Apache Ant dataset

R Analysis document

Boris Baldassari – Castalia Solutions

Contents

Introduction **1**
About the Ant project . . . 1
About this dataset . . . 2
Maisqual Ant weekly . . . 3
Maisqual Ant releases . . . 3

Metrics **4**
Source code metrics . . . 4
Artefact counting metrics . . . 4
Line counting metrics . . . 4
Control flow complexity metrics . . . 4
Halstead metrics . . . 5
Object-oriented measures . . . 5
Configuration management metrics . . . 6
Communication metrics . . . 6

Rule checking **8**
SQuORE . . . 8
Checkstyle . . . 9
PMD . . . 10
Bibliography . . . 11

Using the dataset **11**

Introduction

About the Ant project

The early history of Ant begins in the late nineties with the donation of the Tomcat software from Sun to Apache. From a specific build tool, it evolved steadily through Tomcat contributions to be more generic and usable. James Duncan Davidson announced the creation of the Ant project on the 13 January 2000, with its own mailing lists, source repository and issue tracking. There have been many versions since then: 8 major releases and 15 updates (minor releases). The data set ends in July 2012, and the last version officially released at the time of creating this data set (2012) is 1.8.4. It should be noted that the data set of official releases may show inconsistencies with the weekly data set, since the build process extracts and transforms a subset of the actual repository content.

Ant is arguably one of the most relevant examples of a successful open source project: from 2000 to 2003, the project attracted more than 30 developers whose efforts contributed to nominations for awards and to its recognition as a reliable, extendable and well-supported build standard for both the industry and the open source community. An interesting aspect of the Ant project is the amount of information available on the lifespan of a project — check the plot displayed below: from its early beginnings in 2000, activity had its climax around 2002-2003 and then decreased steadily. Although the project is actively maintained and still

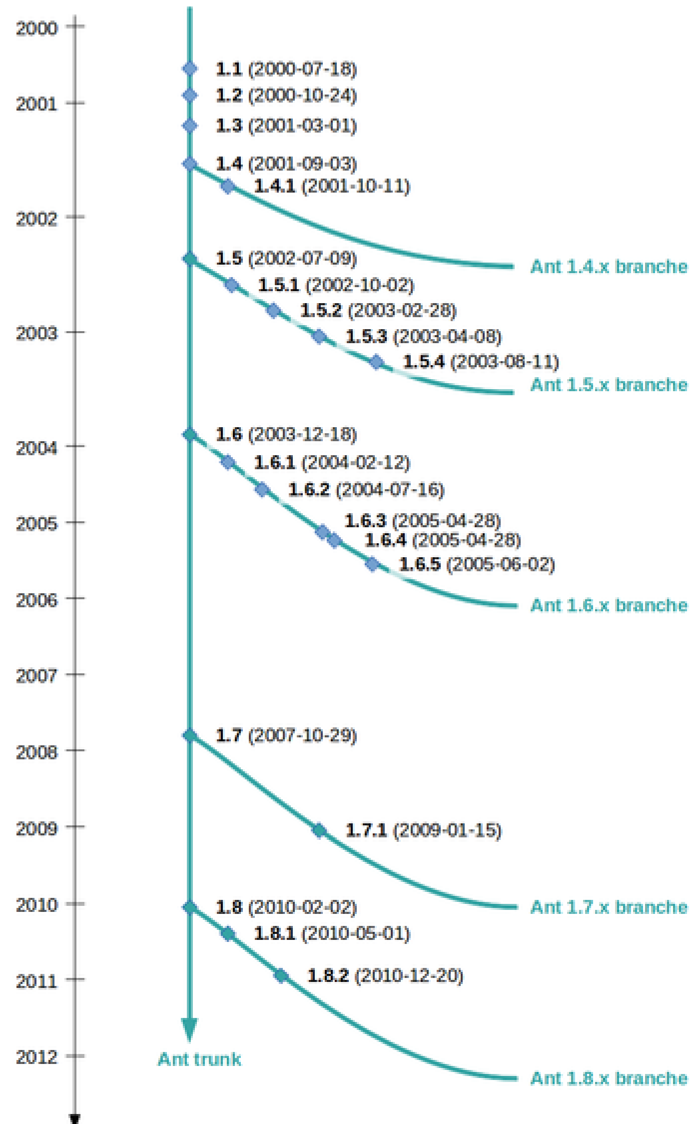


Figure 1: Ant releases

brings regular releases the list of new features is decreasing with the years. It is still hosted by the Apache Software Foundation, which is known to have a high interest in software product and process quality.

About this dataset

The Maisqual Ant data set was published by SQuORING Technologies under an open source licence during the Maisqual research project. It features a number of metrics extracted from code, configuration management and communication channels, plus a bunch of findings from popular rule-checking tools.

There actually two datasets:

- Weekly metrics, from January 2000 to July 2012.
- Release metrics, from first version to 1.8.4 (published in 2012).

Each dataset is composed of 3 files:

- Application-level metrics,
- File-level metrics,
- Function-level metrics.

Maisqual Ant weekly

The Maisqual Ant data set features 636 versions of the Apache Ant project, extracted on every monday since the beginning of the project. You can check the detailed definition of metrics later in the document.

Application-level metrics

- Number of attributes: `161`
- Number of records: `636`

File-level metrics

- Number of attributes: `127`
- Number of records: `654695`

Function-level metrics

- Number of attributes: `121`
- Number of records: `6887472`

Maisqual Ant releases

The Maisqual Ant data set features analysis of the 22 official releases of the Apache Ant project published at the time of writing. Metrics proposed are the same as for the weekly data set, excepted for some time-based metrics which make no sense in an irregularly time-paced list (e.g. number of mails exchanged during last month).

The releases data set can be downloaded from the Maisqual web site, hosted by SQuORING Technologies.

Application-level metrics

- Number of attributes: `120`
- Number of records: `22`

File-level metrics

- Number of attributes: `118`
- Number of records: `18460`

Function-level metrics

- Number of attributes: `118`
- Number of records: `194287`

Metrics

Source code metrics

Artefact counting metrics

Artefact counting metrics are `FILE`, `FUNC` and `CLAS`.

- The number of files (`FILE`) counts the number of source files in the project, i.e. which have an extension corresponding to the defined language (.java for Java or .c and .h files for C).
- The number of functions (`FUNC`) sums up the number of methods or functions recursively defined in the artefact.
- The number of classes (`CLAS`) sums up the number of classes defined in the artefact and its sub-defined artefacts. One file may include several classes, and in Java anonymous classes may be included in functions.

Line counting metrics

Line counting metrics are often used to grasp the size of code from different perspectives. We propose the following counting metrics: `STAT`, `SLOC`, `ELOC`, `CLOC`, `MLOC` and `BRAC`.

- The number of statements (`STAT`) counts the total number of instructions. Examples of instructions include control-flow tokens, plus else, cases, and assignments.
- Source lines of code (`SLOC`) is the number of non-blank and non-comment lines in code.
- Effective lines of code (`ELOC`) also removes the number of lines that contain only braces.
- Comment lines of code (`CLOC`) counts the number of lines that include a comment in the artefact. If a line includes both code and comment, it will be counted in SLOC, CLOC and MLOC metrics.
- Mixed lines of code (`MLOC`) counts the number of lines that have both code and comments
- Braces (`BRAC`) counts the number of lines that contain only braces.

The relationships among these metrics are the following:

```
SLOC = ELOC + BRAC
LC = SLOC + BLAN + CLOC - MLOC
LC = (ELOC + BRAC) + BLAN + CLOC - MLOC
COMR = ((CLOC + MLOC) × 100)/(ELOC + CLOC))
```

Control flow complexity metrics

- The maximum nesting level (`NEST`) counts the highest number of imbricated code (including conditions and loops) in a function. Deeper nesting threatens understandability of code and induces more test cases to run the different branches. Practitioners usually consider that a function with three or more nested levels becomes significantly more difficult for the human mind to apprehend how it works.
- The number of execution paths (`NPAT`) is an estimate of the possible execution paths in a function. Higher values induce more test cases to test all possible ways the function can execute depending on its parameters. An infinite number of execution paths typically indicates that some combination of parameters may cause an infinite loop during execution.
- The cyclomatic number (`VG`), a measure borrowed from graph theory and introduced by McCabe in [XXX] is the number of linearly independent paths that comprise the program. To have good testability and maintainability, McCabe recommends that no program modules (or functions as for Java) should exceed a cyclomatic number of 10. It is primarily defined at the function level and is summed up for higher levels of artefacts.

- The number of control-flow tokens (`CFT`) counts the number of control-flow oriented operators (e.g. if, while, for, switch, throw, return, ternary operators, blocks of execution). else and case are typically considered a part of respectively if and switch and are not counted.

Halstead metrics

Halstead proposed in his Elements of Software Science [82] a set of metrics to estimate some important characteristics of a software. He starts by defining 4 base measures:

- the number of distinct operands (`DOPD` , or $n2$),
- the number of distinct operators (`DOPT` , or $n1$),
- the total number of operands (`TOPD` , or $N2$),
- the total number of operators (`TOPT` , or $N1$).

Together they constitute the following higher-level derived metrics:

- program vocabulary:
$$n = n1 + n2$$
- program length:
$$N = N1 + N2$$
- program difficulty:
$$D = (n1/2).(N2/n2)$$
- program volume:
$$V = Nlog2n$$
- estimated effort needed for program comprehension:
$$E = D\ddot{O}V$$

In the data sets, only the four base measures are retrieved: dopd, dopt, topd, and topt. Derived measures are not provided in the data sets since they can all be computed from the provided base measures. Rules-oriented measures

- `NCC` is the number of non-conformities detected on an artefact. From the practices perspective, it sums the number of times any rule has been transgressed on the artefact (application, file or function).
- The rate of acquired practices (`ROKR`) is the number of respected rules (i.e. with no violation detected on the artefact) divided by the total number of rules defined for the run. It shows the number of acquired practices with regards to the full rule set.

Object-oriented measures

Three measures are only available for object-oriented code. They are the number of classes (`CLAS`), the maximum depth of inheritance tree (`DITM`), and the above-mentionned rate of acquired rules (`ROKR`).

- The number of classes (`CLAS`) sums up the number of classes defined in the artefact and its sub-defined artefacts. One file may include several classes, and in Java anonymous classes may be included in functions.
- The maximum depth of inheritance tree (`DITM`) of a class within the inheritance hierarchy is defined as the maximum length from the considered class to the root of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritance, the `DITM` is the maximum length from the node to the root of the tree [161]. It is available solely at the application level. A deep inheritance tree makes the understanding of the object-oriented architecture difficult. Well structured OO systems have a forest of classes rather than one large inheritance lattice. The deeper the class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior and, therefore, more fault-prone [77]. However, the deeper a particular tree is in a class, the greater potential reuse of inherited methods [161].

- ROKR is another measure specific to object-oriented code, since it is computed relatively to the full number of rules, including Java-related checks. In the case of C projects only the SQuORE rules are checked, so it loses its meaning and is not generated.

Configuration management metrics

Configuration management systems hold a number of meta-information about the modifications committed to the project repository. The following metrics are defined:

- The number of commits (SCM_COMMITS) counts the commits registered by the software configuration management tool for the artefact on the repository (either the trunk or a branch). At the application level commits can concern any type of artefacts (e.g. code, documentation, or web site). Commits can be executed for many different purposes: e.g. add feature, fix bug, add comments, refactor, or even simply re-indent code.
- The number of fixes (SCM_FIXES) counts the number of fix-related commits, i.e. commits that include either the fix, issue, problem or error keywords in their message. At the application level, all commits with these keywords in message are considered until the date of analysis. At the file level, it represents the number of fix-related revisions associated to the file. If a file is created while fixing code (i.e. its first version is associated to a fix commit) the fix isn't counted since the file cannot be considered responsible for a problem that has been detected when it wasn't there.
- The number of distinct committers (SCM_COMMITTERS) is the total number of different committers registered by the software configuration management tool on the artefact. On the one hand, having less committers enforces cohesion, makes keeping coding and naming conventions respected easier, and allows easy communication and quick connection between developers. On the other hand, having a large number of committers means the project is active; it attracts more talented developers and more eyes to look at problems. The project has also better chances to be maintained over the years.
- It should be noted that some practices may threaten the validity of this metric. As an example occasional contributors may send their patches to official maintainers who review it before integrating it in the repository. In such cases, the commit is executed by the official committer, although the code has been originally modified by an anonymous (at least for us) developer. Some core maintainers use a convention stating the name or identifier of the contributor, but there is no established or enforced usage of such conventions. Another point is that multiple online personas can cause individuals to be represented as multiple people [86].
- The number of files committed (SCM_COMMIT_FILES) is the number of files associated to commits registered by the software configuration management tool. This measure allows to identify big commits, which usually imply big changes in the code.

To reflect recent activity on the repository, we retrieved measures both on a limited time basis and on a global basis: in the last week (e.g. SCM_COMMITS_1W), in the last month (e.g. SCM_COMMITS_1M), and in the last three months (e.g. SCM_COMMITS_3M), and in total (e.g. SCM_COMMITS_TOTAL).

Communication metrics

Communication metrics show an unusual part of the project: people's activity and interactions during the elaboration of the product. Most software projects have two communication media: one targeted at the internal development of the product, for developers who actively contribute to the project by committing in the source repository, testing the product, or finding bugs (a.k.a. developers mailing list); and one targeted at end-users for general help and good use of the product (a.k.a. user mailing list).

The type of media varies across the different forges or projects: most of the time mailing lists are used, with a web interface like MHonArc or mod_mbox. In some cases, projects may use as well forums (especially for user-oriented communication) or NNTP news servers, as for the Eclipse foundation projects. The variety of media and tools makes it difficult to be extensive; however data providers can be written to map these to

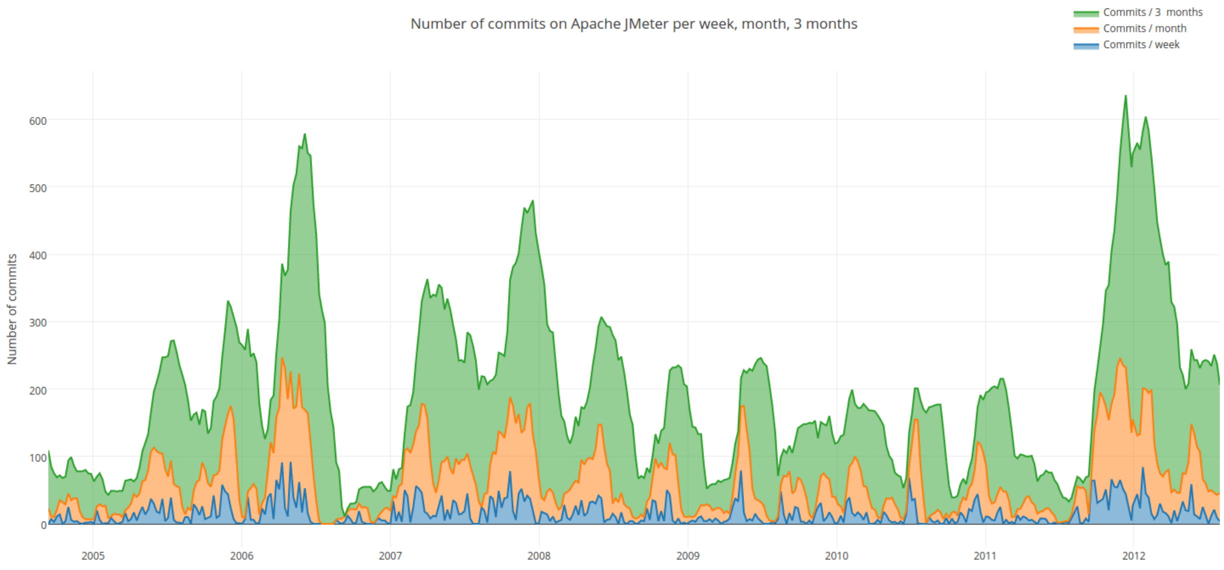


Figure 2: Number of commits on Apache JMeter per week, month, 3 months

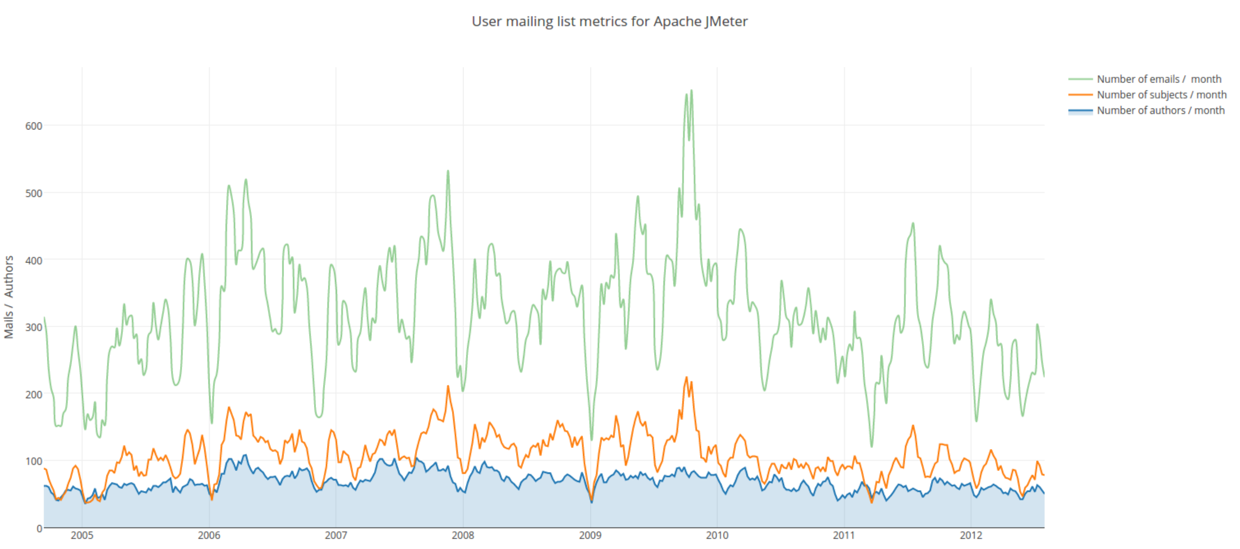


Figure 3: User mailing list metrics for Apache JMeter

the common mbox format. We wrote connectors for mboxes, MHonArc, GMane and FUDForum (used by Eclipse). The following metrics are defined:

- The number of posts (`COM_DEV_VOL` , `COM_USR_VOL`) is the total number of mails posted on the mailing list during the considered period of time. All posts are counted, regardless of their depth (i.e. new posts or answers).
- The number of distinct authors (`COM_DEV_AUTH` , `COM_USR_AUTH`) is the number of people having posted at least once on the mailing list during the considered period of time. Authors are counted once even if they posted multiple times, based on their email address.
- The number of threads (`COM_DEV_SUBJ` , `COM_USR_SUBJ`) is the number of diffent subjects (i.e. a question and its responses) that have been posted on the mailing list during the considered period of time. Subjects that are replies to other subjects are not counted, even if the subject text is different.
- The number of answers (`COM_DEV_RESP_VOL` , `COM_USR_RESP_VOL`) is the total number of replies to requests on the user mailing list during the considered period of time. A message is considered as an answer if it is using the Reply-to header field. The number of answers is often associated to the number of threads to compute the useful response ratio metric.
- The median time to first reply (`COM_DEV_RESP_TIME_MED` , `COM_USR_RESP_TIME_MED`) is the number of seconds between a question (first post of a thread) and the first response (second post of a thread) on the mailing list during the considered period of time.

Rule checking

Rules are associated to practices that have an impact on some characteristic of quality. We list hereafter the rule checks that we decided to include in our data sets, with information about their associated practices and further references. The categories defined in the rule sets are mapped to the ISO 9126 decomposition of quality (analysability, changeability, stability, testability, etc.) and to development concerns (programming techniques, architecture, etc.).

Many of these rules can be linked to coding conventions published by standardisation organisms like MISRA (the Motor Industry Software Reliability Association, which C rule set is well spread) or CERT (Carnegie Mellon's secure coding instance). They usually give a ranking on the remediation cost and the severity of the rule. There are also language-specific coding conventions, as is the case with Sun's coding conventions for the Java programming language [171].

In the proposed data sets, there are 39 rules from Checkstyle, 58 rules from PMD, and 21 rules from SQuORE. We define below only a subset of them, corresponding to the most common and prejudicial errors. In the data sets, conformity to rules is displayed as a number of violations of the rule for the given artefact.

SQuORE

SQuORE rules apply to C, C++ and Java code. The following families of rules are defined: fault tolerance (2 rules), analysability (7 rules), maturity (1 rule), stability (10 rules), changeability (12 rules) and testability (13 rules). The most common checks are:

- No fall through (`R_NOFALLTHROUGH`). There shall be no fall through the next case in a switch statement. It threatens analysability of code (one may not understand where execution goes through) and stability (one needs to modify existing code to add a feature). It is related to rules Cert MSC17-C, Cert MSC18-CPP, Cert MSC20-C and Misra-C (2004) 15-2.
- Compound if (`R_COMPOUNDIFELSE`). An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement. It is related to Misra-C (2004) rule 14.9.
- Final else (`R_ELSEFINAL`). All if … else if constructs shall be terminated with an else clause. This impacts changeability, because developers can identify quickly what is the default treatment, and fault tolerance because unseen cases are caught. It is related to Misra-C (2004) rule 14.10.

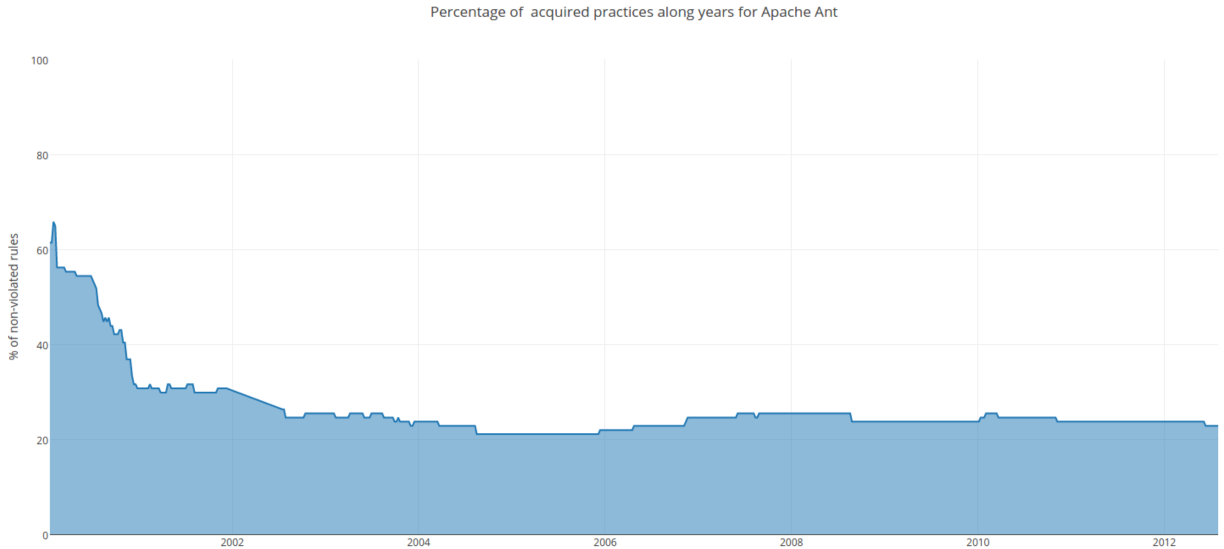


Figure 4: Percentage of acquired practices along years for apache ant

- No multiple breaks (`R_SGLBRK`). For any iteration statement there shall be at most one break statement used for loop termination. It is related to Misra-C (2004) rule 14.6 and impacts the analysability (developers will have trouble understanding the control flow) and testability (more paths to test) of code.
- No goto (`R_NOGOTO`). Gotos are considered bad practice (Misra-C (2004) rule 14.4) and may be hazardous (see CERT MSC35-CPP): they threaten the analysability of code, because one needs to scroll through the file instead of following a clear sequence of steps, and makes the test cases harder to write. Similarly, the no backward goto (`R_BWGOTO`) rule searches for goto operators that link to code that lies before the goto. Backward gotos shall not be used. They shall be rewritten using a loop instead.
- No assignment in Boolean (`R_NOASGINBOOL`). Assignment operators shall not be used in expressions that yield a boolean value. It is related to Cert EXP45-C, Cert EXP19-CPP and Misra-C (2004) rule 13.1.
- No assignment in expressions without comparison (`R_NOASGCOND`). Assignment operators shall not be used in expressions that do not contain comparison operators.
- Case in switch (`R_ONECASE`). Every switch statement shall have at least one case clause. It is releated to Misra-C (2004) rule 15.5.
- Label out of a switch (`R_NOLABEL`). A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement. It is related to Misra-C (2004) rule 15.1.
- Missing Default (`R_DEFAULT`). The final clause of a switch statement shall be the default clause. It is related to Cert MSC01-C, Cert MSC01-CPP and Misra-C (2004) rule 15.3.
- Code before first case (`R_NOCODEBEFORECASE`). There shall be no code before the first case of a switch statement. It is related to Cert DCL41-C.

Checkstyle

We identified 39 elements from the Checkstyle 5.6 rule set, corresponding to useful practices generally well adopted by the community. The quality attributes impacted by these rules are: analysability (23 rules), reusability (11 rules), reliability (5 rules), efficiency (5 rules), testability (3 rules), robustness (2 rules) and portability (1 rule). All rules are described on the project page at http://checkstyle.sourceforge.net/config.html.

- Javadoc checks (`JAVADOCMETHOD` , `JAVADOCPACKAGE` , `JAVADOCTYPE` , `JAVADOCVARIABLE`) ensure that Javadoc comments are present at the different levels of information as it is recommended by Sun [171].

- Import checks: UNUSEDIMPORTS looks for declared imports that are not used in the file. They clutter space and are misleading for readers. AVOIDSTARIMPORT checks that no generic import is used; specific import statements shall be used to help the reader grasp what is inherited in the file. REDUNDANTIMPORT looks for duplicates in imports, which uselessly takes up visual space.
- Equals Hash Code (EQUALSHASHCODE). A class that overrides equals() shall also override hashCode(). A caller may use both methods without knowing that one of them has not been modified to fit the behaviour intended when modifying the other one (consistency in behaviour). It impacts reusability, reliability and fault tolerance.
- No Hard Coded Constant (MAGICNUMBER). Hard coded constant or magic numbers shall not be used. Magic numbers are actual numbers like 27 that appear in the code that require the reader to figure out what 27 is being used for. One should consider using named constants for any number other than 0 and 1. Using meaningful names for constants instead of using magic numbers in the code makes the code self-documenting, reducing the need for trailing comments. This rule is related to programming technics and changeability.
- Illegal Throws (ILLEGALTHROWS) lists exceptions that are illegal or too generic; throwing java.lang.Error or java.lang.RuntimeException is considered to be almost never acceptable. In these cases a new exception type shall be defined to reflect the distinctive features of the throw.
- New line at end of file (NEWLINEATENDOFFILE). Checks that there is a trailing newline at the end of each file. This is an ages-old convention, but many tools still complain when they find no trailing newline. Examples include diff or cat commands, and some SCM systems like CVS will print a warning when they encounter a file that does not end with a newline.
- Anonymous Inner Length (ANONINNERLENGTH). Checks for long anonymous inner classes. For analysability reasons these should be defined as self-standing classes if they embed too much logic.
- Multiple String Literals (MULTIPLESTRINGLITERALS) checks for multiple occurrences of the same string literal within a single file. It should be defined as a constant, both for reusability and changeability, so people can change the string at first shot without forgetting occurrences.

PMD

We selected 58 rules from the PMD 5.0.5 rule set. These are related to the following quality attributes: analysability (26 rules), maturity (31 rules), testability (13 rules), changeability (5 rules), and efficiency (5 rules). The full rule set is documented on the PMD web site: http://pmd.sourceforge.net/pmd-5.0.5/rules/.

- Jumbled incrementer (JUMBLEDINCREMENT) detects when a variable used in a structure is modified in a nested structure. One shall avoid jumbled loop incrementers — it is usually a mistake, and even when it is intended it is confusing for the reader.
- Return from finally block (RETURNFROMFINALLYBLOCK). One shall avoid returning from a finally block since this can discard exceptions. This rule has an effect on analysability (developers will have trouble understanding where the exception comes from) and fault tolerance (the return method in the finally block may be stopping the exception that happened in the try block from propagating up even though it is not caught). It is related to the Cert ERR04-J rule.
- Unconditional if statements (UNCONDITIONALIFSTATEMENT), empty if statements (EMPTYIFSTMT), empty switch statements (EMPTYSWITCHSTATEMENTS), empty synchronized block (EMPTYSYNCHRONIZEDBLOCK), and empty while statements (EMPTYWHILESTMT) are useless and clutter code. They impact analysability – developers will spend more time trying to understand what they are for, and they may have undesirable side effects. As for the empty while statements, if it is a timing loop then Thread.sleep() is better suited; if it does a lot in the exit expression then it should be rewritten to make it clearer. All these are related to Cert MSC12-C.
- Empty catch blocks (EMPTYCATCHBLOCK) are instances where an exception is caught, but nothing is done. In most circumstances an exception should either be acted on or reported, as examplified in the Cert ERR00-J rule. Empty try blocks (EMPTYTRYBLOCK) and empty finally blocks (EMPTYFINALBLOCK) serve no purpose and should be remove because they clutter the file's analysability.
- The God Class (GODCLASS) rule detects the God Class design flaw using metrics. God classes do too many things, are very big and overly complex. They should be split apart to be more object-oriented.

The rule uses the detection strategy described in Object-Oriented Metrics in Practice [118].

- Avoid Catching Throwable (`AVOIDCATCHINGTHROWABLE`). CatchingThrowable errors is not recommended since its scope is very broad. It includes runtime issues such as OutOfMemoryError that should be exposed and managed separately. It is related to the ERR07-J rule.
- Avoid Catching NPE (`AVOIDCATCHINGNPE`) Code should never throw NullPointerExceptions under normal circumstances. A catch block for such an exception may hide the original error, causing other, more subtle problems later on. It is related to the Cert ERR08-J rule.
- Non Thread Safe Singleton (`NONTHREADSAGESINGLETON`) Non-thread safe singletons can result in bad, unpredictable state changes. Static singletons are usually not needed as only a single instance exists anyway: they can be eliminated by instanciating the object directly. Other possible fixes are to synchronize the entire method or to use an initialize-on-demand holder class. See Effective Java [25], item 48 and Cert MSC07-J.

Bibliography

Panagiotis Louridas. Static Code Analysis. IEEE Software 23(4):58–61, 2006.

```
@article{Louridas2006,
    author = "Louridas, Panagiotis",
    journal = "IEEE Software",
    number = 4,
    pages = "58--61",
    title = "{Static Code Analysis}",
    volume = 23,
    year = 2006
}
```

Nick Rutar, Christian B Almazan and Jeffrey S Foster. A comparison of bug finding tools for Java. In 15th International Symposium on Software Reliability Engineering. 2004, 245–256. URL

```
@inproceedings{Rutar2004,
    author = "Rutar, Nick and Almazan, Christian B and Foster, Jeffrey S",
    booktitle = "15th International Symposium on Software Reliability Engineering",
    pages = "245--256",
    title = "{A comparison of bug finding tools for Java}",
    url = "http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1383122",
    year = 2004
}
```


Using the dataset

You can import them easily into LibreOffice by simply double-clicking on the file and selecting '!' (exclamation mark) as the separator. Don't use anything else as separator, since fields may contain other characters, e.g. commas. The following image shows a plot made with LibreOffice:

```r
app  <- read.csv('metrics_ant_evo_app_v3.1.csv', sep='!')
names(app)
```

```
  [1] "Application"                      "Version"
  [3] "BLAN"                             "BRAC"
  [5] "CFT"                              "CLAS"
  [7] "CLOC"                             "COM_DEV_AUTH_1M"
  [9] "COM_DEV_AUTH_3M"                  "COM_DEV_AUTH_1W"
 [11] "COM_DEV_RESP_TIME_MED_1M"         "COM_DEV_RESP_TIME_MED_3M"
```

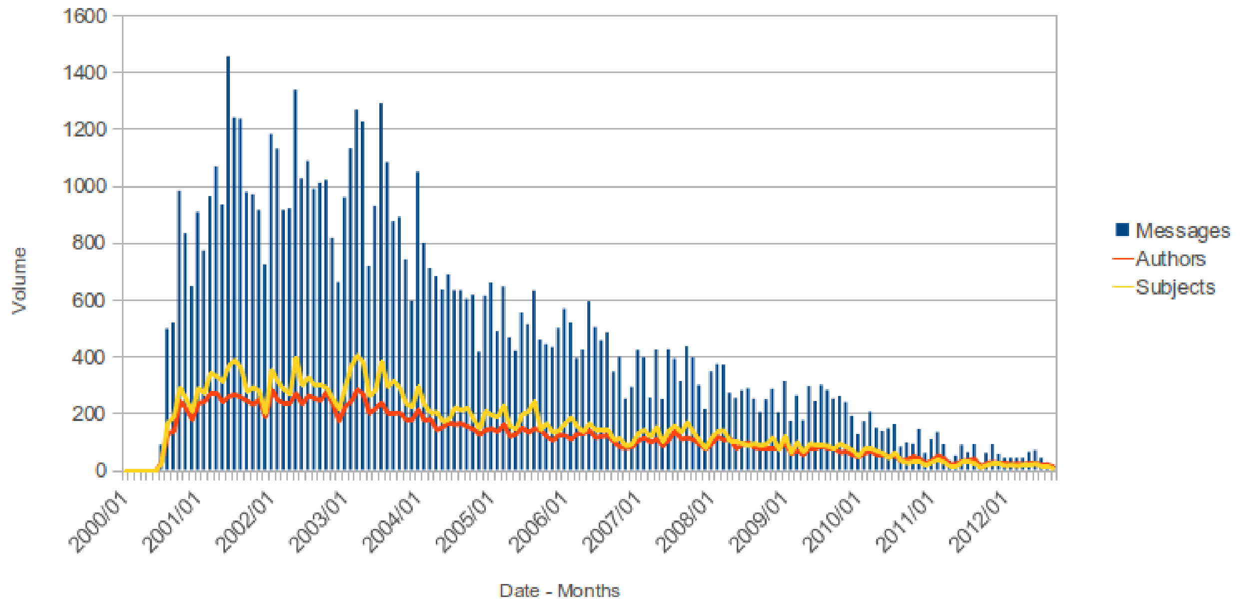
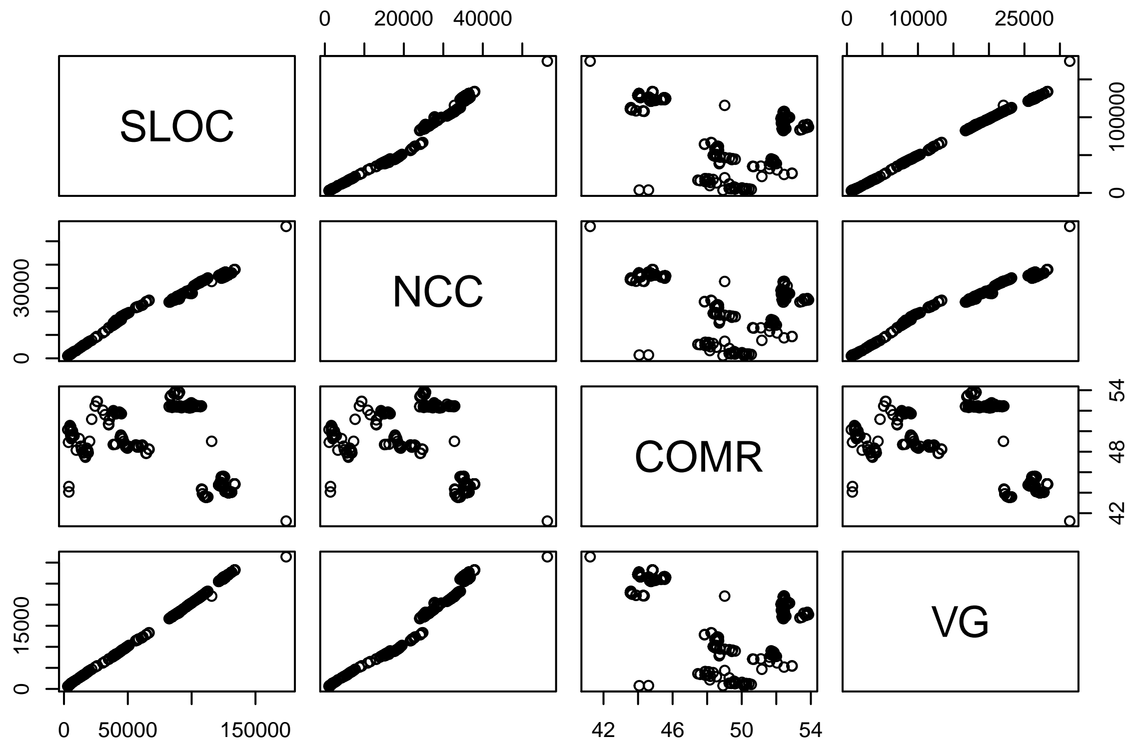


Figure 5: Mails Ant user mailing list

```
[13] "COM_DEV_RESP_TIME_MED_1W"          "COM_DEV_RESP_VOL_1M"
[15] "COM_DEV_RESP_VOL_3M"               "COM_DEV_RESP_VOL_1W"
[17] "COM_DEV_VOL_1M"                    "COM_DEV_VOL_3M"
[19] "COM_DEV_VOL_1W"                    "COM_DEV_SUBJ_1M"
[21] "COM_DEV_SUBJ_3M"                   "COM_DEV_SUBJ_1W"
[23] "COM_USR_AUTH_1M"                   "COM_USR_AUTH_3M"
[25] "COM_USR_AUTH_1W"                   "COM_USR_RESP_TIME_MED_1M"
[27] "COM_USR_RESP_TIME_MED_3M"          "COM_USR_RESP_TIME_MED_1W"
[29] "COM_USR_RESP_VOL_1M"               "COM_USR_RESP_VOL_3M"
[31] "COM_USR_RESP_VOL_1W"               "COM_USR_SUBJ_1M"
[33] "COM_USR_SUBJ_3M"                   "COM_USR_SUBJ_1W"
[35] "COM_USR_VOL_1M"                    "COM_USR_VOL_3M"
[37] "COM_USR_VOL_1W"                    "COMR"
[39] "DITM"                              "ELOC"
[41] "FILE"                              "FUNC"
[43] "LADD"                              "LC"
[45] "LMOD"                              "LREM"
[47] "MLOC"                              "NCC"
[SNIP...]
```

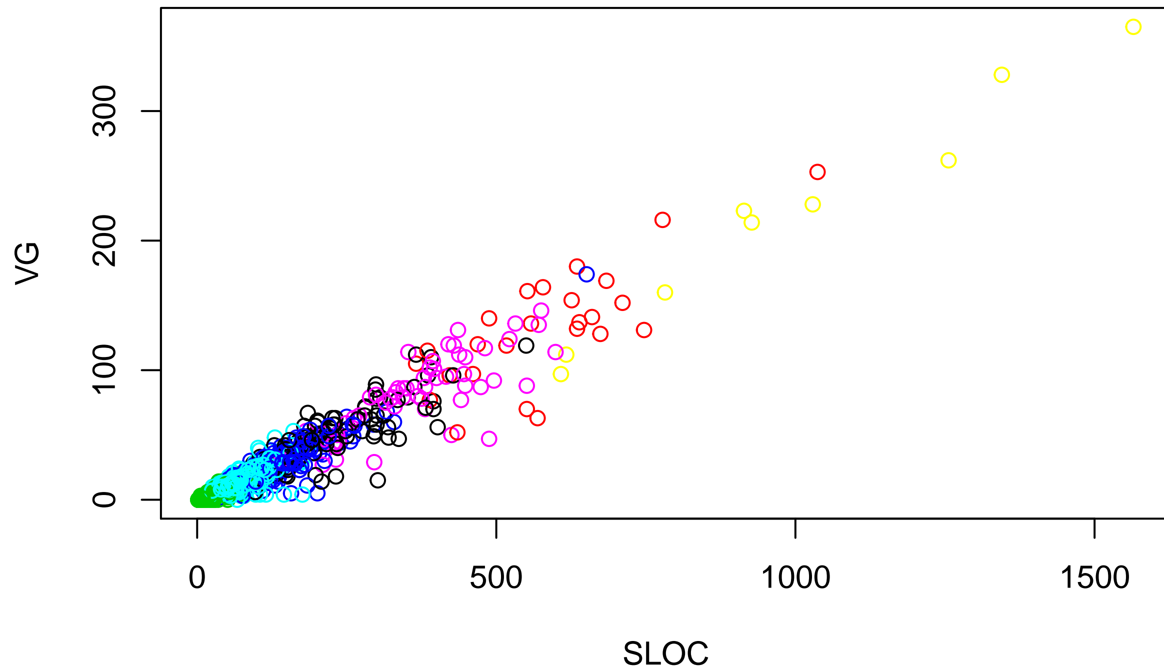
From there you have all data at hand. You can easily plot it:

```
plot(app[,c("SLOC", "NCC", "COMR", "VG")])
```



You can apply your own data mining techniques to the set as well. As an example in the following extract we want to find categories of files depending on their non-conformities. For that purpose we apply k-means clustering to the `NCC` metric at the file level and display the results, showing different colours for the output clusters with the size (`SLOC`) and complexity (`VG`) as axes.

```
files_kmeans_sloc_7 <- kmeans(files[files$Version == '2008-07-21',c('NCC')], 7)
groups_sloc_7 <- table(files_kmeans_sloc_7$cluster)
plot(files[files$Version == '2008-07-21',c('SLOC', 'VG')],
  col=files_kmeans_sloc_7$cluster)
```



It's also quite easy to build time series objects:

```
app.xts <- xts(x=app, order.by = parse_iso_8601(app$Version))
```