Assessing and improving the quality of Fortran code in scientific software: FortranAnalyser

Michael García-Rodríguez (micgarcia@uvigo.es)
Universidade de Vigo & CIM-UVigo
Juan A. Añel
Universidade de Vigo & CIM-UVigo
Javier Rodeiro-Iglesias
Universidade de Vigo

Article

Keywords:

Posted Date: March 30th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1442971/v1

Abstract

Scientific software often presents shortcomings when performing a static evaluation of the code. Moreover, a significant amount of this software is written in Fortran. However, only a few software tools analyse Fortran code, and they are not free software. Here we introduce FortranAnalyser, a multi-platform static analysis tool that generates a report helping to improve Fortran code quality. We explain its development cycle and main features and compare it to other existing tools. Also, we provide examples of application to codes from different scientific fields, with a particular case study of the process followed to improve one of the most used global climate models.

Introduction

Code quality has been an issue of concern since the early days of software. Coding in a language that is easy to interpret by computers and humans alike has been an end in itself. Already more than four decades ago, several approaches were proposed to evaluate software quality. However, some of the ideas proposed at the time can no longer be applied because computers no longer operate in the way they used to. Over the last years, the quality of the software developed and used in science and engineering has been put in the focus, mostly because of its impact on repeatability, reproducibility and replicability[1] of the results produced with it. The lack of formal training of many scientists in programming is often cited as one of the sources for some of the problems of the software developed in the mentioned fields[2]. However, although there is a large room for improvement, scientific software development has improved, incorporating better practices already applied in other areas of programming[3,4]. Many authors have proposed practices that can help to continue improving the quality of scientific software and the experience of its development and use[5–7]. However, this is a complicated issue. Over the years and according to Moore's law[8], the evolution of code has implied the need for increasing amounts of computing power. This need is due to increasing software heterogeneity, complexity and technological advancements. Some of the solutions and practices proposed for better code quality and the development of scientific software are better suited for increasingly popular programming languages. However, for many large scale projects, with a significant part of legacy code, they are hard to apply.

Fortran was created as a cost-effective solution to develop complex codes, which could be applied efficiently by human beings[9]. Fortran is still a dominant language for large-scale simulations of physical systems, e.g., coupled climate modelling[10], high-performance computing[11], or benchmarking[12]. It is the 19^{th} most used programming language (as of January 2022)[13] and continues to be essential[14]. Although some discussion has been ongoing since 1984 on whether it is necessary to retire Fortran[15], its superior speed is the main reason for its continued use[11].

Static code analysis is a way of analysing the code of a program without running it, checking its quality. There are many software tools for performing static code analysis for different languages. Nonetheless, it has been pointed out that the existing tools are hardly of application to evaluate the scientific software

developed in Fortran[16]. Also, to our best knowledge, it does not exists a tool to perform static analysis of Fortran code to quite the same level than for other languages. The most relevant options are listed in Table 1 .

However, these analysers have many weak points. Some of them are the following:

- Coverity : Proprietary tool commercialised by Synopsis, Inc. Coverity can analyse Fortran code after integration of the previously existing tool "Forcheck", bought in 2017 by Synopsys, Inc. It does not assess the use of good programming practices.
- Intel® parallel studio XE : It does not assess the use of good programming practices;
- K-scope : Only supports Fortran 77 and 90;
- Ftncheck : Only supports Fortran 77.

From the characteristics of the tools described above, they lack the desirable sophistication and openness level. Therefore, facing the need of evaluating the quality of the code of some of the most used scientific applications, it is necessary to have a more up-to-date and free tool to check Fortran code. Many of the existing tools can carry out lexical analysis, but only a few are valid for semantic analysis. This type of analysis is the main objective of FortranAnalyser (“http://fortrananalyser.ephyslab.uvigo.es/”) (see "" section), a software developed with the aim of finding common programming flaws in the Fortran code for any standard. Therefore, in this paper we present the evaluation of some Fortran programmes commonly used in the fields of science and engineering, as case-studies to test FortranAnalyser.

Next, we explain what FortranAnalyser does and how it does it. We expose the methodology used in its development and its functional requirements. The integrated development environment (IDE) is described, together with the dependencies, version, and repository managers used. User interaction with the system and the project structure are explained in "" section. Finally, we calculate individual scores for each analysed routine and software and provide a total score. We justify the metrics applied and prove its usefulness with a real case: improving one of the most used global climate models through its different versions.

Software Description

FortranAnalyser (currently in its version 2.0) is developed in Java[20] – “IcedTea project”[21] (free software under GPLv3 license[22]), with multi-language and multi-platform support. It can currently be used in English (default language), Spanish, Galician, Portuguese or French for both the user interface and quality report, and can be expanded to support any natural language. It is licensed under GPLv3 to ensure the computational reproducibility[23,24] .

FortranAnalyser generates a quality report in PDF complying with the ISO 9001[25] quality standard of documentation. The report is saved inside a new directory called “*tmp*” in the root directory with the name “*QualityReport.pdf* ”.

Development

To develop FortranAnalyser, we used an iterative and incremental methodology. Initially, we defined several milestones (see Fig. 1a) according to the functionalities that we considered for FortranAnalyser (see Fig. 1b). Each of the milestones constitutes an iteration, and the final product is then achieved incrementally.

The functionalities are detailed with their corresponding uses, together with the class diagram of the system (Fig. 2). A sequence diagram to better understand the interaction between the actor and the system is exposed in Fig. 3).

The functional requirements of the application are:

- exit: allows to close the application successfully;
- language: allows to select the language of the user interface and the quality report;
- select directory: allows to select the directory of the Fortran project;
- analyse: FortranAnalyser generates the quality report according to the quality metrics defined in "" section.

The IDE selected was Netbeans v8.2[26] because of the large variety of add-ons available and its directly integrated Graphical User Interface (GUI) builder. It also contains a directly integrated mean of managing third-party libraries: Maven[27]. We use GitLab[28,29] as the version manager. Figure 4 shows the flowchart for this project.

Quality Metrics

This section explains all the metrics used by FortranAnalyser to determine the quality of Fortran code. They are designed and implemented following well-established criteria on software development[30–32].

List of metrics

1. the “IMPLICIT NONE” statement ([30], pp. 57–66): Including ”IMPLICIT NONE” at the beginning of a file ensures that the compiler will generate the appropriate warnings when a variable lacks an explicit declaration. In Fortran, it is advisable to declare all the variables correctly to avoid unnecessary memory allocation (e.g. real variables need double memory than integer variables);

2. omments [30]:

- at the beginning of the document: it is verified that two out of the first three lines of each file are comments. These are supposed to contain a brief description of what does the code;
- in the declaration of a variable, function and subroutines: each one of these must have associated a comment before or after. The comment must describe its type, use and what it does;

- in control structures: each control structure must have a comment before or after. It must describe what condition is being checked to undertake a specific operation.

3. the “EXIT” statement ([30], pp. 145–147): releases resources allocated, allowing their use in other task;

4. number of nested loops ([30], pp. 148–150): nested loops increase the resources needed for a task. For this reason, it is established as a quality metric that there should be no more than three nested loops and no commentary at the opening of each loop;

5. ratio: refers to the ratio of the number of lines with comments to the number of computable elements. It checks whether all these computable elements (e.g. loops, "if" statements, and functions) include comments before or after them. It assumes that comments are in English, as it is the universal language of science[33]. However, it should be bear in mind that comments in other languages do not occupy the same number of characters. That said, FortranAnalyser can evaluate the ratio from comments in any language. However, the calculation is more accurate if they are in English;

6. the “CYCLE” statement ([30], pp. 145–147): avoids unnecessary code execution, letting to iterate to the next element.

Justification of scores

In software programming, there are elements with a significant influence on the execution and structuring of the code, and others are not essential. For example, the "IMPLICIT NONE" statement is critical, as it helps save memory and code debugging and maintenance. Nested loops make it difficult to detect the exact point at which a run-time problem occur, especially when the code is very long. Less critical elements are the "EXIT" and "CYCLE" statements, as for the structuring of the code, they do not help avoid run-time errors. However, their use is highly advisable as, in some cases, they can affect performance.

Based on this, we define two types of metrics (see Table 2):

- type A: metrics that have considerable influence on the structuring of the code. They are assigned a score of 2.0 points and include:

1.the use of the “IMPLICIT NONE”;

2.number of nested loops;

3. the use of comments;

4. ratio.

- type B: metrics that have less influence: "EXIT" and "CYCLE" statements. They are assigned a score of 1.0 point.

The sub-metrics on the use of comments sum 2.0 points, but this score is divided by the total number of items. For this reason, each of them is assigned a value of 0.4 points.

A Fortran project is composed of files, each with a given number of lines, the sum of which is the total number of lines in the software. Following the above mentioned metrics we rate each file or routine in a Fortran project. The total score of the full project will be the summation of the score for each file multiplied by the number of lines in the file and divided by the total number of lines of the project. Mathematically ("n": total number of files):

$$score = \sum_{i=0}^{n} \frac{Score\ file_i \cdot Lines\ file_i}{Total\ lines}$$

A total score is obtained by considering the weighted value of each file analysed in the project as a whole. The maximum score obtained by any project can be 10.

Quality report

From the information described in the preceding subsections, FortranAnalyser generates a quality report (see supplementary material for an example). Part of this information is used to compute the quality metrics[32]. Other is provided for illustrative purposes as it can help the developer to improve the code. The report includes information about:

1. number of lines in the file : it determines the size of the programme analysed. Obviously, there is no direct relationship between the number of lines and the complexity of the code. However, this variable is used in the calculation of other metrics, such as the number of lines with comments;

2. number of variables declared : it is important to economise on the variables declared, otherwise, they may waste memory;

3. number of calls to subroutines : every call increases the interdependence between modules and therefore, the coupling. An ideal and well-structured code is expected to show low coupling, which allows maintaining a high level of cohesion[34];
4. number of subroutines declared : its use helps to design a project with a smaller number of lines, improving its mainte- nance and readability[34];
5.number of functions declared : the use of functions avoids duplication when a task is performed repeatedly.
Once the analysis has been performed, the previous information is printed in the report. It contains a table explaining the ten metrics applied, together with the score obtained for each one of them, and last, the total score. For each file analysed, the score obtained is shown individually. An example is shown in Fig. 5.

Assessment Of Software Through Different Disciplines

In order to test FortranAnalyser in a range of different settings, we analysed a small selection of codes used in multiple scientific fields. Specifically, we used codes belonging to climate models, linear algebra, and computational chemistry. Table 3 shows all the codes analysed together with their sizes and number of Fortran files in each, including configuration files.

Climate models are a complex type of software and make extensive use of Fortran. To check FortranAnalyser, we used models from the 5th phase of the Coupled Model Intercomparison Project[10]. The most recently completed phase of CMIP was phase five[57], which includes a total of twenty-six models[58]. We obtained the code for eleven climate models belonging to this phase[59] and analysed them using FortranAnalyser. Each model received a score ranging from 0 to 10 points, as shown in Table 4.

We also randomly selected six libraries from those used in our local supercomputing center, the Supercomputing Center of Galicia (CESGA), in the field of linear algebra. Table 5 shows the score obtained for each library.

Finally, we analysed some quantum chemistry programmes. The quantum chemistry and solid-state physics software list[60] includes the most versatile software packages used in quantum chemistry. From this list, we randomly selected five applications to analyse using FortranAnalyser, and the results were delivered to the quality report, as shown in Table 6 .

Tables4,5and6 summarise the scores obtained by the different codes. They show that no software received the maximum possible score of 10.

Regarding the climate models in **Table4** , the average score was 3.6. The “GISS” obtained the worst rating at 2.8. “NICAM” got the best score of all climate models, and all the software analysed in these cases: 4.3.

For the linear algebra libraries in **Table5** , the worst score was obtained by “ARPACK” (0.710). That score was the lowest of all the software analysed in these study cases. The report generated by FortranAnalyser indicates the main reason for this, which was the absence of comments and the *implicit none* statement. In contrast, “expokit” received the best score: 2.3.

Finally, for the quantum chemistry computer programmes in **Table6** , “EXCITING” received the worst score with 1.8 points. “SIESTA” got the best score among all the quantum chemistry computer software analysed, with 4.0 points.

Examples Of Application For Improvement Of Code

BASIC ARTIFICIAL CASE

To show how the static analysis can help to improve existing Fortran codes, we provide an example of an imperfect code (see Figure6) together with the same code after improvement. First, we analyse the quality report generated by FortranAnalyser for software with a code that scores poorly in all the metrics cited in the “Quality Metrics”. Then we compare it to the quality report obtained after modifying the code to address the shortcomings pointed out by the analysis.

Running FortranAnalyser using this code yields a quality report with a score of 2.8 points because there are no subroutines (0.4 points) or function declarations (0.4 points), and the complexity of nested loops is accomplished (2.0 points).

On the other hand, if a maintainer or developer decided to improve the code, after checking the report produced by FortranAnalyser (see “QualityReportCylinderV1.pdf” in supplementary material), it would be possible to build a better version, e.g., “Cylinder_v2.f90” (see Fig. 7). The recommendations for improvement that apply in this case according to the quality report are:

- add comments at the beginning of the document;
- add comments on control structures;
- add comments on if cases;
- add comments on loops;
- add comments on variable declarations;
- use the “CYCLE” sentence on loops;
- use the “EXIT” sentence on loops;
- use of the sentence “implicit none”;
- check the ratio (calculation of the number of lines with comments to the number of computable elements);

After analysing the improved version of the code, the quality report generated shows a score of 10.0 points. The output file shows all metrics with higher scores, and the updated report can be seen in “QualityReportCylinderV2.pdf” in the supplementary material.

REAL CASE: IMPROVEMENT OF THE IPSL MODEL

The "Institut Pierre Simon Laplace" (IPSL) is one of many research centres contributing to the CMIP. The IPSL is responsible for developing climate models with the aim of expanding our understanding of climate, by developing the climate model that bears the same name: IPSL[37]. Specifically, FortranAnalyser was used to develop the IPSLCM6 (the CMIP6[61] version), released in April 2018 after an Agile cycle of development including more than 20 releases, starting from IPSLCM5 (the CMIP5 version) released in 2010. It is possible to obtain the code for both versions of the model by contacting the institute. It should be noted that not all the model is implemented in Fortran. In Table7, it is possible to check the number of Fortran files that constitute the model. The development cycle between versions has taken seven years.

During this time, the developers produced a new version of their model and improved each component in terms of physics and parallelism. Additions have been made respecting the coding rules in line with the FortranAnalyser criteria. Among them, systematic quality control procedures were implemented.

The corresponding quality reports are included here as supplementary material. Table 7shows a summary of all results of the reports. Three of the sub-models that make up IPSLCM6 (“XIOS", “LMDZ" and “OASIS") have been substantially improved as reflected by their scores, increased by 1.39 points. In some cases, the number of lines of code has tripled. “IOIPSL" obtains the same score. “IOSERVER" was not included in the CMIP6 version of the model. Finally, “NEMO" and “ORCHIDEE" received a lower score than the previous version because its development cycle was closed before the release of FortranAnalyser. In this way, FortranAnalyser was not applied to their CMIP6 update. However, they only received a score of 0.09 points lower in the case of “ORCHIDEE" and 0.17 points in the case of “NEMO".

Conclusion

Simple static code analysis of some commonly used Fortran high-performance computing applications shows a large room for improvement in coding in all cases, both in terms of both programming and computational performance. Lack of comments to the code seems to be quite usual. Also, some best practices of the Fortran language are missed. Our case study shows that the climate models obtained the highest scores (reflecting higher quality) and that the climate models themselves can be easily improved with static code analysers. Static code analysis has the potential to strengthen the quality of the code of these programmes.

In this study, we have performed our analysis using FortranAnalyser. We explain its development and application. The design of the tool and examples of its application to software from different branches of science have been included. FortranAnalyser has had eight releases and an encouraging number of downloads worldwide, considering the small community working with Fortran. Over time, some bugs detected by the community have been fixed, and new features were added. Despite being in its early stages of development, FortranAnalyser is fully functional. It yields organised, structured and visually appealing information and produces analysis information in a high-quality format: PDF. Moreover, FortranAnalyser has a user-friendly interface. It is multilanguage (English, Spanish, Galician and French), multi-platform, and standalone. It does not depend on an IDE or any other previously installed programme and can be used in computers, smartphones or tablets. Finally, another of the significant advantages of FortranAnalyser is its compatibility with all existing versions of Fortran. Table 8shows a comparison between the analysers listed previously in Table 1 and FortranAnalyser with the following information:

- Fortran version: the version of Fortran that the software can analyse;
- multi-platform: the software works on multiple desktop computing platforms;
- languages: number of languages in which the application is available to the user;
- plugin: the software works as a plugin of another programme;
- compilation request: refers to the need to compile the code before using such software;

- licence.

The metrics included in FortranAnalyser, widely accepted in software engineering, are not as numerous as in other tools, although we expect to increase them in future versions. It nevertheless has the advantage of being compatible with all versions of Fortran. The performance of FortranAnalyser only depends on two factors: the technical specifications of the computer on which FortranAnalyser is run, and the size of the programmes analysed. FortranAnalyser thus needs more analysis time for a large piece of software than a small one.

FortranAnalyser has been already awarded prizes, such as the second prize in the "Free Software Awards of Galicia 2017" and the Water Campus Prize to the best Master Thesis. Also, it has sparked interest from the public and private companies that have begun to use it. Potential future developments include supporting additional languages for the user interface and reports and running FortranAnalyser as a cloud service[62]. Also, we expect to add new metrics, such as a nesting metric.

Declarations

Acknowledgements

This work was supported in part by:

- Grant 10DPI305002PR - Xunta de Galicia;
- Grants CGL2015-71575-P and RYC-2013-14560 - Government of Spain;
    - We are grateful to Marie-Alice Foujols from the Institut Pierre-Simon Laplace for their support with the work on the IPSL model and comments to draft versions of this paper.

Data Availability

The datasets generated and/or analysed during the current study are available in the ZENODO repository, 10.5281/zen- odo.6374055

References

1. ACM. Artifact Review and Badging – Version 2.0 - https://www.acm.org/publications/policies/artifact-review-badging (accessed 11.03.2022) (2016).
2. Baxter, S. M., Day, S. W., Fetrow, J. S. & Reisinger, S. J. Scientific software development is not an oxymoron. PLOS Comput. Biol. **2**, 1–4, DOI:10.1371/journal.pcbi.0020087(2006).
3. Nguyen-Hoan, L., Flint, S. & Sankaranarayana, R. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and*

Measurement, ESEM '10, DOI: 10.1145/1852786.1852802(Association for Computing Machinery, New York, NY, USA, 2010).

4. Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A. & Carver, J. C. Software engineering practices for scientific software development: A systematic mapping study. J. Syst. Softw. **172**, 110848, DOI:https://doi.org/10.1016/j.jss.2020. 110848(2021).
5. Wilson, G. *et al.* Best practices for scientific computing. PLOS Biol. **12**, 1–7, DOI:10.1371/journal.pbio.1001745(2014).
6. Riesch, M., Nguyen, T. D. & Jirauschek, C. bertha: Project skeleton for scientific software. PLOS ONE **15**, 1–12, DOI: 10.1371/journal.pone.0230557(2020).
7. Hunter-Zinck, H., de Siqueira, A. F., Vásquez, V. N., Barnes, R. & Martinez, C. C. Ten simple rules on writing clean and reliable open-source scientific software. PLOS Comput. Biol. **17**, 1–9, DOI:10.1371/journal.pcbi.1009481(2021).
8. Moore, G. E. *Cramming more components onto integrated circuits* (McGraw-Hill, 1965), 38 edn.
9. Backus, J. The history of FORTRAN I, II and III, Asociation of Computing Machinery. ACM Sigplan Not. **13**, 25–74, DOI:10.1145/800025.1198345(1998).
10. CMIP. Coupled Model Intercomparision Project. http://cmip-pcmdi.llnl.gov/ (accessed 11.03.2022) (2008).
11. Loh, E. The ideal hpc programming language. *ACM Queue* **8**, 30:30–30:38, DOI:10.1145/1810226.1820518(2010).
12. Petitet, A. and Whaley, R. C. and Dongarra, J. and Cleary, A. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/ (accessed 11.03.2022) (2008).
13. TIOBE Index. https://www.tiobe.com/tiobe-index/ (accessed 11.03.2022) (2019).
14. Cotton, B. Happy 60th birthday, Fortran, https://opensource.com/article/17/11/happy-60th-birthday-fortran (accessed 11.03.2022) (2017).
15. Cann, D. Supercomputing '91 Proceedings of the 1991 ACM/IEEE conference on Supercomputing. *ACM Sigplan Not.* –272, DOI:10.1145/125826.125976(1991).
16. Kanewala, U. & Bieman, J. M. Testing scientific software: A systematic literature review. Inf. Softw. Technol. **56**, 1219–1232, DOI:https://doi.org/10.1016/j.infsof.2014.05.006(2014).
17. Intel Corporation. Intel Parallel Studio XE, https://software.intel.com/sites/products/evaluation-guides/docs/studioxe- evalguide-SSA-with_Fortran_020812.pdf (accessed 11.03.2022) (2017).
18. Riken Center of Computational Science. K-Scope, https://www.r-ccs.riken.jp/ungi/soft/kscope/ (accessed 11.03.2022) (2012).
19. Sourceforge project. Ftncheck, https://www.dsm.fordham.edu/ ftnchek/ (accessed 11.03.2022) (2000).
20. Schildt, H. *Java 8* (Anaya Multimedia, Madrid, 2014), 1 edn.

21. Hughes, A. J. Iced-Tea Web, https://icedtea.classpath.org/wiki/Main_Page (accessed 11.03.2022) (2007).
22. GNU. GPLv3 License. https://www.gnu.org/licenses/gpl-3.0.en.html (accessed 11.03.2022) (2007).
23. Añel, J. A. The Importance of Reviewing the Code. COMMUN. ACM **54**, 40–41, DOI:10.1145/1941487.1941502(2011).
24. Añel, J. A. Comment on Most computational hydrology is not reproducible, so is it really science? by Christopher Hutton et al. Water Resour. Res. 2572–2574 (2017).
25. ISO. ISO/IEC 9001:2015 (2015).
26. Apache Software Foundation, Oracle Corporation and Sun Microsystems. Netbeans, https://netbeans.org/ (accessed 11.03.2022) (1997).
27. Apache. Maven, https://maven.apache.org/ (accessed 11.03.2022) (2013).
28. Torvalds, L. Git: free and open source distributed version control system, http://www.git-scm.com (accessed 11.03.2022) (2005).
29. Dmitriy. GitLab, https://about.gitlab.com/ (accessed 11.03.2022) (2011).
30. Chapman, S. J. *Fortran for Scientists and Engineers* (McGraw-Hill, 2 Penn Plaza, New York, NY 10121, 2018), 4 edn.
31. Laza Fidalgo, R. & García Pérez-Schofield, J. B. *Metodología y tecnología de la programación* (Pearson Educación, Madrid, 2008), 1 edn.
32. Piattini Velthuis, M. G. & García Rubio, F. O. *Calidad en el desarrollo y mantenimiento del software* (Ra-Ma, Madrid, 2003), 1 edn.
33. Drubin, D. G. & Kellogg, D. R. English as the universal language of science: opportunities and challenges. MOL. BIOL. CELL. **23**, DOI:10.1091/mbc.E12-02-0108(2012).
34. Sommerville, I. *Software engineering* (Pearson Educación, Madrid, 2005), 7 edn.
35. CESM. CESM, NSF-DOE-NCAR, http://www.cesm.ucar.edu/experiments/cmip5.html (accessed 11.03.2022) (2008).
36. National Center for Environmental Prediction. CCFS, Cola and NCEP, http://cfs.ncep.noaa.gov/cfsv2.info/ (accessed 11.03.2022) (2010).
37. Dufresne, J. L. et al. Climate change projections using the IPSL-CM5 Earth System Model: from 276 CMIP3 to CMIP5. *CLIM. DYNAM.* **40**, 2123:2165, DOI:10.1007/s00382-012-1636-1(2013).
38. NASA. GISS, https://www.giss.nasa.gov/about/ (accessed 11.03.2022) (2008).
39. National Center for Atmospheric Research. CCSM4, https://www.earthsystemgrid.org/project/ccsm.html (accessed 11.03.2022) (2014).
40. The research council of Norway. NorESM1, NCC, https://folk.uib.no/ngfhd/EarthClim/index.htm (accessed 11.03.2022) (2011).
41. NOAA. GFDL, https://www.gfdl.noaa.gov/ (accessed 11.03.2022) (1955).
42. GMAO. GEOS5, https://gmao.gsfc.nasa.gov/GEOS/ (accessed 11.03.2022) (2008).

43. Max Planck Institute. ECHAM6, http://www.mpimet.mpg.de/en/science/models/mpi-esm/echam/ (accessed 11.03.2022) (2015).
44. Max Planck Institute. MPI, http://www.mpimet.mpg.de/en/science/models/mpi-esm/ (accessed 11.03.2022) (2015).
45. Nonhydrostatic ICosahedral Atmospheric Model. NICAM, http://nicam.jp/hiki/?About+NICAM (accessed 11.03.2022) (2015).
46. CESGA. ARPACK, https://www.cesga.es/es/soporte_usuarios/usr-servicio-computacion/Aplicaciones?app=ARPACK (accessed 11.03.2022) (1990).
47. CESGA. Openblas, https://www.cesga.es/es/soporte_usuarios/usr-servicio-computacion/Aplicaciones?app=openblas (accessed 11.03.2022) (1990).
48. CESGA. Qrupdate, https://www.cesga.es/soporte_usuarios/usr-servicio-computacion/Aplicaciones?app=qrupdate (ac- cessed 11.03.2022) (1990).
49. CESGA. SuiteSparse, https://www.cesga.es/es/soporte_usuarios/usr-servicio-computacion/Aplicaciones?app=SuiteSparse (accessed 11.03.2022) (1990).
50. CESGA. Expokit, https://www.cesga.es/soporte_usuarios/usr-servicio-computacion/Aplicaciones?app=expokit (accessed 11.03.2022) (1990).
51. CESGA. Igakit, https://www.cesga.es/es/soporte_usuarios/usr-servicio-computacion/Aplicaciones?app=igakit (accessed 11.03.2022) (1990).
52. CP2K. https://www.cp2k.org/ (accessed 11.03.2022) (1989).
53. ELK. ELK, http://elk.sourceforge.net/ (accessed 11.03.2022) (2010).
54. Humboldt-Universität. EXCITING, http://exciting-code.org/ (accessed 11.03.2022) (2014).
55. icmab. SIESTA, https://departments.icmab.es/leem/siesta/ (accessed 11.03.2022) (2002).
56. EPW. Electron-phonon Wannier, http://epw.org.uk/Main/About (accessed 11.03.2022) (2009).
57. CMIP5. Coupled Model Intercomparison Project Phase 5 - Overview (2015), https://pcmdi.llnl.gov/mips/cmip5/ (accessed 11.03.2022) (2015).
58. CMIP5. CMIP pcmdi, Data Access - Availability (2015), https://pcmdi.llnl.gov/mips/cmip5/availability.html (accessed 11.03.2022) (2008).
59. Añel, J. A., García-Rodríguez, M. & Rodeiro, J. Current status on the need for improved accessibility to climate models code. Geosci. Model. Dev. **14**, 923–934, DOI:10.5194/gmd-14-923-2021(2021).
60. Computational chemistry software. Computational chemistry software - http://www.linux4chemistry.info/ (accessed 11.03.2022) (2013).
61. WCRP. World Climate Research Programme - https://www.wcrp-climate.org/wgcm-cmip/wgcm-cmip6 (accessed 11.03.2022) (2016).
62. Añel, J. A., Montes, D. P. & Rodeiro Iglesias, J. *Cloud and Serverless Computing for Scientists* (Springer, 2020).

Tables

Table 1
Fortran code analysis tools.

Name	compatible with
Coverity	all Fortran versions
Intel® parallel studio XE[17]	all Fortran versions
KScope[18]	Fortran 77 and Fortran 90
Ftncheck[19]	Fortran 77

Table 2
Summary of score for each metrics.

Metric		Score
The use of the IMPLICIT NONE		2.0
	at the beginning of the document	0.4
	in the declaration of a variable	0.4
Comments	in the declaration of a function	0.4
	in the declaration of a subroutine	0.4
	in control structures	0.4
Use of the EXIT statement		1.0
Number of nested loops		2.0
Ratio		2.0
Use of the CYCLE statement		1.0

Table 3
Characteristics of the software analysed.

Software	Size (MB)	Fortran Files
CESM1[35]	9.4	34
CFSv2-2011[36]	9011.2	1,144
IPSL, France[37]	27.9	2,220
GISS[38]	100.6	2,494
CCSM4[39]	19.8	1,138
NorESM1[40]	77.3	1,532
GFDL[41]	174.8	1,286
GEOS5[42]	187.0	1,455
ECHAM6[43]	34.0	658
MPI-M[44]	95.6	1,109
NICAM[45]	1400.0	247
ARPACK 2.1[46]	4.3	399
openblas 0.2.20[47]	156.3	3,691
qrupdate 1.1.2[48]	1.8	89
suiteSparse 4.5.5[49]	69.8	15
expokit[50]	4.3	35
igakit[51]	0.5	1
CP2K-4.1[52]	2010.2	1,227
ELK 4.3.6[53]	19.6	820
EXCITING boron[54]	100.7	3,095
SIESTA 4.1[55]	62.0	851
EPW 6.1[56]	86.5	1,331

Table 4
CMIP5 software scores.

Model	Score
CESM1	3.929
CCFSv2-2011	2.920
IPSL	3.974
GISS	2.774
CCSM4	3.536
NorESM1	3.703
GFDL	3.179
GEOS5	4.131
ECHAM6	3.863
MPI-M	3.630
NICAM	4.341

Table 5
Linear algebra libraries software scores.

Library	Score
ARPACK	0.710
openblas	0.839
qrupdate	1.356
suiteSparse	0.719
expokit	2.251
igakit	2.000

Table 6
Quantum chemistry computer programme scores.

Programme	Score
CP2K-4.1	3.468
ELK	3.649
EXCITING	1.822
SIESTA	3.954
EPW	3.640

Table 7
Comparison between IPSLCM5 and IPSLCM6 without configuration files

	IPSLCM5A			IPSLCM6.1.9-LR		
	score	nº files	nº lines	score	nº files	nº lines
IOIPS	4.04	11	18,729	4.04	11	19,673
IOSERVER	3.6	59	9,509	-	-	-
XIOS	2.32	39	16,034	3.66	155	32,030
NEMO	4.48	807	255,671	4.31	759	394,171
LMDZ	3.87	615	193,705	5.12	1,735	735,206
ORCHIDEE	4.12	56	55,245	4.03	88	122,028
OASIS	2.51	186	67,004	3.90	128	91,191
TOTAL	4.14	1,587	548,893	4.72	2,748	1,303,108

Table 8
Comparison of characteristics of existing tools to analyse Fortran code.

Analyser	Fortran Version	MultiPlatform	Natural Languages	Plugin	Compilation request	License
FortranAnalyser	all	Yes	4	No	No	GPLv3
Forcheck	all	No	1	No	Yes	GPLv3
Intel Paralell Studio XE	all	Yes	1	Yes	Yes	Privative
KScope	70 and 90	Yes	2	No	No	Apache 2.0
FtnCheck	90	Yes	1	No	No	MIT

Figures

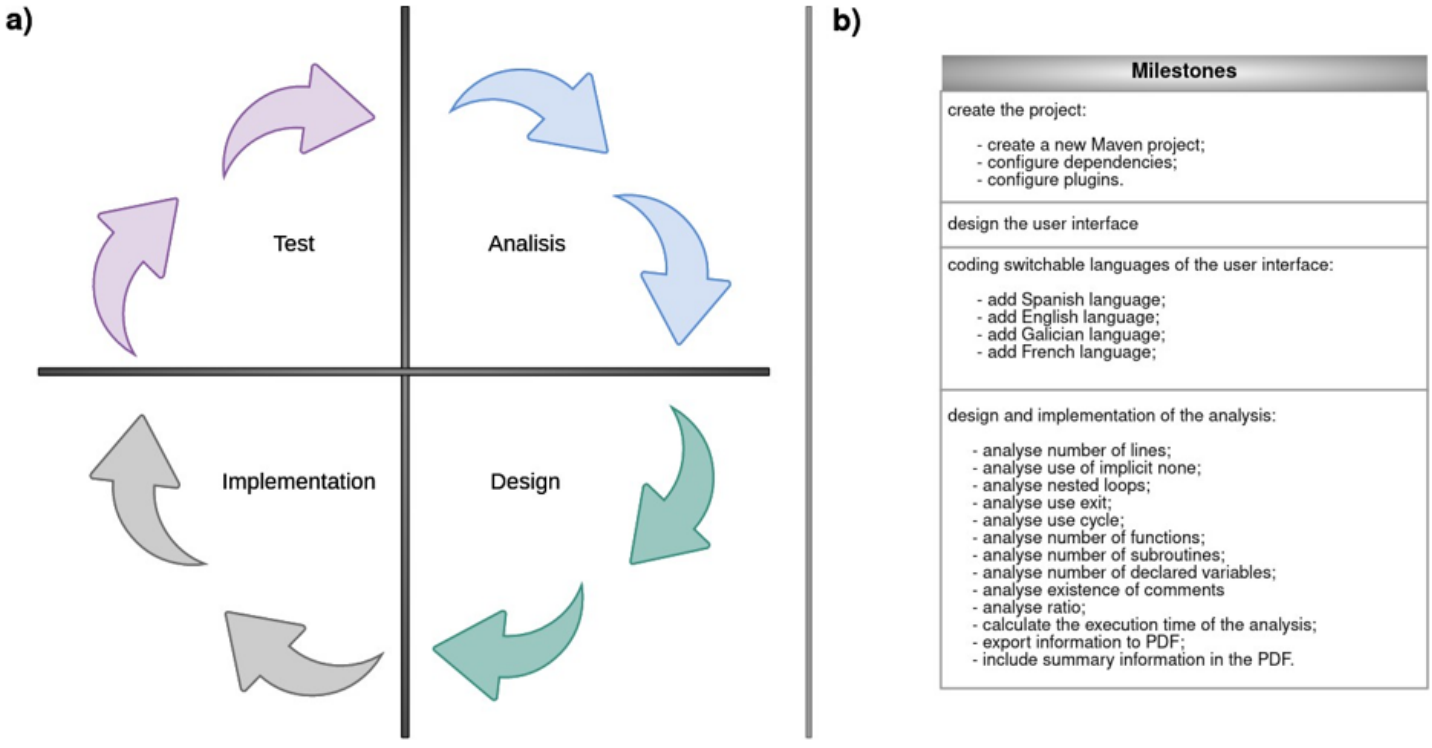

Figure 1

Development methodology used for the development of FortranAnalyser and listing of each one of the milestones.

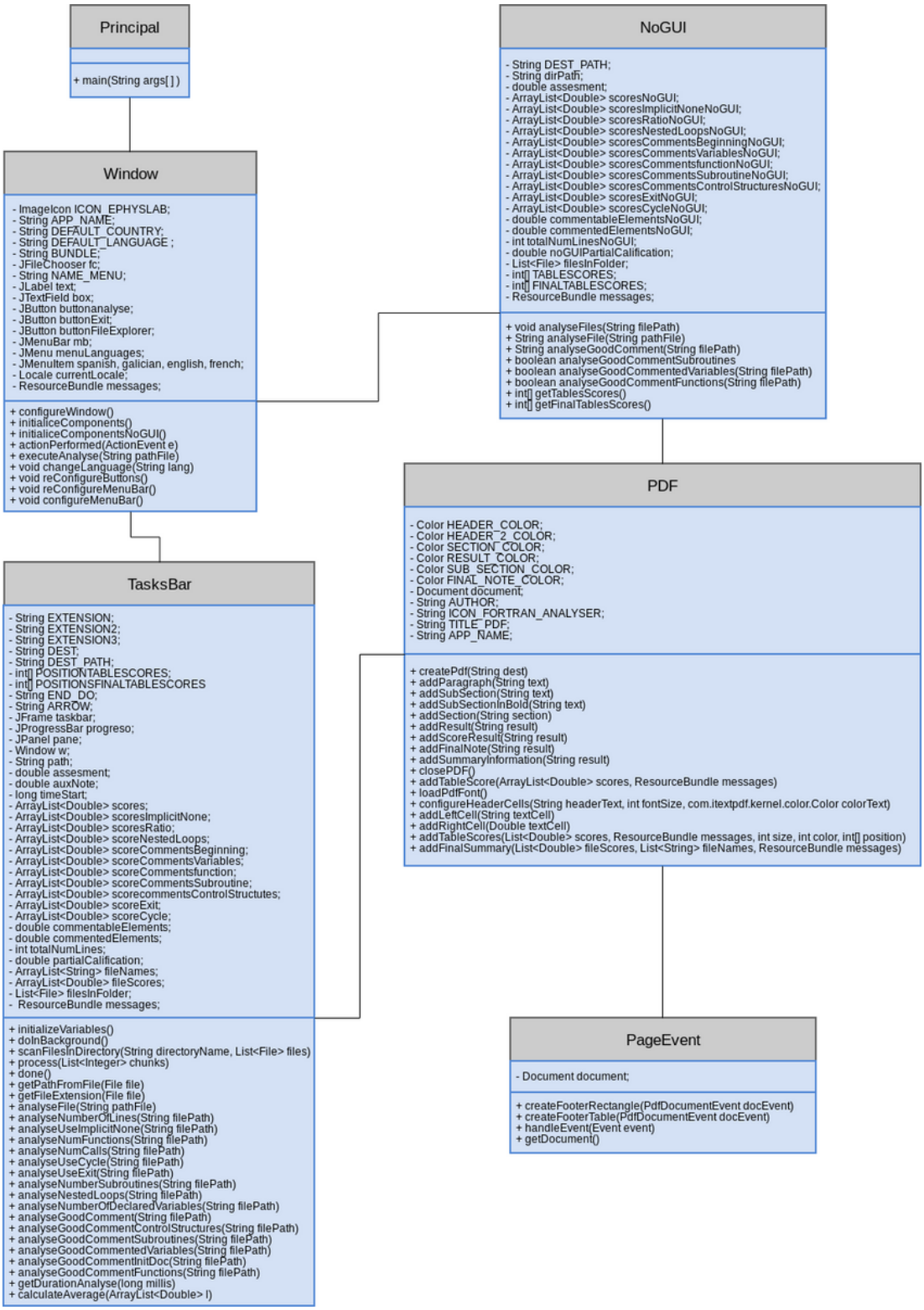

Figure 2

Class diagram.

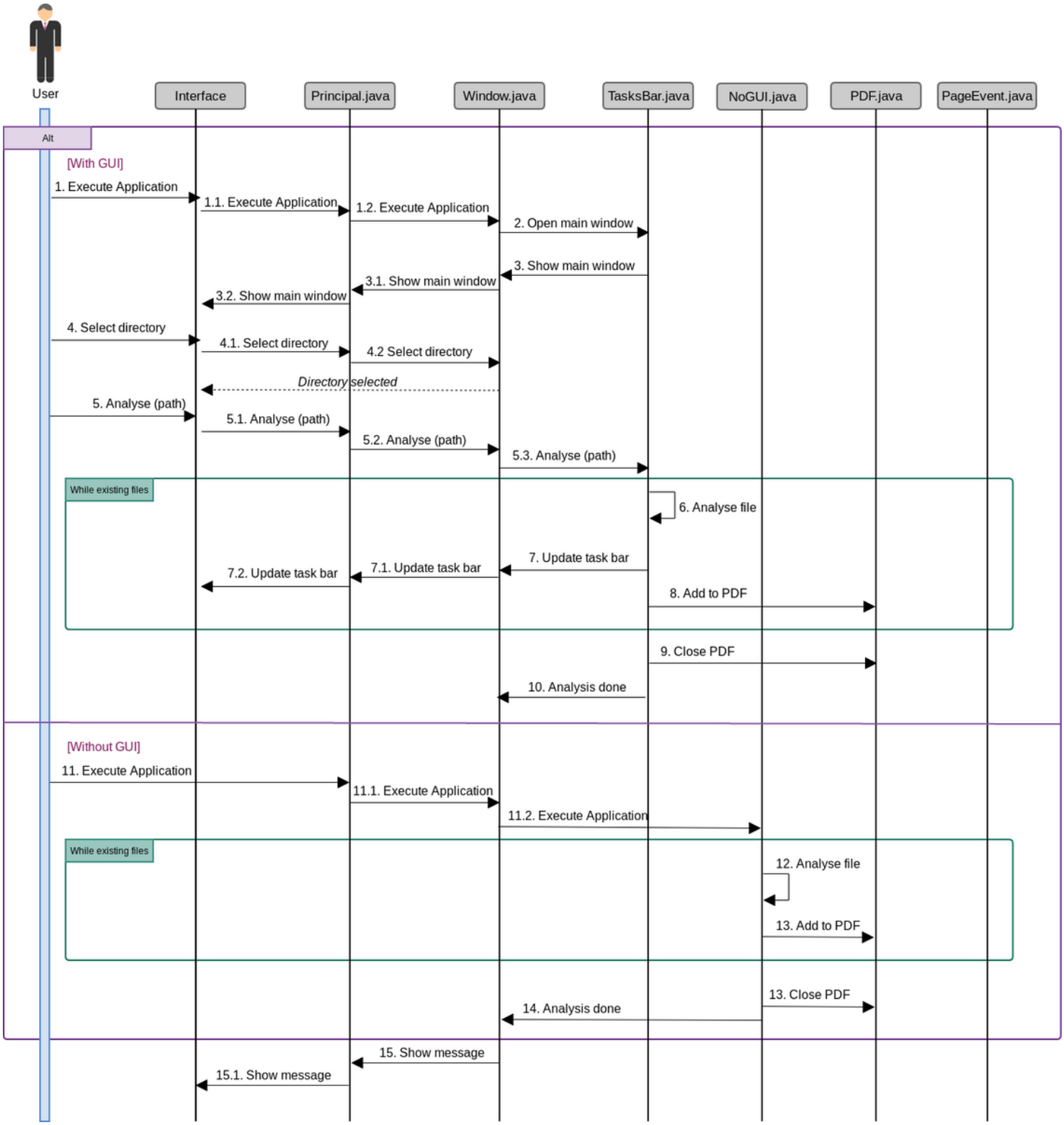

Figure 3

Detailed sequence diagram.

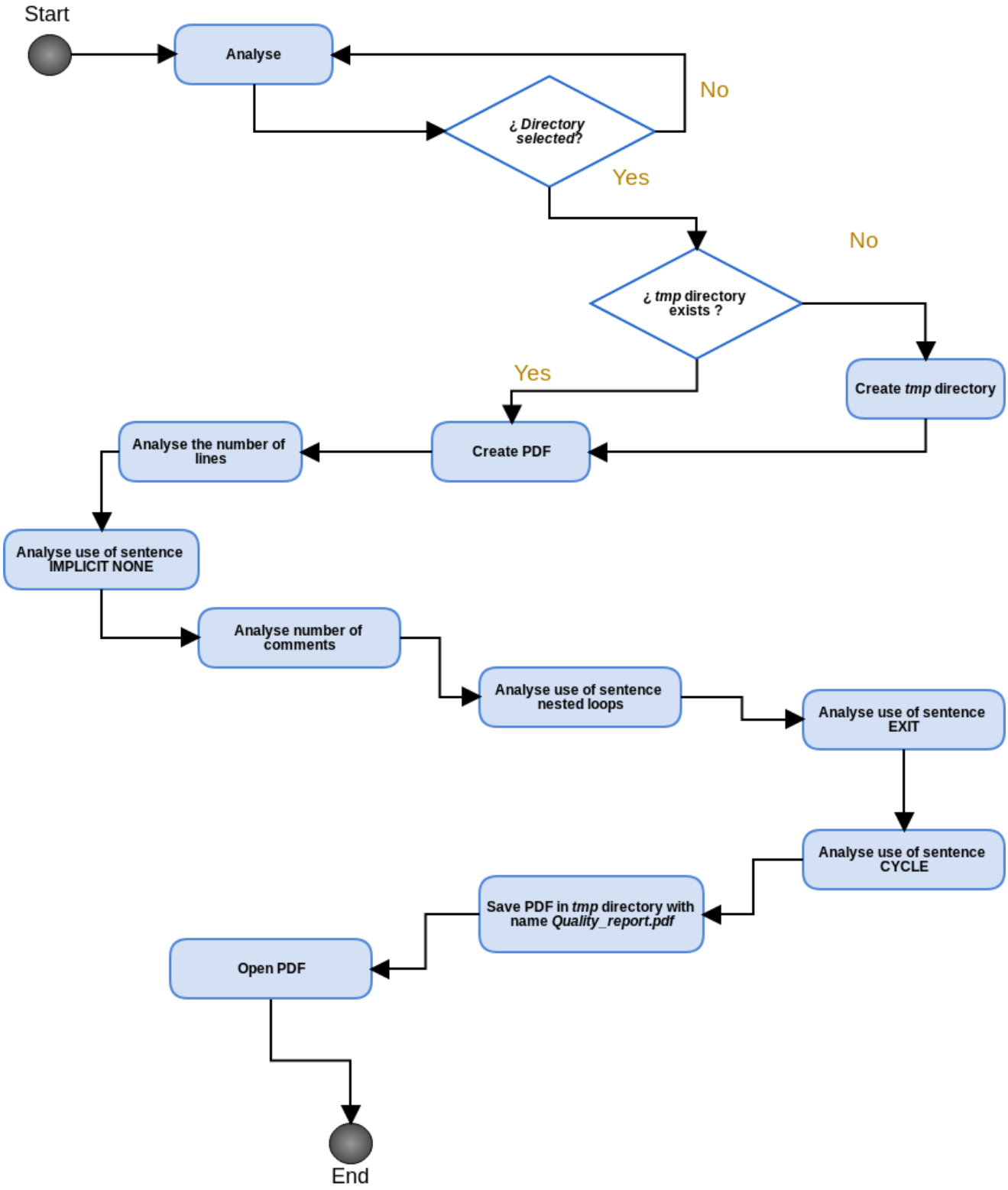

Figure 4

Flowchart diagram for the development of FortranAnalyser.

DISTANCE2.f90

Number of lines: 42
Implicit none:true
Number of functions: 1
Number of subroutines calls: 0
Number of comments:1
Number of declared variables: 17
Complies with maximum nesting complexity of loops and comments format: true
Good comments at:
--> declared functions: false
--> At the begining of the file: false
--> variables declatarion: false
--> soubroutines declaration: true
--> Control structures: false
Number of subroutines: 0
Use of the sentence EXIT to exit in the repetitive structures: true
Use of the sentence CYCLE to avoid making certain statements, iterating to the following element: true

Metric	Score
the use of the IMPLICIT NONE	2.0
number of lines with comments	0.038
number of nested loops	2.0
comments in the declaration of a function	0.0
comments at the beginning of the document	0.0
comments in the declaration of a variable	0.0
comments in the declaration of a subroutine	0.4
comments in control structures	0.0
use of the EXIT statement	1.0
use of the CYCLE statement	1.0

file calification: 6.44

Figure 5

Example of a report generated by FortranAnalyser. The code associated is available as supplementary material.

```
program cylinder

  integer :: ierr
  real :: radius, height, area
  real, parameter :: pi = 3.141592653589793

  interactive_loop: DO

    write (*,*) 'Enter radius and height.'
    read (*,*,iostat=ierr) radius,height

    if (ierr /= 0) then
      write(*,*) 'Error, invalid input.'
    end if


    area = 2*pi * (radius**2 + radius*height)

    write (*,'(1x,a7,f6.2,5x,a7,f6.2,5x,a5,f6.2)') &
      'radius=',radius,'height=',height,'area=',area

  end do interactive_loop

end program cylinder
```

Figure 6

Cylinder_v1.f90.

```
!! Example of a program with a good score obtains
!! by FortranAnalyser.
!! The following program calculates
!! the surface area of a cylinder and illustrates
!! free-form source input and other features
!! introduced by Fortran 90.
program cylinder

!! Use implict none sentence
  implicit none

  !! variable with the output of the IOSTAT
  !! specifier to continue program execution
  !! after an I/O error and to return
  !! information about I/O operations.
  integer :: ierr

  !! Anwer of the user to perform or not another
  !! calculation
  character(1) :: yn

  !! Variables that refers to the radius, height,
  !! and area of the cylindre
  real :: radius, height, area

  !! PI number
  real, parameter :: pi = 3.141592653589793

  !! loop to ask user to introduce information
  !! about the height and the radius.
  interactive_loop: do

    write (*,*) 'Enter radius and height.'
    read (*,*,iostat=ierr) radius,height

    !!IOSTAT: A value of '0' for normal completion
    if (ierr /= 0) then
      write(*,*) 'Error, invalid input.'
      cycle interactive_loop
    end if


    area = 2*pi * (radius**2 + radius*height)

    write (*,'(1x,a7,f6.2,5x,a7,f6.2,5x,a5,f6.2)') &
      'radius=',radius,'height=',height,'area=',area

    yn = ' '

    !! Answer the user if he want to perform another
    !! calculation
    yn_loop: do
      write(*,*) 'Perform another calculation? y[n]'
      read(*,'(a1)') yn

      !! in case the user want another calculation
      if (yn=='y' .or. yn=='Y') exit yn_loop

      !! in case the user want to exit of the programme
      else if (yn=='n' .or. yn=='N' .or. yn==' ') exit interactive_loop

      else cycle yn_loop


    end do yn_loop

  end do interactive_loop

end program cylinder
```

Figure 7

Cylinder_v2.f90.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- QualityReportCylinderV1.pdf
- QualityReportCylinderV2.pdf