Tvorba aplikací v prostředí Black Box Component Builder

Jindřich Černohorský

Katedra měřicí a řídicí techniky, VŠB – Technická univerzita v Ostravě, tř.17 listopadu, 708 33 Ostrava-Poruba, Česká republika

Abstrakt

Po éře objektově orientovaných technologií se objevují na scéně ještě silnější kalibry -komponentní technologie. Nejde o výměnu stráží. Objektově orientované programování je základem komponentních přístupů i když se ukazuje , že k vytvoření komponent nejsou není OOP nezbytně nutné. Prostředí software se však stává stále složitější a proto nabývá stále více na významu jeho návrhová fáze. V oblasti návrhu je proto klíč jak k vytvoření vhodných komponent, tak k jejich spojování, kooperaci a znovupoužití. Typická paradigmata jsou presentována ve tvaru návrhových vzorů [1] a jsou použitelná v řadě situací a aplikací. Příspěvek se zabývá některými aspekty vývoje programového vybavení v prostředí Black Box Component Builder, které využívá řadu návrhových vzorů a které bylo navrženo právě pro tvorbu komponentních aplikací.

Úvod

Z hlediska architektury lze aplikace rozdělit do tří základních kategorií : *aplikační programy, knihovny (toolkity) a frameworky*.

<u>Aplikační program</u> (například dokumentový editor, tabulkový procesor) preferuje jako nejvyšší prioritu *vnitřní znovupoužití*, *udržovatelnost* a *rozšiřovatelnost*. Vnitřní znovupoužití zajišťuje, že se nenavrhuje a neimplementuje víc než je třeba. Použité návrhové vzory, které redukují závislosti, mohou zvýšit vnitřní znovupoužití. Volnější spřažení zvyšuje pravděpodobnost, že jedna třída objektů může spolupracovat s řadou dalších. Aplikace je také lépe udržovatelná, když v ní použité návrhové vzory omezují závislost na platformě a vrstvě systému a omezení spřaženosti také rozšiřuje rozšiřitelnost.

<u>Knihovna</u> je množina souvisejících a znovupoužitelných tříd navržená proto, aby poskytovala užitečnou, obecnou funkcionalitu. (Příklad: soubor tříd pro zpracování seznamů, asociativních tabulek, zásobníků, atp.) Knihovna neovlivňuje architekturu aplikace, v níž je použita. Pouze jí poskytuje svoji funkcionalitu. Knihovny zdůrazňují

znovupoužití kódu. Návrh knihoven je obtížnější, než návrh aplikace, protože knihovna musí být použitelná ve více aplikacích, kterým by měla co nejlépe vyhovět, aniž zná dopředu jejich požadavky.

Framework je množina kooperujících objektů, která představuje znovupoužitelný návrh pro specifickou třídu software. Například framework může být orientován na vývoj grafických editorů pro různé oblasti jako jsou třeba kreslení, hudební kompozice, CAD, atp. Přizpůsobení frameworku konkrétní aplikaci se děje vývojem aplikačně specifických podtříd z abstraktních tříd frameworku. Framework diktuje architekturu nové aplikace: celkovou strukturu, její rozdělení na třídy a objekty, jejich klíčové zodpovědnosti, jak vzájemně spolupracují a jaká je posloupnost jejich řízení.

Architektura frameworku se často opírá o *inverzi řízení mezi uživatelskou částí aplikace (rozšířením frameworku) a základem frameworku*, nad nimž je vytvořena. Když se používá knihovna, napíše se hlavní tělo aplikace a volá se kód, který se znovu-používá. V aplikaci vytvořené s pomocí frameworku je nadřazeným tělo frameworku a nově se píše kód, který tělo volá. *Je tedy nutno napsat operace s určitými jmény a volacími konvencemi.* To redukuje fázi návrhu.

Je-li obtížné navrhnout individuální aplikaci, pak těžší je navrhnout knihovnu, a nejtěžší ze všeho je návrh frameworku. Projektant frameworku usiluje o to, aby jedna architektura byla funkční pro všechny aplikace v oblasti. Jakákoliv podstatná změna v návrhu frameworku může značně redukovat jeho přínosy , protože hlavní přínos frameworku pro aplikaci spočívá v architektuře, kterou definuje.

Frameworky často kladou velké požadavky na fázi učení, která se musí překonat předtím, než se stanou užitečné. Pokud návrhové vzory nemohou tyto požadavky eliminovat plně, mohou je udělat méně náročné tím, že učiní klíčové elementy návrhu frameworku explicitněji zřejmé.

Black Box Component Builder.

Black Box Component Builder (BBCB) je vývojový framework určený pro tvorbu aplikací provozovaných buď na platformě Windows95/NT nebo Apple Macintosh. Jeho součástí je programovací jazyk Component Pascal (CP), který je posledním členem vývojové řady

(Algol->) Pascal -> Modula-2 -> Oberon-2 -> Component Pascal

Jazyk byl navržen tak , aby byl optimálně použitelný pro rozšiřování frameworku BBCB. Bylo by zajímavé porovnat jeho vlastnosti s vlastnostmi jiných programovacích jazyků včetně jeho předchůdců. Prostor vymezený tomuto příspěvku to však nedovoluje. Proto si všimneme pouze toho, co má největší význam právě pro rozvíjení frameworku BBCB, protože to je na celém systému nejzajímavější. Pro porozumění zápisům v dalším textu však musíme ve stručnosti uvést, jak CP zachází s objektovým programováním.

Objektové programování v Component Pascalu.

Při definici třídy vychází Component Pascal z definice záznamu

```
TYPE Base = RECORD              nebo       TYPE Base = POINTER TO RECORD
    I : INTEGER;                              I : INTEGER;
END;                                       END;
```

a umožňuje definovat jeho rozšíření

```
TYPE Extend = RECORD( Base)        nebo      TYPE Extend = POINTER TO RECORD(Base)
    R : REAL;                                   R : REAL;
END;                                        END;
```

Metody se k záznamům připojují. Před název PROCEDURY (metody) se napíše příjemce (tj. záznam) metody. Například

```
    PROCEDURE (e:Extend ) Set ( I: INTEGER; R:REAL );
    BEGIN    e.I:=I;    e.R:=R;         END Set;
```

To co jsme právě uvedli nám pomůže porozumět zápisům v dalším textu.

Atributy typového systému ComponentPascalu.

Pro dosažení větší vyjadřovací síly jazyka je použito v CP některých prostředků, z nichž nejvýraznější jsou atributy typového systému. Některé konkrétné modely budou presentovány v dalších souvisejících příspěvcích na této konferenci.

Výsledky funkcí jako kovariantní ukazatelé

Typově vázaná funkce, která vrací ukazatel, může být předefinována tak, že vrací nějaký jeho rozšířený typ. Například funkce

PROCEDURE (v: View) ThisModel (): Model

může být rozšířena na nějaký podtyp *MyView*, o němž předpokládejme že je podtypem *View*, takže má následující signaturu

PROCEDURE (v: MyView) ThisModel (): MyModel

a kde o typu *MyModel* se předpokládá že je podtypem typu *Model*. Kovariantní funkční výsledky jsou typově bezpečné, protože pouze jednoduše zesilují post-podmínky nějaké funkce, což je vždy přípustné. Umožňují tím zpřesnit deklarace rozhraní.

Kompatibilita ukazatelů

Ukazatele jsou kompatibilní podle struktury; například dva ukazatelové typy odvozené z téhož základního typu jsou kompatibilní. To může být užitečné většinou v signaturách procedur, kde je možné užít funkci podobnou následující :

PROCEDURE P (p: POINTER TO ARRAY OF INTEGER)

Funkční výsledky jako ukazatele

Návratovým typem funkce může být *Specifikace typu* ne *Identifikátor typu.* Například je možno napsat

PROCEDURE *Cosi*(): POINTER TO Rec

a v důsledku zjednodušených pravidel kompatibility může mít toto použití skutečně smysl.

Mody parametrů IN a OUT

Tyto mody parametrů jsou podobné modu *VAR* , až na několik omezení. *IN* parametry nemohou být přímo modifikovány uvnitř procedury. *OUT* parametry se předpokládají nedefinované na vstupu procedury (s výjimkou ukazatelů a procedurálních proměnných, které jsou na vstupu do procedury nastaveny na *NIL*). Parametry typu record s modem *OUT* musejí být shodného formálního i aktuálního typu.

Tyto mody parametrů jsou důležité pro signatury procedur distribuovaných objektů a mohou zvýšit vhodnost a efektivnost. Většinou umožňují významně zpřesnit a samodokumentovat deklarace rozhraní. Konkrétně tam, kde dříve byly z důvodu efektivnosti používány pouze parametry *VAR*, je nyní možné užít *IN* parametry a to i pro typy záznam a pole.

Metody s atributem NEW

Component Pascal požaduje, aby definice nových metod byly indikovány explicitně. To se děje připojením identifikátoru *NEW* k signatuře metody. Přirozeně, že atribut *NEW* nemůže být použit pro rozšiřující metody. V následujícím příkladě je metoda *NovaMetoda* definována jako nová v *T* a zděděna za předpokladu, že *T1* je rozšířením *T*:

```
PROCEDURE (t: T) NovaMetoda (x, y: INTEGER), NEW;
BEGIN    ...    END NovaMetoda ;
PROCEDURE (t: T1) NovaMetoda (x, y: INTEGER);
BEGIN       ...     END NovaMetoda;
```

Atribut *NEW* umožňuje překladači například detekovat, že bylo změněno jméno základní metody, ale jména rozšiřujících metod nebyla změněna odpovídajícím způsobem. Kompilátor tak detekuje, je-li metoda deklarována poprvé, ačkoliv již existuje v základním typu nebo podtypu. Tyto kontroly umožňují dosáhnout konsistence po změnách v rozhraních frameworku.

Atributy záznamů Default, EXTENSIBLE, ABSTRACT, a LIMITED

Component Pascal používá k označení rozhraní objektů i k jejich implementaci jednoduchou konstrukci: typ záznam. To umožňuje „zmrazit“ některé implementační aspekty rozhraní a ostatní nechat otevřené. Je to často žádoucí ve složitých prostředích, kde je velmi důležité sdělovat závažná architektonická rozhodnutí co nejpřesněji, protože mohou ovlivnit velký počet klientů.

Z tohoto důvodu mohou být záznamové typy v Component Pascalu doplňovány atributy umožňujícími formulovat několik základních architekturálních rozhodnutí explicitně. Kompilátor pak může verifikovat shodu s těmito rozhodnutími. Atributy jsou *EXTENSIBLE*, *ABSTRACT* a *LIMITED* a umožňují rozlišovat čtyři různé kombinace rozšíření a alokačních možností:

Modifikátor	rozšíření	alokace	přiřazení záznamu
žádný("final")	ne	ano	ano
EXTENSIBLE	ano	ano	ne
ABSTRACT	ano	ne	ne
LIMITED	ne*	ne*	ne

* s výjimkou modulu , kde je objekt definován

Záznamové typy mohou být buď rozšiřitelné nebo nerozšiřitelné ("finální"). Implicitně je záznamový typ finální. Finální typy umožňují "uzavřít" nějaký typ tak, že implementace typu může být analyzována a zlepšována, aniž by to ovlivňovalo nepříznivě klienty. Záznamové typy mohou být buď alokovány staticky nebo dynamicky, nebo alokaci může být bráněno (atribut *ABSTRACT*). Může být také omezena pouze na modul, kde je typ definován (atribut *LIMITED*). S omezenými (limited) typy je tedy alokace a rozšíření možné pouze v definujícím modulu, nikdy ne v importujícím modulu. Finální typy jsou zpravidla jednoduché doplňkové datové typy, například:

```
Point = RECORD      x, y: INTEGER          END
```

Proměnné těchto typů mohou být kopírovány s použitím přiřazovacího operátoru, například *pt := pt2*. Na druhé straně rozšiřitelné záznamy nemohou být nikdy kopírovány, ani předávány jako hodnotové parametry (tyto parametry předpokládají přiřazení záznamů). Finální typy, podobně jako rozšiřitelné typy, mohou být rozšířeními jiných záznamových typů a mohou mít metody. Rozšiřitelné typy jsou deklarovány následujícím způsobem:

```
Frame = EXTENSIBLE RECORD     l-, t-, r-, b-: INTEGER       END
```

Prázdné typy druhu *EXTENSIBLE* se používají zřídka. Typičtější namísto toho je použití typů *ABSTRACT*, které jsou speciálními případy typů *EXTENSIBLE* a které není možno alokovat (vytvořit instance). Přesněji to znamená , že typy hodnot (ve smyslu *values)* nemohou být nikdy abstraktní, avšak typy proměnných (*variables)* mohou být abstraktní.

Typ hodnoty může být odlišný od typu proměnné, která tuto hodnotu obsahuje jenom tehdy, je-li proměnná typu odkaz; tj. nějaký ukazatel, nebo VAR, IN, nebo OUT parametr. Proto jenom tyto proměnné mohou být deklarovány jako proměnné abstraktního typu. Ve všech ostatních případech, například u statických proměnných, položkách záznamů, typech položek polí, a hodnotových parametrech, musejí být použity ne-abstraktní typy nebo ukazatele (ukazující případně na abstraktní typy). Protože alokace operací *NEW* vytváří hodnotu typu argumentu, můžeme použít *NEW* jen pro proměnné ne-abstraktního typu.

Příklad abstraktního typu:
TextView = POINTER TO ABSTRACT RECORD (Views.View) END

Abstraktní typy jsou návrhové prostředky, definující rozhraní objektů. Jsou prvořadými prostředky Component Pascalu pro modelování rozhraní komponent. Označování záznamu jakožto abstraktních umožňuje indikovat přesněji použití záznamu: tj. spíše jako konstruktu pro rozhraní než konstruktu pro implementaci.Přesto však může mít abstraktní typ všechny typy metod (viz níže), tj., není na něm vyžadováno, aby byl beze zbytku abstraktní.

Typy *LIMITED* jsou speciálními případy finálních typů. Jsou zvláštní tím, že jejich proměnné mohou být vytvořeny pouze uvnitř je definujícího modulu, a nemohou být kopírovány. Například, klient nemůže provést operaci *NEW* s proměnnou limitovaného typu. Protože alokace je pod plnou kontrolou definujícího modulu, programátor tohoto modulu musí zaručit, že nově alokované proměnné jsou správně inicializovány dříve, než budou dostupné v zákaznických modulech.

Typické řešení tohoto problému představují tzv. faktory funkce nebo faktory objekty, které jsou vytvářeny za účelem vytvoření nových instancí dynamických typů druhu *LIMITED*.
Příklad:
Semaphore = POINTER TO LIMITED RECORD END;
PROCEDURE New (level: INTEGER): Semaphore;

V BBCB frameworku, je většina abstrakcí vyjádřena jako abstraktní typy, které jsou implementovány pomocí (ne-exportovaných) finálních typů. To je jiný přístup, který umožňuje zaručit korektní inicializaci, ale je příliš nevhodný pro jednoduché nerozšiřující se abstrakce. K tomu nemohou být *LIMITED* typy nahrazovány rozšířeními vytvořenými na straně klienta. To je důležité, protože to umožňuje chránit nerozšiřovatelné služby (například real-time kernel) před použitím nelegálních typů.

Syntaxe záznamu
Syntaxe záznamu je taková:
RecordType = [EXTENSIBLE | ABSTRACT | LIMITED] RECORD ["(" QualIdent ")"] FieldList {";" FieldList} END.

Smyslem zavedení uvedených atributů je zvětšení statické vyjadřovací síly rozhraní tak, aby důležité architektonické prvky mohly být zachyceny explicitně a takovým způsobem, že překladač může kontrolovat souhlas nějaké implementace nebo klienta s kontraktem definovaným rozhraním. V čistém implementačním jazyce by těchto nových atributů nebylo třeba. Tuto potřebu vyjádřit nějaká omezení návrhu mají pouze komponentě orientovaná implementace *a návrhový* jazyk. Dohled nad takovými omezeními dovoluje návrháři frameworku stanovit důležité invarianty celého softwarového systému (= systémovou architekturu), a tím zlepšit bezpečnost, udržovatelnost a rozšiřovatelnost systému.

Metody Default a EXTENSIBLE

Podobně jako typy záznam , mohou být i metody těchto typů označeny atributy. Přípustné atributy jsou zde default (žádný atribut), *EXTENSIBLE*, *ABSTRACT* a *EMPTY*. Podobně jako u typů záznam i metody jsou implicitně finální:

```
PROCEDURE (t: T) StatickaProcedura (x, y: INTEGER), NEW;
BEGIN       ...     END StatickaProcedura;
```

Metody, které jsou jak nové (new), tak finální může kompilátor považovat za normální procedury, protože tyto metody nevyžadují pozdní vazbu. Nicméně jejich použití bude patřičnější, budou li patřit jasně k určitému konkrétnímu typu. Rozšiřitelné metody jsou jako takové označovány například takto:

```
PROCEDURE (t: T) Method (x, y: INTEGER), NEW, EXTENSIBLE;
BEGIN       ...     END Method;
```

Je však obecnější užívat abstraktních nebo prázdných (empty) metod, které jsou speciálními případy rozšiřitelných (extensible) metod.

Deklarování metody jako finální se dosáhne jednoduše vynecháním atributu *EXTENSIBLE*:

```
PROCEDURE (t: T) KonecnaZdedenaMetoda (x, y: INTEGER);
BEGIN    ...     END KonecnaZdedenaMetoda;
```

Používáme li návrhový styl black-box, je většina metod, které musí být vytvořeny pro rozšíření frameworku uvedeného druhu, tj. nevyžadují speciální atributy v hlavičce procedury. Finální metody mohou být svázány s libovolným typem záznam. Rozšiřující (extensible) metody mohou být svázány pouze s rozšiřujícími typy (např. s typy *EXTENSIBLE* nebo *ABSTRACT*).

Protože finální metody nemohou být "přepsány", nemohou být také jimi garantované invarianty a post-podmínky změněny. Připomeňme si, že korektní "rozšíření" metody znamená, že rozšiřující metoda zjemňuje (specializuje) rozšiřovanou metodu. Sémanticky to znamená , že rozšiřující metoda se může spokojit se slabší před-podmínkou a může vytvořit silnější post-podmínku ve srovnání s rozšiřovanou metodou.

Abstraktní (ABSTRACT) metody

Abstraktní metoda se deklaruje takto:

```
PROCEDURE (t: T) SomeMethod (s: SET), NEW, ABSTRACT;
PROCEDURE (t: T) CovariantMethod2 (): NarrowedType, ABSTRACT;
```

Abstraktní metody jsou rozšiřitelné (extensible). Kompilátor kontroluje, zda určitý typ implementuje všechny abstraktní metody, které dědí. Konkrétní rozšíření abstraktní metody (nebo typu) může být považováno za jeho *implementaci*. Abstraktní metody mohou být svázány pouze s abstraktními typy, a nemohou být volány cestou super volání (supercall).Abstraktní metoda nemá odpovídající tělo procedur, existuje pouze

jako hlavička (signatura). Proto již není třeba psát do těla procedury příkazem *HALT* aby se zabránilo volání neimplementovaných metod.

Prázdné (EMPTY) metody

Metoda může být deklarována jako prázdná (*empty)*. Prázdné metody jsou rozšiřitelné. Prázdná metoda je velmi podobná abstraktní metodě v tom, že vytváří prostor (hook) pro funkcionalitu, která může být implementována až v dalších rozšířeních. Nicméně prázdné (empty) metody jsou konkrétní a mohou být volány. Avšak jejich volání nemá žádný účinek, nebyly-li rozšířeny (implementovány),

Prázdné metody zastupují volitelná rozhraní. V prázdné (empty) proceduře není možno zapsat nějaký kód. Proto prázdná (empty) metoda nemá odpovídající tělo procedur a není ji možno volat cestou super volání (supercall). Prázdné (empty) procedury také nemohou vracet funkční výsledky a nemohou mít *OUT* parametry.
Příklad:

```
PROCEDURE (t: T) Broadcast (msg: Message), NEW, EMPTY;
```

Syntaxe metod

Syntaxe metod vypadá takto:

```
TBProc = PROCEDURE Receiver IdentDef [FormalPars] [Attribution].
Attribution = ["," NEW] ["," (EXTENSIBLE | ABSTRACT | EMPTY)].
```

Export metod typu "implement-only "

Metoda může být exportována i jako *implement-only*, s použitím exportní značky "-" namísto exportní značky "*". Export druhu implement-only znamená, že metoda může být implementována vně definujícího modulu, ale nemůže tam být volána.

Zda je metoda exportována normálně nebo jako implement-only je rozhodnuto při první deklaraci metody (*NEW* metoda). Pozdější rozšíření musejí zachovat tentýž druh exportu. Exportované metody druhu implement-only jsou volány z modulu, kde je metoda poprvé definována.Exporty druhu implement-only chrání klienty frameworku před poškozením invariantů frameworku, a ponechávají přitom možnost vytvořit nové implementace typů zavedených frameworkem.

Super-volání (Supercalls)

Protože existuje problém sémantické nestálosti základní třídy [2] (semantic fragile base class problem), doporučuje se vyhnout se super-voláním, pokud je to možné. Nový software je totiž možno navrhnout i tak, že super-volání nebudou potřeba. Toho se dosáhne tím, že se návrh opírá o kompozici a ne o dědičnost implementace. Super volání jsou již považována za zastaralou a překonanou vlastnost [3]. Zatím jsou však v BCCB udržována pro zpětnou kompatibilitu.

Procedurální typy

Procedurální typy jsou méně flexibilní než objekty s metodami. Objekty jsou rozšiřitelné, procedurální typy nejsou. Procedurální typy mohou způsobit značné implementační obtíže pokud jde o bezpečné odinstalování („unload“) kódu. Z těchto důvodů , jsou i procedurální typy považovány za zastaralé [3] . Zatím jsou však udržovány pro zpětnou kompatibilitu a pro implementaci kódu rozhraní na nízké úrovni („callbacks“). V dlouhodobém výhledu však mohou být zcela omezeny

ANYREC a *ANYPTR*

Každý bázový typ record je implicitně považován za nějaké rozšíření nového (new) abstraktního (abstract) standardního typu *ANYREC*, i tehdy když je deklarován bez explicitního bázového typu. *ANYREC* je prázdný (empty) záznam (record) který tvoří kořen všech hierarchií typu record. *ANYPTR* je nový standardní typ, který odpovídá typu *POINTER TO ANYREC*.

Tyto nové typy usnadňují dosažení interoperability mezi nezávisle vyvinutými frameworky tím, že umožňují zcela generické parametry. Lze tedy vycházet z následujících pseudo-definic:

```
ANYREC = ABSTRACT RECORD END;
ANYPTR = POINTER TO ANYREC;
PROCEDURE (a: ANYPTR) FINALIZE-, NEW, EMPTY;
```

Procedura *FINALIZE* je prázdná (empty). Může být implementována pro nějaké rozšíření typu ukazatele. Procedura je volána v nějakém nespecifikovaném čase potom , když se objekt stane nedostupným cestou ostatních ukazatelů , tj když již globálně neukotvený, a předtím, než je objekt dealokován. Finalizátory jsou třeba pro uvolnění zdrojů, které nejsou přímo objekty Component Pascalu; například, sektory souborů, ovladače fontů, ukazatele oken operačního systému atp. Pořadí finalizace není definováno. Objekt je finalizován pouze jednou.

Struktura BBCB frameworku.

BlackBox je komponentní framework, protože může být rozšířen za běhu o nové komponenty. V důsledku toho již jednotlivá aplikace v tradičním slova smyslu neexistuje. Spíše jde o jednou stále rostoucí množinu interagujících komponent. Hlavní výzvou v návrhu frameworku je to jako zajistit uzavřenou integraci komponent a přitom zajistit jejich kooperaci a robustnost.

Hierarchická struktura BlackBoxu.

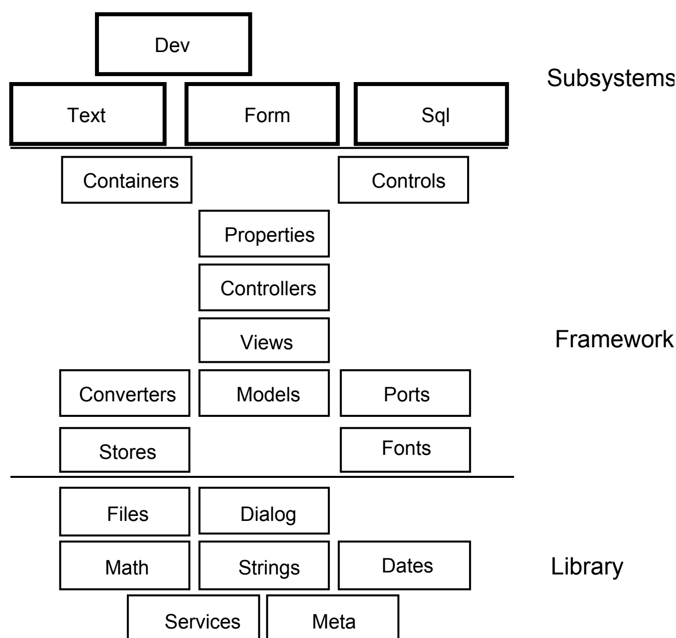
BlackBox je tvořen hierarchií modulů, které dodávají systematicky konstruované abstrakce. Entity, které nejsou užity, mohou být ignorovány (vlastnost „progressive disclosure“). Na spodní úrovni BlackBoxu jsou rozdíly mezi hardwarovými platformy skryty na nejvyšší možnou míru. Na horní úrovni jsou na nejvyšší možnou míru skryty rozdíl mezi uživatelskými rozhraními.

Například v BlackBoxu nejsou vestavěny žádné předpoklady o pod ním ležícím systému oken, například zdali poskytuje překrývání nebo odkládání oken. Ve skutečnosti v BlackBoxu nenajdeme abstrakci „window“ stejně jako abstrakci aplikace.

Protože BlackBox tvoří vrstvu mezi operačním systémem a aplikací, musí být efektivní (účinný). To se netýká pouze rychlosti, ale i nároků na operační a diskovou paměť. Avšak efektivita BlackBoxu přirozeně závisí na kvalitě operačního systému a hostitelského hardware.

Rozhraní BlackBoxu jsou bezpečná. Aplikace v BlackBoxu bude typově-bezpečná pokud nebude užívat nízkoúrovňových nástrojů modelu SYSTEM. Modul, který je typově bezpečný, může způsobit nejvýše nějaké lokální škody i přesto, že obsahuje libovolné chyby. Konkrétně nemůže narušit invarianty garantované ostatními moduly. Lokalizování účinků chyb je zvlášť důležité, jestliže se setkává v jediném dokumentu více komponent odlišného původu.

Klíčovou vlastností BlackBoxu je rozšiřitelnost. Je možné rozšířit většinu BlackBoxových datových typů, aby se existujícímu systému dodala nová funkcionalita. Mnoho základních služeb může být nahrazeno nebo doplněno novými zákaznickými verzemi takovým způsobem, že existující aplikace mohou využít těchto nových možností. Aby bylo možno dosáhnout tohoto stupně rozšiřitelnosti a konfigurovatelnosti, jsou rozhraní modulů konstruována systematickým způsobem.



Uvedený diagram ukazuje modulární strukturu BlackBoxu. Malé pojmenované obdélníky značí moduly. Modul, který může importovat nějaký jiný modul je v hierarchii umístěn výše. Velké zvýrazněné obdélníky znázorňují subsystémy. Každý z nich je tvořen několika spřízněnými moduly.

Hierarchie je strukturována do tří vrstev. Dolní knihovní vrstva obsahuje moduly, která poskytuje řadu vzájemně nijak nesouvisejících služeb. Modul **Services**

poskytuje nízkoúrovňové systémové služby. Modul **Meta** zprostředkuje přístup k informacím typu run-time (metaprogramování). **Math** obsahuje soubor matematických funkcí **Strings** podporuje řetězové operace a konverzní rutiny mezi řetězci a číselnými typy. **Dates** obsahuje datové a časové typy, operace a konverzní rutiny na nich. Modul **Dialog** dispon1uje řadou funkcí pro interakci s uživatelem. **Files** podporuje abstraktní soubory hierarchicky organizovaného souborového systému.

Vrstva frameworku obsahuje služby pro zobrazení, vstup a uložení dokumentů a pro interakce uživatelů orientované na zpracování dokumentů. Knihovna **Fonts** se zabývá identifikací fontů, jejich rozměry a vyhledáváním. Module **Ports** poskytuje abstraktní popis objektu *port*, *rider* a *frame.* Module **Domains** definují sdružovací konstrukty pro libovolné objekty. Module **Stores** definuje základní typ pro všechny rozšiřitelné i uložitelné objekty a souborové formáty (mappery) pro exterualizaci internalizaci paměti (stores). Module **Converters** poskytuje mechanismus pro konverzi mezi formáty souborů a pamětí. Module **Models** definuje základní typ pro vyjádření uložitelných dat stejně jako operací pro řízení modifikací modelu. Modul **Views** definuje centrální abstrakci nějakého pohledu (*view*), který umožňuje prezentaci nějakého modelu.Modul **Controllers** definuje abstrakci *controller*. Controller podporuje interakci uživatele s nějakým pohledem (view) a jeho modelem. Modul **Properties** definuje abstrakci *property*. Tyto abstrakce dovolují modifikovat atributy pohledu generickým a pokud možno neinteraktivním způsobem. Modul **Contaniers** definuje model, *view* a *contoller* pro obecné kontainery, např. pohledy, které mohou obsahovat proměnlivý počet libovolných jiných pohledů. Tato základní abstrakce umožňuje skrýt většinu odlišností mezi uživatelskými rozhraními OLE a OpenDoc. Modul **Controls** obsahuje rozhraní pro řídicí
symboly jako například tlačítka, kontrolní boxy, textová pole atp.

Poslední aplikační vrstva zahrnuje tři standardní subsystémy textový, formulářový (dialogový) a subsystém vývojového prostředí. Další moduly obsažené v této vrstvě v obrázku nejsou zobrazeny.

Moduly, které nejsou zajímavé z obecného hlediska anebo neobsahují přenositelné abstrakce se nazývají soukromé. Jejich rozhraní nemusejí být portabilní, nemusejí být dokumentována nebo se mohou libovolně měnit mezi jednotlivými verzemi. Většina jmen soukromých modulů začíná prefixem „Host“ nebo „Std“, které jsou rezervovanými subsystémovými prefixy. Implementace může obsahovat další soukromé globální moduly jako **Kernel**, **Windows**,**Loader** nebo **Documents**.

Soukromé moduly nejsou v diagramu zakresleny. Některé z nich mohou být popsány v dokumentaci specifické pro hostitelský systém pro danou implementaci BlackBoxu, ale neměly by být považovány za vlastní součást BlackBoxu. Totéž platí pro moduly realizující rozhraní na hostitelskou platformu, které mohou být také k dispozici proto, aby byl možný přímý přístup k vlastnostem specifickým pro danou platformu.

Vývojář užívající BlackBox může používat svůj vlastní jedinečný prefix před jmény modulů, aby byla vyloučena případná záměna jmen a moduly jiných vývojářů. Firma Oberon Microsystems působí jako referenční středisko pro takové prefixy tím, že zajišťuje jejich registraci.

Úklid paměti - podmínka bezpečnosti

Rozšiřitelné programy nejsou nikdy ukončeny a tak žádný programátor nemůže vědět o celém programovém prostředí v němž jeho speciální softwarový modul bude případně pracovat. Softwarové komponenty mohou být velmi těsně spřaženy (integrovány) což znamená, že několik rozšíření může přistupovat ke stejné datové struktuře současně, pokud jsou tato data exportována. Žádný programátor proto nemůže s jistotou vědět, zdali se na jeho data odvolává také někdo jiný. Tím vzniká otázka, kdo a kdy má takovou datovou strukturu dealokovat.

V BlackBoxu tento problém vůbec neexistuje ponechává tuto problematiku důvěryhodným systémovým službám namísto toho, aby zatěžoval programátora explicitními dealokacemi. Tento uklízeč paměti se v podstatě snaží dokázat pro každý paměťový blok, že se na něj už nikdo neodvolává. Když se důkaz podaří, přenechá blok pro další použití. Úklid paměti není nějaká vlastnost objektově orientovaného přístupu. Avšak pro komponentní software je to nepostradatelná vlastnost.

Programovací konvence

Autoři Component Pascalu doporučují při programování dodržovat řadu konvencí, které umožňují snazší orientaci v systému a jeho vývoj. Nejdůležitější programovací konvence se soustřeďují na aspekty udržovatelnosti. Mělo by být co nejjednodušší změnit existující program bezpečným způsobem a to i když byl napsán před delší dobou a někým jiným. Údržba programu může být často zlepšena zvětšením počtu „úseků“ programu , které můžeme snadno lokalizovat. Pak může být snadněji rozpoznatelné které změny vyvolají změny v dalších částech programu.

Vstupní podmínky („precondition“) jsou jedním z nejužitečnějších nástrojů pro detekci neočekávaných rušivých jevů. Vstupní testy umožňují zaznamenat sémantické chyby co nejdříve, tj. co nejblíže k jejich zdroji. Po větších změnách v návrhu mohou dobře navržené testy tohoto typu („assertion“) výrazně snížit náklady na ladění a testování. Proto se doporučuje kdykoliv to možné, užívat statické prostředky pro vyjádření toho co se ví o návrhu programu. Typový a modulární systém Component Pascalu je vhodný k tomuto účelu. pomůže Nekonzistence tak pomůže najít kompilátor. Číslování invariantů by mělo být prováděno konsistentně se zbytkem v Component Pascalu.

Volné	0..	19	k volnému použití
Před-podmínky procedury	20..	59	testování parametrů při vstupu do
Post-podmínky procedury	60..	59	testování výsledků na konci
Invarianty lokálních chyb)	100..	120	testování mezistavů (detekce
Rezervy	121..	124	rezervy pro budoucí užití
Zastaralé		125	indikuje volání zastaralé procedury
Ještě neimplementováno		126	procedura není ještě implementována
Volání procedury rozhraní		127	volání neimplementované procedury

Ve zvoleném místě se pak zapíše například podmínka ASSERT (p#NIL,25) jako v dále uvedeném modulu JinTrap. Není-li při průchodu programem tato podmínka splněna, běh se zastaví a na obrazovce se objeví okno Trap obsahující výpis proměnných které je možno interaktivně zkontrolovat a odhalit příčinu chyby.

```
MODULE JinTrap;
VAR p: POINTER TO ARRAY OF INTEGER;
PROCEDURE Do*;
    BEGIN
        ASSERT( p#NIL, 25 );    (* p bude evidentně  NIL , takže naskočí
    TRAP25*)
    END Do;
END JinTrap.
```

```
TRAP 25  (precondition violated)
 JinTrap.Do   [00000013H]
 StdInterpreter.CallProc   [000003B5H]
  .a            BOOLEAN         FALSE
  .b            BOOLEAN         FALSE
  .c            BOOLEAN         FALSE
  .ok           BOOLEAN         TRUE
  .parType      INTEGER         0
 StdInterpreter.Command   [00000ECAH]
  .ptype        INTEGER         0
 StdInterpreter.Call   [00000FF1H]
  .ch           CHAR            0X
  .i            INTEGER         11
……..   atd.   … zde spousta dalších výpisů
```

Řada dalších doporučení, která zde ovšem neuvádíme, se týkají způsobu používání jednotlivých prvků jazyka. Jejich smyslem je sjednotit formát a usnadnit čtení textu programu programátorům , kteří program nevytvořili.

Závěr

Black Box Component Builder je unikátním, velmi zajímavým příkladem speciální kategorie vývojového nástroje, frameworku, umožňující vývoj aplikací se speciální programovou architekturou, umožňující přenést řadu vlastností frameworku i na vyvinutou aplikaci. Framework sám slouží jak k návrhu a vývoji, tak k provozování těchto aplikací. Že jde o technologii velmi účinnou dokazuje i realizace frmeworku JBed tvořeného vývojovým prostředím Denia a operačním systémem Portos , které byly vytvořeny pro tvorbu a monitorování vestavných aplikací typu Hard Real-Time [6]. I v případě, že BBCB framework nebude použit pro vývoj konkrétní aplikace, má smysl se s ním seznámit, protože poskytuje řadu inspirativních podnětů i pro vývoj aplikací založených na jiných nástrojích.

Literatura

1. E.Gamma, R.Helm, R.Johnson, J.Vlissides : "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994, ISBN 0-201-63361-2
2. Clemens Szyperski: "Component Software, Beyond Object-Oriented Programming" Addison-Wesley, 1997, ISBN 0-201-17888-5
3. Cuno Pfister: Component Software, A Case Study Using BlackBox Components" http://www.oberon.ch/docu/case_study/index.html
4. Hanspeter Mössenböck :"Object-Oriented Programming in Oberon-2", 2nd Edition Springer Verlag, New York, 1995, ISBN 3540600620
5. Martin Reiser, Niklaus Wirth : "Programming in Oberon, Steps beyond Pascal and Modula", Adison-Wesley, Wokingham England, 1992, ISBN 0-201-56543-9
6. http://www.jbed.com

*Tento příspěvek vznikl v rámci práce na grantovém projektu **Vývoj programových a technických prostředků pro přenositelné a bezpečné aplikace řízení procesů** Grantové agentury české republiky registrovaném pod číslem **102/97/0542***