The Implementation of NTS

Karel Skoupý & Philip Taylor

1. Introduction

Other authors in this series of papers on NTS have explained the reasons for the creation of the NTS project (Joachim Lammarsch), the rationale behind the choice of programming language (Jiři Zlatuška), and future directions in which the project may develop (Hans Hagen). This paper addresses the rather more detailed area of the actual implementation itself, and is intended to provide the reader with as much detail as can reasonably be accommodated in a paper which is intended to appear in the Conference Proceedings. A considerably more detailed version of the paper will eventually be available as an accompaniment to (or possibly integrated in) the JavaDoc documentation which will accompany the released version of NTS.

2. NTS Packages

The implementation language of NTS is Java. It is strictly object oriented, and all of the program code is encapsulated in object methods. The objects are instances of certain classes, and cluster together to form *packages*, discussion of which forms the majority of what follows. At the time of writing, not all packages were complete; in particular, mathematics and hyphenation remain to be implemented, although the majority of the design work for these two remaining tasks is virtually complete.

2.1. Package `base`

The main purpose of this package is to define the elementary data types used in the rest of the system. It is a minimal element of the NTS package hierarchy, meaning that no class here is dependent on any classes from other NTS packages.

The most important classes are:

`Dimen` represents a dimension measured in printers' points (or `mu` units). The fact that the internal representation is the same as that of dimensions in TeX ensures the strict compatibility needed. The public interface of this class tries to be completely independent of its internal representation. To the outside world, a `Dimen` looks like a fraction of points. There are methods for conversion from integer, fraction given by its integral numerator and denominator, floating point number, and *vice versa*. It also provides basic arithmetic operations. In the case of binary operations, the versions for combination with other convertible numeric types are supported too. The general public interface allows for a complete change of the precision or even the internal unit of representation without affecting the other code.

`Glue` reflects another familiar type from TeX. It has its natural dimension and the amount and order of stretchability and shrinkability. It provides arithmetic methods such as adding two `Glue`s, multiplying by a scalar number and also versions for other convertible types.

`Num` represents an integral number. It is just an integer, but wrapped into an object so it can be stored directly in the table of equivalents and can be distinguished from ordinary integers in the code. It serves mainly as the representation of the value of numeric registers (`\count`) (and is somewhat symmetric to `Dimen` and `Glue` for `\dimen` and `\skip` registers).

All the basic classes above provide methods for getting character string representations which can be displayed on screen, in the log file or used by the `\the` primitive.

`LevelEqTable` is the last important and relatively complex class. It is used to implement TeX's table of equivalents and the hash table. Whilst TeX uses an associative hash table only for the meanings of control sequences, NTS stores many more kinds of equivalents in an associative manner.

Any object can be associated with a particular combination of *kind* and *key*. Different kinds are defined for different sorts of equivalents: one is for control sequence meanings, another is for each class of register, still another for catcodes, etc. The key is (according to the kind) either an object (e. g. a control sequence name) or a number (most others). This associative approach for storing register values naturally avoids the limitation on any particular number of registers. Although NTS is compatible with TeX in providing only 256 registers of each sort, this limitation is artificially added and can be easily removed in future.

As the name suggests, besides storing equivalents, the LevelEqTable also maintains pushing and popping of levels which are issued from the grouping in the input language and the corresponding saving and restoring of associated values.

Although the registers were moved from a static array to an associative table, there is still another sort of value which is not associative but which is subject to saving and restoring. These are parameters (such as \tolerance, \hsize, …)—the current value of a parameter is stored in one concrete place. The LevelEqTable provides an interface for these external equivalents too, and maintains saving and restoring for them.

2.2. Package io

This package contains classes and interfaces for reading characters from an input file and writing to the log file. Both of those files may also represent the user's console. The package is independent of the other NTS packages as well as of the package base.

CharCode is an interface and is very interesting (at least, we think so!). There had been discussion as to whether or not to represent character codes by some Java primitive type or by a class. It was decided that the class should be used to clearly distinguish it from other usage of primitive types. Eventually (during development), it turned out that an even more abstract representation (as an interface) matches best its purpose. It declares methods for getting the corresponding character or numeric value, comparing with another CharCode, character or number for matching, making corresponding uppercase or lowercase CharCode, writing on character" oriented output file and several predicates. Most of the methods are there because the TEX language uses characters heavily not only for typesetting but also as numeric values, parts of keywords; in addition, certain characters have an influence on scanning and on log output (\endlinechar, \escapechar, \newlinechar).

Currently the implementation of CharCode used in NTS is just a class containing an ordinary character. But there exists the possibility to use very different representations (e. g. named characters) without changing any NTS code. Such objects can slip through the whole system provided that on the end is an output object which recognizes them and handles them properly. Even several independent implementations of CharCode could co" exist in some future application.

Name has the same relation to CharCode as String does to char. It is used to represent the names of control sequences, jobname, fontnames and filenames scanned from the input.

InputLine represents one line from the input file or from the user's console. There are methods for getting the next CharCode or just peeking to see what the next code is without altering the current reading position. It interprets extended character codes (such as ^^M), ignores the trailing blanks and appends \endlinechar if needed. Another class, LineInput, serves as an input sequence of InputLines from a file or console.

Log is an important interface for printing information on a log file or on the user's console. It declares methods for printing values of primitive types, Strings, CharCodes and Loggables (see below). Several methods are declared to control output line breaking. Class StandardLog implements the Log interface in the standard TEX way.

Loggable is a very simple interface which declares one method for printing on a Log. It is very handy because most of the important classes in NTS implement this interface and so their logging is conveniently handled.

2.3. Package command

Classes in the package command form the interpreter for the TEX input language. Although it is a large package, it has nothing to do with typesetting as yet. In fact, at least one third of the TEX source is not about typesetting at all. It is responsible for the process of scanning input tokens, expanding them and for most of the mode" independent processing such as macro definitions and register assignments.

Token is an abstract class. It declares methods for getting the meaning of a Token, assigning a new meaning (if allowed), matching another Token, and a number of predicates which tell if it is redefinable, is a brace, a letter, and so on.

There are several kinds of Tokens, and they form a small hierarchy of subclasses. Typical examples include CtrlSeqToken, ActiveCharToken, SpaceToken, LetterToken, LeftBraceToken, . . .

Tokenizer is able to provide some sequence of Tokens. There are various subclasses of Tokenizer such as: LineInputTokenizer for tokenization of the input file, Macro.Expansion for macro bodies with supplied parameters, InsertedTokenList for a token list from a token register inserted into the input stream, or BackedToken for just one backed" up token. Tokenizers are pushed onto a TokenizerStack—the analogue of TEX's input stack.

Command is an abstract class which represents each TEX command. Mostly the commands are primitives, each of which is registered under its name in the table of equivalents, but there are important exceptions such as Macro or the meaning of a character. In TEX, the tokens and command codes are represented by the same type and they are often interpreted in both ways which may lead to confusion. NTS strictly separates the concept of token and command. As outlined above, a Token is a piece of input which can have some meaning. The type of this meaning is the Command discussed here.

A Command has methods for execution and expansion. Only some Commands can be expanded, and this property is indicated by another predicate method. The heart of NTS is a cycle very similar to TEX's main_control. In one step, a token is fetched from the input and its meaning is examined. If it is expandable, the appropriate method for expansion is called. If there is some result of expansion, the method is responsible for pushing it onto the input stack. If it is not expandable, the method for executing the command is called.

There is one curious fact about expandable commands, they are executed if their expansion is suppressed by \noexpand. In this case they behave exactly like \relax (they do not do anything apart from re" setting TEX's internal state and terminating any active look" ahead) and they even pretend that they are \relax when examined by \show. For that reason, the whole subtree of expandable commands is derived from the Relax command.

Another important part of Commands interface are the methods for indicating availability and getting some value of a certain type. It is useful when the command occurs on the right side of an assignment, for example, and therefore the registers, parameters, and a few others can provide numeric, dimension, glue or token list values.

The Figure 1 illustrates the hierarchy of command classes giving just some examples. In the time of writing this article, there were more than 150 command classes (together with typesetting commands).

CommandBase is a superclass of Command. It defines only static methods which are related to scanning various elements of input (such as numbers, dimensions, file names, keywords, . . .), maintaining the table of equivalents, input stack and several instances of Log output. As we will see later, there are more objects than just Commands which require such services and so are derived from this abstract class for convenience.

2.4. Package node

Now at last we are getting to typesetting! The classes in this package represent material to be typeset. There are also general interfaces to font metrics and output generators. The package is relatively low in the hierarchy, the classes are dependent only on the base and io packages. That gives them a good chance to be re" used in a different typesetting system which may provide a completely different input language or user interface.

Node is an interface which defines the elementary building block of typesetting material. It has methods to get its sizes (even when affected by some stretching or shrinking), to describe itself on a logfile and to be typeset. There is a hierarchy of classes which implement the Node interface. Some of these are elementary, for example: RuleNode, HKernNode, VKernNode, HSkipNode, VSkipNode, PenaltyNode; other objects are complex and can contain lists of subsidiary nodes: HBoxNode, VBoxNode.

Packer is used when we need to compute the sizes of complex boxes which are built out of lists of nodes. This process is called packaging in

```
Command
├── Prim
│   ├── BeginGroupPrim
│   ├── MessagePrim
│   ├── OpenInPrim
│   ├── PrefixPrim
│   │   ├── AssignPrim
│   │   │   ├── ParamPrim
│   │   │   │   ├── DimenParam
│   │   │   │   └── GlueParam
│   │   │   ├── RegisterPrim
│   │   │   │   ├── CountPrim
│   │   │   │   └── DimenPrim
│   │   │   └── ShorthandDefPrim
│   │   │       ├── CountDefPrim
│   │   │       └── DimenDefPrim
│   │   └── DefPrim
│   ├── ShowPrim
│   └── ShowThePrim
├── SkipDefPrim.Shorthand
├── ToksDefPrim.Shorthand
└── Relax
    └── Expandable
        ├── ExpandablePrim
        │   ├── CondPrim
        │   │   ├── IfCatPrim
        │   │   ├── IfNumPrim
        │   │   └── IfXPrim
        │   ├── InputPrim
        │   └── MeaningPrim
        ├── Macro
        └── Undefined
```

Figure 1: Part of the class hierarchy in package `command`

TEX. The algorithm is essentially the same for horizontal and vertical lists of boxes, only the horizontal and vertical dimensions are flipped for different cases. The abstract class `Packer` defines an abstract algorithm and provides a placeholder to get the appropriate box dimensions. There are then special subclasses for horizontal and vertical boxes which are in turn subclassed outside this package to give the right kind of warnings if something is not decent.

`FontMetric` is an abstract interface for font metric information objects. At the moment, there are only the familiar `tfm` files but that is not a restriction of NTS—it is prepared for any kind of font metric which can be adapted to this interface. There are methods for getting an identification and various numeric or dimension parameters for TEX compatibility. But first of all there are methods to get a `Node` for particular `CharCode`, to get a normal inter" word space and to get a special object which is able to produce the representation of characters, ligatures and kerns for a given sequence of `CharCode`s.

`TypeSetter` has similar characteristics to `FontMetric`. It defines an interface for general typesetting output. There are methods for typesetting a character or a rule at the current position and adjusting this position.

2.5. Package `builder`

This package takes care of the areas concerned with TEX's horizontal, vertical and maths modes. While in TEX there is just one global integer variable which indicates one of the seven possible modes (the three mentioned above are each internal or external, and there is one "no mode") on the top of semantic stack, NTS uses objects which build typesetting material for different modes.

The package is more dependent on the TEX paradigm than is node but is still independent of the TEX language. It is relatively simple and small. Some amount of complexity must be solved when typesetting commands and different modes interact but that issue is addressed in another package.

`Builder` is the root of the hierarchy of classes for different modes. It declares some predicate methods for getting certain characteristics of a given mode and methods for adding a node, kern or skip to the list of nodes which is currently being built. It makes the appropriate versions (horizontal or vertical) of kerns and skips and performs other adjustments if needed. Currently only the modes known from TEX are supported but there is provision for other types of mode (chemical, picture, ...).

2.6. Package `typo`

The package `typo` is a superstructure of the package `command`. It contains all the `Command` subclasses which deal with typesetting currently developed (there will be a package maths for mathematical typesetting commands but it does not yet exist). It utilizes the packages builder and node as well.

TypoCommand is similar to CommandBase but is intended for typesetting commands. It is an intermediate abstract Command class which defines several useful static methods. It maintains a stack of Builders and the current font metric. There are methods for scanning a font metric or box specification from the input, and adding a character or space to the current Builder.

Many classes in this package are derived directly from classes in the command package because they can inherit some useful behaviour from them. They cannot be included in the command package because they need some information which is available only in the typo package (usually by calling some static method of TypoCommand). There are basically two kinds of these: one is \if primitives such as ifhmode or ifvbox which just need some information about the current Builder or a certain box register; another is the commands which are mode independent but typographic such as \setbox, \wd and \chardef.

BuilderCommand is an abstract superclass for commands which are mode dependent. In an open system such as NTS we want new features to be capable of being easily added. There is, for example, a superclass Command which defines a particular set of methods which can be implemented by the new commands in any sensible way. This kind of polymorphism is directly supported by the chosen programming language.

But what to do if in future we want to add a new mode by developing a Builder which offers some new functionality not declared in the Builder interface and some specialized commands which can utilize this new functionality? If we do not want to extend the basic interface (at least until a new version) or even perhaps cannot do it (we are developing a plug" in), the only chance is to examine the type of the current Builder and to use the infamous cast operator if it is our new Builder.

But there is another problem with existing mode dependent commands. How should they behave in the new mode? For this purpose, the BuilderCommand maintains a hash table which associates an Action with each combination of Builder class and Command instance. The association is defined on the level of the NTS configuration and it automatically follows the class hierarchy of Builders. Thanks to this versatility, it is very easy to specify the behaviour of commands in different modes for modified systems. The Action is a subclass of CommandBase so it inherits many methods for scanning the input, dealing with log files and error messages. Actions are usually implemented as inner classes of the corresponding BuilderCommand. A fragment of NTS configuration looks like:

```
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);
```

The BuilderCommand corresponding to the TEX primitive \hrule defines three actions: it performs the normal operation in vertical mode, finishes the current paragraph (if any) in horizontal mode and complains inside a \hbox. The \vrule performs normally in any horizontal mode and enters a new paragraph in vertical mode. There is in fact only one class (RulePrim) which has two instances with names hrule and vrule and different parameters; they are assigned different Actions for the same modes. All the Actions NORMAL, START_PAR, FINISH_PAR and BAD_HRULE are instances of inner classes inside RulePrim or its superclass.

Other examples of BuilderCommand are: HBoxPrim, VBoxPrim, VTopPrim, LowerPrim, MoveLeftPrim, BoxPrim, KernPrim, CharPrim, ExSpacePrim, AccentPrim, AnySkipPrim.

Group is another subclass of CommandBase. Its subclasses cover the various types of group in TEX. There are groups such as SimpleGroup for a pair of braces, SemiSimpleGroup for the \begingroup and \endgroup, HBoxGroup, VBoxGroup or VTopGroup. The Group itself is defined and the stack of Groups is maintained in CommandBase but most of the subclasses belong to the package typo.

Groups have one problem in common with Builder. Their closing commands behave differently in combination with certain type of Group. The right brace cannot match \begingroup and \endgroup cannot match the left brace. The problem is solved in exactly the same way as for combinations of commands and Builders.

2.7. Package tfm

The package tfm implements a particular type of font metric information—the TEX font metric file—for use in NTS. It can be used as an example for implementing other types of font metric.

TeXFm is a class which represents the low" level—almost raw—format of a TEX font metric file. Some complications are hidden but its public interface reflects just the information which is available in the file. It uses several auxiliary classes because the tfm format is too complex to be captured by only one program file. As an example, the whole process of reading a tfm file is done by the class TeXFmLoader which creates an instance of TeXFm. TeXFm itself has methods for getting information concerning the characters, ligatures and kernings for pairs of characters, extensible recipes and sequences of enlarging characters. Another method is provided for printing its representation as a property list. This is used by a small Java application tftopl which—thanks to TeXFm—shares most of the code with NTS.

TeXFontMetric is an adaptation of TeXFm which implements the FontMetric interface from the node package. It is a wrapper which uses the natural methods of TeXFm and provides the methods required by the rest of NTS. This approach is probably useful for future implementations of other types of font metric. We cannot expect that some third party will provide the exact interface even if a Java class is supplied for access.

2.8. Package dvi

This package implements the dvi format as one of the possible output formats for NTS. In many aspects it will be similar to the package tfm but it is too early to say more as the development of this part has just started.

2.9. Package tex

This package is an umbrella for the other NTS packages, and it is by far the messiest part of the system. All the classes and packages so far are designed to provide a clear and elegant interface and to be as independent of other classes and packages as possible. But in TEX itself, there are so many unclear dependencies. That was one reason for starting the whole NTS project in the first place. The classes in this package join all the independent units together, and in addition all the weird cases were exported from the clean design of other packages to here if this was possible. That is the main reason that the code sometimes looks rather messy here.

Besides this, there are classes for maintaining the error pool so that commands are not dependent on the way in which the error messages are given.

The most interesting part of this package is the class Primitives which contains the configuration of the whole system. There was already an example in package typo.

3. Modularity and Configurability

To develop a system which is as modular as possible was one of the main desiderata. In the current TEX implementation, there are a lot of dependencies. Experience shows that it is very difficult and dangerous to make some non" trivial changes since these can lead to a number of possible unclear side" effects. The approach taken in developing NTS has been to make all dependencies explicit and clear. All classes have a well" defined interface of public methods which is used for all communication. There are no uncontrolled changes of global variables. This manner of programming is greatly supported by the Java object" oriented language.

Another motivation for making code units independent is to allow substitutions of some modules

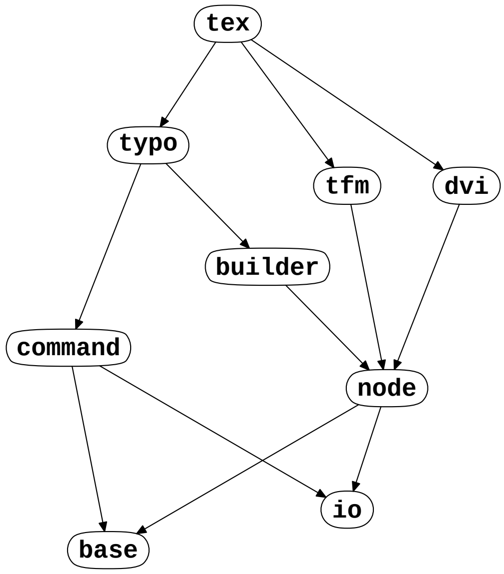

Figure 2: Dependencies of the packages in NTS

by other modules with the same interface but a different underlying implementation. Independent classes or packages can be also used as building blocks for another system. The NTS packages are therefore designed rather as class libraries with a strict hierarchy.

An interesting problem concerned with the decomposition of TEX into independent units is the problem of cyclic dependencies. There are many of them. A simple example is the relation between TEX's "eyes" and "stomach". The stomach is fed by commands which originate at the eyes, but the action of the eyes depends on `\catcode` settings which originate from the stomach.

This makes particularly difficult to maintain a non” cyclic hierarchy of packages. On the other hand, it is very desirable if we want to use only some of them in another application. The method that NTS uses to avoid such cyclic dependencies is via abstract interfaces. If some class needs information or an action which is not available at the current level of hierarchy, it defines an interface and accepts an object which implements it as a parameter (of its constructor or of some method). The parameterisation is then made at some higher level, usually in the umbrella (or maybe NTS's brain)—the package `tex`.

4. Epilogue [PT]

The previous text is virtually entirely Karel Skoupý's work; not only has he done a splendid job in performing the analysis and re” implementation, he has also demonstrated a remarkable facility for documenting his work in an approachable and readable manner. I would like to record my sincere appreciation of Karel's work, which is, I know, shared by all other members of the team. However, Karel left a few topics deliberately unaddressed, and asked me to fill in the gaps: what follows, therefore, is my interpretation of Karel's verbal explanation, and may differ from reality to a greater or lesser extent.

4.1. The problems of TEX compatibility

With the exception of a few eminent practitioners such as Edsger Dijkstra, programmers used to estimate that they would spend about 15% of their time coding and about 85% debugging. This analysis excluded design time, of course, and improved programming paradigms—from structured programming through dataflow to object” orientation and beyond—have done much to reduce this ratio. However, in the case of NTS, Karel estimates that the vast proportion of "lost" time was taken up with implementing strict TEX compatibility. Of course, "lost" is a subjective term: since strict compatibility was fundamental to the project, the time was not lost *per se*; but in many ways it can reasonably be regarded as non” productive time. The primary reason for the disproportionate amount taken to achieve compatbility is that many (most?) of TEX's *nuances* are not defined in "The TEXbook" but can only be determined by the most minute examination of the code itself, or by laborious testing and comparison.

In restrospect, was all this extra work justified? I believe so. We did not set out to design and implement a New Typesetting System *ex nihilo*, but rather to create the basis for a new typesetting system by re” implementing TEX. The reason for this is clear: we intended that, for the first few years of NTS's life, it should be capable of being used as a *complete replacement* for TEX, just as can ε-TEX. The slightest deviation from 100%-conformity with TEX could have spelled disaster, since one genuine report of a discrepancy between the behaviour of TEX and the behaviour of NTS which the team were unable (or unwilling) to rectify could destroy user” confidence in the system. Therefore, even though it has taken Karel *far* longer to bring the NTS project to its present state of near” completion than was originally anticipated, we believe that this time was not wasted but was instead used in the most constructive way possible: to ensure that non” conformity with TEX can never be held up as an argument against the adoption of NTS.

4.2. Java: was this the right language?

When NTS was first mooted, the group debated for some time which language should be used for its implementation. All agreed that “a modern, rapid” prototyping, language” (Joachim Schrod was the first member of the group to use this phrase) should be adopted, but which? Jiři Zlatuška, in his contribution to this series of papers, outlines many of the reasons why Java was ultimately preferred, even though Java does *not* strictly qualify as “a modern, rapid” prototyping, language” and certainly falls short of both PROLOG and LISP in this respect. However, what Java *does* have is *portability* (“compile once, run anywhere” is Sun’s justifiable claim), *object orientation*, *Internet awareness*, *total standardisation* (modulo a certain well” known company’s attempts to foist off a variant on an unsuspecting community, thereby destroying Java in the process), and almost universal availability (Java is probably available for more modern platforms from a single, standardised source than any other programming language available today).

But what of its shortcomings? Some might argue that its lack of multiple inheritance is a weakness; but Karel has never reported to us that lack of this feature caused him any difficulties. Far more important is Java’s weakness in one single area: type” safeness. Sure, Java is type” safe, with rigorous checking enforced both at compile and run” time; but what Java lacks is *type definitions*. Thus (for example) it would have been only too easy for Karel to overload the type “integer” with all manner of different meanings, and no amount of type” checking could ever have determined that (for example) somewhere in the code he was accidentally assigning a catcode to a character variable. In practice, Karel has overcome this by using classes and objects to a far greater extent than he would have wished; this will undoubtedly result in poorer run” time performance (one can hardly expect even the finest optimising compiler to generate code for an object assignment that takes no more machine cycles than the code for an integer assignment), but has the enormous benefit of ensuring that errors of coding such as that suggested above can never occur. And this, in turn, will make it far easier for others to extend and enhance NTS, since the adoption of this rigorous prgramming paradigm will ensure that even a complete newcomer to the code cannot fall into traps such as this.

Thus, in hindsight, I remain convinced that Java *was*, and *is*, the right language for the implementation of NTS. But let us hope that some successor to Java, in the not” too” distant future, eliminates this weakness from the language design; then, perhaps, NTS can be re” implemented in this new language, with all the benefits of Java but with far greater run” time efficiency. That is surely something worth—not waiting for—but hoping for; a truly modular and portable TEX” compatible system which is as efficient as TEX itself. NTS: Not Today, but Soon!

◇ Karel Skoupý
skoupy@informatics.muni.cz

◇ Philip Taylor
P.Taylor@mail.rhbnc.ac.uk

Autorem logo konferencji jest Maciej Strzelczyk