Zpravodaj Československého sdružení uživatelů TeXu

Karel Skoupý

NTS: nový sázecí systém

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 9 (1999), No. 3, 115–131

Persistent URL: http://dml.cz/dmlcz/149843

discussion sometimes pays off. Being an extension not related to output (DVI, POSTSCRIPT, PDF) or specific typesetting, it proves that extensions like that can be hooked into a typesetting system without disturbing and breaking existing code.

One may wonder if $\mathcal{NTS}$ will make *those other TEX's* obsolete. Given that it takes time to come up with real new things, we can be sure that the predecessors of $\mathcal{NTS}$ will be around for a long time, if only because they have virtually no bugs, are supported by macro packages and most of all do their job well. This gives the $\mathcal{NTS}$ developers the time needed to come up with real good concepts and implementations.

Given that the TEX community has demonstrated that extending a stable program is feasible without harming existing, often critical, typographic production processes, $\mathcal{NTS}$ has a great future, although, in many areas, we haven't yet reached the limits of traditional TEX. It takes time and discussion, but talking of life-long tools, we have some time left. It is up to the $\mathcal{NTS}$ team to stimulate and guard this process, and we hope we will not fail you.

It sounds like I'm believing it myself, doesn't it?

Hans Hagen
PRAGMA Advanced Document Engineering
Ridderstraat 27
8061GH Hasselt
The Netherlands
E-Mail: pragma@wxs.nl

$\mathcal{NTS}$: nový sázecí systém

KAREL SKOUPÝ

Počátky projektu

Diskuse o potřebě dalšího vývoje TEXu či vytvoření jeho nástupce se rozvinula již počátkem devadesátých let. Vedly k tomu důvody technické i politické. Politické důvody by se daly velice stručně vyjádřit jako potřeba udržení a zvýšení atraktivity TEXu pro zachování a rozvoj komunity uživatelů. Více informací o tom lze najít zejména v [8].

Technické důvody spočívají především v požadavcích na ještě vyšší kvalitu a rozsáhlejší možnosti zpracování dokumentu. Týká se to například sazby ve

více sloupcích, sazby na rejstřík, optimalizace stránkového zlomu a mnoha dalších problémů, viz. [4]. Zároveň by byla jistě vítána možnost jednoduššího začlenění sázecího systému v rámci jiné aplikace, sdílení jen některých částí (sázecího stroje) jinými systémy a konečně podpory dalších vstupních i výstupních formátů.

Donald E. Knuth, autor TEXu, vyjádřil podporu snahám o vytvoření nového systému, který by byl nástupcem TEXu, pro něho samotného však práce na sázecím systému skončila. Zavázal se pouze odstraňovat případné chyby, ale TEX je již zakonzervován v jeho současné podobě. Vyhradil si také právo na používání názvu TEX, nový sázecí systém se tedy musí jmenovat jinak.

Během diskusí se uvažovalo o několika možnostech dalšího vývoje. Od nejkonservativnější – ponechání TEXu tak, jak je – po nejradikálnější – vytvoření zcela nového sázecího systému pro další století. Projekt začal relativně konservativní cestou, jejímž výsledkem je ε-TEX. Tento systém je vlastně modifikací původních zdrojových textů TEXu, která přidává velmi užitečné primitivy a ruší některá omezení. Práce je prováděna ve stejném programovacím jazyce (Pascal-`WEB`) a je zachována přísná kompatibilita s TEXem.

Již na počátku se však uvažovalo i o radikálnější variantě. Tou se mělo stát úplné přeprogramování TEXu (pomocí jiného programovacího jazyka) tak, aby byl nový systém funkčně kompatibilní s TEXem, jeho architektura aby však byla maximálně otevřená a přístupná novým změnám a rozšířením. Tento náročnější projekt byl ale odložen na dobu, kdy budou k dispozici potřebné finance. Situace se změnila v roce 1997, kdy německé sdružení uživatelů TEXu DANTE e.V. rozhodlo o sponzorování projektu a na úkol mohl být najat programátor na plný úvazek (autor článku).

Základní požadavky

Jak již bylo uvedeno, jedním ze základních požadavků na nový systém je kompatibilita s TEXem. Dosažení tohoto cíle není snadné, má však svoje dobré důvody. Málokdo z uživatelů TEXu používá holý systém, většina využívá formátů a balíků maker, jako je například LATEX. Nový sázecí systém, který by nebyl podporován těmito nadstavbami, by byl těžko akceptovatelný, i kdyby poskytoval mnoho nových možností na úrovni primitiv. Další výhodou kompatibility je možnost testování systému pomocí existujících vstupních souborů. Existuje jich obrovské množství. Důležitým kritériem kompatibility bude absolvování `TRIP` testu (`CTAN:systems/knuth/tex/trip.tex`). Nový systém by jím měl úspěšně projít (až na drobné, dobře zdůvodněné odlišnosti). V neposlední řadě kompatibilita prokáže, že nový systém toho umí alespoň tolik, co starý TEX. V budoucích verších systému se však může stát, že kompatibilita bude bránit dalším rozšířením a v takovém případě bude lepší ji opustit.

Významnějším požadavkem je otevřená a modulární struktura nového systému. Zkušenost totiž ukázala, že modifikace stávajícího programového kódu TEXu je velice obtížná. Jedná se o monolitický program se spoustou globálních proměnných, které jsou přístupné odkudkoliv. Řada datových struktur je optimalizována pro co největší úsporu paměti, řada algoritmů zase pro maximální rychlost, někdy až nestrukturovaným způsobem. Vzhledem k výkonům tehdejších počítačů a k dostupným technologiím to všechno mělo v době vývoje TEXu jistě svůj dobrý důvod, vede to ale nejen k horší čitelnosti, zejména však k nepředvídaným následkům při změnách programu.

Na druhé straně je celý program velmi dobře dokumentován. Knuth pro účely dokumentace vyvinul svůj vlastní nástroj `WEB`, ten podporuje také definice a použití maker a umožňuje změnu pořadí úseků kódu pro zvýšení přehlednosti. Například kód, který nějak manipuluje s určitou datovou strukturou je typicky seskupen poblíž její definice – lze v tom spatřovat významný krok směrem k objektově orientovanému programování.

Nový systém by se měl tedy skládat ze samostatných modulů, které mají jasně definované rozhraní a jsou na sobě co nejméně závislé. Objektově orientované programování se zdá být pro tento účel nejvhodnějším.

Programovací jazyk

Prvním úkolem pracovní skupiny v rámci projektu re-implementace byl výběr programovacího jazyka. V minulosti se uvažovalo o jazycích vyšší úrovně pro rychlou tvorbu prototypu jako jsou PROLOG nebo LISP. Nakonec se ale rozhodování přiklonilo k objektově orientovaným jazykům a výběr se zúžil na tři kandidáty: Common Lisp Object System (CLOS), C++ a Java. Ačkoliv má každý z nich svoje specifické výhody oproti ostatním, byl po pečlivém výběru vybrán jazyk Java. Podrobnějšímu zdůvodnění této volby je věnován článek [19].

Uveďme některé charakteristiky jazyka Java. Jako nejmladší z výše uvedených jazyků se od nich mohl v mnohém poučit. Velmi zdařilým způsobem implementuje třídy, objekty a rozhraní. Třídy jsou překládány do takzvaného byte-kódu a mohou být dynamicky kombinovány v době běhu. Velmi příjemný je zabudovaný čistič nevyužité paměti (garbagge collector), který vylučuje chyby související s porušením paměti. Typy a jejich kontrola mohou také zachytit množství chyb od triviálních až po závažné. Zpracování výjimek vyžaduje přesné deklarace, které zabraňují opomenutí hraničních situací. Java je dokonale přenositelná a to i na úrovni přeloženého byte-kódu. Díky dnes již standardně poskytované možnosti překladu do nativního kódu počítače při spuštění aplikace je také efektivita téměř srovnatelná s aplikacemi psanými v C++. Pevnou součástí Javy jsou též balíky podporující sítě, grafiku, grafické uživatelské rozhraní a mnoho dalších.

Za velmi důležité považujeme spojení Javy s Internetem. Lze si představit, že bude možno (automaticky) natáhnout nové zásuvné $\mathcal{NTS}$ moduly, sdílet prostřednictvím sítě fonty a vstupní soubory nebo si systém interaktivně přizpůsobit svým požadavkům.

Implementace

Následující část je věnována popisu implementace současné rozpracované verse systému. Nejprve krátce vysvětlíme pojmy specifické pro objektově orientované programování a jazyk Java.

Objekt je programová entita, která sdružuje data a algoritmy pro manipulaci s nimi. Objekt má určitý stav, který je dán hodnotou jeho datových atributů a funkce, které umožňují tento stav zjišťovat a měnit, nebo ho využívat k jiným výpočtům.

Metoda je funkce sdružená s objektem (nebo třídou). Množina těchto funkcí příslušná k určitému objektu definuje jeho chování.

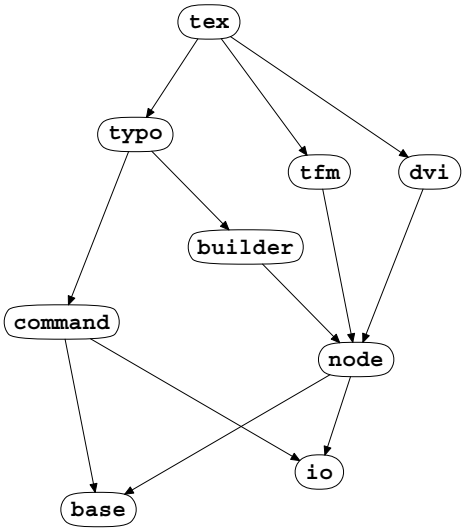
Třída (class) je abstraktní typ objektu. Obsahuje deklarace datových atributů objektu a definice jeho metod. Lze na ni pohlížet také jako na šablonu, podle které se vytváří objekty – její instance. Z již existujících tříd je možno odvozovat nové třídy. Odvozená třída dědí všechny atributy i metody své nadtřídy, může k nim přidávat vlastní atributy a metody nebo existující metody předefinovávat. Atributy a metody deklarované v rámci třídy mohou mít specifikována přístupová práva a to: pouze v rámci dané třídy, pro svoje potomky, v rámci balíku a jako veřejně přístupná. Veřejné metody a atributy představují rozhraní příslušného objektu pro okolní svět. Pokud je metoda (nebo atribut) deklarována jako statická, neváže se ke konkrétní instanci (objektu) ale k celé třídě.

Abstraktní třída je třída, která nevytváří žádné instance. Některé její metody mohou být abstraktní – mít definován počet a typy parametrů ale nedefinované tělo. Typicky je základem hierarchie podtříd, které předefinovávají její abstraktní metody.

Rozhraní (interface) je abstraktní typ objektu, který definuje pouze jeho vnější chování, ne však implementaci. Třída může deklarovat, která rozhraní implementuje (splňuje).

Balík (package) je soubor tříd a rozhraní, která spolu nějak souvisí. Lze ho chápat jako knihovnu určenou k distribuci. Slouží k jasnější organizaci programu a identifikaci jeho částí.

Polymorfismus je princip, který umožňuje objektům stejného typu odlišné (specifické) chování. Objekty, které mají stejné rozhraní, mohou být totiž různě implementovány.



Obrázek 1: Závislosti balíků v $\mathcal{NTS}$

Další popis je rozdělen podle balíků. Pro každý je uvedena stručná charakteristika a popis nejdůležitějších tříd a rozhraní.

Balík `base`

Hlavním posláním tohoto balíku je definice základních datových typů, které jsou používány zbytkem systému. Jedná se o minimální prvek hierarchie balíků $\mathcal{NTS}$, to znamená, že žádná třída zde nezávisí na jiných třídách z ostatních $\mathcal{NTS}$ balíků.

Nejdůležitějšími třídami zde jsou:

`Dimen` představuje rozměr měřený v tiskařských bodech (nebo `mu` jednotkách). Skutečnost, že interní representace je stejná jako u rozměrů v TEXu, zaručuje potřebnou kompatibilitu. Veřejné rozhraní této třídy se snaží být zcela nezávislé na její interní representaci. Vnějšímu světu se `Dimen` jeví jako zlomek (v tiskařských bodech). Definuje metody pro převody z celých čísel, zlomků daných čitatelem a jmenovatelem, reálných čísel a naopak. Poskytuje také základní aritmetické operace. V případě binárních operací jsou podporovány i jejich verse pro kombinace s ostatními převoditelnými číselnými typy. Obecné veřejné rozhraní dovoluje zásadní změnu rozsahu či dokonce vnitřní jednotky representace bez jakýchkoliv důsledků pro ostatní kód.

`Glue` odráží další, z TEXu dobře známý typ. Má svůj přirozený rozměr a velikost a řád roztažitelnosti a stlačitelnosti. Poskytuje aritmetické metody jako sčítání dvou `Glue`, násobení jednoduchým číslem a také varianty pro ostatní převoditelné typy.

`Num` representuje celé číslo. Je to normální integer, je ale zabalený do objektu a může být tedy přímo uložený v tabulce ekvivalentů a lze jej rozlišit od běžných integerů v kódu. Slouží zejména pro vyjádření hodnot číselných registrů (`\count`) (a je jaksi symetrická k `Dimen` a `Glue` pro registry `\dimen` a `\skip`).

Všechny výše uvedené základní třídy poskytují též metody pro získání jejich vyjádření řetězcem znaků, které může být zobrazeno na obrazovce, v log souboru nebo použito primitivou `\the`.

`LevelEqTable` je poslední důležitou a poměrně složitou třídou. Používá se k implementaci tabulky ekvivalentů a hašovací tabulky v TEXu. Zatímco TEX používá asociativní hašovací tabulku pouze pro významy řídících sekvencí, $\mathcal{NTS}$ ukládá mnohem více druhů ekvivalentů asociativním způsobem.

Jakýkoliv objekt může být spojen s jistou kombinací *druhu (kind)* a *klíče (key)*. Pro různé typy ekvivalentů jsou definovány různé druhy: jeden pro významy řídících sekvencí, jiný pro každou třídu registrů, další pro nastavení `\catcode`, atd. Klíč je (v závislosti na druhu) buď objekt (například jméno řídící sekvence) nebo číslo (pro většinu ostatních). Asociativní způsob ukládání hodnot registrů přirozeným způsobem zabraňuje omezení na jakýkoliv určitý počet registrů. Ačkoliv $\mathcal{NTS}$ je s TEXem kompatibilní v tom, že poskytuje pouze 256 registrů každého typu, toto omezení je zavedeno uměle a může být v budoucnu snadno odstraněno.

Jak už samo jméno napovídá, `LevelEqTable` také obhospodařuje zvyšování a snižování úrovní zanoření vyvolané skupinami ve vstupním jazyce a také uschovávání a obnovování asociovaných hodnot.

Ačkoliv registry byly přesunuty ze statických polí do asociativní tabulky, zůstává zde stále ještě jeden druh hodnot, který sice není asociativního typu, podléhá ale uschovávání a obnovování. Jedná se o parametry (jako je `\tolerance`, `\hsize`, ...). Jejich současná hodnota je uložena na jednom konkrétním místě. `LevelEqTable` poskytuje rozhraní i pro tyto externí ekvivalenty a spravuje jejich úschovu a obnovu.

Balík `io`

Tento balík obsahuje třídy a rozhraní potřebné pro čtení znaků ze vstupních souborů a pro zápis do log souborů. Oba tyto soubory mohou představovat i uživatelův terminál. Balík je nezávislý na ostatních balících stejně jako `base`.

`CharCode` je rozhraní a je velmi zajímavé. Vedli jsme v $\mathcal{NTS}$ týmu diskuse o tom, zda mají být kódy znaků representovány některým základním

typem jazyka Java. Padlo rozhodnutí, že je potřeba použít třídu, aby ji bylo možné odlišit od ostatních základních typů. Později (během vývoje systému) se ukázalo, že nejlépe svůj účel splňuje ještě abstraktnější vyjádření pomocí rozhraní. Toto rozhraní deklaruje metody pro získání odpovídající znakové nebo číselné hodnoty, srovnání s jiným `CharCode`, znakem nebo číslem na shodu, vytvoření odpovídající varianty `CharCode` pro malá a velká písmena, zápis na znakově orientovaný výstup a několik predikátů. Většina metod je zde proto, že jazyk TEXu používá znaky velice často nejen pro sazbu, ale také jako číselné hodnoty nebo součásti klíčových slov. Navíc mají některé znaky vliv na vstupní načítání a výstup do log souboru (`\endlinechar`, `\escapechar`, `\newlinechar`).

V současné době používá $\mathcal{NTS}$ pro implementaci rozhraní `CharCode` pouze objekt obsahující obyčejný znak. Existuje ale možnost použití zcela rozdílných representací (např. pojmenované znaky) bez jakékoliv změny programového kódu $\mathcal{NTS}$. Takové objekty mohou proplouvat celým systémem za předpokladu, že je na konci očekává výstupní objekt, který je rozpozná a správně použije. V nějaké budoucí aplikaci může dokonce i několik nezávislých implementací `CharCode` existovat současně.

`Name` se má k `CharCode` stejně jako se má `String` k `char`. Používá se k representaci jmen řídících sekvencí, jména úlohy a jmen fontů a souborů načtených ze vstupu.

`InputLine` odpovídá jednomu řádku ze vstupního souboru nebo uživatelova terminálu. Má metody pro získání dalšího `CharCode` nebo pro pouhé nahlédnutí jaký je příští kód beze změny současné čtecí posice. Interpretuje rozšířené znakové kódy (jako `^^M`), ignoruje koncové mezery a připojuje `\endlinechar`, je-li to potřeba. Další třída, `LineInput`, slouží jako vstupní posloupnost řádků `InputLine` ze souboru nebo terminálu.

`Log` je důležité rozhraní pro tisk informací do log souboru nebo na obrazovku terminálu. Deklaruje metody pro tisk hodnot základních typů a typů `String`, `CharCode` a `Loggable` (viz. níže). Je zde deklarováno i několik metod pro řízení výstupního zalámání řádků. Třída `StandardLog` implementuje rozhraní `Log` standardním TEXovským způsobem.

`Loggable` je velmi jednoduché rozhraní deklarující jednu metodu pro tisk do objektu typu `Log`. To se velice hodí, protože většina důležitých tříd v $\mathcal{NTS}$ toto rozhraní implementuje, a proto je jejich tisk velmi pohodlný.

Balík `command`

Třídy v balíku `command` tvoří interpret vstupního jazyka TEXu. Ačkoliv se jedná o rozsáhlejší balík, nemá zatím nic společného se sazbou. Ve skutečnosti nejméně jedna třetina zdrojového kódu TEXu nemá vůbec co dělat se sázením. Zdejší třídy jsou zodpovědné za proces načítání vstupních symbolů, jejich expanse a za

většinu zpracování nezávislého na sázecím módu, jako jsou definice maker a přiřazení hodnot registrům.

`Token` je abstraktní třída odpovídající vstupnímu symbolu. Deklaruje metody pro získání významu symbolu, přiřazení nového významu (pokud je to dovoleno), testování shody s jiným symbolem a množství predikátů, které říkají, zda lze předefinovat význam, zda se jedná o složenou závorku, písmeno a podobně.

K disposici je několik druhů symbolů a tvoří malou hierarchii podtříd. Typickými příklady jsou `CtrlSeqToken`, `ActiveCharToken`, `SpaceToken`, `LetterToken`, `LeftBraceToken`, ...

`Tokenizer` je schopen poskytnout nějakou posloupnost symbolů. Existují různé podtřídy jako například: `LineInputTokenizer` pro načítání symbolů ze vstupního souboru, `Macro.Expansion` pro tělo makra se specifikovanými makro parametry, `InsertedTokenList` pro seznam symbolů vložených na vstup z registru příslušného typu nebo `BackedToken` pro právě jeden zpátky vložený symbol. Objekty typu `Tokenizer` jsou vkládány do zásobníku `TokenizerStack`, který je analogií vstupního zásobníku TEXu.

`Command` je abstraktní třída representující každičký TEXovský příkaz. Příkazy jsou většinou primitivy TEXu registrované pod svým jménem v tabulce ekvivalentů, existují ale významné výjimky jako `Macro` nebo význam běžného znaku. V TEXu jsou symboly a příkazy representovány jedním typem a často jsou interpretovány oběma způsoby, což může být matoucí. $\mathcal{NTS}$ oba koncepty přísně odděluje. Jak bylo naznačeno výše, `Token` je symbolem načteným ze vstupu, který může mít určitý význam. Typem tohoto významu je právě příkaz `Command`, který je popisován zde.

`Command` má metody pro vykonání a expansi. Pouze některé příkazy mohou být expandovány a tato vlastnost je zjišťována pomocí další metody s pravdivostní hodnotou. Srdcem $\mathcal{NTS}$ je cyklus velmi podobný TEXovskému `main_control`. V rámci jednoho kroku je získán symbol ze vstupu a zkoumá se jeho význam. Je-li příslušný příkaz expandovatelný, volá se odpovídající metoda pro expansi. Vede-li expanse k neprázdnému výsledku, je metoda zodpovědná za jeho vložení do vstupního zásobníku. Není-li příkaz expandovatelný, volá se metoda pro jeho vykonání.

Ještě jedna zajímavost o expandovatelných příkazech. Pokud je jejich expansi zabráněno pomocí `\noexpand`, jsou vykonávány. V tom případě se chovají přesně jako `\relax` (nedělají nic kromě obnovení vnitřního stavu TEXu a zastavení dopředného hledání) a dokonce předstírají, že jsou `\relax`, pokud jsou zkoumány pomocí `\show`. Z tohoto důvodu je celý podstrom expandovatelných příkazů odvozen z příkazu `\relax`.

Další důležitou součástí rozhraní třídy `Command` jsou metody pro získání hodnoty určitého typu a indikaci, zda je příkaz schopen takovou hodnotu

poskytnout. Je to užitečné, pokud se příkaz vyskytne například na pravé straně přiřazení. Proto registry, parametry a několik dalších příkazů umí poskytnout hodnoty typu číslo, rozměr, výplň (glue) nebo seznam symbolů. Obrázek 2 ilustruje hierarchii tříd pro příkazy uvedením některých příkladů. V době psaní tohoto článku bylo pro příkazy implementováno více než 150 tříd (společně s příkazy pro sazbu).

`CommandBase` je nadtřídou třídy `Command`. Definuje pouze statické metody, které souvisí s načítáním různých vstupních prvků (jako čísla, rozměry, jména souborů, klíčová slova, ...), udržováním tabulky ekvivalentů, vstupního zásobníku a několika instancí výstupních objektů typu `Log`. Jak uvidíme později, kromě `Command` existuje více objektů, které využívají tyto služby, a proto jsou pro větší komfort odvozeny z této abstraktní třídy.

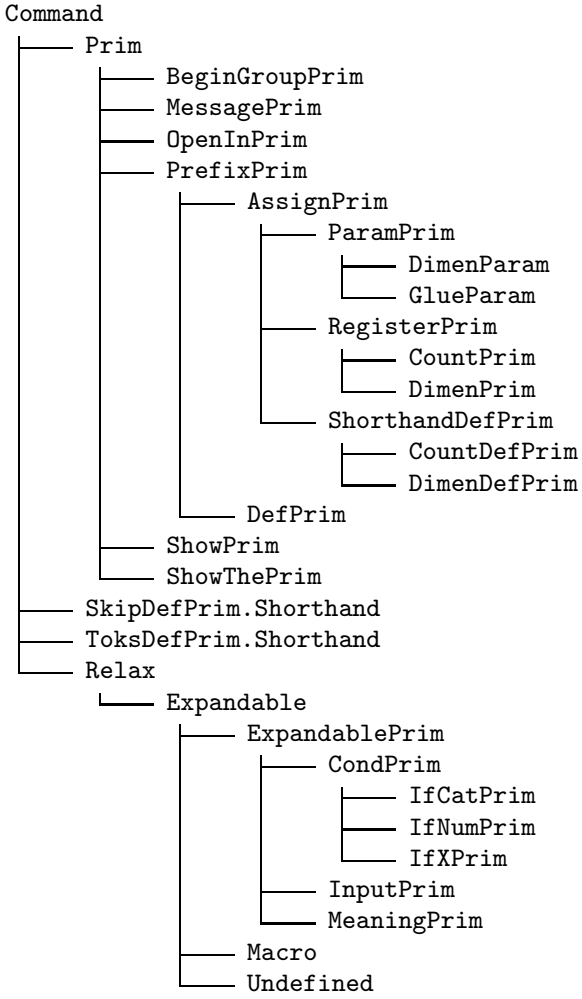
Balík `node`

A nyní se konečně dostáváme k sazbě! Třídy v tomto balíku representují sázecí materiál. Jsou zde také obecná rozhraní pro metriky fontů a generátory výstupních formátů. Balík je relativně nízko v hierarchii, třídy jsou závislé pouze na balících `base` a `io`. To jim dává dobrou šanci k samostatnému použití v jiném sázecím systému, který může používat zcela odlišný vstupní jazyk nebo uživatelské rozhraní.

`Node` je rozhraní představující uzel – základní stavební kámen sázecího materiálu. Má metody pro získání svých rozměrů (dokonce i když jsou ovlivněny roztažením nebo stlačením), k popisu sama sebe v log souboru a ke svému vysázení. Existuje celá hierarchie tříd implementujících rozhraní `Node`. Některé jsou jednoduché, například: `RuleNode`, `HKernNode`, `VKernNode`, `HSkipNode`, `VSkipNode`, `PenaltyNode`; jiné objekty jsou složené a mohou obsahovat seznamy podřízených uzlů: `HBoxNode`, `VBoxNode`.

`Packer` se používá, když chceme vypočítat rozměry složených boxů, které jsou vytvořeny ze seznamů uzlů. Tento proces se v TEXu nazývá pakování. Algoritmus je v podstatě stejný jak pro vodorovné, tak svislé seznamy uzlů, stačí jen prohodit vertikální a horizontální rozměry. Abstraktní třída `Packer` definuje abstraktní versi algoritmu a poskytuje prázdné virtuální metody pro získání odpovídajících velikostí uzlů. Potom jsou zde specialisované podtřídy pro horizontální a vertikální boxy, ze kterých jsou pak ještě mimo tento balík odvozeny další podtřídy, které ohlašují ta správná varování, pokud se něco nepovede v rámci tolerancí.

`FontMetric` je abstraktní rozhraní pro objekty poskytující informace o metrice fontu. V současné době jsou k disposici pouze známé `tfm` soubory, to ale není omezení pro $\mathcal{NTS}$. Ten je připraven pro jakýkoliv typ metrik, který lze přizpůsobit tomuto rozhraní. Poskytuje metody pro získání identifikace a různých číselných a rozměrových parametrů, aby byla dodržena



Obrázek 2: Část hierarchie tříd v balíku `command`

kompatibilita s TEXem. Ale především jsou zde metody, které umožňují získat instanci `Node` pro určitý `CharCode`, normální mezeru mezi slovy a speciální objekt, který umí produkovat representaci znaků, ligatur a kerningů pro danou posloupnost znakových kódů.

`TypeSetter` má podobné charakteristiky jako `FontMetric`. Definuje obecné rozhraní pro generátor výstupního formátu. Poskytuje metody pro vysázení znaku nebo čáry na současnou posici a posun této posice.

Balík `builder`

Tento balík se stará o oblasti, které souvisí s horizontálními, vertikálními a matematickým módy TEXu. Zatímco TEX má pouze jednu celočíselnou proměnnou, která odpovídá jednomu ze sedmi možných módů (tři výše uvedené jsou buď interní nebo externí a jeden prázdný „no mode“) na vrcholu sémantického zásobníku, $\mathcal{NTS}$ používá objekty, které budují sázecí materiál pro jednotlivé módy.

Tento balík je ve srovnání s balíkem `node` již více závislý na způsobu, jakým pracuje TEX, je ale stále nezávislý na vstupním jazyce. Je poměrně malý a jednoduchý. Určitá míra složitosti musí být vyřešena pro součinnost sázecích příkazů s jednotlivými módy, tuto problematiku ale řeší balík `typo`.

`Builder` je kořenem hierarchie tříd pro jednotlivé módy. Deklaruje některé predikátové metody pro získání jistých charakteristik daného módu a metody pro přidávání uzlu, kernu nebo skoku na konec seznamu, který je právě rozpracován. Vytváří příslušnou versi (horizontální nebo vertikální) kernů a skoků a je-li třeba, provádí další přizpůsobení. V současné době jsou podporovány pouze módy známé z TEXu, do budoucna je ale možné uvažovat o dalších módech (chemický, obrázkový, ...).

Balík `typo`

Balík `typo` je nadstavbou balíku `command`. Obsahuje všechny podtřídy třídy `Command`, které pracují s dosud implementovanou sazbou (přibude ještě balík `maths` pro příkazy matematické sazby). Využívá také balíky `builder` a `node`.

`TypoCommand` je podobný třídě `CommandBase` ale slouží sázecím příkazům. Je to meziúrovňová abstraktní třída, jež definuje několik užitečných statických metod. Udržuje zásobník módů (`Builder`) a současnou metriku fontu. Poskytuje metody pro načítání specifikace fontu nebo boxu ze vstupního souboru a přidání znaku nebo mezery do právě budovaného seznamu.

Mnoho tříd z tohoto balíku je odvozeno přímo ze tříd balíku `command`, protože od nich mohou zdědit užitečné chování. Nemohou však být obsaženy v balíku `command`, neboť vyžadují některé informace přístupné pouze v balíku `typo` (obvykle voláním některé statické metody třídy `TypoCommand`). V zásadě jsou dvojího druhu: jedním jsou `\if` primitivy jako `ifhmode` nebo

`ifvbox`, které potřebují jenom nějakou informaci o současném módu nebo určitém box registru; druhým případem jsou sázecí příkazy, které jsou nezávislé na módu, například `\setbox`, `\wd` a `\chardef`.

`BuilderCommand` je abstraktní nadtřídou příkazů, které jsou závislé na módu. V otevřeném systému, jako je $\mathcal{NTS}$, požadujeme, aby mohly být nové funkce snadno přidávány. Například abstraktní třída `Command` definuje skupinu metod, které mohou nové příkazy implementovat jakýmkoliv rozumným způsobem. Tento druh polymorfismu je podporován přímo zvoleným programovacím jazykem.

Ale co uděláme, pokud budeme chtít v budoucnu naprogramováním příslušné podtřídy `Builder` přidat další mód, který ale nabízí nové funkce dosud nedeklarované v rozhraní `Builder` a nějaké specialisované příkazy, které tyto nové funkce umí využít? Nebudeme chtít rozšířit základní rozhraní (alespoň ne do stávající verse systému) nebo dokonce nebudeme moci (pokud budeme programovat zásuvný modul). Jedinou možností bude zjistit aktuální typ současného `Builder` a použít nechvalně známý operátor přetypování, pokud se bude jednat o náš vylepšený mód.

Je zde ale další problém s existujícími příkazy, závislými na módu. Jak se mají chovat v novém módu? Pro tento účel třída `BuilderCommand` udržuje hašovací tabulku, která přidružuje objekt typu `Action` ke každé kombinaci třídy `Builder` a objektu typu `BuilderCommand`. Přiřazení je realizováno na úrovni konfigurace $\mathcal{NTS}$ a automaticky zohledňuje hierarchii tříd `Builder`. Díky takové pružnosti je velmi jednoduché předepsat chování příkazů v jednotlivých módech pro pozměněný systém.

Třída `Action` je podtřídou `CommandBase` a dědí tedy metody pro načítání vstupu, zacházení s log soubory a chybová hlášení. Podtřídy `Action` jsou obvykle implementovány jako vnitřní třídy příslušné podtřídy `BuilderCommand`. Takto vypadá úryvek z $\mathcal{NTS}$ konfigurace:

```
RulePrim    hrule = new RulePrim("hrule",
                                 default_rule, Dimen.NULL,
                                 Dimen.ZERO, Dimen.ZERO);

RulePrim    vrule = new RulePrim("vrule",
                                 Dimen.NULL, default_rule,
                                 Dimen.NULL, Dimen.ZERO);

hrule.defineAction(VertBuilder.class, hrule.NORMAL);
hrule.defineAction(ParBuilder.class, hrule.FINISH_PAR);
hrule.defineAction(HBoxBuilder.class, hrule.BAD_HRULE);

vrule.defineAction(HorizBuilder.class, vrule.NORMAL);
vrule.defineAction(VertBuilder.class, vrule.START_PAR);
```

Objekt typu `BuilderCommand` odpovídající TEXovské primitivě `\hrule` definuje tři akce: provádí normální operaci ve vertikálním módu, dokončí současný odstavec (pokud nějaký začal) v horizontálním módu a postěžuje si uvnitř horizontálního boxu. `\vrule` se zpracovává normálně v jakémkoliv horizontálním módu a započne nový odstavec ve vertikálním módu. Ve skutečnosti existuje pouze jedna třída (`RulePrim`), která má dvě instance pojmenované `hrule` a `vrule` s rozdílnými parametry. Jsou jim přiřazeny různé `Action` pro stejné módy. Všechny objekty typu `Action`: `NORMAL`, `START_PAR`, `FINISH_PAR` a `BAD_HRULE` jsou instancemi vnitřních tříd uvnitř `RulePrim` nebo jejích nadtříd.

Dalšími příklady `BuilderCommand` jsou: `HBoxPrim`, `VBoxPrim`, `VTopPrim`, `LowerPrim`, `MoveLeftPrim`, `BoxPrim`, `KernPrim`, `CharPrim`, `ExSpacePrim`, `AccentPrim`, `AnySkipPrim`.

`Group` je další podtřídou třídy `CommandBase`. Její podtřídy pokrývají různé typy skupin v TEXu. Existují skupiny jako `SimpleGroup` pro pár složených závorek, `SemiSimpleGroup` pro `\begingroup` a `\endgroup`, `HBoxGroup`, `VBoxGroup` nebo `VTopGroup`. Sama třída `Group` je definována uvnitř třídy `CommandBase` a tam je také udržován zásobník otevřených skupin. Většina jejích podtříd ale patří do balíku `typo`.

Třídy `Group` mají s třídami `Builder` jeden společný problém. Příkazy pro uzavírání skupin se chovají rozdílně v kombinaci s daným typem skupiny. Pravá složená závorka nemůže párovat `\begingroup` a `\endgroup` nemůže párovat levou složenou závorku. Problém je řešen úplně stejným způsobem jako pro kombinace příkazů a módů.

Balík `tfm`

Balík `tfm` implementuje pro $\mathcal{NTS}$ typ metriky fontu používaný v TEXu (`tfm`). Může sloužit jako příklad pro implementaci ostatních typů metrik.

`TeXFm` je třída representující nízkoúrovňový, téměř syrový, formát TEXovského souboru fontové metriky. Některé komplikace jsou odstíněny, ale veřejné rozhraní odráží pouze informace dostupné ze souboru. Používá několik pomocných tříd, `tfm` formát je totiž natolik složitý, že by bylo nepraktické ho podchytit v jednom programovém souboru. Například celý proces načítání `tfm` souboru je realizován třídou `TeXFmLoader`, která vytváří instanci třídy `TeXFm`. Samotná `TeXFm` má metody pro získání informací o jednotlivých znacích, ligaturách a kernincích pro dvojice znaků, receptů pro roztažitelné znaky a posloupností rostoucích znaků. Další metoda je k disposici pro tisk vlastní representace ve formě seznamu vlastností. Toho je využito v malé Java aplikaci `tftopl`, která díky třídě `TeXFm` sdílí většinu kódu s $\mathcal{NTS}$.

`TeXFontMetric` je adaptací `TeXFm`, která implementuje rozhraní `FontMetric` z balíku `node`. Je to vlastně obal, který využívá přirozené metody třídy `TeXFm` a definuje metody vyžadované zbytkem $\mathcal{NTS}$. Tento přístup je pravděpodobně užitečný pro budoucí implementace dalších typů metrik. Rozhraní jsou čistší a kromě toho nemůžeme očekávat, že někdo nezúčastněný poskytne rozhraní vyhovující přesně našim požadavkům, a to ani v případě, že poskytne pro přístup k metrice přímo třídu jazyka Java.

Balík `dvi`

Balík `dvi` implementuje výstupní formát `dvi` jako jeden z možných (budoucích) výstupních formátů $\mathcal{NTS}$. V mnohém se podobá balíku `tfm`.

`DviFormatWriter` představuje nižší vrstvu, která přesně pokrývá možnosti formátu `dvi`. Definuje metody pro výstup jednoho znaku nebo černého obdélníku, posun současné posice na stránce, začátek a konec stránky a pro definice a přepínání fontů. Tyto metody si hlídají konsistenci zapisovaných dat, není například možné vysázet znak nebo ukončit stránku, pokud stránka nezačala, dovolí ale nadefinovat font. Kromě toho si ve vlastní režii provádí optimalizace posunu současné posice.

`DviTypeSetter` je adaptací `DviFormatWriter` na rozhraní `TypeSetter` z balíku `node`. Řešení je analogické jako v případě tříd `TeXFm` a `TeXFontMetric` popsaných výše.

Balík `tex`

Tento balík zastřešuje ostatní $\mathcal{NTS}$ balíky a jedná se o zdaleka nejchaotičtější část systému. Doposud všechny třídy a balíky byly navrženy se záměrem poskytnout čisté a elegantní rozhraní a být co nejméně závislé na ostatních třídách a balících. Ale v samotném TEXu je spousta nejasných závislostí. To bylo také jedním z hlavních důvodů, proč projekt $\mathcal{NTS}$ vůbec vznikl. Třídy v tomto balíku navzájem spojují všechny dosud nezávislé jednotky a navíc sem byly přesunuty problematické případy, které nezapadaly do čistého návrhu ostatních balíků, pokud to bylo možné. To je hlavní důvod, proč zdrojové texty tohoto balíku někdy působí chaotickým dojmem.

Kromě toho jsou zde ještě třídy pro správu seznamu chybových hlášení a příkazy tak nejsou závislé na konkrétním způsobu, jakým jsou chyby hlášeny.

Nejzajímavější částí balíku je třída `Primitives`, která obsahuje konfiguraci celého systému. V popisu balíku `typo` se už vyskytovala jedna ukázka.

Modularita a konfigurovatelnost

Vybudování systému, který je co nejvíce modulární, bylo jedním z hlavních cílů projektu. V současné implementaci TEXu je množství závislostí. Zkušenost ukázala, že složitější změny jsou velmi obtížné a nebezpečné, protože mohou vést k mnoha nejasným a nezamýšleným důsledkům. V $\mathcal{NTS}$ bylo rozhodnuto učinit všechny závislosti zjevnými a čistými. Všechny třídy mají dobře definované rozhraní svých veřejných metod a to je využíváno pro veškerou komunikaci s příslušnými objekty. Nejsou zde žádné nekontrolovatelné změny globálních proměnných. Tento programovací přístup je významně podporován objektově orientovaným jazykem Java.

Dalším motivem pro vytváření nezávislých programových jednotek je možnost záměny některých modulů za moduly se stejným rozhraním ale rozdílnou implementací. Nezávislé třídy a balíky mohou být také použity coby stavební kameny jiného systému. Balíky $\mathcal{NTS}$ jsou tedy navrhovány spíše jako knihovny tříd s přísnou hierarchií.

Při rozkladu TEXu na nezávislé jednotky jsou zajímavým problémem kruhové závislosti. Existuje jich celá řada. Jednoduchým příkladem je vztah mezi TEXovýma „očima“ a „žaludkem“. Žaludek se sytí příkazy, které vnikají někde v očích, činnost očí ale závisí na nastaveních `\catcode`, ke kterým dochází v žaludku.

To činí snahu o udržování necyklické hierarchie balíků obzvláště obtížnou. Na druhé straně je ale tato snaha velmi žádoucí, pokud plánujeme použít pouze některé z balíků v nějaké jiné aplikaci. $\mathcal{NTS}$ k odstranění kruhových závislostí využívá metodu abstraktních rozhraní. Potřebuje-li určitá třída informaci nebo akci, která není dostupná na odpovídající úrovni hierarchie, definuje rozhraní a přijímá objekt vyhovující tomuto rozhraní jako parametr (konstruktoru nebo příslušné metody). Samotná parametrizace se pak provádí na některé z vyšších úrovní, zpravidla pod střechou (nebo možná v mozku $\mathcal{NTS}$), v rámci balíku `tex`.

Závěr

V průběhu prací vyvstaly některé otázky. Vyplatí se opravdu dodržovat kompatibilitu s TEXem? Ukázalo se totiž, že splnění tohoto požadavku vyžaduje mnohem větší úsilí, než se původně předpokládalo. Existuje mnoho hraničních situací, které je potřeba ošetřit a které se dají zjistit pouze pečlivým studiem zdrojového textu a testováním samotného TEXu. Přitom se zdá, že některé okrajové efekty ani nebyly zamýšleny. Na druhé straně je známo, že mnoha „špinavých triků“, které využívají mezní situace, se používá i v LATEXu a dalších nadstavbách. Vzhledem k důvodům, které byly uvedeny na začátku článku, tedy považujeme kompatibilitu za velmi důležitou a odhadujeme, že se v budoucnosti vyplatí.

Další otázka je spojena s použitým programovacím jazykem. Při praktickém používání se zjistilo, že některé jeho nedostatky byly podceněny. V Javě není možno odvodit uživatelské typy z primitivních typů (čísla, znaky), pro dodržení typové bezpečnosti se tedy používají třídy pro všechny odvozené typy. To pravděpodobně povede k nižší efektivitě. Mnohem důležitějším cílem projektu je ale poskytnout program typově čistý, bezpečný, jasný, dobře strukturovaný a pokud možno krásný, tak aby byl snadno udržovatelný a přístupný pro jeho další vývojáře a uživatele. Bohužel i zde má Java svoje nedostatky, chybí výčtové typy a především parametrické typy, které jsou v C++ již delší dobu známé jako šablony (templates). Vývoj Javy ale ještě není ukončen a je možné, že se řešení těchto problémů v dohledné době dočkáme. Konkrétní programovací jazyk ostatně není natolik podstatný, analýza programového kódu TEXu a objektově orientovaný návrh jsou důležitější.

Původní odhad doby potřebné k vytvoření první verse $\mathcal{NTS}$, kterou má být úplná re-implementace TEXu, byl jeden rok. Ukázalo se, že tento odhad byl poněkud optimistický. Systematická návrhářská a programátorská práce začala na jaře roku 1998 a stále ještě zbývá nezanedbatelnou část díla dokončit. Zejména chybí dělení slov, budování stránek, zarovnání a celá matematická sazba. Množství vykonané práce ale nelze měřit pouze množstvím kódu. Je třeba zahrnout analýzu a studium, návrh a vytvoření základní architektury nového systému. Z tohoto hlediska můžeme již dnes hovořit o dobrém výsledku. Je dokončena celá vstupní část systému včetně zpracování maker, většina zpracování nezávislého na módu, práce s metrikami fontů, manipulace s boxy, lámání odstavců a výstup do `dvi` formátu. To všechno zcela kompatibilně s TEXem, jak potvrdila úspěšná ukázka na konferenci EuroTEX '99.

Současný odhad dokončení první verse je začátek roku 2000 a její oficiální představení po předchozím testování a dokončení dokumentace se plánuje na TUG '2000 v létě. Cílem je poskytnout volně přístupnou základnu k experimentování pro všechny zájemce. Z praktických důvodů samozřejmě počítáme s tím, že i další vývoj systému bude pracovní skupinou koordinován.

Další informace o projektu $\mathcal{NTS}$ je možno získat z jeho oficiální stránky na: `http://nts.tug.org`.

Odkazy

[1] Ken Arnold, James Gosling: *The Java Programming Language, Second Edition*, Addison Wesley Publishing Company, Reading, Mass., prosinec 1997.

[2] Michael Barr: *TEX wish list*, v *TUGboat*, Vol. 13, No. 2, pp. 223–226, červenec 1992.

[3] Nelson H. F. Beebe: *Comments on the future of TEX and METAFONT*, v *TUGboat*, Vol. 11, No. 4, pp. 490–494, listopad 1990.

[4] Hans Hagen: *Some $\mathcal{NTS}$ thoughts*, v EuroTEX '99 Proceedings, pp. 233–240 září 1999, ISSN 1438-9959, též Zpravodaj Československého sdružení uživatelů TEXu, 9(3), 109–115(1999).
[5] Roger Hunter: *A future for TEX*, v *TUGboat*, Vol. 14, No. 3, pp. 183–186, říjen 1993.
[6] Donald E. Knuth: *The future of TEX and METAFONT*, v *TUGboat*, Vol. 11, No. 4, pp. 489–489, listopad 1990.
[7] Donald E. Knuth: *TEX: The Program*, Addison Wesley Publishing Company, Reading, Mass., 1986.
[8] Joachim Lammarsch: *The History of $\mathcal{NTS}$*, v EuroTEX '99 Proceedings, pp. 228-232, září 1999, ISSN 1438-9959.
[9] Frank Mittelbach: *E-TEX: Guidelines for future TEX*, v *TUGboat*, Vol. 11, No. 3, pp. 337–345, září 1990.
[10] Karel Skoupý: *$\mathcal{NTS}$: a New Typesetting System*, v TUG '98 Proceedings, pp. 167-171 srpen 1998 a v *TUGboat*, Vol. 19, No. 3, pp. 318–322, září 1998.
[11] Karel Skoupý, Philip Taylor: *The implementation of $\mathcal{NTS}$*, v EuroTEX '99 Proceedings, pp. 246-260 září 1999, ISSN 1438-9959.
[12] Philip Taylor: *TEX: The next generation*, v *TUGboat*, Vol. 13, No. 2, pp. 138–138, červenec 1992.
[13] Philip Taylor: *The future of TEX*, v EuroTEX '92 Proceedings, pp. 235-254, září 1992, ISBN 80-210-0480-0 a v *TUGboat*, Vol. 13, No. 4, pp. 433–442, prosinec 1992.
[14] Philip Taylor: *NTS: the future of TEX*, v *TUGboat*, Vol. 14, No. 3, pp. 177–182, říjen 1993.
[15] Philip Taylor: *NTS update*, v *TUGboat*, Vol. 14, No. 4, pp. 381–382, prosinec 1993.
[16] Philip Taylor: *Report of the 2nd meeting of the NTS group, February 1994*, v *TUGboat*, Vol. 15, No. 2, pp. 96–97, červen 1994.
[17] Philip Taylor: *Minutes of the NTS meeting held at Lindau on October 11/12th 1994*, v *TUGboat*, Vol. 15, No. 4, pp. 434–437, prosinec 1994.
[18] Philip Taylor: *NTS & ε-TEX: a status report*, v *TUGboat*, Vol. 18, No. 1, pp. 6–12, březen 1997.
[19] Jiří Zlatuška: *$\mathcal{NTS}$: Programming languages and paradigms*, v EuroTEX '99 Proceedings, pp. 241-245 září 1999, ISSN 1438-9959.

Karel Skoupý
E-Mail: skoupy@fi.muni.cz