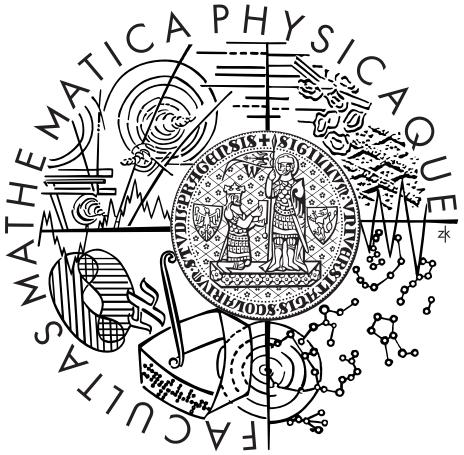
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Peter Fabian

Refactoring tree editor TrEd

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: doc. Ing. Zdeněk Žabokrtský, Ph.D.
Study programme: Computer Science
Specialization: Mathematical Linguistics

Prague 2011

I would like to thank my supervisor, Mr. Žabokrtský for the inspirations and his patience. I would also like to thank my family for their endless support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date .....

Název práce: Refaktorizace editoru stromů TrEd

Autor: Peter Fabian

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí diplomové práce: doc. Ing. Zdeněk Žabokrtský, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Cílem práce bylo refaktorizovat editor stromů TrEd, zlepšit tím jeho modularitu, usnadnit údržbu a další vývoj aplikace. Důraz byl také kladen na zachování rychlosti programu. Zdrojový kód TrEdu byl prozkoumán metodami statické a dynamické analýzy, které pomohly identifikovat problémová místa. Bylo vytvořeno 50 nových modulů a přesunuto značné množství kódu. Byla také vytvořena sada testů, zvolena pravidla pro psaní nového kódu a sepsána dokumentace části stávajícího kódu. Kód byl po refaktorizaci opětovně podroben kvalitativní i kvantitativní analýze a její výsledky byly porovnány se stavem před refaktorizací.

Klíčová slova: Editor stromů TrEd, Perl, refaktorizace, analýza kódu

Title: Refactoring tree editor TrEd

Author: Peter Fabian

Department: Institute of Formal and Applied Linguistics

Supervisor: doc. Ing. Zdeněk Žabokrtský, Ph.D., Institute of Formal and Applied Linguistics

Abstract: The main goal of the thesis was to refactor tree editor TrEd, improve its modularity, maintainability and make its further development less difficult. Static and dynamic analysis of TrEd have been performed in order to help us find acrid spots in the source code. More than 230 subroutines and data structures have been moved between packages, 50 new packages and a test suite with more than 1,300 tests have been created. A new coding style have been chosen for further development and most severe violations of this standard have been fixed. After the changes done on the source code, it have been analyzed again and the results have been compared with the previous state.

Keywords: Tree Editor TrEd, Perl, code refactoring, code analysis

Contents

1. Introduction

"TrEd is a fully customizable and programmable graphical editor and viewer of tree-like structures such as dependency trees. Among other projects, it was used as the main annotation tool for syntactical and tectogrammatical annotations of The Prague Dependency Treebank, as well as for decision-tree based morphological annotation of The Prague Arabic Dependency Treebank." [8] It is actively used in several academic institutions around the world. Since the original author of TrEd, Petr Pajas, retired from working on this project and TrEd needs to be maintained and new features need to be added, many people could benefit from its improved modularity and robustness.

Changing the internal structure of computer programs and improving their design is called refactoring. "It is the process of improving the internal structure of the application's source code without changing its behavior" [3]. Many types of refactoring exist, [3] presents a great list of smaller and bigger refactorings in his book. However, it's still the programmer, who has to decide whether to split a large function into more smaller ones (i.e. extract subroutine) or inline a small function into another one, or decide whether to introduce a temporary variable or use function call instead.

Refactoring, and in fact any change in software design and structure, is affected by many small decisions. Unlike the behavior of the program and correctness of its output, which is usually subject to program specification, the internal structure of application is more affected by the individual styles and attitudes of the developers who write the code. And since TrEd is almost exclusively a product of one developer (Petr Pajas), who worked on it for almost 10 years, its whole structure is subject to his own programming style.

Refactoring is also a challenging task, because it requires broad and deep knowledge of programming languages used by the application, code testing techniques and principles of applications' design and architecture.

This thesis is thus dedicated to improvement of TrEd's internal structure, which should result in enhancing its external and internal quality.

The second chapter of this thesis is devoted to analysis of TrEd's source code. We give an overview of how much source code TrEd consists of and how it can be divided into several categories. Afterwards, we use various tools for static and dynamic analysis of Perl source code to identify the weakest points and candidates for refactoring. Chapter 3 describes the design and the implementation of key parts of TrEd. Chapter 4 presents coding style chosen for TrEd based on widely adopted publication – Perl Best Practices by Damian Conway. The fifth chapter describes changes done on TrEd's source code during the work on this thesis. To measure the quality of these changes, several metrics were evaluated. Finally, the last chapter consists of discussion on how could be TrEd further improved and possibilities of future work.

2. Code Analysis

Code analysis is the process of automatic analyzing the source code of computer programs. There are two basic types of code analysis:

1. **static analysis** and
2. **dynamic analysis**.

Static analysis examines the source code of the application without running it, while dynamic analysis examines the information gathered during the execution of the program.

The first one tries to analyze all the paths through the program and is usually easier to do for compiled languages like C/C++ or at least for languages with a data type system like Java. Dynamic analysis maps just one run of the program and thus can hardly analyze the program as a whole. It is usually the only option for languages like Perl, where one can create new syntactic constructs and use built-in functions like `eval` and `do`, which can execute user input or interpret arbitrary files as Perl code.

Code analysis can help us to identify the most problematic parts of application, which are usually called code smells in the world of refactoring. These are the areas of the program's source code which indicate that some refactoring could improve the quality of the code in question.

In this chapter, we first take a look at code smells as they are described in [3]. Then, we shortly describe the overall code structure of TrEd and try to identify the most acrid areas of TrEd's source code by examining the results of static and dynamic analysis of original TrEd.

2.1 Code smells

Since Martin Fowler wrote his book [3], signs of code that indicate the need for refactoring are called smells. Originally, Fowler uses term code smells, other authors broadened this term to architecture smells [5], which indicate also signs of program architecture that indicate design faults. Code smells are usually not strictly defined quantitatively, but one needs intuition as well as experience to see what code to refactor. Fowler and Beck [3] presents 22 hints or advices how to look for code smells and when they feel that refactoring is necessary.

2.1.1 Duplicated Code

Removing duplicated code is an easy way how to reduce number of lines of code of the program without losing functionality. Not only Fowler, [2], but also general wisdom states clearly that duplicated code belongs to a subroutine.

Code duplication can easily introduce bugs as the programmer easily loses track of all the code copied over the program. When a modification is needed or some piece of duplicated code is buggy, it must be changed in every copy, since the modification is usually relevant for all the instances of copied code. It is very easy to forget to change one of the instances of duplicated code, especially if the

maintenance programmer is not the one who wrote the code. Furthermore, code duplication can be a sign of design without proper abstraction.

The percentage of code duplicity in TrEd is not very high; `tred` and `btred` contain some common code and several subroutines have the same implementation on different places in code, e.g. `uniq`, appeared three times in various packages in TrEd's core. More information on code duplication in TrEd can be found in Section 2.2.4.

2.1.2 Long Subroutines

The longer the procedure, the more it is difficult to understand, maintain and test it. Fowler takes this approach to the limit and states that if you want to comment a piece of code, make it a subroutine and give it a descriptive name. "The key is not the length of the subroutine, but rather the semantic distance between what the function does and how it is done" [3].

TrEd's start up and some parts of modules are written in a fairly sequential fashion, thus long subroutines are not very rare. As an example: `TrEd::Extensions::_populate_extension_pane` subroutine was more than 500 lines long before refactoring, `main::startMain` was even more than 950 lines long before refactoring. Subroutines should do one thing only and do it efficiently. Long subroutine is hard to grasp, especially when they use a mix of local and global variables, some of them with short undescriptive names like $l or $f.

Subroutines this long are also almost untestable, since it is very hard to find all the paths through the subroutine and the number of combinations of many conditions and loops can grow huge.

More exact numbers about the length of subroutines can be found in Section 2.2.2.

2.1.3 Large Class

Large classes are usually trying to do too much and use many variables. TrEd doesn't use classes much since it is not written in object-oriented manner, but it uses Perl packages to group subroutines and variables together. Large packages are from conceptual point the same evil as long methods. They should be split into smaller packages with more fine-grained and clear functionality.

TrEd's `main` package, as stated above, is a good example of a large package – it is more than 13,000 lines long and contains almost 600 subroutines. The largest package, if we do not count the main package, is the default macro file with API for macros and extensions – more than 4600 lines of code.

Class trying to do too much often shows up as too many instance variables.

2.1.4 Long Parameter List

In objected-oriented programs the lists of parameters are usually shorter than in procedural programs. The reason for this is that on one hand, object's methods use object's variables and on the other hand, it is usually possible to pass another object as a parameter and use object's methods to get needed values.

In Perl, programmers can use named parameters and pass only one (usually anonymous) hash reference as a function parameter. Many functions take hash

reference holding TrEd's configuration as their parameter. [2] advices to use named arguments whenever the subroutine has more than 3 parameters. In fact, it is a similar approach to Fowler's Introduce Parameter Object refactoring [3], the only difference is that the parameter object is anonymous and temporary.

2.1.5 Divergent Change

If changes in one external concept means that changes in two different conceptual areas in one class are necessary, it is usually vital to split the class (or package) into two or more smaller classes. Typical example is the main package of TrEd, which contains many subroutines with various responsibilities and concepts.

This smell is rather a conceptual one. From a certain point of view, it may overlap with *Large Class* code smell (Section 2.1.3).

2.1.6 Shotgun Surgery

This code smell is the opposite of the previous code smell: a change in one package repeatedly requires little changes in many other classes. In this case all the little changes can be abstracted into a separate package that covers a single concept.

The true question, though, is to where to draw the line between having semantically different concepts grouped in one package and many small packages that conforms to one logical concept.

The packages in TrEd are intertwined and changes in one package often requires many changes in other packages. In the case of TrEd, I think that it is caused by sharing global variables and by not adhering to encapsulation.

2.1.7 Feature Envy

Since one of the basic principles for creating packages (or classes) is to group together data and the processes that operate on this data, we should be cautious, when a function works with data from another package. If a subroutine in one package uses lots of pieces from another package, it is an example of feature envy code smell. The solution is to move the subroutine to the other package. Of course, there are cases when a subroutine works half of the time with data from one class and half of the time with data from another class. In this case, the subroutine can usually be split into two pieces, each of them shall be put to package where it belongs.

The example from TrEd could be subroutine `applyWindowStylesheet` in `TrEd::Utils` package, which operated on a `TrEd::Window` object and applied specified stylesheet to its tree view. This method has been moved to `TrEd::Window` package in the process of refactoring.

2.1.8 Data Clumps

Sometimes data appear together in groups in more places in a source code. Data that's always hanging with each other (e.g. street name, street number, zip) can be extracted into a class. This can also help in reducing the number of arguments passed to subroutines using this data.

I have not spotted the presence of this code smell in TrEd.

2.1.9 Primitive Obsession

There is no reason to be reluctant to use small objects and classes and create and use them frequently. Even some languages that did not initially supported objects like Perl adds support for them now. Therefore there is no good reason to be obsessed with primitive types.

Since Perl is a high level language, its built-in data types (namely hashes and arrays) are used to build more complex objects. The `Filelist` package can serve as an example that TrEd does not suffer from primitive obsession. The filelist is in fact just an array of files.

2.1.10 Switch (Case) Statements

In object-oriented programming, switch statements are more rare than in classic structural programming. This is because the switch statement can be elegantly transformed into polymorphic classes.

TrEd does not use objects very often, nor switch statements. Cascading if-elseif-else with many possibilities is, however, not very rare.

2.1.11 Parallel Inheritance Hierarchies

Parallel inheritance hierarchies are a special type of 2.1.6. Every time a subclass of one class is created, a subclass of another class has to be created as well. TrEd almost does not use classes, therefore this code smell is not present in the source code.

2.1.12 Lazy Class

A class or package that is not used just adds to program complexity without any measurable benefits. If the class does too little, its data and subroutines can be attached to the class that uses it and the lazy class can be discarded.

The example of a class that is not used neither in TrEd, nor in the extensions and macros is `Tk::EditableCanvas` class. This class has been deleted in the processes of refactoring.

2.1.13 Speculative Generality

If there is a class designed to do something in the future but never ends up doing it, it is a good candidate for removal. Constructing patulous APIs whose functions are never used just makes program more complex and harder to maintain.

In TrEd, for example the `TrEd::MinMax::shuffle` function, which shuffles elements of an array randomly, is never used. This subroutine has been deleted in the processes of refactoring.

2.1.14 Temporary Field

Variables used only temporarily by some of the functions, which are of no value to other functions after they have been used, could be confusing. These variables can be moved into separate class along with the methods which use them.

This problem does not occur in TrEd, because it does not use classes to encapsulate variables inside them. Package variables are usually part of the API of corresponding packages and lexical variables are usually passed as arguments of functions. This behaviour should, however, be changed. Package variables should be encapsulated and made reachable via accessor methods.

2.1.15 Message Chains

If a message in program has to be delivered using several middle men, it means that the client is tightly coupled to the structure of the navigation. Hiding a delegate shortens the message chain and may improve understandability.

In TrEd, sometimes, there are longer message chains, but because it is not written in object-oriented way, they are not very common. Objects used in TrEd, e.g. `Treex::PML::Document` sometimes do use longer message chains, but these are used to communicate with other objects in `Treex::PML` library.

2.1.16 Middle Man

If a considerable amount of work of a class is dedicated just to delegate messages to another class, the "middle man" can be sometimes avoided and a direct access can be used.

As well as with the previous example, we can mention `Treex::PML` library here, too. Since it had been a part of TrEd until it was removed as a separate library, TrEd sometimes use direct access to objects in this library, e.g. `Treex::PML::Factory` is bypassed when a new node is created in `TrEd::Window ::TreeBasics` module.

2.1.17 Inappropriate Intimacy

If two classes are intertwined together too much, they use each others methods and data often, a common subset of these classes can be extracted to new class. Circular references between classes and packages are not good from conceptual point of view – they are harder to understand, maintain, reuse and test [5].

Sometimes, however, special kinds of classes (e.g. iterators) can exhibit such behaviour. These are, of course, designed to work this way and refactoring them is not desirable.

2.1.18 Alternative Classes with Different Interfaces

If two methods in different classes or packages have different names, but do the same thing, they should be renamed. More methods could be added until the classes don't have the same interface. (If a duplication of code should occur, common code could be moved to common superclass).

This code smell was encountered when new dialog packages was being created. Since the subroutines which created the dialogs had been extracted from main package, each subroutine had different name. These were later in the process of refactoring unified to make all the dialogs have the same common interface.

2.1.19 Incomplete Library Class

If a method is missing from library and we can't change the library, we can either create this method in our object or make our own extension/subclass of the library.

In TrEd there is a case of `List::Util` module from Perl's core modules. Since its usage in safe compartment was problematic, the functionality needed in safe compartment was reimplemented as `TrEd::MinMax` module.

2.1.20 Data Class

The data classes are basically just big storages of data. These data should be encapsulated and more methods which work with the data should be added over time the class evolves.

An example of such package in TrEd is the `TrEd::Config`[1] package. For the sake of speed, especially because `tredDebug` is read often, these variables are exported and can be accessed directly.

2.1.21 Refused Bequest

Subclass which does not use methods of superclass is a small smell. Stronger code smell appears if the subclass does not support the interface of the superclass.

This is not a problem in TrEd, because it uses only very little inheritance.

2.1.22 Comments

If a block of code needs several lines of explanation, maybe it should better constitute a distinct function, which can be extracted from its original position. This approach allows for better abstraction and increases understandability of code.

Comments in TrEd were used to extract not only subroutines, but whole packages, e.g. `TrEd::ManageFilelists` has been created this way.

2.2 Static code analysis

As mentioned earlier, static code analysis examines code without running it (or, at least, uses results of analysis which can be obtained without executing the code). The term static code analysis usually means an analysis performed automatically by computer programs, while analysis by humans is usually called code review or code comprehension. The static code analysis tools try to find locations of possible errors, obsoleted implementation, dangerous language constructs or code duplication within the source code. More sophisticated tools are able to create data-flow diagrams and help with formal verification of computer programs.

Unfortunately, more sophisticated tools are available for mainly for C++ or Java, but their support for Perl is very limited. However, some of the tools for

[1] more details about `TrEd::Config` package can be found in Section 3.8

detection of code duplication are independent of language. There are also a few Perl tools for static code analysis of Perl are available on CPAN[2], too.

The main tools used for static code analysis of TrEd are:

1. `Perl::Metrics::Simple` CPAN module[3]
2. `Perl::Critic` CPAN module[4]
3. CCFinderX[5]

The first one, `Perl::Metrics::Simple`, is a code metrics tool. It counts the number of lines of code inside subroutines, outside of subroutines, it can also calculate the *McCabe* or *cyclomatic complexity*[6] of subroutines and look at the source code from quantitative point of view. The cyclomatic complexity of code is a code measure which is usually computed by counting a number of decision points (conditions, loops, logical expressions, etc.) in the subroutine. A little bit more exactly, it can be seen as the number of independent paths in control-flow graph of the examined code.

The second tool, `Perl::Critic`, is more aimed at quality of code. It uses a set of rules and policies to determine possibly dangerous language constructs or warn against using unclear coding style.

The last tool used for static code analysis of TrEd is CCFinderX, a tool to find and identify code clones within large code bases.

Besides these three tools, various other tools like `AutoDia`[7], `UML::Sequence`[8] or Perl Subroutine Call Tree script[9] were used, but as the number of subroutines and used modules in TrEd is fairly large, the graphical representations of dependencies between modules or call graphs contain too much information and are of little practical use. The visualization of TrEd structure would require tools that can display more abstract structures or tools that can isolate only part of the web of dependencies and allow programmer to focus on smaller portions of application. Some of the output from these tools is presented in Chapter 3.

2.2.1 Overview

The original source code of TrEd before refactoring can be divided to

- TrEd's core,
- modules,
- macros,
- extensions,

[2]Comprehensive Perl Archive Network, http://search.cpan.org/
[3]http://search.cpan.org/dist/Perl-Metrics-Simple/
[4]http://search.cpan.org/dist/Perl-Critic/
[5]http://www.ccfinder.net/
[6]see [6] for details
[7]http://search.cpan.org/dist/Autodia/
[8]http://search.cpan.org/dist/UML-Sequence/
[9]http://www.teragridforum.org/mediawiki/index.php?title=Perl_Static_Source_Code_Analysis

Significant part of TrEd's functionality had been moved to `Treex::PML` library by its original author, Petr Pajas, before our work on this refactoring started. `Treex::PML` library, which is available on CPAN, provides API for manipulating linguistically annotated treebanks and implements a generic data-model of an XML-based format called Prague Markup Language (PML) [9]. It also provides an IO system with `Treex::PML::Document` objects for representing trees in XML files and `Treex::PML::Backend` classes that supports loading treebank files in various file formats, e.g. CSTS[10], FS[11], NTRED[12], PML[13], TrXML[14], etc.

This library was not considered to be a part of TrEd for purposes of this thesis and its refactoring is not considered here.

The TrEd's *core* includes `tred`, `btred` (command-line macro processor of the tree editor TrEd), `ntred` (bTrEd server controller/hub/client), `jtred` and `any2any` Perl scripts.

The *modules* which implement basic TrEd functionality have mainly `TrEd::` namespace prefix. Several additions to Tk library, corrections of default Tk modules and wrappers for backward compatibility belong to this category, too.

TrEd *macros* provide a system for executing code written by TrEd users to extend TrEd's functionality. This code can be evaluated interactively in TrEd or as a batch on arbitrarily long filelists in bTrEd and nTrEd. The macros also contain API for extensions so they can conveniently use TrEd's functions without exposing TrEd internals and implementation details.

The *extensions* are packages which may contain TrEd stylesheets, additional Perl libraries, other resources needed to add support for new file types (like xml schemas, etc) and executable macros to glue all the package together. The extensions are a powerful tool as they can also add new toolbars and other user interface elements, chenge key bindings in TrEd and possibly introduce a new level of functionality like PML Tree Query extension.

2.2.2 Code Metrics

One of the basic static code analysis methods is code metrics, i.e. counting how many lines of code a program contains, how many lines of documentation per line of code is present, how many packages and subroutines the program comprises of, etc. One can hardly tell that e.g. 20 % of the code should be documentation and that average subroutine should not be longer than 10 lines of code, but these numbers can give you a signal, where to look for odd code constructs, ridiculously long functions or huge packages. These numbers can lead you to the low hanging fruit and show you where to start with refactoring.

Of course, many lines of code or documentation written by a programmer does not necessarily mean the code is high-quality and maintainable, and that the documentation is understandable and up-to date with the current code. Scarcity

[10] SGML-based format called CSTS used in the Prague Dependency Treebank 1.0, see also http://ufal.mff.cuni.cz/pdt/Corpora/PDT_1.0/Doc/csts/DTD-HOME.html

[11] Feature structure format, see also http://ufal.mff.cuni.cz/pdt/Corpora/PDT_1.0/Doc/fs.html

[12] backend for exchanging data with remote ntred servers

[13] XML-based data format intended primarily for interchange of linguistic annotations, see also http://ufal.mff.cuni.cz/jazz/PML/

[14] XML-based representation of the FS format used in Prague Dependency Treebank 1.0, see also http://ufal.mff.cuni.cz/pdt/Corpora/PDT_1.0/Doc/whatis.html

	Files	Lines of code	Lines of #	Lines of POD	Subroutines
Core files	4	18,199	844	1,476	426
Modules	53	22,826	1,208	2,105	686
Macros	22	9,582	458	2,553	375
Extensions	254	164,012	7,284	5,948	3,677
Treex::PML	49	21,013	540	6,239	811
Total[15]	333	214,619	9,794	12,082	5,164

Table 2.1: TrEd code overview

	LOC/file	sub/file	lo#/LOC	loPOD/file	loPOD/LOC
Core files	4549.8	106.5	0.046	369.0	0.081
Modules	430.7	12.9	0.053	39.7	0.092
Macros	435.5	17.0	0.048	116.0	0.266
Extensions	645.7	14.5	0.044	23.4	0.036
Total	6,061.7	151.0	0.192	548.2	0.476

Table 2.2: TrEd code overview – relative

of documentation can be, however, a sign of a code that is underdocumented. Reading the implementation of each function just to find out its purpose (not to mention hunting down the correct number and type of function parameters, remember we are using Perl) costs time and effort and slows down every maintenance or adding new features.

Lines of code per subroutine or number of subroutines per package can be a sign of poor design of application. [5] states a guideline that "If an element consists of more than 30 subelements, it is highly probable that there is a serious problem", i.e. a subroutine should not have more than 30 lines of code, a package/class should not contain more than 30 subroutines and a subsystem should not contain more than 30 classes. These numbers can provide a hint where to look for "architecture smells". Bigger subroutines, packages and subsystems are not only more difficult to understand, but also it is almost impossible to test and verify them. A function 500 lines long, including 20 loops, 90 conditions and 10 anonymous inline subroutines without a single line of documentation is very hard to understand and maintain, not to mention writing tests (this is the case for `TrEd::Extensions::_populate_extension_pane()` function).

On the other hand, [7] mentions several studies that showed that lower length of subroutines is not correlated with lower error rate. Smaller subroutines are reportedly cheaper to fix, but on average they contain more errors. The code needed to be changed very rarely for subroutines which were around 100 lines long and most dangerous routines are those with more than 500 lines of code [7].

The basic statistic of TrEd before refactoring can be seen in Table 2.1 and Figure 2.1.

The "lines of code" column represents the actual number of lines in all files included in the category. Lines of comments are counted as all the lines starting with hash sign (#). These usually contain either explanation of code's purpose or intention, but these can also be used to comment out obsolete code. POD,

[15] without `Treex::PML`

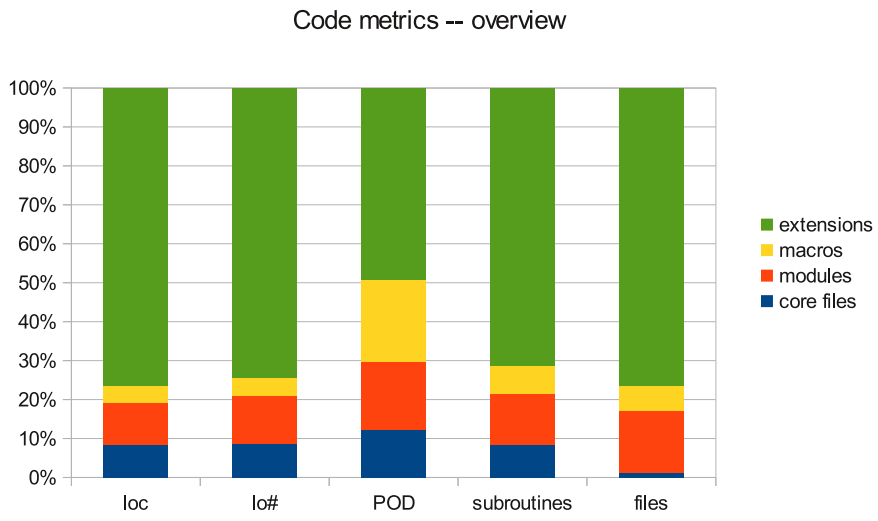


Figure 2.1: TrEd's source code – code metrics overview

which stands for Plain Old Documentation is a simple format used frequently for documentation of Perl code. The documentation in TrEd is, however, only intended for the end user. There is only a little of programmer's documentation available in original TrEd and it is usually restricted to sparse remarks and comments about tricky spots in source code.

As we can see in Figure 2.1, macros contain relatively more documentation than the rest of the code (approximately 31%). On the other hand, extensions, compared to their large amount of code, contain relatively small percentage of documentation (around 8%). One of the aims of this thesis is to increase the documentation levels, mainly in TrEd core and modules, which originally contained 12 % and 14 % of documentation, respectively. Moreover, we have to take into consideration that some of the documentation were only unedited POD templates.

Table 2.1 show us that the amount of code in TrEd's core files is almost 18,200 lines. More than 13,000 lines of TrEd's core files is located in main `tred` file. This file contains 360 subroutines and hardly any programmer's documentation. The amount of code in modules is slightly larger than the amount of code in the core files. One of the objectives of this thesis is to move most of the code from the main namespace of four core files to modules. Macros account for another 10,000 lines of code. Turning the macros into standard Perl code is another objective of this thesis.

TrEd's core

As we can see in Table 2.3 and Figures 2.4 and 2.5, a quarter of subroutines in TrEd core is more than 30 lines long and the cyclomatic complexity of a quarter of subroutines is over 13. [6] advices to split functions whenever their cyclomatic complexity exceeds 10; 138 out of 430 (32%) subroutines in TrEd's core have cyclomatic complexity above 10.

In Table 2.4, we shall observe that the longest subroutines in TrEd's core are

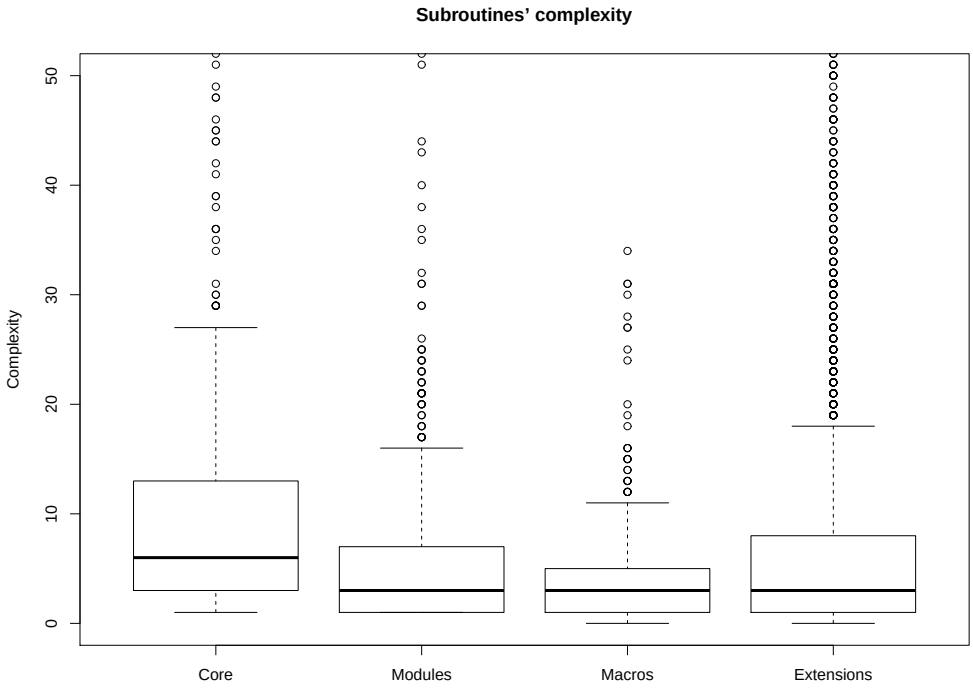


Figure 2.2: TrEd's source code – subroutines' complexity overview

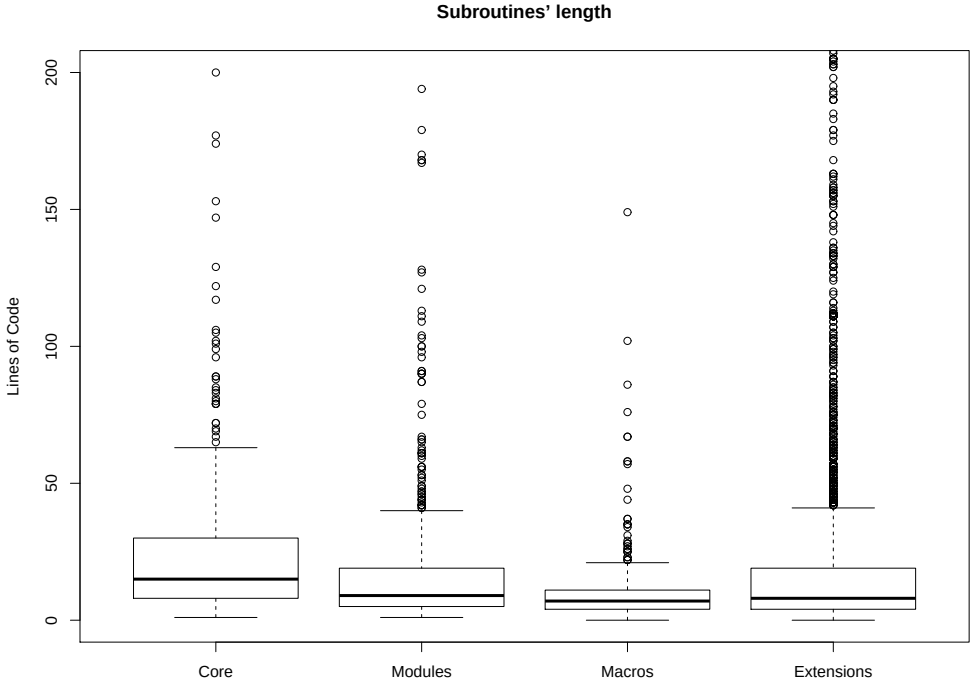


Figure 2.3: TrEd's source code – subroutines' length overview

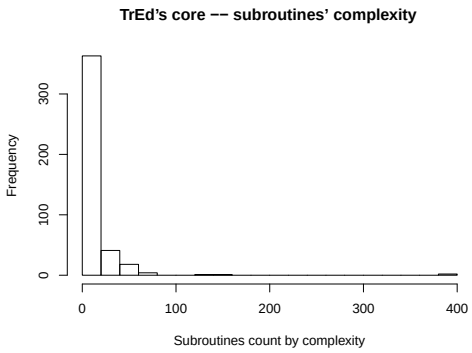


Figure 2.4: TrEd's core source code: subroutines' complexity

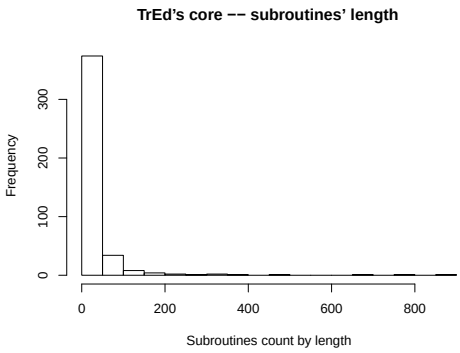


Figure 2.5: TrEd's core source code: subroutines' length

	Complexity	Length
Min	1.00	1.00
1st Q	3.00	8.00
Median	6.00	15.00
Mean	12.96	33.46
3rd Q	13.00	30.00
Max	385.00	866.00

Table 2.3: TrEd's core code overview

initializations of TrEd and bTrEd or dialogs used in TrEd. These subroutines are good candidates for refactoring – they can be split into smaller subroutines and possibly moved outside of TrEd to packages of their own (especially the dialog creating subroutines).

Modules

When we compare TrEd's modules with its core files, we can see in Table 2.5 that the code in modules is more well-behaving. Only 122 subroutines out of 739 (16.5%) exceeds the cyclomatic complexity of 10. As we see from the distribution of subroutines' length and complexity (Figures 2.6 and 2.7), most of them are

Subroutine name	Source File	Lines of code
startMain	./tred	797
startMain	./btred	665
printDialog	./tred	361
filelistDialog	./tred	323
initSidePanel	./tred	306
openFile	./tred	214
editStylesheetDialog	./tred	213
findNodeDialog	./tred	177
macrolistDialog	./tred	153
createCanvasBindings	./tred	147

Table 2.4: Longest subroutines in TrEd's core

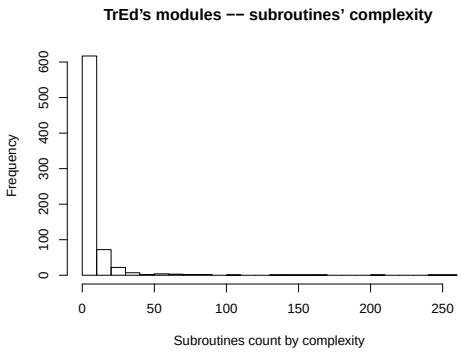


Figure 2.6: TrEd's modules: subroutines' complexity

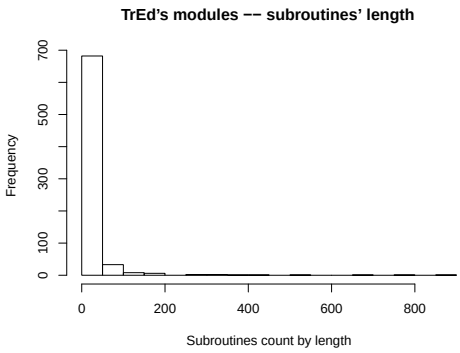


Figure 2.7: TrEd's modules: subroutines' length

	Complexity	Length
Min	1.00	1.00
1st Q	1.00	5.00
Median	3.00	9.00
Mean	8.22	23.55
3rd Q	7.00	19.00
Max	260.00	863.00

Table 2.5: TrEd's modules code overview

quite short and simple (half of the subroutines is shorter than 9 lines of code).

The list of longest subroutines in TrEd modules is presented in Table 2.6. We can see that `TrEd::TreeView` and `Tk::TrEdNodeEdit` modules contain at least two of the longest packages. Especially the redraw function was not split because additional function calls on every redraw can be costly and make the response of TrEd unnecessarily longer. At least some refactoring could, however, improve the readability and maintainability of these code areas.

Subroutine name	Source File	Lines of code
redraw	TrEd/TreeView.pm	674
_populate_extension_pane	TrEd/Extensions.pm	503
print_trees	TrEd/Print.pm	433
draw_canvas	Tk/Canvas/SVG.pm	399
draw_canvas	Tk/Canvas/PDF.pm	332
set_config	TrEd/Config.pm	320
recalculate_positions	TrEd/TreeView.pm	280
add_member	Tk/TrEdNodeEdit.pm	278
add_buttons	Tk/TrEdNodeEdit.pm	194
Tk::Widget::TrEdNodeEditDlg	Tk/TrEdNodeEdit.pm	179

Table 2.6: Longest subroutines in TrEd's modules

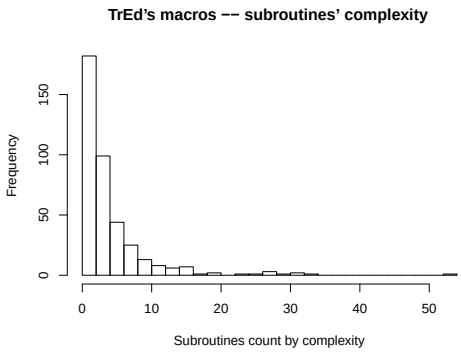


Figure 2.8: TrEd's macros: subroutines' complexity

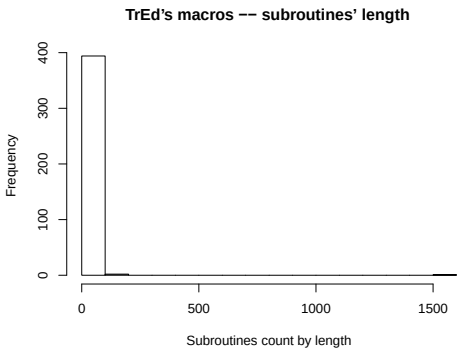


Figure 2.9: TrEd's macros: subroutines' length

	Complexity	Length
Min	0.00	0.00
1st Q	1.00	4.00
Median	3.00	7.00
Mean	4.48	14.01
3rd Q	5.00	11.00
Max	53.00	1502.00

Table 2.7: TrEd's macros code overview

Macros

Only 10% (38 out of 375) of subroutines in macros exceeds the cyclomatic complexity of 10. The subroutines in macros are shorter both on average and in extreme values. As we can see in Figures 2.2 and 2.3, the subroutines in macros are usually the shortest and least complex in TrEd. If we don't count the declarative menu structure macro, the longest subroutine is only 149 lines long and almost all macro subroutines are shorter than 100 lines.

Extensions

The amount of code in 29 extensions counts up to more than 160,000 lines of code. Almost one third of the code mass is located in PML Tree Query extension. The second largest extension is TectoMT extension with more than 20,000 lines of code. Some parts of the source code are, however, generated decision trees or grammars.

The Tree Query extension, which is a graphical client for PML Tree Query[16] is a client for Tree Query search tool, which supports many kinds of linguistically annotated treebanks. The code for this extension is more than 50,000 lines long and functionally ranges from custom toolbars for TrEd to network communication client and SQL evaluation engine.

The TectoMT[17] extension provides basic libraries for TectoMT-based applications in TrEd. TectoMT is a modular software system primarily aimed at machine translation. The code of this extension is more than 20,000 lines long

[16] see also http://ufal.mff.cuni.cz/∼pajas/pmltq/
[17] see also http://ufal.mff.cuni.cz/tectomt/index.html

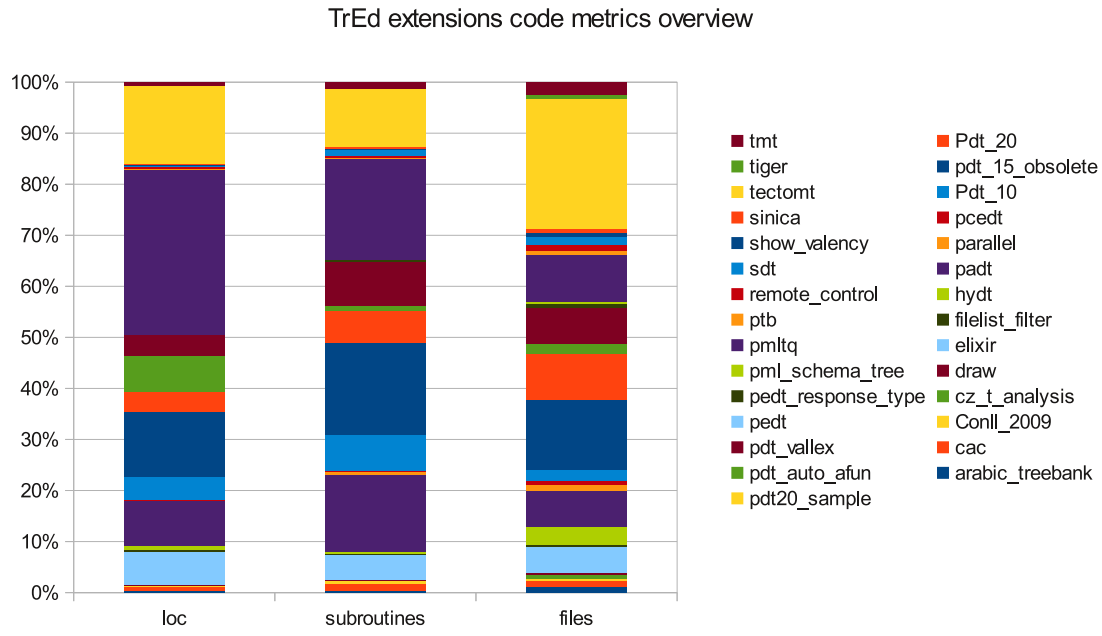


Figure 2.10: TrEd's extensions – code metrics overview

and contains libraries for natural language processing tasks, e.g. segmentation, tokenization, tagging, parsing, etc.

The relative amount of code of extensions is show in Figure 2.10 and Table 2.8.

The size of the Treex::PML library is listed here mainly for illustrative purposes. This library used for manipulating XML data was extracted from TrEd before the work on this thesis began.

2.2.3 Perl::Critic

`Perl::Critic` is a highly configurable static code analysis engine which uses rules to uncover weak spots in Perl source code. The rules applied to source code can be enabled, disabled, added or removed according to developer's need. Most policies used by `Perl::Critic` are based on Damian Conway's book Perl Best Practices [2] and are explained thoroughly in Chapter 4. This module has been developed by Elliot Shank and it is available on CPAN. The warnings produced by this Perl module are divided into five categories by their importance or severity. The range spawns from cosmetic and style-related hints to rules that prevent introducing serious bugs.

This tool is aimed at measuring the quality of code; it should help Perl programmers to write more reliable, efficient and robust code.

We use the same division of source code, as in the previous section, that is TrEd's core files, modules and macros. Since the refactoring of extensions was not a part of this thesis, we present the figures for them only for informative purposes. The same applies to `Treex::PML` library, too.

The most frequent `Perl::Critic` warnings from TrEd's source code were the mild ones with low severity. If we filter out only those with severity 3-5, we

Extension name	LOC	Subroutines	Files
arabic_treebank	530	12	3
cac	1321	54	3
conll_2009	321	16	1
cz_t_analysis	212	4	2
draw	60	4	1
elixir	10661	183	13
filelist_filter	603	8	1
hydt	1263	15	9
padt	14415	556	18
parallel	263	17	3
pcedt	159	7	2
pdt_10	7228	258	5
pdt_15_obsolete	21186	669	35
pdt20	6190	226	23
pdt20_sample	0	0	0
pdt_auto_afun	11785	39	5
pdt_vallex	6465	312	18
pedt	0	0	0
pedt_response_type	265	14	2
pml_schema_tree	13	0	1
pmltq	52920	734	23
ptb	507	7	2
remote_control	343	12	3
sdt	536	42	4
show_valency	182	11	2
sinica	344	12	2
tectomt	24975	416	65
tiger	108	6	2
tmt	1157	44	6
total	164012	3678	254

Table 2.8: TrEd extensions code metrics overview

can create a list of most common pitfalls of TrEd's source code, see Table 2.9. Total number of violations divided by the severity of violations can be seen in Table 2.10. By average, there is approximately 0.31 violations of `Perl::Critic` rules in original TrEd.

The most common warnings are caused by not using proper flags when using regular expressions, not using final return in subroutines and using package variables frequently. Adding requested flags to regular expressions would cause more harm than good, because it changes semantics of these expressions and it would therefore require to rewrite the regular expressions cautiously.

Adding final return to subroutines, on the other hand, can be useful, because it makes the return value of subroutines well defined (in a sense that we know what can be the returned value, even though undefined value as a special case can be returned). Returning `undef` value explicitly is another common idiom found in TrEd. Using it can lead to hard to spot bugs, and should be avoided. Elimination of these warnings is a good refactoring suggestion.

Using package and global variables is a big problem in TrEd. The main `tred` file contained more than 70 package variables (which were global in the whole program). A lot of package variables are exported from other modules, too. This situation allows for distant changes in these modules and the main application, creates very complex relationships between modules and can be very hard to track down. Reducing the number of global variables in TrEd is another good refactoring proposition.

Adding `strict` and `warnings` pragmas is a very good idea, because it helps to identify and track down bugs. Several bugs were found just by enabling these pragmas (e.g. a typing error where scalar variable was used instead of an array).

Some of the warnings, however, can not be avoided. String `eval` have to be used for macro evaluation and some of the sophisticated code requires the `strict` pragma to be disabled. However, these areas should be marked and well-documented in the source code.

2.2.4 CCFinderX

CCFinderX is a graphical tool created for detecting clones in large source code bases. We used this tool to analyse TrEd's source code to find possible code duplicates to make the code less redundant and thus more maintainable. Duplicated code can become a maintenance nightmare, because it requires extra effort to edit duplicated code – one has to make changes in all the spots, where the duplicates appear. Even if maintainers know about duplication, it is easy to forget to alter one of the copies and introduce a subtle bug (see also 2.1.1).

The graphical output from CCFinderX is presented in Figure 2.11. It is a scatter graph which tells us how much each file is similar to all other files and to itself. Every dot in the image means a duplicit code, grey lines divide examined files. The more lines of code a file contains, the larger is the square which represents it. Dots in squares on the diagonal mean that there are duplicates inside the file represented by the square, dots in squares outside the diagonal mean that there exist some common code sequences among files defined by the square's coordinates.

We can see on the scatter graph that there is not much duplication of code in

Policy name	Count	Severity
RegularExpressions::RequireExtendedFormatting	836	3
Subroutines::RequireFinalReturn	745	4
Variables::ProhibitPackageVars	318	3
Subroutines::RequireArgUnpacking	260	4
Variables::ProhibitReusedNames	192	3
CodeLayout::ProhibitHardTabs	177	3
ValuesAndExpressions::ProhibitMixedBoolean-Operators	177	4
ErrorHandling::RequireCheckingReturnValue-OfEval	133	3
ControlStructures::ProhibitDeepNests	105	3
ErrorHandling::RequireCarping	93	3
Subroutines::ProhibitExplicitReturnUndef	74	5
Subroutines::ProhibitExcessComplexity	73	3
TestingAndDebugging::RequireUseWarnings	62	4
BuiltinFunctions::RequireBlockMap	56	4
ControlStructures::ProhibitNegativeExpressions-InUnlessAndUntilConditions	51	3
TestingAndDebugging::RequireUseStrict	48	5
BuiltinFunctions::RequireBlockGrep	47	4
Subroutines::ProhibitSubroutinePrototypes	45	5
NamingConventions::ProhibitAmbiguousNames	43	3
Variables::RequireLocalizedPunctuationVars	42	4
ControlStructures::ProhibitCascadingIfElse	38	3

Table 2.9: Most common Perl::Critic warnings for TrEd

	1	2	3	4	5	total	violations per loc
Core	2,381	2,555	997	464	89	6,486	0.36
Modules	2,077	3,135	1,120	676	171	7,179	0.31
Macros	632	768	212	422	64	2,098	0.22
Total	5,090	6,458	2,329	1,562	324	15,763	0.31

Table 2.10: Code violations by severity in original TrEd

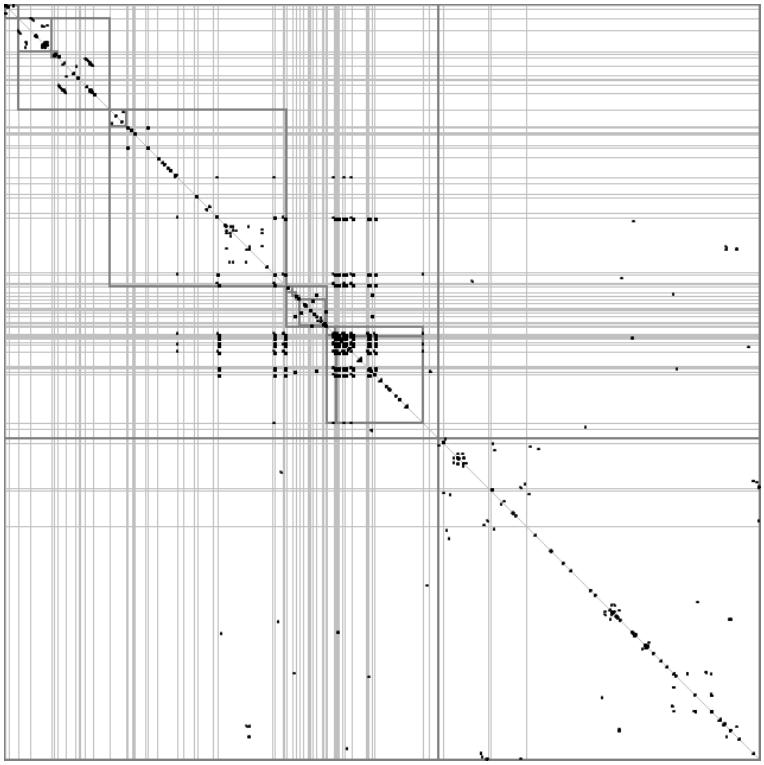


Figure 2.11: TrEd's code duplication

TrEd; the files with highest ratio of similarity among other files are `Tk/Adjuster.pm` and `Tk/Separator.pm`. The zoomed in part of the scatter graph is shown in Figure 2.12. By inspecting the files side by side, we can see (Fig 2.13), that the duplicated code was identified correctly. The code for plotting PDF and SVG canvas has the second largest ratio of similarity and, as can be easily seen by inspecting these files, they really share a lot of common code.

Removing duplicates is one of the most basic method of refactoring. Putting common code is vital for future changes in the program. Besides these two largest duplicates, there are also smaller duplicated chunks of code scattered among the source code of TrEd. We identified these and tried to eliminate them as much as possible (e.g. subroutine `uniq` for filtering duplicit elements from arrays were implemented 4 times in different source files, etc.).

The duplicity which can not be spotted easily by automatic tools like CCFinderX is the duplicity of functions between TrEd and bTrEd. Both these programs use the common core features, but in some other aspects, they differ considerably. While TrEd has to take care of managing graphic user interface by making use of Tk library, bTrEd, on the other hand can be used as a server and includes code for running batch scripts easily.

2.3 Dynamic Code Analysis

Dynamic code analysis is the analysis of code obtained by running the application. For Perl, we used a tool available on CPAN: `Devel::NYTProf`. `Devel::NYTProf` is a powerful, fast, feature-rich Perl source code profiler [1].

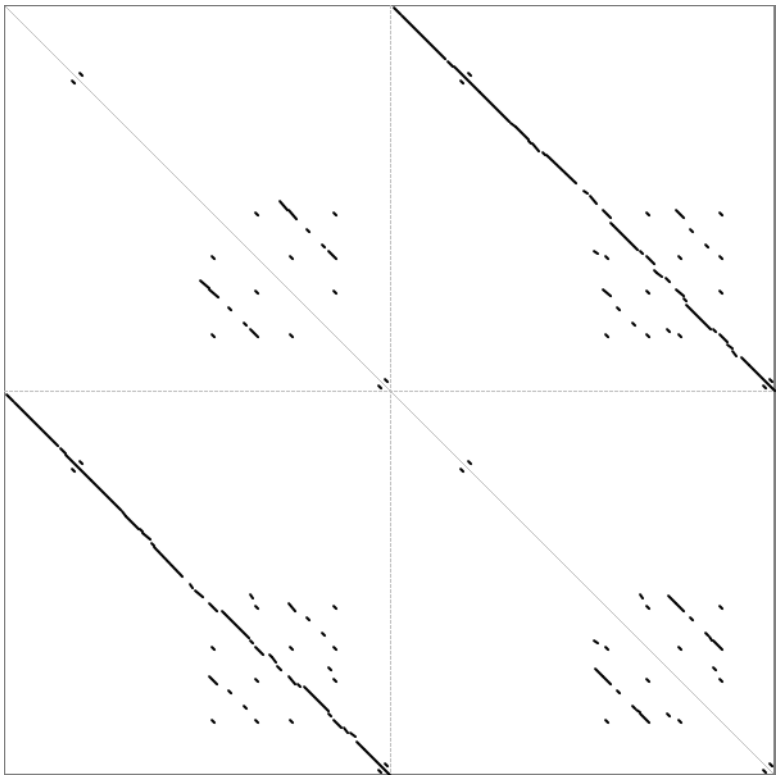


Figure 2.12: The duplication of code between Tk::Adjuster and Tk::Separator

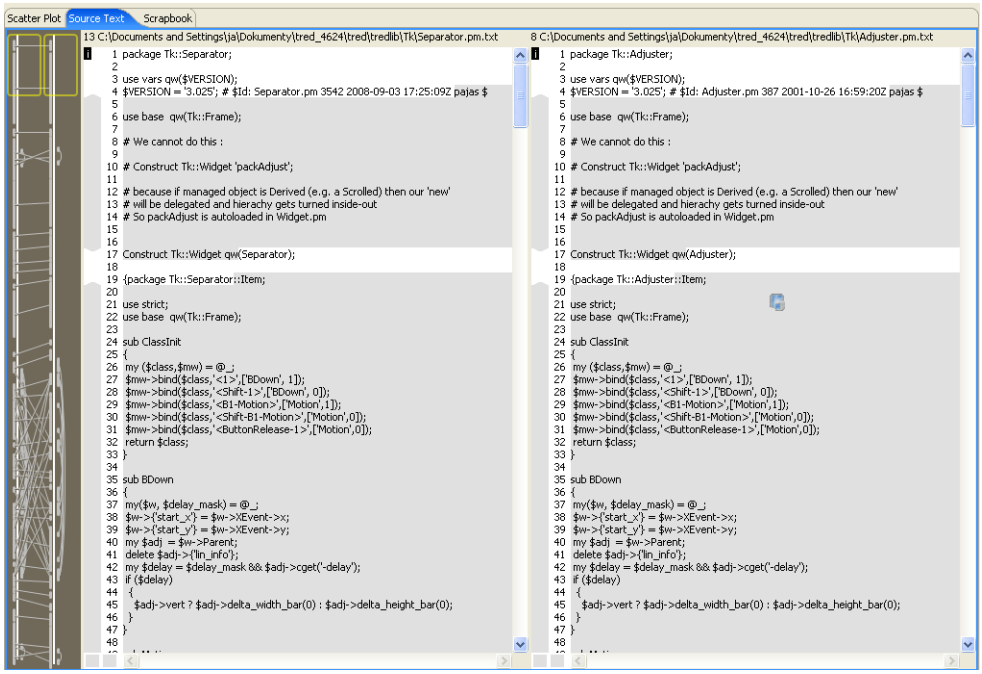


Figure 2.13: The duplication of code between Tk::Adjuster and Tk::Separator, side by side comparison

Profilers can measure the time spent executing various parts of code in computer programs (lines of code, subroutines, blocks of code) and call counts of subroutines. It is used to identify bottlenecks of programs and possibly estimate execution times of computer programs. [7] refers to a research paper by Donald Knuth who found out that usually less than 4 percent of a program accounts for more than 50 percent of its runtime. It is therefore important to identify the bottlenecks correctly and not waste time optimizing code that has only little effect on program performance. Making assumptions about the speed of execution of a program based on unverified claims usually leads to wrong results.

Since TrEd is a complex linguistic application that runs relatively small parts of code code repeatedly for every node of a tree, it is important to be efficient and have as little overhead as possible.

During the refactoring, it is important not to slow down the execution time of TrEd and its batch counterpart, bTrEd. We chose to examine three model situations for purposes of dynamic analysis:

1. simple bTrEd script,
2. start-up of TrEd with one sample file loaded,
3. browsing PML trees in TrEd.

The chosen btred script does not require any user interaction, it is a batch script from btred's user manual that runs for every node in every tree in 10 files from Prague Dependency Treebank 2.0 (PDT 2.0[18]) sample files.

The second model situation measures the start-up time of TrEd. The start of TrEd was measured because a lot of complex initializations happen during the start of TrEd (i.e. loading macros, stylesheets, creating key bindings, etc.) and during loading files.

The third model situation begins by starting TrEd and then browses through 50 trees in 2 files, showing multiple trees at once and displaying information about nodes in the side panel. Browsing between trees in a file is probably the most common operation performed in TrEd, therefore we want to be sure that during the refactoring, TrEd's performance was not influenced negatively.

For the first two model situations, the time spent executing the scripts is probably more important than the number of called subroutines. The third model situation uses Tk timers to perform actions at specified time, therefore measuring time is of little importance there.

The extensions installed during these experiments were

1. Prague Dependency Treebank 2.0 Sample Data,
2. Prague Dependency Treebank 2.0 Annotation,
3. PDT-ValLex Editor and
4. PML Tree Query Interface for TrEd.

[18] see also http://ufal.mff.cuni.cz/pdt2.0/

The first three extensions are used for displaying sample data in TrEd, the last one is used for querying treebanks.

The testing platform used for the evaluation of these tests was a Core i5 M430 workstation with 4GiB of RAM running Kubuntu 11.04. The original TrEd version was 1.4607, the refactored TrEd was a svn checkout of revision 139.

2.3.1 bTrEd Evaluation

The simple script run by bTrEd on 10 sample files is an example script from bTrEd's documentation. It can print five most frequent functors in each processed file and uses hooks[19] to print out the result.

```
#!btred -TNe count()

my %cnt;

sub file_opened_hook { %cnt=() }

sub file_close_hook {
  my @sorted = sort{$cnt{$b}<=>$cnt{$a}} keys %cnt;
  my $filename=FileName();
  $filename=~s/.*\///;
  print "Five most frequent functors in ",$filename,
    ": ",join(" ",@sorted[0..4]),"\n";
}

sub count{ $cnt{$this->{functor}}++ unless $this eq $root}
```

The average time bTrEd spent evaluating sample script was 11.46 seconds. If we filter out foreign module calls, most frequently called code lives in `TrEd::Macros` package. As we can see in Table 2.11, the most of the work in bTrEd is, however, done by `Treex::PML` and `XML::LibXML` libraries which are called very frequently and considerable amount of time was spent in them reading input files and creating structures bTrEd can work with. Creating these structures is also time consuming: as we can see in Table 2.12, only one bTrEd's function is in the list of functions where bTrEd spends most of its time.

2.3.2 TrEd Start

The average start time of TrEd was 4.74 seconds. Except for Tk initializations, most time was spent initializing macros and extensions. From the results of TrEd start analysis, we can see that more than 17% of all the statements were executed and more than 18% of time was spent in `TrEd::Macros` package. The `preprocess` and `initialize_macros` subroutines are (after the Tk initializing subroutines) the most time consuming parts of TrEd's start-up code. That is understandable, because using expression form of `eval` to run macros means that this small blocks of code has to be compiled every time they are encountered. One of the objectives in this thesis is to explore the possibility to remove macros and turn as much of their code into standard Perl code as possible. That way less of the code would be run using `eval` and it can also lead to more transparent code (for more information about removing macros, see Section 5.1).

[19] for more details about hooks, see TrEd documentation and Section 3.14.2

Subroutine	Call count	% of all sub calls
TrEd::Macros::CORE:match	393,900	27.3
Treex::PML::Factory::ANON	63,265	4.4
Scalar::Util::weaken	49,507	3.4
XML::LibXML::Node::DESTROY	44,954	3.1
UNIVERSAL::isa	32,269	2.2
Treex::PML::Struct::DESTROY	25,288	1.8
Treex::PML::StandardFactory::createStructure	25,287	1.8
Treex::PML::Struct::new	25,287	1.8
Treex::PML::StandardFactory::createList	23,588	1.6
Treex::PML::List::new_from_ref	23,573	1.6
Total – 10 most frequent	706,918	49.1
Total	1,441,122	100.0

Table 2.11: Most frequently called subroutines in original bTrEd – simple bTrEd script

Subroutine	Exclusive time [ms]	% of total time
Treex::PML::Instance::Reader::ANON	872	7.6
Treex::PML::Instance::Reader::ANON	864	7.5
XML::CompactTree::XS::_read-SubtreeToPerl	787	6.9
Treex::PML::Instance::Reader::ANON	582	5.1
TrEd::Macros::preprocess	470	4.1
XML::LibXML::Reader::new	274	2.4
Treex::PML::Factory::ANON	270	2.4
XML::LibXML::Reader::nextElement	245	2.1
Treex::PML::Instance::Reader::ANON	235	2.0
Treex::PML::Instance::Reader::ANON	213	1.9
Total – 10 longest	4,812	42.0
Total	11,460.0	100.0

Table 2.12: Subroutines taking the longest time to execute in bTrEd – simple bTrEd script

Subroutine	Call count	% of all sub calls
TrEd::Macros::CORE:match	403,658	61.0
Encode::_utf8_off	20,717	3.1
TrEd::Macros::CORE:readline	20,538	3.1
utf8::CORE:match	15,558	2.4
TredMacro::CORE:match	9,157	1.4
TredMacro::CORE:subst	9,129	1.4
Carp::CORE:substcont	6,209	0.9
XML::LibXML::Node::DESTROY	5,000	0.8
Treex::PML::Factory::ANON (BEGIN)	4,635	0.7
Scalar::Util::weaken	4,234	0.6
Total – 10 most frequent	498,835	75.4
Total	661,347	100.0

Table 2.13: Most frequently called subroutines in original TrEd – start of TrEd

Subroutine	Exclusive time [ms]	% of total time
Tk::update	576.0	12.1
Tk::DoOneEvent	502.0	10.6
TrEd::Macros::preprocess	481.0	10.1
TrEd::Macros::CORE:match	203.0	4.3
TrEd::Macros::initialize_macros	165.0	3.5
Treex::PML::Instance::Reader::ANON	93.8	2.0
Treex::PML::Instance::Reader::ANON	92.8	2.0
TredMacro::_import	88.3	1.9
utf8::SWASHNEW	62.1	1.3
Tk::END	51.8	1.1
Total – 10 most longest	2315.8	48.9
Total	4740.0	100.0

Table 2.14: Subroutines taking longest time to execute in original TrEd – start of TrEd

2.3.3 Browsing in TrEd

To simulate real work in TrEd, we have created a scenario of browsing through two sample files and inspecting 50 trees one after another. This scenario uses Tk timers to invoke functions in TrEd to perform specified actions. First, we visit 5 nodes in each of 25 trees in the first file, then we go back 20 times, but staying at the same depth in the tree. Afterwards, next file in file list is opened and another 25 trees are visited, setting current node to 5 first nodes of each tree.

Most of the work in this scenario is done by `Tk::TreeView` and `TrEd::Macros` modules. The first one takes care of redrawing the trees, the second runs all the hooks and extension code from PDT 2.0 extension. The average running time of this model situation was 67.58 seconds. The average number of executed statements was $21,200,350$ and the average number of subroutine calls was $6,657,664.6$. These numbers are not interesting per se, but we will compare them with results of refactored TrEd in Chapter 5.

3. Design of TrEd

TrEd is a complex graphical editor of tree-like structures. It supports many formats of data input and output, various styles of displaying trees and it is highly customizable to allow for convenient work with numerous tree formats.

This chapter presents an overview of data flow paths in TrEd, what steps are needed to initialize the it, how it works with files, filelists, macros, hooks and stylesheets. To emphasize the internal structure of TrEd, we chose to describe the names of the packages and modules as they appear in TrEd after the refactoring. Many of the modules had been part of `main` namespace before were extracted and separated.

3.1 Overview

The overall view of interaction between modules and the flow of input data can be seen in Figure 3.1. It should be emphasized that the main tred package communicates with almost all other modules. Drawing all the arrows on the diagram would make it less readable, therefore we decided to leave them out. On the top of the diagram we can see the input files. TrEd can open either standalone files directly using `TrEd::File` module, or request opening a filelist through `TrEd::ManageFilelists` module. This module does not open files, it just works with filelists themselves. If the main TrEd application wants to open a file from a loaded filelist, it asks `TrEd::Filelist::Navigation` module to go to desired file number (or name). The `TrEd::Filelist::Navigation` module then asks the `TrEd::File` module to open requested file. At this point, both ways of file opening meet at one point – the `open_file` subroutine. This subroutine then checks whether an autosave recovery version of currently opened file does not exist and asks `TrEd::FileLock` module whether the file is locked. `TrEd::File` module then creates `Treex::PML::Document` object, which is handed over to the currently focused `TrEd::Window` object. `TrEd::Window` class uses `TrEd::TreeView` object to render the tree according to stylesheet chosen either automatically or by user request.

TrEd uses hooks mechanism to run code on various occasions, e.g. when a file is opened or closed, when a node is moved, etc. One of the hooks (`guess_context_hook`) is run every time a file is opened to switch to proper context or annotation mode. The context or annotation mode is in fact the name of Perl namespace in which the macros and hooks are evaluated. New annotation modes are created by new macros and extensions.

The macros and extensions are usually invoked by an event triggered by the user – when a key combination is pressed or if the macro is invoked from TrEd main menu. Macros should communicate with TrEd by using `TrEd::MacroAPI` methods. They can add new functionality by specifying new key bindings and

We can also see minor modes in the bottom of the Figure 3.1. Minor modes specify their own minor hooks, which may be run before or after another hook is run. They are declared, activated or deactivated using `TrEd::MinorModes` package. In the upper right corner we can spot `TrEd::Undo` module, which uses the `Data::Snapshot` package to save current data snapshot for performing undo op-

eration later and save this data to temporary storage of `Treex::PML::Document` object (see also Section 3.7).

3.2 Libraries

TrEd uses many CPAN modules which extend its functionality. The most important of them are:

- `Tk` – toolkit for creating graphic user interface of TrEd
- `PDF::API2` – provides support for printing trees as PDF documents
- `Treex::PML` – provides API for manipulating linguistically annotated treebanks
- `XML::LibXML` and `XML::LibXSLT` – provides an interface to libxml2 and XML parsers
- `Compress::Raw::Zlib`, `Compress::Raw::Bzip2`, `IO::Compress::Gzip`, `Archive::Zip` – adds support for reading and writing compressed files

3.3 TrEd start-up

The initialization of TrEd is a fairly complex task and it is not a good idea to alter the order of steps during this phase. Here we support a brief overview of steps needed to initialize TrEd before starting main entry point of TrEd – `startMain` function.

1. Command line arguments are handled by `Getopt::Long` module. The full list of these can be found in [8].
2. New variable, `$libDir`, is prepended to `@INC` array. It is set to TREDHOME environment variable, if the environment variable is defined. These default locations are tested otherwise (relative to TrEd's script directory):
   (a) `/tredlib`
   (b) `../lib/tredlib`
   (c) `../lib/tred`
3. HOME environment variable is set on Windows platform.
4. Encoding for STDOUT & STDERR is set to utf-8.
5. Runtime user configuration file is located, the configuration is initialized with values read from the configuration file (see also Section 3.8)
6. Recent files are initialized according to configuration
7. Filelists from configuration file are opened (see also Section 3.5)

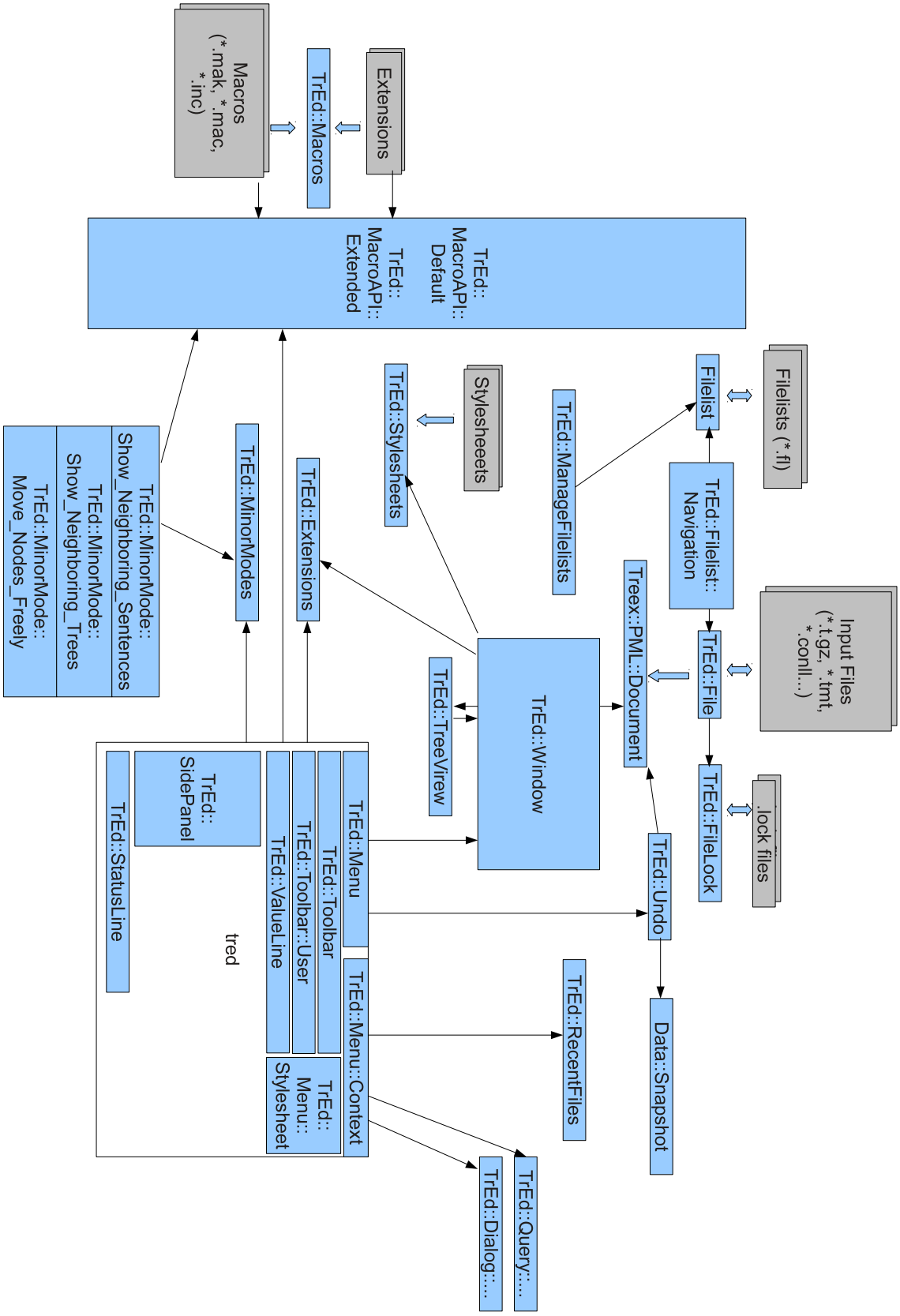


Figure 3.1: TrEd's modules interaction

8. A command to another TrEd instance is passed, if another instance of TrEd is running and TrEd has been started with `-C` start up switch
9. File which contains the process id of currently started TrEd process is written into `.tred.d` directory
10. `Treex::PML` is initialized
11. Stylesheet paths are initialized (see also Section 3.11)
12. Directory with documentation and help files is found
13. Symbol `TRED` is defined for extensions that may still use it (and, possibly, other symbols specified by command line argument)
14. Callbacks called when tree, node, or current node is changed are set
15. Backends for opening various file types are initialized (in TrEd and in Treex::PML library)Section
16. Locale and appropriate charset are set according to configuration options and utf support
17. Standard filelists are opened (see also Section 3.5)
18. `startMain` function is called

The initialization and building of graphic user interface (GUI) continues in `startMain` function. All the steps which require Tk objects (e.g. MainWindow) take place here, as they are created at the beginning of this function. Overview of the initialization steps follows.

1. `Tk::MainWindow` object is created, workarounds for some Tk-specific behaviors are applied.
2. `TrEd::UserAgent`, an `LWP::UserAgent` subclass which provides a GUI Tk-based dialog for asking for credentials is created and handed over to `Treex::PML::IO`
3. Default widget appearance options are set
4. Extensions are prepared, their resource paths are initialized
5. Default stylesheets are loaded
6. Bookmarks are initialized with last action from the configuration file
7. Initial macro context is set (the default is `TredMacro`)
8. Window geometry and fonts are prepared
9. New TrEd::Window object is created, Tk canvas for this object is created (no 7 on Figure 3.2)
10. Main menu is created (no 1 on Figure 3.2)

   (a) Recent files menu is populated with items
   (b) Bookmarks menu is populated
11. Context menu is created (no 3 on Figure 3.2)
12. Value line is created (no 5 on Figure 3.2)
13. Status line is created (no 8 on Figure 3.2)
14. Canvas scale and Minor modes menu are created (numbers 10 and 9 on Figure 3.2, respectively)
15. Toolbar buttons and stylesheet menu are created (numbers 2 and 4 on Figure 3.2, respectively)
16. Macros from extensions are evaluated
17. `initialize_bindings_hook` is run
18. Menus are updated
19. `init_hook` is run
20. Selected stylesheet is applied
21. Side panel is shown, if the user enabled it (numbers 6 on Figure 3.2)
22. Signal handlers are initialized
23. Default (or user-chosen) filelist is loaded
24. `start_hook` is run
25. Macro specified on command line is run, if there was any specified by an argument
26. Main menu bindings are created
27. Tk MainLoop is started.

After the initialization, TrEd is ready for user input. Graphic user interface is show to the user, as we can see on Figure 3.2. This interface consists of several basic elements, from top to bottom they are: the main menu and contexts menu (numbers 1 and 3 on Figure 3.2), toolbar and stylesheets menu (numbers 2 and 4), value line (no 5), side panel (no 6), tree view (no 7), status line (no 8), minor modes menu (no 9) and finally scale slider (no 10). The functionality of these UI elements is covered in [8].

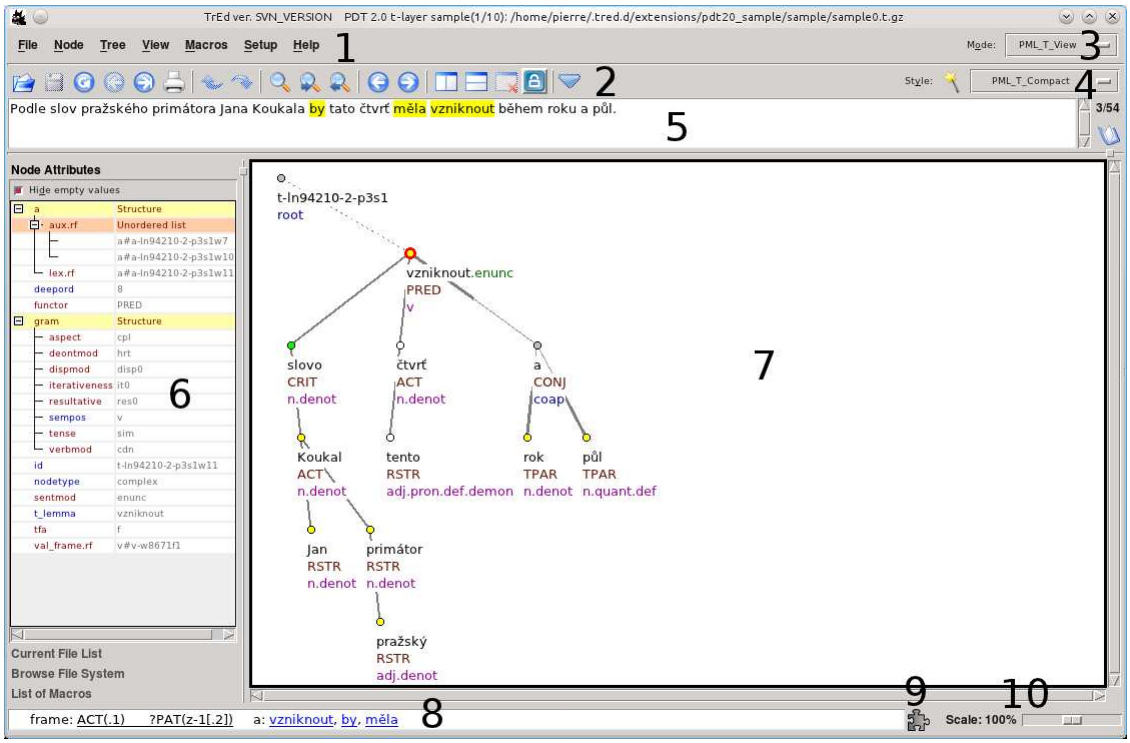


Figure 3.2: TrEd's user interface

3.4 TrEd::File

This package provides basic file opening operations for TrEd – open, close, save files, etc. A file is opened by using `open_file` or `open_standalone_file` functions. Both these functions are based on `load_file` function, which performs the actual opening of specified files. This function also creates a `Treex::PML::Document` object, which is then stored within TrEd::Window as the currently opened file. The `Treex::PML::Document` objects represent a document containing a set of trees that can be accessed via this object. The transformation of file into the tree-like structure is carried out by `Treex::PML` library, which uses multiple backends (subclasses of `Treex::PML::Backend`) to support manipulation with various file types, even compressed or network resources. The `Treex::PML::Documents` can be accompanied by meta data of two types – persistent which are saved when the file is closed and temporary non-persistent data for application purposes.

For the purposes of this module, persistent data which contain information about related files are of great importance. Files loaded by `Treex::PML` library needs an XML schema to be opened appropriately. The schema could contain information about files related to specific `Treex::PML::Document`. These files are loaded by the `open_file` function in this module by default. The relationship between opened files is stored as their non-persistent meta information. Files loaded automatically on behalf of another file are secondary to file which caused them to be loaded. The file, which initiated the loading of related files, on the other hand, is a primary file to all the related files. These relationships can be found out by calling appropriate functions in this module (`get_secondary_files`, `get_primary_files` and their recursive variants).

The non-persistent information in `Treex::PML::Document` is also used to store undo information about the file. For more information about undo functionality, see the documentation of TrEd::Undo module (Subsection 3.7).

When the file is opened, a status is returned. This status is a hash reference, which contains information about whether the opening went without any errors or the error messages and warnings emitted during opening of the file. These statuses are "merged" when multiple files are opened at once to inform about the whole opening operation.

TrEd also uses the *autosave* mechanism. It saves all the opened files every 5 minutes (by default, time interval can be changed in configuration). During opening a file, the `open_file` function checks whether any autosave file exists and if it does, it asks the user whether he wants to recover the file from autosaved copy.

Another mechanism to protect from file losses used by TrEd is file locking. Every time a file is opened, a file lock is created. Then, the lock is unlocked during the closing of each file. The locking of files is an important feature which prevents from inconsistencies in files, e.g. it should protect users from overwriting each other's changes made during editing the same file concurrently. The locking mechanism is described in documentation of `TrEd::LockFile` module (see also Subsection 3.6).

3.5 Filelists

Filelists are just plain text files that optionally contain the name of the filelist on their first line and then one file name on each line. During start-up, filelists are loaded from these locations:

1. `.tred.d/filelists/` directory under user's home directory
2. custom paths added by extensions
3. locations specified by configuration file

3.5.1 Filelist

`Filelist` class allows creating objects which represent TrEd's filelists.

Filelist consists of its name on the first line followed by any number of files, each file on one line. Filelist can also contain patterns for the glob function, which are expanded when the file list is being loaded.

If filelist contains relative file names, files are searched relative to the path where the filelist is stored.

This class supports two types of list items: patterns and files. If a filelist contains a pattern that represents 10 files, then `list()` function returns just the pattern, while `files()` function returns a list of filenames which match aforementioned pattern.

Filelists also supports specifying position in a file exactly by appending a suffix at the end of a file name. The suffixes can be of these forms:

- ##123.4 – points to 4th node in tree number 123

- #a123 – points to first node with id #a123 in the file
- ##123.#a123 – points to first node with id #a123 in tree number 123
- #123.4 – points to 4th node in tree, whose root's form equals to #123

They are supported by `open_file` function in `TrEd::File` module (Section 3.4).

`Filelist` class utilizes lazy loading of filelists. `load()` method marks file to be loaded later and exits. Filelist is then loaded when it is needed, i.e. when any of the functions needs to work with the list.

Filelists are created, saved, modified and accessed using the `Filelist` class (by using functions `new`, `add`, `rename`, `remove`, `save`). For a complete list of subroutines, see the documentation of `Filelist` class.

Functions which provide navigation in filelists are grouped in module `TrEd::Filelist::Navigation`. Probably the most important of the functions is `go_to_file` function which changes the current file from the filelist, opens the desired new file from filelist and updates associated GUI elements. This function is used in all the other file list navigation subroutines: `next_file`, `prev_file`, etc. For more details about subroutines of this module, see the documentation.

The last module which takes care of most of filelist-related operations from GUI is `TrEd::ManageFilelists`. It loads filelist specified in configuration files and standard filelists (stored in `.tred.d/filelists` directory) during TrEd start-up, creates and fills menus for filelists and provides user interface for creating, saving, loading, activating, editing and removing filelists.

3.5.2 TrEd::Bookmarks

Bookmarks are just a special kind of filelists in TrEd. Their difference from filelists is that all the files specified in the bookmark filelist also have the exact position appended to the file name.

The `TrEd::Bookmarks` module also adds support for bookmarking last spot of action in TrEd via calling `last_action_bookmark` subroutine. It is called whenever an attribute of a node is changed, when hooks and macros with undo support[1] are evaluated or when the node release event is triggered.

3.6 TrEd::FileLock

The importance of the locking mechanism was briefly mentioned in Section 3.4. Locks in TrEd are implemented by using lock files. Lock file is a file with the same name as original (locked) file, only suffix `.lock` is appended to its name. Basic information about the lock is written into the lock file: the owner of the lock, the time when the file has been locked, hostname of the computer, process id and the time of the last modifiaction of the locked file. During the opening of file in `TrEd::File` module, the same information is also written to the non-persistent meta data area of locked `Treex::PML::Document` (using function `set_fs_lock_info`) for later comparison with the `.lock` file.

[1] see also Section 3.14 for more information about macros and hooks and Section 3.7 for more information about undo feature

Subroutine `set_lock` locks file given to it as an argument (just creates the lock without asking). More appropriate way of locking a file is by using `lock_file` subroutine, which also check locks before it writes its own lock. If the lock file already exists, it also prompts the user with a GUI dialog, where he can decide whether he wants to steal the lock, open file ignoring locks, or cancel the opening operation.

The lock file, if it was successfully created, can be read by function `read_lock`, which just reads the information stored in the `.lock` file.

The lock file can be checked for consistency with our lock information stored in memory by `check_lock` subroutine. This subroutine can differentiate 12 possible situations[2], which are analyzed during saving files in `TrEd::File` package and during creating new lock file in this module. The user is then prompted whether he wants to save the file and ignore locks, etc.

Locking does not happen for these protocols[3]: "ntred" and protocols specified by `TrEd::Config::noLockProto` configuration variable.

If a file requested for opening is already locked, the lock can be stolen from the original owner, ignored or the opening of file could be cancelled.

3.7 TrEd::Undo

The support for undoing and redoing changes made on trees in TrEd is provided by `TrEd::Undo` module. There are 9 types of undo operations[4], that differ in the type and amount of data they store on the undo stack.

The default undo type is UNDO_DISPLAYED_TREES. Some of the undo types stores only a reference to current node (UNDO_ACTIVE_NODE), while others whole list of trees (UNDO_DATA_AND_TREE_ORDER). The preparation of the undo stack frame to be pushed to undo stack is carried out by `prepare_undo` subroutine. It uses `Data::Snapshot` module to create the snapshot of the data that need to be stored recursively. The undo stack frame is the pushed to undo stack by `save_undo` subroutine. The undo stack is stored in non-persistent meta data in `Treex::PML::Document` object. The undo and redo operations are then performed by `undo` and `re_do` subroutines, which restore saved data from undo stack frames (with help of restore operation from `Data::Snapshot` module), update the undo stack and GUI.

All the situations when undo information is stored are documented in Appendix B.

3.8 TrEd::Config

TrEd is a highly configurable program. It stores its settings in `.tredrc` file in the user's home directory. The search paths for the configuration file as well as the format for this file and all the supported options are thoroughly documented in [8].

[2] for a complete list, see Appendix A

[3] protocol of file can be found out by asking `Treex::PML::IO::get_protocol` subroutine

[4] for a complete list of undo types, see Appendix B

By default, `TrEd::Config` module exports all the variables read from the configuration file (or initialized with their default values) and thus makes them accessible to everybody who includes this module. This behaviour may change in the future and fast accessors can be used to interact with configuration options in a more transparent way.

3.9 Converting

TrEd supports various input and output character encodings. These settings are configurable by editing TrEd's configuration file or command line switches can be used to set input and output encodings. Support for Perl version older than 5.8 is also available. If the Encode module is not present, standard perl `tr///` operator is used to convert between supported encodings[5]. TrEd also supports rendering of Arabic texts with support of `TrEd::ArabicRemix` and `TrEd::ConvertArab` modules.

An old conversion interface provided by `TrEd::CPConvert` module is still available, since it is used in PDT-vallex and PDT15_obsolete extensions.

TrEd also supports loading and execution of macros and extensions written in possibly any encoding. Because every file could have different character encoding, extra care must be taken when loading these files into TrEd.

3.10 Annotation Modes

The annotation mode or contexts are sets of related macros which serve to similar purpose. For every window there is one active annotation mode, which affects the current key bindings, menu bindings and hooks. The user can switch the annotation modes by using the Annotation modes menu in upper right corner of TrEd (no 3 on figure 3.2).

3.11 TrEd::Stylesheet

The purpose of stylesheets in TrEd is mainly to specify, which information should be printed for every node and every edge of the tree and to alter the appearance of trees depending on the data they contain. The patterns from stylesheets are precompiled in `TrEd::TreeView`, thus it is not necessary to compile the patterns for every node or edge of the tree.This code cache is entirely managed by `TrEd::TreeView`. Extensive documentation about the format of stylesheets can be found in [8].

3.12 TrEd::Window

The `TrEd::Window` is a class which holds information about currently displayed file, tree number, currently active node, active annotation mode and stylesheet.

[5] iso-8859-2, ascii, iso-8859-1, windows-1250, windows-1256 and iso-8859-6 are supported if there is no Encode module available

Probably the most important part of this class is the `TrEd::TreeView` class, which draws trees on the canvas.

3.13 Binding System

TrEd uses binding system provided by Tk library and supported by `TrEd::Macros` module. When a key or a combination of keys are pressed, `eval_macro` function is called which then calls `resolve_event` function. This function tries to find a key binding for current annotation mode (or context) via `TrEd::Macros keyBindings` hash. If it does not succeed, it looks for binding in `TredMacro` context and among default bindings defined in `TrEd::Binding::Default` package[6]. The key bindings are one of the two main ways to invoke user macros. They can be, however, be used to invoke any action in TrEd. The standard way of creating bindings in macros were to use preprocessor directive `#bind`, e.g.:

```
#bind my_macro to key Ctrl+A
```

After the refactoring, these can be substituted by calling textttBind() or `TrEd::Macros::bind_macro()` functions.

3.14 Macro System

Historically, macros in TrEd were created for the users to add new functionality to main compiled program easily. As TrEd grew larger, more macros were added around its source code to support various optional features. Macros in TrEd are de facto Perl files, but they can use some special directives, inspired by C preprocessor, to include other macros in their own source code, bind functions to TrEd's events, etc. Their capabilities are documented in great detail in TrEd's user documentation [8]. In the next subsections we describe two types of macros – ordinary macros and hooks and we will also describe packages of macros called extensions.

3.14.1 Macros

Macros in TrEd are ordinary text files which contain a piece of valid Perl source code. Except for the Perl code, they can also contain preprocessor directives like `#include`, `#ifndef`, etc. The macros extend TrEd's functionality and allows user to add powerful extensions which enhance the TrEd program. In this section, the main principles of coping with macros are described.

When loading macros, the first concern is that TrEd has to be able to locate user-defined macros. As many other resources in TrEd, macros are searched relative to `TREDLIB` environment variable, too. By default, macros are looked for in directory tree under `tredlib/contrib`, they were included autmatically by macro `contrib.mac`, which included all other `contrib.mac` files present in subdirectories of contrib directory. This is no longer true in refactored TrEd, since

[6]The default bindings are described in [8]

the macros were moved to main TrEd codebase and transformed into standard Perl routines, namespaces and logical constructs.

The default macro file, `tred.def`, implemented public API for all the macros, hooks and extensions. This API was further expanded by various other macros (node groups macro, ntred macro, minor modes macros, etc.). This API is described thoroughly in [8]. These macro files were turned into these packages:

- `TrEd::MacroAPI::Default`,
- `TrEd::MacroAPI::Extended`,
- `TrEd::NodeGroups` and
- `TrEd::NtredMak`.

The standard minor modes were changed to these modules:

- `TrEd::MinorMode::Move_Nodes_Freely`,
- `TrEd::MinorMode::Show_Neighbouring_Sentences` and
- `TrEd::MinorMode::Show_Neighbouring_Trees`

If it was possible, this code was placed in its own namespace (sometimes it was not possible because it would require rewriting extensions which was not desirable), the rest of the code was put into `TredMacro` namespace, which is the default context or annotation mode in TrEd.

After this change, the support for running macros is still present in TrEd, only some of the standard macros were turned into standard Perl packages. This would allow the extensions to be transformed into standard Perl packages later[7].

The main problem with macro files was their usage of "C-preprocessor-like" directives (`#include`, `#ifdef`, `#endif`) where standard Perl could be used. Some directives can be replaced easily by calling appropriate functions from API (e.g. `#bind`, `#insert`, with some limitations also `#encoding`, for explanation of their meaning, see [8]). This approach increased the complexity not only of TrEd maintenance, but also the complexity of creating user macros.

Macros are usually just Perl packages, their package name activated as binding-context[8] is called "context" or "annotation mode" in TrEd. Depending on the currently active context, some actions in TrEd are performed with the specified prefix, or in specified Perl namespace.

`TrEd::Macro` package is the most important package for macro evaluation. When macros or extensions request a key to be bound, or new menu item to be added, these information are added to `%keyBindings` and `%menuBindings` hashes, under the key which equals to their context name in this package. The package also stores currently defined symbols by using `#define` directive. All of these data structures holding important information about macros can be easily edited and accessed via subroutines in `TrEd::Macro` package.

[7] for more information about the transformation of macros into Perl packages, see also Section 5.1

[8] the package name is turned to TrEd context by using `#binding-context MyPackage` directive, see [8] for details

The macros are loaded by `read_macros` subroutine at the start up of TrEd (see also 3.3). In the original TrEd, this function loaded all the default macros (`tred.def`, `tred.mac`, `contrib.mac`, etc.). In refactored TrEd, only extensions' macros are loaded this way. The loading mechanism, however, remains the same: all the files are read into a big array of lines of code, these are then preprocessed to support all the directives described in [8].

Macros could be invoked in two basic ways:

1. When the key binding associated with macro has been triggered,
2. Via the main menu.

The binding system is described more thoroughly in Section 3.13.

All the macros are invoked by calling `do_eval_macro` subroutine. This subroutine also takes care of initialization of macros, if it has not been done yet. The initialization means that all the loaded macros are evaluated for the first time to be accessible in future `eval`s.

Before the macro is run, actual position in TrEd is bookmarked[9], these macro variables are set: `this`, `root`, `libDir`, `FileNotSaved`, `forceFileSaved`, `Redraw`[10]. These are then accessible in all macros and extensions.

The macro is then evaluated. During the execution, it can change the current node, tree or edit file displayed in TrEd. All this changes are reflected in TrEd and displayed tree can be redrawn according to macro instructions after the macro has returned.

Reloading of macros needs some extra attention, because it uses the Perl internals in a quite unusual way.

3.14.2 Hooks

Hooks are special kind of macros that are executed on specific occasions in TrEd, e.g. when new file is opened or closed, when the tree is redrawn, etc. List of all hooks with their parameters and occasions when they are called is described in great detail in [8].

Hooks differ from macros in the following aspects [8]:

- User cannot choose a name for a hook; on the contrary, hook is recognized as a macro having a special name identifying it as being a certain hook.
- Sometimes parameters are passed to hooks.
- No modifications of the tree or current node are reflected after the hook returns, i.e. the tree is not redrawn, changes to `$this` variable are not reflected. If necessary, a hook must provide this functionality itself.
- Unlike macros, hooks are not expected to modify the tree unless they explicitly state that, typically by calling `ChangingFile(1)`.

[9] see also Section 3.5.2

[10] The meaning and possible values of all the variables accessible in macros and extensions are documented in detail in [8].

Some hooks use undo operation, list of them can be found in Appendix C.

When a hook with undo is run, the actual position in TrEd is bookmarked, `libDir`, `FileNotSaved` and `forceFileSaved` variables are made available for the hook. After the hook is run, no redrawing or changing current position happens.

3.14.3 Extensions

TrEd's extensions are used to pack macros, PML schemas, libraries, stylesheets and other resources together to be easily installable by TrEd users via the Extensions manager.

During TrEd's start-up, paths to all extension, stylesheets and libraries packed with extensions is initialized by `TrEd::Extensions::prepare_extensions`. The macros, which are part of extensions are loaded along with other macros during TrEd's start-up (see Section 3.3).

3.14.4 Minor Modes

Minor modes are small pieces of Perl code, which specify and declare additional hooks – minor pre-hooks and minor post-hooks – which can be associated to virtually any other hook to run before or after it. This way, the minor hooks provide additional functionality regardless of currently selected annotation mode.

The standard minor modes for TrEd includes these three modes:

- Move nodes freely, which allows moving nodes and whole trees around
- Show neighboring sentence, which displays sentences before and after the current sentence in the value line
- Show neighboring trees, which displays neighboring trees in the tree view of current window

Minor modes have been transformed from macros to ordinary Perl modules during refactoring, they does not use macro directives any more.

Minor modes can also add special hooks that can be run before and after the regular hooks. They are called minor prehooks and minor posthooks. Every time a minor mode is created, these hooks are stored in TrEd's main hash called `$grp`. For example the Show neighboring trees minor mode registers `current_node_change_hook`, `node_style_hook` and `get_nodelist_hook` post hooks, which means that every time one of the named hooks is run, the post hook defined in this minor mode is run afterwards. Since minor modes are de facto just TrEd macros, they can be defined by extensions and user macros.

4. Coding style

> *#!/usr/bin/perl APPEAL: listen (please, please); open yourself, wide; join (you, me), connect (us,together), tell me. do something if distressed; @dawn, dance; @evening, sing; read (books,$poems,stories) until peaceful; study if able; write me if-you-please; sort your feelings, reset goals, seek (friends, family, anyone); do*not*die (like this) if sin abounds; keys (hidden), open (locks, doors), tell secrets; do not, I-beg-you, close them, yet. accept (yourself, changes), bind (grief, despair); require truth, goodness if-you-will, each moment; select (always), length(of-days) # listen (a perl poem) # Sharon Hopkins*

Computer programs can be written in many ways and have the same effect, and there is a wide range between beautiful well-documented code and winners of program obfuscation contests. Coding style matters even more in languages like Perl, whose slogan says "There is more than one way to do it" [10] and in which even syntactically correct poems could be written[1]. If everybody used his own style of writing Perl scripts, Perl programs consisting of these scripts would be very difficult to manage, read and maintain by other programmers. According to [11], 80% of software's cost goes to maintenance, so the software should be optimized so it can be easy to read and maintain rather than easy to write.

TrEd started as a small project used for visualization dependency trees in year 2000. It was written by Petr Pajas. Since not only personal style evolves, but Perl evolved, too, TrEd's sources "maps" all these changes in its source codes. Perl version 5.6 was released in March 2000, so it was very fresh and TrEd was written mainly with Perl 5.5 and 5.4 in mind. Since then, unicode support was added and improved several times, new IO implementation was added (perldoc-58-delta), new features were added (e.g.: UNIVERSAL::DOES), regular expression subsystem was revised (perldoc-5100-delta), etc. TrEd was updated for newer Perl version and now supports Perl versions from 5.8 up.

Because of this long evolution, various styles were used when writing TrEd's code. Unifying all these styles is crucial for further TrEd's development and maintenance. Coding style chosen for refactored TrEd is derived from style guidelines and hints from [2].

Perl Best Practices ([2]) is a book which contains many useful tips how to write efficient, maintainable and robust code. Since every programmer writes code according to his experience with languages he learned, coding style is usually a set of coding habits more than anything else and many programmers code just by instinct. When implementing algorithms, they choose the minimal approach: use short names of variables like `x`, `n` or `temp`, use default variable `$_` and global variables instead of passing a function parameter, use the one loop they are used to, etc. Therefore, a unified set of well-thought coding style rules is vital for every program

Set of rules and coding guidelines provides a common mental framework for all the programmers that develop and maintain the program and allows for easier communication and cooperation among team members.

[1] see the quotation above

Brief overview of adapted rules and explanation, which were adopted and why follows.

4.1 Code Layout

Code layout is a basic medium, in which all other coding practices happen. These rules were adopted to unify coding style in the whole TrEd's source code, which is a necessary step when working on bigger projects, especially if more people are involved and everyone uses his own Perl dialect. When developers choose a coding style, it does not really matter, what will it look like exactly. It is important for the rules to be well-thought and consistent. Programmers should stick to the chosen code layout, so they get used to it and could easily understand code and navigate through the code base. The following hints, tips and rules are based on [2]

4.1.1 Bracketing

What to do

Use Kerninghan & Ritchie bracketing style. Place opening brace at the end of control construct, start block of code on the next line and indent it. Finally put closing bracket on a separate line and use the same level of indentation as for the opening control construct.

Why

The reason for this rule is that it minimizes the number of empty lines without any loss of readability. This approach saves one line of code for each control structure if you compare it with putting a bracket on separate line and finding the beginning of the control structure is equally simple as if bracket is put on a separate line.

What about TrEd

This rule was more or less obeyed in original TrEd's source code.

4.1.2 Keywords

What to do

Separate control keywords from the following opening bracket.

Why

Control structures, as their name suggest, control the flow of the program, it is therefore important for them to be visible and stand out in the code.

What about TrEd

This rule was obeyed in original TrEd's source code.

4.1.3 Subroutines and Variables

What to do

Don't separate subroutine or variable names from the following opening bracket.

Why

This rule is essential for the previous to work. One can easily distinguish control construct from a function call visually by using or not using a whitespace to separate the keyword/function name from the opening bracket.

What about TrEd

This rule was obeyed in original TrEd's source code, except for some subroutines prototypes.

4.1.4 Builtins

What to do

Don't use unnecessary parentheses for builtins and "honorary" builtins.

Why

Calling built-in functions without brackets enhances readability of code. It is also easier to distinguish between built in functions and user-defined functions. [2] also encourages not to use parentheses with functions imported from Perl's core packages, which ought to be part of the language, but isn't.

What about TrEd

This rule was not obeyed in original TrEd's source code. Sometimes the parentheses were used, sometimes not, without any specific pattern.

4.1.5 Keys and Indices

What to do

Separate complex keys or indices from their surrounding brackets.

Why

Enhanced readability and less code density.

What about TrEd

Since this rule is not specific which array indices and hash keys are complex, it is up to the feeling of the reader, whether the keys and indices stand out enough.

4.1.6 Operators

What to do

Use whitespace to help binary operators stand out from their operands. Don't use whitespace with unary operators. Treat named unary operators such as `sin`, `cos` as built-in functions (see 4.1.4).

Why

Enhanced readability and less code density.

What about TrEd

This rule was not obeyed in TrEd at all. Binary operators were usually glued together with surrounding expressions and variables.

4.1.7 Semicolons

What to do

Place a semicolon after every statement. Don't place a semicolon after the single statement in `map` and `grep` blocks.

Why

Although this advice may seem trivial, not placing a semicolon after the last statement of the block[2] can cause compilation problems and subtle bugs. After adding another line of code after the "last" line, these two lines becomes one statement which can be hard to spot.

What about TrEd

This rule was obeyed in TrEd.

4.1.8 Commas

What to do

Place a comma after every value in a multi-line list.

Why

The reasoning is similar to 4.1.7. Avoid trivial mistakes.

What about TrEd

This rule was not obeyed in TrEd. Sometimes trailing comma was used, sometimes not.

4.1.9 Line Lengths

What to do

Use 78-column lines.

Why

Old terminals can usually display 80 characters per line, 2 characters is a small safety net.

What about TrEd

This rule was not obeyed in TrEd, lines longer than 100 columns are not rare, even lines with more than 200 columns exist. By shortening the line length to 78 columns using perltidy script, the number of lines of code, however, increased.

4.1.10 Indentation

What to do

Use four-column indentation levels.

Why

The research cited by [2] showed that using four column lines is the best compromise between not wasting horizontal space and comprehensibility of the program.

What about TrEd

TrEd used two-column indentation and mixed spaces with tabs.

4.1.11 Tabs

What to do

Indent with spaces, not tabs.

Why

[2] In Perl, semicolons separate, not terminate statements, so the semicolon after the last statement is not obligatory

Tabs can behave differently in different text editors. They actual indentation would then vary depending on your editor's settings. The problems also arise when code is copied and pasted elsewhere. Thus, spaces are the only reliable choice for indentation.

What about TrEd

TrEd mixed spaces with tabs, usually odd indentation levels used tabs, while even indentation levels used spaces.

4.1.12 Blocks

What to do

Never place two statements on the same line. Not even for `map` and `grep` blocks.

Why

Two or more statements on one line reduces the readability and comprehensibility of both statements. Vertical space is already saved by advice 4.1.1.

What about TrEd

TrEd's source code occasionally used multiple statements on one line.

4.1.13 Chunking

What to do

Code in paragraphs. Each paragraph should contain statements to accomplish one task. Put a one-line comment above each such paragraph summarizing its purpose to enhance maintainability.

Why

According to psychological research cited by [2], humans can only focus on only a few pieces of information at once. Coding in paragraphs allows humans to increase the size of program that fits into short-term memory by enlarging the basic unit, basic piece of information from a line of code to a single paragraph.

What about TrEd

This rule was not obeyed in TrEd.

4.1.14 Elses

What to do

Don't cuddle an else.

Why

Misaligned, cuddled version of else is harder to spot, thus not cuddling it enhances readability of if-else control structure. This way the else branch is more vertically and horizontally distinct which improves the identifiability of the keyword.

What about TrEd

TrEd used cuddled version of else in its source code consistently.

4.1.15 Vertical Alignment

What to do

Align corresponding items vertically.

Why

Using tables when initializing non-scalar variables, initializing more scalar variables that relates to the same concept or assigning to hash enhances readability of code.

What about TrEd

TrEd's source code did not obey this rule.

4.1.16 Breaking Long Lines

What to do

Break long expressions before an operator. Indent next lines with the same level as the start of expression to which they belong. Put terminating semicolon on separate line. But only do that for the last expression, in other cases, see 4.1.17

Why

Operator at the beginning of line is unusual and thus it is a signal to the reader to be cautious. Putting the information up front makes it easier to spot. Important things like control keywords and operators should be kept on the left side of line[2].

What about TrEd

TrEd's source code usually did the contrary, i.e. puts the operator at the end of line.

4.1.17 Non-terminal Expressions

What to do

Factor out long expressions in the middle of statements.

Why

Enhanced readability and smaller amount of information per line.

What about TrEd

This approach was not common in TrEd at all. Ternaries and other long expressions were frequently used as subroutines' arguments, inline anonymous subroutines with tens lines of code were part of data structures and function calls.

4.1.18 Breaking by Precedence

What to do

Always break a long expression at the operator of the lowest possible precedence.

Why

Not doing so can easily confuse the reader who could then misunderstand the performed computation.

What about TrEd

TrEd did not obey this rule.

4.1.19 Assignments

What to do

Break long assignments before the assignment operator.

Why

Reasoning is the same as for 4.1.16. For the assignment, it is however probably better to split the expression before the assignment. If the right side of assignment is still too long, use aforementioned advices.

What about TrEd

TrEd did not obey this rule, assignment operator was usually at the end of line.

4.1.20 Ternaries

What to do

Format cascaded ternary operators in columns. Break a series of ternary operators before every colon, aligning the colons with the operator preceding the first conditional.

```
my $dialog_title = $opts_ref->{only_upgrades} ? 'Update Extensions'
                 : $opts_ref->{install}       ? 'Install New Extensions'
                 :                              'Manage Extensions'
                 ;
```

Why

The conditional expressions forms a column as well as the possible results or values. This enhances readability and extensibility of ternary operator a lot.

What about TrEd

TrEd did not obey this rule, ternaries were usually on one line only.

4.1.21 Lists

What to do

Parenthesize long lists. Treat comma in multi-line lists as value terminator, not value separator. Put opening parenthesis on the same line as the beginning of the statement, indent list elements and put closing parenthesis on separate line at the same indentation level as the preceding statement.

Why

It is important to visually distinguish multi-line lists initialization from other language constructs (especially because comma has different meaning in scalar context, so it can be easily confused).

What about TrEd

TrEd did obey this rule regarding the use of parenthesis, the indentation, however, differs.

4.1.22 Automated Layout

What to do

Enforce your chosen layout style mechanically. Use perltidy.

Why

Since people are not perfect at tedious tasks, use computer to format code for you (at least whenever it is possible).

What about TrEd

TrEd used Emacs as a formatting tool. However, this approach caused mixing of spaces and tabs, which is not desirable.

4.2 Naming Conventions

Since Perl borrows syntactic (and also semantic) parts of the language from other languages, e.g. C, awk [10], the naming conventions for Perl also comes from these languages. Therefore, it is more common for Perl programs to use similar naming conventions as C programs than to use naming conventions as Java or C# programs.

The reasons for introducing naming conventions are roughly the same as for agreeing on consistent code layout (see 4.1). Good naming convention reduces the effort to read and understand programs [4].

There are two aspects of consistent naming of variables and subroutines: syntactic and semantic. The syntactic rules tells us that if we choose to use `$noun_adjective` name for variables, we should use the same pattern for all of them. We should not mix using underscores with using CamelCase notation. At least rules for treating multi-word variables and abbreviations should be drawn up. The semantic rules should tell us to use names that reflect the purpose of the variables and functions and not to use function names like `process_data` or variable names like `$tmp, $x, $y, $foo`, etc.

4.2.1 Identifiers

What to do

Use grammatical templates when forming identifiers, e.g.

- namespace -> Noun::Adjective;
- variable -> [adjective_]*noun,
- lookup_variable -> [adjective_]*_noun_preposition etc.

Reserve one word variable names for variables used only in one block. Try to be specific when creating names.

Why

The hierarchical naming of namespaces helps them being organized (not for Perl itself, but for human readers of the source code). The more specific are the names of the variables, the more easily a mistake can be detected. Specific variable names helps people understand the source code.

What about TrEd

TrEd source code contained a mixture of using underscores and CamelCase notation for both the variables and functions. No grammatical templates were used.

4.2.2 Booleans

What to do

Name booleans after their associated test.

Why

This naming convention makes reading the code more natural and self-explaining at the same time.

What about TrEd

TrEd did not obey this rule for booleans. Usually default behaviors of boolean context and complicated expressions are used more than test-named functions in conditions.

4.2.3 Reference Variables

What to do

Mark variables that store references with a _ref suffix.

Why

A common mistake in Perl is to use reference to array or hash as if it was not a reference, but the dereferenced array or hash. Using _ref with all the references makes it easy to spot the mistake when the reference is misused.

What about TrEd

TrEd did not obey this at all.

4.2.4 Arrays and Hashes

What to do

Name arrays in the plural and hashes in the singular.

Why

The source code can be read naturally. Hash elements are usually accessed individually while arrays are usually processed in loops, `grep` and `map` functions.

What about TrEd

TrEd's source code contained both arrays and hashes with names in singular and plural.

4.2.5 Underscores

What to do

Use underscores to separate words in multiword identifiers.

Why

Spaces and hyphens are not allowed as identifiers. Alternative CamelCase does not scale well to using all capital letters with constants.

What about TrEd

TrEd source code contained a mixture of using underscores and CamelCase notation for both the variables and functions.

4.2.6 Capitalization

What to do

Distinguish different program components by case. Use lowercase letters only for names of subroutines, methods and variables. Use mixed-case for class names and namespaces. Use uppercase for constants

Why

The visual distinction helps with semantic distinction of the code chunks. Enhances understandability of source code.

What about TrEd

Since TrEd sometimes used CamelCase names for variables, this rule was not obeyed either. Uppercase was usually used for constants.

4.2.7 Abbreviation

What to do

Abbreviate it by retaining the start of each word. Leave the trailing s for plural.

Why

Other approaches like leaving out the vowels is usually more difficult to decipher.

What about TrEd

TrEd's source code usually obeys this rule.

4.2.8 Ambiguous Abbreviations

What to do

Abbreviate only when the meaning remains unambiguous.

Why

Otherwise the code can be read in more semantically distinct ways, which introduces confusion.

What about TrEd

Since no special attention was paid when naming variables, names like `grp` or `msg` were not rare. Ambiguous abbreviations of variables, e.g. `filename` abbreviated to `f` or `protocol` to `proto` were common.

4.2.9 Ambiguous Names

What to do

Avoid using inherently ambiguous words in names.

Why

Introduces source code which is open to multiple interpretations. Can be, however, hard to spot by the writer of the original code.

What about TrEd

Names like `$last_tree` or `$no_secondary` were quite frequent.

4.2.10 Utility Subroutines

What to do

Prefix "for internal use only" subroutines with an underscore.

Why

This rule conforms to the syntactic "heritage" of C language. Subroutines that are never be exported, because they are used to simplify or augment implementation of a module or class. Since these subroutines are visually distinct, one can easily see when it was used outside the package it was defined and automatic code analysis tools can warn against such uses of "private" functions.

What about TrEd

TrEd did use this convention, but not thoroughly. For example subroutine that sorts strings according to number of underscores was named `_u_sort()`.

4.3 Values and Expressions

4.3.1 String Delimiters

What to do

Use interpolating string delimiters only for strings that actually interpolate. For a sequences of strings on following lines choose one delimiter and use it for all of them.

Why

Following this rule avoids unintentional interpolations and prevents unnecessary errors.

What about TrEd

In TrEd's source code interpolating string delimiters were used for non interpolating strings, too.

4.3.2 Empty Strings

What to do

Don't use `""` or `''` for an empty string. Use `q{}`.

Why

It is a visually non-ambiguous way use an empty string.

What about TrEd

TrEd usually used interpolating quotes.

4.3.3 Single-Character Strings

What to do

Don't write one-character strings in visually ambiguous ways. Use `q{ }` for a single space and interpolated tab for tabulators (`"\t"`).

Why

Reduce ambiguity and subtle errors.

What about TrEd

TrEd's source code used both variants of quotes for single-space strings.

4.3.4 Escaped Characters

What to do

Use named character escapes instead of numeric escapes.

Why

Using named special characters makes the code more readable.

What about TrEd

Numeric escapes were used only for some of the Arabic character classes in regular expressions.

4.3.5 Constants

What to do

Use named constants, but don't `use constant`, use `Readonly` module instead.

Why

Using numeric constants in code is confusing and cryptic, making less obvious what the code is trying to do. Naming constants improves the level of abstraction and the readability of the code. Constants created by `use constant` does not interpolate in strings, they are treated as barewords anywhere a string is expected (which can introduce subtle bugs) and they can not be created at runtime or lexically scoped. Instead, they are created at compile time and they are package scoped.

What about TrEd

Using raw numbers was fairly common practice in TrEd's source code, as well as using `constant` pragma.

4.3.6 Leading Zeros

What to do

Don't pad decimal numbers with leading zeros. If you intend to use octal numbers, use built-id `oct()` function.

Why

Any integer that begins with zero is treated as an octal number in Perl. Using zero padding for octal numbers makes the code less self-explaining and more difficult to maintain.

What about TrEd

Numbers with leading zeros were used for changing permission for files in TrEd.

4.3.7 Long Numbers

What to do

Use underscores to improve the readability of long numbers.

Why

Using optical grouping of "thousands" improves readability of numbers and it is much easier to check them for correct number of zeros, e.g. it is much harder to count the number of zeros in `100000000000` than it is in the same number with underscores: `100_000_000_000`.

What about TrEd

No long numbers were used in TrEd's source code.

4.3.8 Multi-line strings

What to do

Lay out multi-line strings over multiple lines. Use concatenation instead of implicit newlines.

Why

Implicit newlines usually break indentation level of code, which is always a bad practice. Splitting strings on newline when dealing with long strings is usually the most natural way for the reader to read the string.

What about TrEd

TrEd's source code usually obeys this rule.

4.3.9 Here Documents

What to do

Use a heredoc when a multi-line string exceeds two lines.

Why

It is not efficient for the programmer, nor easy to read for the maintainer to use the concatenation approach with strings longer than several lines.

What about TrEd

In TrEd, longer strings, such as documentation or help information within code were usually written as heredoc.

4.3.10 Heredoc Indentation

What to do

Use a "theredoc" when a heredoc would compromise your indentation. Put heredoc in a `Readonly` constant or a subroutine if it needs to interpolate variables.

Why

This approach improves the readability of the program. If the heredoc is put into a subroutine, only a small part of the code will loose clear indentation.

What about TrEd

TrEd's sources did not obey this rule, heredoc was usually in the middle of the code.

4.3.11 Heredoc Terminators

What to do

Make every heredoc terminator a single uppercase identifier with a standard prefix.

Why

Using a terminator that stands out in the code makes heredoc less tough to understand and read. Standard prefix for the terminator, such as "END_", makes it more obvious after the heredoc when compared to using different terminator each time.

What about TrEd

`EOF` or similar abbreviations were used to terminate heredoc.

4.3.12 Heredoc Quoters

What to do

When introducing a heredoc, quote the terminator.

Why

Since heredoc is not used very often, according to [2], default interpolation behaviour without quotes is not familiar to most Perl programmers. Using the same conventions as for strings makes the code more understandable and the intent of the programmer is more clear.

What about TrEd

TrEd's source code contains approximately one third of the heredoc terminators without quotes.

4.3.13 Barewords

What to do

Don't use barewords.

Why

In Perl, a bareword is name that has no other interpretation in the grammar (such as subroutines or file handles). These will be treated as if it were a quoted string [10]. Thus, introducing a new unrelated function or file handle can change the semantics for a bareword with the same name. Barewords are error-prone and should be avoided.

What about TrEd

TrEd's source code did not use barewords often.

4.3.14 Fat Commas

What to do

Reserve `=>` for key-value or name-value pairs when creating hashes, constants or passing named arguments to functions.

Why

Using "fat comma" instead of regular comma reinforces the connection between the name/key and value pair. This improves the logical structure of the code and thus also its the readability.

What about TrEd

Fat commas were used in TrEd's source code for creating name-value and key-value pairs.

4.3.15 Thin Commas

What to do

Don't use commas to sequence statements.

Why

There are two roles of commas in Perl. In scalar context it evaluates its left argument in void context, throws that value away, then evaluates its right argument in scalar context and returns that value. In list context, a comma is just the list argument separator, and inserts both its arguments into the LIST.

It does not throw any values away [10]. To avoid confusion and problems, it is useful to use comma only as a list separator.

If a sequence of statements needs to be treated as a single statement, use `do` block.

What about TrEd

Thin commas were rarely used in TrEd as a sequencing operator.

4.3.16 Low-Precedence Operators

What to do

Don't mix high- and low-precedence booleans. Avoid `and` and `not` and reserve `or` for specifying fallback for builtins.

Why

Logical operators in Perl can be written in C-fashion, i.e. `&&, ||, !` or in English words (`and, or, not`). The C-style operators have, however, higher precedence than the more verbose English operators. To avoid confusion and a source of potential problems when these two types of operators are mixed, avoid using both types in the same expression. To increase the comprehensibility, use C-style operators.

It is also not clear, whether the change of precedence was intentional or it is a bug. Logical "and" has higher precedence than "or". But if you use `||` operator, its precedence is higher than the precedence of `and` operator, which can be confusing. The code snippet below illustrates this problem:

```
while ( $A and !$B and !$C || $D )
{
    ...
}
```

If we use brackets to indicate the precedence explicitly, the original evaluation works like this: `$A && !$B && (!$C || $D)`. If we would use operators only of one type, the expression would mean `($A && !$B && !$C) || $D`.

What about TrEd

TrEd's source code used and mixed these operators fairly often. Some of the bugs in original TrEd were caused by this approach.

4.3.17 Lists

What to do

Parenthesize every raw list.

Why

Since the precedence of comma is even lower than the precedence of assignment, one can easily create sequence of commands instead of creating a list. It is thus safer to use parenthesis when creating lists.

What about TrEd

TrEd followed this rule.

4.3.18 List Membership

What to do

Use table-lookup to test for membership in lists of strings; use `any()` for membership of lists of anything else.

Why

Function `any()` from CPAN module `List::MoreUtils` is similar to `grep`, but it returns as soon as its test block succeeds, which can save time when searching through lists linearly (although the worst time for the operation remains the same). Hashes are ideal for lookups, because the complexity of looking up a key in hash is much lower than searching through a list of values.

What about TrEd

TrEd used `grep` function for linear search quite often. Sometimes also `first()` function from `List::Util` or own implementation of the same function were used. Hashes were used for testing membership.

4.4 Variables

4.4.1 Lexical Variables

What to do

Avoid using non-lexical variables.

Why

Using non-lexical variables is similar to using global variables. They can be changed from any package and change the behaviour of the whole program. Global variables create a link between otherwise unrelated pieces of code in a way that all these code pieces can subtly interact with each other using non-lexical variables. You can be never sure if some other routine you call does not change the value of the variable and makes your program to fail.

What about TrEd

TrEd used many non-lexical variables. In the process of the refactoring we tried to get rid of as many of these as possible.

4.4.2 Package Variables

What to do

Don't use package variables in your own development. Lexical variables are a much better choice. And if they need to be accessed outside the package, provide a separate subroutine to do that.

Why

When using package variables for saving the state of the module, you can be never sure that other code that uses the package/module won't corrupt its internal state. Using lexical variables and appropriate accessor methods is a safer choice.

What about TrEd

TrEd used package variables in its source code.

4.4.3 Localization

What to do

If you're forced to modify a package variable, localize it.

Why

Using `local` forces the package variable change to be effective only in the current scope, thus it does not affect the rest of the program and all other uses of the package. This helps with maintainability of the program.

What about TrEd

TrEd's source code localized package variables frequently, but not always.

4.4.4 Initialization

What to do

Initialize any variable you localize. Even if you specifically want localized variable to be undefined, it's better to say so explicitly.

Why

Using a local operator on a global variable gives it a temporary value each time local is executed. When the program reaches the end of that dynamic scope, this temporary value is discarded and the original value restored [10]. But whenever a variable is localized, its value is reset to `undef` [2]. It is therefore necessary to initialize its value even desired value for the localized global variable should be undefined to make the intention of the programmer clearer.

What about TrEd

TrEd's source code leaves at least half of the localized variables uninitialized.

4.4.5 Punctuation Variables

What to do

`use English` for the less familiar punctuation variables.

Why

Since punctuation variables can not be avoided completely, at least the less common ones should be named explicitly by using the `English` module. This makes the code less error-prone and improves readability of it greatly.

What about TrEd

`English` module was not used in TrEd.

4.4.6 Localizing Punctuation Variables

What to do

If you're forced to modify a punctuation variable, localize it.

Why

Since all punctuation variables are global, the reason for localizing them is the same as for 4.4.3. Following this rule helps with maintainability of the program because it is much harder to affect other packages and subroutines if the global variables are localized.

What about TrEd

Punctuation variables were not modified in TrEd's core.

4.4.7 Match Variables

What to do

Don't use the regex match variables: `use English qw( -no_match_vars );`. Use `Regexp::MatchContext` CPAN module instead of using standard match variables.

Why

Not following this rule causes every regular expression to remember the pre-match, match and post-match substrings which slows down matching using regular expression in the whole program (since the `$'` and `$&` variables are global and affect all the program). `Regexp::MatchContext` enables the program to use pre-match and post-match without the performance penalty imposed by the standard match variables.

What about TrEd

Neither `English` module was used in TrEd, nor special match variables.

4.4.8 Dollar-Underscore

What to do

Beware of any modification via $_.

Why

$_ is often an alias for other variable and used frequently without being named explicitly. Therefore, it is very easy to introduce subtle bugs by changing $_;

What about TrEd

TrEd's source code usually obeys this rule.

4.4.9 Array Indices

What to do

Use negative indices when counting from the end of an array.

Why

It is clearer and less repetitive not to use `$#array` or `@array` in scalar context to count array indices from the end. Negative indices provides a cleaner notation.

What about TrEd

In TrEd, unnecessary final index variable was used occasionally.

4.4.10 Slicing

What to do

Take advantage of hash and array slicing, but avoid array slices with negative range for indices.

Why

Array and hash slices are more natural and intuitive than mindless repetition. Using slices also creates code that is more extensible and less error-prone, since any repetition and copy-and-paste during programming is a potential problem.

What about TrEd

TrEd's source code used array and hash slices quite frequently (when appropriate).

4.4.11 Slice Layout

What to do

Use a tabular layout for slices, but only until both of the lists are shorter than one line.

Why

Following this rule enhances readability of the code.

What about TrEd

This rule was no obeyed in TrEd's source code, but it is useful only for a small amount of code.

4.4.12 Slice Factoring

What to do

Factor large key or index lists out of their slices.

```
Readonly my %CORRESPONDING => (
        age        => 1,
        comments   => 6,
        fraction   => 8,
        hair       => 9,
        height     => 2,
        name       => 0,
        occupation => 5,
        office     => 11,
        shoe_size  => 4,
        started    => 7,
        title      => 10,
        weight     => 3,
    );

    @staff_member_details[ values %CORRESPONDING ]
        = @personnel_record{ keys %CORRESPONDING };
```

(Example from [2]).

Why

This approach is easily comprehensible, readable and scales well. It is also easy to maintain.

What about TrEd

TrEd did not use this approach.

4.5 Control Structures

4.5.1 If Blocks

What to do

Use block if, not postfix if.

Why

Block if structure stand out in the source code much more than postfix if. Postfix if also does not scale well, if the number of consequent statements increases.

What about TrEd

Almost half of the if control structures in TrEd used postfix notations.

4.5.2 Postfix Selectors

What to do

As an exception to rule 4.5.1, reserve postfix if for flow-of-control statements (`return`, `next`, `last`, `croak`, `redo`, `goto`, `die`, and `throw`).

Why

The fact, that these commands may interrupt the control flow, it is important for them to be as visible as possible and to appear as left as possible [2].

What about TrEd

As for rule 4.5.1, postfix if was used in TrEd frequently, not only for flow-of-control statements. There does not seem to be a specific pattern or rule when postfix if and when block if was used in TrEd's source code.

4.5.3 Other Postfix Modifiers

What to do

Don't use postfix `unless`, `for`, `while`, or `until`.

Why

Postfix modifiers are harder to maintain, less clear and understandable and does not scale well.

What about TrEd

TrEd's source code used postfix modifiers frequently.

4.5.4 Negative Control Statements

What to do

Don't use `unless` or `until` at all.

Why

Positive control statements are more familiar and comprehensible to most of the developers. Negative control statements does not scale well – when the condition contains two or more elements or if the condition is negative itself.

What about TrEd

`unless` was used frequently in TrEd's source code, `until` very rarely (appeared only twice).

4.5.5 C-style Loops

What to do

Avoid C-style for statements.

Why

Using `while` or `foreach` loop makes the code easier to comprehend and maintain [2].

What about TrEd

TrEd's source code contained only a few C-style for loops, there were 17 of them in main tred file.

4.5.6 Unnecessary Subscripting

What to do

Avoid subscripting arrays or hashes within loops.

Why

Iterating through indices and then repeating array access in loop is not only less effective, it is also more error-prone and harder to maintain (due to copy-and-paste and off-by-one errors).

What about TrEd

Since C-style loops are rare in TrEd's source code, unnecessary subscripting is not common, too.

4.5.7 Necessary Subscripting

What to do

Never subscript more than once in a loop.

Why

Repeating the same array or hash lookups are computationally expensive and increase maintenance cost if the code should change in the future.

What about TrEd

TrEd's source code usually obeys this rule.

4.5.8 Iterator Variables

What to do

Use named lexical variables as explicit `for` loop iterators.

Why

Using the `$_` variable and taking advantage of its default usage by some of Perl functions makes the code less readable and comprehensible. Therefore the maintainability of the code also suffers, since the name of `$_` is not very descriptive and it can be misunderstood and misused easily in a more complicated context.

What about TrEd

Implicit loop iterator variable was used quite frequently in TrEd's source code.

4.5.9 Non-Lexical Loop Iterators

What to do

Always declare a `for` loop iterator variable with `my`.

Why

Perl always uses private lexical variable for the scope of the loop, even if its name is the same as other variable above the loop in the same scope. Not using `my` keyword explicitly can be confusing and misleading.

What about TrEd

`my` keyword was usually used when declaring for loops in TrEd's source code.

4.5.10 List Generation

What to do

Use `map` instead of `for` when generating new lists from old.

Why

`map` is computationally cheaper than using `push` repeatedly and more readable than preallocating desired array size. It is usually also easier to understand because it is more high-order function.

What about TrEd

`map` function is used frequently in TrEd's source code, not only for generating new lists, but sometimes also for list transformations (see also 4.5.12).

4.5.11 List Selection

What to do

Use `grep` and `first` instead of `for` when searching for values in a list.

Why

Using `first` is more effective than for loop, `grep` is well known to UNIX world programmers, it is shorter, thus easily readable.

What about TrEd

`grep` is used very frequently for selecting values from lists in TrEd.

4.5.12 List Transformation

What to do

Use `for` instead of `map` when transforming a list in place.

Why

`map` allocates new storage for the transformed list while `for` loop can use the storage that is already allocated.

What about TrEd

TrEd's source code contains approximately one third of the heredoc terminators without quotes.

4.5.13 Complex Mappings

What to do

Use a subroutine call to factor out complex list transformations.

Why

Long list of instructions in `map`'s block is hard to read and understand. Using separate function is more scalable and comprehensible.

What about TrEd

In TrEd's source code complex mappings were used quite frequently. At least 30 of them were used in the main tred file.

4.5.14 List Processing Side Effects

What to do

Never modify `$_` in a list functions (`grep`, `map`, `first`, etc.).

Why

It is not usual for `map` and `grep` functions to alter the original list. Since `$_` in these functions holds alias, not a copy of a list element, it can be easily modified unintentionally by all the functions that takes `$_` silently as their default argument.

What about TrEd

Modifying `$_` in TrEd's source code could be a potential source of problems in statements where hash keys could be created in tests like in:

```
grep {$_->{treeNo} == $tree_no}
```

There are several cases where the `$_` is altered by calling another function or substitute `s///` operator in TrEd's source code.

4.5.15 Multipart Selections

What to do

Avoid cascades of `if-elsif-elsif-else` statements wherever possible.

Why

Code with cascading if statements is poorly readable because it very easily spans across several pages of source code. It is also not effective to execute, if the common cases are not listed first.

What about TrEd

Cascading if statements are quite common in TrEd's source code, more than 3 `elsif`s appears almost 30 times in the source code.

4.5.16 Value Swithces

What to do

Use table look-up in preference to cascaded equality tests.

Why

This solution is cleaner and more effective than cascading if chain if the course of actions is chosen by testing one variable against a number of predefined values.

What about TrEd

Tests in TrEd are usually more complex, therefore it is not very feasible to use this approach.

4.5.17 Tabular Ternaries

What to do

When producing a value, use tabular ternaries.

Why

If a single value is produced in a chain of cascading if statements, it is better written as (multiple) tabular ternary operator. This scales well and is easily readable. The computational efficiency is, however, the same as when using cascading if statements.

What about TrEd

Tabular ternaries were sometimes used in TrEd's source code to produce values.

4.5.18 do-while Loops

What to do

Don't use `do...while` loops.

Why

Postfix loop construct are harder to read (see also 4.5.2) and less scalable. Moreover, in Perl, you can't use `redo`, `next` and `last` commands within a `do...while` loop, because it is not really a loop, but rather a modified `do` block.

What about TrEd

Numbers with leading zeros were used for changing permission for files in TrEd.

4.5.19 Linear Coding

What to do

Reject as many iterations as possible, as early as possible.

Why

This approach usually improves comprehensibility and readability of code and is, by definition, more effective.

What about TrEd

TrEd usually uses this approach quite thoroughly.

4.5.20 Distributed Control

What to do

Don't contort loop structures just to consolidate control.

Why

Sophisticated conditional tests that are usually accompanied by several flag variables are difficult to understand. If the loop control is distributed inside the loop when appropriate.

What about TrEd

TrEd uses many difficult conditions and flags, not only for conditions, but also for loop structures.

4.5.21 Redoing

What to do

Use `for` and `redo` instead of an irregularly counted `while`.

Why

Using `for` and `redo` loops reduces the risk of off-by-one errors and forgetting to increment iteration variable. Actually, there is usually no need for an iteration variable when using these loops. `redo` can be used if the loop needs irregular counting.

What about TrEd

Many `while` loops were used in Perl.

4.5.22 Loop Labels

What to do

Label every loop that is exited explicitly, and use the label with every `next`, `last`, or `redo`.

Why

Enhances readability and comprehension, maintainer finds particular loop faster. It is also particularly useful when more nested loops are added throughout the application's lifetime.

What about TrEd

Loop labels were used only when it was necessary because of using loop control statements (`next`, `last` and `redo`).

4.6 Documentation

In the programming world there is a well known saying[3]: "Any code of your own that you haven't looked at for six or more months, might as well have been written by someone else". Good documentation is very important for maintaining programs. Without a specification what a code is supposed to do, the maintainer is left only with what the code really does and has to wonder if that is really what it is supposed to do. Unfortunately, TrEd's documentation is very sparse and mostly user-oriented. No technical or algorithmic documentation was available. One of the aims of this thesis is to document TrEd's functionality. Therefore a set of standard documentation boilerplates was created to document its source code. These boilerplates and other documentation-related rules are presented in this section.

4.6.1 Types of Documentation

What to do

Distinguish user documentation from technical documentation. Put user documentation in the "public" sections of your code's Plain Old Documentation (POD), relegate the technical documentation to "non-public" places. Don't put implementation details in user documentation.

Why

Users usually don't read the code of the application, only developers do. These two types of documentation should be separated.

What about TrEd

Technical documentation was present only for public API for TrEd and bTrEd macros. TrEd, bTrEd and other Perl source code contained very little documentation (sometimes only unedited standard POD boilerplate). However, the `Treex::PML` library which is now a separate CPAN package, is documented very well.

4.6.2 Boilerplates

What to do

Create standard POD templates for modules and applications.

Why

Coherent and standard way of documentation helps the readers to understand it and known structure makes it easier to navigate through the documentation.

What about TrEd

[3] also called Eagleson's Law

TrEd used a short standard boilerplate, but it was usually not edited and filled, if it was present in the file. New boilerplate, as suggested by [2] was adopted for creating TrEd's documentation:

```
=head1 NAME

<Module::Name> - <One-line description of module's purpose>


=head1 VERSION

This documentation refers to
<Module::Name>
 version 0.0.1.


=head1 SYNOPSIS

    use <Module::Name>;

# Brief but working code example(s) here showing the most common usage(s)


=head1 DESCRIPTION

A full description of the module and its features.
May include numerous subsections (i.e., =head2, =head3, etc.).


=head1 SUBROUTINES/METHODS


A separate section listing the public components of the module's interface.
These normally consist of either subroutines that may be exported, or methods
that may be called on objects belonging to the classes that the module provides↩
    .
Name the section accordingly.

In an object-oriented module, this section should begin with a sentence of the
form "An object of this class represents...", to give the reader a high-level
context to help them understand the methods that are subsequently described.


=head1 DIAGNOSTICS


A list of every error and warning message that the module can generate
(even the ones that will "never happen"), with a full explanation of each
problem, one or more likely causes, and any suggested remedies.


=head1 CONFIGURATION AND ENVIRONMENT


A full explanation of any configuration system(s) used by the module,
including the names and locations of any configuration files, and the
meaning of any environment variables or properties that can be set. These
descriptions must also include details of any configuration language used.


=head1 DEPENDENCIES

A list of all the other modules that this module relies upon, including any
restrictions on versions, and an indication of whether these required modules ↩
    are
part of the standard Perl distribution, part of the module's distribution,
or must be installed separately.


=head1 INCOMPATIBILITIES
```

```
A list of any modules that this module cannot be used in conjunction with.
This may be due to name conflicts in the interface, or competition for
system or program resources, or due to internal limitations of Perl.


=head1 BUGS AND LIMITATIONS


A list of known problems with the module, together with some indication of
whether they are likely to be fixed in an upcoming release.

Also a list of restrictions on the features the module does provide:
data types that cannot be handled, performance issues and the circumstances
in which they may arise, practical limitations on the size of data sets,
special cases that are not (yet) handled, etc.

The initial template usually just has:


There are no known bugs in this module.
Please report problems to
<Maintainer name(s)> (<contact address>)
Patches are welcome.

=head1 AUTHOR


<Author name(s)> (<contact address>)


=head1 LICENCE AND COPYRIGHT

Copyright (c) <year> <copyright holder>
(<contact address>). All rights reserved.

followed by whatever licence you wish to release it under.
```

4.6.3 Extended Boilerplates

What to do

Extend and customize your standard POD templates.

Why

If the policy of company requires other information in the documentation, e.g. examples, frequently asked questions, add them to standard boilerplate.

What about TrEd

TrEd takes the conservative approach and uses only the standard sections (see 4.6.2).

4.6.4 Location

What to do

Put user documentation in source files instead of separate `.pod` files.

Why

Separate files are more difficult to maintain and it is possible that they get lost somewhere along the way to the user. Placing documentation in the source file makes it always available.

What about TrEd

TrEd's POD is usually situated at the end of source code.

4.6.5 Contiguity

What to do

Keep all user documentation in a single place within your source file. Don't interleave POD sections between chunks of source code.

Why

To keep both the structure of user documentation and code as clear and comprehensible as possible, it is usually a good idea not to interleave documentation between the code.

What about TrEd

The approach in documented API file `tred.def` was to interleave documentation between the chunks of code. Refactored TrEd's source code contains technical-specific details of documentation near the code, POD documentation at the end of each file.

4.6.6 Position

What to do

Place POD as close as possible to the end of the file.

Why

Anyone looking at the source code is probably interested in the source code itself, not a documentation. Also the Perl compiler is usually more efficient if it doesn't have to skip the documentation at the beginning of each file.

What about TrEd

TrEd's source code contained POD at the end of file.

4.6.7 Technical Documentation

What to do

Subdivide your technical documentation appropriately. Use separate `.pod` files for design documents, change logs, etc.

Why

If the user wants to read how to use a module or subroutine, and he is not interested in implementation details, it is better not to overwhelm him. If someone is interested in the implementation, he looks in the source code and finds link to other POD files easily.

What about TrEd

TrEd contains a mix of user and technical documentation in separate xml file.

4.6.8 Comments

What to do

Use block templates for major comments. A templated block comment should be used to document each component[4] of a module or application.

Why

Block template is usually much easier and faster to grasp if it is in structured form than free text. Coherent documentation also greatly improves the speed of orientation in it.

[4] a subroutine, method, package and the main code of an application

What about TrEd

TrEd contained very little technical documentation before refactoring. A standard template which was chosen to document subroutines follows:

```
################################################################################
# Usage          : file_schema($fsfile)
# Purpose        : Return schema from file's metadata
# Returns        : Schema for $fsfile
# Parameters     : Treex::PML::Document ref $fsfile -- the file whose schema
#                  we are searching for
# Throws         : no exception
# Comments       : Should return the same value as calling $fsfile->schema()
#                  (according to Treex::PML doc)
# See Also       : Treex::PML::Document::metaData(),
#                  Treex::PML::Document::schema()
```

4.6.9 Algorithmic Documentation

What to do

Use full-line comments to explain the algorithm. Code in paragraphs (see also 4.1.13) and prefix each paragraph with a single-line comment. Unpacking subroutines arguments and return statement don't need a commentary.

Why

Code written in commented paragraphs is easy to read and comprehend. It can be also a good sign for subroutine complexity: whenever a comment above the paragraph is more than one line long, the paragraph should maybe be split or extracted into separate subroutine.

What about TrEd

TrEd's source code didn't follow this rule.

4.6.10 Elucidating Documentation

What to do

Use end-of-line comments to point out subtleties and oddities.

Why

Using special variable and subroutine names derived from a jargon or other very specific language area can be hard to read and understand without an explanation.

What about TrEd

TrEd's source code contains names from linguistic area, but does not usually contain any end-of-line comments.

4.6.11 Defense Documentation

What to do

Comment anything that has puzzled or tricked you.

Why

If something tricked you once, it will probably puzzle you again in the future. Therefore it is wise to comment that puzzling piece of code.

What about TrEd

TrEd contains only a very little commentary, even in puzzling and tricky areas of code.

4.6.12 Indicative Documentation

What to do

Consider whether it's better to rewrite than to comment.

Why

Sometimes it is better to write simpler code than to document the sophisticated one. The decision depends, of course, on the programmer. This tip is also mentioned in [3] and 2.1.22.

What about TrEd

Since this is rather a refactoring tip than a documentation rule, it is not measurable.

4.6.13 Discursive Documentation

What to do

Use "invisible" POD sections for longer technical discussions. Keep it as close to the code as possible.

Why

Don't burden the user with implementation details and the algorithms used in the source code.

What about TrEd

TrEd's source didn't contain technical discussions and documentation.

4.6.14 Proofreading

What to do

Check the spelling, syntax, and sanity of your documentation.

Why

To communicate effectively, documentation shouldn't contain spelling errors, obscure syntactic constructions. It should be clear, short and comprehensible. As one of my professors like to say: "Write technical documentation as if you were writing it to your clever colleague, who doesn't know about the problem you're trying to solve. Write users' documentation as if you were writing it to a total ignorant who doesn't care about your program"[5].

What about TrEd

The `Perl::Critic` module is used in refactored TrEd to check spelling, original TrEd did not use any of these tools

4.7 Built-in Functions

The most common functions used in Perl programs are functions which are part of the Perl language itself. It is thus very important to use them in a correct and efficient way. This chapter, inspired by [2], gives some advice about using these functions.

[5] RNDr. Rudolf Kryl

4.7.1 Sorting

What to do

Don't recompute sort keys inside a `sort`.

Why

Doing expensive computations inside the block of a `sort` is inefficient. Since the implementation of `sort` uses merge sort algorithm, the block after the function will be called $O(N.logN)$ times[6].

What about TrEd

Hash or array look-ups are the only more complex operations used in `sort` function in TrEd.

4.7.2 Reversing Lists

What to do

Use `reverse` to reverse a list, `reverse sort` to sort a list in descending order.

Why

Using `reverse sort` is more comprehensible than `sort { $b cmp $a }` and it can also be more effective, because Perl recognizes and optimizes this sequence of built-ins [2].

What about TrEd

Reversed sorting was accomplished only once using the non-preferred way in TrEd's source code.

4.7.3 Reversing Scalars

What to do

Use `scalar reverse` to reverse a scalar.

Why

Be explicit about reversing a string by forcing a scalar context with `scalar` keyword.

What about TrEd

TrEd's source code does not use string reverse.

4.7.4 Fixed-Width Data

What to do

Use `unpack` to extract fixed-width fields.

Why

`unpack` is usually much faster than using `substr` or regular expressions. It scales well and supports extracting string chunks in arbitrary order.

What about TrEd

TrEd does not work with fixed-width fields.

[6] where N is a number of sorted elements

4.7.5 Separated Data

What to do

Use `split` to extract simple variable-width fields.

Why

The most effective way of dealing with variable-length data separated by a specific pattern is to use `split` function [2].

What about TrEd

TrEd used `split` function to parse configuration file or stylesheets (when appropriate).

4.7.6 Variable-Width Data

What to do

Use `Text::CSV_XS` or `Text::CSV::Simple` to extract complex variable-width fields.

Why

There is no need to reinvent a wheel, if a well tested highly configurable module already exists. Writing own bug-free parser that handles various types of quotes and escape sequences can be difficult.

What about TrEd

TrEd used its own parser for configuration files.

4.7.7 String Evaluations

What to do

Avoid string `eval`.

Why

String `eval` starts compiler every time it is called, it is therefore expensive, especially in loops. The second large drawback is that string `eval` doesn't provide compile-time checking. Module `Sub::Install` can be used for creating subroutines generated according to user input.

What about TrEd

TrEd used string `eval` to run macros and extensions. This mechanism was also used to catch exceptions in code that may fail. This is, however, necessary for running macros, hooks and extensions.

4.7.8 Automating Sorts

What to do

Consider building your sorting routines with `Sort::Maker`.

Why

Don't reinvent the wheel, if you need various sort functions, use module `Sort::Maker` which is tested and optimized.

What about TrEd

TrEd does not use this module. Sorting in TrEd is usually done using built-in `sort` function which is sufficient for its needs.

4.7.9 Substrings

What to do

Use 4-arg `substr` instead of lvalue `substr`.

Why

Assigning value to a function can be less comprehensible, because it's not very common (even in Perl). Using `substr` with 4 arguments is also faster because no intermediate string representation has to be created.

What about TrEd

Lvalue `substr` is never used in TrEd's source code.

4.7.10 Hash Values

What to do

Make appropriate use of lvalue `values`.

Why

Using lvalue `values` for assigning new values to hash is more efficient in loops, because no hash look-up is needed. This approach can be, however, only used with Perl compilers newer than 5.6.

What about TrEd

This approach is not used in TrEd because it supports older Perl compilers.

4.7.11 Globbing

What to do

Use `glob`, not `<...>`.

Why

Most people associate `<...>` with Perl's I/O system. Using `glob` function is more comprehensible and less error-prone.

What about TrEd

TrEd uses `glob` function to expand file names.

4.7.12 Sleeping

What to do

Avoid a raw `select` for non-integer sleeps.

Why

Using a side-effect of a function with other purpose just for that side-effect is bad practice for both understandability and maintainability of the program. Use `Time::HiRes` module, if it is possible.

What about TrEd

TrEd does not use sleeping.

4.7.13 Mapping and Grepping

What to do

Always use a block with a `map` and `grep`.

Why

Using block versions of these built-ins make the transformation/filter stand out more clearly. It is also less error-prone and more scalable.

What about TrEd

TrEd's source code contains function-type call of `grep` and `map` quite often

4.7.14 Utilities

What to do

Use the "non-builtin builtins", i.e. subroutines from `Scalar::Util`, `List::Util`, and `List::MoreUtils` modules: `blessed`, `refaddr`, `reftype`, `readonly`, `tainted`, `openhandle`, `weaken`, `is_weak`, `first`, `max`, `maxstr`, `min`, `minstr`, `shuffle`, `reduce`, `uniq`, etc.

Why

Althought calling a subroutine can be in some cases more computationally expensive, it's cleaner, more scalable and readable solution.

What about TrEd

TrEd's source code usually takes advantage of these subroutines. Subroutines from `List::Util` were reimplemented because of name mangling of `$a` and `$b` variables in macro contexts. Subroutine `uniq` was implemented redundantly in several packages.

4.8 Subroutines

Subroutines are the smallest units of program modularity and abstraction. This section inspired by [2] lays out 12 rules how to use them effectively.

4.8.1 Call syntax

What to do

Call subroutines with parentheses but without a leading `&`.

Why

Calling subroutines with parentheses makes it easy to distinguish them from built-in subroutines and avoid confusion when calling multiple subroutines. This approach improves readability and comprehensibility of the code. Using `&` to call a function may be confusing, because, depending on the context, it can also be considered a bitwise and operator. Moreover, if the function is called without arguments, the default argument passed to the function is not `$_`, but `@_`, which can be confusing

What about TrEd

Subroutines in TrEd were called with leading `&` quite often to achieve automatic passing of all the arguments of the original function to the called function. This is avoided in new code.

4.8.2 Homonyms

What to do

Don't give subroutines the same names as built-in functions.

Why

Perl uses two classes of built-in functions – those more privileged, more built-in, which are called even if there is a subroutine with the same name in current scope – and those less privileged that can be overriden by subroutines in current scope. This confusing behaviour can easily become a maintenance nightmare.

What about TrEd

Some of the modules use names which are colliding with standard Perl functions, e.g. module `TrEd::Convert` uses colliding names of subroutines.

4.8.3 Argument List

What to do

Always unpack `@_` first. Use one line single list assignment or a series of `shift`s if the arguments need to be checked/sanitized.

Why

Directly using arguments as `$_[n]` is slightly more efficient, but is not self-documenting. The code is therefore less comprehensible. What is more, elements of `@_` are aliases to original arguments and their value can be unintentionally changed.

What about TrEd

Function arguments were usually unpacked on the very first lines of subroutines. Only in specific cases where `@_` was just passed to another function or numbered arguments were used for efficiency reasons.

4.8.4 Named Arguments

What to do

Use a hash of named arguments for any subroutine that has more than three parameters.

Why

For humans it is usually easier to remember names than to remember specific order of arguments to function. The advantage of using hash rather than raw list is that errors in the hash of named arguments will be usually reported at compile time in the caller's context, while errors in raw list will be usually reported at run time in the calee's context.

What about TrEd

TrEd subroutines that use many parameters didn't use hash of named arguments. Refactored TrEd introduced this approach for functions with more than 4 arguments.

4.8.5 Missing Arguments

What to do

Use definedness or existence to test for missing arguments. Don't use simple boolean test, i.e. `if !$argument1`.

Why

Simple boolean test fails if the subroutine's argument is 0 or empty string, which is usually not what is meant. Better approach is to use `defined` and test whether the number of arguments is in desired interval.

What about TrEd

TrEd used boolean test (not only) for subroutines' arguments frequently.

4.8.6 Default Argument Values

What to do

Resolve any default argument values as soon as `@_` is unpacked.

Why

It is more clear and comprehensible to prepare the arguments with their default values before using them in the subroutine.

What about TrEd

TrEd's source code usually follows this rule.

4.8.7 Scalar Return Values

What to do

Always `return scalar` in scalar returns.

Why

If a function that should return only scalar value could return a list or a scalar variable depending on the context, using explicit `return scalar` to force scalar context can help avoid introducing subtle bugs.

What about TrEd

TrEd's source code follows this rule.

4.8.8 Contextual Return Values

What to do

Make list-returning subroutines return the "obvious" value in scalar context. Ask the users of the subroutines which one the obvious value is. Usually, for homogenous lists it is the count of the items in the list. For heterogenous lists the "obvious" value is the most significant value or a serialization of all the list items. Finally, an iterative list-returning subroutine should return the result of one iteration in the scalar context.

Why

Using the behaviour that is expected from the subroutine is beneficiary for the author and the user of the subroutine.

What about TrEd

TrEd's source code sometimes used this approach, but most of the functions were considered to be called only in one specific context.

4.8.9 Multi-Contextual Return Values

What to do

When there is no "obvious" scalar context return value, consider `Contextual::Return` instead.

Why

When there is a need to return different values from a subroutine in scalar context (e.g. string vs boolean context), `Contextual::Return` CPAN module

can be used to achieve more fine-grained context resolution. In some cases this approach can help to clean API, on the other hand, using more properly named subroutines might be a solution as well.

What about TrEd

TrEd's source code doesn't use `Contextual::Return` module and doesn't need to resolve context with such details.

4.8.10 Prototypes

What to do

Don't use subroutine prototypes.

Why

Subroutine prototypes use different rules for passing arguments to subroutine. Since the caller has to remember which subroutines are prototyped, it's easy to make a mistake. Maintenance of this code is therefore more difficult and there is usually a good alternative to using prototyped subroutine.

What about TrEd

TrEd sometimes uses subroutine prototypes, especially in macros and in `TrEd::MinMax` module, whose functions should work as a replacement for some of `List::Util` functions.

4.8.11 Implicit Returns

What to do

Always return via an explicit `return`.

Why

The return value of a function without an explicit `return` is the last value evaluated in the subroutine. The return value can be the return value of the last boolean expression in a loop or conditional statement. It can difficult to track down, which statement would be the last one in every of the possible subroutine's paths. The code is much more predictable and easily maintainable when explicit `return` is used.

What about TrEd

TrEd used functions without explicit return values frequently (about 500 subroutines).

4.8.12 Returning Failure

What to do

Use a bare `return` to return failure. Don't use `return undef`.

Why

If a function returns `undef` explicitly and it is called in list context, the subsequent boolean test on the return value returns true, because a list with one element – an undefined value – is evaluated to true in boolean context, which can introduce subtle bugs. Bare `return` returns `undef` in scalar context and empty list in list context, which is evaluated in boolean context as usually expected.

What about TrEd

TrEd returned `undef` explicitly quite frequently, mote than 70 subroutines did it this way. This rule is obeyed in refactored TrEd and explicit `return undef` were changed to bare `return`s.

4.9 Input and Output

Input and output (I/O or IO) are usually the slowest parts of the computer systems, being the bottleneck for many programs. It is therefore very useful to know how to optimize these operations in Perl.

TrEd uses for most of its input and output `Treex::PML` library, which also takes care of (de)compression of input and output files. However, besides the input linguistic files, many configuration files, macros and stylesheets are used in TrEd. The rules in this section were followed to make TrEd's IO operations more clear, maintainable and less buggy.

4.9.1 Filehandles

What to do

Don't use bareword filehandles.

Why

Bareword filehandle is a kind of package variable, therefore it is shared by all the subroutines that use the same bareword. Even worse situation arise when another function reads a different file with the same filehandle name – the previous file is closed and the filehandle is then associated with the new file.

Another problem with bareword filehandle is that if it collides with function name, it fail silently. `open` function and angle brackets used for reading input file don't treat barewords the same way. This situation can lead to unexpected bugs that are difficult to find.

What about TrEd

TrEd follows this rule.

4.9.2 Indirect Filehandles

What to do

Use indirect filehandles.

Why

Indirect filehandle is a lexical variable which is always a better choice compared to package variable (see also 4.4.1 and 4.4.2). The worst thing that might happen is that the previous variable with the same name will be hidden, but if `use warnings` is used, a programmer is warned about this situation.

What about TrEd

TrEd almost always follows this rule. One exception is when using `sysopen` call in `TrEd::Cipher` package, the other was in `TrEd::Config` package, where it wasn't necessary and, therefore, it was eliminated.

4.9.3 Localizing Filehandles

What to do

If you have to use a package filehandle, localize it first. When referring to the symbol table entry, use asterisk to make the use explicit.

Why

Using `local` has nearly all the advantages of using lexical variables, except that localized variable can be seen in deeper scopes. For other advantages of lexical filehandles, see previous rule (4.9.2).

What about TrEd

TrEd used localized filehandles only several times.

4.9.4 Opening Cleanly

What to do

Use either the `IO::File` module or the three-argument form of `open`.

Why

The behaviour of the two-argument `open` can be altered if name of the file contains angle brackets. The opening mode can't be changed by file name if three-argument `open` or `IO::File` module is used for IO.

What about TrEd

Two-argument open was used several times[7] in TrEd.

4.9.5 Error Checking

What to do

Never `open`, `close`, or `print` to a file without checking the outcome.

Why

These three IO operations usually fail most frequently. Use low precedence `or` as suggested in 4.3.16.

What about TrEd

The return value of `open` was usually checked in TrEd (only 3 ignored return values), but return values of `close` and `print` functions were usually ignored (37 and 290 times, respectively).

4.9.6 Cleanup

What to do

Close filehandles explicitly, and as soon as possible.

Why

Files are shared resource and it's always a good idea to release shared resources as soon as possible. Perl internal resources such as file buffers can be released sooner, too. Moreover, the number of filehandles is usually limited, so it's recommended not to leave the filehandles open for a long time.

What about TrEd

TrEd usually follows these rules, however, according to `Perl::Critic`'s default settings, this rule was broken 17 times.

[7] 8 times

4.9.7 Input Loops

What to do

Use `while (<>)`, not `for (<>)`.

Why

Using `for (<>)` evaluates the IO operator in list context, which means the whole file is read into an internal list at once and then processed iteratively in the for loop. This way of processing files is very memory inefficient if just a sequential access to the file is needed.

This approach also can't be used interactively because the for loop does not start until the end of file has been encountered.

What about TrEd

TrEd obeys this rule.

4.9.8 Line-Based Input

What to do

Prefer line-based I/O to slurping.

Why

Slurping – reading the file at once to a variable – is generally less scalable and slower than reading the file one line at a time and it should be used only when it's really necessary.

What about TrEd

TrEd slurps only small files such as configuration files. For other IO it uses `Treex::PML` CPAN module.

4.9.9 Simple Slurping

What to do

Slurp a filehandle with a `do` block for purity, e.g.

```
my $slurped_file = do { local $/; <$in> };
```

Why

This approach first set record separator (`$/`) to `undef` locally and then reads the whole file as a single line. This approach is faster and scales better than using `join` function with empty string or concatenating iteratively.

What about TrEd

TrEd used only angle brackets to do simple slurping.

4.9.10 Power Slurping

What to do

Slurp a stream with `Perl6::Slurp` for power and simplicity.

Why

Using a function from a CPAN module may be more appropriate (it is a cleaner and more comprehensible approach), although some sources[12] indicate that this module might have some problems with handling UTF-8 properly.

What about TrEd

TrEd doesn't use `Perl6::Slurp` CPAN module.

4.9.11 Standard Input

What to do

Avoid using `*STDIN`, unless you really mean it.

Why

The STDIN can be, depending on the situation, input and output redirection from another process, attached to various file descriptors. It is thus safer to use `*ARGV`, which allows users to specify input files on the command line or by using pipes/redirection.

What about TrEd

TrEd obeys this rule.

4.9.12 Printing to Filehandles

What to do

Always put filehandles in braces within any `print` statement.

Why

Helps with readability because it distinguishes the file handle argument from the rest of the arguments.

What about TrEd

TrEd does not follow this rule. There are more than 250 occurrences of filehandle without braces within `print` statement.

4.9.13 Simple Prompting

What to do

Always prompt for interactive input.

Why

Interactive programs should tell the user that they expect some kind of input to perform the tasks they are designed to, not just wait silently.

What about TrEd

TrEd is an interactive program, but it uses GUI, so it uses a different approach to user interaction.

4.9.14 Interactivity

What to do

Don't reinvent the standard test for interactivity. Use `IO::Interactive` module instead.

Why

Don't reinvent the wheel, especially if it is complicated to develop and maintain.

What about TrEd

TrEd does not use `IO::Interactive` module, it uses GUI built with Tk framework.

4.9.15 Power Prompting

What to do

Use the `IO::Prompt` module for prompting.

Why

Use already existing module if you need to prompt for input on the command line frequently. It is a well-thought feature-rich abstraction for prompting the user.

What about TrEd

As stated above, TrEd uses GUI, bTrEd is a batch program that does not use interactive features. Module `IO::Prompt` is thus not used.

4.9.16 Progress Indicators

What to do

Always convey the progress of long non-interactive operations within interactive applications.

Why

The indicators of progress for lengthy operations is needed for any interactive application so that user is informed when he can interact with the application again.

What about TrEd

TrEd uses the facilities of Tk framework (usually progress bars) when working interactively in GUI.

4.9.17 Automatic Progress Indicators

What to do

Consider using the `Smart::Comments` module to automate your progress indicators.

Why

Although it can be useful to turn specially-crafted comments into (automatically generated) progress indicators, some sources[12] warns against usage of this module, because it is a source filter which are considered to do more harm than good[12].

What about TrEd

TrEd does not use `Smart::Comments` or other source filters.

4.9.18 Autoflushing

What to do

Avoid a raw `select` when setting autoflushes.

Why

One-argument `select` affects the default filehandle for all the subsequent `print` calls globally, therefore it is a good practice to avoid it completely. For flushing automatically, using `autoflush()` function from `IO::Handle` module is preferred.

What about TrEd

TrEd does not use `select` function at all. Autoflush was turned on by using special variable (`$|`) only in scripts used for TrEd releasing.

5. TrEd Refactoring

Now when we have performed basic static and dynamic analysis of TrEd's code base in Chapter 2, familiarized ourselves with TrEd's implementation in Chapter 3, and laid out rules and coding standards for the program and evaluated adherence of original TrEd to these rules, we can describe changes made in TrEd's source code.

After identification of the problematic areas in TrEd and quantitative expression of code smells described earlier in Chapter 2, we present the comparison of various metrics used to measure the quality of code and the extent of refactoring which has been done on TrEd's code base.

The main goals of TrEd's refactoring were to improve the quality of TrEd's source code. Of course, no exact measure of software quality exists, but generally, our efforts have been spread among these areas:

1. Adopt a set of rules and hints to make the source code consistent.
2. Document as much of the source code as possible.
3. Make the source code modular, especially TrEd's main package.
4. Create a test suite to cover as much of the code as possible.
5. Fix bugs found in the process of refactoring.

Adopting a set of reasonable rules for the source code is always essential, and these rules are needed even more if many people are involved in the project. Since TrEd supports extensions and user macros, writing each piece of code using different style would be very undesirable. Common coding style helps in faster orientation in source code and improves readability of code for everyone.

Documenting the source code is very important for its future maintenance. It is very hard to check the correctness of a function, if there is no specification what the function should do. We then only have to guess what it is probably supposed to do by examining what it really does (and we might still wonder, whether some specific behaviour, which seems exotic to us, is intended or it is a bug).

The TrEd's main package, as mentioned earlier, was more than 13,000 lines long and contained almost 500 functions. We can consider it to be a sort of a God object antipattern[1] occurrence. Making code modular makes it not only more maintainable and easier to understand, but it is also essential for creating a test suite. Having 500 subroutines intertwined in one file, which communicate by modifying almost 80 global variables makes the module almost untestable.

Creating a testing suite is probably the only way how to make development of a complex program sustainable. Testing suite is not an ultimate tool for creating bug-free software, but it can greatly increase the quality of a program. Good testing suite makes adding new features and refactoring easier, since the developers can be sure that exercised parts of the program remain functional.

[1] if a significant part of a program's functionality is coded into a single "all-knowing" object, which maintains most of the information about the entire program and manipulate all the data, it becomes an all-encompassing God object

5.1 Conceptual Changes

Probably the strongest code smell in original TrEd was the main `tred` file, as mentioned earlier. The subroutines within this file were categorized according to data they manipulate and whenever possible, they were moved to a distinct package along with the data they access and modify. Some functions were split or extracted to separate their concerns (as mentioned earlier, almost 200 subroutines were created this way), some were merged together and the duplicates were removed. These changes are visualized in Figure 5.1. While the sizes of the boxes or their positions are not significant, the widths of the arrows indicate the number of moved subroutines. The thicker the arrow, the more subroutines were moved in that direction. The thick arrow pointing from `main` package to `TrEd::File` package represents 18 subroutines and even thicker arrow to the group of dialogs represents 31 subroutines. On the other hand, sometimes only 1 subroutine was moved, e.g. from package `main` to `TrEd::Macros` or `TrEd::Extensions`. In total, 235 subroutines and variables were moved between packages; 193 of the subroutines and variables were moved from the `main` package to some other packages. One package was deleted, 50 new Perl packages was created.

Complete list of moved functions and created packages can be found in Appendix D.

Another significant change that was already mentioned in Section 3.14.1 is the transformation of macros to standard Perl packages. Since all the extensions use TrEd's macro system, it was not possible to remove the macros completely. However, the default macros (`tred.def` and `tred.mac`) and all the macros from the `contrib` directory, which are not used by extensions (`node_groups.mak`, `ntred.mak` and minor mode macros – `move_nodes_freely.mak`, `show_neighboring_sentences.mak`, `show_neighboring_trees.mak`) have been turned into Perl packages.

The default macros, which contained the API for extensions and other macros have become `TrEd::MacroAPI::Default` and `TrEd::MacroAPI::Extended` packages. The directives used by macros should be replaced by function calls. For some of them, API subroutines already exist, for the rest, `TrEd::Macros` subroutines can be used.

- `#bind` → `Bind()` or `TrEd::Macros::bind_macro()`
- `#unbind` → `UnbindBuiltin()` or `TrEd::Macros::unbind_key()`
- `#insert` → `TrEd::Macros::add_to_menu()`
- `#remove-menu` → `TrEd::Macros::remove_from_menu()`
- `#binding-context` → `TrEd::Macros::set_current_binding_contexts()`
- `#define` → `TrEd::Macros::define_symbo()`
- `#ifdef`, `#elsif`, `#endif` → `TrEd::Macros::is_defined()`, standard Perl condition statements and scopes
- `#key-binding-adopt` → `TrEd::Macros::copy_key_bindings`

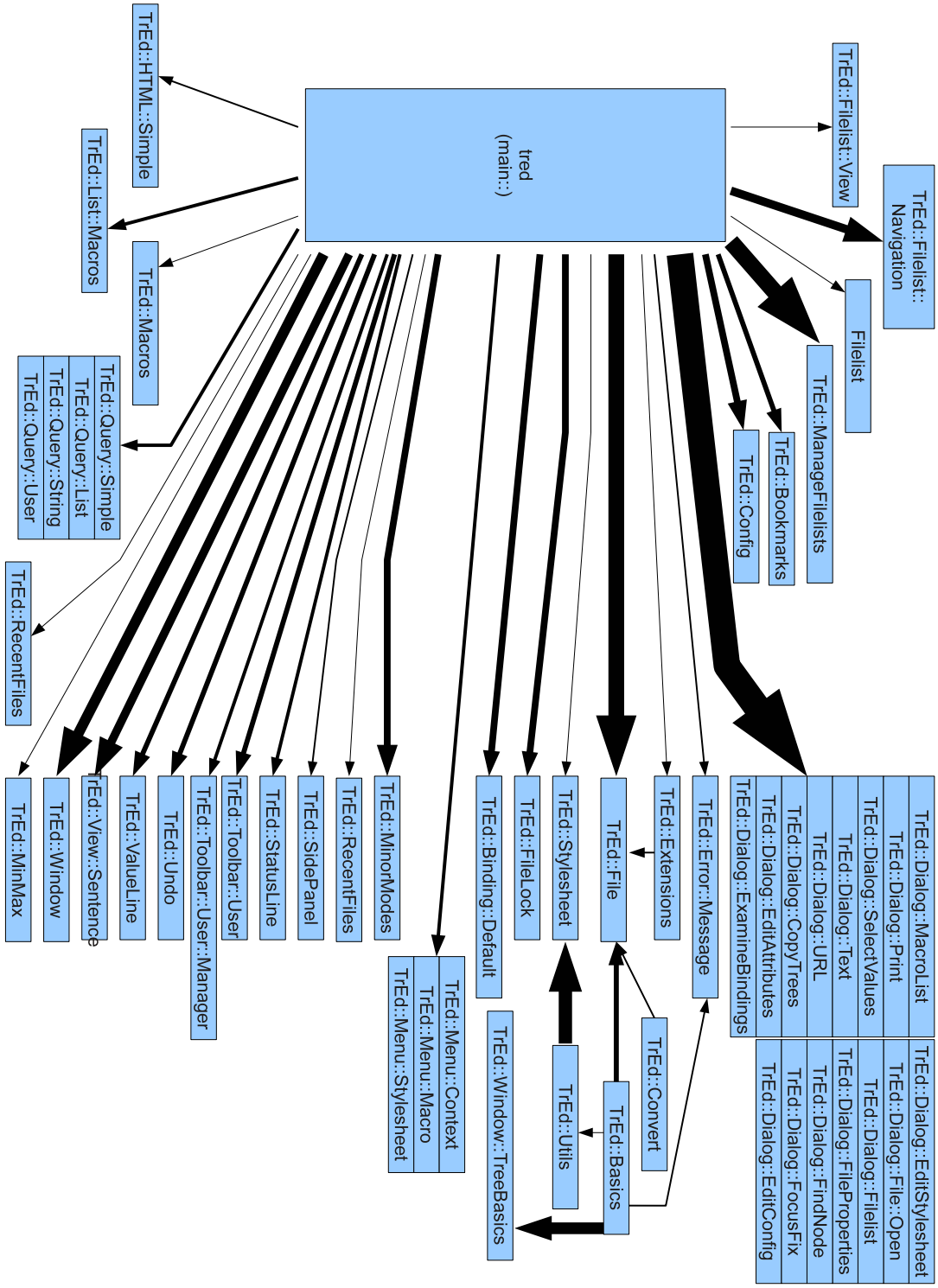


Figure 5.1: Code flow diagram of TrEd refactoring

- `#menu-binding-adopt` → `TrEd::Macros::copy_menu_bindings`

Replacing `#include` and `#ifinclude` macro directives requires more changes in extensions and macros using this directive. We believe that libraries used by extensions should be packed along with them and then they can be used by standard Perl means, i.e. `use` or `require` package (e.g. PML Tree Query already uses this approach for `Tree_Query::` modules). Libraries used by more extensions should become part of TrEd code base or they can be packed as another extension which other extensions would depend on (extensions already support the feature of installing other dependent extensions with them).

For `#encoding` macro directive, no replacement exists yet. The libraries used by macros can use `use encoding` Perl pragma, but the encoding of macro itself has to be defined somehow. Without the specification of encoding, TrEd would not know, how to handle input text.

5.2 Static Analysis

Similarly to Chapter 2, we performed static and dynamic analysis of refactored TrEd and compared the results with original TrEd. First, we focus on overall code characteristics.

5.2.1 Code Metrics

As we can see in Table 5.1, the number of TrEd core files remained the same, while the number of lines of code has been reduced by 28% and the number of subroutines has been reduced by 33%. The number of methods in TrEd's main Perl file `tred` changed from 356 to 214. All this subroutines were moved to packages and modules. As we can see, the number of modules almost doubled, as well as the amount of code they contain. We can also observe that number of macros was halved. This is because macros in TrEd's core were transformed into regular Perl packages.

We can see that total number of files increased from 79 to 116. New files were created for modules extracted from `tred` file, but some files were also deleted, e.g. Tk::EditableCanvas class which was not used by any of TrEd's components.

Table 5.1 also tells us that approximately 5 thousand lines of inline documentation and 10 thousand lines of Plain Old Documentation were written or generated. To keep the statistics clean, we should, however, mention, that documentation of subroutines overlaps. POD was generated from subroutines' in place documentation automatically.

The last interesting figure in Table 5.1 is the increase in number of subroutines. Since no new functionality was added, this accounts for 198 extracted routines from larger subroutines to make the code more readable, modular and maintainable.

The increase in number of lines of code in TrEd (this is counted simply by counting new lines in each file) is not only caused by the documentation, but it is also a result of using `perltidy` script with screen width set to 78 characters. This cause many long lines to be split into few shorter ones. As a side effect of

	Files	Lines of code	Lines of #	Lines of POD	Subroutines
Original TrEd					
Core files	4	18,199	844	1,476	426
Modules	53	22,826	1,208	2,105	686
Macros	22	9,582	458	2,553	375
Total	79	50,607	2,510	6,134	1,487
Refactored TrEd					
Core files	4	13,003	1,048	1,476	285
Modules	102	58,773	5,938	14,635	1,310
Macros	10	3,582	266	360	63
Total	116	75,358	7,252	16,471	1,658

Table 5.1: TrEd code metrics overview – comparison

	LOC/file	sub/file	lo#/LOC	loPOD/file	loPOD/LOC
Original TrEd					
Core files	4549.8	106.5	0.046	369.0	0.081
Modules	430.7	12.9	0.053	39.7	0.092
Macros	435.5	17.0	0.048	116.0	0.266
Total	5,416.0	136.5	0.147	524.8	0.440
Refactored TrEd					
Core files	3250.8	71.3	0.081	369.0	0.114
Modules	576.2	12.8	0.101	143.5	0.249
Macros	358.2	6.3	0.074	36.0	0.101
Total	4,185.2	90.4	0.256	548.5	0.463

Table 5.2: TrEd code metrics overview – relative comparison

this layout settings, the code is less dense, since fewer operations per line of code happen.

The comparison of relative code metric statistics can be seen in Table 5.2. Even though the number of source files increased, the relative amount of POD documentation per file increased, too, and the number of # comments per line of code doubled. The average number of subroutines in file dropped from 136 to 90. The average length of TrEd's Perl file decreased by 22 % (even though refactored TrEd's source code is more often accompanied by documentation).

Because the code in extensions was not refactored, we leave their figures out. Static analysis of extensions' code can be found in Section 2.2.2.

Boxplots in Figures 5.2 and 5.3 show that the overall distribution of cyclomatic complexity of subroutines has not been affected much by refactorings done during our work on the thesis. The length of subroutines increased mainly due to using shorter length of line and added documentation. The longest code sequences are present outside of subroutines in `btred` and `ntred` files (i.e. the initialization of these programs).

Subroutines with the highest cyclomatic complexity in refactored TrEd are listed in Table 5.3. If we compare them with most complex subroutines in original TrEd (listed in Table 5.4), we can see that the complexity of these subroutines has

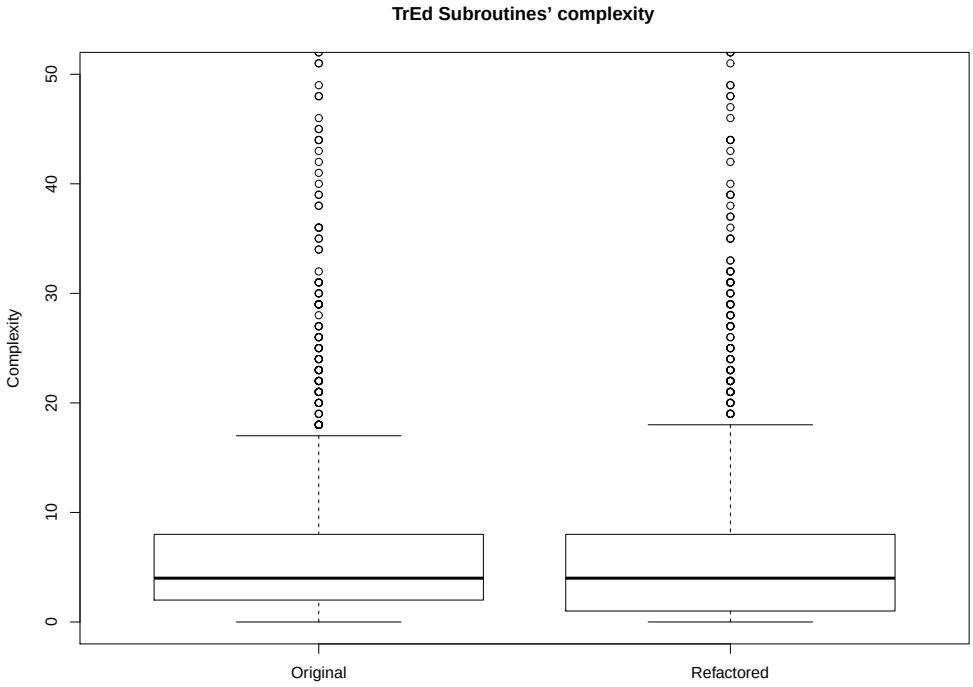


Figure 5.2: TrEd's source code – subroutines' complexity comparison

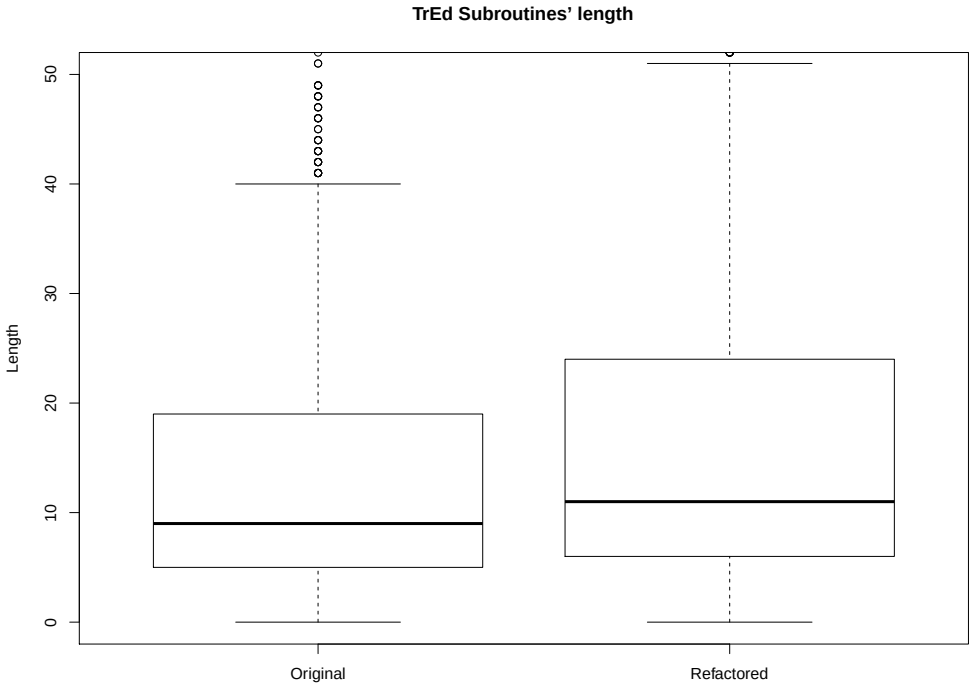


Figure 5.3: TrEd's source code – subroutines' length comparison

Subroutine name	Source File	Complexity
start_server	btred	326
redraw	TrEd/TreeView.pm	265
print_trees	TrEd/Print.pm	203
draw_canvas	Tk/Canvas/SVG.pm	162
draw_canvas	Tk/Canvas/PDF.pm	159
add_member	Tk/TrEdNodeEdit.pm	137
recalculate_positions	TrEd/TreeView.pm	112
preprocess	TrEd/Macros.pm	92
Popup	Tk/Balloon.pm	90
open_file	TrEd/File.pm	86

Table 5.3: The most complex subroutines in refactored TrEd

Subroutine name	Source File	Complexity
startMain	btred	384
redraw	TrEd/TreeView.pm	260
_populate_extension_pane	TrEd/Extensions.pm	250
print_trees	TrEd/Print.pm	203
draw_canvas	Tk/Canvas/SVG.pm	162
startMain	tred	160
draw_canvas	Tk/Canvas/PDF.pm	159
set_config	TrEd/Config.pm	149
add_member	Tk/TrEdNodeEdit.pm	137
openFile	tred	135

Table 5.4: The most complex subroutines in original TrEd

been reduced in average by 37 points. As these are still very complex subroutines, their further refactoring is subject to future work.

5.2.2 Perl::Critic

Warnings emitted by `Perl::Critic` source code analysis tool tells us, how much TrEd adheres to new coding style described in Chapter 4. Because many subroutines moved between the categories we described earlier in Chapter 2, we can not directly compare the numbers of code violations in TrEd's core or modules. The Table 5.5 is therefore only of limited value and may be used for future purposes, if the refactoring of TrEd would continue.

We can, however, compare the number and type of violations in the whole codebase of TrEd. In Table 5.6 we can find comparison of number of violations in TrEd before and after refactoring, ordered by severity of the warnings. The number of most severe violations was reduced by almost 42 %, from 324 to 189. Also, the number of violations of severity 4 was reduced by almost 20 %. In general, more than 1,600 violations were fixed, which means that the number of all violations was cut down by 10 % and the number of violations per line of code dropped by 38 % to 0.19.

Not all of the violations can be, however, avoided. Using string evals or dis-

	1	2	3	4	5	total	violations per loc
Core	1,359	1,450	620	232	39	3,700	0.28
Modules	2,999	4,474	1,538	972	136	10,119	0.17
Macros	71	128	64	52	14	329	0.09
Total	4,429	6,065	2,222	1,256	189	14,148	0.19

Table 5.5: Code violations by severity in refactored TrEd

	1	2	3	4	5	total	violations per loc
Before refactoring	5,090	6,458	2,329	1,562	324	15,763	0.31
After refactoring	4,429	6,065	2,222	1,256	189	14,148	0.19

Table 5.6: Comparison of code violations by severity in TrEd

abling strict pragma were necessary in some parts of code. Some of the violations are difficult to fix, e.g. using regular expressions with recommended flags would need to reconsider every regular expression in TrEd, because adding these flags changes the semantics of regular expressions.

The most common violations of Perl::Critic rules with severity level above 3 can be seen in Table 5.7.

5.3 Dynamic Analysis

Performing a dynamic analysis of TrEd and bTrEd before and after refactoring is necessary, because we have to be sure that refactoring did not affect the overall TrEd performance in a negative way. Therefore, we performed the same tests on refactored TrEd as we did with original TrEd (see Section 2.3).

We chose to examine three model situations for purposes of dynamic analysis:

1. simple bTrEd script,
2. start-up of TrEd,
3. browsing PML trees in TrEd.

These three model situations are described in further detail in Chapter 2. Let's have a look at the results of running the programs under aforementioned circumstances.

5.3.1 bTrEd Evaluation

The average time of running the script in refactored bTrEd was 10.98 seconds, while the average time of original bTrEd was 11.46 seconds. The performance improvement is not statistically significant, since the 95 % confidence intervals of the runtimes overlap. Most of the work in this script is done by `Treex::PML` library; the order and number of calls of subroutines is almost identical to the Tables 2.11 and 2.12. The small improvement in refactored bTrEd is probably

Policy name	Count	Severity
RegularExpressions::RequireExtendedFormatting	813	3
Subroutines::RequireFinalReturn	577	4
Variables::ProhibitPackageVars	496	3
Subroutines::RequireArgUnpacking	278	4
Variables::ProhibitReusedNames	156	3
ErrorHandling::RequireCheckingReturnValue-OfEval	124	3
ValuesAndExpressions::ProhibitMixedBoolean-Operators	116	4
ControlStructures::ProhibitDeepNests	85	3
ErrorHandling::RequireCarping	75	3
Subroutines::ProhibitExcessComplexity	69	3
Subroutines::ProhibitSubroutinePrototypes	44	5
NamingConventions::ProhibitAmbiguousNames	43	3
TestingAndDebugging::RequireUseWarnings	41	4
BuiltinFunctions::RequireBlockMap	39	4
ControlStructures::ProhibitCascadingIfElse	38	3
ControlStructures::ProhibitNegativeExpressions-InUnlessAndUntilConditions	36	3
Subroutines::ProtectPrivateSubs	36	3
Variables::RequireLocalizedPunctuationVars	35	4
BuiltinFunctions::ProhibitStringyEval	30	5
TestingAndDebugging::RequireUseStrict	30	5
BuiltinFunctions::ProhibitComplexMappings	30	3

Table 5.7: Most common Perl::Critic warnings for refactored TrEd

Subroutine	Call count	% of all sub calls
TrEd::Macros::CORE:match	272,719	50.5
TredMacro::CORE:match	16,590	3.1
TredMacro::CORE:subst	16,563	3.1
utf8::CORE:match	15,553	2.9
Encode::_utf8_off	14,059	2.6
TrEd::Macros::CORE:readline	13,912	2.6
Carp::CORE:substcont	6,123	1.1
XML::LibXML::Node::DESTROY	5,000	0.9
Treex::PML::Factory::__ANON__ (BEGIN)	4,635	0.9
Scalar::Util::weaken	4,234	0.8
Total – 10 most frequent	369,388	68.4
Total	539,896	100.0

Table 5.8: Most frequently called subroutines in refactored TrEd – start of TrEd

caused by a smaller amount of macro code that needs to be evaluated during the start-up phase of bTrEd. This caused fewer calls to regular expression matching operator (reduced from 393,900 calls to 269,837 calls), `Encode::_utf8_off` subroutine (from 20,644 calls to 13,986 calls) or readline subroutine.

5.3.2 TrEd Start

The change of start-up time of TrEd could be affected by two factors. First, we have split the initialization into several smaller subroutines, so there could be a little overhead of calling them if we compare it with sequential initialization. Second, the amount of code evaluated as macros shrank, therefore there are fewer calls to `readline` function or regular expression matching operator. Both these factors have, however, little impact on the performance of TrEd in this test. Even though the number of `TrEd::Macros::CORE:match` calls dropped by 32 %, the difference between the start-up times is not statistically significant. The average start-up time of TrEd before refactoring was 4.80 seconds and after refactoring it was 4.64 seconds. Most of the time during the initialization was spent initializing macros and reading input file into memory by `Treex::PML` library.

5.3.3 Browsing in TrEd

After the refactoring and some small optimizations of `TrEd::TreeView` module, we can compare the number of calls of subroutines and statements in TrEd before and after the refactoring. The number of executed statements lowered from $21,200,350$ to $14,913,680$ and the number of subroutine calls dropped from $6,657,664$ down to $4,617,669$. Putting the most frequent options in an often called subroutine at the top of an "if" clause decreased the number of calls to `UNIVERSAL::isa` from $1,115,769$ to $466,563$. The situation with `UNIVERSAL::DOES-::does` function is very similar, as we can see from the comparison Table 5.10. The number of presented attributes is the same regardless of TrEd version, which is an indication that the number of drawn elements was the same in both versions

Subroutine	Exclusive time [ms]	% of total time
Tk::DoOneEvent	501.0	14.7
Tk::update	408.0	11.9
TrEd::Macros::preprocess	368.0	10.8
TrEd::Macros::CORE:match	162.0	4.7
TrEd::Macros::initialize_macros	118.0	3.4
TredMacro::_import	95.2	2.8
utf8::SWASHNEW	51.7	1.5
Tk::configure	38.2	1.1
Exporter::import	34.2	1.0
Tk::destroy	32.6	1.0
Total – 10 most longest	1808.9	39.0
Total	4640.0	100.0

Table 5.9: Subroutines taking longest time to execute in refactored TrEd – start of TrEd

Subroutine	Call count before	Call count after
UNIVERSAL::isa	1,115,769	466,563
UNIVERSAL::DOES::does	993,025	337,852
UNIVERSAL::DOES xsub	883,632	234,273
TrEd::Macros::CORE:match	492,594	373,945
TrEd::TreeView::_present_attribute	166,515	166,515

Table 5.10: Most frequently called subroutines during browsing of trees – comparison

of TrEd.

5.4 Testing

Creating a test suite for TrEd was one of the goals of this diploma thesis. Standard Perl utilities used for writing tests – `Test::More` and `Test::Exception` – were used for creating tests. For the purposes of evaluation to what extent the code is exercised by the tests, `Devel::Cover` module was used. All these modules are available on CPAN. The tests we created cover these TrEd modules :

- `Filelist`
- `TrEd::ArabicRemix`
- `TrEd::Window::TreeBasics`
- `TrEd::Binding::Default`
- `TrEd::Cipher`
- `TrEd::Config`

- `TrEd::Convert`
- `TrEd::ConvertArab`
- `TrEd::CPConvert`
- `TrEd::TrEd::Error::Message`
- `TrEd::TrEd::Extensions`
- `TrEd::FileLock`
- `TrEd::Macros`
- `TrEd::MinMax`
- `TrEd::Stylesheet`
- `TrEd::Utils`
- `TrEd::Utils`

If we consider all the TrEd modules, which are reachable from test files (that is 43 out of 102 TrEd's modules), the test coverage, as measured by the `Devel::Cover` tool is 69.2 %, which means, that almost 70 % of subroutines in these modules have been called by tests. More than 1,300 tests for 250 subroutines have been written for TrEd.

We encountered several problems connected with testing TrEd. Fowler in his book about refactoring [3] strongly encourages to use tests during refactoring. That is certainly true for small-steps refactorings he describes. The problem with creating tests for TrEd's refactoring is that there wasn't any documentation available, which would tell us, what is the purpose of the code in subroutines. We had to guess it by the implementation of the subroutine itself and by its connection to other parts of the program. To create test for such subroutine, we had to map all the edge cases and odd results subroutine may return, in case some other part of code would rely on this behaviour. Creating tests by looking at the implementation of the tested piece of code is called white box testing. The disadvantage of white box testing is that you actually don't test whether the function returns what it is supposed to, but you rather test the actual implementation of this process.

The other interesting problem lies in the fact that in Perl, there is no encapsulation. In fact, all the subroutines in all created packages are public. Therefore you usually don't test only the public interface of your package, because everything is public. Moreover, if you want to refactor private subroutine and follow the advices of [3], you should be able to test your "private" subroutines as well.

Probably the most practical problem is the problem of complex subroutines, with very high cyclomatic complexity. Because of many conditions, input parameters and flags, side effects and changes in global variables, these are practically untestable. Now if we want to refactor a subroutine, we should write tests for it, but the tests are almost impossible to write, if we don't refactor it first. In these situations, we usually chose to refactor carefully and later, when the smaller extracted subroutines become stabilized, we write tests for them.

The last problem we tried to solve is how to test the GUI elements. Since the test suite has to be automatic, no user intervention should be needed in order to run the tests. User choices in simple dialogs can be emulated by generating events in Tk, but testing the results of drawing trees and other elements of GUI is more difficult. The area of GUI testing is rather complex and it is beyond the scope of the thesis.

6. Future Work

TrEd is a complex program with many sophisticated features. The complete refactoring is, unfortunately, beyond the scope of this master thesis. TrEd, and especially its core and API for extensions and macros, would certainly benefit from more tests. The test written during the refactoring cover approximately one fifth of all the subroutines in TrEd. Not all of the rest of the subroutines are, however, easily testable, because some of them require user interaction or further refactoring. Creating an extensive testing suite and framework for testing on various platforms and Perl versions would greatly improve the quality of TrEd.

Further refactorings could also improve the separation of levels of abstraction in the source code. Some modules (e.g. `TrEd::File`) take care of all the aspects of opening file from the data model (creating the underlying `Treex::PML::Document` objects) to some features of presentation domain (creating dialogs and prompts when the user input or intervention is needed). Especially bTrEd could benefit from the separation of concerns. If there would be a common subset of operations needed to be done both in TrEd and bTrEd for tasks like opening a file or loading macros, a new layer representing only the data model, without the dependency on GUI elements and Tk library could be introduced.

Taking the modulation of TrEd even further could allow changing the drawing algorithm of trees, making them more flexible or interactive.

The conceptual change in the way extensions work could also improve the overall design of the application. A base class for all the extensions could be created. The extensions could then be implemented as subclasses of the base class. They could override the general behaviour with specific implementations or simply extend the functionality of the base class.

In longer run, TrEd could be rewritten using objects, e.g. relatively new, but popular `Moose` object system.

Standardization of the release process, either by transforming TrEd into CPAN package, or by creating distribution-specific packages would make the installation procedure less error-prone and dangerous for the user. The possibility of breaking the target system by overwriting system libraries would be considerably reduced.

TrEd would also benefit from a standardized debugging system which would allow users to send relevant error reports, in case something breaks while working with the program.

7. Conclusion

The main goal of this thesis was refactoring of Tree Editor TrEd. The most important requirements were to increase modularity, maintainability and improve the understandability of the program. To quantify these concerns, a thorough analysis of source code and TrEd's behaviour has been made. We have seen that even though most of TrEd's source code was well-behaved, there were some acrid areas which needed special attention. One of these areas was the `main` package of TrEd itself, which was more than 13,000 lines long and contained more than 350 subroutines. To improve modularity, 50 new Perl packages were created by extracting existing subroutines from other packages and grouping them together by their concerns. More than 230 subroutines and related data structures were moved between packages. To demonstrate the increase of modularity and avoid future regressions in code base, more than 1,300 test cases were created. To improve the understandability of TrEd, approximately 15,000 lines of Plain Old Documentation was written.

Automatic tools were employed to analyze the code base of TrEd. The number of most severe violations of rules from [2] was reduced by 42 %. The cyclomatic complexity of most complex subroutines in TrEd was reduced by 20 % on average.

These numbers, of course, does not automatically mean that the quality of TrEd's source code has improved and the internal structure of TrEd is clearer. However, we believe, that the reorganization made the program easier to comprehend, which would allow faster pace of implementing new features and reduce the time needed for debugging in the future. We think, that the test suite written for TrEd can prove very valuable in the future, because an automatic testing environment for TrEd is expected soon.

Since TrEd was the main annotation tool for Prague Dependency Treebank and Prague Arabic Treebank, and is going to be included in future releases of Prague Czech-English Dependency Treebank, and possibly other treebanks as well, the quality requirements are high and we hope this thesis proves to be a useful step in TrEd's development.

References

[1] Bunce, T. *Devel::NYTProf Documentation* [online]. c2010, Revised December 1, 2010 [cit. 2011-07-27]. URL: <http://search.cpan.org/dist/Devel-NYTProf/>.

[2] Conway, Damian. *Perl Best Practices*. 1st ed. Sebastopol: O'Reilly Media, Inc., 2005. ISBN: 0-596-00173-8.

[3] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Kent Beck, John Brant, William Opdyke, Don Roberts. 1st ed. Boston: Addison-Wesley, 1999. ISBN: 0-201-48567-2

[4] Jones, Derek M. *Operand names influence operator precedence decisions (part 1 of 2)* [online]. c2008, Revised November 24, 2009 [cit. 2011-05-15] URL: <http://www.knosof.co.uk/cbook/accu07.html>.

[5] Lippert, Martin. *Refactoring in Large Software Projects*. Stephen Roock. 1st ed. Chichester: John Wiley & Sons Ltd., 2006. ISBN: 0-470-85892-3.

[6] McCabe, Thomas J. *A Complexity Measure*. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO.4, DECEMBER 1976

[7] McConnell, Steven C. *Code Complete*. 2nd ed. Redmond: Microsoft Press, 2004. ISBN: 0-735-61967-0.

[8] Pajas, P. *TrEd Documentation* [online]. c2011, Revised July 7, 2011 [cit. 2011-07-28]. URL: <http://ufal.mff.cuni.cz/ pajas/tred/ar01-toc.html/>.

[9] Pajas, P. *Treex::PML Documentation* [online]. c2010, Revised May 24, 2011 [cit. 2011-07-26]. URL: <http://search.cpan.org/dist/Treex-PML/>.

[10] Wall, Larry. *Programming Perl*. Tom Christiansen & Jon Orwant. 3rd ed. Sebastopol: O'Reilly & Associates, Inc., 2000. ISBN: 0-596-00027-8.

[11] *Code Conventions for the Java™Programming Language* [online]. c1995, Revised April 20, 1999 [cit. 2011-05-22]. URL: <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>.

[12] *PBP Module Recommendation Commentary* [online]. c2008, Revised October 24, 2010 [cit. 2011-06-09]. URL: <https://www.socialtext.net/perl5/pbp_module_recommendation_commentary>.

List of Tables

Appendices

A. TrEd::FileLock

Possible outcomes of `check_lock` function – the file (and associated lock) could be:

- opened by us ignoring the original lock, but later locked again
- opened by us ignoring the original lock and later changed by the lock owner
- opened by us ignoring the original lock, by a user who still owns the lock, but has not saved the file since
- stolen and changed (previously locked by someone else)
- stolen, but not yet changed (previously locked by someone else)
- changed by another program
- ours
- locked by someone else
- opened by us ignoring a lock by another user, who released the lock, but the file has changed since
- opened by us ignoring a lock by another user, who released the lock without making any changes
- changed by another program and our lock was removed
- originally locked by us, but the lock was stolen from us by an unknown user

B. TrEd::Undo

The complete list of undo types used in `TrEd::Undo` module:

- UNDO_ACTIVE_NODE
- UNDO_ACTIVE_ROOT
- UNDO_DATA_AND_TREE_ORDER
- UNDO_TREE_ORDER
- UNDO_DISPLAYED_TREES
- UNDO_CURRENT_TREE_AND_TREE_ORDER
- UNDO_ACTIVE_ROOT_AND_TREE_ORDER
- UNDO_DATA
- UNDO_CURRENT_TREE

Situations when the undo is performed and what type of undo frame is stored on the undo stack:

- create new node via main menu: UNDO_ACTIVE_NODE
- remove active node via main menu: UNDO_ACTIVE_ROOT
- insert new tree after the current tree via main menu: UNDO_TREE_ORDER
- insert new tree before the current tree via main menu: UNDO_TREE_ORDER
- move current tree backward via main menu: UNDO_TREE_ORDER
- move current tree forward via main menu: UNDO_TREE_ORDER
- remove whole current tree via main menu: UNDO_CURRENT_TREE_AND_TREE_ORDER
- edit attributes via `TrEd::Dialog::EditAttributes`: UNDO_DATA
- SaveUndo subroutine in `TrEd::MacroAPI::Default`: user-driven
- make current node the root via main menu: UNDO_ACTIVE_ROOT_AND_TREE_ORDER
- (with undo variant of) hook evaluation: UNDO_DISPLAYED_TREES
- (with undo variant of) macro evaluation: UNDO_DISPLAYED_TREES
- node release event: UNDO_DISPLAYED_TREES

C. TrEd::Macros

Hooks which use saving state by employing undo functions:

- node_doubleclick_hook
- node_click_hook
- node_motion_hook
- node_release_hook
- text_doubleclick_hook
- text_click_hook
- line_click_hook

D. TrEd Refactoring

List of all functions which changed their location in TrEd during refactoring:

Name before	Package before	Package after	Name after
uniq	main	Utils	uniq
%backend_map	main	TrEd::Config	%backend_map
@open_types	main	TrEd::Config	@open_types
%save_types	main	TrEd::Config	%save_types
%vertical_key_arrow_map	main	TrEd::Config	%vertical_key_arrow_map
%context_override_binding	main	TrEd::Binding::Default	%context_override_binding
%default_binding	main	TrEd::Binding::Default	%default_binding
createCmdLineFilelists	main	TrEd::ManageFilelists	createCmdLineFilelists
createBookmarksFilelist	main	TrEd::Bookmarks	create_bookmarks_filelist
resolve_default_binding	main	TrEd::Binding::Default	ine...
default_binding	main	TrEd::Binding::Default	_run_binding
change_default_binding	main	TrEd::Binding::Default	change_binding
get_default_binding	main	TrEd::Binding::Default	get_binding
get_open_filename	main	TrEd::Dialog::File::Open	get_open_filename
bookmarkFilelist	main	TrEd::Bookmarks	bookmark_filelist
lastFileNo	main	TrEd::Window	last_file_no
currentFileNo	main	TrEd::Window	current_file_no
update_sidepanel_filelist_view	main	TrEd::Filelist::View	update
update_a_filelist_view	main	TrEd::Filelist::View	update_a_filelist_view
filelistFullFileName	main	TrEd::Filelist::Navigation	filelist_full_filename
_filelistFullFileName	main	TrEd::Filelist::Navigation	_filelist_full_filename
nextOrPrevFile	main	TrEd::Filelist::Navigation	next_or_prev_file
nextRealFile	main	TrEd::Filelist::Navigation	next_real_file
prevRealFile	main	TrEd::Filelist::Navigation	prev_real_file
nextFile	main	TrEd::Filelist::Navigation	next_file
prevFile	main	TrEd::Filelist::Navigation	prev_file
tieNextFile	main	TrEd::Filelist::Navigation	tie_next_file
tiePrevFile	main	TrEd::Filelist::Navigation	tie_prev_file
tieGotoFile	main	TrEd::Filelist::Navigation	tie_go_to_file
setWindowFile	main	TrEd::Window	set_current_file
resumeFile	main	TrEd::File	resume_file
isFocused	main	TrEd::Window	is_focused
initAppData	main	TrEd::File	init_app_data
setFSLockInfo	main	TrEd::FileLock	set_fs_lock_info
setLock	main	TrEd::FileLock	set_lock
readLock	main	TrEd::FileLock	read_lock
removeLock	main	TrEd::FileLock	remove_lock
checkLock	main	TrEd::FileLock	check_lock
closeFile	main	TrEd::File	close_file
textDialog	main	TrEd::Dialog::Text	create_dialog
userQuery	main	TrEd::Query::User	new_query
addToRecent	main	TrEd::RecentFiles	add_file
getNodeNo	main	TrEd::Window::TreeBasics	get_node_no
bookmarkThis	main	TrEd::Bookmarks	actual_position
addBookmark	main	TrEd::Bookmarks	bookmark_actual_position
lastActionBookmark	main	TrEd::Bookmarks	last_action_bookmark
updateBookmarks	main	TrEd::Bookmarks	update_bookmarks
createFilelistsMenu	main	TrEd::ManageFilelists	createFilelistsMenu
_makeNewFilelist	main	TrEd::ManageFilelists	makeNewFilelist
updateTitle	main	main	update_title_and_buttons
newFileFromCurrent	main	TrEd::File	new_file_from_current
openStandaloneFile	main	TrEd::File	open_standalone_file
reloadFile	main	TrEd::File	reload_file
_new_status	main	TrEd::File	_new_status
loadFile	main	TrEd::File	load_file
merge_status	main	TrEd::File	merge_status
openFile	main	TrEd::File	open_file
lockFile	main	TrEd::FileLock	lock_file
lockOpenFile	main	TrEd::FileLock	lock_open_file
openSecondaryFiles	main	TrEd::File	open_secondary_files
saveFile	main	TrEd::File	save_file
get_value_line	main	TrEd::ValueLine	get_value_line
set_value_line	main	TrEd::ValueLine	set_value_line
update_status_info	main	TrEd::StatusLine	update_status
_u_sort	main	TrEd::MinMax	underscore_sort
update_context_list	main	TrEd::Menu::Context	update_context_list
update_macro_menus	main	TrEd::Menu::Macro	update_macro_menus

Table D.1: TrEd code flow overview

Name before	Package before	Package after	Name after
updateCurrentContextMenu	main	TrEd::Menu::Macro	updateCurrentContextMenu
update_macrolist_view	main	TrEd::List::Macros	update_view
update_attribute_view	main	TrEd::SidePanel	update_attribute_view
toggle_attribute_view_hide_empty	main	TrEd::SidePanel	toggle_attribute_view_hide_empty
update_status_line	main	TrEd::StatusLine	_update_status_line
set_status_line	main	TrEd::StatusLine	_set_status_line
update_value_line	main	TrEd::ValueLine	update
get_nodes_win	main	TrEd::Window	get_nodes
selectFilelistNoUpdate	main	TrEd::ManageFilelists	selectFilelistNoUpdate
selectFilelist	main	TrEd::ManageFilelists	selectFilelist
findFilelist	main	TrEd::ManageFilelists	find_filelist
addFilelist	main	TrEd::ManageFilelists	add_filelist
looseFilePositionInFilelist	main	Filelist	loose_position_of_file y
switchFilelist	main	TrEd::Dialog::Filelist	switch_filelist
createFilelistBrowseEntry	main	TrEd::Dialog::Filelist	createFilelistBrowseEntry
getFilelistLinePosition	main	TrEd::Dialog::Filelist	getFilelistLinePosition
insertToFilelist	main	TrEd::ManageFilelists	insertToFilelist
removeFromFilelist	main	TrEd::ManageFilelists	removeFromFilelist
createNewFilelist	main	TrEd::ManageFilelists	createNewFilelist
addNewFilelist	main	TrEd::ManageFilelists	add_new_filelist
deleteFilelist	main	TrEd::ManageFilelists	deleteFilelist
filelistEntryPath	main	TrEd::ManageFilelists	filelistEntryPath
feedHListWithFilelist	main	TrEd::Dialog::Filelist	feedHListWithFilelist
selectFilelistDialog	main	TrEd::ManageFilelists	selectFilelistDialog
bookmarkToFilelistDialog	main	TrEd::ManageFilelists	bookmarkToFilelistDialog
removeFilelistsDialog	main	TrEd::ManageFilelists	removeFilelistsDialog
loadFilelist	main	TrEd::ManageFilelists	loadFilelist
filelistDialog	main	TrEd::Dialog::Filelist	create_dialog
generateSortedMacroTable	main	TrEd::List::Macros	sorted_macro_table
macrolistCreateItems	main	TrEd::List::Macros	create_items
createMacroList	main	TrEd::List::Macros	create_list
macrolistDialog	main	TrEd::Dialog::MacroList	create_dialog
copyTreesDialog	main	TrEd::Dialog::CopyTrees	create_dialog
initTTFonts	main	TrEd::Dialog::Print	initTTFonts
updatePrintDialogState	main	TrEd::Dialog::Print	updatePrintDialogState
_fix_combo_box_return	main	TrEd::Dialog::Print	_fix_combo_box_return
_fix_combo_box	main	TrEd::Dialog::Print	_fix_combo_box
printDialog	main	TrEd::Dialog::Print	printDialog
savePrintConfig	main	TrEd::Dialog::Print	savePrintConfig
getWindowPatterns	main	TrEd::Window	get_patterns
getWindowHint	main	TrEd::Window	get_hint
getWindowContextRE	main	TrEd::Window	get_context_RE
update_value_line_current	main	TrEd::ValueLine	update_current
cascadeMenus	main	TrEd::Menu::Macro	cascadeMenus
keyBind	main	TrEd::Binding::Default	normalize_key
startMain	main	main	create_split_windows
startMain	main	main	populate_recent_files_menu
startMain	main	main	create_toolbar_buttons
startMain	main	main	create_stylesheet_menu
startMain	main	main	create_value_line_subframe
newUserToolbar	main	TrEd::Toolbar::User::Manager	create_new_user_toolbar
updateToolbarMenu	main	TrEd::Toolbar::User::Manager	_updateToolbarMenu
toggleUserToolbar	main	TrEd::Toolbar::User	toggle_user_toolbar
hideUserToolbar	main	TrEd::Toolbar::User	hide
showUserToolbar	main	TrEd::Toolbar::User	show
userToolbarVisible	main	TrEd::Toolbar::User	visible
getUserToolbar	main	TrEd::Toolbar::User	get_user_toolbar
removeUserToolbar	main	removed	removed
getConfigFromFile	main	TrEd::Config	get_config_from_file
saveRuntimeConfig	main	TrEd::Config	save_runtime_config
updateRuntimeConfig	main	TrEd::Config	update_runtime_config
RepeatedShowDialog	main	TrEd::Dialog::FocusFix	repeated_show_dialog
ShowDialog	main	TrEd::Dialog::FocusFix	show_dialog
fileDialog	main	TrEd::Dialog::File::Open	show_dialog
urlDialog	main	TrEd::Dialog::URL	create_dialog
askSaveReferences	main	TrEd::File	ask_save_references
saveFileAs	main	TrEd::File	save_file_as
doSaveFileAs	main	TrEd::File	do_save_file_as
renameFileInFilelist	main	Filelist	rename_file
askSaveFiles	main	TrEd::File	ask_save_files_and_close
closeAllFiles	main	TrEd::File	close_all_files
askSaveFile	main	TrEd::File	ask_save_file
saveConfig	main	TrEd::Config	save_config
editConfig	main	TrEd::Dialog::EditConfig	show_dialog
QueryString	main	TrEd::Query::String	new_query
Query	main	TrEd::Query::Simple	new_query
listQuery	main	TrEd::Query::List	new_query
getApplicableContexts	main	main	get_allowed_contexts
deleteStylesheet	main	TrEd::Stylesheet	delete_stylesheet
selectValuesDialog	main	TrEd::Dialog::SelectValues	create_dialog
reloadSentenceView	main	TrEd::View::Sentence	reload_view
sentViewSelectAll	main	TrEd::View::Sentence	select_all_sentences
sentViewSelectNone	main	TrEd::View::Sentence	select_none
sentViewGetSelection	main	TrEd::View::Sentence	get_selection
sentViewToggleCollapse	main	TrEd::View::Sentence	toggle_collapse

Table D.2: TrEd code flow overview

Name before	Package before	Package after	Name after
viewSentences	main	TrEd::View::Sentence	show_sentences
viewSentencesDialog	main	TrEd::View::Sentence	show_sentences_dialog
openSimpleHtml	main	TrEd::HTML::Simple	open
closeSimpleHtml	main	TrEd::HTML::Simple	close
dumpSentView	main	TrEd::View::Sentence	dump_view
populateSentencesDialog	main	TrEd::View::Sentence	populate_dialog
editFilePropertiesDialog	main	TrEd::Dialog::FileProperties	create_dialog
editAttrsDialog_schema	main	TrEd::Dialog::EditAttributes	dialog_schema
editAttrsDialog	main	TrEd::Dialog::EditAttributes	create_dialog
format_tred_pod	main	TrEd::Dialog::EditStylesheet	format_tred_pod
tred_pod_add_tags	main	TrEd::Dialog::EditStylesheet	tred_pod_add_tags
_sytylesheetInsertAttr	main	TrEd::Dialog::EditStylesheet	_sytylesheetInsertAttr
editStylesheetDialog	main	TrEd::Dialog::EditStylesheet	show_dialog
findNodeDialog	main	TrEd::Dialog::FindNode	findNodeDialog
redraw_win	main	TrEd::Window	redraw
ensureCurrentIsDisplayed	main	TrEd::Window	ensureCurrentIsDisplayed
treeIsVertical	main	TrEd::Window::TreeBasics	tree_is_vertical
treeIsReversed	main	TrEd::Window::TreeBasics	tree_is_reversed
redraw_win	main	TrEd::Window	redraw
centerToXY	main	main	center_to_coords
examineEvent	main	TrEd::Dialog::ExamineBindings	examineEvent
examineBindingsDialog	main	TrEd::Dialog::ExamineBindings	create_dialog
findMacroDescription	main	TrEd::Macros	findMacroDescription
prepare_undo	main	TrEd::Undo	prepare_undo
prepare_redo	main	TrEd::Undo	prepare_redo
save_undo	main	TrEd::Undo	save_undo
re_do	main	TrEd::Undo	re_do
undo	main	TrEd::Undo	undo
resetUndoStatus	main	TrEd::Undo	reset_undo_status
declareMinorMode	main	TrEd::MinorModes	declare_minor_mode
enableMinorMode	main	TrEd::MinorModes	enable_minor_mode
disableMinorMode	main	TrEd::MinorModes	disable_minor_mode
toggleMinorMode	main	TrEd::MinorModes	toggle_minor_mode
_minor_ctxt_abbrev	main	TrEd::MinorModes	_minor_ctxt_abbrev
configure_minor_mode	main	TrEd::MinorModes	configure_minor_mode
update_minor_modes	main	TrEd::MinorModes	update_minor_modes
update_minor_mode_menu	main	TrEd::MinorModes	update_minor_mode_menu
valueLineClick	main	TrEd::ValueLine	_click
prepareExtensions	main	TrEd::Extensions	prepare_extensions
loadStdFilelists	main	TrEd::ManageFilelists	loadStdFilelists
saveStdFilelist	main	TrEd::ManageFilelists	saveStdFilelist
uniq	TrEd::Basics	TrEd::Utils	uniq
gotoTree	TrEd::Basics	TrEd::Window::TreeBasics	go_to_tree
nextTree	TrEd::Basics	TrEd::Window::TreeBasics	next_tree
prevTree	TrEd::Basics	TrEd::Window::TreeBasics	prev_tree
newTree	TrEd::Basics	TrEd::Window::TreeBasics	new_tree
newTreeAfter	TrEd::Basics	TrEd::Window::TreeBasics	new_tree_after
pruneTree	TrEd::Basics	TrEd::Window::TreeBasics	prune_tree
moveTree	TrEd::Basics	TrEd::Window::TreeBasics	move_tree
makeRoot	TrEd::Basics	TrEd::Window::TreeBasics	make_root
newNode	TrEd::Basics	TrEd::Window::TreeBasics	new_node
pruneNode	TrEd::Basics	TrEd::Window::TreeBasics	prune_node
setCurrent	TrEd::Basics	TrEd::Window::TreeBasics	set_current
errorMessage	TrEd::Basics	TrEd::Error::Message	error_message
_messageBox	TrEd::Basics	TrEd::Error::Message	_message_box
absolutize_path	TrEd::Basics	TrEd::File	absolutize_path
absolutize	TrEd::Basics	TrEd::File	absolutize
fileSchema	TrEd::Basics	TrEd::File	**removed
getSecondaryFiles	TrEd::Basics	TrEd::File	get_secondary_files
getSecondaryFilesRecursively	TrEd::Basics	TrEd::File	get_secondary_files_recursively
getPrimaryFiles	TrEd::Basics	TrEd::File	get_primary_files
getPrimaryFilesRecursively	TrEd::Basics	TrEd::File	get_primary_files_recursively
_inst_version	TrEd::Extensions	TrEd::File	moved to get_module_version
setExtension	TrEd::Extensions	TrEd::File	update_extensions_list
@stylesheetPaths	TrEd::Utils	TrEd::Stylesheet	stylesheet_paths
$defaultStylesheetPath	TrEd::Utils	TrEd::Stylesheet	default_stylesheet_path
loadStyleSheets	TrEd::Utils	TrEd::Stylesheet	load_stylesheets
initStylesheetPaths	TrEd::Utils	TrEd::Stylesheet	init_stylesheet_paths
readStyleSheets	TrEd::Utils	TrEd::Stylesheet	read_stylesheets
readStyleSheetsNew	TrEd::Stylesheet	TrEd::Stylesheet	_read_stylesheets_new
readStyleSheetsOld	TrEd::Stylesheet	TrEd::Stylesheet	_read_stylesheets_old
saveStyleSheets	TrEd::Utils	TrEd::Stylesheet	save_stylesheets
removeStylesheetFile	TrEd::Utils	TrEd::Stylesheet	remove_stylesheet_file
readStyleSheetFile	TrEd::Utils	TrEd::Stylesheet	read_stylesheet_file
saveStyleSheetFile	TrEd::Utils	TrEd::Stylesheet	save_stylesheet_file
getStylesheetPatterns	TrEd::Utils	TrEd::Stylesheet	get_stylesheet_patterns
setStylesheetPatterns	TrEd::Utils	TrEd::Stylesheet	set_stylesheet_patterns
updateStylesheetMenu	TrEd::Utils	TrEd::Menu::Stylesheet	update
getStylesheetMenuList	TrEd::Utils	TrEd::Stylesheet	get_menu_list
applyWindowStylesheet	TrEd::Utils	TrEd::Window	apply_stylesheet
splitPatterns	TrEd::Utils	TrEd::Stylesheet	split_patterns
apply_initial_config	main	removed	
set_config	main	main	apply_initial_config
dirname	TrEd::Convert	TrEd::File	dirname
filename	TrEd::Convert	TrEd::File	filename
everything	Tk::EditableCanvas	removed	-

Table D.3: TrEd code flow overview

E. Contents of The Attached CD

The attached CD contains TrEd in its original version, before refactoring and the refactored version.

The original version is present in directory `tred_original` and contains installer for windows, packages for linux and Mac OS X and an automatic install script for these platforms.

Windows installer:

- `tred_wininst_1.4639_small.exe`

Linux and MacOS installers:

- `install_tred.bash` – the installation script
- `tred-current.tar.gz` – package with TrEd version 1.4639
- `tred-dep-unix.tar.gz` – dependencies of TrEd

The refactored TrEd is present in directory `tred_refactored`. It contains two installer for Windows platform, two packages for linux and Mac OS X and an automatic installer script for these platforms. Please be aware that the installation script downloads fresh packages from the website, the packages for UNIX platforms has to be installed manually.

Windows installers:

- `tred-installer.exe` – TrEd and its dependencies
- `tred-installer-perl-included.exe` – TrEd, its dependencies and Strawberry Perl 5.12

Linux and MacOS installers:

- `install_tred.bash` – the installation script
- `tred-current.tar.gz`
- `tred-dep-unix.tar.gz`

F. How To Make a Release of TrEd

A standard UNIX `make` utility is used to release TrEd. The first, necessary step is to check out a copy of svn:
`svn co https://svn.ms.mff.cuni.cz/svn/TrEd_refactored/`.

Then a configuration file `admin/env.sh` needs to be edited. The configurable options are placed at the top of the file:

- `INSTALL_BASE` – the directory, under which TrEd is installed during the release
- `WWW` – path to temporary local copy of www directory
- `REMOTE_WWW` – path to remote www directory, where the TrEd and its documentation would be uploaded
- `PROJECT_DIR` – TrEd directory (the directory, from which the Makefile is executed)
- `LOG` – path to a log file used for logging of svn checkouts during the release

A makefile uses a set of shell scripts to manage the process of releasing TrEd. All the scripts can be invoked individually, but they may, however, have some dependencies. It is therefore strongly encouraged to use the Makefile instead of running the scripts individually.

The process of releasing TrEd consists of the following steps:

1. new version of TrEd is installed in the directory specified by `INSTALL_BASE`
2. new version of `Treex::PML` library is prepared
3. dependencies of TrEd are downloaded and and build
4. extension packages are build
5. TrEd is uploaded to `REMOTE_WWW`