Deduced Software Design Principles from Experiences with the Technical Debt in Reused Software

Olaf Maibaum[1], Thomas Terzibaschian[2], Christian Raschke[3], Andreas Gerndt[1]

[1] German Aerospace Center (DLR), Simulation and Software Technology
Lilienthalplatz 7, 38108 Braunschweig, Germany
Phone: +49 531 295 2974, Mail: olaf.maibaum@dlr.de

[2] German Aerospace Center (DLR), Institute of Optical Sensor Systems
Rutherford-Str. 2, 12489 Berlin, Germany
Phone: +49 30 67055 586, Mail: thomas.terzibaschian@dlr.de

[3] Astro- und Feinwerktechnik Adlershof GmbH
Albert-Einstein-Straße 12, 12489 Berlin, Germany
Phone: +49 30 6392 1053, Mail: c.raschke@astrofein.com

Abstract: Software reuse conserves software design decisions for the future, but technology evolves over time. Previous design decisions thereby become technical debt in designs for future projects in the case of software reuse. This may result in risk and cost increases for future projects. The small satellite BIROS is an example for such a project and is reflected in this paper. The software reuse prohibits the utilization of state-of-the-art test techniques without significant modification to its software architecture, which is not a software reuse. The necessary changes in the software architecture are presented in this paper and how it protects against faults that arose during the system test and commissioning phases.

1. INTRODUCTION

The AOC subsystem of the BIROS satellite [1] has a long history. The first lines of C++-code in the software were written in the year 2000 for the BIRD ACS. In 2008, the BIRD ACS was reused in the TET-1 satellite. For TET-1, the setup process of the software was restructured, the hardware related drivers on the IO level of the software were replaced, and new control modes were introduced. These modifications were necessary due to changes in the sensor and actuator hardware as well as lessons learned from BIRD. The BIROS AOCS software, reused from TET-1 [2], has been extended for several experiments such as the qualification of a thruster system, fast slew maneuvers with high torque wheels, the AVANTI experiment for the autonomous formation flight with the BEESAT-4 as well as an optical downlink.

In the year 2000, the development team's knowledge and the state of test techniques led to design decisions in the software architecture which are not up to par with state-of-the-art techniques used by software development today. Due to software reuse and an unchanged software architecture, the design decisions from the BIRD software have been conserved. This complicates adding new functionality and the discovery of software faults in the space-proven application software. Testing with a continuous integration test approach is not possible. This disadvantage is a technical debt that originates from the original BIRD software.

Unit tests are an essential part of the continuous integration test process. In the year 2000 such software development processes were not state-of-the-art, e.g., the development of the CppUnit framework as a unit test framework for C++ started in the year 2000 [6]. The accepted test approach was based on functional tests, which were done for

BIRD with the controller modes. Therefore, the chosen software architecture for BIRD did not support unit tests with current unit test frameworks. One positive aspect in the software architecture is the coupling of software components through a component manager, which allows the replacement of software components. Unfortunately, the interface design did not allow any replacement of the software components with test stubs. The serial bus interfaces were designed to be replaceable by stubs, but all these interfaces are fixed members of the software components. These software design decisions make unit testing impossible because the software components cannot be tested in isolation from one another.

At first, we present the software changes made in BIROS, followed by a selection of arising issues due to the software changes during the system test and commissioning phases. The paper continues with the presentation of the software architectures used in the TET satellite line and the new one used for currently active projects at the department such as autonomous optical navigation (ATON) [8], cold gas experiments (MAIUS, QUANTUS) [9] and DLR's compact satellite (Eu:CROPIS) [10, 11]. The paper closes with a comparison of the test processes applied in both software architectures.

2. SOFTWARE CHANGES IN BIROS

Besides minor software changes in order to increase software robustness in case of malfunctions, major software changes were made due to the requirements of additional experiments on the BIROS satellite. The minor software changes will also be applied to TET-1 after they have been successful proven in space on the BIROS satellite.

The major software changes are:

- a control interface to the nitrogen thruster system for orbit maneuvers
- a 4-quadrant-sensor interface to orient an optical communication system to the laser buoy of the optical ground station (PrOSIRIS) [3],
- extension of the wheel system with three high torque wheels for high agility attitude control [4],
- high speed image reading from star tracker hardware for the optical tracking of a remote target and read out of further information from the star tracker,
- interface for AVANTI [5] to request star tracker data, to access and control the thruster system, and command attitude modes in the AOCS,
- and extending the attitude control state machine with additional control modes. These consist of the target pointing mode for the optical downlink, fast slew maneuver mode for high agility attitude control, thruster firing to perform orbit maneuvers, and client observation for optical observation of the remote target with the star tracker system.

The main challenge for the software changes is that the behavior of TET-1 has to remain unchanged. In case of new control modes, the work is straight forward because each mode is designed as its own class and only the mode transitions have to be added to the existing modes. In this case, only a test of the mode transitions is necessary to verify the software changes. Similarly, the new control interface to the nitrogen thruster only adds new classes to the software in addition to integrating the data into the state

vector of the AOCS. This has no influence on the existing software except on the computational load of the bus computer.

The 4-quadrant-sensor is an analog sensor system which shares the interface to the analog digital conversion with the sun sensors. For this, a redesign of the interface was necessary to fulfil different timing requirements of the analog digital conversion scan for the both sensor types.

The new high torque wheels share the communication bus with the four control wheels in the TET-1 design. So, the communication with the high torque wheels should integrate into the same interface to the communication bus without negatively affecting the four wheel behavior. These changes affect the communication process and computation of the wheel subsystem's control torque. Furthermore, a new control strategy for a seven wheel configuration is necessary.

The image reading for the AVANTI experiment had a significant impact on the existing processing of the star tracker. The existing image read process to read out an image from the star tracker was too slow to fulfill AVANTI's requirements. Increasing the size of the communication buffer to read a large data chunk in one read request was not an option due to memory issues. Therefore, the period of the main star tracker control loop had to be increased without changing the timing of the star tracker control state machine. The resolution was to split the main processing cycles into ten sub-cycles and perform the normal processing of the state machine in two of the sub-cycles. The read out of the image data was performed in the remaining eight cycles outside the state machine as a parallel process.

3. ARISING ISSUES

The software changes from TET-1 to BIROS uncovered several software issues hidden within the space proven software.

One issue was a resulting controlled spin when the AOCS lost information on the inertial orientation in one of the inertial modes and fell back into the auto acquisition mode to acquire a sun orientation until it have a valid inertial vector. This fallback behavior was introduced for BIROS and is overtaken to TET-1. The issue was tracked down to be a side-effect of a modification in the tele-command handler for the inertial mode of TET-1 which serializes unused parameter data as a target for the mode control. This did not have an effect on TET-1's operation, but in BIROS, changed its fallback behavior in that it would start to spin according to the serialized invalid parameter data. This behavior was first observed in the commissioning phase when the fallback was activated. During system testing, this behavior was not observed, possibly due to command sequences with zero data in the corresponding serialized data area. Another reason could be the differences between the ground support equipment in the laboratory and the processing of tele-commands in the ground control center, e.g., null initialization of unused parameters in the fix sized tele-commands.

Using a star tracker as an optical device is a complicated and error prone undertaking. Timing issues arise in the interface when image reads are performed in parallel during normal operation. Several times, a loss of messages was observed during system tests, which seldom also occur during normal image read operation of TET-1. Due to this, the

state machine of the star tracker interface was occasionally stuck in a mode. Unfortunately, the behavior of the star tracker hardware differs from the engineering model in the laboratory environment to that of the flight model in the space environment. Therefore, most of the critical issues only appeared in the space environment and could not be reproduced in a laboratory environment. An example of this was the fixes for the potential source of messages losses in BIRD's software, where they would work in the engineering model but not in the flight model located in a space environment. During the commissioning phase, a software update, successfully tested in the laboratory environment, broke the communication with the star tracker in the flight model in space.

The AOCS records important events such as mode changes in the control mode state machine. The AVANTI experiment adjusts the target of the control mode every 30 seconds which causes a reentry into the control mode. Since the interface between AVANTI and the AOCS use the same mechanism as tele-commands, the log was flooded by control mode events and it becomes difficult to find important events in the log. This needed a new software requirement that such activities were not logged anymore. Thus, a filter was introduced into the event recording mechanism, based on the source of the mode commands, which distinguishes between tele-command, AOCS FDIR activity and the AVANTI experiment as possible sources of commands.

4. COMPARISION OF SOFTWARE ARCHITECTURES

The software architecture has a big impact on the testability and maintainability of software systems. The design decisions of the AOCS software architecture of the TET line are based on the reused software from the BIRD project. BIRD's architecture decisions were significantly influenced by predefined systems such as its operating system and the application framework. Besides this, a major factor was the reusability of the software for future space missions. Testability was not recognized as a valid driver for software quality. Therefore, software verification is limited to manually testing the controller functions while driver interfaces have to be tested in a HIL environment. In current projects, the lessons learned from BIRD as well as test automation requirements influence software design decisions. The following two subsections show the different concepts of software architectures and close with a comparison of both software architectures.

4.1 Reused Architecture in TET Satellite Line

The TET satellite line uses a component based design. Interactions between the software components are handled by a component manager. It uses a key-value map to return a pointer to the requested software component. Figure 1 shows the basic design of the component-oriented architecture and the process required to setup and access other components. This enables the software components to be replaced with stubs when the stub is registered with the component manager with the key of the stubbed software component. What was not done in the TET line was to define the interface routines of software components as virtual methods in order to allow them to be stubbed.

The organization of software components in the TET satellite line uses a tree structure. The implemented interface owns the bus interface. For example, the wheel system owns the wheels and the bus interface to the serial bus. This architecture type is shown in figure 2.

Components in the TET line use threads with endless loops in the thread body. The timing of threads is controlled by calling time control methods in the endless loops.

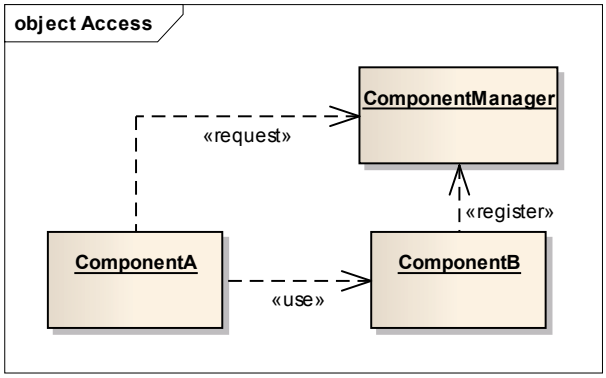

Figure 1 Component design in TET satellite line

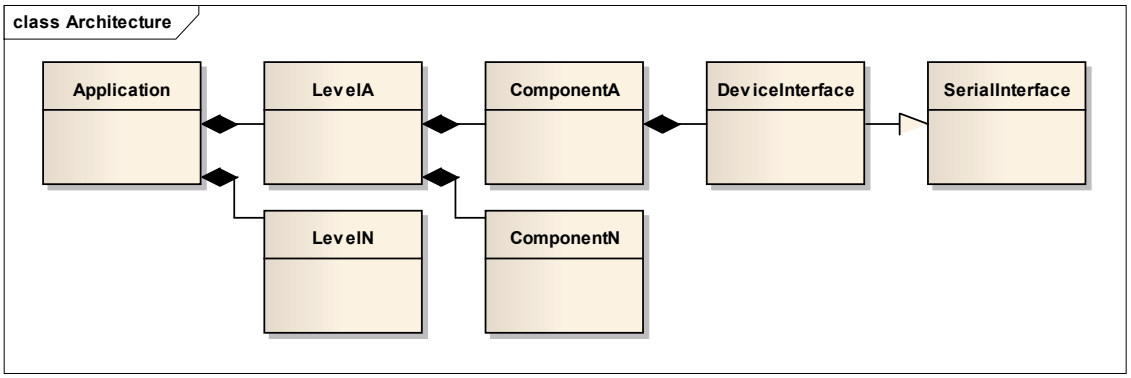

Figure 2 Architecture of an application

4.2 New Architecture in Current and Future Projects

In current projects, a data and event flow oriented view on the system is used. For example, a thermal application has to capture, in a defined time frame, the thermal state from thermal sensors and provide this information as the thermal state. In case of a thermal deviation, this deviation is reported which triggers the process of bringing the system back into an acceptable thermal state, e.g., by request a matching attitude mode on the attitude target channel. This channel is also used to control the target attitude when a tele-command is received on the ACS tele-command channel by the uplink distribution which is activated by a tele-command package on the uplink channel. Figure 3 show the system tasks and data channels for the example above.

For the software architecture, there exists the restriction that computation tasks are stateless. Data in the system is only held in channels. The purpose of this requirement is that all data is accessible as a housekeeping data item and to simplify a reconfiguration in a distributed reliable onboard system, e.g, the OBC-NG (On-board Computer – Next Generation) system developed at DLR [7].

The term “software component” is used in the software architecture only to group the tasks and related channels to a purpose, e.g., an interface component with all computa-

tional tasks according to a hardware device. This will simplify the setup of the software system. Interfaces to devices are no longer the child of the device driver implementation. They are provided by overloading a pure abstract class and handed over as a reference to the implementation of a device driver.

The execution of the computational tasks inside a software system is managed by the Tasking framework developed from the lessons learned in BIRD. It connects the computational tasks with the data channels. The execution of a computational task is scheduled by the availability of data at the incoming channels. Timing is provided through the connection of an event channel which can be configured as a periodical timer event or triggered with a timing delay from another task. Tasks have the restriction that they are non-blocking and terminate after a specified time. Endless loops inside a task are forbidden.

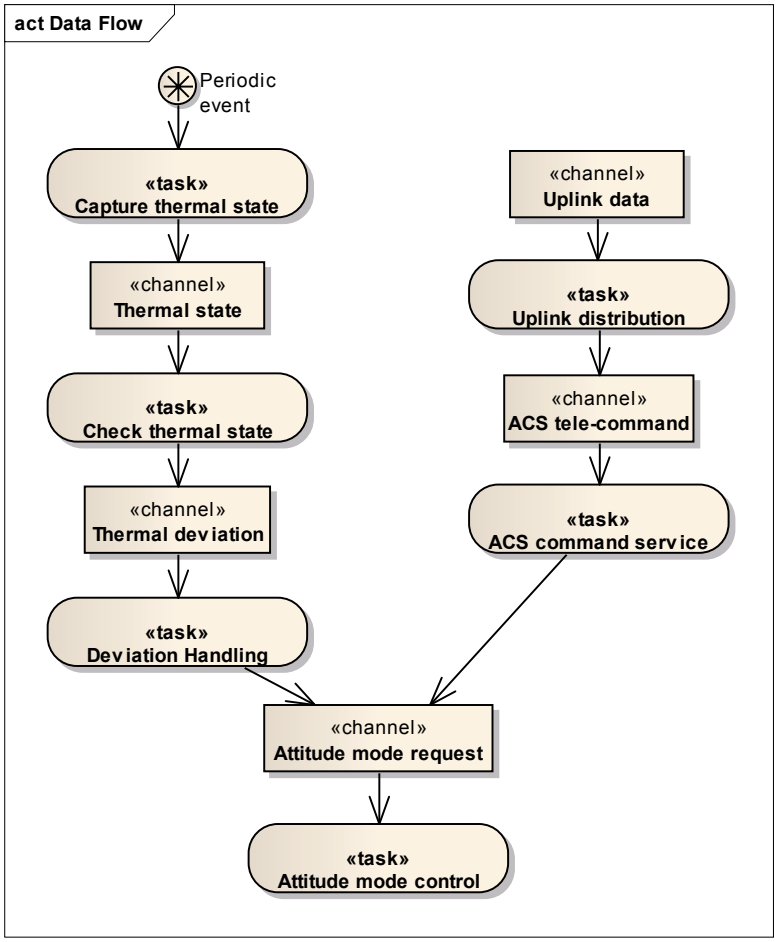

Figure 3 Example of tasks on data flow

4.3 Test Processes for both Architectures

The software tests for both architectures differ. All software tests in the TET line are restricted to manual system tests with real hardware. The only automatization during tests is the replay of tele-command sequences at the EGSE, for example to set up the system under test. Newer projects follow the test first strategy. Here, a unit test is developed before the software functionality is implemented. The precondition for a check-

in of changes to the software repository is the successful execution of all unit tests. Besides the manual start of the unit tests, a Jenkins server performs the unit tests as a nightly build or triggered by a check-in in the SVN repository or in Gitlab. E-Mail reports or wall displays inform on the current state of the software under development.

As mentioned before, the software test on the TET line is currently restricted to manual tests due to the technical debt incurred from the reuse of BIRD's software. For some software parts, functional tests are performed which can also be adapted to state-of-the-art unit test frameworks like Gtest or CppUnit. Compared to these, the manual tests with the HIL environment are time consuming. Debugging is only possible with output statements in the software. Such an approach is messy when the faults are caused by a logical failure which can be detected with unit tests and debugging tools. In the end, e.g., the first tests on the communication with hardware devices take several weeks in the laboratory until everything works as expected.

With the test first strategy development starts with an empty interface for the new functionality followed by the unit test to check the correct implementation of the functionality. In the next step, the implementation of the new functionality is developed until all unit tests can be executed without any failure. This causes the time required to complete the implementation is increased. To simplify the development of unit tests, several test support libraries are developed, e.g., a helper class for the used Tasking Framework with methods to control the scheduling of tasks and the tear up and down for test cases.

The unit tests contain three steps, testing operations on data items, testing functionality of tasks and at least testing functionality of a group of interacting tasks. The restriction to design interfaces only as channels simplifies the development of tests. For each test, only instances of the necessary input and output channels are needed, which are connected to the task(s) under test. If a task needs other instances, such as serial interfaces, a reference to the test stub is handed to the constructor instead of the real serial interface instance as is the case in the final implementation of system.

At mentioned earlier, the test first approach increases the development time, but most logical errors are discovered by the unit tests. The opportunity to debug the software and test it in the development environment eases the bug fixing of these errors. In the end, the first communication with the hardware devices in the laboratory is significantly sped up. As an example, for the ACS sensors of the Eu:Cropis satellite this step was accomplished within two days instead of several weeks as in TET or BIROS. The observed bugs in this step are related to integration issues and missing documentation of the byte ordering of a hardware device. All these errors could be fixed during the two days of test. Only one timing issue was still open after these tests, which was detected in the next days as a running condition during the startup of the software and a wrong clock configuration in the board supply package. During further development steps the automatic execution of unit tests by a Jenkins server will protect the existing software functionality against side-effects of bug fixes or changes from newly implemented functionality.

5. CONCLUSIONS

The paper shows that the investment of development time for unit tests decrease the time needed for tests in the HIL test environment. The software will be in a near bug

free state and eliminates the work pressure for the development team when the software is tested in the real hardware environment. For maintenance or extension of functionality, the automated unit tests avoid new bugs being seeded into the tested software. Not all causes of issues will be discovered by unit tests. The resulting spin of the wrong command modification will only be detected when a unit tests is formulated with the full command pipeline, which is typical not handled by unit tests. In the case of modifications to the star tracker interface, unit tests can detect side-effects of modifications that affect existing functionality. To check a possible source of message loss, a special test case can be formulated to examine it. Given that the formulated test case detects an error, the test case should verify that the fix alleviates the error. If the unexpected behavior remains, the star tracker hardware, not covered by unit tests, can be determine as the source of the error. The fault of the log being flooded was due to missing requirements, not software errors. Nonetheless, test cases are useful when implementing a filter inside the existing mechanism.

In the end, the BIROS AOCS worked as expect after all remaining issues where fixed. The AVANTI experiment was successfully performed and we expect that the next experiments and the operation of the FIREBIRD mission will also be successful. In the long term, it is necessary to bring the space proven software from the BIRD and TET-1 satellites, step by step, into a more maintainable and adaptable state by increasing their testability. During normal operation, the software and hardware perform without issues. Nonetheless, there will be always this one code line in the 30.000 which will not work as expected in the next software reuse. Only a good test approach is able to minimize the chance that this code has no negative effect.

6. REFERENCES

[1] H. Reile, E. Lorenz, and T. Terzibaschian. The FireBird Mission - A Scientific Mission for Earth Observation and Hot SpotDetection. Digest of the 9th International Symposium of the International Academy of Astronautics, pp. 17-20 (2013)

[2] O. Maibaum, T. Terzibaschian, C. Raschke, and A. Gerndt. Software Reuse of the BIRD ACS for the TET Satellite Bus. In: Digest of the 8th International Symposium of the International Academy of Astronautics. pp. 409-412. Wissenschaft und Technik Verlag, Berlin (2011)

[3] C. Schmidt, M. Brechtelsbauer, F. Rein, C Fuchs OSIRIS Payload for DLR's BiROS Satellite. International Conference on Space Optical Systems and Applications, Kobe (2014)

[4] C. Raschke, T. Terzibaschian, W. Halle High Agility Demonstration with a New Actuator System by Small Satellite BIROS. 9th Airtec 2014, Frankfurt/Main (2014)

[5] G. Gaias, J.-S. Ardaens Design challenges and safety concept for the AVANTI experiment. Acta Astronautica, vol. 123, pp. 409-419 (2016)

[6] CppUnit – C++ port of JUnit. https://sourceforge.net/projects/cppunit/files/cppunit/ (04.2017)

[7] A Component-Based Middleware for a Reliable Distributed and Reconfigurable Spacecraft Onboard Computer. Peng Ting et.al. In Proceedings of the IEEE Symposium on Reliable Distributed System, pp. 337-342. Budapest (2016)

[8] Lidar-Aided Camera Feature Tracking and Visual SLAM for Spacecraft Low-Orbit Navigation and Planetary Landing. F. Andert, N. Amman, B. Maass. In Advances in Aerospace Guidance, Navigation and Control. Springer International Publishing. pp. 605-623 (2015).

[9] A compact and robust diode laser system for atom interferometry on a sounding rocket. V. Schkolnik et.al. In Applied Physics B (2016) 122:217. Springer-Verlag.

[10] Food Production in Space – Operating a Greenhouse in Low Earth Orbit. D. Schulz, C. Philpot, G. Morfil, B. Klein, and T. Beck. Space Ops 2016. Daejeon (2016)

[11] Attitude Control System of the Eu:CROPIS Mission. A. Heidecker, K. Takahiro, O. Maibaum, and M. Hölzel. 65th International Astronautical Congress, Toronto 29.09-03.10.2014. International Astronautical Federation (2014)