Onboard software

12

Santiago Iglesias Cofán and Arno Formella
Computer Science Department, University of Vigo, Ourense, Spain

1 Introduction

Thanks to hardware evolution over the last 50 years, software has been increasingly introduced into the onboard computers (OBCs) of satellites. The capabilities of flight software are constantly increasing due to this continuous evolution of technology [1]. As a result, software has acquired fundamental responsibilities and plays a crucial role in almost all space missions [2, 3].

Nowadays, the onboard software (OBSW) also referred to as flight software (FSW) has become integral to the functionality that a satellite provides. Proof of this is the close relationship between the concept of operations (CONOPS) and the software requirement specifications as seen in the degree of functional suitability, reliability, performance efficiency, maintainability, and portability being achieved by the software development.

However, software is subject to faults which may lead to failures, and a mission can fail due to an implementation error. For this reason, it is very important to follow a well-defined development process and good engineering practices to try to minimize risks.

One of the principal challenges that a software engineer has to face is to achieve a reasonable balance between the characteristics listed earlier and the constraints imposed onto the project, such as available hardware, time schedules, human resources, test environment, etc.

This chapter focuses on the software operating on the onboard computer (OBC), also referred to as the onboard data handling system (OBDH) of a satellite which usually controls and manages all other subsystems. The following sections describe the most important aspects to be taken into account when developing OBSW and are organized as follows. Section 2 introduces the main responsibilities of the OBSW of a CubeSat, Section 3 shows different architecture aspects to be considered in the software design phases, Section 4 explains the importance of performing a complete and reliable development process in CubeSat software projects, Section 5 shows some specific details of real-life missions, and finally, Section 6 concludes the chapter with an overview of software development in CubeSat projects.

CubeSat Handbook. https://doi.org/10.1016/B978-0-12-817884-3.00012-6

2 Responsibilities of the onboard software

The OBC is the central subsystem in the satellite [4]. The OBC, from a hardware perspective, is described in Chapter 10, while this chapter focuses on OBSE running on the OBC. In general, the OBC stores, loads, and executes the OBSW to perform mainly the following actions:

- control of subsystems,
- control of payloads,
- management of communication channels,
- telemetry generation and telecommand handling (TMTC), and
- failure detection, isolation, and recovery (FDIR).

The control of subsystems consists of commanding different components integrated into the overall system, such as real-time clocks (RTC), electronic power system (EPS), altitude and orbital control system (AOCS), platform sensors, etc., and collecting their data to be stored onboard, to be transmitted to the ground segment, or to be monitored for failure detection.

The control of payloads is similar to the control of subsystems, being considered less critical. Depending on the type of payload and the software that the payload itself executes, the generated scientific data can be managed and transmitted by the payload itself. However, general control, especially operation scheduling, is usually coordinated by the OBC.

The management of communication channels means the selection and the usage of different physical or virtual channels to perform the contact with a ground station. Depending on the type of information and the requirements of the mission, the OBSW could transmit or receive data through different frequency bands (UHF, S-band, X-band, etc.) and use different transport protocols and data encoding.

The TMTC refers to services that the OBSW provides for satellite operation. Reporting of housekeeping data, task scheduling, event logging, and storage of telemetry are some examples of services that an OBSW could offer to satellite operators (at the ground station) for commanding the space segment and obtaining information from each of its components.

FDIR is a very important responsibility to be assumed by the OBSW and consists in monitoring sensors, registers, memory values, or any other condition that allows to detect certain software or hardware failures. The OBSW can notify the problem to the ground segment, handle or fix the problem to a certain degree onboard, or, at least, try to prevent an error propagation. An important issue in FDIR is the analysis of possible failure chains and the possibility to override onboard actions by ground commands.

3 Software architecture

A software architecture is a description of how a software system is organized [5]. It further defines and models the interactions between components and system elements.

The software architecture is a major topic in the software development process, because it drives the implementation and testing phases as it is the backbone that supports and guides these stages. The design of the software architecture has a direct impact on costs, time, efficiency, and complexity of the project. Requirements, risks, hazards, mission constraints, and available hardware are crucial factors that must be taken into account in the design of an adequate software architecture. Usually, the software architecture allows developers to maintain, scale, and add new features to the existing software. In addition, understanding the underlying design principles is vital to encounter bugs or to detect unexpected behaviors.

The decomposition of the software into components is a common element of the software architecture, because it allows developers to break down the overall task into simpler subproblems and to assign responsibilities, both in functionality and in development, in a coherent way. The selection of adequate programming languages and other software technologies plays a fundamental role, because they can notably affect the design of solutions, the reuse of existing libraries, and the testability of the software. The usage of object oriented or procedural languages, the utilization of single-tasking or multitasking operating systems, the employment of file systems or the direct access of memory, etc., are decisions to be made at this stage.

As the design of a software architecture is a frequent activity in any software development process, it is common to reuse solutions and decisions already known. These solutions are often called patterns and they can be tailored for a specific use case in an overall design process. Common software architecture patterns are described below:

Pipe-and-Filter is an architectural pattern in which the data are passed through a set of software components that transform or filter it. The result or output of each component becomes the input of the next component forming a processing pipeline. Components are independent to each other, they only share the data to be processed.

Layered Architecture is a pattern that consists of several horizontal layers stacked on top of each other. Each layer has a responsibility to perform a set of actions. Normally, lower layers provide services to the upper layers through a well-defined programming interface. In this way, the implementation details and underlying technologies in lower levels remain hidden to the higher level components.

Service-Oriented Architecture is a pattern where components are entities that provide services to other components through a communication protocol. This protocol is the interface among the services. The pattern allows to use components to be distributed with a somewhat loose coupling.

Note that all of these patterns can be present simultaneously in an OBSW architecture at different levels of the design hierarchy. For example, the main architecture of the system can be seen as a stack of layers that provide more and more abstraction in the bottom-up direction. This allows designers to update or replace the implementation without affecting upper layers. Note that it is very important to define and use sufficiently generic interfaces between the layers. As an example, Fig. 1 shows the following layers:

- BIOS (basic input/output system)
- RTOS (real-time operating system)

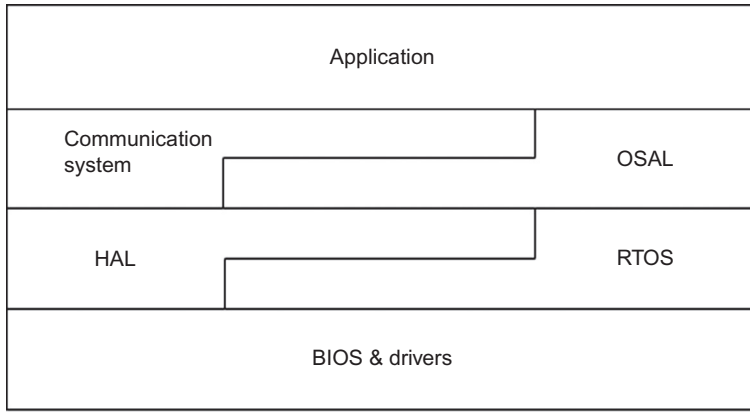

Fig. 1 OBSW layers. Example of a layered structure for an architecture of an onboard software.

- HAL (hardware abstraction layer)
- OSAL (operating system abstraction layer)
- CSL (communication system layer)
- APP (application layer)

The BIOS is a software layer, typically located in a nonvolatile memory, that is used to initialize all hardware components that need such an initialization or configuration step. In most cases, the BIOS contains a bootloader that loads certain other components of the OBSW to memory. This process is called booting and it finalizes once the operating system is started.

The RTOS is a software system that manages the computer hardware taking into account specific time requirements. Often, it provides multitasking and interrupt management through particularly suited scheduling algorithms. FreeRTOS [6], VxWorks [7], and RTEMS [8], are examples of such real-time operating systems. As for all software reusage, one must take into account possible risks and vulnerabilities that might be present.

The HAL allows the operating system, the communication system, and the application layer to use features of the platform in a more homogeneous way regardless of the employed hardware. The management of memories and communication buses and other instruments should be handled in this component as well.

The OSAL allows the communication system and the application layer to use common features of the operating system in a homogeneous way regardless of the underlying operating system. The management of tasks and files (or more general, permanent storage) should be located in this component.

The CSL is the software layer that allows interactions with external systems. In general, the communication system is inspired in the OSI model [9], which organizes the communication process in a certain number of sublayers.

The application layer (Fig. 2) represents all services that the OBSW should provide to the rest of the satellite. This last layer, in turn, can be based on a service-oriented

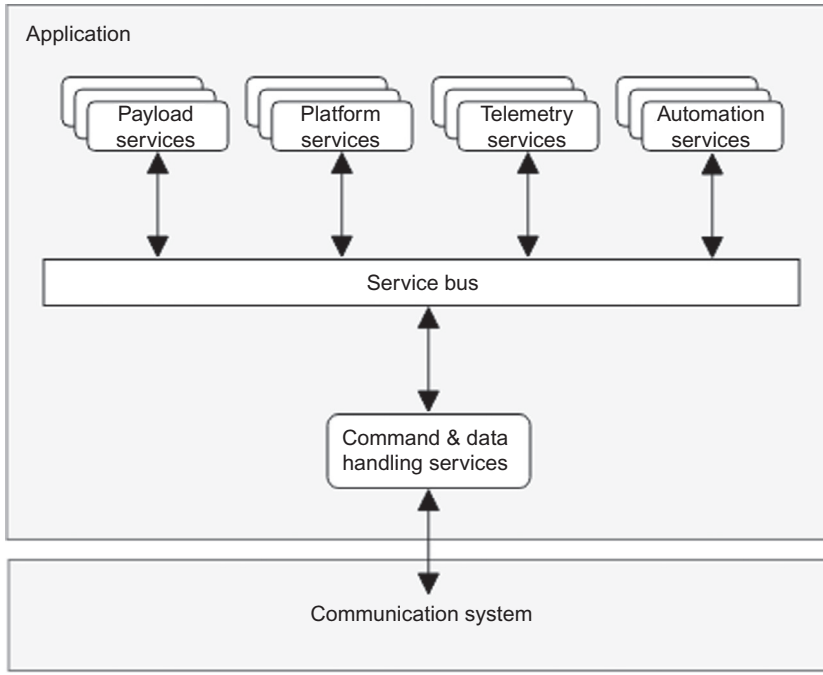

Fig. 2 Application layer components. A typical overview of a service-based software architecture where a service bus is connected to the communication system.

architecture, the logic of each service can be organized in further sublayers, and the processing of input and output data on the other hand as a pipe-filter pattern.

A good architecture design is one that allows mission designers to respond to current and future software needs. Therefore, sometimes a mixed approach might be necessary.

In the following, certain application components are described in detail.

Some of the following components are inspired by the ECSS-E-ST-70-41C standard [10], which defines a set of services that satisfy all fundamental operational requirements for spacecraft monitoring and control. In this standard, a service is defined as a functional element of the space system that provides a number of closely related functions that can be remotely operated. All these services use features that are provided by CSL, OSAL, and HAL components.

3.1 Command and data handling services

The command and data handling services have the responsibility of managing the communication channels and acting as a gateway between the ground segment and the OBSW. This involves routing of received telecommands to the corresponding service component through the service bus and deciding whether a specific telemetry is forwarded to the ground segment or stored onboard in nonvolatile memory.

3.2 Telemetry services

The telemetry services are a set of components whose objective is to provide information about the system. They include housekeeping reporting, event logging, parameter statistics, and onboard monitoring.

3.3 Automation services

The automation services are a set of providers that allows to operate the satellite using preloaded commands or preprogrammed actions that can be executed autonomously. Time-based scheduling, request sequencing, position-based scheduling, event-action, FDIR, and operational modes are some examples of automation components.

3.4 Payload services

The payload services are a set of components that provide operational control over mission payloads. These type of components are tightly coupled with the mission and the payloads to be managed. It is very important for the design of these services to dispose of detailed specifications of all interfaces, their capabilities, and restrictions respective to all payloads.

3.5 Platform services

The platform services are services that allow the system to manage different instruments of the platform such as RTC, global positioning system, AOCS, or special memories. These services are highly dependent on the available hardware components and their capabilities.

3.6 Service bus

The service bus is a component that connects the other software services. It is responsible for the communication among the services. All internal messages that are produced in the application are routed to the corresponding component via the service bus.

4 Software development process

The main goal of a satellite software development program is to produce a high-quality product considering all project constraints, such as time, cost, resources, and scope. The definition of software quality, according to ISO/IEC 25010 [11], is the degree to which the system satisfies the explicitly stated or further implied needs of its stakeholders, and thus provides value. These needs are as follows:

Functional suitability represents the degree to which a software provides functions that meet specifications.

Performance efficiency represents the performance of the product relative to the amount of resources it uses.
Compatibility represents the degree to which a software can coexist and exchange information with other systems.
Usability represents the degree to which a software can be used by users to achieve specified goals with effectiveness, efficiency, and satisfaction.
Reliability represents the degree to which a software performs specified functions under specified conditions, especially actions executed on hardware or software faults, interruptions, etc.
Security represents the degree to which a software protects information.
Maintainability represents the degree of effectiveness and efficiency with which a software can be modified in order to improve it, to correct it, or to adapt it to other requirements.
Portability represents the degree of effectiveness and efficiency with which a software can be transferred from one hardware or software environment to another.

The main way to achieve a high degree of compliance of these needs consists in a well-organized and implemented software development process. There are two important issues to be considered when defining the process: first, the software engineers should be present early in technical meetings of the development team, so they can provide a promptly feedback on software-related risks; second, one should avoid a shift of responsibilities to the application software, because such a decision might increase the complexity of the software to be developed to an undesired level.

Depending on the nature of the CubeSat project, two different approaches can be applied during the software development process [1,12]:

Predictive methodologies focus on analyzing and planning the future.
Adaptive methodologies focus on adapting quickly to changing requirements or needs.

The predictive methodologies, usually based on waterfall models, are driven by a plan. The plan predicts what will happen in the following phases. These methodologies are useful when requirements are well known beforehand and risks caused by changes can be controlled sufficiently. An application of these methodologies is recommended for large projects, because the generated documentation allows to manage the project and its coordination within different teams of developers.

The V-model is an example of a predictive methodology. It usually consists of seven phases: requirement analysis, architecture design, component design, coding, unit testing, integration testing, and acceptance testing. The first three phases are merely design stages where the system is increasingly refined. The last three phases are testing stages that validate the results of the design stage. The coding phase is the connection between these two groups. This methodology focuses on the software verification and validation.

The adaptive methodologies based on prototyping models are driven by the value of the product. This means that the main objective is to satisfy needs of stakeholders quickly with the goal of minimizing risks as soon as possible. These methodologies are useful when requirements are not completely known in advance. The process of developing functional prototypes in an iterative and incremental way allows developers to discover contingencies and solve problems in early stages. The application

of these methodologies is recommended for small/medium projects, because the creation of prototypes requires a high interaction of all members of a colocated team.

Agile methodologies are an example of adaptive methodologies [13]. In particular, Scrum, a popular agile software development framework, defines an iterative and incremental framework for the management of an agile development. Scrum establishes responsibilities for each team member, a collection of artifacts to be employed, and a set of meetings with all involved team members including the final customer, if needed, to guide the overall development process. Such a process is organized into short iterations whose final results are a valuable product. Scrum is inspired by the agile manifesto, hence, it is a flexible, lightweight, and highly disciplined process.

In space projects, predictive methodologies are the most common approach, mainly due to the size and the critical degree of such projects. Besides, the cost of a requirement change might force an abort of the mission.

In the case of CubeSat projects, however, the agile adaptive methodologies are acquiring a more and more relevant role, because they allow to reduce the development time and costs due to the removal of certain overhead that is produced by formal communications and rigid structures of the documentation to be generated. The agile philosophy normally fits targets of CubeSat projects in a better way than a more heavyweight predictive methodology.

Independently of the selected methodology, software verification and validation must be taken seriously. Here, verification means the process of evaluating work-products (not the actual final product) as the outcome of a development phase to determine whether they meet the specified requirements for that phase, i.e., whether the product is built right; and validation means the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements, i.e., whether it is the right product built. It is possible that, according to the specific size of the overall project, part of the verification and/or validation could be performed externally to the development team [14].

In the following sections, the main activities of the software development process are described [5].

4.1 Planning

Planning is the process of identifying the required actions to achieve a specific objective. Some of these actions to be identified could be

- setting goals,
- definition of responsibilities,
- viability studies,
- evaluation of risks,
- selection of development tools and technologies,
- scheduling of development tasks,
- adaptations and reuses from other projects,
- definition of the documentation to be generated.

It is important that the scheduling of actions that result from these activities is consistent with the due date of the tasks from the hardware and system teams in order to avoid dependencies that might block the ongoing software development or testing. As shown in the responsibilities of the following section, the OBSW is strongly coupled to the hardware. Therefore, it is critical to agree, at least, on the delivery of adequate system models on-time.

4.2 Analysis

Analysis is the process of understanding and defining services that are required from the system. All imposed constraints must be identified. This activity is vital to guarantee the correctness of the design and the implementation activities.

In space missions, software requirement analysis is usually derived from the mission and system analysis. The main result of the analysis is the definition, specification, and prioritization of functional and nonfunctional requirements. Each requirement should be unambiguous, testable, and traceable.

The concept of operations, use cases, system interfaces, and hazard analysis are useful for detailing the requirements.

4.3 Design

The software design is a description of the structure of the software to be implemented. Besides, it defines all interfaces between the components, the data models and, sometimes, the specific algorithms to be used. The requirements specification, the spacecraft system design, and the data description are required inputs for performing the software design in its architectural, interface, and component parts.

There are two typical iterations in the design process: preliminary design review (PDR) and critical design review (CDR). In the PDR, the design of the architecture and the communication flow should be presented. Moreover, the telemetry and telecommand format should be defined, at least at a high level. The CDR should include all elements of the PDR and the originated changes since its last review. Further, the individual design of the components, the detailed interface description, and a test plan should be presented.

To express the design concepts UML (unified model language) [15] is usually employed. UML is a mostly graphical language whose objective is to standardize the notation of a software design whenever possible.

4.4 Implementation

The implementation is the art of converting a design with its specifications into an executable application [5]. A large set of programming languages exist, which can be used for the final coding. The selection of a programming language should be motivated by one of the following reasons:

- portability,
- abstraction,

- determinism,
- efficiency,
- simplicity.

Assembly, C, C++, Ada, and Rust are typical examples of languages that are used in space projects. It is not unusual for different parts of the system to be coded in different languages.

On the other hand, automatic code generation is a technique that allows developers to produce executable code from a processable specification. Automatic code generation permits to perform high-level changes (in the specification) quite quickly. The advantage is that changes and modification can be adapted fast and with less probability of error. The disadvantage is that the code generator can be very complex and hard to be tested.

In this stage of the project, it should be noted that traceability is a relevant aspect to be considered, because tracing the fulfillment of requirements provides quality to the overall process. Further it is important to manage configurations and to control different versions of the software properly and efficiently in order to be able to locate possible defects—for instance, via binary search strategy over a fine-grained revision history—and perform the improvements consistently.

Last but not least, satellite flight systems software should be developed taking into account available techniques, good practices to guarantee reliability and portability. The use of adequate coding standards, programming philosophies, and interface descriptions allow to reach high-quality code. Moreover, there are tools that can be employed for a static and dynamic analysis of the developed software that help to detect defects, visualize memory usage, analyze deadline matching, profile the code coverage, and estimate power consumption.

4.5 Testing

The objective of testing is to show that a software system does what it is intended to do. It allows to discover program defects and inefficiencies. The main goals are as follows:

- Demonstrate to the developer and the customer that the software meets its requirements.
- Discover situations in which the behavior of the software is incorrect or does not meet some specification.

Such testing is a vital task for space projects, because, due to the existing dependencies within the overall space project, the possibility of fixing problems in production is very limited and the costs produced by a software failure could jeopardize the entire mission. To verify and validate software it is common to perform different levels of testing.

Unit test : This type of test aims at checking a specific case of a specific functionality.
Integration test : This type of test aims at checking the communication and joint functionality between different components.
Acceptance test : This type of test aims at checking high-level requirements.

Note that the ease with which software can be tested depends mainly on the design primarily made. Hence, test plans should be made part of the requirements specification for the software design. Automating the test procedures and repeating the tests when changes have been applied to the software help increase reliability.

Last but not least, simulations play an important role in the test stage of a space mission. They allow mission developers to discover risks, exploit alternatives, reduce uncertainty in early phases, and provide additional validation input and output. Further, simulation of components in software are useful to test failure cases that are very complicated or just impossible to produce with real elements. For example, electronic components happen to work well and cannot be asked to produce a certain fault repeatedly for the purpose of testing.

There are two different main levels of simulation:

- Simulation of hardware systems consists in replicating the behavior of a device or part of it. To do so, driver interfaces or HALs are used to mimic the functionality that is provided by the hardware.
- Simulation of software systems consists in replicating the behavior of a software component or an entire subsystem. The use of mocks, dummy components, and stubs are common techniques that are employed in this type of simulations.

To develop and perform simulations the interface definition is crucial. The closer the simulations models the real behavior, especially regarding time requirements, the better the testing can be performed. All thinkable scenarios should be dealt with, especially when human interactions are part of the overall system. However, simulations can never replace test-like-you-fly validation.

5 Mission experiences

In this section four CubeSat missions, namely Xatcobeo, HumSAT, Serpens [16–18], and Lume-1 missions, are briefly commented on from the point of view of the software engineering.

Xatcobeo (2012) was a technology demonstration mission in which a software-defined reconfigurable radio, a panel deployment mechanism, and a system for measuring the amount of ionizing radiation were verified in space. HumSAT (2013) and Serpens (2015) were space missions whose objective was to provide a messaging service to areas without infrastructure through low-cost terminals on the basis of the store-and-forward concept.

Lume-1 (2018) was a space mission whose objective was to receive wildfire alerts from terrestrial sensors and to provide information to unmanned air vehicles (UAVs) and ground segment elements as an additional support to the fire extinguishing procedure.

In the case of the Xatcobeo, HumSAT, and Serpens satellites, the challenge of the OBSW was to meet the mission requirements employing a very limited hardware. The main software requirements were as follows:

- Antenna deployment.
- Managing payloads via IIC-bus.

- Managing subsystems (TTC, EPS, and RTC) via IIC-bus.
- Managing telemetry of subsystems and payloads.
- Executing time-scheduled tasks.

The principal hardware characteristics of the OBC were as follows:

- An FPGA implementing a soft 32-bit microblaze microprocessor.
- 1 MByte of static RAM for code and data (volatile).
- 512 MByte of NAND-based flash memory (nonvolatile).

For these reasons, the employed software architecture was a simple layered architecture on a tailored operating system based on a finite-state machine. The operating system was developed without employing explicit concurrency and without dynamic memory management (besides the program stack). A file system was not used, so a custom NAND flash memory manager had to be developed. These decisions had been taken due to the lack of nonvolatile RAM; 1 MByte RAM was not enough to store the program and data, the file system footprint, and possible thread contexts.

In the case of Lume-1, the main purpose of the software development was to build a generic software platform where specific mission services can be integrated with a set of common services for any mission. Thus, the main challenge was to develop a generic software infrastructure that could be employed by other platforms in other missions with different payloads and subsystems. In this mission, the following hardware features were available:

- AVR-32 processor with 32 MHz clock frequency,
- 512 KBytes of build-in flash for code and data (nonvolatile),
- 32 MBytes of RAM (volatile),
- 32 KBytes of FRAM (nonvolatile), and
- 128 MBytes of NOR-based flash memory (nonvolatile).

With these capabilities and requirements, a distributed architecture was developed using an existing real-time operating system. The operating system provides a uniform interface to interact with several microcontrollers. In addition, it allows designers to use and manage concurrency. Additionally, an existing file system was adapted to handle persistent data. These two software components allowed the software development team to build a generic and extensible architecture where a great part of command and data handling services, as described earlier, were implemented.

According to our experience, two different approaches were described earlier. Both have their pros and cons. In the first three cases, every piece of software was developed, managed, tested, and debugged by the software team, so the software was well known and could be ported quite easily to similar low-performance platforms. The software was simulated completely in a linux development environment. In the case of Lume-1, there were pieces of code that had to be trusted and the minimum hardware requirements should be considered in future missions. On the other hand, Lume-1 uses high-level software concepts, so its scalability is much better and the degree of functionality is improved considerably.

However, in both cases, there was a set of requirements that was covered considering the specific project constraints. Thus, it is very important to know and analyze

the scope, the available resources, and the available time, because these three elements will condition the design, the implementation, and the test stages.

Finally, this section concludes with the main lessons learned by the software team in these CubeSat projects:

- The usage of standards makes the integration easier and reduces problems.
- Errors are found faster thanks to a rigorous development process.
- The usage of third-party software implies more hours of integration, testing, and debugging.
- There are never enough tests.
- Simple solutions are the best.
- Automatic code generation improves productivity and reduces human errors as long as it is combined with automatic testing.
- Continuous integration is a development practice that prevents many problems.

6 Conclusion

In this chapter, the most important aspects of onboard satellite software development have been treated. One thing that the reader may have noted is the similarity between the OBSW development for a CubeSat and the development of any other software system. The reason for this similarity is quite simple: the main goal of any software development is to meet all requirements in the best possible way.

The compliance with requirements in aerospace is not more important than that in other sectors but, most of the time, the cost of errors or the nonfulfillment of requirements is much greater in space projects than in any other field of engineering. For that reason flight software is normally developed with a very high degree of perfectionism. The perfectionism is achieved by implementing a reliable development process in which each stage is performed employing standards, good practices and techniques, quality management tools, etc.

With the arrival of CubeSats, mission costs are going down. The miniaturization of technology allows developers to reduce costs and improve the performance of missions. However, this sometimes produces a decrease in quality and especially in software quality. Nowadays, a CubeSat has the same capabilities and performance as bigger satellites from 30 years ago. This means that a CubeSat might have the same software requirements as a traditional conventional satellite. Thus, a CubeSat should satisfy the same exigency and quality levels as other satellites from the point of view of software engineering. The fact that the software flies onboard a less costly platform does not mean that the software and its development become less costly as well. Of course, at present there are many more technologies and possibilities than in the past and the usage of general purpose hardware allows mission developers to reduce software costs to some extent, but it should never be at the expense of quality.

References

[1] M. Macdonald, V. Badescu, The International Handbook of Space Technology, Springer Verlag, 2014. ISBN 978-3-642-41100-7.

[2] Space engineering: software, ESA-ESTEC, Requirements & Standards Division, 2009. Tech. Rep. ECSS-E-ST-40C.
[3] Space engineering: software engineering handbook, ESA-ESTEC, Requirements & Standards Division, 2013. Tech. Rep. ECSS-E-HB-40A.
[4] J. Eickhoff, Onboard Computers, Onboard Software and Satellite Operations. Springer Verlag, 2012, https://doi.org/10.1007/978-3-642-25170-2.
[5] I. Sommerville, Software Engineering, Pearson Education, 2011. ISBN 978-0-13-703515-1.
[6] The FreeRTOS kernel, (2019), http://www.freertos.org.
[7] VxWorks, (2019), https://www.windriver.com/products/vxworks/.
[8] RTEMS Real Time Operating System (RTOS), (2019), https://www.rtems.org/.
[9] Information technology—open systems interconnection—basic reference model: the basic model, International Organization for Standardization, 2011. Tech. Rep. ISO/IEC 15504.
[10] Space engineering: ground systems and operations, ESA-ESTEC, Requirements & Standards Division, 2008. Tech. Rep. ECSS-E-ST-70C.
[11] Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models, International Organization for Standardization, 2011. Tech. Rep. ISO/IEC 25010:2011.
[12] NASA software safety guidebook, National Aeronautics and Space Administration, 2004. Tech. Rep. NASA-GB-8719.13.
[13] Space engineering: agile software development handbook, ESA-ESTEC, Requirements & Standards Division, Tech. Rep. ECSS-E-HB-40-01A, 2020.
[14] ESA guide for independent software verification and validation, ESA-ESTEC, 2005. Tech. Rep. ESA ISVV Guide.
[15] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, D. Tolbert, Unified Modeling Language (UML) Version 2.5.1, Tech. Rep., Object Management Group (OMG), 2017. https://www.omg.org/spec/UML/2.5.1.
[16] A. Castro, R. Walker, F. Emma, F. Aguado, R. Tubio, W. Balogh, Hands-on experience—the HumSAT system and the ESA GEOID initiative, ESA Bull. 149 (2012) 45–50.
[17] R. Tubío-Pardavila, S.A. Vigil, J. Puig-Suari, F. Aguado Agelet, The HUMSAT system: a CubeSat-based constellation for in-situ and inexpensive environmental measurements, in: AGU Fall Meeting Abstracts2014, p. A23I-3365.
[18] J.A.V. Vilán, F.A. Agelet, M.L. Estévez, A.G. Muino, Flight results: reliability and lifetime of the polymeric 3D-printed antenna deployment mechanism installed on Xatcobeo & Humsat-D, Acta Astronaut. 107 (2015) 290–300.