ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV027

THE EVOLUTION OF THE DOOCS C++ CODE BASE

L. Fröhlich*, A. Aghababyan, S. Grunewald, O. Hensler, U. F. Jastrow, R. Kammering, H. Keller†, V. Kocharyan, M. Mommertz, F. Peters, A. Petrosyan, G. Petrosyan, L. Petrosyan, V. Petrosyan, K. Rehlich, V. Rybnikov, G. Schlesselmann, J. Wilgen, T. Wilksen,
Deutsches Elektronen-Synchrotron DESY, Germany

Abstract

This contribution traces the development of DESY's control system DOOCS from its origins in 1992 to its current state as the backbone of the European XFEL and FLASH accelerators and of the future Petra IV light source. Some details of the continual modernization and refactoring efforts on the 1.5 million line C++ code base are highlighted.

INTRODUCTION

DOOCS [1, 2] started out at DESY in 1992 as a control solution for vacuum devices – ion getter pumps and similar equipment – for superconducting cavity test stands. Later it was ported to the HERA [3] proton storage ring to replace an older predecessor. It was built around Sun Microsystem's remote procedure call protocol (*SunRPC*) that continues to be used today in many well-known services such as the Network File System (NFS) under the newer name Open Network Computing RPC (*ONC RPC* [4]).

At the time, object-oriented programming was quickly establishing itself as the dominant programming paradigm, which lead to the new system being called the "distributed object-oriented control system", DOOCS. It was implemented in C++, the natural choice for developing reliable software that could use high-level abstractions, work with limited resources, and access hardware efficiently. Java, the next "big" language to champion object-orientation, would not be released to the public until 1994/1995 [5].

The C++ of the year 1992 was quite different from modern versions of the language. It was six years before the first ISO standardization of the language [6], and many features taken for granted today were still in their infancy: Templates, namespaces, and exception handling had just been specified [7] but were not or only partially available on the compilers of the time. The standard template library (STL) and even the `bool` type would not be standardized until 1998. It is therefore only natural that early DOOCS sources, although heavily using classes and inheritance, look more like low-level C than modern C++ from today's perspective.

These first years established the backbone of DOOCS and formed many of the conventions that shaped the development of the code over the following two decades. Although the code was continually extended and maintained, its basic style changed relatively little until some years after the introduction of C++11 [8]. Although the new standard could not be adopted for the core libraries until 2018 due to lack of compiler support on the Solaris platform, it lead to renewed interest in C++ and modern programming styles and attracted new developers. Gradually, more and more effort was put into the modernization of the code base. We are going to highlight a few of the changes made during this ongoing modernization effort below. For orientation, we first give a brief overview of the DOOCS code base.

CODE ORGANIZATION

DOOCS consists of multiple libraries, tools, and servers, the biggest part of which is written in C++[1]. Two libraries form the core of almost every DOOCS application:

- The DOOCS client library (*clientlib*) provides the basic functionality to list `names()` from the DOOCS namespace and to send `get()` and `set()` requests over the network. It also offers interoperability with other control systems such as EPICS [10] and TINE [11, 12] by exposing some of their native functionality via the DOOCS API.
- The DOOCS server library (*serverlib*) provides the building blocks for a DOOCS server which accepts requests from the network. It contains classes for *properties* of various data types and allows instantiating these properties as members of *locations* with user-defined functionality. It also handles archiving, configuration management and related tasks.

These two libraries are complemented by approximately one hundred other DOOCS-related libraries of various purposes, ranging from support for specific hardware like cameras over interfaces to data acquisition (DAQ) systems to high-level algorithms for particle tracking.

Most of the DOOCS applications are servers. They connect hardware devices such as beam position monitors or vacuum pumps to the network, process and archive data, run feedback loops, and execute complex algorithms for advanced data evaluation. Currently, our repositories contain source code for more than 500 different server types. For most of these, multiple instances are running at one or more of DESY's accelerator facilities. Between libraries and servers, the C++ code base consists of ~8000 source files with a total of ~1.5 million lines of code[2].

Figures 1 and 2 show the development of the number of lines of code in the client- and serverlib over time. It is worth noting that the clientlib was dominated by C code between

* lars.froehlich@desy.de
† retired

[1] Notable exceptions are the JDOOCS client library for Java and tools based on it such as the graphical user interface builder *jddd* [9]. Client libraries for Python, Matlab, and LabView also exist and have spawned a multitude of tools in these languages.

[2] We count lines of code excluding comments and blank lines using the `cloc` [13] utility.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV027

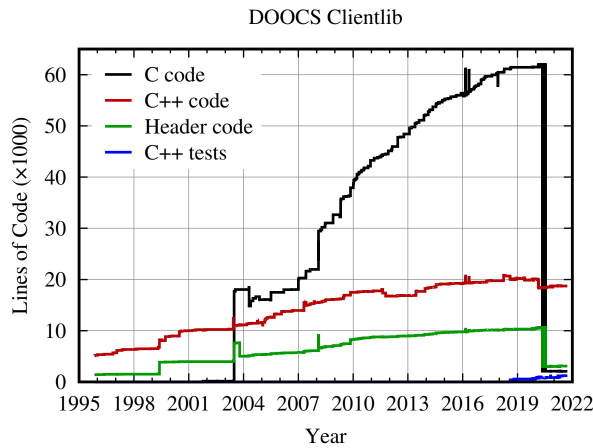

Figure 1: Lines of code excluding comments and blank lines in the DOOCS clientlib. The plot differentiates between C implementation files, C++ implementation files, header files, and C++ unit tests. From 2003 to 2020, the library was dominated by C code from the included TINE client library.

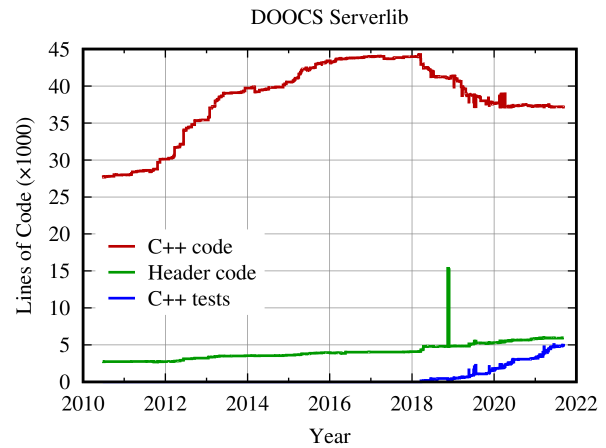

Figure 2: Lines of code excluding comments and blank lines in the DOOCS serverlib. The plot differentiates between C++ implementation files, header files, and C++ unit tests. Development started in 1992, but the history preceding the year 2010 has been lost.

2003 and 2020 because of the inclusion of the full client and server sources for the TINE control system. Since last year, TINE is linked as an external shared library. This has allowed us to preserve the full set of interoperability features without having to deal with compiler warnings from its legacy code base. A small amount of C code still remains for the interface with the RPC library.

MODERNIZATION OF CODE AND DEVELOPMENT TECHNIQUES

Modernizing a code base whose oldest parts date back to 1992 has its risks. For the most part, bugs in the existing code have been ironed out many years ago, and refactoring tends to introduce new ones. The almost complete absence of unit tests until 2018 means that inadvertent behavior changes introduced through modifications are very hard to detect. Many desirable changes in libraries are also impossible without sacrificing the stability of the application programming interface (API), causing the need to adapt huge numbers of dependent projects.

While all of these can be good reasons to leave existing code untouched, we also observed several tendencies that slowly became a cause for concern:

- Certain parts of the code had become so complex that they could only be understood by single developers.
- Introducing new features was becoming harder and more error-prone.
- Attracting new programmers to actively participate in the development was becoming harder.

All of this lead us to the conclusion that a careful modernization not only of the code base, but also of our development techniques would be necessary to prepare DOOCS for future projects. We set ourselves the following goals:

- Improve the *readability* of the code to make it easier for all developers to understand how it works.
- Improve overall *maintainability* by better code organization and reduced complexity.
- Improve *stability* by eliminating sources of bugs, memory leaks, and undefined behavior.
- Improve *teamwork* by collaborative development techniques and code reviews.

We discuss some of the measures taken in this spirit below, but the list is necessarily incomplete. In fact, outside of the core libraries individual developers have wide freedoms to adopt them only partially or to go far beyond them.

More Modern C++ Constructs

A huge part of the changes that started in the middle of the 2010s regards the use of the C++ language and its standard library itself:

Language standard: Solaris was dropped as a development platform for DOOCS in 2018. This has allowed us to use the C++14 [14] standard with the compilers for all remaining supported platforms since then. We expect to upgrade to C++17 [15] and possibly later standards in the coming years when support for a few old Linux distributions is dropped.

Templates had practically been banned from the core libraries after bad experiences with compilers in the 1990s. Through their (re-)introduction, the amount of duplicated code could be reduced dramatically while improving the uniformity of interfaces and their type safety at the same time. One of the visible effects for DOOCS users is the availability of a full set of signed and unsigned integer types in various bit widths.

Const-correctness was largely ignored in the original APIs. Retrofitting `const` modifiers across API and inheritance boundaries can be a delicate task as it often involves changes

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV027

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

to multiple projects at the same time. Although a lot of work remains to be done, the public APIs have already improved a lot and make it much easier to write const-correct code.

String handling in the original core libraries almost exclusively meant dealing with C strings or raw memory. Some of that code was rewritten to use the standard `string` or `string_view` classes, dramatically reducing complexity, line count, and room for errors. Virtually all of the user-facing API now uses these classes in function signatures instead of `const char*` or even `char*`.

Memory handling was mostly based on raw pointers and explicit use of `new` and `delete`, making new code prone to memory leaks. This type of code is gradually being replaced by STL containers, RAII (resource acquisition is initialization) types, and smart pointers.

Algorithms from the STL were not widely used in the original code base. Their increased application helps make many code passages much more succinct and readable.

Standard types have replaced entire platform-specific libraries. For instance, the standard `thread` is now used instead of its counterpart from the POSIX threads library.

Strong types are gradually being introduced to replace fundamental types like `int` for specific applications (e. g. `Timestamp`, `EventId`). This makes it harder to confuse function parameters and allows enriching the type with functionality and invariants.

Unit Tests

Apart from a few very limited integration tests, none of the core libraries had a real test suite until 2018. At this point, we started writing unit tests using the Catch2 [16] framework. Most tests are written to assert the current behavior of the code before modifying or refactoring it. Unfortunately, parts of the code have complex dependencies that make testing hard. This situation can only be improved through partial rewrites in the long run. Both the client- and the serverlib remain severely undertested with 1 line of tests for 16 lines of code for the former and 1:8 for the latter, but the test suites are growing continuously.

Build System

We are in the process of changing our build system from `make` to Meson [17]. Overall, this makes our build setup much simpler and less platform dependent. Meson was chosen over similar alternatives because of its comparatively simple syntax and because of in-house expertise.

General Utility Library GUL14

Even in comparatively small C++ code bases, some basic functionality gets implemented again and again because it is not available through the standard library – for instance, a function to determine if a string contains another string[3]. This has lead to the development of several *base libraries* [18–20] in the industry attempting to fill this gap.

[3] A `contains()` member function for the standard `string` class is only expected for the 2023 language standard.

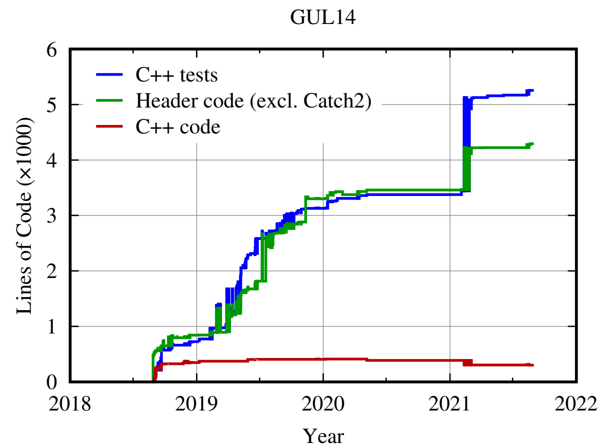

Figure 3: Lines of code excluding comments and blank lines in GUL14. The plot differentiates between header files, C++ implementation files, and unit tests. We do not count the header of the Catch2 unit testing framework which is included in the library.

We evaluated several of them and, amid concerns about their impact on our build process, found all of them lacking some specific functionality that we needed.

In August 2018, we therefore started the development of our own *General Utility Library for C++14* (GUL14). The library gathers code of wide applicability that has no external dependencies except for the C++ and C standard libraries. Specifically, it is free from control system specific code. Among other fields, it covers string operations, concurrency, time, statistics, numeric utilities, containers, and backports of features from post-C++14 standard libraries (e. g. `string_view` and `span`).

As it forms the basis for all other DOOCS libraries and applications, we strive for a very high quality level for GUL14:

- *Style:* Code must adhere to our C++ style guide and it should follow the C++ Core Guidelines.
- *Documentation:* Every function, type, or other entity in the API must be fully documented. This includes a clear description of purpose and functionality. For functions, all parameters, return values, thrown exceptions, and pre-/postconditions are described. For classes, all class invariants are clearly stated.
- *Unit tests:* Every entity in the library must have a set of associated unit tests.
- *Code review:* Every commit to the library must be reviewed by at least one developer. Every developer must ensure that all of the other quality criteria are fulfilled.

As shown in Fig. 3, the library is dominated by template code residing in header files. The unit test suite has more lines of code than the library itself.

GUL14 is open source and published under the GNU Lesser General Public License v2.1 [21]. Unfortunately we currently cannot publish the code on a third-party platform

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV027

like Github, but the source code and documentation are publicly available [22] and external contributions are welcome.

Code Review

Code reviews are an excellent method to screen code changes for defects. Maybe more importantly, they foster mutual learning between developers and thereby help spread expert knowledge. Unfortunately, they also take a substantial amount of time. Owing to the already high workload on our development team, we have adopted a multi-layered guideline:

1. For GUL14, every commit must be reviewed and accepted by at least one other developer.
2. For the client- and serverlib, all but trivial commits must pass review.
3. For other libraries, code review is strongly encouraged.
4. For servers and tools, code review is optional.

Code reviews typically take place via merge requests on an on-site Gitlab web server.

Continuous Integration

The code for the core libraries is automatically compiled and tested on multiple platforms by the Gitlab system after each commit. In part, this includes address sanitizer and undefined behavior sanitizer builds, which generally provides excellent early warnings against faulty code. We are also looking into continuous delivery (CD) workflows for the future.

Cross-Project Refactoring

Some refactorings, especially API changes in libraries, require invasive modifications in user code. Although we generally try to avoid such changes, a few of them are deemed essential to fix architectural problems. Since DOOCS is used almost exclusively at DESY with only few external users, we are in the lucky situation that we have control over almost all user code in our version control repositories. Refactorings of low complexity (such as renaming a class member or a header file) can therefore be automatically applied via text substitution scripts. Where possible, the old API is marked as deprecated and remains available for 1–2 years until it is finally removed.

Training

Educating new programmers is essential to maintain a capable control system software team. To this end, we are providing a regular series of remote lectures on DOOCS related topics. A C++ style guide provides orientation on common coding conventions.

CONCLUSION

Next year, DOOCS will turn 30. As in any code base of this age, some problems have accumulated over time and tend to hinder further development. Since the mid-2010s, however, we have invested more and more effort into the continuous refactoring of the code and into a modernization of our development process. We try to take advantage of established best practices, of recent improvements of the C++ language itself, and of modern tooling to improve the quality of the code and the cooperation between the developers. The feedback received from our users makes us hopeful that we are on the right path to prepare DOOCS for the future.

ACKNOWLEDGEMENTS

We would like to thank J. Georg, M. Hierholzer, D. Kalantaryan, M. Killenberg, and T. Kozak for contributing patches and bug reports for the core DOOCS libraries. Innumerable colleagues from the the M-, FS-, and FH- groups have helped shape the development of DOOCS over the years and deserve our gratitude. We also acknowledge the continuous support by the machine and research divisions at DESY, the Helmholtz Association of German Research Centers, and the European X-Ray Free-Electron Laser Facility GmbH.

REFERENCES

[1] O. Hensler and K. Rehlich, "DOOCS: A distributed object oriented control system", in *Proc. XV Workshop on Charged Particle Accelerators*, Protvino, Russia, 1996.

[2] DOOCS homepage, `https://doocs.desy.de`

[3] G. A. Voss and B. H. Wiik, "The electron-proton collider HERA", *Annual Review of Nuclear and Particle Science*, vol. 44.1, pp. 413-452, 1994.

[4] R. Srinivasan, *RPC: Remote procedure call protocol specification version 2*, IETF Request for Comments 1831, August 1995, `https://www.ietf.org/rfc/rfc1831.txt`.

[5] D. Bank, *The Java saga*, Wired 3.12, Dec. 1995.

[6] *International standard ISO/IEC 14882:1998*, International Organization for Standardization, Geneva, Switzerland, Sept. 1998.

[7] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, ISBN 0-201-51459-1, May 1990.

[8] *International standard ISO/IEC 14882:2011*, International Organization for Standardization, Geneva, Switzerland, Sept. 2011.

[9] E. Sombrowski et al., " 'JDDD': A Java DOOCS data display for the XFEL", in *Proc. ICALEPCS07*, Oct. 2007, pp. 43–45.

[10] L. R. Dalesio et al., "EPICS architecture", *Proc. ICALEPCS91*, Nov. 1991, pp. 278–282.

[11] P. Bartkiewicz and P. Duval, "TINE as an accelerator control system at DESY", *Meas. Sci. Technol.*, vol. 18, pp. 2379–2386, 2007. `doi:10.1088/0957-0233/18/8/012`

[12] P. Duval *et al.*, "Control system interoperability, an extreme case: Merging DOOCS and TINE", in *Proc. PCaPAC2012*, Dec. 2012, pp. 115–117.

[13] cloc tool for counting lines of code, `https://github.com/AlDanial/cloc`

[14] *International standard ISO/IEC 14882:2014*, International Organization for Standardization, Geneva, Switzerland, Dec. 2014.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV027

[15] *International standard ISO/IEC 14882:2017*, International Organization for Standardization, Geneva, Switzerland, Dec. 2017.

[16] Catch2 unit test framework, https://github.com/catchorg/Catch2

[17] Meson build system, https://mesonbuild.com/

[18] Google Abseil library, https://abseil.io/.

[19] Bloomberg BDE libraries, https://github.com/bloomberg/bde

[20] Facebook folly library, https://github.com/facebook/folly

[21] GNU Lesser General Public License version 2.1, https://www.gnu.org/licenses/old-licenses/lgpl-2.1

[22] General Utility Library for C++14, https://winweb.desy.de/mcs/docs/gul/index.html

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI