Mascot 1.0

Reference Manual

August 29, 2012

i

Contents

Chapter 1

Overview

1.1 Purpose

Mascot is a style-checking tool for OCaml[1] sources. It is deeply inspired by tools like Check-Style[2]. Its name stems from the following acronym: *Mascot is an Adaptable Style-Checking Ocaml Tool.*

Style-checking tools are useful to enforce conventions about how a program is actually written. The OCaml compiler with both its errors and warnings reports ensures conformity to typing rules, Mascot aims to help the (team of) developer(s) to ensure conformity to style rules.

However, style being a matter of taste, the tool is highly configurable and each (team of) developer(s) is able to choose the style rules it want to conform to. A plugin system also allows one to develop its own style checks. Style errors can be reported using a variety of format, ranging from bare text to CheckStyle-compatible XML (useful to provide integration with third-party tools such as the continuous integration Jenkins engine available at http://jenkins-ci.org).

1.2 License

Mascot is distributed under the terms of the GPL version 3. This licensing scheme should not cause any problem, as the tool is only used during development and will not contamine code.

1.3 Contributions

In order to improve the project, I am primarily looking for testers and bug reporters. Pointing errors in documentation and indicating where it should be enhanced is also very helpful.
Bugs and feature requests can be made at http://bugs.x9c.fr.
Other requests can be sent to mascot@x9c.fr.

---

[1]The official OCaml website can be reached at http://caml.inria.fr and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.
[2]Style checker for Java sources, available at http://checkstyle.sourceforge.net.

Chapter 2

Building Mascot

2.1   Step 0: dependencies

Before starting to build Mascot, one first has to check that dependencies are already installed. The following elements are needed in order to build Mascot:

- OCaml, version 4.00.0;

- `make`, in its GNU Make 3.81 flavor;

- a classical Unix shell, such as `bash`;

- **optionally:** Findlib[1], version 1.3.3.

2.2   Step 1: configuration

The configuration of Argot is done by executing `./configure`. One can specify elements if they are not correctly inferred by the `configure` script; the following switches are available:

- `-ocaml-prefix` to specify the prefix path to the OCaml installation (usually `/usr/local`);

- `-ocamlfind` to specify the path to the `ocamlfind` executable;

- `-no-native-dynlink` to disable the build of the native version, even if native dynamic linking is available.

The Java[2] version will be built only if the `ocamljava`[3] compiler is present and located by the makefile. The syntax extension will be compiled only to bytecode.

2.3   Step 2: compilation

The actual build of Mascot is launched by executing `make all`. When build is finished, it is possible to run some simple tests by running `make tests`. Documentation can be generated by running `make doc`.

---

[1]Findlib, a library manager for OCaml, is available at http://projects.camlcity.org/projects/findlib.html.

[2]The official website for the Java Technology can be reached at http://java.sun.com.

[3]OCaml compiler generating Java bytecode, by the same author – http://www.ocamljava.org

2.4   Step 3: installation

Mascot is installed by executing `make install`. According to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`. The actual installation directory depends on the use of `ocamlfind`: if present the files are placed inside the Findlib hierarchy, otherwise they are placed in the directory 'ocamlc -where'/kaputt (*i. e.* `$PREFIX/lib/ocaml/kaputt`).

Chapter 3

Running Mascot

Once installed, Mascot can be run by executing `mascot.byte` (respectively `mascot.opt`) either directly, or through `‘ocamlfind query mascot‘/mascot.byte` (respectively `‘ocamlfind query mascot‘/mascot.opt`) if installed through `ocamlfind`.

A typical invocation of the tool is:
`mascot.byte -config` *configuration-file* `-output` *destination-file* *files-to-analyze*
where:

- *configuration-file* defines the checks to be performed

- *output* defines the format of the generated report (*e.g.* bare text, HTML)

The complete list of available command-line switches is:

- `-I <path>` Add to search path

- `-available` Print available checks

- `-binary <file>` Output report to binary format

- `-checkstyle <file>` Output report to checkstyle format

- `-config <file>` Set configuration file

- `-csv <file>` Output report to csv format

- `-default-config` Print default configuration

- `-disable-cache` Disable cache

- `-html <file>` Output report to html format

- `-ignore <file>` Set ignore file

- `-jdepend <file>` Output report to jdepend format

- `-no-error` Disable report of errors

- `-no-info` Disable report of infos

- `-no-warning` Disable report of warnings

- `-parameters <category.check>` Print check parameters

- `-plugin <file>` Load plugin

- `-preprocessor <command>` Set non-camlp4 preprocessor

- `-report-only` Build report from binary files (no analysis)

- `-summary <file>` Output report to summary format

- `-syntax camlp4o|camlp4of|camlp4oof|camlp4orf|camlp4r|camlp4rf` Set camlp4 preprocessor

- `-syntax-extension <file>` Dynamically load camlp4 syntax extension

- `-text <file>` Output report to text format

- `-version` Print version and exit

- `-xml <file>` Output report to xml format

Chapter 4

Using Mascot

4.1 Configuration file

The configuration file, passed through the `-config` command-line switch, allows one to define the set of checks to perform over the source code base. The contents of the file should conform to the following BNF grammar (aside from comments that follow the OCaml convention):

```
file ::= category_list
category_list ::= category_list category | ε
category ::= category ident { check_list } opt_separator
opt_separator ::= ; | ε
check_list ::= check_list check | ε
check ::= ident = check_value opt_separator
check_value ::= true | false | { parameter_list }
parameter_list ::= parameter_list parameter | ε
parameter ::= ident = parameter_value opt_separator
parameter_value ::= true | false | integer | string | ident | ident_list
ident_list ::= [ ] | [ nonempty_ident_list ]
nonempty_ident_list ::= nonempty_ident_list , ident | ident
```

A `true` value, or a parameter list will enable a given check, while `false` will disable it. By default, all checks are disabled which means that `false` is only useful to explicitly state that a given check is unwanted.

Code sample 1 defines three checks allowing to ensure that no line contains neither more than 78 character nor a tabulation, and that no `open` statement is used in the code.

4.2 Ignore file

As everyone knows, every rule (as wise as it could be) deserves some exceptions. For this reason, one is allowed to supply Mascot with an ignore file indicating those exceptions. The contents of the file should conform to the following BNF grammar (aside from comments that follow the OCaml convention):

Code sample 1 Configuration file.

```
category typography {
  line_length = { maximum = 78; }
  tab_character = true;
}

category code {
  open = true;
}
```

$file ::= ignore\_list$
$ignore\_list ::= ignore\_list \; ignore \mid \epsilon$
$ignore ::= \text{ignore} \; ident \; . \; ident \; filename\_opt \; intervals$
$filename\_opt ::= \text{in} \; string \mid \epsilon$
$intervals ::= \text{at} \; interval\_list \mid \epsilon$
$interval\_list ::= interval\_list \; , \; interval \mid \epsilon$
$interval ::= integer \mid integer \; .. \; \text{end} \mid integer \; .. \; integer$

Code sample 2 defines two exceptions: the presence of `open` statement at a line ranging from 20 to 24 (both inclusive) in the file named `a.ml`, and the presence of magic numbers in the whole `b.ml` file.

Code sample 2 Ignore file.

```
ignore code.open in "a.ml" at 20..24
ignore code.magic_number in "b.ml"
```

4.3   Output formats

Currently, Mascot is bundled with support for eight output formats: CheckStyle-compatible XML, CSV, HTML, JDepend-compatible[1] XML, simple summary, bare text, binary, and XML.

CheckStyle-compatible XML and JDepend-compatible XML formats are provided for easy interface with third-party tools like the continuous integration engine Jenkins. While Mascot performs a new analysis *from scratch* each time it is invoked, the Jenkins plugins for CheckStyle and JDepend will keep an history of the successive reports and will be able to identify resolved and new check violations at each build. Those plugins will also draw history diagrams to give a glimpse of the project evolution.

CSV and XML formats are provided for easy integration with third-party tools that are not CheckStyle- or JDepend-aware. The formats should make integration in a toolchain as easy as data importation, maybe at the price of some transformation.

---

[1]Quality metrics generator for Java source, available at http://www.clarkware.com/software/JDepend.html

The binary format allow to store analysis results to binary files, and to merge results latter and to produce a report using another output format through the `-report-only` command-line switch.

Finally, the summary, bare text, and HTML formats are provided for standalone use of the Mascot tool. The bare text mode should output check violations in a way similar to a compiler would output errors, while the HTML mode alllows easy navigation in the list of check violations. The HTML mode displays some image icons; these have been designed by Mark James, released under the Creative Commons Attribution 2.5 License, and are available at `http://www.famfamfam.com/lab/icons/silk/`.

Chapter 5

Extending Mascot

The developer can extend Mascot in two ways:

- by writing a new output mode;

- by writing new checks.

Writing a new output mode is quite straighforward: it is sufficient to write a module whose type is compatible with `MascotLibrary.Output.T`, and to register it through the `MascotLibrary.Plugin. register_output` function. Developing such a module is as simple as choosing a name for the output mode, and coding a `run` function that will be executed if the output mode is requested from the command-line.

Writing a new check is a bit more involved: first one has to determine the kind of check to perform. Currently, six kinds are supported by the Mascot tool:

- `Lines` for checks that work on source lines (as bare `string`s);

- `Ocamldoc` for checks that work on the output of the ocamldoc tool;

- `Tokens` for checks that work on source tokens (in the lexical meaning);

- `Structure` for checks that work on the syntax tree of a module implementation;

- `Signature` for checks that work on the syntax tree of a module signature;

- `Annotations` for checks that work on annotation files;

- `Binary_annotations` for checks that work on binary annotation files.

For each kind, there is an associated module type (*e.g.* `MascotLibrary.Check.Lines.T`) to conform to in order to develop a new check. For each check, the developer should provide:

- a category;

- a name;

- whether the check can support multiple instanciations;

- a short description;

- a full documentation;

- a rationale;

- (optionally) some limits, that is plausible reasons for exceptions;

- a list of parameter descriptors;

- a function actually performing the check (the signature of the function being of course different according to the check kind).

Then, the module implementing the check can be registered through one of the `MascotLibrary.Plugin. register_xyz_check` function.

More information can be found in the documentation generated in the `ocamldoc` directory built by running `make doc`.

13

Appendix A

Available checks

The following pages list, in alphabetical order, the checks bundled with Mascot. Each one is named in `category.check` form and a description as well as the rationale underlying the check are provided. When applicable, the list of parameters with their types and default values is also given.

code.builtin_types — redefinition builtin of types

Checks that no builtin type is redefined.
Rationale: Redefining a builtin type does not only result in code that is harder to apprehend, it is also not possible anymore to reference the original type.

code.catch_all — catch of all exceptions

Detects patterns catching all exceptions.
Rationale: Either the loop is a waste of time, either some code is missing.
Limits: Patterns with a 'when' clause are not considered as catching all exceptions.

code.complex_condition — complex if/while/when conditions

Checks for complex conditions of 'if', 'while', and 'when' constructs, that is conditions with too many boolean operators
Rationale: Complex conditions make the control flow of a code harder to understand.
Limits: Detection will not be accurate if either a boolean operator is redefined, or a synonym is declared.
Parameters:

- maximum (integer defaulting to 5): maximum number of boolean operators

`code.deprecated` — **deprecated elements**

Detects use of deprecated elements.
Rationale: One should not rely on the elements marked as deprecated in the official library.
Limits: Does not check for deprecated elements outside of the official library.

`code.empty_for` — **empty 'for' constructs**

Detects empty 'for' constructs.
Rationale: Either the loop is a waste of time, either some code is missing.

`code.empty_try` — **empty 'try' constructs**

Detects empty 'try' constructs.
Rationale: As no exception could be thrown, the construct is useless.
Limits: Asynchronous exception may occur.

`code.empty_while` — empty 'while' constructs

Detects empty 'while' constructs.
Rationale: The only use of such a construct is to wait for an external condition to become true. However, this is a wasting-CPU activity.

`code.idempotent_operations` — detects some idempotent operations

This check detects some idempotent operations over 'int' and 'float' types.
Rationale: Obviously idempotent operations are often the hint of a mistake (e. g. use of the digit '0' instead of the letter 'o').
Limits: If a predefined operator is redefined, its associated set of idempotent operations may not be the same.

`code.identifier_length` — identifiers that are too short

Checks that every identifier has a minimum length.
Rationale: Short names often implies that code is harder to read and apprehend, because one has to refer to its binding to remember its exact meaning.
Limits: In some context, a given letter has a very precise meaning that make a longer identifier useless (if not cluttering). Moreover, in very short function it is not always necessary to use long names for parameters.
Parameters:

- `exclude_common` (boolean defaulting to `true`): exclude common names from check

- `minimum` (integer defaulting to `2`): minimum identifier length

`code.ignore_unit` — **ignore of unit expression**

Detects use of 'ignore' over expressions of type 'unit'.

Rationale: The use of 'ignore' over expressions of type 'unit' is superfluous, and may be the hint of an error.

Limits: The 'ignore' function may have been redefined locally.

`code.length` — **elements that are too long**

Checks that no function, class, or module is too long.

Rationale: An element whose length is above a given limit should be refactored into smaller units in order to improve readability.

Parameters:

- `maximum` (integer defaulting to `300`): maximum number of lines

`code.magic_number` — **magic numbers**

Looks for magic numbers.

Rationale: Repeated elements need to be updated at once when a modification is made, leading to possible desynchronization if modification is only made partially. Moreover, it is better to define symbolic constants rather than to repeat the integer constant in various places.

Limits: Constants between -5 and 5 (both inclusive) are ignored by this check, but some other constants may be (at least in some context) self-explanatory. More, in some places (such as patterns) the compiler will require an integral constant, and reject a symbolic constant.

code.negated_if — negated 'if' condition

Detects 'if/then/else' constructs with an associated condition whose top-level function application is a 'not'.
Rationale: For readability reasons, it can be useful to switch branches.
Limits: Does not work if the 'not' function has been redefined.

code.nested_for — nested 'for' constructs

Ensures that there is no deeply nested 'for' loops.
Rationale: Above a given limit, the nesting of 'for' constructs render the code hard to read (and is also the hint of possible loop parallelism).
Parameters:

- `maximum` (integer defaulting to `3`): maximum nested for loops

code.nested_if — nested 'if' constructs

Ensures that there is no deeply nested 'if' constructs.
Rationale: Above a given limit, the number of paths generated by 'if' constructs renders the code hard to comprehend.
Parameters:

- `maximum` (integer defaulting to `8`): maximum paths

`code.nested_try` — nested 'try' constructs

Ensures that there is no deeply nested 'try' constructs.
Rationale: Above a given limit, the nesting of 'try' constructs renders the control flow hard to comprehend.
Parameters:

- `maximum` (integer defaulting to 3): maximum nested try constructs

`code.nested_while` — nested 'while' constructs

Ensures that there is no deeply nested 'while' loops.
Rationale: Above a given limit, the nesting of 'while' constructs render the code hard to read.
Parameters:

- `maximum` (integer defaulting to 3): maximum nested while loops

`code.no_effect_assignment` — detects assignments with no effect

Detects useless assignments (e. g. 'x := x').
Rationale: Assignments with no effect are either the sign of an error, or the hint of possible simplification
Limits: This check does not protect against cycles of no-effect assignments.

`code.open` — **open statements**

Disallow the presence of open statements.

Rationale: Open statements make source code difficult to read, because it is harder to know where a given name is bound.

Limits: Disallowing open statements lead to longer code because every external reference has to be qualified.

Parameters:

- `allowed_modules` (symbol list defaulting to `[]`): modules allowed in open statements

`code.parameter_count` — **functions with too many parameters**

Enforces that functions have a reasonable number of parameters.

Rationale: Functions with too many parameters provide difficult to use APIs. In order to increase readability, one is advised to group parameters into data structures such as records.

Parameters:

- `maximum` (integer defaulting to `8`): maximum number of parameters

`code.physical_comparisons` — **detects physical comparisons**

This check detects the use of either '==' or '!='.

Rationale: Novices coming from a C/C++/Java background often use them for a comparison that should be done through either '=' or '¡¿'.

Limits: If any of the comparison operators has been redefined, the check will yield false positives. Moreover, in some contexts, it is perfectly legitimate to use physical comparisons.

`code.predefined_exceptions` — redefinition of predefined exceptions

Checks that no predefined exception is redefined.

Rationale: Redefining an exception with the same name as a predefined one leads to code that is difficult to read because the name is actually overloaded and only careful scope analysis can determine which exception is actually raised or caught at a given code point.

`code.predefined_operators` — redefinition predefined of operators

Checks that no predefined operator is redefined.

Rationale: Replacing a predefined operators leads to code that is difficult to read, as the reader has to carefully remember which operators where redefined. Moreover, some redefinitions replace with implementations that do not have the same mathematical properties (such as commutativity), leading to incorrect assumptions.

`code.string_literals` — duplicate string literals

Looks for string literal duplicates.

Rationale: Repeated elements need to be updated at once when a modification is made, leading to possible desynchronization if modification is only made partially.

Limits: In OCaml, strings are mutable which means that it could be useful (although discouraged) to have multiple instances that are identical at program start but evolve in different manners. The developer may still investigate the possible use a the 'String.copy' function along with literal factorization. However, in some places (such as patterns) the compiler will require an integral constant, and reject a symbolic constant.

Parameters:

- `exclude_empty` (boolean defaulting to `true`): whether the empty string should be ignored

- `threshold` (integer defaulting to `1`): threshold for report

code.textual_duplicate — **textual duplicates**

Checks for the presence of duplicated text.

Rationale: Duplicated text is often due to a copy/paste operation, and if possible should lead to code factorization.

Limits: Being based on text, the check will not identify almost-identical code that differs only in terms of whitespaces or identifiers names.

Parameters:

- `offset` (integer defaulting to `0`): number of lines to ignore at file start

- `trim_lines` (boolean defaulting to `false`): whether to remove leading and trailing whitespaces

- `window` (integer defaulting to `10`): minimum size of duplicates

code.tuple_size — **lengthy tuples**

Detects lengthy tuples.

Rationale: When a tuple has too many members, it should be refactored into a record.

Parameters:

- `maximum` (integer defaulting to `4`): maximum number of members

code.useless_binding — **detects useless bindings**

Detects useless bindings (e. g. 'let x = x').

Rationale: Useless bindings are either the sign of an error, or the hint of possible simplification.

Limits: This check does not protect against useless synonyms.

code.useless_external_parentheses — useless external parentheses

Checks 'if', 'when', 'while', and 'for' constructs do not have superfluous external parentheses.
Rationale: These constructs are themselves 'parenthesized', and accept expression of a fixed type. As a consequence, parentheses hinder readability.

documentation.class_comment — enforce presence of class comments

Enforces that each exported class is commented, in order to produce a fully populated ocamldoc.
Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.
Limits: Some well-named classes need no documentation, and adding a documentation to them may be regarded as undesired noise.
Parameters:

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

documentation.class_type_comment — enforce presence of class type comments

Enforces that each exported class type is commented, in order to produce a fully populated ocamldoc.
Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.
Limits: Some well-named class types need no documentation, and adding a documentation to them may be regarded as undesired noise.
Parameters:

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

documentation.exception_comment — enforce presence of exception comments

---

Enforces that each exported exception is commented, in order to produce a fully populated ocamldoc.

Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.

Limits: Some well-named exceptions need no documentation, and adding a documentation to them may be regarded as undesired noise.

Parameters:

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

documentation.module_comment — enforce presence of module comments

---

Enforces that each exported module is commented, in order to produce a fully populated ocamldoc.

Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.

Limits: Some well-named modules need no documentation, and adding a documentation to them may be regarded as undesired noise.

Parameters:

- `require_author` (boolean defaulting to `false`): enforce presence of @author tag (toplevel only)

- `require_version` (boolean defaulting to `false`): enforce presence of @version tag (toplevel only)

documentation.module_type_comment — enforce presence of module type comments

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

---

Enforces that each exported module type is commented, in order to produce a fully populated ocamldoc.

Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.

Limits: Some well-named module types need no documentation, and adding a documentation to them may be regarded as undesired noise.

Parameters:

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

`documentation.spell` — spell-check the text of ocamldoc comments

Spell-checks the text of ocamldoc comments through an external aspell-compatible spell-checking engine. When a mistake is reported, suggestions may also be proposed in order to facilitate correction.

Rationale: Documentation is the entry-point of library users and/or code base committers, and it is annoying to leave some typos undetected. Moreover, users may feel uncomfortable to report such errors because they appear insignificant while correcting them clearly enhance the codebase quality.

Limits: Some common abbreviations, or even full words may be too specific to appear in the common dictionary. This will require the user to either extend an existing dictionary, or to create a new one.

Parameters:

- `dictionary` (string defaulting to `"en"`): dictionary identifier

`documentation.type_comment` — enforce presence of type comments

- `path` (string defaulting to `"aspell"`): path to aspell-compatible executable

Enforces that each exported type is commented, in order to produce a fully populated ocamldoc.

Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.

Limits: Some well-named types need no documentation, and adding a documentation to them may be regarded as undesired noise.

Parameters:

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

`documentation.value_comment` — enforce presence of value comments

Enforces that each exported value is commented, in order to produce a fully populated ocamldoc.

Rationale: It is difficult to use a library, or to maintain a program if it is not fully and properly documented.

Limits: Some well-named values need no documentation, and adding a documentation to them may be regarded as undesired noise.

Parameters:

- `strict` (boolean defaulting to `false`): whether empty comment should be rejected

`interface.duplicate` — **detect duplicate entries in signatures**

Checks for the duplicates in signatures, that is ones that export elements with the same name. *Rationale:* Signatures may indeed not contain duplicate, keeping only the last of elements sharing the same name.

`interface.exported_count` — **presence of complex structures**

Checks for the presence of complex structures, that is ones that export too many elements. *Rationale:* Structures with many items are difficult to apprehend and are an hint that some refactoring should be performed in order to build smaller units. *Parameters:*

- `maximum` (integer defaulting to `32`): maximum exported elements

`interface.exported_empty` — **presence of empty structures**

Checks for the presence of empty structures, that is ones that export no element. *Rationale:* Empty structures are often the hint of a mistake. *Limits:* Empty structures may be useful if they contain side effects.

`interface.method_count` — **presence of complex classes**

Checks for the presence of complex classes, that is ones that define too many elements.
Rationale: Classes with many elements are difficult to apprehend and are an hint that some refactoring should be performed in order to build smaller classes.
Parameters:

- `maximum` (integer defaulting to `32`): maximum class elements

`metrics.coupling` — **computes dependencies**

Computes the dependencies using the ocamldep tool, and both afferent and efferent couplings.
Rationale: Afferent and efferent couplings allows to identify the role of a module, that is its place in the source base.

`metrics.halstead` — **Halstead complexity**

Computes the Halstead complexity of expressions.
Rationale: The Halstead metric allows to identify functions that are too complex to understand, and thus to maintain.
Limits: As any metric, the Halstead complexity only gives bare hints about whether an expression should be decomposed into smaller elements.

`metrics.mccabe` — McCabe complexity

Computes the McCabe complexity of expressions.

Rationale: The McCabe metric allows to identify functions that are too complex to understand, and thus to maintain.

Limits: As any metric, the McCabe complexity only gives bare hints about whether an expression should be decomposed into smaller elements.

`miscellaneous.regexp` — regular expression matching

Checks whether a line partially matches a given regular expression.

Rationale: A limited but flexible check allowing to look for various potential mistakes, or discouraged practices.

Limits: Matching is limited to individual lines, meaning that it is impossible to match regular expressions spanning multiple lines.

Parameters:

- `case_sensitive` (boolean defaulting to `true`): whether regular expression matching is case sensitive

- `expr` (string defaulting to `""`): regular expression to match

- `level` (symbol defaulting to `error`): report level ('info', 'warning', or 'error')

`typography.file_length` — long files

- `threshold` (integer defaulting to `0`): threshold (number of line matches) for report

Checks for the presence of long files.

Rationale: Long files are difficult to read and master. They are also the hint that related modules should be refactored into smaller and more maintainable units.

Parameters:

- `maximum` (integer defaulting to `500`): maximum file length

`typography.header` — **coherent headers for source files**

Checks that all source files share the same header, as defined in a given file.

Rationale: For copyright (or copyleft) issues, it is usually good practice to start each file with an header indicating both the distribution license, and the list of authors and/or copyright holders.

Limits: The check cannot be used if different files use different header (e. g. due to different licenses).

Parameters:

- `file` (string defaulting to `"/dev/null"`): path to header file

`typography.line_length` — **long lines**

Checks for the presence of long lines.

Rationale: Long lines are notoriously difficult to properly edit and print. Above all, they are difficult to read; it is a common typographic convention to favor short lines in order to increase reading speed.

Parameters:

- `maximum` (integer defaulting to `80`): maximum line length

`typography.spaces_around_blocks` — **white spaces around block delimiters**

Checks that typographic rules about block delimiters are respected.

Rationale: Typographic conventions (in source code, as well as in ordinary text) should be followed because the uniformity they provide allows one to read faster.

typography.spaces_around_operators — **white spaces around operators**

Checks that typographic rules about operators are respected.

Rationale: Typographic conventions (in source code, as well as in ordinary text) should be followed because the uniformity they provide allows one to read faster.

typography.spaces_around_punctuation — **white spaces around punctuation signs**

Checks that typographic rules about punctuation are respected.

Rationale: Typographic conventions (in source code, as well as in ordinary text) should be followed because the uniformity they provide allows one to read faster.

typography.tab_character — **tabulation character**

Checks for the presence of tabulation characters, both in code and constant string literals. In the former case, they should usually be replaced by an escape sequence.

Rationale: Tabulation characters lead to incorrect indentation, and poor portability across editors and printers.

`typography.trailing_new_line` — end-of-file newline

Checks whether the last line of the file is an empty one.
Rationale: End-of-file newlines are not only useless, they may also badly influence page layout when printing.

`typography.trailing_white_space` — end-of-line whitespace

Checks for the presence of whitespaces (either plain spaces or tabulations) at ends of lines.
Rationale: End-of-line whitespaces are not only useless, they may also badly influence page layout when printing. Moreover, they are quite often the leftovers of a refactoring operation.
Parameters:

- `tolerance` (integer defaulting to `0`): how many whitespace characters may be tolerated