Medical Device Regulation

First Aid for Old Code

In addition to the international standard IEC 62304 – medical device software – software life cycle processes, a new Medical Device Regulation (MDR) valid since May 2021 came into force in the European Union. It is not the first time that quality assurance for medical device software has come into focus. IEC 62304, MDR and other standards stipulate that manufacturer must ensure quality assurance throughout the entire life cycle of a product. In the case of current systems, this is generally hardly problematic. The situation is quite different when new functions are to be added to older devices: who in the company still knows the code or where to find the - hopefully existing - documentation?

Although the IEC 62304 and MDR only affect the medical sector, the issue is nonetheless known in other industries as well.

Here, tools such as Imagix 4D, which analyze the structure of a program and thus support the developers, can help.

by Klaus Lambertz, Managing Director of Verifysoft Technology GmbH

The digital transformation has also gained massive momentum in medical technology. Almost every new device has software, wireless connections and the ability to read data from sensors. This creates important new opportunities in therapy and diagnostics, but also new risks. Scenarios in which hackers attack medical devices no longer belong in the category of science fiction but are becoming real. It is therefore essential to minimize risks for patients and operating personnel by ensuring the best possible quality of the equipment. This is the aim of the numerous standards that are relevant in medical technology. With the Medical Device Regulation EU 2017/745 (MDR), the requirements of quality assurance are becoming even more central.

IEC 62304, MDR or ISO 14971 demand in unison, but usually without concrete assistance, that a manufacturer must implement quality assurance and risk management processes. In the case of embedded systems and stand-alone software, two areas must be distinguished in the context of quality assurance. First, during development, the goal is to avoid code errors (verification) and to ensure the required functionality (validation). On the other hand, systems must be considered throughout their life cycle. Products that are essentially based on software can be subject to significant changes over the course of their lifetime, for example due to updates or new functions being added. Changes are also done when a library used during development is replaced by a newer version. The relevant standards and regulations take both aspects into account. For example, the MDR makes clear: "For products whose components include software, or for products in the form of software, the software shall be developed and manufactured in accordance with the state of the art, taking into account the principles of software life cycle, risk management, including information security, verification and validation."

Problems come with age

New products are unproblematic with regard to quality assurance throughout the lifecycle if appropriate processes have been anchored in the company. Especially with today's agile development methods such as Continuous Integration/Continuous Deployment, documentation plays a major role. The principle of "clean code", i.e. code that is clean and free of all superfluous convolutions, has also become an important component of many development departments. One element of this is refactoring, which is intended to improve the code. Code is not perfect from the beginning, all parts must be subjected to permanent reviews. The task of refactoring is to bring the code into a form that is desirable for the developers, i.e. easily comprehensible. Refactoring has two main goals: To make the extensibility of the code as easy as possible and at the same time to ensure maintainability. In addition, it should be achieved that the code can be reused in whole or in parts in later projects. Unlike debugging, however, refactoring does not affect the behavior of the program. The code is not functionally changed. Strictly speaking, errors or security problems found during refactoring are not eliminated, but only marked for cleanup.

If refactoring is done as an integrative measure in the ongoing development process, the effort is manageable. However, this is not the case with older systems: The longer an application is in operational use without optimization of the code, the more difficult this code becomes to understand. This is because over the lifecycle of an application, changes and adaptations must be made again and again, which influence the behavior of the software. The effort required for maintenance and modernization increases rapidly as knowledge of the architecture and functionalities dwindles.

The extreme case is legacy code: legacy code is challenging in terms of maintenance or enhancement. Most of the time, the code is extremely confusing. Often the most basic documentation is missing. And the developers responsible at the time are retired or scattered to the four winds. Nevertheless, the software has to be extended with new, up-to-date functionalities. Often, errors have to be eliminated that remained undiscovered until now. For the developers entrusted with this task, a detective search for traces in the old structure then begins, which also demands archaeological qualities.

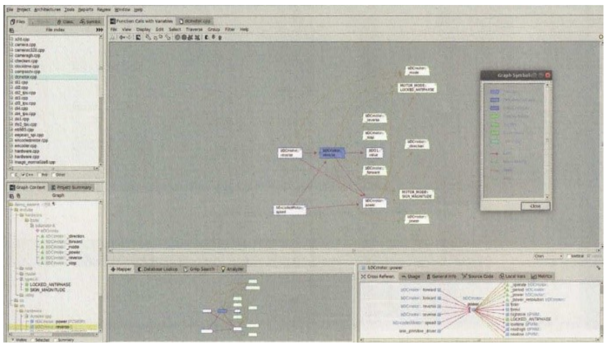
Fig.1: Function call diagrams show the sequence of called functions and further information.

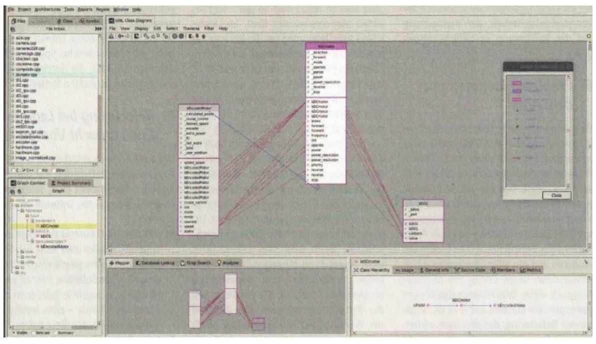
Fig.2: The UML class diagram expresses properties of classes and relationships between them in UML notation.

Refactoring of legacy code requires preparation

In order to prepare old code in such a way that a planned refactoring can be carried out with reasonable effort, the following aspects of the software should first be examined:

- Files: In the C context, files are either headers or compilable files, i.e. the physical components of the software. Here it is important to know what relations these have to each other - for example, what common headers they have.
- Subsystems: Which subsystems are there and in which relations do they stand to each other? What architecture underlies the subsystems?
- Data types: Types are usually pointers, enums, classes, structs and the like. Here, the relationships between types and variables are of particular interest.
- Functions: The call hierarchy of functions within a project is elementary important to understand the code. Both incoming and outgoing calls should be considered. The control flow between functions is also relevant, i.e. at which point a jump is made to another function. In turn, branches and loops in the program must be known since these influence the control-flow.

These analyses cannot be performed manually above a certain project complexity. Simply identifying the relationships between the various files in a project is an error-prone task. The use of suitable tools is inevitable to carry out analyses that can be automated as far as possible. A proven tool for the analysis of source code in C/C++ and Java is Imagix 4D, a tool developed by Imagix Corp. USA and distributed in Europe by Verifysoft Technology.

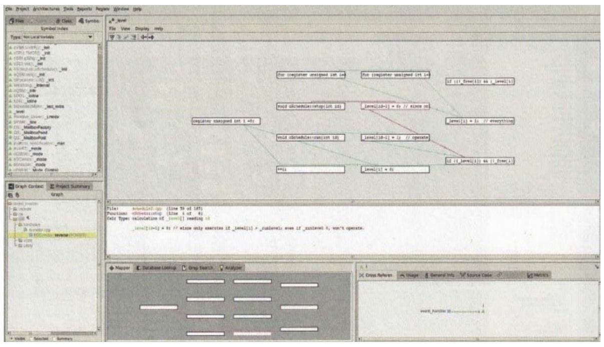

Fig. 3: The calculation tree of a variant shows which values and other variables contribute to a variable and which other variables are affected

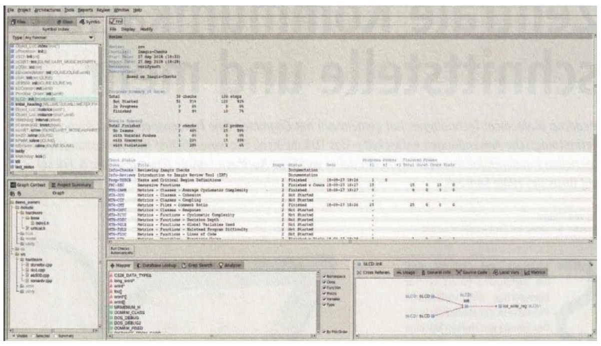

Fig. 4: With the review function, Imagix 4D supports otherwise purely manual processes as a central semi-automated tool

Graphical preparation of the structure

Imagix 4D analyzes the source code of a software and graphically prepares the information relevant for refactoring. This provides the developers with a representation of the entire project, showing all relations in the required level of detail. Depending on the question, the tool has different display modes. To provide an overview of the dependencies of all subsystems present in a project, the information is prepared in the form of a design structure matrix. This allows, for example, the granularity of the subsystems to be broken down from the root directory to the level of individual functions. For a better understanding of the subsystem architecture, this can in turn be displayed as a diagram. In the case of unclear architectures, for example with a large number of files directly in the root directory, filters help to find the right focus. Numerous other views, for example for displaying function dependencies or control flows, provide developers with further detailed information. Another important feature is the search for anomalies in the code to specifically increase the quality of the application. These include recursions, deadlocks, unused variables or inappropriate type conversions.

With this knowledge, it is possible to understand the existing code and trace its functionality. In the next step, the code can then be cleaned up and brought into a coherent form as part of refactoring. In addition, the use of Imagix 4D makes it possible to create comprehensive documentation with reasonable effort - indispensable for any certification that may be required. On this basis, the application can then be provided with new functions. In addition, the basis is prepared to continue operating the old project with current approaches of agile development.

Conclusion

Lifecycle quality assurance is not new in medical technology, but it is gaining weight due to recent technological developments and the MDR. To ensure quality even for long-lived products or products widely used in the field, the code must first be known - a real problem in many cases. Refactoring has established itself as an integral part of agile development methods for a reason. Legacy applications also benefit from it - especially if they have not yet reached the end of their lifecycle. For this, however, the existing source code must be analyzed in detail, as refactoring should in no way change the behavior of the software. Trial-and-error approaches are out of place here.

Without suitable tools, the analysis of complex applications is hardly possible, and errors cannot be excluded with economically justifiable effort. A graphical presentation of the architectures and the underlying structures gives developers a good understanding of how an application is structured and where the right entry points for refactoring are. This means that even old systems can be brought up to a level where the quality and risk management requirements can be met. Everyone benefits from this: greater safety for patients and a longer product life for manufacturers.

Further information can be found at

https://www.verifysoft.com/en_imagix4d.html