překryta vždy. Její implementace je však velmi jednoduchá. Návratovou hodnotou metody totiž musí být velikost vyrovnávacího bufferu, který je třídou abstraktního datového pohledu využíván.

5.2.5 Zbývající požadavky na tvorbu aplikace

Poté, co jsou implementovány třídy abstraktních datových pohledů, které bude daná aplikace používat a existuje již také kód jádra aplikace včetně deklarace vnitřních datových struktur, lze přistoupit k sestavení celé aplikace.

První fázi představuje určení vzájemného vztahu mezi instancemi konkrétních datových pohledů a datovými objekty, které budou tyto instance vlastnit.

Následně lze přistoupit k druhé fázi a to vytvoření instancí datových pohledů. V této fázi se již během volání konstruktoru provede asociace ADP a ADO. Současně také dochází k vytvoření logického vztahu mezi rodičovskými a dětskými pohledy. Tento vztah je určen pomocí identifikátoru pohledu, který každý datový pohled dostane při svém vytvoření. K fyzickému propojení mezi abstraktními pohledy dochází až v okamžiku, kdy je volána metoda *Open*().

V prvních dvou fázích tak pomocí kompozice vznikne vlastní struktura aplikace, která je stále ještě zcela nezávislá na použitém operačním systému. K určení typu operačního resp. síťového systému dochází až ve třetí fázi. Zde je nejprve třeba zvolit knihovnu, která podporuje ten konkrétní systém a pak už není nic jednoduššího než vytvoření instance konkrétní továrny abstraktních datových pohledů a její přiřazení do systému pomocí funkce *AssignADVFactory*().

Nyní již zbývá jen spustit překladač a linker a aplikace je připravena k provozu. V případě, kdy je třeba přejít například z prostředí operačního systému Windows95 do prostředí X Window, není nic jednoduššího, než použít při překladu jinou knihovnu pro podporu abstraktních datových pohledů a funkcí *AssignADVFactory*() nastavit v systému továrnu pohledů pro X Window.

6 Závěr

Na závěr této diplomové práce bych chtěl zhodnotit výsledky, kterých bylo během jejího zpracování dosaženo. V některých směrech se jedná o výsledky vynikající, v jiných směrech se alespoň podařilo prokázat možnost či opodstatněnost použití konceptu.

První otázkou je, zda bylo dosaženo cílů, které byly vytyčeny na počátku diplomové práce. Domnívám se, že tyto cíle byly splněny. Při návrhu systému abstraktních datových pohledů byly dodrženy všechny požadované vlastnosti a charakteristiky. Implementace ADP plně podporuje oddělení uživatelského rozhraní aplikace od jejího výkonného jádra, umožňuje vytváření složitějších pohledů pomocí kompozice, na tvar vizuálních pohledů nejsou kladeny žádné omezující požadavky, abstraktní datové pohledy umožňují používat principů dědičnosti, mají schopnost zachytávat a zpracovávat události a mnoho dalšího.

Navíc se podařilo splnit i mnoho doplňujících a rozšiřujících požadavků. Především se podařilo koncept ADP rozšířit i na použití pohledů pro datové transformace a komunikaci po síti. Dále se podařilo dosáhnout absolutní nezávislosti návrhu abstraktních datových pohledů na používaném operačním či síťovém systému a tím bylo dosaženo maximální přenositelnosti jednou napsaného kódu aplikace.

Pro popis abstraktních datových pohledů bylo použito objektově orientované modifikace specifikace VDM. Tato specifikace umožňuje jednoduchým, ale přitom přesným způsobem popsat základní chování jak samotné aplikace, tak abstraktních datových pohledů, které jsou v této aplikaci používány. Při návrhu i používání této specifikace se prokázalo, že možnosti popisu programového systému pomocí specifikací jsou zatím ne zcela doceněny.

Otázkou zůstává, jaká je možná využitelnost výsledků diplomové práce v praxi. Na tomto místě je třeba říci, že cílem diplomové práce nebylo v žádném případě vytvořit profesionální systém určený k všeobecnému použití. Z tohoto pohledu je třeba se dívat i na implementovaný systém abstraktních datových pohledů. V každém případě se ukázalo, že základní myšlenka abstraktních datových pohledů je životaschopná a že tento koncept je vhodný pro tvorbu především vysoce interaktivních aplikací. Samozřejmě výsledkem práce nemohl být masově používaný systém. Aby bylo možno takového cíle dosáhnout, bylo by třeba vyvinout ještě značného úsilí mnoha lidí na poli výzkumu i implementace. Přesto však lze říci, že praktické použití výsledků této práce je možné.

Závěrem bych chtěl vyjádřit naději, že tato diplomová práce přispěje alespoň k malému pokroku v oblasti návrhu a tvorby znovupoužitelných softwarových komponent a jejich popisu pomocí matematických specifikací.

Resumé

During last few years technological development in the area of software engineering has reached a great deal of progress. At present a basis of modern technics of programming is formed by reusability of software components in many different ways. One of the most popular ways is building new software systems by the composition of high-level components. This diploma work describes a new programming concept called Abstract Data View (ADV) which is a design and programming mechanism providing a clean separation of the user interface from the aplication both at the design and implementation level. Moreover this concept can be used not only for a graphical presentation of an application state, but also as a tool for a module and application interconnection. These enhanced principles of the ADV concept lead up to more flexible application design and higher level of abstraction and reusability. The ADV concept in this paper is described through comprihesive examples written in an object-oriented modification of a VDM notation and the ANSI C++ programming language. A software package which implements all the ADV behaviour and its functionality forms inseparable part of this work.

Literatura

[1] L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, C. J. P. Lucena: *User Interface High-Order Architectural Models*, University of Waterloo, Computer Science Department, Waterloo, Ontario, Canada, 1993.

[2] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, T. M. Stepien: *Abstract Data Views: An Illustrative Example*, University of Waterloo, Computer Science Department, Waterloo, Ontario, Canada, February 1992.

[3] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, T. M. Stepien: *Abstract Data Views: The Concept and Its Formal Specification*, University of Waterloo, Computer Science Department, Waterloo, Ontario, Canada, February 1992.

[4] D. D. Cowan, C. J. P. Lucena: *Abstract Data Views: A Module Interconnection Concept to Enhance Design for Reusability*, University of Waterloo, Computer Science Department, Waterloo, Ontario, Canada, November 1993.

[5] D. D. Cowan, T. M. Stepien, R. Ierusalimschy, C. J. P. Lucena: *Application Integration: Constructing Composite Applications from Interactive Components*, University of Waterloo, Computer Science Department, Waterloo, Ontario, Canada, March 1992.

[6] I. Fořt: *Windows 3.1 – techniky programování*, Grada, Praha, 1993.

[7] E. Gramma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, Reading, Massachusetts, 1995.

[8] C. B. Jones: *Systematic Software Development Using VDM*, Prentice Hall, New York, second edition, 1990.

[9] J. A. Larson: *Interactive Software – Tools for Building Interactive User Interfaces*, Yourdon Press Computing Series, 1992.

[10] A. Limpouch: *X Window System – programování aplikací*, Grada, Praha, 1993.

[11] H. Partl, E. Schlegl, I. Hyna, P. Sýkora: *LaTeX – Stručný popis*, ČVUT, Praha, 1992.

[12] A. B. Potengy, C. J. P. Lucena, D. D. Cowan, R. Ierusalimschy: *Program Design & Implementation With Abstract Data Views*, University of Waterloo, Computer Science Department, Waterloo, Ontario, Canada, March 1993.

[13] R. Seroul, S. Levy: *A Beginner's Book of TeX*, Springer-Verlag, New York, 1991.

[14] M. Woodman, B. Heal: *Introduction to VDM*, McGraw-Hill, London, 1993.

[15] *Borland C++ 4.5 – ObjectWindows 2.0 Online Help*, Borland International, Scotts Valley, CA, 1994.

[16] *Turbo Pascal – Turbo Vision Guide*, Borland International, Scotts Valley, CA, 1990.

[17] *Win32 SDK Online Help*, Microsoft Corporation, Seattle, WA, 1995.

A Popis událostí systému

Všechny události, které mohou v systému abstraktních datových pohledů nastat mají svoji zprávu. Identifikátor zprávy začíná znaky MSG_xxxx, makro, které zajistí, že daná událost bude zpracována začíná znaky EV_xxxx a metody, které danou událost obslouží začínají znaky Evxxxx. Za symboly xxxx se dosadí jméno zprávy psané velkými písmeny. Toto jméno je uvedeno v názvu metody, která zprávu obsluhuje. Seznam metod s popisem jejich funkčnosti následuje.

BOOL EvPresentState();
Obsluhuje zprávu, která žádá o novou prezentaci stavu; volá metodu *PresentState*(). Zpravidla by neměla být překryta.

BOOL EvChangeState(**void** *data);
Obsluhuje zprávu, která požaduje změnu stavu; volá metodu *ChangeState*(). Zpravidla by neměla být překryta.

BOOL EvOwnerChanged(**void** *address);
Obsluhuje zprávu, která říká, že ADO s adresou *address* změnil svůj stav. V případě, že aktuální ADP tento ADO zobrazuje, dojde k nové prezentaci stavu.

BOOL EvAnswerQuery(TBaseADV *parent);
Interní zpráva, přijímá odpověď na zaslanou zprávu MSG_QUERYADDRESS.

BOOL EvQueryAddress(**const** TId &applicant);
Interní zpráva, ADP posílá pohledu *applicant* svoji fyzickou adresu v paměti.

BOOL EvChangeId(**const** TId &from, **const** TId &to);
Interní zpráva, informuje o změně identifikátoru ADP z *from* na *to*.

BOOL EvOpen();
Interní zpráva, požadavek na otevření pohledu.

BOOL EvClose();
Interní zpráva, požadavek na uzavření pohledu.

void EvLButtonDblClk(UINT modKeys, **const** TADVPoint &point);
Dvojitý stisk levého tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvLButtonDown(UINT modKeys, **const** TADVPoint &point);
Stisk levého tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvLButtonUp(UINT modKeys, **const** TADVPoint &point);
Uvolnění levého tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvMButtonDblClk(UINT modKeys, **const** TADVPoint &point);
Dvojitý stisk prostředního tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvMButtonDown(UINT modKeys, **const** TADVPoint &point);
Stisk prostředního tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvMButtonUp(UINT modKeys, **const** TADVPoint &point);
Uvolnění prostředního tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvRButtonDblClk(UINT modKeys, **const** TADVPoint &point);
Dvojitý stisk pravého tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvRButtonDown(UINT modKeys, **const** TADVPoint &point);
Stisk pravého tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvRButtonUp(UINT modKeys, **const** TADVPoint &point);
Uvolnění pravého tlačítka myši na pozici *point* s příznaky *modKeys*.

void EvMouseMove(UINT modKeys, **const** TADVPoint &point);
Pohyb kurzoru myši na pozici *point* s příznaky *modKeys*.

BOOL EvKeyDown(UINT key, UINT repeatCount, UINT flags);
Stisk tlačítka klávesnice; *key* obsahuje kód definovaný pro každou klávesu v hlavičkovém souboru, *repeatCount* určuje počet opakování stisku té samé klávesy, *flags* nese příznaky o stisku kláves.

BOOL EvKeyUp(UINT key, UINT repeatCount, UINT flags);
Uvolnění stisku tlačítka klávesnice; *key* obsahuje kód definovaný pro každou klávesu v hlavičkovém souboru, *repeatCount* určuje počet opakování stisku té samé klávesy, *flags* nese příznaky o stisku kláves.

BOOL EvChar(UINT key, UINT repeatCount, UINT flags);
Zpráva o vygenerování znaku; *key* obsahuje kód definovaný pro každý znak v hlavičkovém souboru, *repeatCount* určuje počet opakování stisku té samé klávesy, *flags* nese příznaky o stisku kláves.

BOOL EvSetFocus();
Informuje o tom, že pohled se stal zaostřeným (získal tzv. focus) a bude příjímat zprávy od klávesnice.

BOOL EvKillFocus();
Informuje o tom, že pohled ztratil focus.

BOOL EvReleaseCapture();
Informuje o tom, že pohled ztratil capture.

BOOL EvBind(**const** TId &id);
Požadavek na navázání spojení s ADP *id*.

BOOL EvRebind(**const** TId &id);
Požadavek na znovunavázání spojení s ADP po změnách v systému.

BOOL EvUnbind(**const** TId &id);
Požadavek na rozvázání spojení s ADP *id*.

B Popis metod některých tříd

B.1 Třída TBaseADV

TBaseADV(**const** TId &parent, **void** *owner = NULL);
Konstruktor třídy; parametr *parent* obsahuje identifikátor rodičovského pohledu nebo okna systému, ve kterém má být právě vytvářený pohled umístěn. Parametr *owner* obsahuje ukazatel na abstraktní datový objekt, který má být s pohledem asociován.

void Open();
Voláním metody dojde k vytvoření fyzických vazeb mezi pohledy. Teprve po zavolání metody je abstraktní datový pohled připraven pracovat.

void Close();
Metoda opačná k *Open*(), musí být volána vždy před zrušením datového pohledu.

void Enable();
Volání metody povoluje ADP příjem zpráv systému.

void Disable();
Volání metody zakazuje příjem zpráv od systému.

void PresentState();
Metoda je volána vždy, když je třeba prezentovat stav asociovaného datového objektu. Podrobnosti lze nalézt v kapitole 5.2.

BOOL ChangeState(**void** *data);
Metoda je volána vždy, když je požadována změna stavu datového objektu.

BOOL SendADVMessage(TMessage &msg);
Okamžité zaslání zprávy cílovému objektu.

BOOL PostADVMessage(TMessage &msg);
Zaslání zprávy cílovému objektu, kdy není zabezpečena okamžitá reakce.

BOOL DispatchMessage(TMessage &msg);
Metoda, která zabezpečuje přijímání a obsluhu zpráv od systému.

TBaseADV *GetNewCopy() **const**;
Metodu je třeba překrývat tak, aby vždy vracela novou kopii aktuální instance. Metoda se využívá při tvorbě komplexnějších pohledů, které se skládají z dětských pohledů stejného typu.

int Transfer(**void** *buffer, TDirection direction);
Metoda zabezpečuje transfer stavu ADP z a do bufferu. V případě, že se má pohled účastnit tohoto transferu, je třeba metodu překrýt.

int TransferData(TDirection direction);
Metoda vykonává transfer dat do vlastního pohledu i dětských pohledů.

void SetTranferBuffer(**void** *buffer);
Nastavení bufferu pro transfer dat.

void EnableTransfer();
Povolení transferu dat.

void DisableTransfer();
Zákaz transferu dat pro tento pohled.

TBaseADV *ParentADV() **const**;
Vrací ukazatel na rodičovský ADP, pokud existuje. Jinak vrací NULL.

TBaseADV *ChildADV() **const**;
Vrací ukazatel na první dětský pohled, pokud takový pohled existuje. Když neexistuje, vrací NULL.

TBaseADV *NextADV() **const**;
Vrací ukazatel na sourozenecký pohled, který se nachází na konci Z–pořadí.

TBaseADV *PrevADV() **const**;
Vrací ukazatel na sourozenecký pohled, který se nachází hned pod aktuálním pohledem.

TBaseADV *FirstThat(TCondFunc func, **void** *param = NULL);
Metoda umožňuje vyhledávání prvního pohledu ze seznamu dětských pohledů, který splňuje požadovanou funkci.

TBaseADV *LastThat(TCondFunc func, **void** *param = NULL);
Metoda umožňuje vyhledávání posledního pohledu ze seznamu dětských pohledů, který splňuje požadovanou funkci.

void ForEach(TIterFunc func, **void** *param = NULL);
Metoda umožňuje pro každý dětský pohled vykonat akci určenou funkcí *func*. Tato funkce je vždy volána s parametrem *param*.

int CountSiblings() **const**;
Vrací počet sourozeneckých pohledů.

int CountChildren() **const**;
Vrací počet dětských pohledů.

void SetId(**const** TId &id);
Metoda umožňuje změnu identifikátoru pohledu.

const TId & GetId() **const**;
Metoda vrací identifikátor aktuálního pohledu.

const TId & GetParentId() **const**;
Metoda vrací identifikátor rodičovského pohledu.

operator TId() **const**;
Operátor má stejnou funkci, jako metoda *GetId*().

void SetupADV();
Toto je první metoda, která je volána po tom, co fyzicky vznikne abstraktní datový pohled. Uvnitř metody se zajišťuje otevření všech dětských oken a provádí se také transfer dat.

void CleanupADV();
Tato metoda je volána jako poslední před uzavřením abstraktního datového pohledu. Zabezpečuje uzavření všech dětských pohledů a zpětný transfer dat.

void SetupChildOwners();
Metoda je volána před prezentací abstraktního pohledu. Cílem této metody je připravit vlastníky dětských pohledů. Podrobnosti lze nalézt v kapitole 5.2.

void PrepareParentAddress();
Tato metoda je volána interně a zabezpečuje zjištění fyzické adresy rodičovského pohledu, když je znám jeho identifikátor.

BOOL ContainsAddress(**void** *address);
Tato virtuální metoda je volána jako reakce na na změnu abstraktního datového objektu. Cílem této metody je zjistit, zda adresa *address* neodpovídá některé adrese v asociovaném datovém objektu. Standardně ze zjišťuje, zda adresa objektu, který změnil stav, není shodná s adresou uloženou v proměnné *Owner*. V případě, kdy je pohled asociován s ADO, který má složitější strukturu, je třeba tuto metodu překrýt tak, aby byly správně rozeznány i jednotlivé složky ADO.

void OwnerChanged(**void** *address = NULL);
Tato metoda by měla být volána vždy v okamžiku, kdy abstraktní datový pohled modifikuje stav asociovaného datového objektu. Volání zabezpečuje předání informace o změně stavu ostatním pohledům, které jsou s daným objektem asociovány.

B.2 Třída TVisualADV

TVisualADV(**const** TId &parent, **const** TADVPoint &offset, **void** *owner);
Konstruktor třídy; *parent* obsahuje identifikátor rodičovského datového pohledu nebo okna, *offset* zadává relativní souřadnice tohoto pohledu vzhledem k rodičovskému pohledu, *owner* obsahuje ukazatel na asociovaný ADO.

void Show();
Volání této metody zabezpečí zobrazení skrytého ADP.

void Hide();
Volání této metody zabezpečí skrytí abstraktního datového pohledu.

void Move(int dx, int dy);
Volání metody přesune datový pohled od *dx* bodů vodorovně a *dy* bodů svisle.

void SetFocus();
Nastaví focus datového pohledu. V tomto případě budou chodit události od klávesnice právě tomuto datovému pohledu a v případě, že nebudou obslouženy, tak rodičovským pohledům.

void SetCapture();
Nastaví capture datovému pohledu. V tomto případě budou chodit události od myši právě tomuto pohledu a to i v případě, že se myš nenachází právě nad tímto pohledem. Standardně chodí události myši tomu pohledu, nad kterým se myš právě nachází.

void ReleaseCapture();
Zruší nastavení capture pro datový pohled.

void SetupDraw();
Metoda, která je volána těsně před započetím vykreslování datového pohledu. Je to vhodné místo pro změnu tvaru pohledu a kontrolu invariant pohledu.

void Draw(TADVContext *context);
Metoda je volána při požadavku na překreslení pohledu.

B.3 Třída TConnectADV

TConnectADV(**const** TId &parent, **const** TId &map, **void** *owner);
Konstruktor třídy, ve kterém *parent* určuje identifikátor rodičovského pohledu, *map* určuje identifikátor pohledu, se kterým bude udržována síťová komunikace a *owner* určuje adresu asociovaného datového objektu.

void OutputState(**void** *data);
Tato metoda je volána v okamžiku, kdy je třeba přepsat stav asociovaného datového objektu do bufferu *data*, aby mohl být přenesen k jinému pohledu. Buffer je ještě před započetím volání inicializován na délku, kterou určuje metoda *StateSize*().

void InputState(**void** *data);
Tato metoda je opakem metody předchozí a zabezpečuje přenos stavu uloženého v bufferu na stav asociovaného datového objektu.

int StateSize();
Metoda vrací délku požadovaného bufferu pro tento typ abstraktních datových pohledů.

void SetMapId(**const** TId &id);
Metoda slouží k nastavení identifikátoru pohledu, který bude tvořit s aktuálním pohledem dvojici, mezi kterou bude docházet ke komunikaci a přenosu stavů asociovaných datových objektů.

const TId & GetMapId() **const**;
Metoda vrací identifikátor pohledu, se kterým je udržována komunikace.

B.4 Třída TADVContext

void Clear();
Volání metody způsobí smazání okna grafického pohledu.

void PaintBitmap(**const** TADVRect &dst, **const** TADVPoint &src,
TADVBitmap *bmp);
Metoda do pohledu v oblasti *dst* překopíruje část bitové mapy *bmp* od souřadnice *src*.

void FillRgn(TADVRegion *region = NULL);
Vyplní zadaný region abstraktního pohledu aktuální barvou štětce. Pokud je parametr roven NULL, vyplní se plocha celého pohledu.

void FillRect(**const** TADVRect &rect);
Vyplní obdélník *rect* v pohledu barvou štětce.

void FillEllipse(**const** TADVRect &rect);
Vyplní elipsu vepsanou do obdélníka *rect* barvou aktuálního štětce.

void FillPie(**const** TADVRect &rect, **const** TADVPoint &start,
const TADVPoint &end);
Vyplní výsek elipsy zadané parametrem *rect* od polopřímky zadané spojnicí středu elipsy s bodem *start* do polopřímky zadané bodem *end* barvou aktuálního štětce.

void FillChord(**const** TADVRect &rect, **const** TADVPoint &start,
const TADVPoint &end);
Vyplní úseč elipsy zadané parametrem *rect* a tětivou určenou body *start* a *end*.

void FillPolygon(**const** TADVPoint *points, **int** count);
Vyplní polygon zadaný polem bodů *points* barvou štětce.

void DrawRgn(TADVRegion *region = NULL);
Vykreslí aktuálním perem obrysy regionu *region*.

void DrawRect(**const** TADVRect &rect);
Nakreslí obdélník zadaný parametrem *rect*.

void DrawEllipse(**const** TADVRect &rect);
Nakreslí elipsu vepsanou do obdélníka *rect*.

void DrawArc(**const** TADVRect &rect, **const** TADVPoint &start,
const TADVPoint &end);
Nakreslí část eliptického oblouku zadaného parametry, jejichž význam je obdobný jako u metody *FillPie*().

void DrawPie(**const** TADVRect &rect, **const** TADVPoint &start,
const TADVPoint &end);
Nakreslí výseč elipsy, která je dána parametry se stejným významem jako u předchozí metody.

void DrawChord(**const** TADVRect &rect, **const** TADVPoint &start,
const TADVPoint &end);
Nakreslí úseč elipsy, které je určena parametry.

void DrawPolyline(**const** TADVPoint *points, **int** count);
Nakreslí lomenou čáru určenou polem bodů *points*.

void LineTo(**const** TADVPoint &point);
Nakreslí úsečku z aktuální pozice pera do bodu *point*.

void MoveTo(**const** TADVPoint &point);
Přesune pero do bodu *point*.

void SetPixel(**const** TADVPoint &p,
const TADVColor &color = TADVColor::Black);
Vykreslí bod barvy *color* na souřadnicích *p*.

void PrintText(**const** TADVPoint &point, **const char** *string);
Vytiskne řetězec *string* na souřadnicích *point*.

void GetTextExtent(**const char** *string, **int** count, **int** &cx, **int** &cy);
Naplní parametry *cx* a *cy* počtem bodů, které budou třeba pro vytisknutí *count* znaků řetězce *string*.