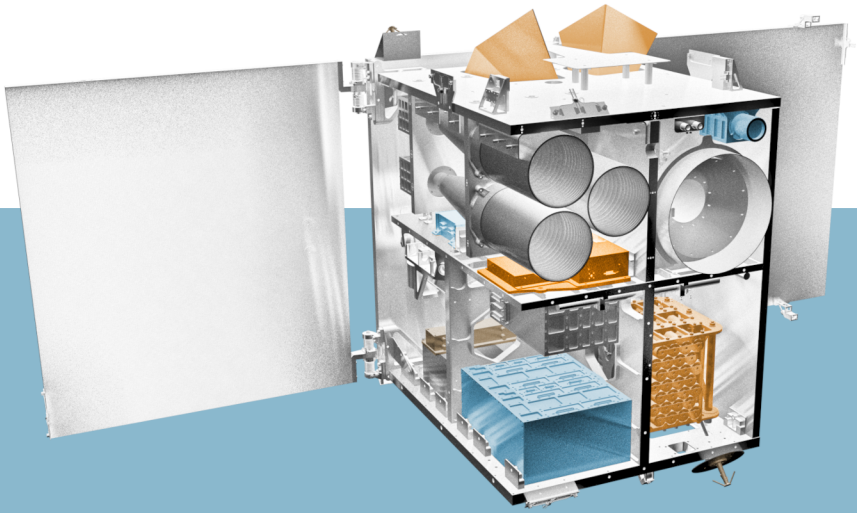

Bastian Bätz

Design and Implementation of a Framework for Spacecraft Flight Software

Design and Implementation of a Framework for Spacecraft Flight Software

A thesis accepted by the Faculty of Aerospace Engineering and Geodesy of the University of Stuttgart in partial fulfillment of the requirements for the degree of Doctor of Engineering Sciences (Dr.-Ing.)

by

Bastian Bätz

born in Karlsruhe

Main referee:	Hon.-Prof. Dr.-Ing. Jens Eickhoff
Co-referee:	Prof. Dr. Klaus Schilling
Co-referee:	Prof. Dr.-Ing. Sabine Klinkner
Date of defence:	9 January 2020

Institute of Space Systems
University of Stuttgart
2020

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de abrufbar.

bastianbaetz@hotmail.com

1. Auflage 2020

To Carolin, Jonathan, Simon, Samuel, my parents, and all the rest of my family.

Contents

Abstract vii

Kurzfassung ix

Preface xi

List of Acronyms xiii

1. Introduction 1
- 1.1. The Nature of Complexity . . . 3
- 1.2. Diagrams and Code Examples . . . 4

2. Software Engineering Tools and Methods 5
- 2.1. Object-Oriented Programming . . . 6
- 2.2. Development Methods . . . 9
- 2.3. Design and Implementation Techniques . . . 14
- 2.4. Embedded Software Development . . . 30

3. Flight Software Domain Analysis 39
- 3.1. Domain Definition . . . 40
- 3.2. Domain Modeling . . . 42
- 3.3. The *Flying Laptop* Mission . . . 43
- 3.4. CCSDS Standards . . . 58
- 3.5. ECSS Standards . . . 68
- 3.6. Existing Flight Software . . . 81
- 3.7. Synthesis . . . 90

4. The Flight Software Framework 99
- 4.1. The Flight Software Framework Architecture . . . 99
- 4.2. The *Flying Laptop* Software . . . 109
- 4.3. Common Interfaces . . . 121
- 4.4. The FSFW-Core . . . 131
- 4.5. Component Templates . . . 147
- 4.6. The FSFW PUS Framework . . . 169
- 4.7. Fault Management . . . 176

5. Evaluation 187
- 5.1. Developing a FSFW-Based Software . . . 187

5.2. Spacecraft Testing with the FSFW 189
5.3. Operating a Spacecraft with the FSFW 190
5.4. Software Reuse with the FSFW 192
5.5. Towards an FSFW-based Software Product Line 193
5.6. The FSFW as Real-Time Embedded Software 194
5.7. Improvements and Open Issues 195

6. Summary **201**

A. Example Code Listings **203**
A.1. Object-Oriented Programming Examples 203
A.2. `DeviceHandlerFailureIsolation::eventReceived` 206
A.3. Container Generation For A Black Box Framework 209

B. Domain Analysis Details **213**
B.1. Historical Domain Analysis Results 214
B.2. Feature Tracing Tables 215

C. FSFW Details **223**
C.1. *Flying Laptop* Mode Tree 223
C.2. More FSFW-Core Features 225

Bibliography **227**

Abstract

The amount of functionality provided by software in technical products is rising continuously, not only for programs in personal computers or mobile devices, but also for that embedded in other machines, such as household appliances, cars or spacecraft.

However, with a rising amount of functionality comes more code, which implicates, in most cases, more complexity. This is also true for spacecraft, which makes the development of spacecraft on-board software or flight software (FSW) a challenging endeavour.

This thesis sets the objective to improve design and development of FSW by applying techniques found in the discipline of software engineering. As most of these techniques do not aim at embedded software, but rather on general purpose software or Internet applications, existing techniques are surveyed for their applicability to the domain of flight software design.

As a result, this survey highlights the possibility to create a component framework, which supports the development of FSW for various variants of space missions. To determine the common needs of spacecraft software, a domain analysis is performed, which identifies requirements or features not for a single, specific satellite, but spacecraft software in general. These generic features cover component management, system management, operations, and autonomy.

With a set of tools at hands and the required features identified, a spacecraft software framework can be designed. This was done at the University of Stuttgart and the resulting framework is called the Flight Software Framework (FSFW). This thesis contributes to the FSFW and describes its overall design in the view of the identified software engineering techniques.

The FSFW architecture defines a FSW application as a set of interacting components, which offer and invoke functionality of other components. The functionality is defined by a small number of interface definitions, e. g. for component mode handling or action invocation. Establishing communication and executing components, as well as providing access to computing resources and hardware interfaces, is the task of the FSFW-Core, which is the central element of the framework.

To support the implementation of components, the FSFW offers a set of component templates, which serve as prototypes for e. g. device handling or controller

components. These templates are complemented by sub-frameworks to command the spacecraft using a common space link protocol, the packet utilization standard (PUS), as well as elements to implement on-board failure detection, isolation and recovery (FDIR).

To ensure the practicality of the complete component framework, the FSFW serves as basis for the software of *Flying Laptop*, a small satellite also developed, built, and currently operated by the University of Stuttgart. This allowed to iteratively find a good design, which proved itself useful in a real-world deployment. Also, it serves as an example to illustrate the various concepts and features of the FSFW within this thesis.

A brief evaluation shows that applying selected software engineering techniques can improve flight software development. This happens by enhancing separation of concerns by encapsulation of functionality in components. Also, the FSFW enables reuse of both the unchanged FSFW-Core and entire components in various space missions.

Kurzfassung

In technischen Produkten steigt der Anteil der Funktionalität, die von Software bereitgestellt wird, kontinuierlich an. Das ist nicht nur bei Programmen in Computern und Mobilgeräten der Fall, sondern auch bei Software, die in andere technische Geräte eingebettet ist, etwa in Haushaltsgeräte, in Automobile, oder eben in Raumfahrzeuge.

Allerdings erfordert das Mehr an Funktionalität auch ein Mehr an Quellcode, was meistens auch einen Zuwachs an Komplexität mit sich bringt. Die Software von Satelliten und anderen Raumfahrtsystemen ist von diesem Trend nicht ausgenommen, was die Entwicklung von Onboard-Software oder Flugsoftware zu einer herausfordernden Unternehmung macht.

Das Ziel dieser Arbeit ist es, zu überprüfen, durch welche Techniken aus der Disziplin des Software Engineerings der Entwurf und die Entwicklung von Flugsoftware verbessert werden kann. Da diese Techniken typischerweise nicht für eingebettete Software, sondern für Desktop- und Internetanwendungen entwickelt wurden, ist zunächst eine Überprüfung auf Eignung für Flugsoftware erforderlich.

Als Ergebnis zeigt diese Untersuchung die Möglichkeit auf, ein komponentenbasiertes Rahmenwerk für die allgemeine Entwicklung von Satellitensoftware zu entwickeln, ein sogenanntes Komponentenframework. Dafür ist eine Übersicht der Eigenschaften von und Anforderungen an Flugsoftware im Allgemeinen erforderlich, die im Verlauf dieser Arbeit erstellt wurde. Diese Domänenanalyse identifiziert Funktionalitäten in den Kategorien: Komponentenverwaltung, Systemverwaltung, Satellitenbetrieb und Autonomie.

Die gefundenen Entwicklungstechniken und Anforderungen ermöglichen den Entwurf eines Komponentenframeworks für Raumfahrtsysteme. Ein solches, das sogennante Flight Software Framework (FSFW), wurde an der Universität Stuttgart entwickelt. Die vorliegende Arbeit legt dafür den theoretischen Rahmen, und erläutert das Framework im Licht der genannten Softwaretechniken.

Die Architektur des FSFW definiert eine Flugsoftware als eine Kombination aus interagierenden Komponenten, die Funktionalitäten anbieten oder von anderen Komponenten nutzen. Diese Funktionalitäten werden von einer limitierten Zahl von Schnittstellendefinitionen festgelegt, etwa zur Zustandsverwaltung von Komponenten oder um Aktivitäten zu starten. Ein zentrales Element

des FSFW ist der sogenannte FSFW-Core, der die Interaktion zwischen Komponenten ermöglicht, deren Ausführung regelt, sowie den Zugriff auf Rechenressourcen und Hardwareschnittstellen gewährleistet.

Um die Implementierung von Komponenten zu unterstützen, bietet das FSFW Komponentenvorlagen an, die als Prototyp für spezifische Komponenten etwa zur Geräteverwaltung oder für Kontrollalgorithmen dienen können. Diese werden ergänzt durch zusätzliche Frameworks, um die Steuerung über ein typisches Kommunikationsprotokoll, dem Packet Utilization Standard (PUS), zu ermöglichen und um Funktionalitäten für die Fehlerbehandlung an Bord des Satelliten bereit zu stellen.

Um die praktische Nutzbarkeit des Frameworks sicherzustellen, wurde es als Basis für die Entwicklung der Flugsoftware von *Flying Laptop* verwendet. Dieser aktive Kleinsatellit wurde ebenfalls an der Universität Stuttgart entworfen und gebaut. Das ermöglichte es, iterativ ein passendes Design zu finden, welches seine Nützlichkeit direkt in einem echten Umfeld unter Beweis stellen muss. In dieser Arbeit dient die Software von *Flying Laptop* zur Illustration der Konzepte und Funktionalitäten des FSFW.

Eine kurze Evaluation zeigt, dass ausgewählte Techniken des Software Engineerings die Entwicklung von Flugsoftware verbessern kann, indem Zuständigkeiten durch die Kapselung in Komponenten besser getrennt werden. Außerdem können sowohl der FSFW-Core, als auch vollständige Komponenten unverändert in verschiedenen Missionen verwendet werden können.

Preface

Somehow, I've always been attracted by difficult challenges. So I barely hesitated when I was offered a Ph.D. position to develop the flight software for a "small" University satellite of more than 100 kg, that was already three years behind schedule.

Fortunately, some foundations were already laid for the software, including the faint idea to write some kind of reusable software framework instead of a one-shot solution by Claas Ziemke. Also, the satellite hardware development was making progress thanks to experienced industry support, including my supervisor Jens Eickhoff, and the outstanding *Flying Laptop* team (sorry, guys, can't name you all).

So I started, with little knowledge in what I engaged, and might have resigned immediately if someone would have told me that the endeavour would take about six years. With me alone, the software would never have finished. But fortunately, I got assistance in the form of Ulrich Mohr, and it was the discussions and disputes we held over good and not-so-good software design that widened our horizon and shaped the Flight Software Framework as well as the flight software for *Flying Laptop*. While many of the smarter ideas of the FSFW stem from Uli's pen, its design was a collaborative effort, not only by Uli and me, and it's hard to tell who invented which part of the system.

And then, thanks to the unbelievable effort of the team and the infinite trust and commitment of our boss, Sabine Klinkner, we could finally see this crazy machine in action. While this was the closing of one chapter, it's not the end of the story, with many new, smart minds working on utilizing *Flying Laptop* or building its successors.

A warning to the following text: Real software development, under schedule pressure and with changing requirements, never happens as organized as this thesis might suggest. The truth is that the framework evolved in parallel with the *Flying Laptop* software, and both evolved with our own experience. The process was highly iterative, and neither did we have all the theory of software engineering available at the start, nor was a domain analysis performed until recently. This doesn't mean our work wasn't structured, but the design just wasn't created on the drawing board, but rather evolved from our hope that abstraction makes it easier to finish the *Flying Laptop* software.

Still, just like the *Flying Laptop* software in later development phases and in orbit miraculously "just worked", I'm glad that most of the theory and the FSFW fit together without too much friction.

Thus, as no one would like to read a book that contradicts itself multiple times and has various dead and open endings, this thesis describes a well defined top-down research and development process. So, enjoy reading (if you like books about software), and take the stringent structure with a grain of salt.

List of Acronyms

ACS attitude control subsystem

AOCS attitude and orbit control subsystem

ADC analog-digital converter

AIS automatic identification system

AIT assembly, integration and test

API application programmer interface

APID application process identifier

ASCII American Standard Code for Information Interchange

AUTOSAR Automotive Open System Architecture

BRDF bidirectional reflectance distribution function

CAN Controller Area Network

CCSDS Consultative Committee for Space Data Systems

CDH command and data handling

CDPI Combined Data and Power Management Infrastructure

CFDP CCSDS File Delivery Protocol

cFE core Flight Executive

cFS core Flight System

COM Common Object Model

COP-1 Communications Operation Procedure-1

CORBA Common Object Request Broker Architecture

COrDeT Component-Oriented Development Techniques

CRC cyclic redundancy code

DDS data downlink system

DLR Deutsches Zentrum für Luft- und Raumfahrt

DMA direct memory access

EDAC error detection and correction

ECSS European Cooperation for Space Standardization

EPS electrical power subsystem

ESA European Space Agency

FDIR failure detection, isolation and recovery

FIFO first in, first out

FLP2 Flexible LEO Platform

FOG fibre-optic gyro

FPGA field-programmable gated array

FSFW Flight Software Framework

FSW flight software

GNSS global navigation satellite system

GPS Global Positioning System

ICD interface control document

I²C Inter-Integrated Circuit

ID identifier

IDE integrated development environment

IO input/output

IRS Institute of Space Systems

IRU inertial reference unit

IP internet protocol

IPC inter-process communication

LAN local area network

LAMP Linux, Apache, MySQL, PHP

LEO low earth orbit

MAL message abstraction layer

M&C monitoring and control

MGM magnetometer

MGT magnetic torquers

MICS Multispectral Imaging Camera System

MO Mission Operations

MOS Mission Operations Service

NASA National Aeronautics and Space Administration

OBC on-board computer

OBCP on-board control procedure

OBOSS Onboard Operations Support Software

OBS on-board software

OBSW on-board software

OCS orbit control subsystem

OMG Object Management Group

OO object-oriented

OOP object-oriented programming

OS operating system

OSAL operating system abstraction layer

OSI Open Systems Interconnection

OSIRIS optical high speed infrared link system

OSRA On-Board Software Reference Architecture

PCDU power control and distribution unit

PAMCAM Panorama Camera

PDU power distribution unit

PLOC payload on-board computer

POD plain old data

POSIX Portable Operating System Interface

PROM programmable read-only memory

PSS power supply subsystem

PST polling sequence table

PUS packet utilization standard

QoS qualtiy of service

RAM random access memory

RF radio frequency

RID report identifier

RMAP Remote Memory Access Protocol

RMI remote method invocation

RMS rate monotonic scheduling

RPC remote procedure call

RTEMS Real-Time Executive for Multiprocessor Systems

RTOS real-time operating system

RTTI run-time type information

RW reaction wheel

S&M structure and mechanics

SAVOIR Space Avionics Open Interface Architecture

SFINAE substitution failure is not an error

SI International System of Units

SLOC source lines of code

SOA service-oriented architecture

SOAP Simple Object Access Protocol

SOIS Spacecraft Onboard Interface Services

SPI Serial Peripheral Interface

STB system testbed

STL Standard Template Library

STR star tracker

SUS sun sensor

TC telecommand

TCS thermal control subsystem

TM telemetry

TTC telemetry, tracking and control

UAV unmanned aerial vehicle

UART Universal Asynchronous Receiver Transmitter

UML Unified Modeling Language

US United States

USB Universal Serial Bus

VC virtual channel

WCET worst-case execution time

1. Introduction

Since its first practical use to solve problems, software has penetrated deeper and deeper into everyday life. First, executable algorithms became indispensable in research and certain economic sectors, such as meteorology and banking. Then, with the advent of personal computers and end-user software, software found applications in the majority of households. The *Internet* paved the way for truly personal and portable computing and brought software to everyone, e. g. in the form of information access and entertainment by smartphone. However, while these products are immediately recognizable as computers, the importance of software also grew as an element embedded in non-computer products, such as children toys, kitchen equipment and cars.

This process is likely to continue in the future. Many of the buzz topics currently discussed in research and media are based on more software embedded in more products. *Smart homes*, *Industry 4.0* and *autonomous driving* are prominent examples. These developments have in common that they are not only about adding small pieces of code here and there, but also to have larger software in charge of performing and controlling complex activities, such as driving a car. Software for such tasks has totally different requirements than that for, say, a stand-alone washing machine. Thus, future embedded software will grow in size and complexity.

Spacecraft FSW is no exception to that rule: There are indications that code size, and therefore complexity rises continuously since the day of the Apollo program [36]. And indeed, for many space mission today, complex control software is an enabling technology, be it to land rovers on Mars or rocket stages on Earth.

Thus, improving flight software (FSW) design and development is a key element for the success of future space endeavours, as complexity rises the same time as budgets are under pressure. Better FSW enables more complex mission while reducing cost by:

- directly reducing FSW development cost and time,
- reducing AIT efforts by allowing tests with partial systems,
- reducing ground observability demands by improving autonomy, and
- simplifying operator training.

Fortunately, the discipline of software engineering and software architecture has produced significant results in the last decades and keyed phrases like *frameworks*, *software components* or *design patterns*. Many of these techniques form the basis of the tremendous success of today's software industry[1].

However, it is not obvious which design concepts are applicable and useful for embedded software in general and spacecraft flight software in particular. Research in that specific topic has been rather sparse, according to [89], p.17 this is a "self-perpetuating" situation due to the backwardness of embedded software. Another reason might be that the required complexity and therefore pressure has not been large enough yet to improve software design.

So, the questions that are tackled in this thesis are: Which software engineering techniques improve FSW design, and how to apply these techniques to have a practical benefit? Or, reformulated as a research hypothesis:

> *Selected software engineering techniques improve development of complex embedded software in general, and spacecraft flight software in particular.*

To find some evidence for this hypothesis, this work is structured as following:

- At first, Chapter 2 gives a brief summary of software engineering methods and techniques, together with an evaluation of their applicability to embedded systems.
- Following, a so-called *domain analysis* is performed in Chapter 3, which aims to capture typical functionality any FSW has to provide.
- Chapter 4 presents the Flight Software Framework (FSFW) as the main work of this thesis. It uses selected software engineering techniques to provide a framework for the implementation of any FSW, following the feature list identified in the domain analysis.
- A brief evaluation of the FSFW is given in Chapter 5, which also covers possible improvements.
- The thesis concludes with a summary in Chapter 6.

Due to the malleable nature of software, it often happens that abstract software design concepts fail to keep their promises when facing real-world challenges. To avoid this pitfall, the flight software of *Flying Laptop*, a small satellite for Earth observation developed at the University of Stuttgart, was implemented on the grounds of the FSFW.

[1] As of 2015, four of the ten most valuable companies worldwide are mainly regarded as software companies [31].

1.1. The Nature of Complexity

An important property that arose in the above introduction is *complexity*. In a famous essay called "No Silver Bullet" [9], Frederick P. Brooks evaluates the term in relation to software.

First, he argues that there are two flavors of complexity for software engineering: *Essential complexity*, which is inherent to the problem to solve, and accidental or *incidental complexity*, which happens when solving the problem with a specific set of tools.

In natural science, essential complexity is managed by constructing and testing abstract models of the original problem, which works because these models still capture the fundamental principles which are supposed to exist in the laws of nature. However, "no such faith comforts the software engineer", as software controls and interfaces other human-built machines with arbitrary requirements. Even worse, as no two parts in software are the same[2], complexity increases "much more than linearly" with size.

Incidental complexity comes into play when a plan to solve the essential issue is put into practice, and includes the translation in a programming language, the challenges of handling hardware elements, and the difficulties of testing a program. According to [9], many original issues of incidental complexity are already resolved by high-level languages and good accessibility of testing environments, so incidental complexity is not the main challenge for software engineering.

To handle essential complexity, Robert D. Rasmussen argues in [91], that complexity is a matter of perspective, i. e. how well a problem is really understood. So, by introducing an overall, *recurring structure*, to help orienting in a complex system, *layering*, to build on understood foundations, as well as *separation of concerns*, to allow focusing on a part of the whole, essential complexity becomes more manageable, and therefore decreases ([91], p. 8).

However, as none of these principles is an intrinsic property of software, it requires effort to introduce and maintain them. This effort is not necessarily essential to the problem, so there is a risk that too much of the above ingredients cause an increase in incidental complexity, jeopardizing the original idea.

To conclude, the challenge of software engineering is to manage the extraordinary essential complexity of the task without accidentally introducing incidental complexity.

[2] Identical code parts are factored out to a common subroutine.

1.2. Diagrams and Code Examples

It is always challenging to illustrate the inner workings of software. In this thesis, to display certain software features, diagrams of the Unified Modeling Language (UML) are used where applicable. The syntax of these diagrams adheres to the UML 2 specification [88]. Code examples in the diagrams are put in comments, and are typically in the form of simple, C++ style pseudo code.

In addition, the thesis contains some examples in the form of source code, mainly in the Appendix. These are written in C++. Default include files (such as `<iostream>`) are only added to the listings if relevant, otherwise they are omitted for brevity.

2. Software Engineering Tools and Methods

The discipline of software engineering is quite young, especially if compared to other engineering topics, such as construction engineering with 4000 years of heritage. Even though the first program is often assigned to Ada Lovelace as early as 1842 [58], only with the advent of usable computers after World War 2 did software become a relevant scientific topic.

From that time on, software development was under the constant pressure of exponentially growing computing performance [33]. This led to the development of assembler and later procedural languages to provide a human-readable abstraction from machine code and enhance programming efficiency. Following, advanced concepts such as object-oriented programming (OOP) were introduced. As OOP is fundamental for many concepts used in this thesis, the first Section 2.1 of this chapter gives a short introduction to those paradigms. Readers familiar with OOP may skip the section.

From the late 1970s on there were only few new trends in language design itself, which led to the statement that nothing of relevance was invented in computer science since then (e. g. [89], p. 1). In the author's opinion, this is misleading: Computer science matured from an intellectual art pondering over fundamental concepts to an engineering discipline devising practical applications. In other words, there was a transition from computer science, to *software engineering*.

The term software engineering has many definitions (e. g. [72]). The main question this discipline poses, is *"How to manage, design, build and maintain software systems?"*. Since its establishment in the 70s [87], a vast amount of time and money was spent to find an answer. This thesis will closer investigate the following software engineering subtopics:

- **Development methods**, dealing with the organization of the software development process, i. e. making sure the right problem is solved correctly.
- **Design and implementation techniques**, dealing with the application of programming languages and paradigms to efficiently solve a certain problem.

The last thirty years showed a lot of innovation in both aspects. The evolution of the first point can roughly be described by the struggle between classical *plan-driven* methods, and more recent *agile* methods. Some background to this struggle and an introduction to selected methods is found in Section 2.2.

However, for this thesis, the main focus is on the second point: The objective is to identify techniques to allow better[1] development of more complex software systems. To achieve this, the discipline of software engineering devised techniques to apply fundamental design principles like *decomposition* and *abstraction*, as well as methods to allow code reuse and to improve productivity of the programmer. Section 2.3 evaluates relevant concepts.

Even though these improvements are widely applied to the development of application software, they are rarely used neither in flight software, nor in embedded software in general [66]. Section 2.4 tries to explain the reservations of embedded programmers and outlines under which conditions abstraction and architecture are beneficial for embedded programming.

2.1. Object-Oriented Programming

The basic idea of object-orientation is to merge data structures and functions operating on them into a single logical unit, which is called an "object". This idea led to object-oriented programming (OOP) as a programming paradigm and various theoretical and practical concepts associated with it. OOP is a very general concept, i. e. it allows many different design and implementation techniques. In addition, different object-oriented programming languages apply different subsets of the concepts in variable forms. Following, some typical features relevant for this work are described in the terminology of the C++ programming language.

2.1.1. Classes and Objects

A class is the definition of a certain type of object. It includes the definition of data, called *attributes* and functions, called *methods*, associated to the class. An object is a concrete instantiation of a given class definition. This implies that there may be multiple instantiations of a given class. To instantiate multiple classes with different attribute settings, classes can have parametrized *constructors*, which are called on object creation to perform basic initialization. If no specific constructor is given, a default constructor is generated. An example of class creation and instantiation can be found in listing A.1.

This definition, however, says nothing about what classes and objects actually *are* in a given project. One approach is the "everything is an object" strategy, which means that any data structure, interaction and algorithm can somehow be implemented as an object. Still, there is a distinction between such ad-hoc objects and those that form the basis of a software design. For this essential

[1] I. e. cheaper and faster, but even more so higher quality software development.

part, it is good practice to create objects that have a meaning to the clients using the software ([9], p. 221).

For example, an object-oriented (OO) embedded controller may consist of the following main objects:

- **Sensor objects**, representing input sensors
- **Controller objects**, representing control algorithms
- **Actuator objects**, representing actuators of the system

There are more details on principles and best practices for class design in the literature, such as the SOLID principle[2] [81].

2.1.2. Encapsulation

A central concept of object orientation is encapsulation of data and functionality. It postulates the simplification of complex software systems by hiding details of logically independent groups of data and function in dedicated software objects. Direct external access to encapsulated information and functionality is not permitted [84].

This concept enables the separation of *private* and *public* elements of a class. Restricting access to public elements reduces coupling of classes, which increases maintainability, as changes of private methods or attributes do not propagate beyond class boundaries. Also, encapsulation eliminates an error source as unwanted access to the "inner workings" of classes is inhibited.

Moreover, encapsulation naturally helps structuring data, as, by design, all data belongs to a certain class instance. In addition, class definitions form namespaces, i. e. the full name of an attribute `counter` of class `A` is `A::counter`. This reduces the risk of naming conflicts.

In listing A.1, it is not possible to change the private attribute `value` from outside the class by accident. Doing so will cause a compile-time error.

2.1.3. Inheritance and Object Composition

Another main feature of OOP is inheritance, which allows to establish a *Is-A* relationship between classes. A sub- or child class `B` *inherits from* a base (or parent or super-)class `A`. `B` reuses all attributes and methods of `A`, and may also extend the class and refine its methods by *overriding* (a feature which requires dynamic dispatch, see section Section 2.1.4 below).

[2] single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion

This is a main driver for code reuse within a project, as it allows the extraction of recurring control flow in similar classes to form a common base class. As an example, a concrete class to handle a specific Pt-1000 temperature sensor, e. g. `Pt1000Sensor`, may inherit from a common base class, e. g. `TemperatureSensorBase`, which is reused by other temperature sensor classes.

Another possibility to reuse code is object composition, which forms a *Has-A* relationship between classes. A reference or an instance of a class `C` may be included in another class `B`.

An example for both inheritance and object composition is found in listing A.2.

2.1.4. Dynamic Dispatch and Interfaces

Dynamic dispatch is a technique that allows an object to select the code of a method at run-time. This is especially relevant in conjunction with inheritance: If a child class overrides a method of the parent class, it must be ensured that an external caller actually executes the child class code even if it has a reference of the parent's class type only.

Interfaces make use of dynamic dispatch by declaring the syntax of a given method call and requiring concrete classes to implement the method. The terminology for the example in listing A.2 is: Class `B` *implements* interface `PrintIF`.

In C++ there is no explicit interface construct, but interfaces are effectively implemented using *abstract classes* with pure virtual functions. Just as all parameter types and the return type defines the signature of a function, the signature of all methods of an interface define the interface's *signature*. To explicitly distinguish classes and interfaces, all interface names in this thesis end with the letters `IF`.

Interfaces are a more powerful technique for code decoupling than simple encapsulation. First, an object implementing different interfaces provides specific views to different callers, only exposing relevant information for each. In addition, a caller can access different objects with the same interface uniformly, without need to care for details. However, C++ interfaces only ensure syntactical correctness, semantical correctness is not ensured. It must be maintained by other means, e. g. concise documentation of the interface.

As an example, a flight software's device handling class may implement a `HasHealthIF` and a `HasParameterIF` interface, which allow external access to the health state and to internal parameters. The implementation must ensure that e. g. health state access actually works as expected.

2.2. Development Methods

Software development strategies can roughly be divided into two categories: So-called *agile* approaches and *plan-driven* methods. While plan-driven techniques rely on a well defined design before beginning the actual coding and on strict rules concerning progress reports and milestones, the agile approach provides more freedom to programmers and allows the design to evolve during coding, often starting with only a minimum idea of how the end-product will look like. Both agile and plan-driven methods have certain strengths and weaknesses, which make them particular useful in specific environments, or *home grounds* [7].

This section gives a short insight to both methods and discusses ideas to avoid getting stuck in one of the two extreme ends when choosing a method.

2.2.1. Plan-Driven Processes

Plan-driven methods are a result of the advent of software engineering as a discipline: When software became both important and complex, e. g. in airplanes or power plants, there was a growing need to manage quality and budgets of software.

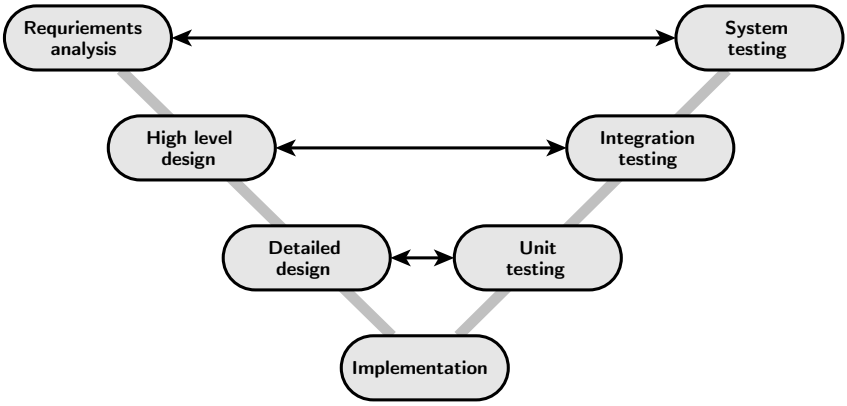

Figure 2.1.: Main concept of a V-model process (after [10])

Main concept is the application of hardware engineering processes to software development, typically in the form of a waterfall or V-model (see Figure 2.1). Software development is characterized as phases of requirements definition and detailing, designing, implementation, integration and test, which in principle

follow one after the other and are accompanied by documentation and verification activities. A customer reviews the project after each phase, mainly by checking the produced documents.

There are a number of standards emphasizing or supporting such approaches, such as US military or IEEE standards. For example, the ECSS standard "Space Engineering - Software" [42] describes the processes and plans for space software projects.

Home Grounds and Deficiencies

Plan-driven methods allow a very good division of responsibility and labour. Given all steps are thoroughly documented, requirements elicitation, design and implementation are independent tasks. As the program is decomposed into smaller units with defined and documented interfaces, many people can implement in parallel. Thus, these methods scale well for larger projects. Also, additional or new staff becomes acquainted to a project quickly by consulting the documentation, as long as they are trained on the process. There is a strong focus on formal software quality and traceability, which is assured by extensive reviews of code and documentation after each step.

However, this formalism is bought dearly with overhead of management and paperwork. This is in itself an issue for small projects. First, there is not enough staff to extensively elaborate on plans and documents. Moreover, the only addressee of a given document might be the one who wrote it. In addition, the elusive nature of software makes it sensitive to changes during a project. In order to avoid confusion between developers and reviewers, these changes need to propagate into every related document. Thus, documentation overhead will consume a significant percentage of workforce of every developer throughout the whole project duration.

Plan-driven methods also have the implicit assumption that all relevant requirements are caught and the planned architecture is good. Sometimes, that is not even remotely true, dooming a software project before a single line of code was written. Also, written specifications are a source of misunderstanding and misinterpretation. Even worse, such methods may hinder innovative projects: As it is difficult to perform a detailed planning up-front, disruptive technologies are often dismissed as unrealistic or too expensive.

Thus, these methods are especially well-suited for large projects with staff well trained on the process. Also, they fit fine if some formalism is required anyway, e. g. due to certification requirements. In such projects, high quality and complete documentation are valued higher than early delivery and testing.

2.2.2. Agile Methods

Watching many plan-driven software projects fail, a group of software developers known as the *Agile Alliance* decided to promote an entirely different approach to software development. In their *Agile Manifesto* [5], they emphasized to rely on human interaction and motivated programmers instead of piles of documentation to develop good programs. The customer-supplier relationship is seen as a collaborative activity for a common goal. Software development is seen as an iterative and incremental process, growing versions of the product are delivered repeatedly (see Figure 2.2). To maintain good quality, there is often a strong focus on software tests, which are developed in parallel with the project.

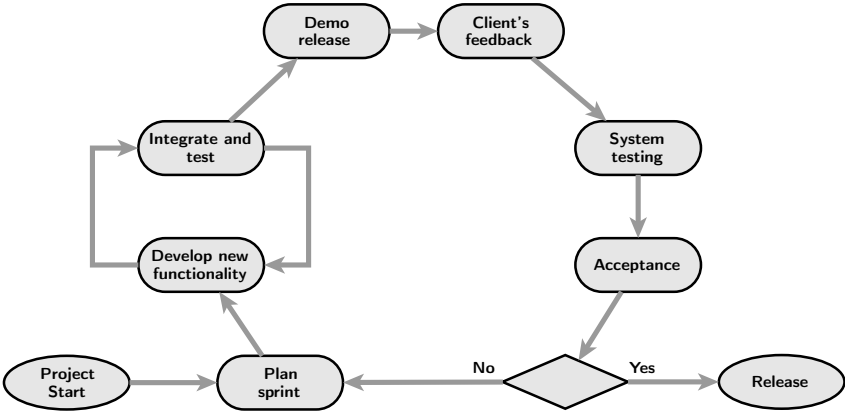

Figure 2.2.: Main concept of an agile process

From these ideas evolved a number of managing and programming techniques, such as *Extreme Programming* (XP) [4] or *Scrum* [94]. Scrum, as an example for agile methods, is a light-weight management technique for software teams. There is a Scrum leader, who maintains an open collection of the requirements in a *product backlog* and organizes meetings in fixed and short intervals, where the backlog is checked and features are selected for the next software revision. In addition, the process requires the team to hold short, daily informal meetings to coordinate its work.

Home Grounds and Deficiencies

One explicit goal of agile development methods is the reduction of overhead. Therefore, they lack the risk of drowning programmers in planning and documentation work. Instead, they are allowed to do what they typically prefer:

To create code. The incremental approach makes an immediate project start possible, even if requirements are not perfectly defined and verified. Also, these methods allow early testing of features to gain user feedback, which is probably more significant than written requirements. The risk of misunderstanding is reduced by frequent informal face-to-face meetings amongst developers and, less frequent, between developers and customers.

Agile methods always rely on tacit knowledge which is distributed through the team by meetings or techniques such as *pair programming* [4]. This approach is infeasible for large teams, a typical limit is somewhere between ten and twenty people [7]. So projects which require larger teams due to sheer size are not suitable for agile methods. Also, assigning new staff is difficult, as explicit personal training is required. Moreover, loosing key personnel and its knowledge may doom a project.

In some cases, the advocated reduction of planning is a severe downside as well: Especially in complex cases, fundamental issues may arise in an oversimplified architecture late in the project, requiring a fundamental redesign. In fact, the fear of such expensive late defects is a main driver for plan-driven methods.

As a very short summary, agile methods are most useful for projects with a small team size, where an early testable version is more useful than explicit documentation.

2.2.3. Finding the Right Balance

So, which side to choose? The authors of [7] argue that agile and plan-driven development is not necessarily mutual exclusive, but complementary. They propose to balance the risks of introducing a certain method and discuss five elements that give an orientation whether a project should use more agility or planning. These are:

- **Project Size:** The number of personnel involved in developing the program. It is safer to apply agile methods to small teams.
- **Criticality:** Determines how important it is to have bug-free software. If people's health may be affected, plan-driven methods should be preferred.
- **Dynamism:** How much change of requirements is expected during project lifetime. Agile methods perform better in dynamic environments.
- **Personnel:** This is a factor to determine how experienced the project members are. The dependency on people is higher in agile projects, therefore more skilled people are required there.
- **Culture:** A factor to measure if people feel more safe in a chaotic or a well ordered environment.

With these factors, a polar graph for a given project can be drawn to identify its home grounds. A small area indicates that agile methods should be preferred and vice versa. This is an indication for the selection of a specific development method. In many cases, it is best to find some middle ground, e. g. doing a classical requirements detailing and design approach and then implementing the design in an agile manner. The right solution depends on the problem at hand.

Home Grounds for a University Small Satellite Software

As an example, consider the flight software development of a small University satellite mission, for which such an analysis was performed in [14].

The amount of personnel in such a team is typically small, and seldom higher than five people. As a satellite's on-board software is developed, the criticality of the project is rather high. On the other hand, no people will be endangered, so there is no safety-of-live criticality.

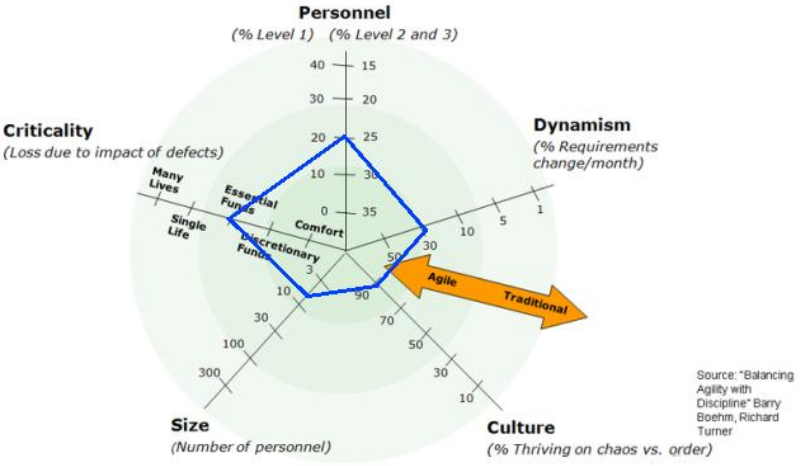

Figure 2.3.: Home grounds of a small University flight software project.

Evaluating dynamism is not as easy: In theory, most elements of a satellite are fixed at project start, therefore requirements should not change rapidly. In University reality, however, the exact set of sensors and actuators is often not fixed, not speaking of the maturity of payloads. Therefore, dynamism is considered high. Even though culture is another vague term, it is safe to assess

that young University attendants feel more comfortable with many degrees of freedom in development than in a fixed well-ordered environments.

The qualification of personnel depends on the specific situation. Still, in a University environment, the actual developers are most likely Ph.D. students, so their experience is intermediate at best.

The resulting graph in Figure 2.3 shows that the home grounds of such a project lies in a region were agile development methods are more appropriate than plan-driven ones. However, the graph also identifies two main risks: The FSW's criticality for mission success, and the dependence on key development personnel.

2.2.4. Outlook on Development Methods

Selecting the right development method is affected by the advancements in programming tools and implementation techniques. Many aspects driving the discussion above are quite persistent over time, as the 1975 essays in *The Mythical Man-Month* [9] show.

However, improved auto-documenting and testing facilities may allow concise documentation of software developed in an agile manner, thus conquering some home ground of plan-driven methods. Also, better implementation techniques, such as integrated development environments (IDEs) or domain-specific frameworks, constantly extend the project size (in terms of complexity) which a small team can handle. For example, the 3 million-source-line-of-code (MSLOC) software for the Mars Science Laboratory, including the sky crane lander and the Curiosity rover, was done by about 35 developers [68].

On the other hand, improved software or system modelling environments may support plan-driven techniques. In principle, software models, if regarded as independent from code, are sensitive to the same problems as documentation, i. e. the maintenance effort to keep them in sync with code is often higher than the gain. However, improved modelling techniques and additional libraries may lead to systems where model and implementation are merged, thus providing a format that is both human and machine readable. So by creating a concise plan of the software system, large parts of the implementation are already done.

2.3. Design and Implementation Techniques

The evolution of implementation techniques is twofold: First, there are programming languages and language features which are intended to improve efficiency and code reuse. In many cases, it took some time until useful techniques

were widely accepted, e. g. the use of high-level languages as opposed to assembler [9] or the concept of structured programming [32]. Except for the applicability of object-oriented programming (OOP) in embedded systems, which is explicitly addressed in Section 2.4, language features are not considered here.

OOP techniques, however, form the basis of the second evolutionary path: Techniques to construct large software from existing, "standardized" parts. In mechanical engineering, this concept is ubiquitous, given the vast catalog of standardized screws, bolts, bearings, etc. However, as many software textbooks (e. g. [30]) point out, the concept was first invented and applied in the second quarter of the 19^{th} century for rifle manufacturing. From that early success, it still was a long way to go to buying a normed M5 screw in any do-it-yourself store.

The idea to copy the "interchangeable component" approach to software engineering was first introduced in [87]. Implementations of this idea are summarized under the term *component-oriented* programming. Some of its flavors are outlined in Section 2.3.1 below.

In parallel, it was noticed that reusability does not happen automatically when using OO techniques. Instead, classes and objects must be designed for reuse, which leads to software *frameworks* [73]. These are further described in Section 2.3.2.

Components and frameworks do not exclude each other. Indeed, as pointed out in [105], "there is no such thing as a component". This means that any software component by itself is worth nothing without a dedicated framework supporting this component, just like an integrated circuit (a default analogy for software components) is useless without the definitions of the pin positions, voltage levels, etc. plus their implementation on a (typically non-generic) printed-circuit board. Therefore, Section 2.3.3 will separately discuss the concept of *component frameworks*.

Service-oriented architecture (SOA) is an extension of the component framework concept, which introduces *services*, as an interface definition between components, to allow independently developed components to interact and form one larger application. SOA is introduced in Section 2.3.4.

In addition, several design and implementation techniques were devised to allow the construction of a software product family, by making use of the techniques introduced above. These techniques are summarized as *software product lines* or *generative programming* (after [30]) and are described in Section 2.3.5.

Another concept is that of *design patterns* [60]. Instead of reusing components, the goal is to identify abstract design ideas and collect them for reuse. The analogy to mechanical engineering is that of generic concepts, such as the general construction of a gearing mechanism or a tongue-and-groove joint, which are

similarly applicable to a multitude of construction problems. An explanation of what a pattern is - and what not - is given in Section 2.3.6.

Most of these concepts are common standard for desktop and mobile, but less so for embedded applications. Therefore, each of the following sections will investigate the applicability of the technique to embedded systems design.

2.3.1. Components

Components somehow are the "running gag" of software engineering: Envisioned in a talk of M.D. McIlroy in 1968 [87], and hyped in the 1990s as the "next big thing" it seems as if their practical breakthrough is still waiting to come. Still, the idea behind components is intriguing: Instead of implementing certain features over and over again, a company with domain expertise sells a ready-to-use piece of software. Developers assemble a number of such pieces to form applications, which are supposed to be built faster and more reliable, as components are already checked and tested.

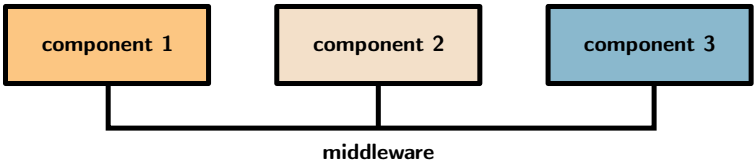

Figure 2.4.: Basic concept of a component-based software. A number of fairly independent components is connected by a dedicated middleware.

In the 1990s, quite some effort was put in the standardization and marketing of component technologies, which, among others, resulted in the Common Object Request Broker Architecture (CORBA) developed by OMG and Microsoft's component object model (COM) technology [98]. Their intention was to provide a *middleware* to manage interaction between any kind of component (see Figure 2.4). These concepts did not really succeed for several reasons: First, there are general issues regarding broken interfaces or *contracts* between independently developed components. Also, the 90s attempts aimed at a maximum of generality, e. g. CORBA is a standard for inter-language, inter-machine remote procedure calls (RPCs). Adhering to those standards did not reduce workload for developers, but increased incidental complexity.

However, viewing components as a failed technology may be a misconception. The problem is that there are many different opinions on what a component is, it is more a "natural concept" ([30], p.9) than an artificial one. For example, components as described in [98] are supposed to have a size between simple library functions and full applications. This may be a bad choice of *granularity*.

When broadening the scope (of size and concept), some remarkable success stories can be found:

- Modern libraries, either as part of or extensions to a given language, include a lot of reusable components in function or class form, e. g. the C++ Standard Template Library (STL) or math libraries[3].
- A lot of applications, such as web browsers, IDEs or media players, support third-party plug-ins, which are components for the given environment.
- Modern applications often have more in common with components than with traditional software packages. For example, web servers often use the Linux, Apache, MySQL, PHP (LAMP) stack to provide typical web services.
- Similarly, many smartphone apps show tight interaction, e. g. when using a navigation app to display geospatial information received in a messenger app.

Components of all the above examples require a certain environment for execution, be it an entire operating system (OS) or a plug-in-capable application. This environment may be viewed as a *framework* for component execution. After introducing frameworks in the next section, the relation of components and frameworks is further studied in Section 2.3.3.

Component technology has not had much influence on embedded software development yet. There are some general research efforts (e. g. [1]) as well as some related to space missions [74], but the most remarkable use of component technology is Automotive Open System Architecture (AUTOSAR) [2], which is outlined in Section 2.3.3.

2.3.2. Frameworks

The idea of frameworks arose when larger systems were designed with object-oriented (OO) languages. The term was coined in [73]. The authors point out that using OOP alone is no guarantee for extensible or reusable code, but that certain techniques need to be applied to reap the benefits of OO design. The approach is to elicit the common interfaces (called *protocols* in [73]) and algorithms of a given program, and create "good abstractions", which form a framework to develop similar applications. This has the important implication that frameworks are always intended for a certain domain (see Section 2.3.5), they are not general-purpose[4].

[3] Math libraries typically provide implementations of the sine function, a prominent component example in [87].

[4] In a way, programming languages are general-purpose frameworks.

The authors argue that frameworks typically evolve in the following steps:

- First, the abstraction of one or more concrete implementations is found and refactored into framework code.
- The next "natural stage" is a *white-box framework*. It can be used to develop similar applications, but requires a deep understanding of the inner workings of the framework. Technically, this is because the framework is mainly specialized by subclassing.
- A *black-box framework*, as a final stage, hides the inner workings of the framework from the application developer. One variant to do so is by plugging software elements or *plug-ins*, into existing framework containers. Thus, the programmer only needs to know the required plug-in interfaces to create a customized software.

These evolutionary steps emphasize the bottom-up approach recommended for framework development, as "an abstraction is usually discovered by generalizing from a number of concrete examples" [73]. An illustration of white- and black-box frameworks is show in Figure 2.5.

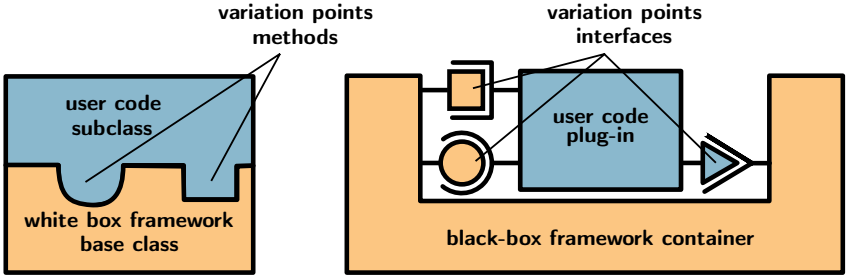

Figure 2.5.: Illustration of a white-box (left) and a black-box (right) framework implementation.

Frameworks are highly successful in application software development. The Eclipse programming environment, for example, is a framework for integrated development environments (IDEs). Via plug-ins, the same framework allows software development in a multitude of languages, e. g. Java or C/C++, but also such things as LaTex document generation. Another influential example is the Android framework to develop mobile phone applications.

The use of frameworks is less common for embedded applications. One reason is the expected resources overhead for using non-optimized software on small embedded targets. Another potential reason is that frameworks work best with

OO-techniques, which are not widely applied in embedded systems. However, many companies maintain an internal code base used for multiple similar products. In many cases, this code base can be considered a framework.

AUTOSAR is a good example of an embedded framework and is further described in the next section.

For space software, there are also some publicly visible attempts to develop frameworks. Examples are the on-board software (OBS)-Framework [90] or the Onboard Operations Support Software (OBOSS) [104]. However, none of them was used in a space mission yet. Still, it is safe to assume that most prime contractors in the satellite industry maintain some kind of software "framework" in-house.

White-Box Frameworks

Since it is relevant in a later section, details on the concept of white-box frameworks and their relation to so-called *abstract base classes*, are provided.

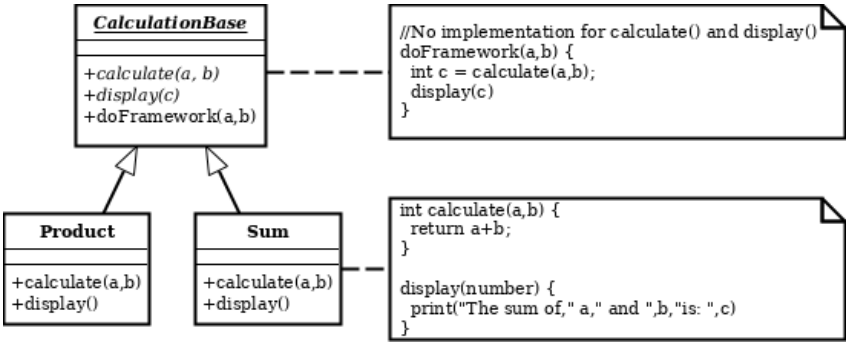

Figure 2.6.: Example setup of a base class and two child classes, which provide different implementations of `calculate()` and `display()`.

An abstract class is a class with method declarations that lack an implementation, i. e. there is no code provided for a method call. In OOP, this is allowed, as classes inheriting from the abstract class can provide an implementation, which is found using the dynamic dispatch features of the language (see Figure 2.6).

When implementing frameworks, methods of the parent class often call the abstract methods provided by the subclass. This creates an *inversion of control*, as the parent class structures the flow of execution for the child class [73] (see Figure 2.7). Thus, the abstract class is the basis of execution, and therefore

is called a *base class*. The abstract methods are called variation or *adaptation points* for the child class, as they allow modification of the parent's behavior.

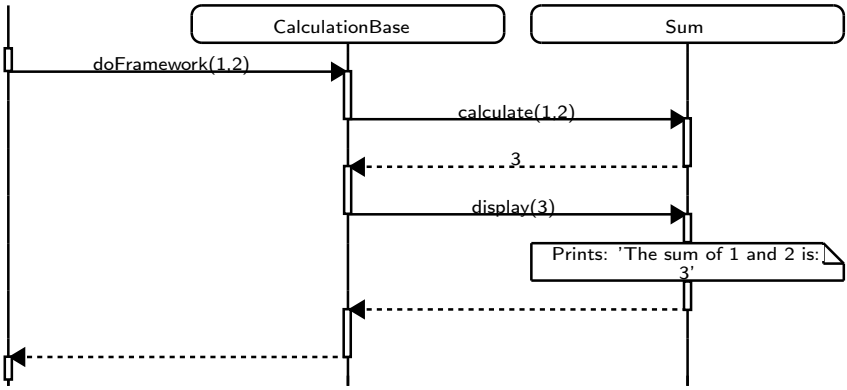

Figure 2.7.: Sequence diagram of interaction between a base and a subclass.

As a result, the framework is based on inheritance. Unfortunately, this requires a rather deep knowledge of the base class, as the circumstances under which the subclass is called and what results are expected, must be well understood. For that reason, such framework types are called white-box frameworks.

2.3.3. Component Frameworks

The concept of components always comes with the implicit assumption of a framework connecting components. Therefore, it makes no sense to "think about components as standalone chunks of functionality that can be 'glued together' after the fact" [105]. The framework is an essential part when working with components, as it manages component interaction (see Figure 2.8). Vice versa, this is not the case: Frameworks work well without components. A GUI building framework does not imply a component-based application.

At least, the component framework needs to describe the techniques for components to communicate via the middleware. In addition, it may define additional rules to create, compose and execute components [62]. As frameworks (as defined in Section 2.3.2) are designed for a given domain, component frameworks are as well. For example, CORBA is a component framework for distributed network applications.

Using a component framework promises the following benefits:

- A component framework provides a software architecture, there is no need to reinvent one.

- The framework guides developers in building components and combining them to form applications.
- Reuse of components is possible over different applications, as long as the same framework[5] is used.

However, the original domain of the component framework and the intended domain must mach: Building a 3D video game in the Eclipse IDE may work, but it will neither run with good performance nor be easy to develop. Still, Eclipse is a good example of a component framework, as long as the target application is a programming environment or something similar.

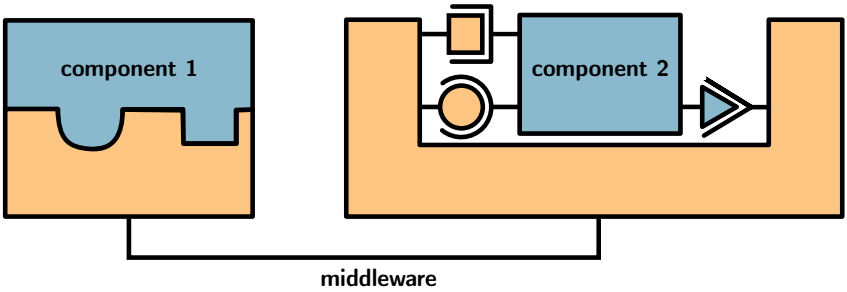

Figure 2.8.: Illustration of a component framework. It combines the concept of independent components with reusable framework elements.

A remarkable embedded example for component frameworks is AUTOSAR [2]. It defines an architecture, including application programmer interfaces (APIs) and interfaces for embedded automotive applications. Components play an elementary role in this architecture, they provide functionality by offering *ports* to other components. The framework is formed by the definition of an AUTOSAR run-time environment and a so-called *virtual functional bus* over which components communicate.

In contrast to the recommendations for framework development described in Section 2.3.2, AUTOSAR is a top-down approach, i. e. it was first specified and implemented later. This approach is prone to overspecification and may therefore not gain wide acceptance. Still, its market share is growing constantly [59][6].

For space, there are ongoing ESA activities to specify an on-board software (OBSW) reference architecture similar to that of AUTOSAR [100]. Included are studies for Component-Oriented Development Techniques (COrDeT) [74]. Like AUTOSAR, these are top-down activities. Another example is the NASA

[5] Or at least a framework with the same API.

[6] According to a personal talk, it took several iterations up to version 3 of the standard to make it actually usable.

core Flight System (cFS), which has been used on a number of smaller missions [82]. Both initiatives are described in Section 3.6.

Component Interaction

An important factor when using components is the way components interact. There are two fundamental ways of communication between software entities:

- **Synchronous:** One entity simply calls a function or method of the other. As the flow of execution moves to that function, the caller is blocked until completion. A phone call is a form of synchronous communication, as the listener typically waits for the speaker to finish.
- **Asynchronous:** A request is put in a message (or similar) and sent to the destination entity. It processes the request when appropriate and generates a response message to the caller. Neither caller nor receiver is blocked, but response reception may take a while. Writing letters is a form of asynchronous communication.

Communication between software entities are typically illustrated using UML *sequence diagrams*, as shown in Figure 2.9.

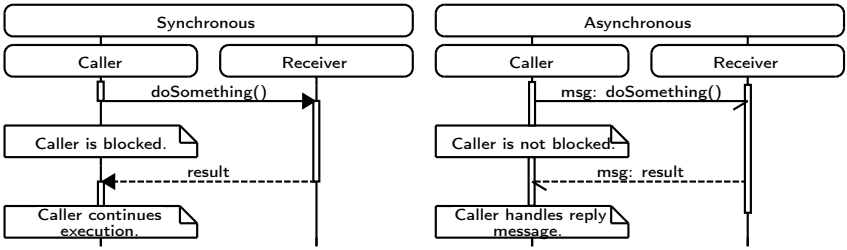

Figure 2.9.: Sequence diagram illustrating synchronous and asynchronous communication. Arrow tips and line styles indicate the type of communication.

The main idea of a component-based architecture is separation of concerns, i. e. a component should be developed with as few dependencies as possible. For real-time systems, this is true not only for functional, but also for temporal decoupling, i. e. the timing of component activities should not depend on other components. Thus, direct synchronous calls between components are problematic, as they create temporal dependencies.

For illustration, consider a control algorithm issuing a torque command to an actuator of the system. When using synchronous calls, the control algorithm commanding the torque would block execution until the request is transmitted

to the actuator and a confirmation is received in the controller. In effect, the control algorithm is unable to process input data for the next actuation cycle, making it unnecessarily slow.

Synchronous interaction is not forbidden in principle, and commonly used in interactive systems, where quick reactions on user input are most important. However, aside from potentially bad overall performance, the execution schedule, i. e. in which intervals the controller actuates the system, becomes difficult to predict.

To resolve that issue, all communication between components in a real-time system must be asynchronous. There are two main techniques to do so:

- **Messages:** The caller puts its request in a message, which is read and handled by the receiving component when appropriate. So-called *message queues* manage reception of multiple messages, typically with FIFO ordering.
- **Shared memory:** A component writes information to a shared memory region, where it is read out by the receiver at another point in time. While both reading and writing the shared memory is synchronous, the entire data transmission is asynchronous. To avoid reading half-written data, shared memory regions need to be locked, which happens with so-called semaphores or *mutexes*.

With these techniques, it is possible to build a component framework for embedded systems which adheres to real-time schedulability rules.

2.3.4. Service-Oriented Architecture

Another, more recent technique is service-oriented architecture (SOA). It focuses on the architecture required to form a software application from distributed, interacting, independently maintained software components which each provide certain *services* to handle a given "business case". To do so, it defines techniques to find and agree upon services to enable interaction between service providers and service consumers [71].

These service definitions are the main extension of the concept of component frameworks described in Section 2.3.3 above. A *service provider* is a software component which offers a certain functionality to other components with a well-defined interface. In general, it publishes a service in a *service registry*, so a *service consumer* receives the information to access the service, e. g. an address, and sets up a connection. This process is called *binding*, after which the service consumer can use the functionality by invoking the provider's service interface. Components can serve as providers and consumers of different and multiple services at the same time.

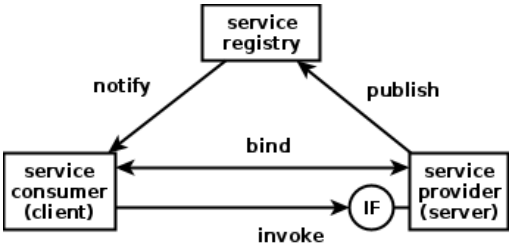

Figure 2.10.: Basic interaction scheme of service providers and consumers in SOA using a service repository.

The basic concept of SOA does not limit the communication technique to use, however, it was originally designed for large enterprise web applications, utilizing protocols like SOAP for interaction.

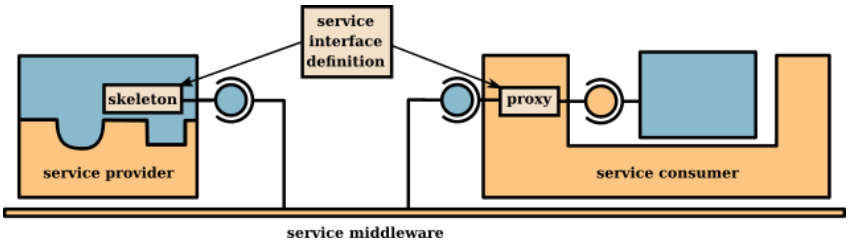

Figure 2.11.: Sketch of interaction between proxy on consumer and skeleton on provider side in a SOA deployment. The inner workings are not relevant for SOA, they may or may not use frameworks.

One common style of implementation is using stubs or *proxies* on client and *skeletons* on server side, which represent the communication partners locally (see Figure 2.11). Skeletons and proxies are generated from a common service interface definition.

Due to the distributed nature, the SOA concepts have not yet found their way into embedded applications. However, there are some initiatives to use SOA for complex systems. Two of these are adaptive AUTOSAR, an extension of the classic AUTOSAR for complex automotive control [2], and the CCSDS Mission Operations (MO) services, which apply SOA concepts in the spacecraft operations domain [19].

However, to make use of the concepts of SOA in embedded systems, two issues need to be resolved:

- **Communication complexity:** SOA communication often uses complex protocols for service discovery and invocation, which may introduce an overhead that outweigh its benefits. This is especially true if the system is rather static and dynamic service discovery is rarely used. In addition, as it is a conceptual offspring, SOA faces the same challenges regarding real-time interaction as component frameworks (see Section 2.3.3).
- **Service interface definitions:** The concept itself does not aid in the actual definition of service interfaces, as it is too generic to provide solutions for specific domains. However, if each software component only offers custom, non-generic services, components are tightly coupled, missing the original idea of loose coupling. Therefore, well-managed service definitions are crucial for a successful SOA deployment. This is further detailed below.

When taking a closer look at the above findings, it becomes apparent that they are not only relevant for using SOA in embedded systems, but for any usage of SOA. Thus, these may be two reasons why service-oriented architecture never fulfilled the huge expectations that were attributed to the concept before about 2009 [80]. Some even claim that SOA is an “architects dream”, but a “developers nightmare” [67], stating that the architecture is fine, but the tooling and understanding for actual implementations is rather poor. The recent concept of microservices may be viewed as an evolution of the original SOA idea [34].

Service Definition Issue

As stated above, good service interface definitions, or “useful abstractions" [73] are decisive for the success of a SOA-like deployment. These service interfaces shall, at the same time, be abstract enough to make them reusable while still having an obvious practical meaning and being simple to use. Moreover, service definitions must be stable over time, as a change typically requires adjustments in all consumers and providers.

To illustrate the issue, imagine a control algorithm component with two adjustable parameters a and b. Figure 2.12 shows two options for interfaces to access and adjust these parameters:

- On the left, there is a specific service interface, which is easy to implement and use. However, this interface is tightly coupled to the content of the component, as adding a new, or different parameter changes the interface. Therefore, this design is not reusable and the interface more or less obsolete.
- On the right, a more abstract service interface is displayed, which requires a component to define IDs for adjustable parameters and is therefore more difficult to implement. However, it does not change with the component’s

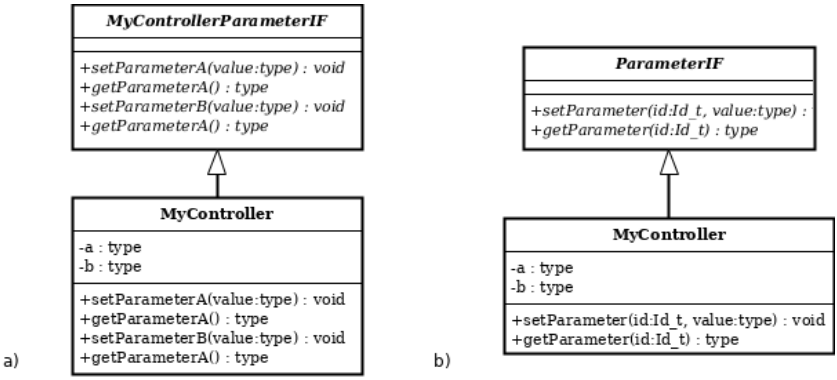

Figure 2.12.: Example of specific (a) and abstract (b) interface definitions.

content and can even be used for other components in the same way. In effect, this interface definition has better reusability and unifies access to parameters.

This simple example highlights the importance to find suitable, generic service definitions for the domain at hand.

2.3.5. Generative programming

Generative programming or *software product lines* are inspired by another hardware engineering analogy: The concept of assembly lines. As pointed out in [30], assembly lines are not about creating identical copies of a product (which obviously is easy for software), but to efficiently create customized similar products. For example, due to customer demand for individualization, there are rarely two identical cars leaving a factory today.

So, for software, individual solutions are not supposed to be coded by hand, but assembled from pre-built existing elements. Thus, the approach builds upon framework and component technology as described above. However, it extends these concepts by introducing *features* as customer-relevant functionality of a software. Based on the feature need of a customer, *generators* are supposed to automate the customization process.

There are few application-size examples of software product lines, some enterprise resource planning solutions such as SAP ERP use modules and configuration parameters to customize software. In the small, the C++ STL introduces default solutions for data containers, which are customizable to programmer's needs by C++ template metaprogramming [30].

There are no known convincing examples for product lines of embedded systems. However, some research was conducted to apply the concepts to the OBS-Framework [29]. Also, the NASA cFS has similar goals. It is further described in Section 3.6.5.

Domain Analysis

For software product lines to work, there is a need to specify the family of systems, or the *domain*, the product intends to cover. In [30], p. 20, there is a distinction between *vertical domains*, which classify systems with regard to their target application, such as Internet shopping systems, and *horizontal domains*, which group system parts with regard to their functionality, e. g. a database system. As a hardware analogy, the domain of "all types of cars" is a vertical, whereas "all types of screws" is a horizontal domain, as it can be used in many vertical domains[7].

The activity to isolate, define and describe a domain is referred to as *domain analysis*. It requires separating the domain from neighbouring topics (called *domain definition* or *domains scoping*) as well as collecting and analyzing knowledge of the selected domain. Getting a domain analysis right requires a certain amount of expertise, ideally by having designed several systems in that domain.

Feature Modeling

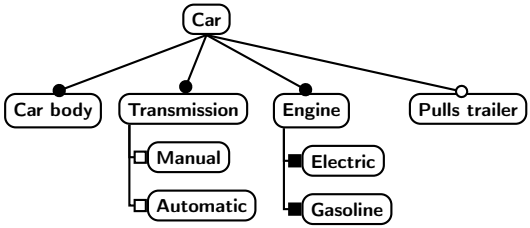

Figure 2.13.: Simple example of a feature tree. Features connected with filled circles are mandatory, those with an empty circle are optional. Filled squares represent groups of OR features, empty squares groups of XOR features. From [30].

The expected result of a domain analysis is a domain model. The authors of [30] propose to display this model in a *feature diagram*, which draws relationships between a basic concept and features and subfeatures in tree form.

[7] However, "all types of cars" may be a horizontal domain in the taxi driving system of a city. Domain classification is a matter of point of view.

Basic concepts and high-level features are typically abstract, whereas the leaves of the tree are concrete implementations or functionalities. Their diagram incorporates notations for mandatory (filled circle), optional (empty circle), as well as OR and XOR alternative features (filled/empty squares). Figure 2.13 illustrates the concept.

2.3.6. Design patterns

Design patterns were introduced in 1995 in a book of the same name by a group of authors since then known as the *Gang of Four*[8] [60]. It had an enormous impact in the OO community, as it tackled an urgent issue of OOP: How to make a good, reusable design. As the authors called it: "Designing object-oriented software is hard, and designing reusable object-oriented software is even harder." ([60], p.11). They proposed a number of abstract solutions for recurring problems in the form of a concise catalogue of *design patterns*. Most of these patterns are classics in OO design today, such as *Singleton*, *Adapter* or *Composite*.

However, design patterns are relevant on the level of implementation only, i. e. to create well-structured OO code. They are of little relevance on the level of architecture. In the wake of the book's success, many publications praised "patterns" of any form as programmer's universal remedy, be it for obvious building blocks or complete software architectures. To avoid misunderstanding, the term "design pattern" will be used in the original sense only.

As pointed out, design patterns are tightly linked to OO design. Thus, they are not in widespread use in embedded programming, as OOP isn't either. Also, some design patterns rely on dynamic memory allocation, a technique which is avoided in embedded programming (see Section 2.4.3). Still, as soon as OO techniques are applied, design pattern help making good, reusable designs.

2.3.7. Summary

The essence of the above recapitulation seems straightforward: There is an increasing amount of reuse in software systems, powered by the evolution of components and frameworks to component frameworks and SOA, making use of OO design patterns. The process of building a customer-defined product is handled by automated generators in software product lines as far as possible.

Unfortunately, evolution is much slower and less controlled, and there are many technical and non-technical challenges on the outlined pathway. For instance, on the non-technical side there is the training of developers accustomed to a certain style of programming. Or, more technical, the difficulty of finding the

[8] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

right level of abstraction to make frameworks truly reusable (see below). Also there is a certain overhead of creating and running reusable software when compared to a fully specialized one. Obviously, there will be no payoff for reusable software if the number of reuses is zero.

However, many of these concepts, especially frameworks and design patterns, are in regular use in today's software industry and are elementary for the exponential growth of digital applications and services. Still, more advanced techniques such as software product lines are far from being in widespread productive use.

This is even more true for embedded systems, were even basic OO techniques face strong scepticism. As argued, this is a valid attitude as long as complexity is low. With rising complexity, the advantages of being capable to handle complexity will outweigh the disadvantage of overhead and indirection, which is already acknowledged by initiatives such as AUTOSAR.

Finally, the transition from handcrafting to assembly line manufacturing for hardware did not simplify the process of inventing new products. But it helped to produce better and more complex products at lower costs. Likewise, there is an intrinsic effort in software design which cannot be reduced arbitrarily. None of the mentioned techniques is a "silver bullet" [9]. However, all off them are useful to handle more complex software systems.

2.3.8. Issues with Abstraction and Generalisation

All of the aforementioned techniques promote some form of *abstraction* or *generalisation*. Abstraction in computer science means to hide currently unneeded details from a user of the abstract element. A named function or subroutine is an abstraction. Generalization means to group similar concepts and form a single entity capable of handling these cases, eventually by providing parameters. For example, base classes collect the generic similarities of different subclasses. Both concepts are essential to manage complex programs and are often used in conjunction. Abstraction helps programmers to focus on the essentials, generalisation avoids duplications, thus reducing maintenance and refactoring effort.

However, there are downsides of both concepts to consider. Abstraction stands in contrast to the concept of coding a solution as direct and as localized as possible, as it introduces indirection to hidden details in the code (see e. g. [30], sec. 4.9.2). This is the same issue as hinted at in the discussion on complexity (Section 1.1) and probably one of the reasons why OOP and related techniques are so unpopular in embedded programming: Hiding implementation details is dangerous if non-functional aspects are essential, such as execution speed and memory footprint.

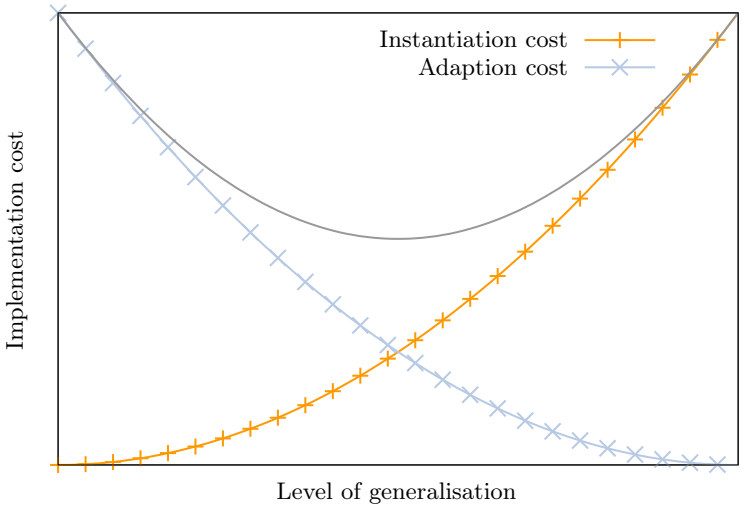

Figure 2.14.: Trade-off between instantiation and adaptation cost

Generalisation always requires a trade-off between the cost to instantiate the generalisation and the cost to adapt or duplicate existing code, as sketched in Figure 2.14. This issue is relevant both in the small and in the large: Is combining two similar functions into one worth the additional complexity? Is it easier to create a new application from scratch or to build one from an existing framework?

In essence, both abstraction and generalisation have a certain cost and therefore should not be applied for their own sake. The ability to come close to the "sweet spot" is a property that constitutes an experienced programmer.

2.4. Embedded Software Development

There are three main points that distinguish development of embedded software from that for desktop applications:

- Embedded software has a very specific **scope**: Together with the hardware, it forms a single product (a *cyber-physical system*) which interacts with its environment.
- As a direct result, **communication** to the outside world is much less standardized than on desktop systems.

- **Resource utilization**, i. e. memory footprint and usage as well as execution duration and timeliness are a major concern.

To illustrate this difference, compare the requirements of a washing machine controller to that of a text processing software. These points will be further illustrated in the following sections. Subsequently, the properties of spacecraft flight software as specific embedded software is discussed.

2.4.1. Scope

A common definition for embedded systems is that of a microprocessor hidden in a product other then a (general-purpose) computer (e. g. [96], p. 1). When taking into account the still-increasing number of "smart" products, the variability within embedded systems is extremely broad, ranging from light switches to aircraft autopilots. On the other hand, for a selected embedded system, the software has a very specific purpose, i. e. providing functionality for the actual product.

If the product is simple, the embedded software will be simple as well. Thus, there is no need to introduce abstractions of any form to manage complexity (c. f. Section 2.3.8). As a result, any form of OOP is ruled out right from the start.

However, at a certain point of complexity, software development will benefit from introducing abstractions of some form. This may start at a project size of as little as 1000 source lines of code (SLOC) ([66], comments).

Real-Time Operating Systems

Despite the broad diversity in embedded systems, the need to interact with the real world imposes two typical requirements: The ability to handle multiple issues in parallel, and to react on them in due time. For example, a spacecraft flight software is supposed to handle telecommand (TC) reception in parallel with control loop execution, which must finish at a certain point in time. This results in multitasking and real-time requirements for embedded software. However, parallel process execution imposes additional challenges: First, as the system is still a single entity, there is a need for inter-process communication (IPC) to safely exchange information between tasks. Second, concurrent competing access on shared resources (such as hardware interfaces) must be managed.

Real-time operating systems (RTOS) provide solutions for these common needs of embedded systems (see [110], chapter 6). With a very small overhead, they deliver proven and tested solutions for parallel task execution on a single processor. Also, if correctly configured, task execution is assured to keep timing

requirements. Additional features, such as clock and interrupt management are often provided as well.

The benefits of using a RTOS in embedded software development is widely accepted, multiple products are in widespread use. For example, the Real-Time Executive for Multiprocessor Systems (RTEMS) is often used for space applications. Thus, RTOSs prove by example that "useful abstraction" are possible for embedded systems.

2.4.2. Communication

The wide range of all and the specific scope of a single embedded system results in a very broad spectrum for external communication, ranging from a simple binary IO pin to complex high-speed networks. This heterogeneity is the reason for the low amount of standardization regarding device access compared to desktop computers.

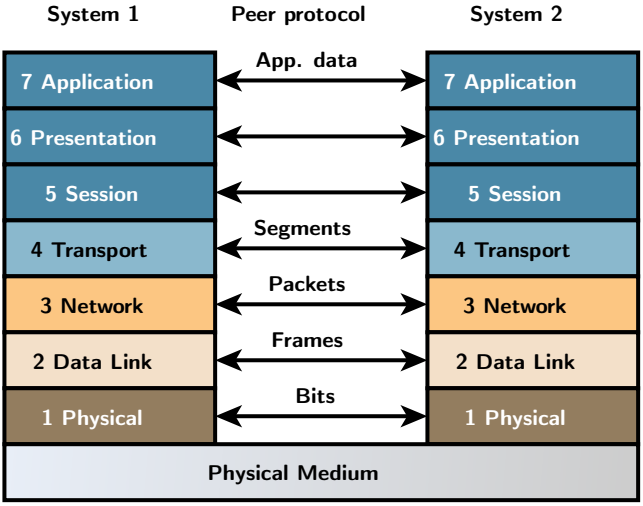

Figure 2.15.: OSI layer stack [70]

As a result, even though some RTOS are shipped with a set of device drivers, a large portion of embedded software development deals with communication. Depending on the availability of drivers, many layers of the Open Systems Interconnection (OSI) reference model need to be covered (see Figure 2.15). In

contrast, drivers and abstractions for most communication issues are readily available in desktop systems[9].

Thus, development time for embedded systems could be reduced by utilizing standardized drivers and abstraction of external devices. However, approaches aiming in that direction are likely to fail due to the variety of interfaces and memory and speed overhead objections (see below). Still, the more complex the system to develop is, the less significant is the overhead introduced by abstract hardware interfaces.

2.4.3. Memory and Speed Overhead

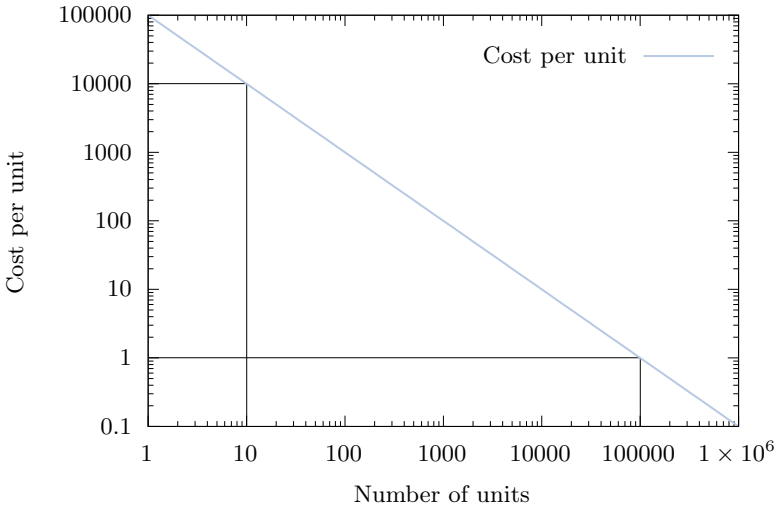

Figure 2.16.: Software cost per unit

Dealing with limited resources was a major challenge for embedded software development in the past. Is it still today? First of all, there is always a limitation in computing resources, no matter how fast embedded processors will be. However, at the same cost, capabilities grow exponentially, continuously extending the possibility to put complex functionality in software. Indeed, one can argue that our ambitions of what embedded systems can do grow with the same rate as hardware capabilities grow[10] ([91], p.6).

[9] No desktop programmer handles keyboard or mouse input on the physical or data link layer.

[10] One may think of drones and autonomous cars.

In a way, this observation defends the current practice to avoid non-functional abstractions for the sake of optimized, efficient code. However, the increasing complexity of embedded systems challenge this practice not so much on a technical, but on an economical level:

On the one hand, with current techniques and methods complex projects are far more expensive than simple ones, but, on the other hand, more memory and faster processors are always more expensive than smaller and slower ones. So, assuming that having resources available for "more abstraction" makes programming more efficient, the trade-off is between the recurring cost of computing hardware and the non-recurring cost of programming the system.

Figure 2.16 illustrates the issue with a simple example. The graph shows the cost per unit for a man-year of software development (roughly estimated at 100000 €). So, if for example 10 units are to be produced, spending 10000 € per unit on hardware to reduce programming effort by a year is still beneficial. For 100000 units, the limit is 1 € per unit, which sounds small, but may still mean some more megabytes of memory or a faster processor. This observation is especially true for spacecraft software, where unit numbers are very low ([36], p.39).

To conclude, spending mony on more potent hardware to improve programmer possibilities may pay off immediately: It allows to introduce abstraction and generalisation which brings down development and maintenance cost.

Overhead of Object-Oriented Programming

The concrete overhead of object-oriented programming (OOP) is the source of a heated, still ongoing debate (e. g. [65] or [66]). As embedded software is mainly programmed in C or C++ [102], the discussion is also focused on these languages.

As argued in [66] or [77], the technical overhead for most C++ features, such as class structures and non-polymorphic inheritance is zero. Main exception is the dynamic dispatch mechanism of subclasses (see Section 2.1.4), which introduces a certain memory and execution overhead [35] by providing runtime type information (RTTI) or type introspection features. Without them, dynamic dispatch and therefore interface programming doesn't work. When implementing them in a procedural language, it is most likely as expensive as the OO implementation. Also, if not used, dynamic dispatch will not consume any resources. So, the question is less how expensive C++ is compared to C, but if the additional features are useful or not for a given project, just like any sort of abstraction introduced.

Another point of discussion is that of dynamic memory allocation, which means creating new structures or objects at run-time. Using dynamic allocation is discouraged in embedded projects for two reasons:

- Allocating memory may take an arbitrary amount of time as it involves looking for free memory on the heap. Thus, it introduces uncertainties regarding execution time and therefore real-time capabilities of the system
- Due to fragmentation of memory, allocation for a certain size may fail even if the total amount of space is sufficient. This is a root cause for difficult-to-track faults in a system.

Using C++ does not automatically introduce dynamic allocation, class hierarchies and interfaces can be built without allocating any memory. Some libraries however, most notably STL containers, use it and therefore should be used cautiously in embedded systems.

To summarize, it is possible to write embedded software in C++ just as efficient and real-time capable as in C. The main aspect of using C++ however is not covered by resource consumption comparisons: As an OO language, C++ allows totally different patterns and paradigms for programming than procedural languages. Applying these paradigms has up- and downsides: On the downside, OOP with C++ is more subtle and complex than procedural programming in C, but mastering the details down to machine code is equally important [66]. This introduces non-technical and often overlooked cost to educate programmers in OO techniques. On the upside, applying the generalization and abstraction capabilities allows representing dependencies between software elements directly in the code, and not only in some programmer agreement. In a way, OOP and therefore C++ helps representing *software architecture* on source code level.

2.4.4. Spacecraft Flight Software

Spacecraft flight software (FSW) is embedded software with a very special heritage. The foundations were laid in the software development programme for the Apollo Guidance Computer, which worked on the limits of what was technically feasible at that time and faced similar challenges as other early software engineering endeavours [101].

Given its origins from human space missions, methodologies to develop FSW have a lot of influence from the safety-critical domain (such as aircraft control software), i. e. plan-driven methods are typically applied (see Section 2.2). This is also true for robotic spacecraft, were high monetary and human effort as well as scientific expectations replace the immediate threat for human life.

In an abstract sense, spacecraft can be characterized as *remote-controlled semi-autonomous embedded control systems.* This characterization specifies the main tasks of a FSW:

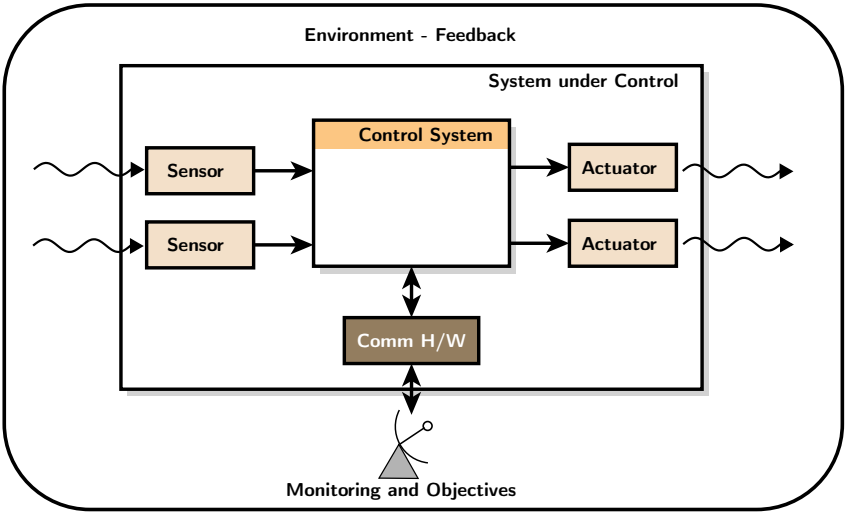

Figure 2.17.: Principle layout of a semi-autonomous embedded control system.

- Most important task of a spacecraft FSW is **control**, i. e. providing commands to change some state in accordance with the current system objective. Control implicitly distinguishes between a *control system*, which is typically implemented in software, and a *system under control*, which may include software, hardware, and the environment ([91], p.34). Figure 2.17 illustrates the concept. Most obvious example is attitude control of a spacecraft.

- Being an **embedded system**, determining and controlling the system's state requires sensors to interpret the physical environment and actuators to adjust that state. Thus, FSW handles interpretation of sensor data and preparation of actuator commands, as well as management of both sensor and actuator hardware. Example hardware are star sensors and reaction wheels.

- A **semi-autonomous** system must perform its control tasks, but also change objectives, without immediate supervision by a human operator. The former requires some awareness of the control system's own state, especially to account for faults in the system. The latter requires some form of scheduling or sequencing mechanism, to allow out-of-sight changes of control goals.

- Apart from crewed missions, spacecraft are **remote-controlled** by nature. Beside the immediate effect of having to manage communication

hardware, which is merely another control task, the most important impact on FSW is the need for remote monitoring and maintenance capabilities. Especially the latter point is challenging, as maintenance includes that of the software itself.

With about sixty years of successful space missions, the above tasks are supposed to be well understood and therefore FSW development should be relatively straightforward. However, there are some indications that this is not the case. For instance, NASA conducted a study on FSW complexity in 2009 [36], which shows that code size, as a hint for complexity, is rising continuously since the time of the Apollo programme (see Figure 2.18). As indicated in Sec-

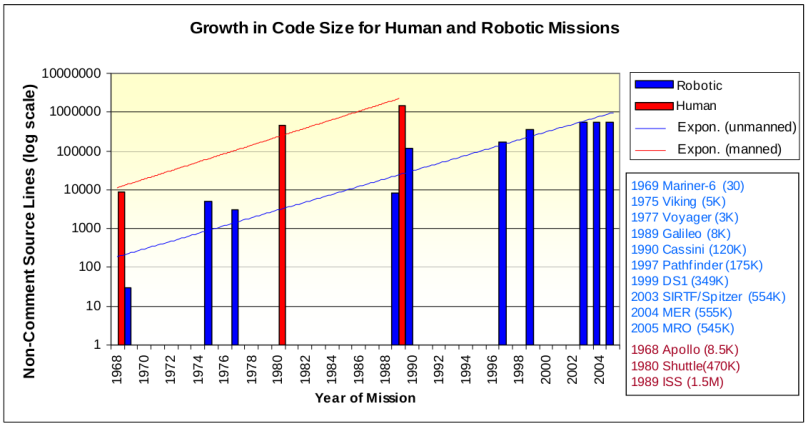

Figure 2.18.: History of flight software growth in human and robotic missions [36].

tion 2.4.3, one reason for growing complexity are greater ambitions regarding autonomy and control. Indeed, for recent deep space missions such as Rosetta or the Mars Science Laboratory, complex control software is an enabling technology. For earth observation or telecommunication spacecraft, this may be less true, but cost pressure, especially for proposed mega-constellations, will still call for enhanced spacecraft autonomy to improve availability and reduce operational cost.

In effect, FSW will most likely continue to become more complex. Denying this trend and resorting to the KISS[11] principle, in the sense of a software with low abstraction, will not help in managing complexity. Instead, it is likely that incidental complexity (see Section 1.1) grows if not kept under control by principles of separation of concern and information hiding.

[11] Keep it simple, stupid!

Thus, the main goal of this thesis is to test the presented software engineering techniques in a spacecraft FSW project and evaluate their benefits.

3. Flight Software Domain Analysis

Flight Software (FSW) is likely to become both more complex and important in the near future, which calls for some strategy to avoid delivery delays and cost overruns due to late software. All techniques presented in Section 2.3 are promising candidates to ease spacecraft FSW development, either by improving reusability of the code, or by introducing abstractions which improve separation of concerns.

However, before selecting a certain set of technologies, it is necessary to figure out *what* it actually is that needs to be designed. Thus, this chapter will perform a *domain analysis* for FSW. As described in Section 2.3.5, the main goal of such an analysis is to find and assess all *features* needed for software of a certain domain. The approach is quite similar to a requirements analysis, but instead of collecting the needs of a single product, those of an entire product family are taken into account.

The resulting *domain model*, which collects and orders all features of the domain, is intended to guide the development of a generic software product or a *software product line*. Moreover, it is a good starting point to identify reusable elements and suitable abstractions for the design.

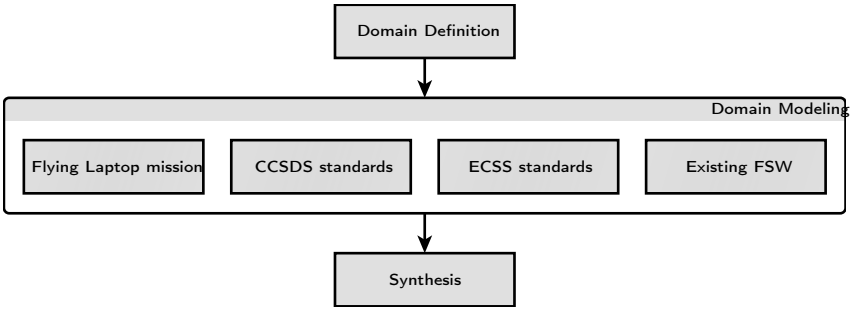

Figure 3.1.: Workflow of the domain analysis performed for FSW.

Methodically, the approach defined in [30] is used, which is illustrated in Figure 3.1.

- The *domain definition* in Section 3.1 sets the scope of the analysis by defining the content of the FSW domain, its boundaries and main elements.
- With that top-level definition, the domain modeling phase is performed in Section 3.2 and following. It requires a survey of existing domain knowledge, which is extracted from available standards, and the author's own working experience. Also, existing similar approaches for reusable satellite software are analyzed and evaluated.
- The resulting requirements and features[1] are ordered and synthesised in Section 3.7. To illustrate the domain model, a feature diagram as described in Section 2.3.5 is used.

The resulting set of features are the starting point for designing a flexible, reusable flight software.

3.1. Domain Definition

Spacecraft flight software is a software to manage a remote-controlled partially autonomous embedded control system, which happens to operate beyond earth boundaries. These four findings form the basis of the flight software domain definition. Some basic requirements for such software have already been outlined in Section 2.4.4. The following list derives the top level FSW features from these requirements:

- *Controlling* the spacecraft requires to execute multiple control algorithms, as well as other tasks, in parallel. Organizing and supporting these activities is summarized under the term **component management**, which includes real-time schedulers or data exchange services.
- As an *embedded system*, FSW needs to provide features to handle *equipment*, which in turn requires to manage *on-board communications*. A space system compromises multiple control domains, or *subsystems*, in parallel. Features in that category are summarized under **system management**.
- Many features of a FSW are dedicated to *remote control* or **operations**. Aside from basic monitoring and control needs, features are required to support on-board *maintenance* of the spacecraft.
- Aside from remote operation capabilities, almost every FSW requires certain **autonomy** features. A common autonomy feature is *fault management* software. In addition, spacecraft software typically provides some kind of *scheduling* or sequencing features.

[1] Due to their similar definition for software, the words "requirements" and "features" are used synonymously in this chapter.

All identified features will be categorized and added to one of these top-level findings.

3.1.1. Differentiation from Neighbouring Domains

Flight software, given the definition in Section 2.3.5, is a vertical domain, i. e. it covers a family of similar target applications. Thus, there are a number of neighbouring domains with similar feature requirements. To perform a proper differentiation, it is useful to re-arrange the previous findings to the form shown in Figure 3.2.

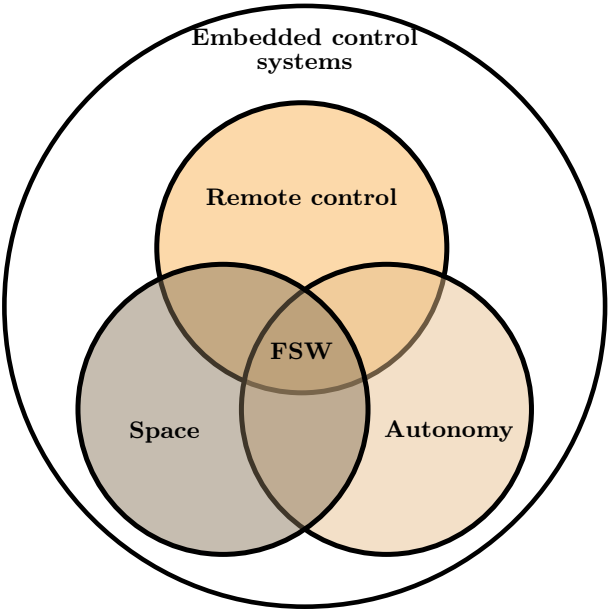

Figure 3.2.: Illustration of the flight software domain.

The embedded control software domain is a superset of the other domains[2]. From the illustration, it is easy to define neighbouring domains by omitting

[2]Even though both non-embedded autonomous systems and space systems may exist, they are rather exceptions than the norm.

one of the attributes of FSW, as indicated by the intersections of two of the three circles. Thus, there are three neighbouring domains:

1. First, there are autonomous space systems, which are not remote controlled, e. g. manned spacecraft. These systems differ in requiring human-machine interfaces for controls and displays. Reliability requirements are typically more strict, given the immediate threat for human life. Still, maintenance and autonomy requirements may be relaxed, as in situ repair is possible.

2. Software for autonomous, remote-controlled vehicles probably forms the closest neighbouring domain to FSW. Apart from remote maintenance capabilities, the requirements for e. g. UAV software are quite similar to those of a spacecraft. Other robotic vehicles, such as autonomous cars, also have similar requirements, however, the need for remote control is even less strong.

3. The last category are remote-controlled space systems with no or very little autonomy. This is a rather small category, which may contain rockets and geostationary satellites, as these systems are under direct ground supervision during most of their mission time. While remote monitoring and control needs are similar, omitting autonomy reduces the need for system state awareness within the software itself.

In effect, this domain analysis focuses on the very specific needs of spacecraft flight software, even though it is tempting to cover other neighbouring domains or even embedded control systems in general. However, a useful software product must provide what a software developer of that domain needs. If the scope is too wide, it will likely be too general to provide any benefit[3]. Therefore, the target entity for which the software is intended will be called a "spacecraft" in the rest of this thesis, meaning an unmanned satellite or space probe and excluding manned spacecraft or rockets. A successful software product may be extended to cover additional domains in the future.

3.2. Domain Modeling

The following sections summarize a survey of existing sources containing requirements or features for a flight software product line. As many sources contain features for multiple of the top-level categories identified in Section 3.1, the survey will be ordered by types of sources first. Each source will be summarized and identified features will be listed. The resulting feature tree for each top-level category will be presented at the end of the survey.

[3] It is too costly to instantiate it, i. e. it is too far on the right in Figure 2.14 of Section 2.3.8.

The first source is the *Flying Laptop* project, a micro satellite mission executed at the University of Stuttgart. The spacecraft mission profile and therefore essential complexity (see Section 1.1) is representative for small satellites. Its design and the resulting FSW requirements are listed in Section 3.3. To avoid a bias from specific technical solutions of *Flying Laptop*, some aspects are put in relation to other space missions. This comparison and the impact on a generic FSW are described in Section 3.3.6.

Another important set of sources for this study are space engineering standards, namely Consultative Committee for Space Data Systems (CCSDS) and European Cooperation for Space Standardization (ECSS) standards. The former standards are internationally agreed space communication standards, the latter are a set of European standards mainly maintained by the European Space Agency (ESA). They are surveyed in Section 3.4 and Section 3.5, respectively.

The third source for the survey is existing flight software. Most FSW code and its documentation is an industrial secret. However, there are a number of commercial and research project, e. g. by ESA or the National Aeronautics and Space Administration (NASA), for which enough information is available. This analysis can be found in Section 3.6. The domain modeling phase concludes with a synthesis of the survey in Section 3.7.

3.3. The *Flying Laptop* Mission

Flying Laptop is the first satellite mission within the small satellite programme of the University of Stuttgart. It is an earth-observation, technology demonstration and educational mission, which was launched successfully on July 14$^{\text{th}}$, 2017 to a 600 km sun-synchronous orbit.

Aside from its representative hardware and mission profile for low earth orbit (LEO) missions, *Flying Laptop* is chosen as reference due to the author's own involvement as software developer in the project. As stated in Section 2.3.5, experience is a decisive factor for a successful domain analysis and definition.

This section will provide an overview of the mission and the spacecraft subsystems, with a focus on elements which are relevant for the spacecraft software. From that description, a set of general requirements for FSW are derived.

3.3.1. Payload and Technology Demonstration

For its main earth observation mission, *Flying Laptop* utilizes three instruments:

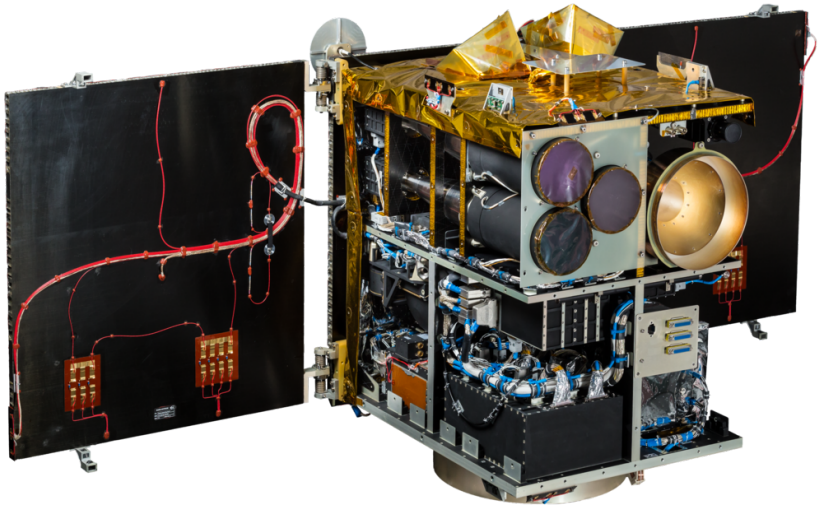

Figure 3.3.: The small satellite *Flying Laptop* before attaching the shear panels.

- The primary imager is the **Multispectral Imaging Camera System (MICS)**, which was designed for bidirectional reflectance distribution function (BRDF) measurements of vegetation in three color channels (upper left compartment in Figure 3.3).
- The system utilizes a secondary imager, the **Panorama Camera (PAMCAM)**, which is used mainly for public outreach (upper right corner in Figure 3.3).
- In addition to the camera systems, an **automatic identification system (AIS)** receiver supports space-based ship tracking.

As a secondary science goal, the satellite's star tracker unit is tested to serve as detector for near-earth asteroids. Further details on the instruments and their development are found in [15] and [106].

Data management for the payload is done by the FPGA-based payload on-board computer (PLOC), which was developed for *Flying Laptop* and manages data acquisition and storage from the instruments [64]. It also handles a dedicated payload downlink, the data downlink system (DDS) (horn antenna in upper right section on Figure 3.3).

In addition, *Flying Laptop* hosts multiple technology demonstrations, most of which are innovative bus components, such as an innovative on-board computer

(OBC), or a star tracker unit augmented by an inertial reference unit (IRU). Another experiment is the optical high speed infrared link system (OSIRIS), which aims to demonstrate direct data downlink via infrared lasers.

3.3.2. Spacecraft Bus

A spacecraft bus exists to provide the necessary infrastructure, such as communications, power or a certain attitude, for the payloads to reach the mission goals. There is a functional decomposition of the *Flying Laptop* bus into several subsystems. Each subsystem constitutes an embedded control system (see Section 2.4.4) with its own set of sensors, actuators and control algorithms. The spacecraft FSW interfaces with every subsystem, as most control tasks are executed within software on the main computer. An overview of the system on a logical interconnection level is found in Figure 3.4.

One requirement for the *Flying Laptop* bus on system level is single failure tolerance, i. e. a fault in any part of the system shall not result in a complete mission loss. Thus, different types of redundancies are implemented in the system, which need handling by the FSW.

A typical redundancy scheme used in *Flying Laptop* is dual redundancy with one element either powered or not, i. e. *hot* or *cold redundancy*. In such a configuration, not every faulty sensor can be clearly identified autonomously[4]. To keep the system stable, a typical reaction is a transition to a safe mode with reduced sensor and actuator demand, which requires human intervention to restore nominal operations. This concept reduces availability of the system, but avoids overly complex and expensive systems, which would be necessary to allow autonomous failure recovery in all cases. Further details on the system's fault management and redundancy schemes can be found in [109].

Following, a short overview of the satellite bus subsystems is provided.

Command and Data Handling Subsystem

The command and data handling (CDH) subsystem is responsible for data management and command distribution. Also, it is the central element to interface sensors and actuators of all other subsystems. For *Flying Laptop*, a central OBC serves as main computing and interface unit for all bus components (see Figure 3.3, lower left compartment, and Figure 3.4, center). The computer, which was developed in collaboration with Airbus Defence and Space, consists of four types of boards, each existing twice for redundancy reasons:

[4] To clearly identify a deviating sensor, three identical sensors are necessary to overrule the faulty one.

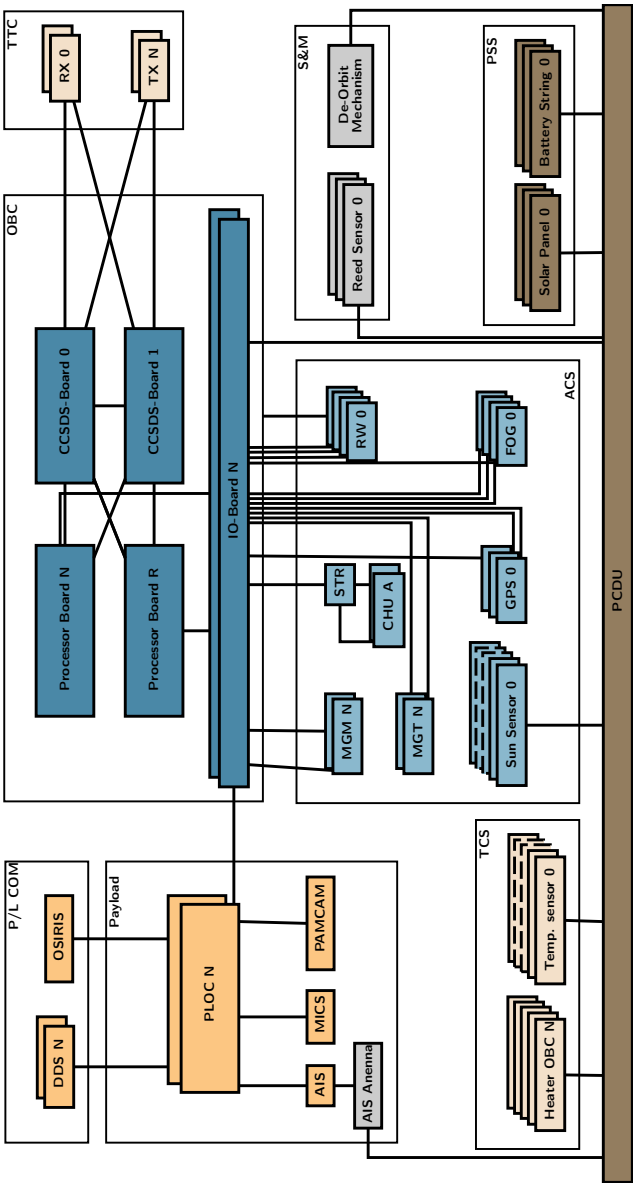

Figure 3.4.: Simplified overview of *Flying Laptop* subsystems and their logical connections. Cross-couplings of stacked elements are omitted.

- A **processor board**, which holds the main processor, a LEON3-FT[5] UT699 microprocessor, and provides four SpaceWire[6] interfaces to connect to the other boards. The processor is clocked at 33 Mhz and provides 8 MiB of EDAC-protected memory.
- A so-called **IO-Board**, which serves as a multiplexer to interface other bus components. Also, it provides memory for telemetry and state vector storage.
- The **CCSDS-Board** serves as a decoder/encoder board for the ground-space link, in accordance with CCSDS standards (see Section 3.4.1). It connects the OBC to the units of the TTC subsystem.
- The **OBC power board** mainly provides regulated power for the other boards, and routes some dedicated signal lines.

A switchover of the cold redundant processor board in case of a hardware fault or a software error is handled by the power management unit, forming the patented Combined Data and Power Management Infrastructure (CDPI) [53]. It is initiated either autonomously utilizing a watchdog functionality or by a dedicated high-priority command to the power management unit, bypassing the processor board. The CCSDS-Boards execute in hot redundancy, to allow reception of TCs even in case of a failure of one board.

The flight software runs on the processor board and therefore depends on its resources and reliability. Most of the fault tolerance features of the LEON3-FT processor are implemented in hardware, providing a high level of confidence in the integrity of FSW execution.

Power Supply Subsystem

The power supply subsystem (PSS) serves two main purposes: Power conditioning, i. e. providing sufficient power from batteries and solar arrays at all times, and power distribution, i. e. providing controllable switches to supply all bus components and payloads. Main element of the PSS system is the power control and distribution unit (PCDU), which, in addition to the above functionality, serves as a reconfiguration unit for the satellite [52] and digitizes analog sensor values, e. g. from temperature sensors [103].

To serve as reconfiguration unit, the PCDU is designed to operate at a high level of autonomy, e. g. evaluating temperature ranges and battery charge status before starting the OBC. Still, the FSW executes the main control and monitoring algorithms for PSS as soon as it is started, and, for example, calculates an improved state of charge.

[5] FT stands for fault tolerant.

[6] *SpaceWire* is a spacecraft communication network standard managed by ESA, see Section 3.4.3.

Attitude Control Subsystem

Due to their small size and mass, it is relatively easy to build small satellites with high agility, which allows off-path observation to increase revisits of a certain area. *Flying Laptop* makes use of this ability to provide multi-angle imagery of a given target. Also, the MICS camera system and especially the OSIRIS experiment define challenging requirements for pointing accuracy of the attitude control subsystem (ACS). Moreover, a basic sun pointing ability is critical in contingency situations, as energy is generated only if the solar arrays are pointed towards the sun. All possible ACS modes of *Flying Laptop* are illustrated in Figure 3.5.

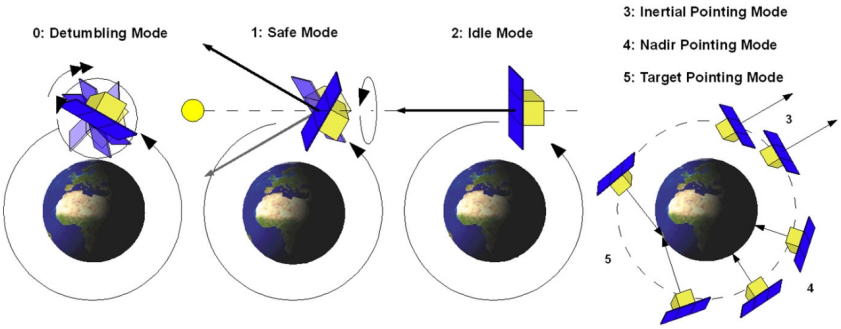

Figure 3.5.: Illustration of *Flying Laptop* ACS modes.

To achieve these requirements, *Flying Laptop*'s ACS operates in two different modes: A *safe mode* using a very basic algorithm, as well as simple, but robust sensors and actuators, namely sun sensors (SUSs), magnetometers (MGMs) and magnetic torquerss (MGTs). On top of that are the *pointing modes*, with more advanced sensors, actuators and algorithms to enable precise pointing of the spacecraft and complex maneuvers, such as target pointing to a ground station. For the pointing mode, the spacecraft additionally uses a star tracker (STR), fibre-optic gyros (FOGs), and GPS receivers as sensors, as well as reaction wheels (RWs) for actuation.

In effect, the ACS is the most complex subsystem, with a wide range of sensors and actuators, each of which has a dedicated redundancy scheme.

To make best use of the sensors and actuators, the FSW needs to execute calculation-intense navigation and control algorithms, which also require on-orbit tuning of parameters.

Further information on the ACS system is found in [63] and [112].

Other Subsystems

There are three more subsystems of the *Flying Laptop* bus, each of which is a dedicated embedded control system. Interaction with these subsystems and FSW is similar to that of PSS or ACS, however, processing demand and number of associated equipment is typically lower.

- **Telemetry, Tracking and Control Subsystem:** This subsystem manages the radio frequency (RF) physical layer, or OSI layer 1, of the space link with a pair of receivers and transmitters. With information obtained from that equipment, FSW performs certain control operations, e. g. activating a transmitter as soon as a signal is received.
- **Thermal Control Subsystem:** The TCS ensures all bus and payload components remain in their operational temperature ranges during the mission. *Flying Laptop* uses an actively controlled, cold-biased system, which mainly works with passive insulation and radiator surfaces, but provides additional temperature control by using electric heaters [97]. A range of temperature sensors are used to measure component temperatures, in conjunction with internal temperature sensors provided by more complex equipment. The thermal control algorithm is executed in FSW.
- **Structure and Mechanics:** The S&M subsystem ensures mechanical stability of the spacecraft during assembly, launch and in orbit, and hosts movable parts such as deployment mechanisms. Most important is the solar array deployment mechanism, other mechanisms are the AIS antenna and an experimental drag sail to speed up deorbit of the satellite [79]. Using the mechanism is initiated and monitored by the flight software.

3.3.3. Ground Segment and Operations

A satellite, as a remote-controlled entity, is useless without a well-matched ground segment, which makes maximum use of the system. For *Flying Laptop*, the ground segment infrastructure evolved in parallel with the satellite. To fulfil the educational goal and train students for large-scale missions, agency and industry standards and products were applied where appropriate, creating a ground software with professional and custom-made elements (see Figure 3.6) [75].

Using ECSS and CCSDS standards for the space link has significant impact on the flight software, as they define detailed requirements with regards to *commandability* and *observability* of a spacecraft. These are discussed in dedicated sections 3.4 and 3.5.

Operating a satellite at an University institute has some additional impact on the FSW:

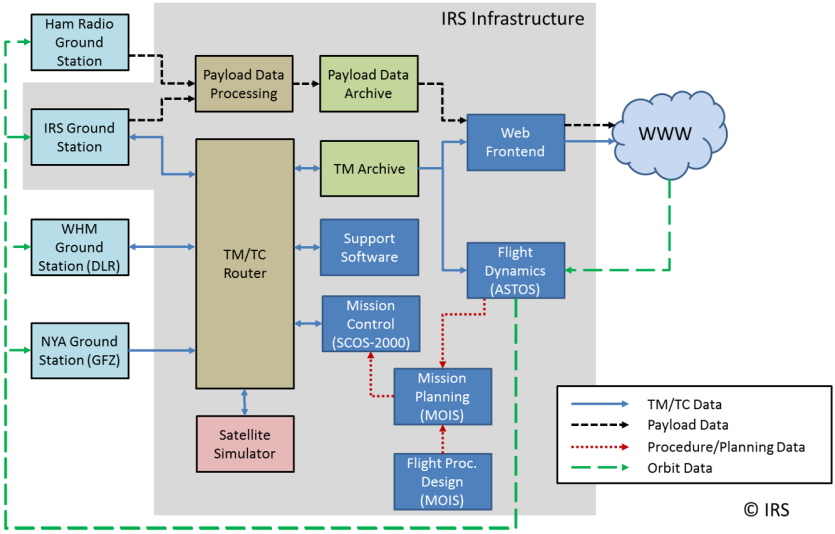

Figure 3.6.: Ground segment overview block diagram (from [75]).

- Due to the limited number of ground stations, the spacecraft must be able to **execute pre-planned tasks** out of ground station visibility[7].
- In addition, there are no resources to ensure staffing for 24/7 shifts during nominal operations. In effect, the spacecraft shall **survive** longer periods of time **without operator interaction**, even in case of an on-board failure.
- Spacecraft operations are conducted by non-professional operators and students. Therefore, **commanding** the satellite shall be **simple and safe**.

In summary, the operational environment demands a robust operability design from the FSW and a certain amount of on-board autonomy and failure handling capabilities.

3.3.4. Software and System Testing

To ensure proper operation of *Flying Laptop* in space, a thorough test program is required. This does not only cover functional verification of the integrated spacecraft, but early functional tests on component and subsystem level. For

[7] I. e. it needs ESA mission execution autonomy level E2 at least (see Section 3.5.3).

example, subsystem compatibility with the OBC was verified in a so-called *flatsat* campaign (see Figure 5.2). Also, flight software testing was conducted early on using a system testbed (STB) as described in [50]. Further information on the system and software testing campaign can be found in [12].

For these tests to be executed successfully, capabilities of the FSW need to be readily available for the test campaigns. This implies the need for a modular software design and incremental development, which provides growing functionality in line with system test milestones [14]. Also, it must be possible to disable any kind of autonomous activity on-board, to avoid interference with system tests.

3.3.5. Resulting Flight Software Features

This section will identify features any flight software needs on the basis of the *Flying Laptop* mission.

Payload

As the exact nature of a payload is inherently mission specific, it is somewhat difficult to derive generic requirements from the *Flying Laptop* payload. However, in distinction to the basic control loops of the satellite bus, many payload operations are short-term activities, such as taking a picture with the cameras of *Flying Laptop*. The FSW therefore must provide means to trigger such an *action* of a software component, such as the camera driver:

Feat.: FLP.1 Action execution
Allow to trigger a dedicated, finite activity of a software component.

Other requirements, such as maintaining the temperature of a camera, or checking a camera's health, are either derived from support activities of the bus, or are discussed in the context of generic requirements of satellite equipment.

Control Algorithm Execution

For the satellite bus, the main and most obvious software feature is: Execute control algorithms. While this is certainly true, it is also rather superficial. Still, it is a good starting point to define more precise requirements.

First of all, a FSW needs to manage *concurrent execution of tasks*, and allowing to define the cycle intervals for each control algorithm. These duties are typically provided by a real-time operating system (RTOS) (see Section 2.4), but providing access to those tasks and management may be part of the software.

Moreover, the FSW needs to provide means to *deliver sensor data* from the software part handling the equipment to the control algorithm and forward generated commands to actuator handling software. Due to discrete and cyclical execution, these in- and outputs need cyclical distribution as well.

Besides periodic data, satellite's control algorithms use a variety of *parameters*, such as gains or system constants, which need only occasional adjustment or fine tuning.

Finally, some controllers, such as the *Flying Laptop* attitude controller provide a number of dedicated *modes*, which alter their behaviour. Also, it may be useful to disable any control activity manually in contingency and testing situations.

The following list sums up these core features:

Feat.: FLP.2 Cyclic execution
Allow software elements to execute cyclically and concurrently at different rates.

Feat.: FLP.3 Periodic data distribution
Distribute sensor measurements and actuator commands within the software.

Feat.: FLP.4 Parameter access
Allow external access to read or update parameters in software components.

Feat.: FLP.5 Controller modes
Allow adjusting the permanent cyclic behavior of a control algorithm and eventually disabling it.

Spacecraft Equipment

Regarding sensors and actuators, *Flying Laptop* is a good reference for a LEO satellite, as it is equipped with the typical range of sensors and actuators for such a mission[8]. Managing this equipment is one of the challenging tasks of a flight software.

To illustrate the issue, consider the *Flying Laptop* reaction wheel (RW) assembly. RWs produce torque by changing the rotation speed of a flywheel. The *Flying Laptop* ACS is equipped with four such wheels in a tetrahedron configuration for redundancy reasons (see Figure 3.7). Each reaction wheel is independently connected to the OBC's IO-Board.

Communication with a single wheel takes place via a serial connection and a vendor-specific protocol to command the wheel and read out telemetry (TM) data.

This example illustrates the challenges of equipment communication:

[8] Disregarding means for orbit control, such as a maneuver engine.

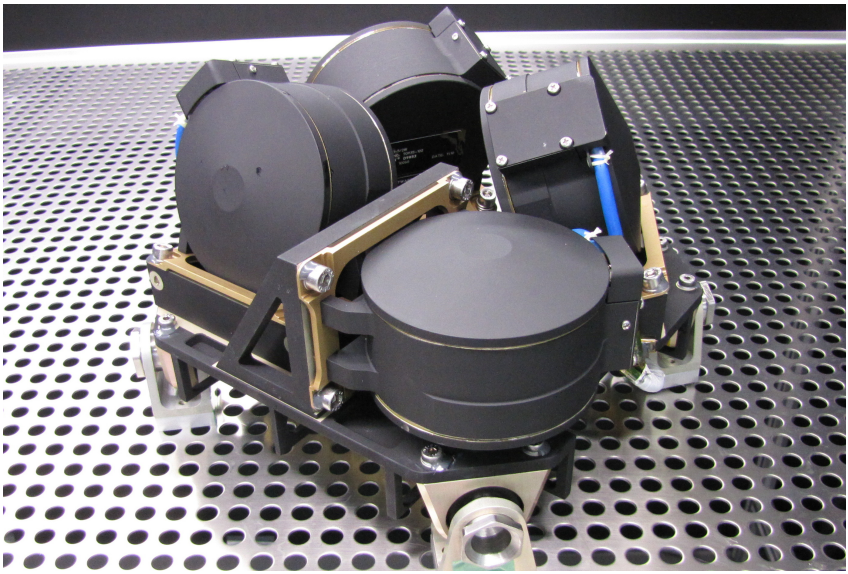

Figure 3.7.: The reaction wheel assembly of *Flying Laptop*.

- Hardware vendors tend to devise dedicated application protocols (OSI layer 7). Thus, a framework can not provide certain protocols out-of-the-box, as there are too many variants.
- Similarly, there is a large number of interface types, such as the RW UART, which are defined by the vendor of a specific equipment, and not by the customer.
- Moreover, equipment communication is often indirect and the actual commands or responses are relayed with a very different protocol. Most equipment of *Flying Laptop*, like each individual RW, is connected to the IO-Board, which by itself is controlled via SpaceWire from the main processor board.

To manage these variants, a reusable FSW shall provide means of *layering* between lower level protocols on the communication media and equipment application protocols. This simplifies combining software components and a specific communication hardware architecture. This would, for example, allow to reuse the software logic to control the wheels even if they are connected to the processor board with a bus system.

Besides enabling communication, the main duty of equipment handling software

is to read out sensors and command actuators and exchange these values with control algorithms. To do so, it is in most cases necessary to *poll sensor devices* or forward actuator commands at regular intervals, for example to command RW torques.

For almost all spacecraft, not all sensors are needed in all system modes, therefore, *equipment modes*, which allow deactivation and initialization of devices are reasonable. Ideally, this includes switching of power lines.

So, the first set of equipment-related software requirements are:

Feat.: FLP.6 Communication Layering
Provide means to separate the low level communication media from the applications which communicate.

Feat.: FLP.7 Polling and Commanding
Fetch sensor data and forward actuator commands at regular intervals.

Feat.: FLP.8 Equipment modes
Provide means to enable and initialize, but also disable some equipment.

Another important topic is equipment monitoring, as outtakes can happen due to random faults or the space radiation environment. If the equipment does not respond anymore, or the packet is obviously incorrect, a equipment error is a likely cause. Thus, a software framework should provide *communication monitoring*, at least for digital sensors and actuators.

Sensor value monitoring is relevant for simple sensors as well. For example, a simple temperature sensor can be monitored by checking its value against absolute limits and eventually the temporal gradient of its output. Thus, value monitoring support is another framework feature.

In case some equipment is found to be not working anymore, it must be disabled, or at least its inputs must be ignored. To make this explicit, a FSW can manage equipment *health states*.

Another, related issue is management of redundant sensors and actuators. For *Flying Laptop*, there are redundancies for all sensors and actuators on-board. For example, the tetrahedron configuration of the RW assembly shown in Figure 3.7 allows a three-out-of-four redundancy scheme, i. e. the system works well even with one wheel disabled. While many aspects of *redundancy management* are mission specific, a FSW may provide support to manage various redundancy schemes.

In summary, the following features are found for monitoring and redundancy management:

Feat.: FLP.9 Communications monitoring
Support the detection of communication errors to monitor equipment.

Feat.: FLP.10 Equipment value monitoring
Provide means to monitor sensor and actuator values and report monitoring violations.

Feat.: FLP.11 Health states
Allow equipment to be tagged non-functional to avoid using it in the future.

Feat.: FLP.12 Redundancy management
Support the implementation of specific redundancy schemes for sensors and actuators.

Operations

The specific challenges of operating *Flying Laptop*, as described in Section 3.3.3, directly outline a number of requirements for FSW:

Feat.: FLP.13 Command scheduling
Allow out-of-sight execution of telecommands (TCs).

Feat.: FLP.14 Survivability
Enable spacecraft survival by detecting faults autonomously and ensuring the spacecraft is in a safe state until operator intervention.

It is more difficult to translate "simple and safe" commanding into a FSW requirement. One approach is to reduce and generalize the number of commands an operator can use, as unification is a prerequisite for simplicity:

Feat.: FLP.15 Common commanding
Provide unified, simple commands to operate different software components of the spacecraft.

Another aspect regarding operations is the functional decomposition of the *Flying Laptop* system into a number of *subsystems*, as described in Section 3.3.2. To simplify operations, it may be useful to represent this subsystem hierarchy in software, especially when speaking of system modes. For example, it would improve visibility if there were a software component, which indicated and controlled the mode of the entire attitude control subsystem.

Feat.: FLP.16 Subsystem representation
Represent the functional decomposition of the system in software and allow operator interaction with these representations.

Testing

A software design for testability is especially important for spacecraft development, not only on software unit, but also on subsystem and system level. Experience from *Flying Laptop*, as described in Section 3.3.4, shows that a modular software design, e. g. with software components which are controllable individually, supports system-level testing early on. For example, the flatsat testing campaign was executed only with equipment handler software running, most control algorithms were not available yet. As these software components can be used unchanged in the final software, confidence in their suitability is high after the tests.

The aspect of disabling controllers for tests has been covered in the control algorithm section above.

Feat.: FLP.17 Software Modularity
Create software components which work at a maximum level of independence.

3.3.6. Comparison to other Space Missions

Given the vast range of spacecraft types and mission profiles, it is likely that some aspects of spacecraft flight software are not found by looking at *Flying Laptop* only. Still, none of the found requirements describes functionality specific to that mission or the *Flying Laptop* bus, indicating that the found features are generally applicable.

However, to ensure no relevant aspect is omitted, this section discusses some variants of other missions for comparison with *Flying Laptop*.

Nano and Large Satellites

Flying Laptop, as a micro satellite, is in terms of size right in the middle of a common satellite classification scheme (see Table 3.1). In terms of its subsystem and equipment structure, as well as operational concepts and redundancy schemes, it is more comparable to larger satellites, even though certain subsystems may be missing.

To check this issue, one can take a look at a 1995 report from the US Air Force, which performed a domain analysis on spacecraft FSW [57]. The main findings are that every spacecraft consists of a number of typical subsystems, of which only those that are jointly referred to as orbit control subsystem (OCS) are missing in *Flying Laptop* (see Appendix B.1 for a graphical summary).

Even though the impact of utilizing a propulsion system is significant on overall spacecraft level, it is yet another control system for FSW. The generic software

Group name	Mass (kg)
Large satellite	> 1000
Mini and Medium satellites	100 to 1000
Micro satellite	10 to 100
Nano satellite	1 to 10
Pico and femto satellites	< 1

Table 3.1.: Classification of satellites by size (from [76]).

functionality can not handle the exact type of a controller and is therefore responsible for the infrastructure aspects only. The same argument applies to other types of payload: In most cases, they do not affect the generic software infrastructure.

Also, there may be requirements missing for smaller spacecraft, i. e. nano satellites. These typically come in the form of *CubeSats*, which are spacecrafts with a standardized outer shape of one or more cubes of 10x10x10cm^3 each. These satellites feature highly miniaturized, simplified subsystems, with typically no or very limited redundancy.

Even though the subsystem structure of CubSats is conceptually similar to that of larger missions, miniaturization allows different technologies to be used. For example, on-board networks in CubeSats often use SPI or I^{2}C buses, which are intended for short-distance communications only. Also, simpler space link protocols such as AX.25 are used [111].

For a FSW to cover these aspects, the following requirements may be considered:

Feat.: **FLP.18** **Embedded serial buses**
Support embedded on-system buses for equipment communication, such as SPI or I^{2}C.

Feat.: **FLP.19** **AX.25**
Support amateur radio protocols for the space link.

On-Board Computer Architecture and Equipment Hardware

There are no standardized on-board computer architectures in spaceflight. Instead, many hardware setups are custom configurations for a specific mission. However, the high-level requirements on FSW are often identical. Therefore, a FSW product should provide a certain amount of hardware abstraction to allow implementing controllers and functionality independent of the actual execution platform.

Enabling portability is significantly more difficult in cases where computing is distributed among multiple, homogeneous nodes[9]. In such cases, as in [3], the framework needs to ensure proper communication between distributed parts of the overall software, i. e. provide a *middleware*.

Feat.: FLP.20 Hardware portability
Provide functionality for applications such that it is independent of the underlying computing hardware.

Feat.: FLP.21 Distributed computing
Allow parts of the application software to be executed on homogeneous interconnected computers, and hide the details of communication.

Just like different computer architectures, other main subnetwork protocols to interface spacecraft sensor and actuator equipment exist than SpaceWire, such as CAN or MIL-STD-1553B buses. These aspects are covered in detail in Section 3.4.3.

3.4. CCSDS Standards

As spacecraft are remote-controlled entities, communication between the system and its remote controller, i. e. the ground station, is fundamental. To avoid incompatibilities between different actors in the space community, the Consultative Committee for Space Data Systems (CCSDS) defines communication standards which ensure interoperation between spacecraft and ground systems. There are protocol definitions for all OSI layers, from the physical RF layer to application layer operation services.

The importance of the CCSDS suite of space standards stems from the fact that they are truly international standards, which are intended to ensure cooperation between different space agencies for satellites and space probes.

Even though many aspects of that domain, such as RF engineering, are not relevant for software, there are a number of documents dealing with spacecraft operations containing important recommendations for a generic FSW. Moreover, CCSDS extended its role over time and released more standards with relevance for FSW, e. g. on-board communication.

This section scans all CCSDS standards for relevant features of a generic FSW.

[9] If the nodes were inhomogeneous, one would be treated as master and the rest is merely "smart" equipment on the subnetwork.

3.4.1. Frame and Packet Protocols

Ground-to-space, space-to-ground, or space-to-space communications are often summarized as a *space link*. One set of standards provided by the CCSDS deals with communication over the space link on frame and packet level, i. e. on OSI layer 2 and 3 (see Section 2.4).

The frame level, as defined by CCSDS, is divided in a *Synchronization and Channel Coding* and a *Data Link Layer* sublevel (see Figure 3.8). The former deals with low-level encoding and forward error correction in data blocks and is typically implemented in hardware (see [22], [20]). The latter defines TC and TM *transfer frames*, which transport some kind of packet over the space link (see [26], [27]). The standards define virtual channels (VCs), which enable transport of multiple data streams over a single space link.

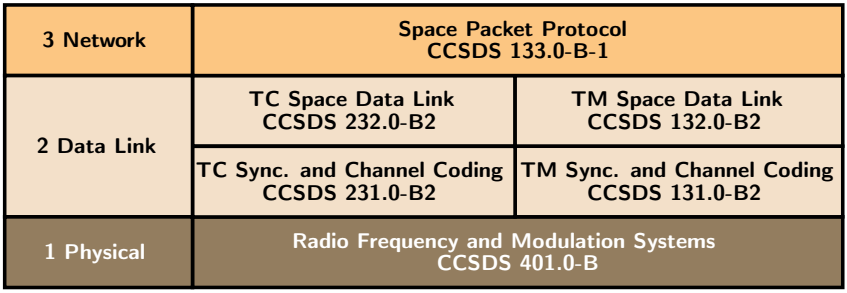

Figure 3.8.: OSI protocol stack of the CCSDS protocols

Typically, this is the level where FSW comes in contact with the protocols. Main activity for both TC and TM frames is frame encoding or decoding, as well as VC multiplexing or demultiplexing. The protocol is asymmetric regarding the TC and TM path. For TC uplink, the standard defines a sliding window protocol called Communications Operation Procedure-1 (COP-1) to ensure all frames are received, and in the correct order [18]. The protocol requires an appropriate implementation in the FSW. As the delivery of spacecraft telemetry is assumed to be less critical, there is no such protocol defined for downlink.

On packet level, the CCSDS Space Packet Protocol [16] defines variable-length space packets for routing of data within space and ground networks. Addressing takes place with a so-called application process identifier (APID) to find the destination entity. The protocol is quite simple, aside from the APID, it mainly defines a length field and a sequence count to check completeness. Still, a FSW using space packets needs to be capable of routing such packets to the intended destination entity on board. Furthermore, on-board generated telemetry packets need to be aggregated for downlink.

Even though these are the recommended protocols for space applications, it is possible to replace each of them with an alternative one. In fact, CCSDS defines a so called *encapsulation service* [17] to allow using other network layer protocols within TC or TM frames.

Another, small aspect of data exchange defined by CCSDS is the definition of time codes in [21] which ensures that time and timing information is exchanged unambiguously. A FSW needs to be able to interpret these time codes and generate time stamps in CCSDS format from its own time source.

In summary, the resulting features from a FSW perspective are:

Feat.: CCSDS.1 CCSDS TM frames
Encoding of TM frames and handling of multiple TM VCs.

Feat.: CCSDS.2 CCSDS TC frames
Decoding of TC frames and handling of multiple TC VCs.

Feat.: CCSDS.2.1 COP-1
Handling of the sliding window protocol on the receiving end for assured TC frame reception. As non-assured transmission is possible as well, this feature is optional.

Feat.: CCSDS.3 Space packets
Encoding and decoding of space packets.

Feat.: CCSDS.3.1 Space packet routing
Routing incoming space packets to the intended on-board destination, as well as routing generated TM packets to a downlinking entity.

Feat.: CCSDS.4 Encapsulation
Allow other network protocols to be transferred over the space link.

Feat.: CCSDS.5 Time Codes
Interpret and generate CCSDS time codes.

3.4.2. Mission Operation Services

Mission operation (MO) services are a CCSDS activity to harmonize communication between spacecraft, ground segments and mission operations centers. Work was initiated in 2006 and is, as of 2018, still ongoing. The current output is a whole suite of interdependent standards (see Figure 3.9), many of which are still under development or review.

Main intent is to create a framework for the development of an Open Systems Interconnection (OSI) application layer protocol which connects each entity of the system-of-systems, following service-oriented architecture (SOA) principles (see Section 2.3.4).

One intention of the protocol is to allow interaction between software components in an end-to-end fashion, e. g. from a payload operations center via

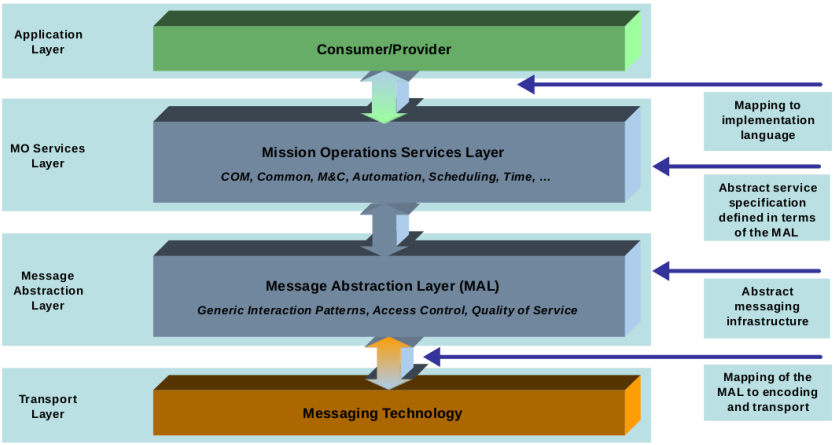

Figure 3.9.: Overview of the MO service framework. From [19].

the mission operations center to a payload on-board the spacecraft. Thus, the protocol needs to operate on multiple transport protocols with very diverging capabilities, e. g. a high speed local network and a resource constrained deep space link.

Therefore, an intermediate layer, the so called *message abstraction layer (MAL)*, is introduced. It specifies basic data types, as well as certain message interaction patterns which can be used to define service interfaces and messages. These interaction patterns are, for example, a simple request-response pattern, but also a complex publish-subscribe mechanism.

As these patterns and the data types are described in an abstract manner the Mission Operations (MO) standard suite defines certain technology *bindings*, e. g. to transport MO messages in space packets [25]. With regard to the OSI model, the MAL and the bindings are conceptually similar to a presentation layer protocol.

On top of the MAL, there is the so called *MO services* layer, which defines standard service specifications for applications, i. e. on OSI layer 7. These specifications rely on the Common Object Model (COM), which defines some common types and services, such as object identifiers and events, that form a template to specify other services.

The standard services, which are defined in terms of the MAL and COM templates, specify some common functionality of space missions, such as monitoring and control (M&C) or scheduling services. Of those standard services, only a

draft of the M&C standard is available to date. The relationship between the MO services stack and the OSI model is shown in Figure 3.10.

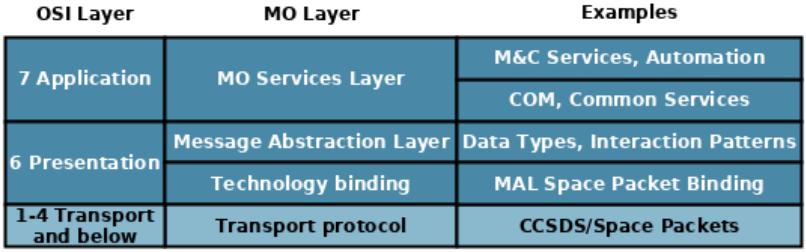

Figure 3.10.: Relationship between OSI model and MO services stack.

The intention of this section is to check the suite of MO service specifications for requirements regarding FSW implementations. Given that the standards do not specifically address the on-board environment, but any computing entity involved in mission operations, some functionality may be challenging to implement with the limited resources on-board a spacecraft.

Message Abstraction Layer

A FSW compatible to MO services needs to provide an implementation for the abstract data types and interaction patterns defined in the MAL, both for space-ground and for on-board communications. The most challenging part of such an implementation is the publish-subscribe interaction pattern. The intention is to distribute information, e. g. events, by publishing messages on a certain topic, and distributing these messages to a set of interested subscribers. This feature requires dynamic management of topics and subscriber lists.

Moreover, the FSW needs to handle the space packet binding, as incoming and outgoing messages are most likely transmitted via space packets. Depending on the implementation, it might be useful to translate messages into a dedicated on-board format for internal distribution as well.

Thus, the feature set resulting from the MAL is:

Feat.: CCSDS.6 MAL interaction patterns
Handling of the send, submit, request, invoke, progress and publish-subscribe interaction pattern as defined in the MAL, both for space-ground and for on-board communications.

Feat.: CCSDS.7 MAL space packet binding
De- and encode MO messages in space packets.
Feat.: CCSDS.8 MAL on-board binding
De- and encode MO messages in a dedicated internal protocol.

Common Object Model and Common Services

All MO service specifications rely on the common services defined in the COM specification [24]. Thus, a FSW needs to provide these services as well. Namely, these are an *event* service for the distribution of events, an *archive* service for storing any kind of information in a service deployment and an *activity tracking* service to inform about successful or failed transmission and execution of messages.

In a more abstract sense, these service specifications are instances of more general concepts: Independent of any specific protocol, a FSW should provide the following features:

Feat.: CCSDS.9 Event distribution
Reporting "something that happens in a system at a given point in time" ([24], section 2.4), and distributing this event to multiple interested parties.
Feat.: CCSDS.10 Information storage
Storing changes in variables, parameters, etc. for later retrieval.
Feat.: CCSDS.11 Activity reporting
Providing fine-grained reports of forwarding and execution of certain activities, especially ground commands.

Monitoring and Control Service

As the name says, this standard specifies what services a FSW shall provide to control and monitor a spacecraft. The current draft [28] specifies six main service types, of which at least the first three are mandatory to operate a spacecraft:

- MO action service: Providing access to activities which control the spacecraft, and monitoring of the progress of such an activity.
- MO parameter service: Monitoring of single on-board parameters and adjusting their value, if appropriate.
- MO alert service: A refinement of the basic COM event service to report and control the reporting of "significant asynchronous events or anomalies" ([28], section 2.8).

- MO check service: The service allows monitoring of value changes of parameters within the service providing entity and reports check violations with COM events.
- MO statistics service: Similar to the check service, this service allows computation of statistical values of a parameter within a service provider and reporting of the results.
- MO aggregation service: To avoid having numerous small packets with single parameter values, this service allows to define reports of multiple parameters which are transmitted in a single message.

As for the basic COM services above, these M&C services are concrete protocol-specific instances of fundamental features of a FSW. The MO action service, as well as the ability to set values in the parameter service, deal with remote control of behavior and activities in the spacecraft, i. e. *commandability*, while the other services provide more or less convenient means to monitor the spacecraft's state, which is referred to as *observability*.

Feat.: CCSDS.12 Commandability
Top-level feature of all aspects that allow adjusting the spacecraft's state and operational goals.
Feat.: CCSDS.12.1 Activity commanding
Handling of remote action invocation and monitoring of their execution.
Feat.: CCSDS.12.2 Parameter adjustment
Refining and adjusting of on-board parameters.
Feat.: CCSDS.13 Observability
Top-level feature that summarizes aspects of monitoring the spacecraft's state.
Feat.: CCSDS.13.1 Event reporting
Reporting asynchronous events that happened on-board the spacecraft.
Feat.: CCSDS.13.2 Housekeeping reporting
Reporting of on-board variables, typically periodic.
Feat.: CCSDS.13.3 Enhanced monitoring mechanisms
On-board preprocessing of variables, e. g. calculating statistics or performing on-board monitoring.

In an overview of planned MO services [19], more standard services are proposed, e. g. an automation service. However, no draft standards are available to elicit requirements from.

3.4.3. Spacecraft Onboard Interface Services

Another CCSDS initiative related to FSW are the Spacecraft Onboard Interface Services (SOIS) [23]. They consist of a set of reports which provide abstract interface specifications for communication on-board a spacecraft. The main

focus lies on data exchange between FSW applications and equipment, i. e. sensors and actuators, but it also contains concepts for communication between applications. Due to the high level of abstraction of these specifications, they do not provide a dedicated protocol to translate into software. Instead, they serve more as a collection of abstract functionality that should exist in some way or the other for on-board communication. This may not be a good guidance for practical implementations, but, for the same reason, makes SOIS a good source to identify generic FSW features.

Conceptually, SOIS makes a distinction between more high-level *application support services*, which provide generic features to applications and *subnetwork services*, which create a layer of abstraction between applications (or support services) and the communication medium.

First of all, the SOIS *application support layer* provides an interesting concept of device abstraction, which aims to reduce coupling between equipment-specific properties, and applications (see Figure 3.11).

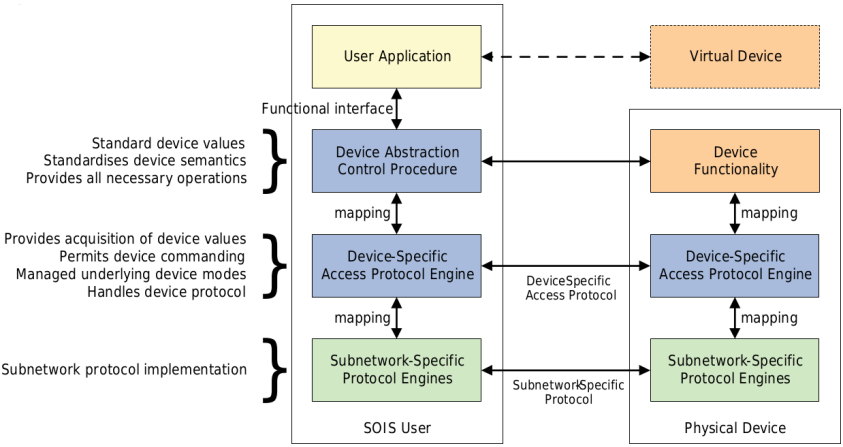

Figure 3.11.: SOIS concept of device virtualization. From [23].

The basic idea is to differentiate between a *device access service*, which provides direct access to the device in its native protocol, and a *device virtualization service*, which serves as a high-level interface for accessing the device, e. g. by converting sensor data to SI units. Moreover, SOIS identifies the need to provide unique addresses for equipment, manage underlying device *modes* and collect and distribute equipment data with a *device data pooling service*.

For example, exchanging data with a RW of *Flying Laptop* in its native, vendor-specific protocol is part of the device access service. If the FSW provides a software component which allows, e. g. setting the speed of a wheel in rotations per

minute, that would be a part of the device virtualization service. As illustrated conceptually in Figure 3.11, the FSW provides means for convenient access to sensors and actuators, e. g. by a control algorithm, which are not handled by the real device, but only by its software representation.

Whether being fully SOIS-compatible or not, a FSW ought to manage all of these features:

Feat.: CCSDS.14 Device access
Provide access to on-board equipment and handle the native protocol of the device, to acquire data and send commands to the equipment.

Feat.: CCSDS.15 Device representation
Provide a representation of the device within software, to allow interaction in a standardized manner.

Feat.: CCSDS.16 Device modes
Manage equipment modes and provide abstraction for configuration processes.

Feat.: CCSDS.17 Data acquisition
Periodically acquire measurements from equipments.

Feat.: CCSDS.18 Data distribution
Provide means to distribute equipment data (e. g. sensor measurements) in software without explicitly requesting the data, e. g. with a data pool.

Feat.: CCSDS.19 Value conversion
Convert sensor values and actuator commands between SI and device specific units.

In addition to these features, SOIS defines certain services to support dynamic discovery of equipment and equipment states, similar to USB devices on personal computers. However, while some ideas are interesting, the activities are somewhat futile, as spacecraft configuration is rather static and any kind of reconfiguration is highly mission-specific.

The SOIS application support layer also proposes services to assist development of on-board applications. One of them is the so-called *message transfer service*, which is a basic concept for communication between applications. Other services are the *file and packet store service*, which defines means to store files and packets on-board a spacecraft, and the *time management service*. These services translate well into abstract framework requirements:

Feat.: CCSDS.20 Inter-process communication
Provide means for applications to exchange messages.

Feat.: CCSDS.21 On-board storage
Provide some means to store some kind of data on-board the spacecraft.

Feat.: CCSDS.21.1 File store
Access and manage files and file stores on-board the spacecraft.

Feat.: CCSDS.21.2 Packet store
Access and manage storage of packets (e. g. TM packets) on-board the spacecraft.

Feat.: CCSDS.22 Time management
All on-board applications shall have access to a common wall clock with a standardized interface.

Regarding subnetwork services, SOIS provides a good categorization of different media types a FSW shall support ([23], section 2.1):

Feat.: CCSDS.23 Multidrop buses
Use a shared line for communication between multiple peers. Typically, communication is asymmetrical with one master and multiple slaves. The shared line needs precise management.

Feat.: CCSDS.24 Point-to-point connections
Direct connection between two peers, which allows bulk data transfer, e. g. for instrument connection.

Feat.: CCSDS.25 Homogeneous networks
An interconnection for larger systems with multiple nodes of similar computation power. Either managed by routers or competing communications, which may cause varying delays.

Also, it lists some typical physical and data-link layer protocols used in spacecraft, which are concrete instances of the above categories:

Feat.: CCSDS.26 MIL-STD-1553B
A serial data bus, originally designed for military avionics, but commonly used in spacecraft CDH subsystems.

Feat.: CCSDS.27 SpaceWire
A communication standard managed by ESA for fast point-to-point connections. Allows building switched networks of multiple nodes.

Feat.: CCSDS.28 CAN
The Controller Area Network (CAN) is a multi-master serial bus designed for communications of small microcontrollers on a simple, but robust bus. It was originally developed in the automotive industry, but found its use as sensor bus in space applications.

Feat.: CCSDS.29 Ethernet
Ethernet is the de-facto standard for cable-bound local area networks (LANs) on ground and is used in payload networks of some space missions as well [107]. Originally designed for a multi-master competing bus, it is typically used in a switched network topology.

One goal of SOIS is to provide an abstract interface for applications to communicate via any of these media types. To do so, a set of *subnetwork services*

are defined, with two different access types: First, there is a *packet service*, which is intended for packet-based communication, e. g. for requesting read-out of a digital sensor. The alternative is the *memory access service*, which allows accessing remote memories, e. g. for accessing mass memory or performing software updates of equipment. The distinction between these different methods of communication via a subnetwork is important and should find a representation in a generic FSW:

Feat.: CCSDS.31 Packet-based subnetwork access
Enable packet-based subnetwork communication.

Feat.: CCSDS.32 Memory-based subnetwork access
Enable memory-based subnetwork communication.

In addition, any media type has different capabilities, e. g. whether transmission is assured and message order is preserved or not. Some of these qualtiy of service (QoS) properties of subnetworks may need enhancements on software level, which is called the *convergence layer* in SOIS. In other words, SOIS proposes to extend OSI layer 3 and 4 capabilities of the used subnetworks, to achieve common capabilities. This would, for example mean to add a functionality to retry sending failed packets over RMAP SpaceWire, as this is not handled by the protocol itself.

However, the underlying basic requirement for a FSW is to provide some means of abstraction for different communication media types:

Feat.: CCSDS.33 Communication abstraction
A FSW framework shall provide means to separate application logic from the underlying communication medium used.

3.5. ECSS Standards

The suite of standards managed by the European Cooperation for Space Standardization (ECSS) contains a vast amount of European knowledge of space systems, ranging from project planning to star sensor performance specifications (see Figure 3.12).

Some of these standards deal with flight software in the narrow sense, such as ECSS-E-ST-40C [42] for software engineering. However, there are even more documents which have an indirect influence on FSW, such as the spacecraft operability standard ECSS-E-ST-70-11C [39]. Generally speaking, the following branches contain documents affecting FSW:

- E-40: Software engineering
- E-50: Communications

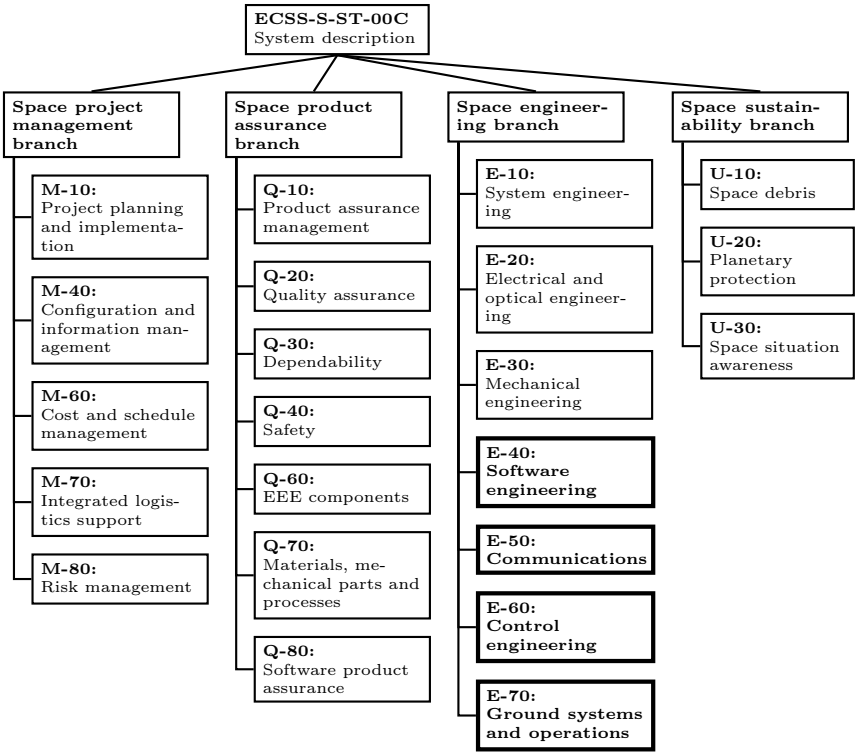

Figure 3.12.: ECSS disciplines in tree form [47]. Standards relevant for FSW have a thick border.

- E-60: Control engineering
- E-70: Ground systems and operations

In principle, the Q-80 discipline for software product assurance is important as well. However, these documents have a focus on the software development process, whereas the goal here is to identify functional requirements for a generic FSW. Therefore, they are not part of the survey.

The following section will give a quick overview of a number of standards, with summaries of the resulting requirements for FSW. As some of the E-60 standards depend on the E-70 standards, the latter will be introduced first.

3.5.1. E-40: Software Engineering

The ECSS-E-ST-40C document describes a process for development, operation and maintenance, as well as verification, validation and review of a space-related software product [42]. Therefore, its main intention is to define requirements for the overall software development process, which is basically a classical V-model approach (see Section 2.2).

In effect, the document does not specify any functional requirements for space software itself and therefore is, despite the name, a rather poor source to identify common flight software features. It does however formulate some clauses for the overall architectural design of a space software.

Clause 5.5.3.2b of [42], for example, states:

> The supplier shall test each software unit ensuring that it satisfies its requirements and document the test results.

Also, clause 5.8.3.3a poses requirements to form a comprehensible representation of the modularization and hierarchy of the software. These aspects translate to the following framework requirements:

Feat.: ECSS.1 Unit tests
Support individual unit tests of each software component of the system.

Feat.: ECSS.2 Hierarchical breakdown
Ensure that the hierarchical structure of the software is comprehensible and traceable.

Another aspect, which is well represented in the standard (in clause 5.8.3.11 and others), is schedulability for real-time software, which is a mandatory requirement for spacecraft FSW:

Feat.: ECSS.3 Schedulability
The dynamic architecture of a FSW shall support real-time schedulability analysis.

Finally, there is one aspect the standard does *not* mention: It does not preclude the use of an object-oriented programming (OOP) paradigm. Quite the contrary, OOP is explicitly mentioned as valid "software design method" ([42], clause 5.4.3.2).

3.5.2. E-50: Communications Standards

The E-50 series of ECSS standards deal with communication protocols for space applications. The first set of these standards (-01C till -05C) defines refinements for CCSDS protocols of the space link. As their relation to FSW is already covered in Section 3.4.1, they are not detailed here.

The remaining standards define or refine protocols for on-board communications, namely SpaceWire [41], MIL-STD-1553B [38], CAN [48], as well as a dedicated standard for discrete connections [40]. For SpaceWire, two standards define transport protocols: The Remote Memory Access Protocol (RMAP) [45] and the CCSDS packet transfer protocol [44].

The interaction between these subnetworks and the FSW is already covered in Section 3.4.3. Therefore, only the previously not mentioned SpaceWire transports are listed below:

Feat.: ECSS.4 RMAP
A protocol to provide access to memory of remote entities via SpaceWire.
Feat.: ECSS.5 CCSDS packet transport
A protocol to transport space packets via SpaceWire.

3.5.3. E-70: Ground Systems and Operations

The E-70 series of standards deal with ground systems and operational aspects of space missions. Those which deal with the internal design of the ground segment have a negligible impact on FSW. However, the E-70 series also provides a number of documents which define the operational interface to the spacecraft. As this is the "user interface" to the remote-controlled spacecraft, the impact on flight software design is high.

Specifically, there are three standards relevant for FSW:

1. Probably the most important ECSS standard is **ECSS-E-ST-70-41C** [49], commonly referred to as *packet utilization standard (PUS)*. It defines the space-ground interface in the form of a CCSDS space packet-based application layer protocol.
2. The space segment operability standard **ECSS-E-ST-70-11C** [39] defines operability aspects of spacecraft handling at different levels of detail.
3. **ECSS-E-70-ST-01C** [43] is rather specific and defines so-called on-board control procedures (OBCPs), which shall allow a limited amount of in-flight reprogramming of the system by operators.

A summary of these standards and resulting requirements are described below.

ECSS-E-ST-70-41C: Telemetry and Telecommand Packet Utilization

The packet utilization standard (PUS), which evolved from an ESA standard first introduced in 1995 to the most recent "C" version released in 2016[10], shaped the design of flight software development in Europe like no other ECSS standard.

Its basic idea is that there are *service providers* on-board the spacecraft, which accept certain *request* types from service users and generate *reports* in reply. Service users are typically on ground, but may as well be on-board. Transport of these requests between ground and space takes place with CCSDS space packets (see Figure 3.13).

The standard further details reports to be either data, verification or event reports. Verification reports are issued at certain stages of request reception and execution.

To distinguish different requests and reports, a service type and subservice type is introduced, for which the standard defines a secondary header with associated fields. In fact, this is the central syntactical specification of the standard.

On top of this basic concept, PUS describes standard service types, which intend to cover all typical monitoring and control (M&C) activities required to operate a spacecraft. The standard service types are summarized in Table 3.2. These service types and their respective subtypes come with detailed protocol definitions regarding the content of each request and report packet.

A tailoring, i. e. choosing a number of services and service capabilities for a given mission, is foreseen. Also, the standard encourages the definition of mission-specific services.

[10] An update from the "A" version from 2003 [37] was introduced to add new capabilities and define more "formal" requirements. Unfortunately, this drastically reduced readability of the standard.

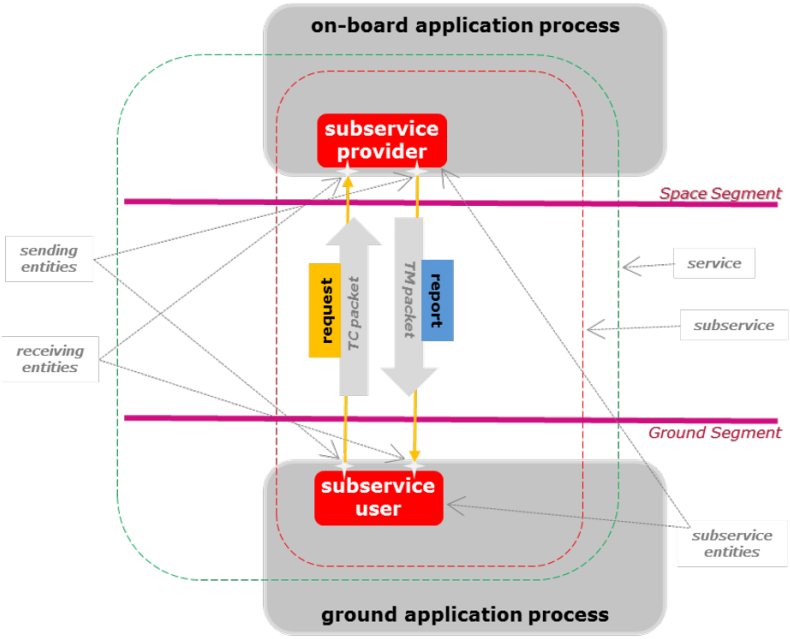

Figure 3.13.: The space to ground PUS service system context (from [49]).

Service type name	ID	Description
Request verification	1	Verify routing, acceptance and execution of TCs.
Device access	2	Provide direct access to on-board equipment.
Housekeeping	3	Control generation of periodic data reports.
Parameter statistics reporting	4	Manage reporting of data statistics.
Event reporting	5	Definition and control of event reports.
Memory management	6	Low-level access to on-board memories.
Function management	8	Invoke execution of some on-board function.
Time management	9	Control reporting of the on-board time.
Time-based scheduling	11	Allow execution of requests at a specified time.
On-board monitoring	12	Monitor variable values on-board.
Large packet transfer	13	Transfer large quantities of data.
Real-time forwarding control	14	Filter outgoing reports.
On-board storage and retrieval	15	Manage storage of TM reports for retrieval.
Test	17	Perform application-level connection tests.
On-board operations procedure	18	Manage execution of OBCPs.
Event action	19	Allow linking submission of TC requests to events.
On-board parameter management	20	Access and manipulate parameter values.
Request sequencing	21	Manage sequences to be executed one-by-one.
Position-based scheduling	22	Execute requests depending on orbital position.
File management	23	Handle and access an on-board file system.

Table 3.2.: Summary of service types.

Protocols and Flight Software Architecture

Ideally, a protocol definition allows interaction between entities regardless of a specific implementation. However, for application layer protocols such as PUS, it is difficult to distinguish which functionality exists due to the need of an application and which stems from protocol definitions.

The PUS standard aims to define the semantics and syntax, or interaction rules and content, of requests and reports for M&C of a spacecraft. However, it appears to be used as an implementation reference, as well as the basis for on-board data distribution (see e. g. [78], p.17). This creates "PUS-based" architectures, describing every functionality in terms of the standard and enforcing a certain implementation scheme. However, it should not be the intention of a standardized protocol to describe a single reference implementation, but to define *access to* an implemented functionality. Thus, the protocol definition should be generic enough to be of use for different styles of implementation.

Unfortunately, the PUS standard does indeed reflect the view of its authors' on what a spacecraft FSW should look like and leaves little room for interpretation. On the upside, the concept of mission specific tailoring and extensions allows adaptation to non-standard software implementations, which balances the overly specific layout of PUS[11].

Thus, even though there is a tendency to be too restrictive in the standard, a wider interpretation and implementation is possible by making use of tailoring.

Generic Features from PUS

For a generic FSW, architectural dependence on a dedicated communication protocol is not desirable. For example, M&C may take place via MO services or some common Internet protocol. Therefore, aside from supporting the PUS foundation model, the service types defined in Table 3.2 serve as reference for generic operator requirements only.

Feat.: ECSS.6 PUS space packet binding
De- and encode PUS requests and reports in space packets.

Most of the standard services from Table 3.2 are related to direct M&C tasks such as the execution of a certain activity and generating housekeeping reports. Service type IDs in that category are, at different levels of complexity: 3, 4, 5, 8, 12, 14, 20, and 21. As the resulting requirements are almost identical to those defined from the MO monitoring and control standard (see Section 3.4.2),

[11] This is a trade-off to optimize instantiation and adaptation cost, as described in Section 2.3.8.

they are not reproduced here. Likewise, the request verification service (Srv. 1) is conceptually identical to the activity reporting service of MO services.

Closely related to M&C tasks is on-board time management, as it is a precondition for precise control of the spacecraft. It is covered by the time management service (Srv. 9).

Feat.: ECSS.7 Clock Management
Provide a fine-grained wall clock to applications and report a precise on-baord time.

Another aspect covered by PUS is that of automation and out-of-sight operations, which is important for spacecraft operations, especially in LEO, with intermittent and limited ground contact. For this purpose, PUS specifies time or position-based scheduling (Srv. 11 and Srv 22) and a packet-based storage and retrieval service (Srv. 15), as well as a service to manage loadable control procedures (Srv 18), or link request execution to on-board events (Srv 19). Loadable conditional procedures, so-called on-board control procedures (OBCPs), are described in a dedicated standard and discussed later. To support such services and activities, a FSW shall provide the following features:

Feat.: ECSS.8 Command injection
Allow injection of TCs as if they were coming from ground.

Feat.: ECSS.9 Command storage
Support storage of TCs for later execution.

Feat.: ECSS.10 Report storage
Support storage of report packets for later retrieval.

Feat.: ECSS.11 Event distribution
Distribution of events to multiple interested parties.

The remaining standard PUS services are related to remote maintenance of the system and low level access to its resources. The device access service (Srv. 2), for example, defines remote access to on-board equipment and subnetworks. Such a service is needed in contingency situations mainly. Likewise, low-level memory access (Srv.6̇) is reserved for maintenance operations. In effect, a FSW shall support remote maintenance with the following features:

Feat.: ECSS.12 Device raw access
Support direct access to equipment commands and replies to the ground system.

Feat.: ECSS.13 Subnetwork raw access
Support direct access to on-board subnetworks from the ground segment.

Feat.: ECSS.14 Memory access
Support access to all memory regions in the system in a low-level fashion.

Feat.: ECSS.15 File management
Provide file management capabilities and allow remote access.

Relation between MO and PUS Services

There is a striking similarity between the ECSS PUS services and the definition of the MO standard services. This is not surprising, as they have the same goal, i. e. defining an application protocol for spacecraft operations. The main distinction is what assumptions about the utilized technology are made:

- **PUS** assumes a low-level procedural software architecture with limited abstractions. Thus, representing concepts such as software components is somewhat difficult with PUS. On the upside, the standard harmonizes well with limited resources of embedded systems.
- The underlying concept of **MO** services is that of a service-oriented architecture (SOA), which makes it easy to use the standard for more abstract conceptual designs. Also, it is intended not only for the ground-space link, but within the ground segment as well. However, some concepts of the standard may be difficult to handle in resource-constraint embedded systems, e. g. unbounded header fields in messages or archives for variable objects.

Still, even though the protocol syntax has many differences, the semantics of the protocols, i. e. the intention and types of services, are very similar.

ECSS-E-ST-70-11C: Space Segment Operability

The space segment operability standard ECSS-E-ST-70-11C is a collection of recommendations and requirements to ensure a spacecraft is operable in a "safe and cost-effective manner" ([39], p.7).

The standard is divided in a part for general requirement, which are top-level for categories such as observability, and a detailed requirements section, which is more grouped by functionality, such as on-board autonomy. Both sections provide interesting input, even though the latter part reads more like a lessons-learned document in some places. The standard does not only cover software-related requirements, but also system requirements, such as safe redundancy schemes. Also, many points are duplications or additions to services of the packet utilization standard. These are not reiterated here. However, there are a number of aspects not covered by the more formal PUS worth considering, which are described following.

First, there is safety and security of spacecraft commanding, for which a generic FSW could provide support. For safety, it can be useful to disable certain critical commands to avoid inadvertently execution, which is expressed in clause 4.5a and 5.5.2 of the standard. For security, a software framework shall support authentication and authorization mechanisms to ensure secure command execution (clause 5.2.1).

Level	Descriptions
E1	Mission exec under ground control, limited OB capability for safety issues
E2	Execution of pre-planned, ground-defined, mission operations on-board
E3	Execution of adaptive mission operations on-board
E4	Execution of goal-oriented mission operations on-board

Table 3.3.: Mission execution autonomy levels, from [39], p, 33.

Feat.: ECSS.16 Critical commands
Provide a mechanism to lock/unlock certain commands in case they are critical.

Feat.: ECSS.17 Authentication and authorization
Support authentication of operators on-board the spacecraft, as well as authorization of command execution.

Section 5.6 of the standard deals with spacecraft configuration and modes. Clause 5.6.1a defines a minimum of a nominal, standby and survival mode for the spacecraft and links these modes to used on-board equipment. In addition, clause 5.6.1d recommends some high-level mechanism to perform on-board mode transitions. Also, observability requirements for mode transitions are defined in clause 5.6.2. These concepts result in the following FSW concepts:

Feat.: ECSS.18 System modes
Allow representation of system and subsystem modes in software.

Feat.: ECSS.18.1 Mode commanding
Enable transition between subsystem modes, either autonomously or by ground command.

Feat.: ECSS.18.2 High-level mode commanding
Provide some high-level mechanism for simple mode commanding of the spacecraft.

Feat.: ECSS.18.3 Transition observability
Ensure the equipment configuration used for a certain mode, as well as the current mode is properly reported.

Another major aspect of the operability standard is on-board autonomy, which is divided in mission execution, mission data management and on-board fault management autonomy. For the first, four levels of mission execution autonomy are defined, which are described in Table 3.3. The software requirements for levels E1 to E3 are already covered by the corresponding PUS services. Even though level E4 is probably difficult to achieve, a FSW supporting that level should provide the following feature:

Feat.: ECSS.19 Mission goal representation
Support representation of mission goals, together with the required resources in software.

For fault management, or failure detection, isolation and recovery (FDIR), a number of recommendations are made in section 5.7.5 of the standard. The capabilities are divided in a lower level F1, which ensures spacecraft survivability in case of a fault and level F2, which aims to "re-establish nominal mission operations following an on-board failure" ([39],p. 35). In summary, the standard proposes a hierarchical FDIR implementation, which handles on-board faults at the lowest possible level. Also, all detected failures and the resulting isolation and recovery actions shall be well-observable, as well as configurable, from ground. Thus, a software framework should support the implementation of FDIR features in the following manner:

Feat.: ECSS.20 Hierarchical FDIR
Support different levels of failure handling, e. g. equipment, subsystem, system level and try to isolate a fault at the lowest possible level.

Feat.: ECSS.21 FDIR reporting
Ensure that detected faults and autonomous reactions are properly observable.

Feat.: ECSS.22 FDIR control
Allow disabling and enabling of FDIR actions by ground command.

Finally, in the later sections 5.8.10 and 5.9.1 the standard imposes requirements to allow reporting of software internal resource demands, such as software or subnetwork bus loads or processor utilization.

Feat.: ECSS.23 Resource monitoring
A software framework shall support reporting of the utilization of internal resources, such as software buses or shared memory.

ECSS-E-ST-70-01C: Spacecraft On-Board Control Procedures

The intention of on-board control procedures (OBCPs) is to enhance on-board automation by introducing loadable procedures, "which can easily be loaded, executed, and also replaced, on-board the spacecraft without modifying the remainder of the on-board software" ([43], p. 12).

The expected advantages of OBCPs versus native software is enhanced flexibility, so operators can react to unforeseen external or internal situations, e. g. an unexpected loss of a critical functionality.

However, it is somewhat difficult to determine the amount of functionality shifted from classical FSW to OBCPs, as this might introduce additional risks by hazardous misconfiguration of procedures. This aspect is addressed in the standard in section 6.2.2.

Still, they have been successfully utilized in a number of missions and are a good method to prepare a spacecraft for unexpected environments.

Feat.: ECSS.24 Procedure execution engine
Support the execution of loadable procedures and allow, but monitor, the usage of on-board resources.

3.5.4. E-60: Control Engineering Standards

The ECSS control engineering standards E-60 consist of a number of volumes primarily dealing with attitude and orbit control of a satellite. Three of them, namely ECSS-E-ST-60-10C, ECSS-E-ST-60-20C and ECSS-E-ST-60-21C define generic performance requirements for controllers, star sensors and gyroscopes. These standards mainly deal with controller design or equipment characteristics and are therefore not relevant to define FSW requirements.

ECSS-E-ST-60-30C [46] describes satellite attitude and orbit control subsystem (AOCS) requirements, not only on performance level, but also regarding functional and operational requirements. While many of these aspects are too specific to be useful as input for a generic FSW, some are good illustrations of required software features, and not covered by other ECSS standards.

For example, controller modes and mode transitions for nominal and contingency situations are defined in clauses 5.1.1.8 to 5.1.1.11. There, the standard specifies that an AOCS subsystem shall utilize a number of modes and allow transitions both by ground TC and autonomously. While the mode concept of system and subsystems is already covered in feature ECSS.18 identified in the operability standard, there is a new aspect to check and manage preconditions before performing a mode transition. This is a viable FSW requirement:

Feat.: ECSS.18.4 Mode transition checks
Check and manage software and equipment conditions to ensure ordered mode transitions.

Likewise, the standard defines some aspects relevant for failure detection, isolation and recovery (FDIR) in clause 5.2.2. There is a definition of a list of actions to be performed in case of a detected anomaly, which translates into the following FSW requirement:

Feat.: ECSS.25 Failure reaction
React on on-board failures by:

- ignoring transient faults
- reconfiguration on subsystem level without any change in the subsystem mode
- reconfiguration on system level including potential mode switches

An example for a transient fault is a single corrupted data packet from a sensor, with subsequent packets being correct again. Unless they accumulate, it is

better to have the FDIR ignore such faults than to be overly sensitive and e. g. force unnecessary safe modes.

The standard also covers requirements for telemetry generation, which are already covered in Section 3.5.3.

3.6. Existing Flight Software

There are a number of different endeavours to ease and improve FSW development. Many of these activities are performed internally in industry, and thus are not accessible for evaluation. Those which are available in public often are initiated as research programmes by a space agency, i. e. ESA or NASA, which try to mitigate the challenges of increasingly complex FSW. Typically, the aim is to derive a FSW framework or a reference architecture[12].

The following section summarizes a survey of those projects, with the purpose of identifying missing requirements for the domain analysis. Also, a basic evaluation of their capabilities takes place, to identify innovative solutions and potential shortcomings.

3.6.1. OBS Framework

The first project to investigate is the so-called *on-board software (OBS) Framework*. It is an extension of the *AOCS framework* described in [89], which was developed in 2002 by the Automatic Control Laboratory group of ETH Zurich. It was an early research project to investigate the use of object-oriented framework technologies for spacecraft FSW [90].

The OBS Framework defines a number of so-called *design patterns* (see Section 2.3.6), which aim to identify some recurring needs of a spacecraft software. Identified design patterns are e. g. a "control block" or the concept of "variable monitoring". Based on these design patterns, the framework identifies a number of abstract interfaces and concrete elements[13] provided by the framework. These concrete elements are some kind of managers, which call a list of registered software elements via the abstract interfaces (see Figure 3.14). For example, a controller manager executes a number of software components implementing the Controllable interface.

While the concept of the OBS Framework is interesting, the design lacks a certain maturity for practical use. For instance, the execution concept depicted in Figure 3.14 results in one component being called from multiple managers

[12] A reference architecture is a template solution for a software architecture of a particular domain [108].

[13] These elements are called "components" but should not be confused with a component-based software design.

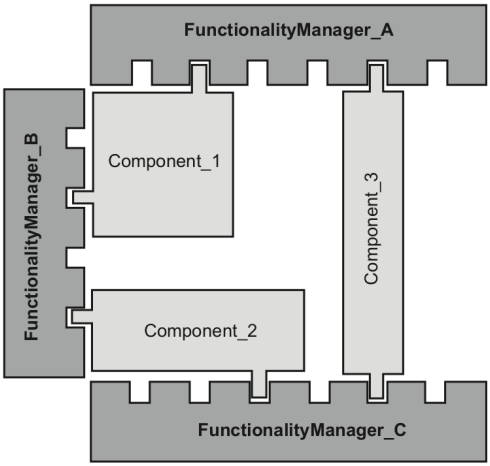

Figure 3.14.: Overall structure of the OBS Framework with different managers executing plug-in components which implement a certain interface ([89], p. 121).

at arbitrary intervals, making a real-time schedulability analysis difficult and eventually breaking the inner state of a component.

In effect, the OBS Framework never left the laboratory state. Still, it is a well-documented early example of a spacecraft software framework. The source code is available in public [90].

3.6.2. OBOSS-3

The Onboard Operations Support Software (OBOSS) framework is another early, but quite sustained project to develop a on-board software framework. Initiated by ESA in 1999 [104], it has been maintained and managed by TERMA with ESA support. In contrast to e. g. the OBS Framework, it has a very specific focus: Its goal is to provide a framework written in Ada, for PUS-based data handling software for satellite buses or instruments.

As such, it delivers a configurable *packet router* and a number of default PUS service implementations, which allow to define application processes and assign service instances (see Figure 3.15). The framework is capable of routing TC and TM packets between different application processes and supports the detailed definition of packet content.

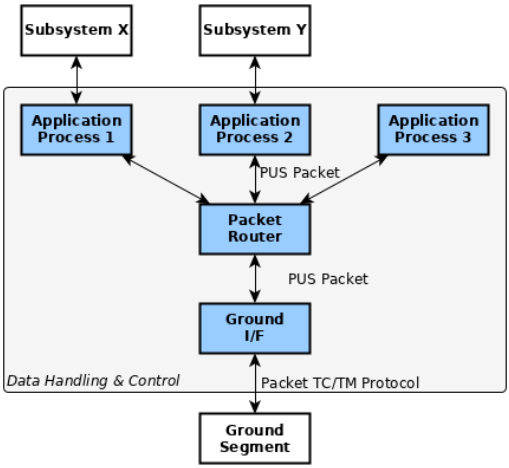

Figure 3.15.: OBOSS architecture outline (from [99]).

While this is a useful activity for a framework, its scope is limited to that of the PUS services. For example, it does not support the design of equipment handling or any additional control logic. Moreover, it is a framework fully dependent on the ECSS PUS standard and therefore not universally usable (see Section 3.5.3).

Still, if it fulfils the need of a project, e. g. to develop a PUS-capable space instrument controller, OBOSS may be a robust and well-established choice.

3.6.3. RODOS

The goal of RODOS is to provide a highly dependable real-time operating system (RTOS) which is explicitly developed for space missions. It is a project of DLR Bremen and is available open source [85]. To reach a high level of dependability, the focus is put on a small, yet complete microkernel to deliver the essential functionality of on RTOS, such as time, processor and memory management. Also, RODOS provides board support packages for various execution platforms, including LEON processor boards.

In addition, the main architecture of RODOS is defined as *network centric*: It means that all applications communicate with each other in a network-like fashion with a publish-subscribe mechanism. This is an interesting approach for application separation, and makes RODOS well-suited for systems with on-board networks, as well as distributed computing topologies.

However, being primarily an operating system, RODOS does not support the actual design of higher-level FSW applications and does therefore not provide additional input for this survey.

3.6.4. ESA On-Board Software Reference Architecture

The On-Board Software Reference Architecture (OSRA) concept is an ESA programme under the hood of the Space Avionics Open Interface Architecture (SAVOIR) initiative. Its goal is to define an agreed FSW reference architecture for future space missions, which supports the space industry by allowing "faster, later, and softer" FSW implementations [56]. The initiative exists for almost ten years to date, but has not yet produced a final reference architecture documentation. Still, a number of handbooks exist, which outline concepts and goals of OSRA.

Being an ESA initiative, OSRA incorporates many requirements of CCSDS and ECSS standards, e. g. CCSDS SOIS (see Section 3.4.3). Even though the reference architecture is not accompanied by or derived from a full sample implementation, and therefore not necessarily useful (see Section 2.3.2), there are many prototyping activities to avoid an overly theoretical concept.

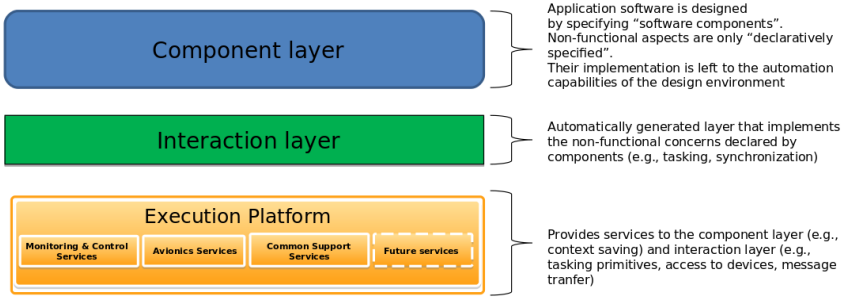

Figure 3.16.: Basic layout of the On-Board Software Reference Architecture (from [56]).

Technically, OSRA defines a model- and component-based approach to define a software architecture for space systems. This architecture is divided into three layers (see Figure 3.16).

- A component layer, which constitutes application elements.
- An execution platform layer which provides services for application components.

- An interaction layer, which bridges interactions between the other two layers.

From the OSRA concept, the most interesting element is the overall component-based approach, with interaction via interfaces, to support separation of concerns and encapsulation, as well as reusability of software. The general idea is in line with that identified in Section 2.3.3 of this thesis.

Also, OSRA describes a number of interesting consequences from that design decision. One point noteworthy is the strict separation of a specific technology or protocol from the application logic. In effect, there is a wrapper or abstraction layer for the RTOS used, but also for the protocols on the space-ground link.

Feat.: **FSW.1** **RTOS abstraction**
Hide the details of the operating system used from the application logic, to avoid dependence on a certain RTOS.

Feat.: **FSW.2** **Space link abstraction**
Ensure the application logic does not depend on a dedicated space link protocol.

OSRA also recommends some good practices for component and interface design. However, it does not provide a detailed concept on *what* a component should be, this is up to the implementation. Also, being a reference architecture and not a ready-to-use framework, many concepts are rather vague, such as how to actually separate the space-link protocol and applications.

Nonetheless, OSRA describes a wide number of interesting topics to use for modern FSW developments.

3.6.5. NASA Core Flight System

The core Flight System (cFS) is an effort of NASA's Goddard Space Flight Center to find a better way of FSW reuse than the traditional "clone and own" strategy ([86], p. 18). The initiative started in 2005 and an early version flew in 2009 on-board the lunar reconnaissance orbiter spacecraft. Since then, NASA continuously enhanced the software and promotes its use for various space missions.

The main idea of the cFS is to enable the creation of software applications which are 100% reusable for different space missions. To do so, a layered approach was chosen, which hides most of the used hardware and RTOS from the application layer. Also, the cFS provides a core system, the core Flight Executive (cFE), which manages application execution with a so-called Executive Service and provides some fundamental services to applications. These services are:

- Event Service: Distribute events between apps.

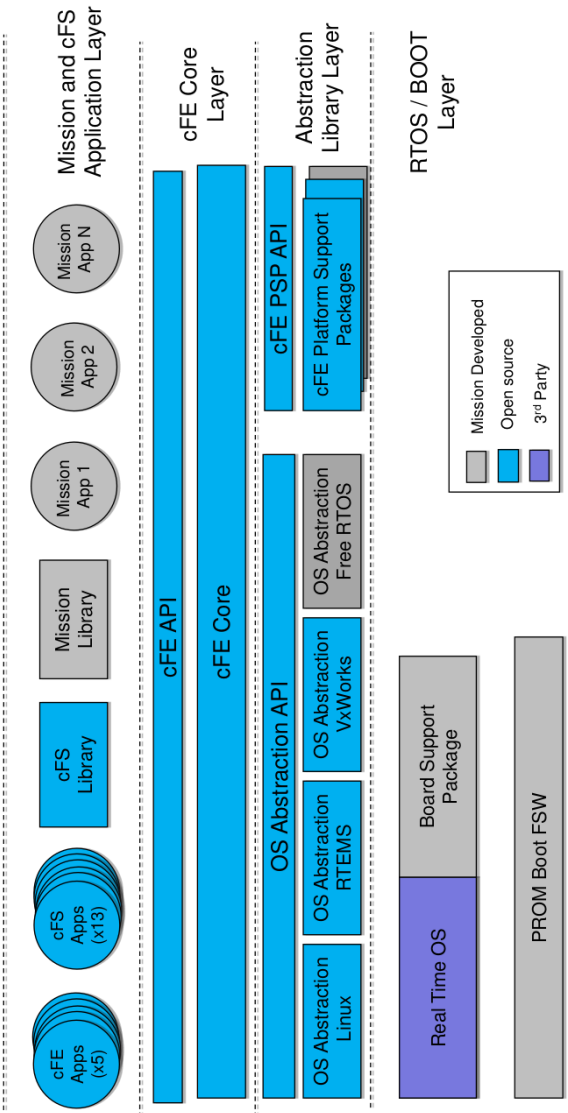

Figure 3.17.: cFS software layers and components (from [86],p. 54)

- Software Bus: Allow asynchronous inter-process communication (IPC) via messages.
- Table Service: Manages tables used by applications to store and share various kinds of configuration data.
- Time Services: Provide a common clock and conversion mechanisms for time codes.

On this foundation, the cFS comes with a number of default applications, which need to be supplemented by mission-specific apps to build a complete FSW. The layered architecture is depicted in Figure 3.17. The system is programmed in C and most of it is available open-source.

The cFE core layer and abstraction layer already provide some interesting features: Like OSRA, the cFS proposes RTOS and hardware abstraction to a certain degree. The Executive Service, however, is more than a simple RTOS abstraction. It allows stopping and starting applications at run-time, which is particular useful for quick ground test setups and partial in-flight software updates. Another interesting application is the Table Service, which allows applications to store run-time adjustable configuration data, such as subscription or filter lists. This is a useful feature for a generic FSW, even though the format of such data is not necessarily in table form.

Feat.: **FSW.3** **Application control**
Allow partial restart and updates of applications for testing and maintenance.

Feat.: **FSW.4** **Run-time configuration data**
Provide applications with run-time adjustable configuration data space.

All other applications and concepts of the core layer are already mentioned elsewhere.

The available default applications of the cFS are another source of FSW features. They are listed in Table 3.4. While the idea of most of these applications has already been captured, e. g. by a similar PUS service, there are some interesting details to consider:

- The Checksum app provides a configurable out-of-the box memory scrubber, which allows checking and eventually correcting radiation induced memory corruptions.
- The Data Storage app uses file-based storage of messages, e. g. housekeeping packets, which makes a simple, yet efficient store for TM packets.
- The Software Bus Network app enables distributed systems by extending the local software bus over some on-board network.

Application	Function
CFDP	Transfers/receives file data to/from the ground using CCSDS CFDP
Checksum	Performs data integrity checking of memory, tables and files
Cmd Ingest Lab	Accepts CCSDS TC packets over a UDP/IP port
Data Storage	Records housekeeping, engineering and science data onboard for downlink
File Manager	Interfaces to the ground for managing files
Housekeeping	Collects and re-packages telemetry from other applications
Health & Safety	Watches critical tasks check-in, services watchdog, monitor CPU utilization
Limit Checker	Monitor values and take action when exceed threshold
Memory Dwell	Allows ground to telemeter the contents of memory locations for debugging
Memory Manager	Provides the ability to load and dump memory
SW Bus Network	Passes Software Bus messages over various "plug-in" network protocols
Scheduler	Schedules onboard activities via (e.g. HK requests)
Scheduler Lab	Simple activity scheduler with a one second resolution
Stored Command	On-board Commands Sequencer (absolute and relative)
TM Output Lab	Sends CCSDS telemetry packets over a UDP/IP port

Table 3.4.: List of cFS applications. Those shown greyed out are meant for a laboratory environment only (from [86], p. 85).

For a generic FSW, this drives the following requirements:

Feat.: FSW.5 Memory scrubbing
Provide a concrete method to check for bit flips in memory.

Feat.: FSW.6 File-based TM storage
Use files to store telemetry data.

Feat.: FSW.7 Software bus gateway
Provide extensions for the internal software bus to allow distributed computing.

In summary, the cFS is a very mature system to develop a FSW, as it provides programmers with common default functionality and the layering allows adaptation to many different hardware environments. Also, it does define an overall software architecture with clean interfaces between applications and the system.

Comparison of OSRA and cFS

As both major FSW endeavours of ESA and NASA have similar goals, i. e. improve reuse of FSW, it may provide some insight to compare the designs of OSRA and the cFS.

Conceptually, many of the cFS applications have a counterpart in the ECSS PUS standard and therefore in OSRA, such as the Memory Manager or the

Housekeeping service. However, there is a fundamental difference: The cFS is an existing software, and the cFS apps are standalone executables. There is no standardized interface document for the system, nor does NASA provide anything similar. In contrast, OSRA, as a reference architecture, defines the concept of a FSW only, with PUS services and SOIS as interface definitions. No default implementation is provided.

This setup reflects the different mission philosophies of NASA and ESA. While the former does not only define, but also builds most space missions in-house, ESA typically specifies a spacecraft, with the actual work done by industry. In effect, the emphasis at NASA on working programming code is much higher than at ESA, where the primary interest is on precise specification of the spacecraft capabilities and the space-ground interface.

Both approaches have their benefits: While the NASA approach ensures that the cFS has high practical relevance, with little effort spent on anything other than a working product, the ESA OSRA concept introduces advanced theoretical concepts which may aid creating more complex software. Also, the focus on abstract standards in contrast to a complete software product avoids the risk of obsolescence due to technological advancements.

3.6.6. GenerationOne

GenerationOne is a Flight Software Development Kit (FSDK) especially designed for small satellite missions by Bright Ascension Ltd., a British space software company [8], which is also involved in some OSRA studies.

The main FSW architecture is component-based in the sense that software components for many types of off-the-shelf hardware are available. A service-based software framework and code generators, which are fed by configuration files, produce much of the fundamental code for using the selected hardware (see Figure 3.18). If a FSW is intended for use in multiple missions, this is an important feature to consider:

Feat.: FSW.8 Off-the-shelf components
Allow the design and composition of off-the-shelf software components, e. g. for common hardware.

Also, similar to the OSRA concept, the product provides hardware, RTOS, and space link protocol layering, which makes components highly portable.

The existing number of off-the-shelf components, together with a well-established development tool chain and support for testing and mission operations, make the generationOne software an interesting product. However, being a

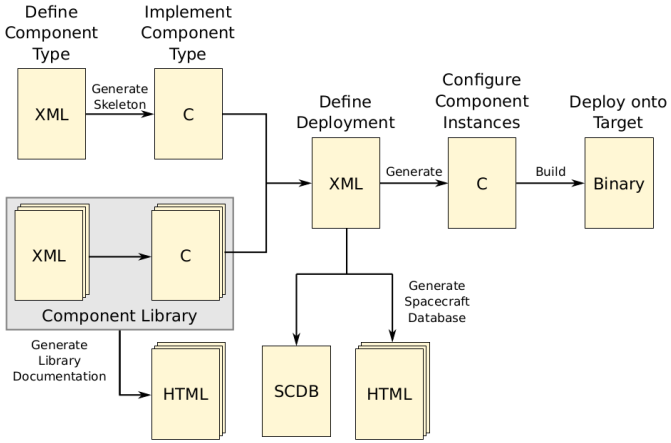

Figure 3.18.: Tool chain of the generationOne Flight SDK to generate FSW and documentation.

closed-source commercial activity, it is difficult to perform an in-depth assessment of its functionality. Also, its current focus on CubeSats limits its applicability for larger missions, which may require additional features, e. g. to manage redundancies.

3.7. Synthesis

The goal of this domain analysis is to define one set of requirements for a FSW product. Therefore, the scattered and unordered findings from the diverse evaluated sources need to be combined to a single set of features.

A first step of this synthesis is to break up the dependency on a certain data source and use the top-level features of Section 3.1 as initial categories, which are displayed in Figure 3.19.

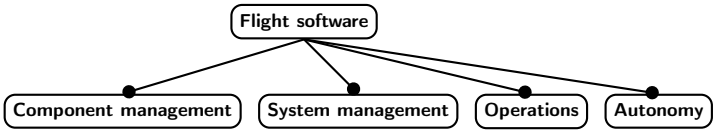

Figure 3.19.: The main top-level features any FSW must provide.

Within these topics, removing duplications and more fine-grained categorization takes place. The results are displayed in the form of dedicated feature diagrams. As illustrated in Section 2.3.5, filled circles indicate mandatory, empty circle optional features. Filled squares represent groups of OR features, empty squares groups of XOR features.

To allow tracing the original sources of requirements, a synthesis table for each category is provided in Appendix B.2, which links the synthesised requirement to its origins and also highlights duplicated requirements from different sources.

3.7.1. Component Management

The *component management* top-level feature contains all features and requirements related to the execution and management of standalone software components. Thus, it is about the handling and support of software elements in a generic flight software. The term component itself is rather vaguely defined here: A component is an independently executing software element of an overall application, e. g. an embedded control algorithm.

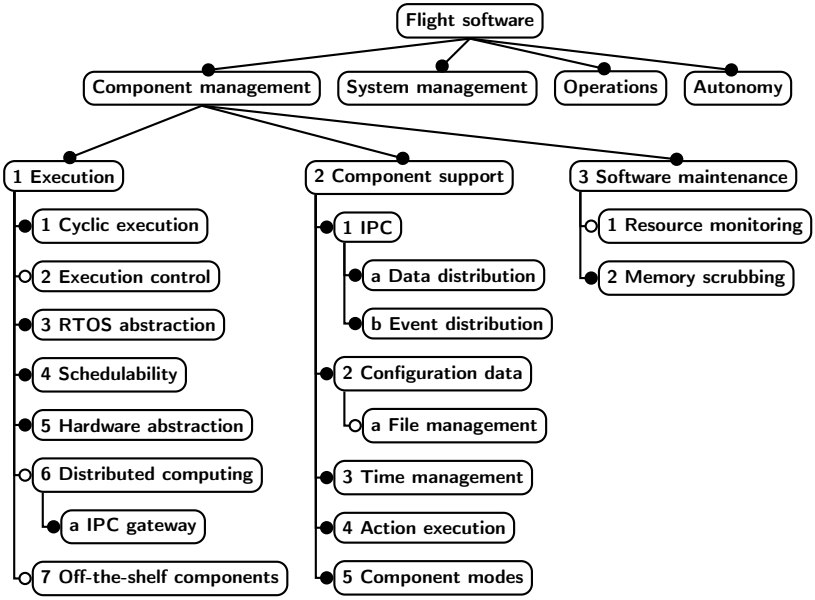

Figure 3.20.: Feature diagram of component management requirements.

Requirements for this category are summarized in Figure 3.20, which is broken in three main sub-categories:

- **Component execution** summarizes the requirements to execute software elements. To avoid dependence on a certain operating system or execution hardware, abstraction layers are introduced. Special cases are supporting execution of components in a distributed environment, as well as the possibility to utilize and provide off-the-shelf components.

- **Component support** collects features which intend to support components in their execution. Many features are similar to the services defined in the SOIS application support layer (see Section 3.4.3), but there are important additions, such as supporting configuration data and component modes, e. g. for the different modes of a control algorithm.

- **Software maintenance** is a category to collect supporting features for the FSW itself. These features are meant to ensure dependable computing in space, such as memory scrubbing to retain memory integrity.

3.7.2. System Management

This category contains requirements related to managing spacecraft equipment and subsystems within the FSW. In contrast to the former category, these requirements deal with the interaction of FSW with external elements, such as sensors, and with their internal representation.

As can be seen in Figure 3.21, the three main categories are:

- **Equipment:** A collection of requirements regarding the handling of a single on-board equipment, typically sensors or actuators. The requirements are not only about data acquisition and commanding of equipment, but also about more abstract concepts such as handling equipment modes.

- **Subsystems:** These requirements deal with representing and utilizing the concept subsystems in software. This includes management of redundant equipment, as well as representing subsystem modes and handling of mode transitions. For simple spacecraft, the entire category may be omitted.

- **On-board communication:** This category summarizes requirements related to on-board communication, e. g. over a network or a direct connection to a device. An important aspect is communication layering.

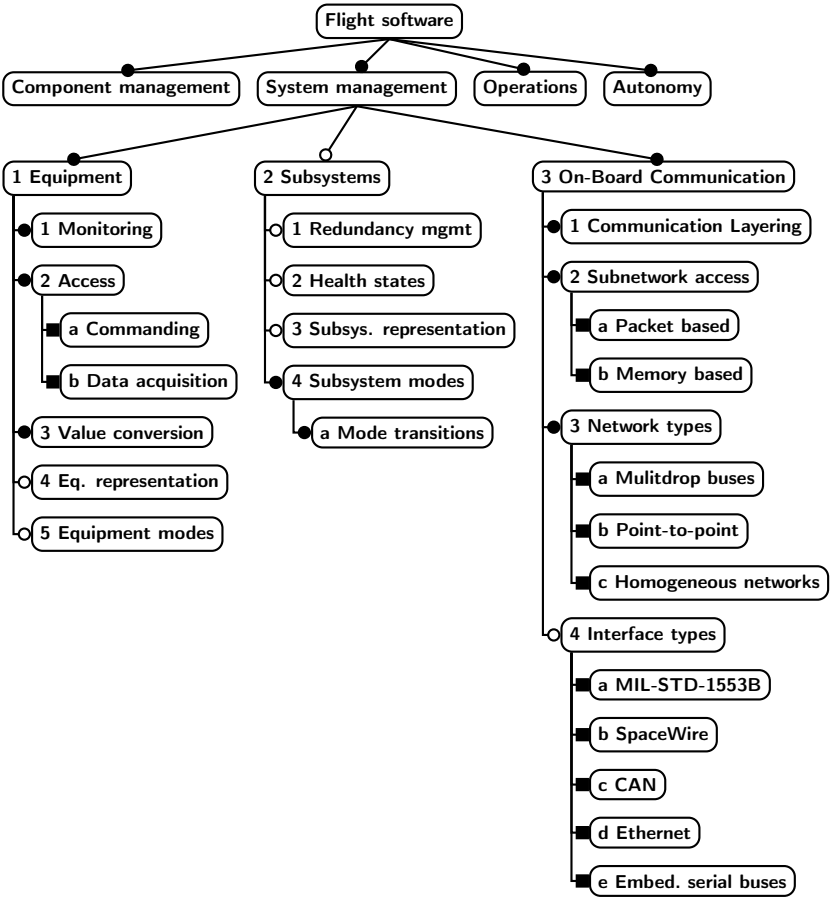

Figure 3.21.: Feature diagram of system management requirements.

3.7.3. Operations

The *operations* category of requirements contains three sub-categories as well: Control, monitoring and the space link, as depicted in Figure 3.22. The first two categories contain requirements for interaction between a ground segment and the spacecraft on OSI application level in an abstract way, i. e. there are no specific services defined, only categories.

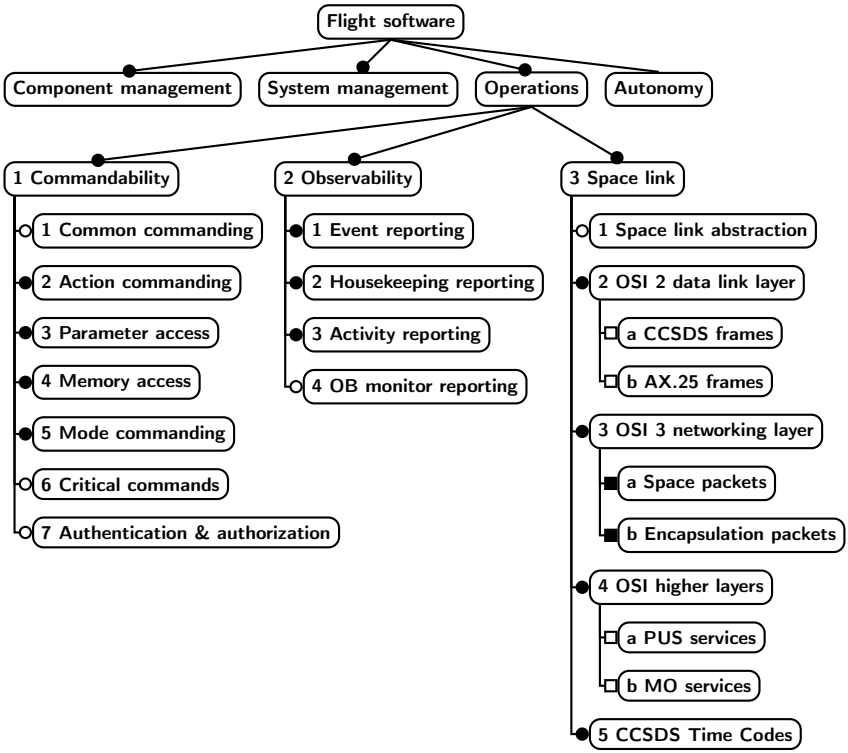

Figure 3.22.: Feature diagram of requirements related to operations.

In contrast, the space link category defines options for a concrete protocol stack to communicate with a spacecraft.

- **Commandability:** This category mainly comprises of different kinds of commands in an abstract sense, which are used for nominal operations and maintenance. In addition, safety and security measures regarding commanding are included.

- **Observability:** In a similar abstract manner as the control category, the monitoring category collects different types of reports a FSW ought to provide.
- **Space link:** The concrete protocols define choices to set up the complete space communication stack. The options are grouped according to the OSI layer model. Most protocols stem from CCSDS standards.

3.7.4. Autonomy

The final category of requirements collects FSW needs with regards to on-board autonomy.

These requirements are sorted in two categories:

- **Autonomous operations** are FSW requirements which enable the spacecraft to perform activities out of sight of a ground station. In a simple form, executing pre-planned commands and storing telemetry and mission products is sufficient. More enhanced FSW may support concepts of on-board planning and autonomous mission execution. Depending on the complexity of the mission, these last points may become arbitrary complex and require additional subfeatures. However, they are optional for typical earth observation satellites.
- The second autonomy aspect is **fault management**, which deals with on-board detection, identification and isolation of equipment faults, as well as a recovery, if possible.

3.7.5. Summary

The identified set of features and requirements form the baseline of what a generic flight software shall provide. In an abstract sense, the main four categories deal with the internal organization of the software, the representation and management of equipment and subsystems, interfacing with ground operators, and on-board autonomy, as illustrated in Figure 3.24.

A key challenge for providing a generic FSW as a product is to handle the high variability in processing hardware, real-time operating system, as well as sensor and actuator equipment, without compromising efficiency of the embedded system. Moreover, using the product must provide a benefit in terms of development effort, i. e. using a generic FSW must be more efficient than doing development from scratch.

Still, the commonalities in FSW duties found in this analysis, and the already performed research such as the cFS or OSRA, indicate that it is possible to create a FSW product which allows improved forms of software reuse and by

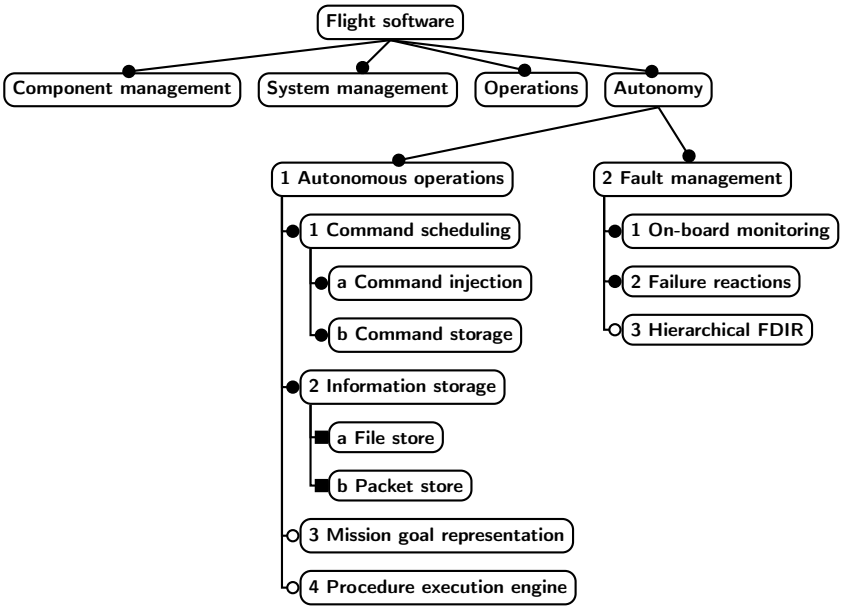

Figure 3.23.: Feature diagram of requirements related to spacecraft autonomy.

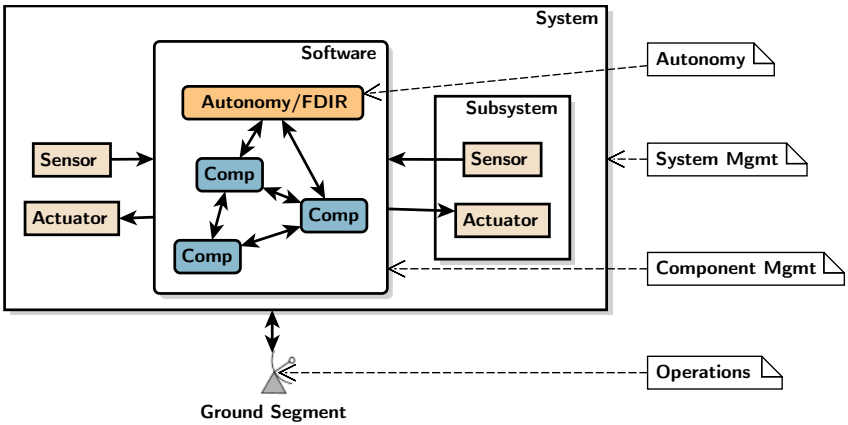

Figure 3.24.: Relationship between feature categories and system elements.

that reduces development time and cost while maintaining or even improving software quality.

Non-Functional Features

Some features identified in the domain analysis do not fit into one of the above categories, as they are not related to a functionality, but to the structure of the software itself.

Namely, these are FLP.17 on software modularity, ECSS.1 on unit tests and ECSS.2 on the hierarchical structure of software, as well as FSW.8 on off-the-shelf components. These features are not included in the feature tree, but taken into account on architectural level of a software design.

4. The Flight Software Framework

This chapter describes the implementation of the Flight Software Framework (FSFW), which utilizes modern software engineering techniques to fulfil the requirements for a generic, reusable flight software.

To introduce the concepts, the main architectural design and its rationales are outlined briefly in Section 4.1. This overview is followed by a description of the software for the small satellite *Flying Laptop* in Section 4.2. This concrete flight software (FSW) is based on the FSFW, and therefore provides a practical example to illustrate its inner workings. The following sections dig deeper into the implementation details of the framework:

- Section 4.3 outlines *common interfaces*, which define a set of default functionality components can offer and use.
- This is followed by a description of the *FSFW-Core* in Section 4.4, which ensures safe execution of components and communication between them, as well as accessing the underlying execution platform from components.
- Section 4.5 describes *component templates*, which facilitate the implementation of components.
- The *FSFW PUS Framework*, a collection of software elements to implement the PUS space link protocol, is described in Section 4.6.
- Finally, Section 4.7 describes the infrastructure to develop *FDIR functionality* as an integrated, hierarchical part of a FSFW-based software.

4.1. The Flight Software Framework Architecture

The goal of the FSFW design is to utilize software engineering techniques to make a reusable FSW, which implements the concepts found in the domain analysis.

To improve separation of concerns, the FSFW aims for a *component-based* software architecture. This means, a well-defined set of interacting components forms the overall software. These components interact by making use of a lightweight component framework, the `FSFW-Core`, which ensures real-time capable information exchange between components.

Creating independent components on the basis of a framework fosters software reuse by using entire components again for different missions. However, as soon as the hardware environment changes, e. g. the processor architecture of the on-board computer (OBC), enabling reuse requires to separate components and the framework from dedicated execution platforms, on-board networks and operating systems. Therefore, the FSFW introduces abstraction layers, to allow portability of the software to different environments.

Moreover, to reduce coupling between components, this thesis defines a set of common interfaces both for inter-component and ground communication. These interfaces resemble a fixed set of service interface definitions from SOA (see Section 2.3.4). They are the main means of interaction between components and define the functionality a component provides.

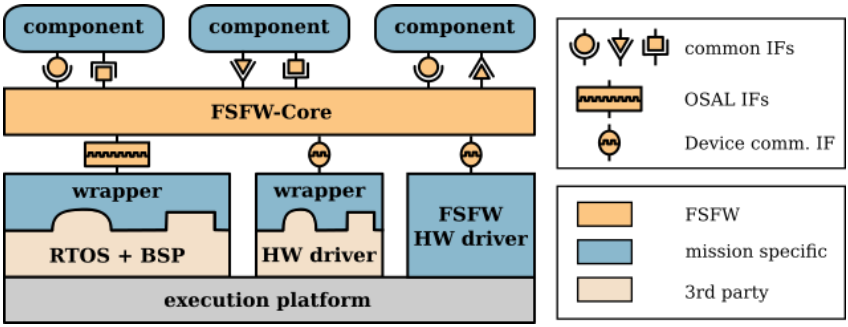

Figure 4.1.: The FSFW architecture with components, the FSFW-Core and execution platform abstractions.

These “invariants of design” ([91], p.31), as depicted in Figure 4.1, form the foundation of the FSFW architecture defined in this thesis. They are the fundamental principles describing how the different parts of the software form one application. The following sections provide details and rationales for the described concepts.

4.1.1. Software Components

A flight software implementation which uses the FSFW as basis is assembled from a number of individual *software components*.

This term requires a precise definition, as its meaning varies in different contexts. OSRA, for example, defines components as an abstract piece of software, which is a standalone service provider that can be assembled with other pieces, and may itself consist of smaller parts [56].

However, this definition doesn't aid developers in defining components for their mission. Therefore, the FSFW employs a narrower definition of what a component is. In the context of the FSFW defined in this thesis, a component is

- a standalone element of real-time capable execution,
- which acts as service provider and eventually service consumer in the sense of SOA services (Section 2.3.4), and
- represents an individual part of the overall system.

The last part of that definition means that each software component represents a part of the spacecraft system. In effect, the FSFW identifies four classes of components:

- **Device handler components**, which represent, control and monitor equipment, i. e. sensors and actuators of the spacecraft system.
- **Controller components**, which perform some form of control activity for the spacecraft.
- **Subsystem components**, including **assemblies**, which represent the engineering concept of a spacecraft subsystem and a redundant set of equipment, respectively.
- **Ground service components**, which provide specific functionality relevant for ground interaction.

This clear definition eases understanding and communication between software and other system engineers, as creating components with an immediate meaning makes their scope and responsibility obvious[1].

To simplify implementing these types, the FSFW provides so-called *component templates*[2]. They support programmers by offering recurring functionality of a specific component type, which ensures the component is compatible to the FSFW core.

Technically, a component is an instantiation of a C++ class, which must implement certain interfaces to allow interaction with other components and the FSFW-Core. Thus, component templates are implemented as *abstract base classes*, a concept explained in Section 2.3.2. This means that components are subclasses of a given template, and need to provide implementations for adaptation points to define component-specific behavior.

[1] This is an important aspect, as object-oriented (OO) frameworks often fail to correctly reflect the ontology of their target domain [92].

[2] The term *template* is used in the sense of a boilerplate, and not related to the C++ template language feature.

Goal of the FSFW is to create reusable components. For example, a device handler component for a certain sensor, e. g. a star tracker unit, shall be reusable without changing its source code, if another mission uses the same sensor.

4.1.2. Common Interfaces

Interfaces define the functionality a component offers to other components. By hiding implementation details from interface users, this powerful concept of object-oriented programming (OOP) reduce coupling between components.

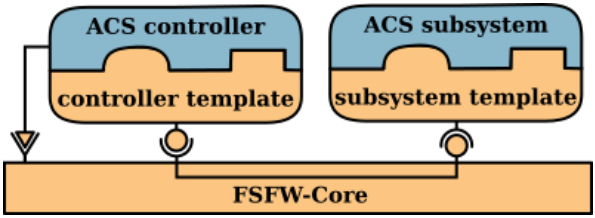

Figure 4.2.: Illustration of components providing and calling interfaces. Subclasses can extend the interfaces provided by component templates. Interaction is managed by the FSFW-Core.

One goal of this thesis is to identify generic interfaces, which define a common functionality of different types of components. To find these interfaces, the FSW feature tree defined in Section 3.7 serves as a guideline. When checking it for recurring features, the following functionality can be identified:

- **Actions:** Actions are sporadic, finite, externally triggered activities of components. This functionality is identified for device handler components in feature S.1.2a and C.1.4, e. g. for commanding a payload, and also for general commanding in feature O.1.2. Thus, it is useful for other sporadic activities as well, e. g. for resetting a control algorithm.

- **Modes:** The concept of a mode, which defines the permanent behavior of a component, occurs surprisingly often in the feature tree, for controllers (C.2.5), equipment (S.1.5), and subsystems (S.2.4). Therefore, a common interface definition to read and set a component mode is reasonable.

- **Health:** The idea of a health state (S.2.2), i. e. whether the component (and represented hardware) is available for operations, has its most obvious use for device handler components. Still, a common interface to read and modify this state is useful, as it is required for e. g. FDIR.

- **Parameters:** Parameters, i. e. variables which are adjusted occasionally, are typical for controller components, but in fact are ubiquitous in a FSW (O.1.3). A well-defined common interface simplifies reading and adjusting such on-board parameters.
- **Memory:** Feature O.1.4 identifies the need for low-level memory access. For reading and writing memory of a local processor, a common interface is, strictly speaking, not necessary. However, there is also requirement S.3.2b to enable access to memory regions in other equipment on a spacecraft over the on-board network, e. g. for dedicated mass memory units. The memory interface unifies the way device handler components make such memory accessible.

The FSFW defines interfaces for each functionality and ensures access is possible using the message-based software bus of the FSFW-Core.

As, ultimately, every component is an object, the FSFW provides interface definitions in the form of standard OO interfaces (see Section 2.1.4 for the programming technique).

Components can choose to implement a selection of these interfaces to offer the corresponding functionality to other components. As shown in Figure 4.2, component templates already implement certain interfaces, but extension by subclasses is possible.

4.1.3. The FSFW-Core

To support execution and allow interaction between components, the FSFW developed in this thesis provides a framework called the *FSFW-Core*. It delivers the following functionality to components:

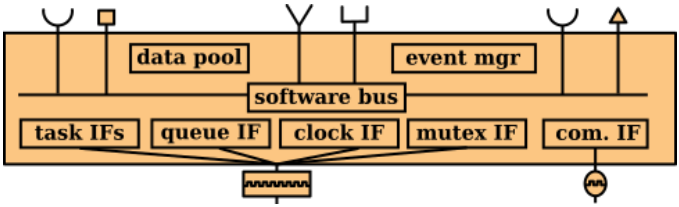

Figure 4.3.: The elements of the FSFW-Core layer.

- **Communication:** Supporting communication between components is a major task of the FSFW-Core. To ensure real-time execution of components, three methods of asynchronous information exchange are provided:

  - A messaged-base interface invocation scheme, which forms a *software bus* for component interaction.
  - In addition, the core provides the possibility to distribute *events* within the system.
  - Message-based interaction is supplemented by a *data pool* for exchange of periodic data, such as sensor measurements.

- **Execution:** The FSFW-Core provides software elements and interfaces to schedule components. This happens either on a periodic basis, typically needed for control algorithms, or in fixed time slots, e. g. if an on-board bus is utilized by different components.
- **Clocks and Timers:** In addition, the FSFW-Core delivers a common clock source to components in different formats and provides facilities to manage and set the on-board time. Also, software timers to measure intervals are provided.
- **Data containers:** The FSFW-Core provides containers to store and modify run-time adjustable data, such as telemetry configurations, of components. These containers are compatible to real-time embedded systems, as they are not susceptible to fragmentation. In many cases, they are replacements for common C++ Standard Template Library (STL) container implementations.

For most of that functionality, the FSFW-Core makes use of the underlying real-time operating system (RTOS). To avoid being dependent on one specific product, functionality is accessed via interfaces only, as shown in Figure 4.3. Likewise, the FSFW-Core defines a software interface to provide abstraction from a specific hardware interface.

4.1.4. Layering

Reusing a software in the embedded domain is much more difficult than in a e. g. web or desktop environment. The reason for this is that the latter builds on a complex set of layers introduced by the operating system infrastructure or the browser, with abstract device access and virtual machines.

To increase reusability of embedded software, it is necessary to efficiently introduce at least some of that layering. To separate the FSFW from hardware, two elements need to be decoupled:

- **The RTOS**: Using an RTOS is typically a good idea, as it helps to achieve abstraction from the underlying execution platform, i. e. the processor. In general, RTOS provide features for tasking, inter-process communication (IPC), and clock management. However, not every RTOS is available for every hardware platform.

- **Hardware interfaces:** The variety in on-board networks and peripherals is large in spacecraft systems. Therefore, to increase the chance of reuse of the software, it is desirable to introduce a layer of abstraction for utilized hardware interfaces.

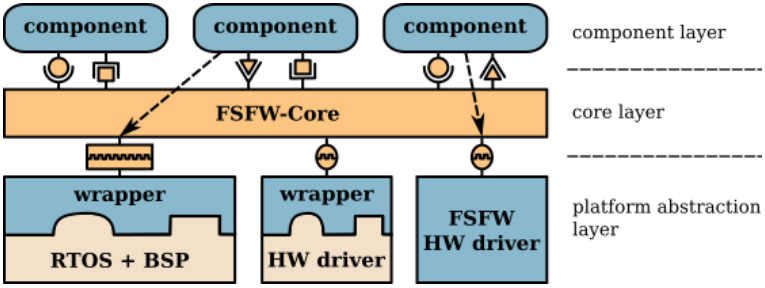

Figure 4.4.: The three main layers of the FSFW.

The FSFW designed in this thesis separates the RTOS with a set of interfaces, e. g. for task creation and control, which the RTOS needs to provide. As third-party software will not comply to these interfaces, dedicated wrappers for every RTOS are required.

A similar approach is followed for hardware interface separation. The FSFW defines a dedicated interface type for drivers, which provides means to configure and use a connection to some peripheral. Sending and receiving data happens by calling the interface, which, in fact, is an OO version of the socket API for network programming on desktop systems.

With this *platform abstraction layer*, the FSFW consists of three layers:

- **Component layer:** The layer that contains the application-level components.

- **Core layer:** This layer connects components, as well as components and the underlying hardware.

- **Platform abstraction layer:** This layer makes the underlying hardware - processors, memory and interfaces - available to components.

A layers perspective of the FSFW is depicted in Figure 4.4. The interface definitions of the FSFW-Core ensure that neither components nor FSFW-Core elements depend on a specific execution platform.

4.1.5. System Cognizance

Representing and managing spacecraft equipment is the most underestimated functionality a FSW needs to provide. In many cases, sensor and actuator management is regarded as part of the control algorithms, or some lower software layer, related to the on-board network. As argued in [91], Sec. 4.2.1, marginalizing equipment handling leads to bad designs, where e. g. fault management relies on some vaguely related parameter to identify equipment faults. Still, without timely, correct commanding and monitoring of devices, even the best controller design is useless, as it neither receives useful input nor are its control commands adequately executed.

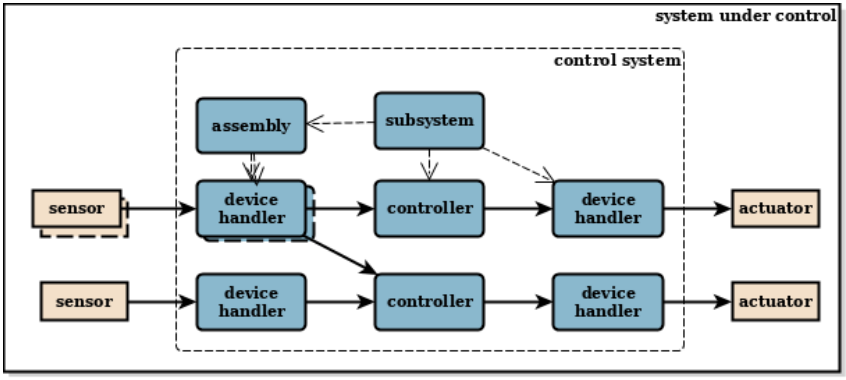

Figure 4.5.: An example set of FSFW components with control data flow (thick arrow) and mode dependencies (thin, dotted arrow).

Therefore, the FSFW tackles equipment handling explicitly: Each equipment is represented in software by a dedicated “control" component, called a *device handler*, whose purpose is to control and monitor the status of, and communication with, external equipment. Also, it represents the equipment internally and for the ground segment.

This has the following advantages:

- Device handlers explicitly control and monitor interaction with the device. This does not happen in some ad-hoc manner, e. g. within the control algorithm.
- Thus, device handlers encapsulate knowledge of device-specific properties such as initialization commands, the communication protocol and measurement representations.

- Being the internal representation of a sensor or actuator, device handlers are the single source of state knowledge of the equipment, e. g. they determine the health state of equipment.

- State information, as well as measurements and command data, is exchanged with other components in a disciplined manner, e. g. via a common interface, which, for example, makes fault management more explicit and therefore more simple.

Likewise, the abstract engineering concepts of *subsystems* and *assemblies* get a dedicated representation in the FSFW. Their goal is to determine and manage the mode of a subsystem, which is an emergent property of the mode and health states of all its controllers, equipment and sub-subsystems. In addition, assemblies encapsulate the rules of handling redundant equipment on-board the spacecraft. The basic concept is depicted in Figure 4.5.

It is important to emphasize that the resulting component hierarchy is *flat*, i. e. device handler components are on the same hierarchical level as controller or other components. This is a good design choice, as it avoids deeply nested, suboptimal hierarchies, which have difficulties to define how, for example, a temperature delivered by an attitude sensor is transported to a thermal controller [91].

In effect, these types of components ensure that the control system is explicitly aware, or *cognizant*, of the state of the system under control. In conjunction with the framework interfaces for mode and health management, this approach simplifies many aspects of spacecraft operations and on-board fault management, as will be revisited later.

4.1.6. The Space Link

Feature O.3.1 of the feature tree in Figure 3.22 highlights the value of avoiding a dedicated space link protocol dominating the framework. This is easy for the lower layer protocols (e. g. the data link layer), as the FSW is largely unaffected of a specific choice. However, as discussed in Section 3.5.3, it is more difficult for the application layer protocol, which is closely related to the functionality of an application.

The strategy to solve that issue in the FSFW is based on the following concepts:

- The FSFW does not use the space link protocol for internal messaging. Instead, a dedicated, lightweight protocol is used. It is, however, possible, to distribute TC and telemetry (TM) packets in their native format where necessary.

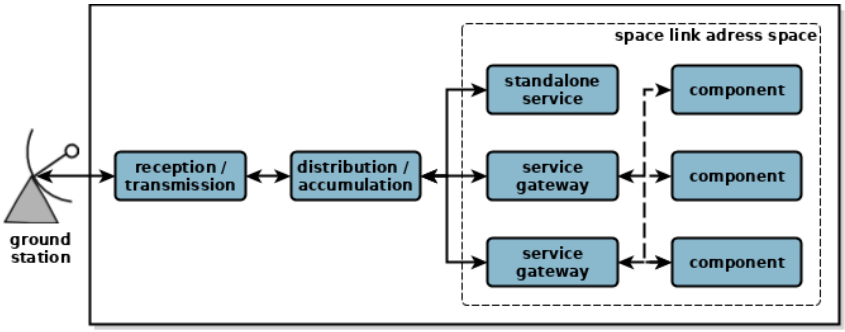

Figure 4.6.: Service gateway components and standalone components in the FSFW. Solid lines indicate TMTC communication, dotted lines messages on the internal software bus.

- To communicate with components from ground, so-called *service gateway* components are introduced, which translate between the space link and the on-board protocol.
- In addition, there are standalone *service components*, which provide dedicated, protocol-specific services, such as a TC scheduler.

These ideas and their interactions are illustrated in Figure 4.6. In effect, only service components depend on the space link protocol, and need adjustment in case that protocol changes, all other component types do not.

As the ECSS PUS, described in Section 3.5.3, is the dominating application layer protocol in Europe, the FSFW comes with a dedicated PUS framework tailored to implement PUS service components on-board. Details of that PUS framework are found in Section 4.6.

4.1.7. Summary

The FSFW allows the creation of a spacecraft FSW by programming a number of independent, but interacting components. The component-based approach and the well-defined framework interfaces make sure that coupling between components is loose, while still maintaining uniformity. Handling of different components is possible with identical command invocations.

With that basic layout, it is an implementation of many concepts defined of the OSRA specification and the CCSDS MO services, while pursing the same goals with regards to practicality and reusability as the NASA core Flight System.

4.2. The *Flying Laptop* Software

The overview of the FSFW architecture described in the previous section may not provide insight in the actual use of a component framework for spacecraft software. Therefore, this section is about leaving the realm of theories and deploying the idea to a real system: The small satellite *Flying Laptop*.

The *Flying Laptop* software was written as part of this thesis to verify the conceptual design of the FSFW with a hands-on example.

As frameworks in general and the FSFW in particular already prescribe an architectural design, getting started with the software development becomes more simple. In fact, the main design process for a FSFW-based flight software implementation is to define the mission specific components, wrappers and drivers. These are the blue elements in Figure 4.1.

This definition process comprises the following three domains:

- **Platform abstraction layer**: Depending on selected hardware and RTOS, defining wrappers for device drivers and the operating system is necessary.
- **Component definition:** For the space mission at hand, the list of components, i. e. device handlers, controllers, subsystems, and service components are to be defined.
- **Space link protocol:** Also, selecting a specific space link protocol is important, as well as the necessary components to handle the protocol itself. Service components are directly affected by that selection.

The presented software layout will serve as example to explain the details of the Flight Software Framework in later sections.

To avoid an excessive description of *Flying Laptop*'s subsystems, many elements, such as specific sensors, will be introduced without explanations beyond those from the introduction to *Flying Laptop* in Section 3.3. Also, some details have been omitted for brevity, such as a control component for the data downlink system (DDS). A more profound description of the system is found in dedicated literature, e. g. [55].

4.2.1. Platform Abstraction Layer

As described in Section 3.3.2, a processing board with a LEON3-FT UT699 microprocessor with SPARC V8 architecture controls *Flying Laptop*, with four SpaceWire lines as main hardware interface. Due to the good support by the board's vendor, the Real-Time Executive for Multiprocessor Systems (RTEMS)

was selected as RTOS. To use the FSFW in that setting, a wrapper to connect RTEMS with the FSFW is necessary.

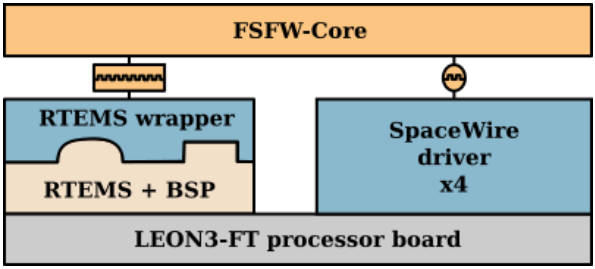

Figure 4.7.: Platform abstraction layer for the *Flying Laptop* flight software.

Also, the SpaceWire connections require to either wrap an existing driver or to write a dedicated FSFW-compatible driver. For *Flying Laptop* it was decided to write a custom driver for the interface [83].

Therefore, with the wrappers in place, components can directly use RTOS features or access the SpaceWire lines via the interfaces defined in the FSFW. The setup is depicted in Figure 4.7.

Due to the OBC design described in [52], the IO-Board and CCSDS-Board are not treated as part of the execution platform, but instead are considered as remote equipment, which is managed by dedicated device handling components.

4.2.2. Device Handler Components

An important aspect of the FSFW is that each on-board equipment is represented by a device handler component, as explained in Section 4.1.5. Thus, each *Flying Laptop* sensor and actuator has such a representation.

This is the case for both bus and payload equipment. As the payload on-board computer (PLOC) is responsible for data management only, all payloads are controlled by the OBC.

For the FSFW, devices are standalone parts of the system with a certain sensing or actuation purpose. Also, they allow some kind of digital communication with the OBC. For this reason, simple equipment, such as sun sensors or temperature sensors of *Flying Laptop* do not require a dedicated device handler component. Instead, they are handled by the component reading out the value and the responsible controller components.

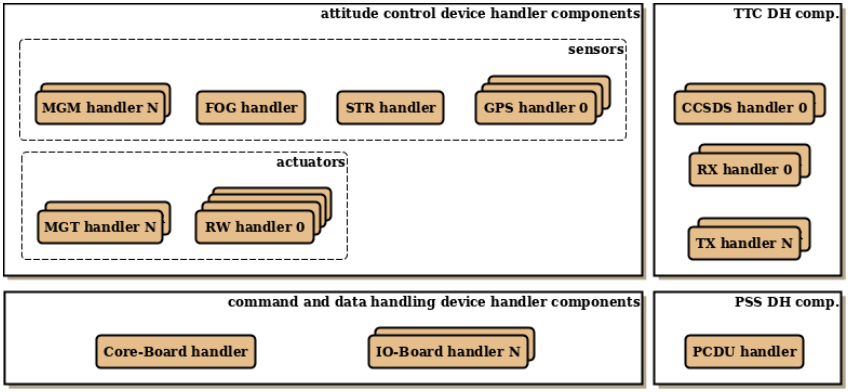

Figure 4.8.: Device handler components of the *Flying Laptop* bus system.

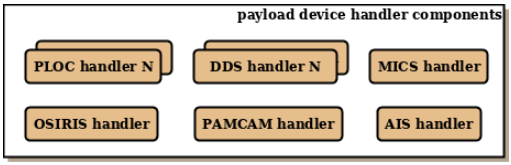

Figure 4.9.: Device handler components of the *Flying Laptop* payloads.

Figure 4.8 displays all device handling components of the *Flying Laptop* bus, ordered by main subsystems for clarity. There is a dedicated device handler instance for every equipment, i. e. there are four reaction wheel (RW) device handler instances[3].

In the same style, Figure 4.9 shows all device handler components of the payload.

Details on how device handler component communicate with other components or their device, as well as what they actually do, can be found in Section 4.4.6 and Section 4.5.1, respectively.

4.2.3. Controller Components

While device handler components control and monitor equipments, the system-wide control algorithms are managed in dedicated controller components.

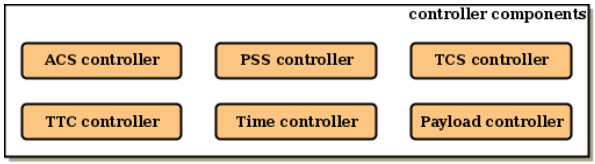

Figure 4.10.: The controller components of *Flying Laptop*.

The *Flying Laptop* software hosts most of the typical spacecraft control loops, which are attitude, thermal, power, and communication control (see Figure 4.10). Also, a time controller manages on-board time and a payload controller automates certain recurring payload activities on-board.

In addition to the actual execution of control algorithms, controller components in *Flying Laptop* are responsible for:

- **Sensor monitoring:** Monitoring sensor values with regards to absolute limits and comparing multiple sensors.
- **Sensor fusion:** Processing and combining sensor input data to generate a single spacecraft state.
- **Actuator processing:** Formatting the output date for single actuators, e. g. by transforming torque commands into the correct coordinate system.

[3] For technical reasons, the four FOGs on *Flying Laptop* are represented by a single device handler.

4.2.4. Subsystems and Assemblies

The FSFW provides means to develop subsystem software components, as introduced in Section 4.1.5. These components are responsible for determining, controlling and representing the mode of associated software components.

Subsystem Components

With the help of subsystem components, it is possible to transform spacecraft system mode tables of operations engineers into executable code. Thus, these components serve two purposes:

1. They allow creating a representation of the system's functional hierarchy in software, by forming a so-called *mode tree*.

2. With that hierarchy established, they allow the abstract definition of a subsystem mode by specifying the required modes of all children. This is a direct representation of the mentioned mode tables, which are not only used to check the current state, but which also enable ordered mode transitions.

Subsystem and assembly components are *branch* nodes of the tree, whereas controller or device handler components represent the *leafs* of the tree (see Figure 4.11). The element of commonality of all nodes is that they have a mode, which is adjustable via a common interface (see Section 4.3.2 for details).

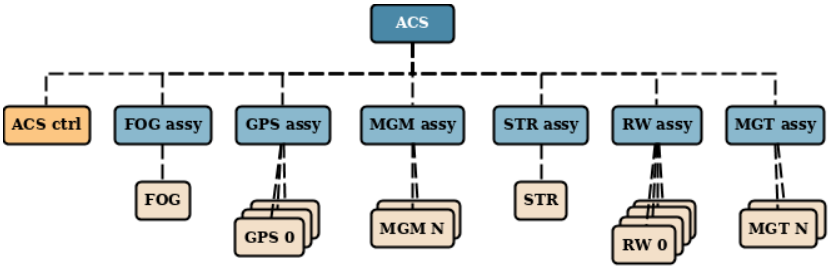

Figure 4.11.: Mode tree of the ACS of *Flying Laptop*, with dotted lines indicating the mode dependency.

In practice, this means that there is a subsystem component for each control domain of a spacecraft, as depicted in Figure 4.12 for *Flying Laptop*.

Assembly Components

Assembly components are a refinement of the general subsystem concept to manage redundant equipment. They extend the general subsystem with the following functionality:

- They determine and represent the mode of a set of redundant equipment units. For example, if at least one of a pair of redundant sensors units is available, the sensors' assembly is available as well.
- In case of a detected malfunction, assembly components are responsible for redundancy switching, e. g. activating a cold redundant sensor if the currently active one fails.

In effect, there is one assembly component for each set of redundant equipment on a spacecraft[4], as shown in Figure 4.12.

Using the Mode Tree

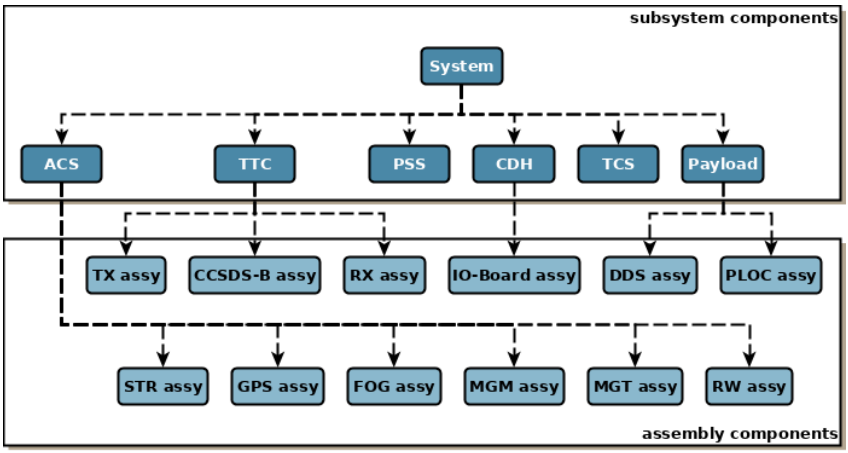

Figure 4.12.: Subsystem and assembly components of *Flying Laptop*, with dotted lines indicating the mode dependency. *System* is a subsystem component as well.

When combining all subsystem, assembly, controller and device handler components of *Flying Laptop*, their hierarchical dependency with regards to the

[4] In some cases, there are also assemblies for single device handlers, e. g. there is a STR assembly in Figure 4.11. This is useful in cases where there are internal redundancies to manage.

mode forms the mode tree of the system. For illustration, the subtree of the ACS is shown in Figure 4.11. The full mode tree is found in Appendix C.1.

While defining such trees seems to be some effort, the mode tree brings three main benefits:

1. It allows changing the mode of all components of a subsystem by issuing a single command to the top node. This node will autonomously perform the specified transitions. This works for subsystems, but also for the entire system by commanding the topmost node. So the mode of *Flying Laptop*'s ACS can be changed from safe to a pointing mode with a single command to the ACS subsystem component.

2. Spreadsheets or databases maintained by operation engineers can serve as input to auto-generate mode tables and transition sequences within the FSFW. This ensures documentation and implementation are always synchronized.

3. The mode tree supports spacecraft failure detection, isolation and recovery (FDIR), as all subsystem components define a so-called *fallback mode*, which aids in reconfiguring the system in case of equipment failures. For example, the ACS subsystem component will trigger a fallback to ACS safe mode if more than one reaction wheel stops working.

Assembly components play a key role in this concept, especially for the last point: During redundancy switches of equipments, assemblies do not change their mode, and therefore do not trigger any fallback transitions of their parent subsystem component. In effect, switching redundant equipment does not have an impact at system level.

As a remark, it should be emphasised that the hierarchy introduced is only relevant for the mode tree, i. e. whose mode depends on which component. From a software perspective, any device handler component is treated equally to any subsystem component. Particularly, it is also possible to command the mode of any component directly.

The inner workings of these features are further described in Section 4.5.3.

4.2.5. Space Link Protocols

The educational goal of *Flying Laptop* aims to train prospective space engineers in utilizing common space standards. To reach this goal, and to fulfil the requirements for the mission regarding communication reliability, a standard CCSDS stack was selected for the lower layers of the *Flying Laptop* space link.

Analogous to the lower layers, a network and application layer stack typical for an European institutional mission was selected, i. e. CCSDS space packets with the ECSS PUS-A application layer protocol[5].

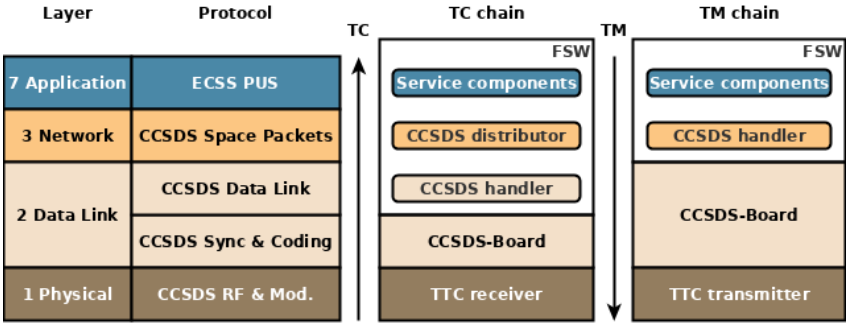

Figure 4.13.: Ground-space protocol stack of the *Flying Laptop* mission and their handling in hard- and software.

The complete stack is depicted in Figure 4.13. As shown, most of the handling below frame level is performed in hardware, whereas software comes into play on networking and application layer level:

- **Layer 1:** The TTC receiver and transmitter hardware handle the CCSDS RF protocol.
- **Layer 2:** The CCSDS-Board is responsible for frame de- and encoding, with the exception of CCSDS TC frame reception. Frame reception, including the COP-1 protocol, is managed in software. The functionality is allocated to the CCSDS-Board device handler component, which extracts space packet candidates from frames and forwards them to the CCSDS distributor.
- **Layer 3:** Network layer routing of TC and TM packets happens in software, with a CCSDS distributor component on the TC side and forwarding of TM packets implemented in the CCSDS-Board device handler.
- **Layers 4-6:** OSI layers 4 to 6 (transport, session and presentation layer) are not used in the space link stack.

[5]The newer C version of PUS was not available yet during design. However, most changes between version A and C are insignificant for the services used on *Flying Laptop*

- **Layer 7:** PUS services are implemented as dedicated service components, of which some handle standard and some mission specific services. These service components are further described in Section 4.2.6 below.

To implement that functionality the PUS framework of the FSFW is used, which is described in Section 4.6.

End-To-End Communication

The basic distribution of telecommand (TC) and telemetry (TM) within the *Flying Laptop* software is illustrated in Figure 4.14.

To enable end-to-end interaction between the ground segment and software components, some PUS services are implemented as gateway services (see Section 4.1.6), which translate PUS packets to the internal protocol and route the request to a software component. Other PUS components are standalone services, which handle the incoming PUS TC directly.

In effect, the complete end-to-end TC and TM chain works as following:

- TCs are received and decoded by the receiver (Rx) and CCSDS-Board hardware. Likewise, TM is encoded by the CCSDS-Board and linked to ground with a transmitter (Tx).
- The *CCSDS handler* component transfers TC frames between CCSDS-Board and FSW, extracts TC space packets and transfers them to the CCSDS distributor component. The CCSDS handler also accepts TM packets, which are forwarded to the CCSDS-Board unchanged.
- The *CCSDS distributor* is responsible for routing TCs with respect to their APID. Any components can register for receiving TCs. For example, direct routing to the star tracker handler is possible, which forwards packets to the PUS-capable star tracker unit.
- There is one APID for all services of the FSW, within which a *PUS distributor* component forwards the command to the destination service component.
- The *service components* either handle incoming TCs directly (standalone service), or translate and forward commands to another software component, e. g. a device handler (gateway service).
- In principle, TM can be sent directly to the CCSDS handler. However, to select and forward TM packets to the TM stores (not shown), a storage selection component is added to the route.

With this setup, ground commands reach any service component on-board the spacecraft and, through gateways, all other components as well.

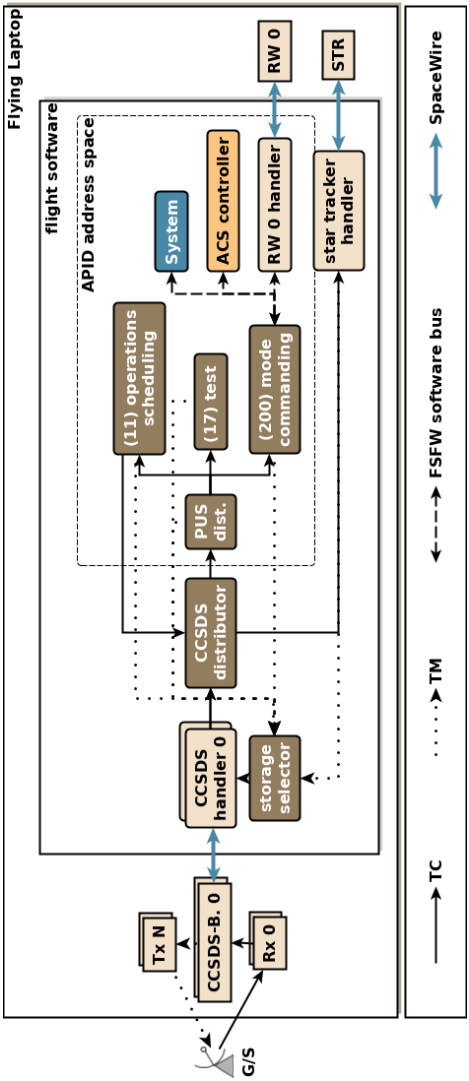

Figure 4.14.: Example of end-to-end communication between ground and different software components and equipment within *Flying Laptop*.

4.2.6. Service Components

To complete the overview of the *Flying Laptop* flight software this section describes the standalone and gateway *service* components.

With the ECSS PUS-A protocol selected for *Flying Laptop*, each service component represents one PUS service, with a dedicated service identifier and a number of subservices.

Most service components are of a standard service type as described in Section 3.5.3, but there are a number of mission specific definitions as well, especially for the gateway services. The tailoring, i. e. the selection of a set of both standard and mission specific services, is further detailed in [55].

Standalone Service Components

Standalone service components handle PUS TC requests directly. The services either interact with FSFW-Core elements, such as the housekeeping service accessing the data pool, or they interact with other service components, such as the TC verification service[6].

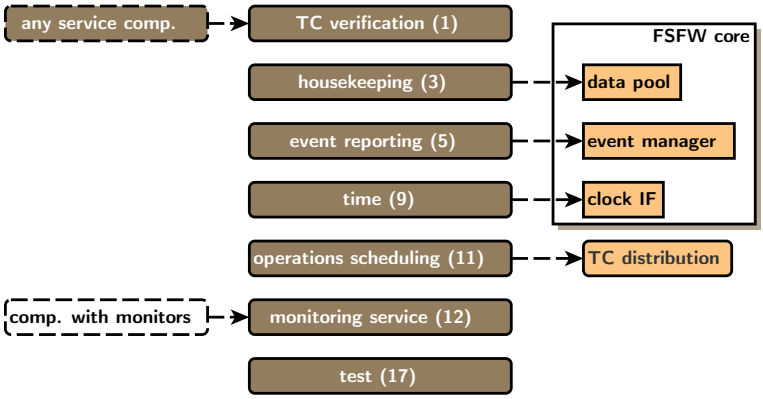

Figure 4.15.: Standalone service components implemented on *Flying Laptop* with indications which elements of the FSFW-Core they use. The numbers correspond to the definitions of the PUS standard.

For example, as shown in Figure 4.14, the operations scheduling service component stores time-tagged TCs internally and forwards them to the CCSDS

[6] Fully self-contained services would have little use to control or monitor the system.

distributor when due. The TC, e. g. a ping command[7], then arrives at the destined service at the planned time as if coming from ground.

All standalone service components used in *Flying Laptop* are illustrated in Figure 4.15.

Service Gateway Components

Another set of PUS services is translated into gateway components, meant to bridge communication between the ground segment and individual components. As such, each gateway component is responsible for translating PUS packets of a single service type to corresponding FSFW messages on the software bus. In practice, that means that e. g. the PUS memory management service (Srv. 6) translates PUS memory commands, so the request can be forwarded to any component providing a memory interface.

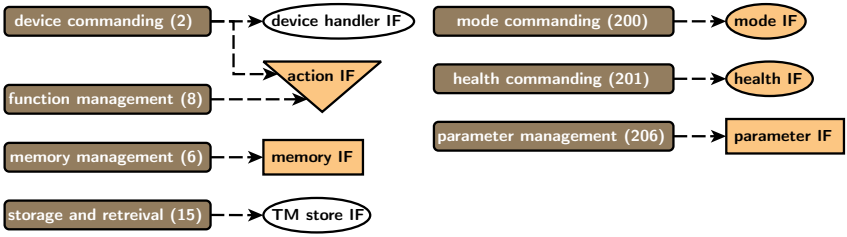

Figure 4.16.: Gateway service components implemented on *Flying Laptop* with indications which interfaces they access. Colored ellipses represent common framework interfaces, dotted ones indicate specific interfaces.

For *Flying Laptop*, there is typically one gateway service for each of the common interfaces defined in Section 4.1.2. For example, there is a custom mode commanding service, which receives mode TCs in PUS format from ground. These are translated within the service and forwarded to the destination component on the internal software bus[8]. For that component, the request is identical to one from any other component in the system. Also, as shown in Figure 4.14, any component implementing the mode interface can be target of such a command. In case of a device handler component, the command may result in one or more commands exchanged between the device handler and its associated equipment.

[7] A ping TC, or PUS test service (17,1) request, is a simple command to test the connection to an application process. It is replied with a (17,2) report TM packet.
[8] A component address must be provided within the data field of the TC.

All gateway service components of *Flying Laptop* are shown in Figure 4.16. There is a deviation from the one-on-one translation to a common interface in two cases:

- The device commanding service translates commands for the action interface as well as for a specific device handler interface. Device handler components use the former to forward high-level commands to devices, while the latter provides low-level access. It is more convenient to use a single service for all commands to equipment.
- For *Flying Laptop*, telemetry stores are implemented as semi-independent components. Access from ground happens through a dedicated TM store interface.

In summary, the combination of standalone and gateway components using the FSFW allows to provide a PUS-compatible ground interface, which bridges conceptual gaps between the PUS protocol and the FSFW component framework. More details on the FSFW PUS framework can be found in Section 4.6.

4.2.7. Summary

The *Flying Laptop* satellite is equipped with the first flight software implementation based on the FSFW. As shown, interacting components provide a suitable architectural design to implement a FSW. With device handler and subsystem components a form of system cognizance is created, which improves observability by precisely allocating state variables to a software component. By using the FSFW PUS framework, it is possible to interact with components from the ground segment in an end-to-end fashion using a tailored PUS protocol.

4.3. Common Interfaces

The first FSFW elements to describe in detail are the *common interfaces*. Goal of these interfaces is to define a common functionality which is shared among different components and component types. This is useful for programming, as accessing that common functionality works uniformly when using the interface. Also, interfaces define a "shared vocabulary" [73] to use when talking about components.

The FSFW defines five common interfaces, as introduced in Section 4.1.2. The following section presents these interfaces, and outlines the syntax and semantics for their implementation.

4.3.1. HasActionsIF

Actions, in the sense of this interface, are activities with a well-defined beginning and end in time. They may adjust substates of components, but are not supposed to change the main mode of operation, which is handled with the `HasModesIF` described below.

Figure 4.17.: Example image of a component providing the `HasActionsIF`.

In illustrations, a triangle represents that common FSFW interface (see Figure 4.17).

Example

A typical example for using an action is commanding a camera system to take a picture. In the same manner, the device handler component for the digital reaction wheels (RWs) of *Flying Laptop* implements the interface, and provides actions to e. g. set the wheel speed. In case some other component calls the interface, the command is checked, parameters are formatted to fit the device native format and forwarded to the wheel, which, then adjusts its speed.

Another example is the attitude control component, for which setting a new pointing target happens using an action.

Definition

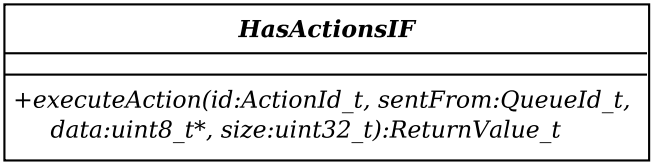

Figure 4.18.: Signature of the `HasActionsIF` interface.

The `HasActionsIF` allows components to define such actions and make them available for other components to use. Implementing the interface is straightforward: There is a single `executeAction` call, which provides an identifier for the action to execute, as well as arbitrary parameters for input (see Figure 4.18). Aside from direct, software-based actions, it is used in device handler components as an interface to forward commands to devices.

Implementing components of the interface are supposed to check identifier (ID) and parameters and immediately start execution of the action. It is, however, not required to immediately finish execution. Instead, this may be deferred to a later point in time, at which the component needs to inform the caller about finished or failed execution. This is further explained in Section 4.4.

Effects

This interface definition is useful for providing and accessing arbitrary functionality of components. However, such an open definition creates the risk of misuse, in that the interface is taken for any kind of activity, e. g. a mode change. The FSFW counteracts this risk by providing a clear definition and other interfaces for specific use cases.

In fact, the main use of the `HasActionsIF` in the FSFW comes from its role in device handler components, where it provides high-level access to forwarding commands to devices, similar to the idea of the Device Virtualization Service in SOIS (see Section 3.4.3).

In that, the interface definition covers the features O.1.2, C.2.4 and S.1.2a of the generic features found in Section 3.7.

4.3.2. `HasModesIF`

A mode represents a certain operational state a component and eventually associated hardware is in. Changing the mode permanently changes the principle of operation for a component. This is in opposition to *actions*, which do not have a permanent effect on the component.

The `HasModesIF` provides external access to a component's mode. Modes have different meanings, depending on the type of component. For better differentiation, each mode can have a number of submodes. In principle, any component could freely define own modes and meanings, but for reasons of uniformity, a `MODE_OFF` and a `MODE_ON`, as well as `SUBMODE_NONE` are provided by default.

Certain component types have additional predefined modes. For example, controller components typically define `MODE_NORMAL`, which, on top of processing the inputs, actually activates the calculated output of the control algorithms.

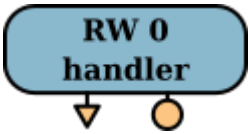

Figure 4.19.: The exemplary RW device handler with a `HasActionsIF` and a `HasModesIF` interface.

The interface symbol is a colored circle (see Figure 4.19).

Example

When the device handler of a RW of *Flying Laptop* is in `MODE_OFF`, the device is not powered and all handler outputs are invalid. When issuing a command to the device handler to start a transition to `MODE_ON`, it will power up the wheel, perform an initialization and confirm successful communication with the wheel. Only then is the mode changed to `MODE_ON`, which means the device is available for operations.

Similarly, commanding the attitude control component to `MODE_OFF` will disable any activities, which is useful for ground testing of equipment. When activated, the submode indicates the exact control strategy, i. e. a pointing or safe mode.

Definition

HasModesIF
+MODE_OFF : Mode_t = 0 +MODE_ON : Mode_t = 1 +SUBMODE_NONE : Submode_t = 0
+getMode(mode:Mode_t, smode:Submode_t *):void* *#checkModeCommand(mode:Mode_t, submode:Submode_t, maxDurationMs:uint32_t*) : ReturnValue_t* *#startTransition(mode:Mode_t, submode:Submode_t) : void* *#setToExternalControl() : void* *#announceMode(recursive:bool) : void*

Figure 4.20.: Signature of the `HasModesIF` interface.

The `HasModesIF` itself requires a component to implement the following method calls:

- `getMode`: Read out the current mode.
- `checkModeCommand`: Check an incoming request to change the mode.
- `startTransition`: Start a transition to a new mode.
- `setToExternalControl`: Take component under direct control.
- `annonceMode`: Announce the current mode in an event.

The full signature is given in Figure 4.20. The interface allows a fine-grained access to the mode properties of components, and pre-checks of allowed transitions before issuing the actual command.

An example shall illustrate the need for the `setToExternalControl` call: Usually, the mode of the RW 0 device handler depicted in Figure 4.19 is under control of a RW assembly component. So, if the assembly mode is `MODE_OFF`, all device handler components are supposed to be in `MODE_OFF` as well. If a manual checkout of a single wheel is required, it is necessary to indicate to the assembly that it shall not disable it during this manual checkout, which is done with the `setToExternalControl` call. This is further detailed in Section 4.5.3.

Effects

The `HasModesIF` is broadly used within the FSFW. It is quite useful to command mode changes of any component uniformly, as it simplifies the number of commands needed to operate different components. With this setting, it covers the features C.2.5, S.1.5, S.2.4, and O.1.5 of the domain analysis.

The unification of mode commanding unfolds its true power in conjunction with the subsystem and assembly components: These components can control a number of different components, i. e. controllers and device handlers, and therefore are able to orchestrate mode changes of entire subsystems. This is further described in Section 4.5.3.

4.3.3. `HasHealthIF`

Due to the harsh environment to which spacecraft are exposed, faults in spacecraft equipment have to be expected during a mission. A typical means to mitigate faults is to include redundant devices. To decide and remember which of multiple redundant devices to use, some information about the health state of equipment is required, e. g. to indicate if a temperature sensor is broken or a complex digital star tracker unit has failed. The `HasHealthIF` provides means for convenient management of this *health state* of a component.

Figure 4.21.: RW device handler with `HasActionsIF`, `HasModesIF` and `HasHealthIF` interface.

In illustrations, it is represented by a circle with a cross (see Figure 4.21).

Example

Consider a situation in which RW 0 of *Flying Laptop*, shows signs of degradation and must be disabled permanently. By setting its health state to `faulty`, the RW assembly controlling all wheel device handlers will automatically disable the wheel by setting its mode to `MODE_OFF`. The assembly itself keeps its own operational mode as long as the remaining three wheels are available. Even after the entire assembly was disabled, e. g. during safe mode, it will not try to activate RW 0 until the health state is reset to `healthy`.

Definition

As seen in Figure 4.22, the interface provides two methods for callers, a `getHealth` and a `setHealth` call.

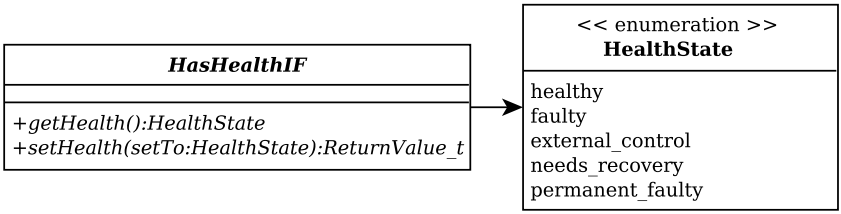

Figure 4.22.: Signature of the `HasHealthIF` interface.

The interface also defines possible health states of components:

- `healthy`: Unsurprisingly, the health state indicates that the component is working and available for operations.
- `faulty`: In opposition to `healthy`, this state indicates some fault in an equipment, which therefore shall not be used anymore.

- `external control`: This state is relevant in conjunction with the mode hierarchies described in Section 4.5.3. It defines that a component is under direct control of an operator and shall not be subject to autonomous mode changes.
- `needs recovery`: To provide a convenient method to restart devices for recovery activities, the FSFW introduces this health state. It represents a transient state and either results in the component being faulty or healthy again.
- `permanent faulty` is a stronger version of `faulty`. In some critical cases, where health states need to be ignored, this state hints that a component should be ignored as long as possible.

By convention, if a component's health state changes, it does not change its mode by itself. Instead, a component that is on a higher hierarchical level is responsible for changing the mode as an effect of the health state, as the RW assembly does in the above example.

Effects

The interface is most useful for device handler components, to represent faults in the associated hardware. With that use of the interface, feature S.2.2 of the feature tree (Figure 3.21) is covered. Also, the `needs recovery` state is useful to initiate a power cycle of equipment, which helps getting rid of transient faults in devices.

However, the interface is used in controller and subsystem components as well. Allowed health states are typically limited to `healthy` and `external control` to indicate that the component is under ground control.

4.3.4. `HasMemoryIF`

Data handling systems of spacecraft maintain different types of memory, e. g. a PROM including the software image or telemetry packet stores. Those memory regions may be local, i. e. in the same address space as the FSW, or remote, e. g. in a dedicated mass memory unit. More so, smart sensors or actuators may expose memory, e. g. for software updates. During nominal operations, there should be no need to access such memory directly. However, the remote nature of spacecraft requires means to do so for maintenance or in contingency situations.

The `HasMemoryIF` provides a common interface to allow low-level access of memory resources in the system. It is represented by a segmented square (see Figure 4.23).

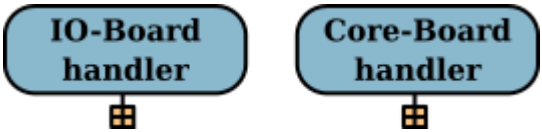

Figure 4.23.: Both the Core and the IO-Board handler provide the `HasMemoryIF` interface.

Example

Loading a new software version to the OBC is a typical use case for accessing memory directly. In *Flying Laptop*, this happens by loading the entire software image to a memory region in the IO-Board, before copying it from there to the PROM banks in the OBC.

As both the IO-Board and the Core-Board handler component implement `HasMemoryIF`, uploading the memory happens with the same command type from ground. Namely, PUS memory load requests are sent, which are translated by the memory service component to access the components' interfaces. The fact that the IO-Board handler needs to convert the call to a SpaceWire message, which is then forwarded to the board, is encapsulated in the handler component.

Definition

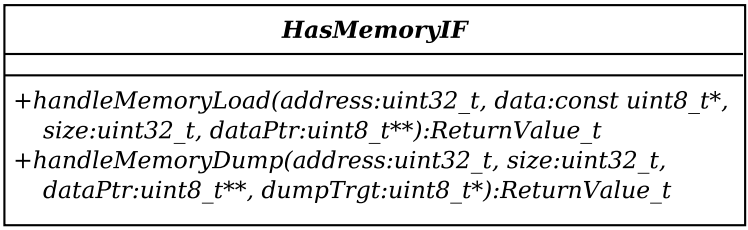

Figure 4.24.: Signature of the `HasMemoryIF` interface.

The `HasMemoryIF`'s signature, as seen in Figure 4.24, contains a `handleMemoryLoad` and a `handleMemoryDump` call. A component implementing the interface shall provide a pointer to the memory address at which the operation aims, so copying data can take place. In case the target address does not exist or access is restricted, the call returns a failure code.

As already mentioned in the example, the exact mechanism of accessing the memory is an implementation detail of the component and therefore fully encapsulated. To allow forwarding the call to remote entities, it is possible to defer completion of the call.

Effects

The `HasMemoryIF` interface as defined by the FSFW allows components to expose access to raw memory in a uniform manner. This is a powerful tool for remote maintenance, as low level checks and updates are possible.

It is tempting to use the interface for nominal operational activities, e. g. to adjust parameters in controller components. However, dedicated interfaces such as the `HasParametersIF` should be used for such purposes, as they are much safer, e. g. by performing type and range checks, than fiddling in the raw memory of a safety-critical program.

The `HasMemoryIF` covers features O.1.4 and S.3.2b found in Section 3.7.

4.3.5. `HasParametersIF`

Every software has configuration parameters, which allow adjusting controller gains, the number of retries of a recovery loop, or any other default setting of an algorithm. For spacecraft, having externally adjustable parameters is reasonable, as they allow minor modifications of the FSW without the need for a complete software upload and restart. This would be necessary if parameters were fixed at compile time.

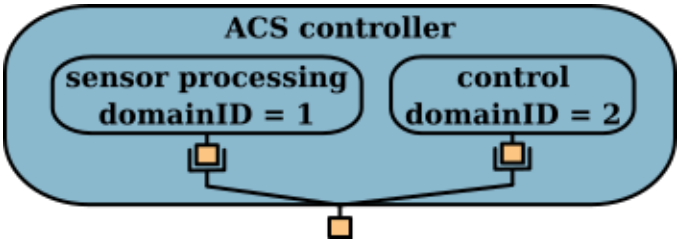

Figure 4.25.: An attitude control component which exposes parameters with the `HasParametersIF` and internal building blocks.

For this reason, the FSFW features the `HasParametersIF` which allows accessing and modifying parameters in components. Parameters are fixed size

variables, vectors or matrices with a well-defined plain old data (POD) type[9]. The representing symbol is a square (see Figure 4.25).

Example

The `HasParametersIF` interface is commonly used in controller components, to fine-tune ground settings and control gains. The attitude controller of *Flying Laptop* for example, exposes more than 70 parameters with that interface. This includes the orientation matrix of all RWs, which can be adjusted if, for some reason, the orientation of a wheel or the numbering is wrong.

However, other components can have parameters as well: The RW device handler has a mode which activates the wheel and sets some idle speed, which is adjustable using the `HasParametersIF`.

Definition

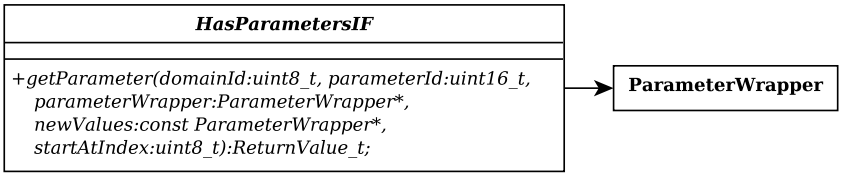

Figure 4.26.: Signature of the `HasParametersIF` interface.

A component can expose any member variable with this interface. To do so, the interface defines a single call, which is `getParameter` (see Figure 4.26). This call is sufficient, as the implementing component only returns a reference to the parameter, wrapped in a special `ParameterWrapper` class, and expects the caller to either adjust or dump the parameter. In case of a parameter load, the new value is provided as well, so components can perform a range check if necessary. By using some smart template metaprogramming tricks, accessing parameters is type and range safe.

Effects

The `HasParametersIF` is a useful tool to allow run-time fine-tuning of a FSW. The simple interface minimizes overhead in the implementing component and allows uniform access to any kind of adjustable value from ground. The interface covers feature O.1.3 of the feature tree.

[9] I. e. characters, integers, or floating-point numbers

In cases where components expose many parameters, which may even be nested in reusable building blocks, unambiguous addressing is important. For this reason, the 32 bit parameter address is split into three segments: A domain ID, a parameter ID, as well as an index to identify the parameter's position within a vector or matrix. Thus, components can maintain internal address spaces, if necessary. The concept is illustrated in Figure 4.25.

4.3.6. Summary

The common interfaces defined by the FSFW cover the majority of functionality a component can provide to other components, as was validated from experience with the *Flying Laptop* software. Unification simplifies control, which facilitates both ground and inter-component interaction:

- For an operator in the ground segment, uniform interfaces reduce the type of possible interactions with components and therefore the effort to understand their behavior. Moreover, common interfaces limit the number of command types, which reduces maintenance effort for ground station configuration data. Thus, the approach supports feature O.1.1 of the operational features.
- Within the FSW, common interface definitions enable the implementation of more complex algorithms, relying on the encapsulation and unification the interfaces introduce. Examples for such algorithms are subsystem components (Section 4.5.3) or the FDIR mechanisms (Section 4.7).

There are some cases where these common interfaces are not the right fit, e. g. for distributing clock information. Therefore, other interface definitions exist within the FSFW, which are described when needed. Also, mission-specific extensions are possible.

4.4. The FSFW-Core

The *FSFW-Core* designed in this thesis is the actual framework part of the Flight Software Framework. Its main purpose is to provide the infrastructure for component interaction and execution, as outlined in Section 4.1.3. To do so, it relies on functionality provided by the underlying RTOS where possible, e. g. by utilizing message queues and mutexes. Also, the FSFW-Core provides abstraction interfaces for components to access these features directly, which together form the operating system abstraction layer (OSAL) of the FSFW-Core.

For example, the RTOS scheduling features are used to facilitate component execution, by assigning components to different types of tasks for periodic or slotted execution.

For interaction, the FSFW-Core ensures that all communication between components is asynchronous and therefore real-time compatible (see Section 2.3.3). It does so by providing message-based interaction with a *software bus* and *event message distribution*, as well as a shared memory-based *data pool*.

In addition to internal communication, the FSFW-Core is responsible for allowing components to use hardware interfaces for external communication. For this reason, the core defines another interface, which must be implemented by drivers, or wrappers of legacy drivers, to be accessible by components.

Moreover, the FSFW-Core provides access to a common clock source and data containers for storage of run-time adjustable configurations.

4.4.1. RTOS Abstraction

To avoid any specific details of an RTOS to leak into the framework, the FSFW-Core defines a set of interfaces over which RTOS features are accessible.

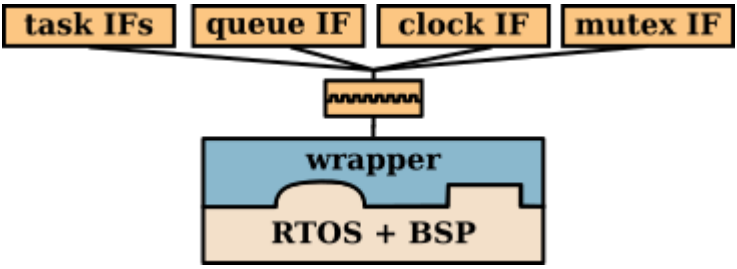

Figure 4.27.: RTOS interfaces defined by the FSFW-Core.

As shown in Figure 4.27, these interfaces are:

- **Task interfaces:** Allow an implementation to control tasks, e. g. start or stop them, as well as adding components for execution.
- **Queue interface:** Allows components to send and receive messages.
- **Clock interface:** Components can use this interface to read or adjust a clock in different formats.
- **Mutex interface:** With this interface, mutexes are controlled, which lock shared resources to avoid data corruption. This interface is used in the FSFW-Core mainly, not in individual components.

This set of interfaces is collectively called the operating system abstraction layer (OSAL).

Apart from the clock interface, all of these interfaces describe software elements of which components need individual instances. To generate these instances, for each interface a so-called *factory* is defined, which creates and returns instances of the elements[10]

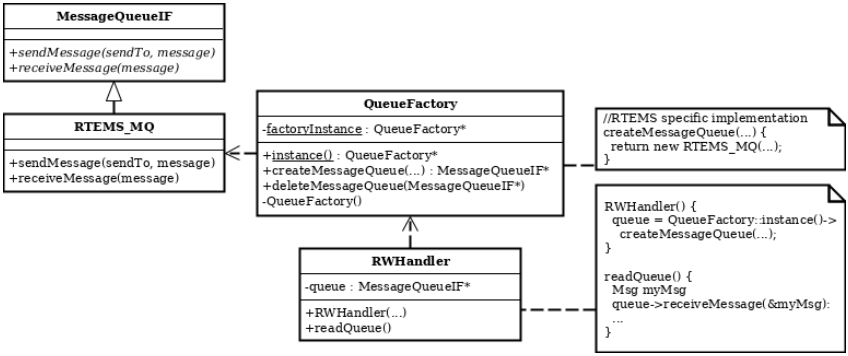

Figure 4.28.: Simplified example of creating a message queue for a component, using the queue factory.

For a given RTOS, implementations for both the concrete elements and the factories need to be provided. In practice, this happens with wrapper classes implementing the interfaces.

For example, Figure 4.28 illustrates the creation of a message queue for the RW device handler of *Flying Laptop* using RTEMS as RTOS. As shown, the act of getting the factory instance and creating a message queue happens in a single call. After creation of the queue, the component can use it via the interface, fully agnostic of the actual implementation. The other interfaces are used analogously.

4.4.2. Component Execution

To achieve temporal decoupling, the FSFW is capable of individual scheduling of components. A fundamental assumption made is *periodic execution*, i. e. all components execute their tasks within a fixed period. As this is the common scheduling scheme for real-time systems, this is a reasonable limitation.

[10] A `factory` is a design pattern classic from [60]. Factories are implemented using another pattern, the `singleton`.

Components become executable by implementing the `ExecutableIF`, which mandates implementation of a `performOperation` method.

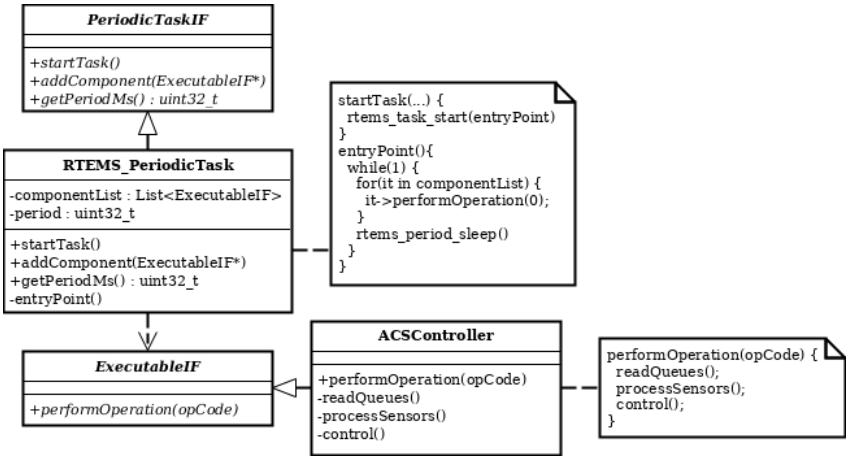

Figure 4.29.: Simplified example class diagram of a task class executing the attitude control component via the `ExecutableIF`.

This method is called by task objects, which manage cyclic execution. These objects implement one of the aforementioned task interfaces.

This means the `performOperation` method of a component is executed and, upon completion, called again after a certain period. Thus, the periodicity is not defined by the component itself, but by the task executing it.

The component itself is persistent, i. e. all attributes and parameters of the component maintain their values between calls of `performOperation`. For example, the attitude control component shown in Figure 4.29 maintains internal states between calls to `performOperation`.

The assignment of components to task objects typically happens in the overall main file of the FSW. For reasons of efficiency, multiple components can be grouped into one task, which then execute with the same period and in the order of registration.

Apart from the assumption of periodicity, the FSFW does not mandate any specific scheduling algorithm. Thus, any form of periodic scheduling devised for real-time systems is possible, e. g. a fixed run-to-completion schedule or priority preemptive scheduling. For a summary of scheduling algorithms, see e. g. [13]. Parallel or quasi-parallel execution of components is possible, as all communication techniques of the FSFW-Core are thread-safe.

Device Handler Component Execution

While the described task management is convenient for typical control components, device communication, especially via shared lines and buses, often has more strict timing requirements. Here, it is necessary to create precise schedules to optimize utilization of the communication lines and adhere to equipment requirements. This communication is typically planned with a so-called channel acquisition scheduling table or *polling sequence table (PST)* [51].

For illustration, consider the interface setup on *Flying Laptop*, in which equipment communication is channeled through a SpaceWire line to the IO-Board (see Section 3.3.2). This poses some challenges on scheduling of device handler components:

- The handlers need to invoke read and write calls from the IO-Board at precise points in time. So, a single period is not sufficient.
- To avoid overloading the SpaceWire line, not all equipments should command at exactly the same time, but calls should be evenly distributed.

These challenges are even more severe in other configurations, e. g. when using a bus such as MIL-STD-1553.

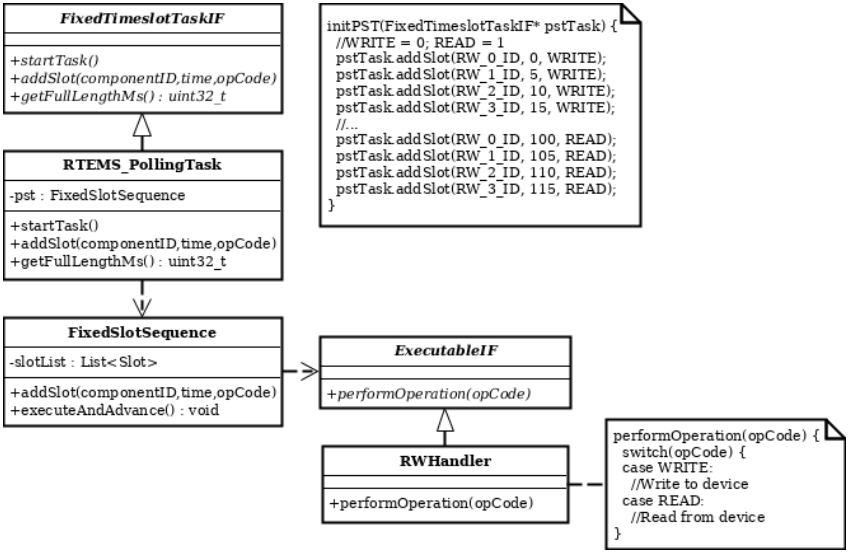

Figure 4.30.: Simplified example class diagram of a task class to execute components in fixed timeslots.

To handle such precise scheduling, the FSFW defines a dedicated tasking interface called `FixedTimeslotTaskIF`, which allows adding components in time slots with a fixed execution time. In addition, the `opCode` parameter of a components `ExecutableIF` is used to identify the execution position within a PST, i. e. if a write or read call is scheduled. A `FixedSlotSequence` class provided by the FSFW-Core supports implementation of the sequence list. Figure 4.30 shows an example class for RTEMS as used in *Flying Laptop*.

Together, these FSFW elements allow exact control over timing of device communication. However, due to the scattered execution of slots in a PST task, it poses some challenges on schedulability analysis. These are addressed in Appendix C.2.2.

4.4.3. The FSFW Software Bus

The FSFW software bus is one of the main methods for inter-component communication. The software bus operates with so-called *command messages* and *replies*, which are transmitted over the bus. Basically, it works as following:

- Every component which intends to communicate over the bus has a unique *component identifier (ID)*.
- Also, each such component has a message queue, which is provided in the form of an interface by the FSFW-Core.
- Any other component can use this ID to send a message from its queue to the destination component.
- The message protocol allows the receiver to identify the type of message and its sender.
- Looking up component IDs is possible at run-time with the help of a global component directory.

Most message types sent over the bus are related to the common interface definitions defined in Section 4.3.

For illustration, consider the reaction wheel device handler from *Flying Laptop* again. Another component, for example the device commanding service forwarding a ground command, wants to read telemetry, i. e. motor current and wheel speed, from a RW. Thus, it wants to invoke a "read telemetry" action via the action interface (see Figure 4.31). As discussed, a direct call is not allowed, so invocation must take place via messages. So, the service component sends an *action message*, which triggers execution on reception.

Likewise, the FSFW-Core defines a dedicated set of message types for every common FSFW interface, i. e. an action message, a mode command message

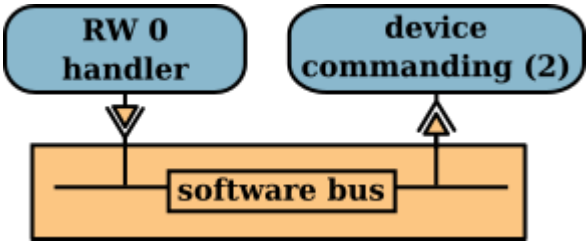

Figure 4.31.: The device commanding service component invokes an action of a reaction wheel device handler via the FSFW software bus.

and so on. These messages contain the parameters required to invoke the method call.

In total, to invoke the action request, the following steps take place:

- The calling component creates an action message, containing the parameters for action execution.
- It obtains the component ID of the receiving component and submits the message.
- When scheduled, the called component receives the message and identifies it as an action message.
- It extracts and checks action ID and parameters.
- With these parameters, the receiving component invokes its own `executeAction` call with parameters from the message.
- Results of the call are returned to the calling component in one or more messages.

To continue with the above example, the device commanding service would create the action message and send it to the RW device handler. On reception, this component would invoke its own interface call, generate a command to the RW device, and return the information to the service component (or indicate failed execution).

Invoking other calls of the common interfaces, e. g. reading a mode, setting a health state, or changing a parameter, happens in a similar fashion. In all cases, specific messages are sent to a component, which returns the results of a call in a reply message.

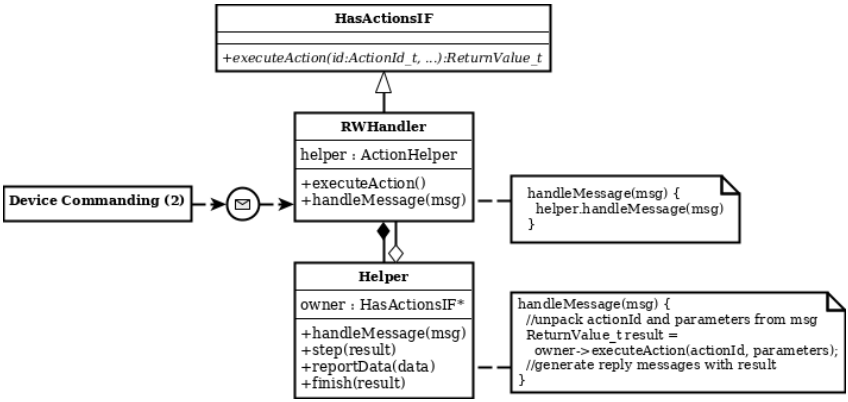

Figure 4.32.: Mapping of an action message to an interface call using the ActionHelper class.

Helper Classes

When looking at the invocation steps above, the act of identifying a message, extracting the parameters and calling the interface within a component only depends on the message format and the signature of the interface. Also, to generate reply messages, only the result of the interface call and the command message protocol is relevant.

Therefore, all steps from parsing an incoming message to returning the reply message are generic for a given common interface. To avoid programming that procedure multiple times, so-called *helper classes* are created. Helper classes encapsulate the generic behavior for each type of interface, e. g. there is an `ActionHelper` class, which effectively maps an action message to a method call of `HasActionsIF` (see Figure 4.32).

Instances of these helper classes can be part of a component and take over its entire command message protocol handling. This interaction forms a specific design pattern, similar to the classic `adapter` or `wrapper` pattern from [60].

Example

With that background, it is possible to complete the above example of interaction between device commanding service and RW device handler component.

The device handler component contains an instance of the `ActionHelper` class, which checks incoming messages for being action messages, parses them and

calls the device handler's `executeAction` method. In addition, the helper creates and sends reply messages to the calling component.

So, effectively, the `ActionHelper` bridges between an asynchronous, message-based protocol and a synchronous protocol defined by the `HasActionsIF` interface.

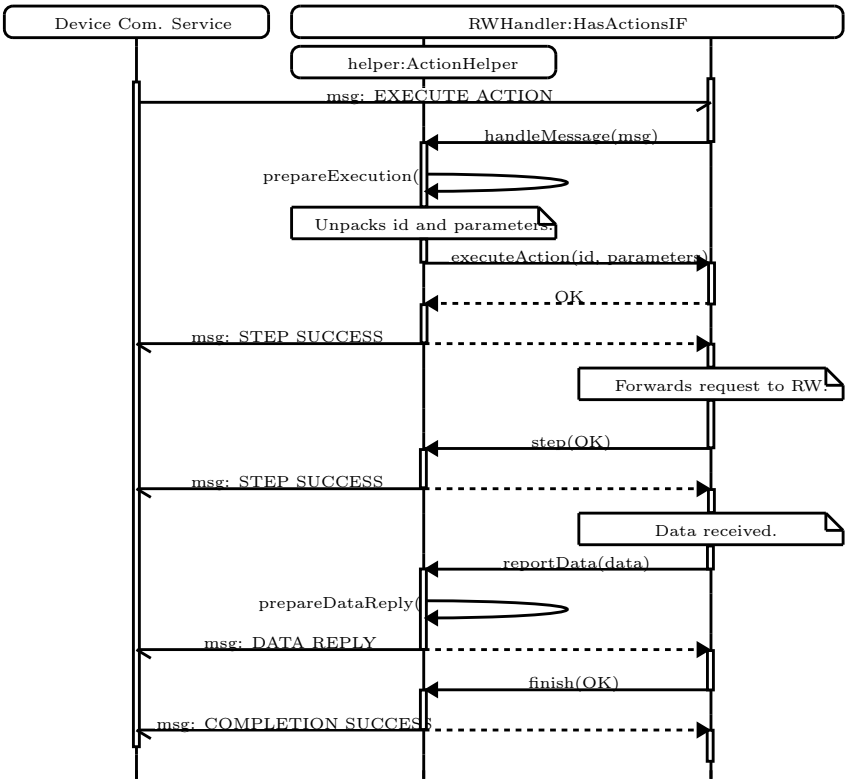

Figure 4.33.: Interaction of caller and component, including a helper object, when executing an action.

The action message protocol extends the interface protocol by introducing the possibility to report execution steps and return data to the caller. In total, it has the following properties:

- An action is initiated with a single dedicated command message which contains the action ID and parameters to pass.
- The executing component is not required to finish the action immediately. Instead, it may issue an arbitrary amount of step messages to inform

about progress of action execution. A “step failed” message aborts the transaction.

- In some cases, actions trigger the generation of data for the initiator. This data may be delivered in a dedicated data reply message before finishing or aborting the action.
- As a minimum, the caller expects a message indicating successful or failed finish of execution (or a failed step) of the action in a decent amount of time.

As shown in Figure 4.32, the helper class offers callbacks to manage reply messages.

For reading telemetry data from a RW, the full exchange of calls and messages between caller and receiver is illustrated in Figure 4.33.

For all other common FSFW interfaces, a similar mechanism is provided, using e. g. `ModeHelper` or `ParameterHelper` classes. Some of those helpers may execute multiple interface calls for a single incoming message request, e. g. to allow the implementation to perform a sanity check before actually executing the command. For reasons of brevity, however, similar sequence diagrams of other framework interfaces have been left out.

Effects

The FSFW software bus allows sending messages between components for asynchronous data exchange. This message passing mechanism is enhanced by linking it to the common FSFW interfaces using dedicated message protocols and helper classes.

This is, in fact, a form of remote method invocation (RMI), which is typically found in middleware frameworks, such as the Common Object Request Broker Architecture (CORBA) or Simple Object Access Protocol (SOAP). The helper classes of the FSFW are similar to proxies used in service-oriented architecture (SOA). However, as these frameworks are intended for heterogeneous, distributed systems they come with many features that are irrelevant for well-defined embedded systems. So, the intention of the FSFW software bus is to use a similar mechanism for component communication, with a minimum of overhead.

To achieve this, the FSFW-Core provides the infrastructure, in the form of message definitions and helpers, primarily for the common interfaces introduced in Section 4.3. In case these interfaces do not fit, the FSFW-Core allows to use the messaging mechanism directly, i. e. without calling any interface, or implement mission specific extensions.

As a remark, the FSFW software bus is no real bus, for which components must compete for resources. Instead, it is more an umbrella term for the message queue and addressing mechanisms of the FSFW-Core.

In favor of a cohesive summary of the software bus functionality, other details, such as the mechanism to exchange large data chunks between components, have been omitted. Additional information is found in Appendix C.2.

4.4.4. Event Distribution

Events are aperiodic incidents in a system. Such events are typically generated at one point and consumed at one or multiple destinations. Therefore, a publish-subscribe mechanism is suitable for event distribution.

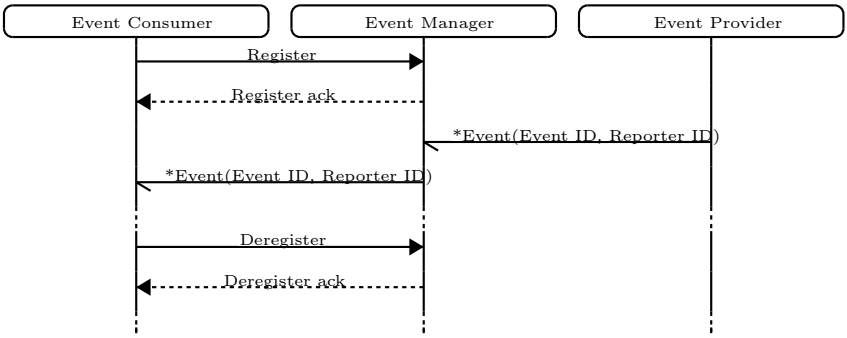

Figure 4.34.: Distribution of events with the event manager as broker of a publish-subscribe mechanism.

The FSFW provides an event manager, which takes the role of a *broker* for distribution of events (see Figure 4.34). It receives event messages generated by components and forwards them to those components that registered for a matching set of events. Components are capable of throwing events at any time.

An event message contains the following information:

- A system-wide unique *event ID*. The FSFW ensures uniqueness by assigning event domains to each software element defining events, within which events are unique.
- A *severity field*, which allows to distinguish between informational and failure events of varying criticality. The FSFW defines information events, as well as low, medium and high severity failure events.

Name	Event ID Domain	ID	Severity	Reporter ID	Parameters
MISSED_REPLY	28	05	low	RW_HANDLER_0	command ID
MODE_INFO	74	01	info	RW_HANDLER_0	current mode
CANT_KEEP_MODE	74	04	high	ACS_CONTROLLER	current mode
TIME_JUMP	77	02	low	TIME_CONTROLLER	none

Table 4.1.: Example event messages used in *Flying Laptop*.

- The *reporter ID*, which is the component ID to which the event is associated.
- Two *parameter values*, which can be used to transport additional information with the event.

Table 4.1 shows event definitions from *Flying Laptop* as examples. Many event definitions are pre-defined in the FSFW, such as the mode-related events in Table 4.1, but mission-specific extensions are possible by utilizing unused domain IDs. Every component can use any event definition, therefore the reporter ID is an integral part of an event message to identify its origin.

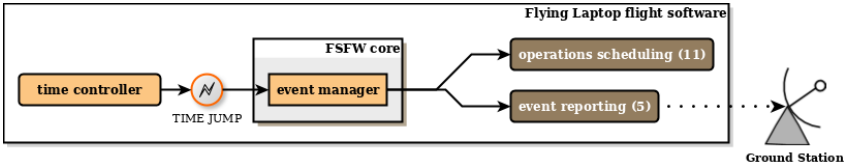

Figure 4.35.: Example of event distribution in *Flying Laptop*.

An example, depicted in Figure 4.35, will illustrate event distribution: It is desirable to disable the time-tagged command schedule in cases where the on-board time gets out of synchronization with the ground segment. In the *Flying Laptop* FSW the time controller takes care of monitoring the on-board wall clock. If a time jump larger then a certain threshold is detected, event distribution happens as following (see Figure 4.35):

- The time controller triggers an event message, which is forwarded to the event manager.
- The event manager forwards the event to all components with a fitting registration. In this case, this is the event reporting and the operations scheduling service component.
- The scheduling component receives the event and acts accordingly, e. g. by disabling the schedule and triggering an event itself (not shown).

- The event reporting service transforms the on-board event in a PUS event report and forwards it to ground.

Components can safely adjust their registration at run-time. For example, this is used in the event reporting service to ignore events for downlink.

As faults always qualify for “aperiodic incidents”, failure event messages play an important role for failure management in the system. The FSFW provides distinct features to implement FDIR mechanisms based on events, which are described in Section 4.7.

4.4.5. The Data Pool

Data pools are a typical element of embedded control software in general and flight software (FSW) in particular. The basic idea is to provide a possibility for exchange of periodic data with *blackboard* logic: If only the most recent value is of interest, it is reasonable to provide a single location for this value, like on a blackboard, and cyclically overwrite the current value with a newer one. As sensor and actuator data have a very short time of usage anyway - they become *stale* -, the data pool is the main mechanism for exchanging control data between components.

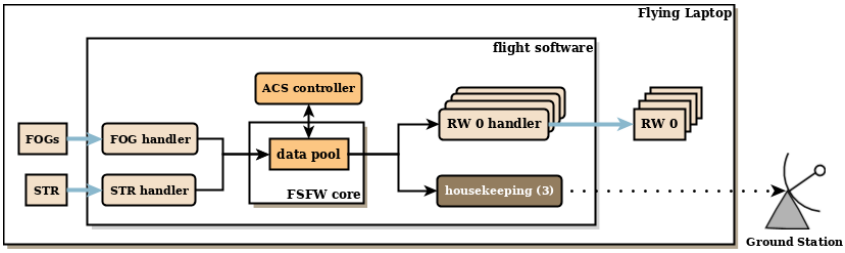

Figure 4.36.: Example use of the data pool for exchanging attitude control data.

The *Flying Laptop* attitude controller, for example, mainly relies on sensor data from the star tracker (STR) and fibre-optic gyros (FOGs), and actuation happens with reaction wheels. Thus, the respective software components exchange these values through the data pool, as illustrated in Figure 4.36. In addition, the housekeeping service component accesses the data pool to compose housekeeping packets for ground observation.

In principle, the data pool implementation of the FSFW-Core is based on a shared memory mechanism, which is segmented in individual variables, each with a unique *data pool ID*. In addition, the data pool provides the following features:

- The data pool allows storage of vectors and single variables of all POD types. Access is type-safe and possible at any time, i. e. thread-safe.
- A locking mechanism allows consistent checkout of a set of variables, i. e. a thread trying to overwrite parts of a set of variables is blocked if the set is currently read out. Thus, neither single variables nor a set of related values can contain inconsistent data.
- A valid flag is managed for each entry in the pool. It allows marking variables as valid or invalid, depending e. g. on the availability of a sensor.
- The data pool provides commit-and-rollback semantics for write access. This means that changes take place on local copies first, their value is written back to the pool with a dedicated `commit` call. When encountering an error condition, the local copy may be discarded.

In *Flying Laptop*, over 600 sensor measurements, actuator commands, and intermediate calculation values are registered in the data pool configuration.

4.4.6. Communication with Equipment

Interaction between the control system and its sensors and actuators happens via digital hardware interfaces. An embedded software framework is quite useless without some dedicated means to interact with peripheral hardware. However, no component, not even a device handler component, should depend on a certain interface, as this would impede reuse when using the component on another mission.

For example, on *Flying Laptop* the RW device handler communicates with the RW device via the IO-Board using a SpaceWire line. In another spacecraft, the same RW may connect to the OBC on a multidrop bus.

To allow reuse of components in such cases, the FSFW-Core defines a communication interface called `DeviceCommunicationIF`. It works with the assumption that received data is polled by a component. The interface provides calls to open and close the interface, as well as adjust options. Also, there are four steps defined for the actual communication:

- Send data to a device.
- Get an acknowledgement for sending.
- Request reading data from a device.
- Read the received data with a dedicated call.

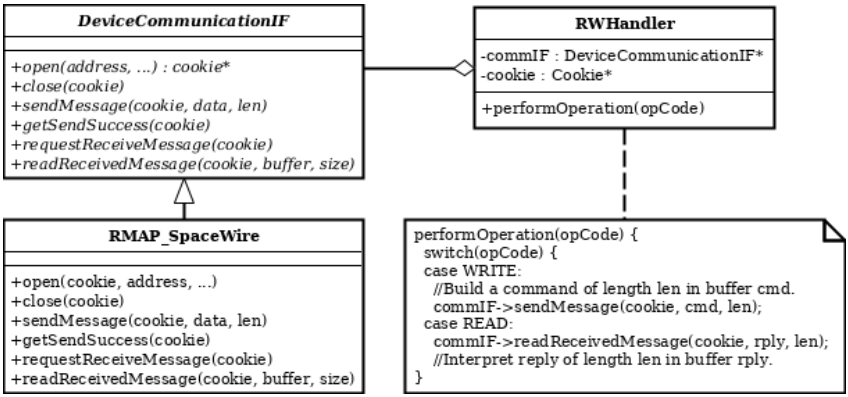

Figure 4.37.: Example usage of the `DeviceCommunicationIF` within a device handler component.

The four steps of communication with equipment provide more flexibility to map the interface to various underlying communication mechanisms than a simple `read` / `write` scheme. Also, the approach fits better to real-time systems which are not using blocking interface calls.

By implementing the interface, drivers, or wrappers for legacy drivers, become usable by components. To identify different connections over a single interface, drivers can return so-called *cookies* to components[11].

As an example, the *Flying Laptop* peripheral connection happens via SpaceWire and the RMAP protocol. As stated in Section 4.2.1, a dedicated driver implementing the interface was developed. The RW device handler component is configured on initialization to use this driver, and is then capable to communicate with the RW device (see Figure 4.37). In case another interface is used, the code of the RW device handler does not change.

4.4.7. Date and Time

In spacecraft, time and data management is more important than in many other embedded systems, as remote operation and precise maneuvers require well synchronized clocks between ground and space segment. The FSFW-Core relies on time management features of the underlying RTOS, making it accessible for components with a dedicated clock interface.

[11] These cookies are similar to sockets of the socket API.

As computers typically use the so-called POSIX time internally [69], but spacecraft timestamps are defined in Consultative Committee for Space Data Systems (CCSDS) time [21], the FSFW-Core also provides a library called `CCSDSTime` for convenient conversion between different time formats.

In addition, there are dedicated message definitions to pass timestamps of events between components, e. g. to distribute the exact time of an incoming pulse-per-second signal.

4.4.8. Data Containers

The FSFW-Core provides a number of container elements for storage of runtime adjustable data. Most of these containers are replacements for STL building blocks, such as lists and maps, with a maximum size determined at compile time. This is necessary to avoid dynamic memory allocation at run-time.

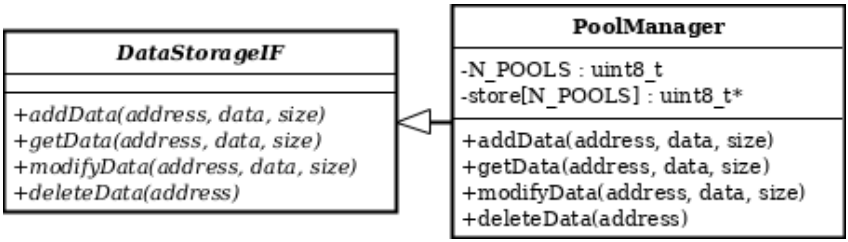

Figure 4.38.: Simplified signature of the `DataStorageIF` interface.

In addition, the FSFW-Core defines an interface to safely store and retrieve variable-sized chunks of data, the `DataStorageIF`. Its signature is shown in Figure 4.38. The core also provides a compatible implementation, called `PoolManager`, which uses a fixed-size pool-based memory allocation scheme.

This storage facility is used in various places within the FSFW-Core, e. g. for efficient large data distribution (see Appendix C.2.1).

Also, it serves as the memory back end for a building block called `PlacementFactory`. It allows creation of arbitrary objects, whose types are determined at compile time, in reserved memory at run-time. This is particular useful to efficiently create and delete temporary or configurable objects of variable size. For example, the *Flying Laptop* housekeeping and operations scheduling components rely on this factory to maintain their housekeeping report definitions and the TC schedule, respectively. As it is a good example of the advantages of C++ features such as template metaprogramming, its full source code is shown in Appendix A.1.1.

4.4.9. Summary

The FSFW-Core is the central element of the Flight Software Framework design, which allows FSW developers to focus on the essential complexity of a certain component instead of worrying about interaction and execution infrastructure.

Moreover, the FSFW-Core codifies the overall component-based software architecture, and therefore aids using the FSFW. It is the key factor for introducing reusability in FSW development, as it mediates between components and the underlying execution platform. Its own reusability is ensured by relying on the OSAL interfaces to provide functionality to components.

With the infrastructure the FSFW-Core provides, it covers most features identified for component management, as illustrated in Figure 3.20. Particularly, RTOS abstraction (C.1.3) and inter-process communication (C.2.1) are achieved. Also, the device communication interface covers the system management feature for communication layering (S.3.1) and subnetwork access (S.3.2).

4.5. Component Templates

In principle, it is possible to use the interfaces and communication techniques defined by the FSFW and implement every component required for a system from scratch. However, there are strong similarities between components with a similar purpose, e. g. for device handling.

Therefore, the FSFW offers a set of *component templates*, which reduce the effort of component creation by providing default implementations for recurring activities. The FSFW identifies four types of components:

- Device handlers
- Controllers
- Subsystems
- Ground services

The FSFW offers component templates for each of these types. However, this section provides insight in only three of them: As ground services depend on the space link protocol, these templates are part of the TMTC framework described in Section 4.6.

4.5.1. Handling Spacecraft Equipment - `DeviceHandlerBase`

Purpose of the `DeviceHandlerBase` template is to simplify implementation of device handler components. To provide a useful template, it is necessary to identify commonalities between the handling of very different equipment, e. g. a reaction wheel and a PCDU, and put it in a base class[12].

Fortunately, the domain analysis synthesized in Section 3.7 already identifies a list of common features for equipment in S.1 features. These are implemented in the `DeviceHandlerBase` class, which is described below.

An important aspect here is that device handlers always act as a mediator between other software components and a specific equipment, so there are always two perspectives on a given functionality: That of the equipment and that of the FSW, i. e. of other components.

To illustrate these views and the common features, an example is introduced first.

Example

Once again, the reaction wheel (RW) device of *Flying Laptop* shall serve as an example. Like all off-the-shelf equipment, the device comes with a technical description in the form of an interface control document (ICD), which describes the properties of various interfaces, e. g. mechanical or electrical.

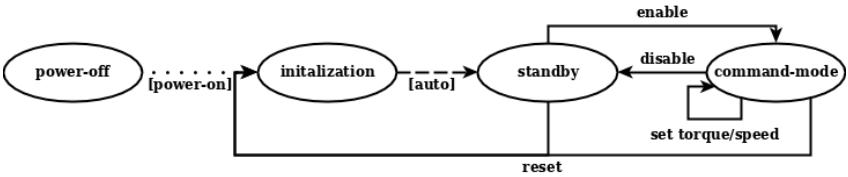

Figure 4.39.: Mode diagram and commands of the *Flying Laptop* reaction wheel. Transitions in brackets happen due to power-up or automatically. From [93].

From a software perspective, the modes of operation and the communication protocol are the most relevant. The ICD of the *Flying Laptop* RWs describes the following modes and protocol [93]:

[12] As discussed in Section 4.2.2, simple sensors such as a single analog temperature sensor are not regarded here.

- There are four modes defined, which are power-off, initialization, standby and command-mode. Initialization is a transient mode after power-up and moves to standby mode automatically. From there, an external command is necessary to bring the device in command-mode.
- Communication works in a strict command-response scheme, i. e. the device sends a single reply for every command.
- On OSI application layer, the device communicates with an ASCII character protocol, with the main command format being `[AA,CC,,DDDD,ZZ]`. `CC` identifies a command code, `DDDD` data to transmit[13].
- Available commands are setting a wheel speed or torque, requesting telemetry, and changing the mode, i. e. commanding from standby to command-mode and vice versa, or resetting to initialization mode.
- When commanding a speed or torque, the data field `DDDD` contains the desired value converted to a 16 bit signed integer in a vendor specific fashion.

Modes and commands are depicted in Figure 4.39.

As described in Section 4.4.6, the wheel is connected to the OBC via the IO-Board. It is powered by the PCDU, which has command-controllable power switches.

Equipment Modes

An important aspect of controlling a device such as the described RW is to handle equipment modes (feature S.1.5 in Figure 3.21).

From the equipment perspective, this means managing the equipment itself, e. g. commanding power switches and sending an enable command in case of the reaction wheels. `DeviceHandlerBase` supports equipment mode management by providing a state machine which performs transitions in case a mode change is commanded by another component. It organizes the following activities:

- Sending power switch commands to a component that provides the `PowerSwitchIF`. In case of *Flying Laptop*, this is the PCDU device handler. The switch (or switches) to control is configured on initialization.
- Allowing subclasses to configure the device by implementing specific transition methods, for start-up, shutdown, and other transitions. The RW handler, for example, sends a reset command to the wheel during start-up to confirm availability.

[13] AA is an address, which is not used for *Flying Laptop*, and ZZ a checksum.

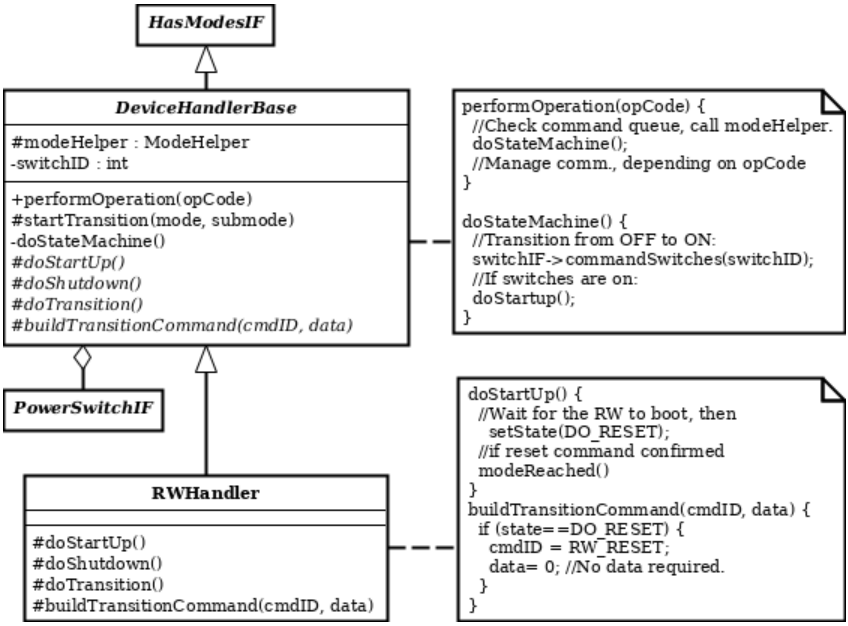

Figure 4.40.: Extract of the `DeviceHandlerBase` class diagram to illustrate mode management functionality.

To adjust the behavior of the state machine to specific properties, `DeviceHandlerBase` defines a number of mode-related abstract methods as adaptation points, as shown in Figure 4.40.

From a FSW perspective, i. e. from other components, the `HasModesIF` is used to adjust the operational state of a device. From that point of view, a common set of modes is desirable, as uniformity helps creating an abstract perspective of device handling. This simplifies handling of redundancies and subsystem modes.

Therefore, the `DeviceHandlerBase` class defines common modes for device handlers, which are:

- `MODE_OFF`: The device is not powered and considered off.
- `MODE_ON`: The device is powered and configured for operation, but supposed to be passive, i. e. components neither send periodic commands nor poll sensors.
- `MODE_NORMAL`: Components periodically request data from sensors and command actuators. Values are stored to and taken from the data pool.
- `MODE_RAW`: A dedicated contingency mode, in which the device handler component relays communication "as-is" from external sources, e. g. from and to the ground system. No commands are generated, and responses are not interpreted in the handler.

As stated above, `DeviceHandlerBase` takes care of recurring activities such as power switching for devices. Also, `MODE_RAW`, in which the device handler component simply relays commands and replies, is entirely implemented in `DeviceHandlerBase`. Still, it is up to specific component implementations to map equipment modes to the common device handler modes and ensure correct transitions.

For the RWs, for example, `MODE_ON` is mapped to the standby mode of the equipment and `MODE_NORMAL` to command-mode, in which the RW handler additionally sends torque commands and requests telemetry from the wheel (see below). The commands for equipment transitions are also provided by the RW handler.

Equipment Access and Value Conversion

Another main functionality of device handler components identified in the domain analysis is device access (feature S.1.2). This again, is a two-fold activity:

- Communication with the equipment in its native protocol must be ensured.

- The device and its measurements ought to be accessible to other components, e. g. for direct ground commands.

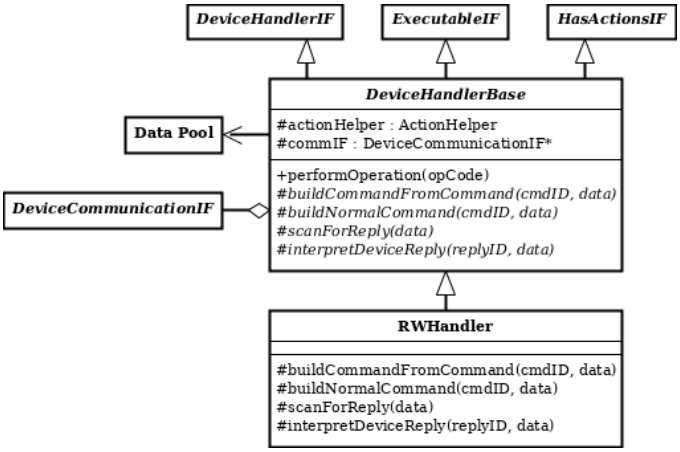

Figure 4.41.: Extract of the `DeviceHandlerBase` class diagram to illustrate equipment access.

The `DeviceHandlerBase` class handles much of the first point by managing a device communication interface of the FSFW-Core, as described in Section 4.4.6. This covers all low-level communication, i. e. below OSI application layer. The base class also defines a communication scheme using the slotted means of device handler communication described in Section 4.4.2, which ensures cyclic commanding and reply reception to and from the equipment.

The remaining task for a subclass is to implement the actual application protocol of an equipment, for which `DeviceHandlerBase` provides a number of adaptation points (see Figure 4.41):

- `buildCommandFromCommand`: Build a command for a device, with a command ID and optional parameters as input. Incoming parameters are in a generic format, typically in SI units. Thus, the subclass is responsible for creating the correct protocol and converting parameters. The RW handler, for example, creates the bracket-based ASCII-protocol in which converted torque values are inserted.
- `scanForReply`: Incoming data from the device is forwarded to the subclass for basic format checking and identification of a reply ID. For example, the RW device handler calculates and checks a message checksum.
- `interpretDeviceReply`: If formal checking was successful, `DeviceHandlerBase` calls this method, in which subclasses extract data from the raw

device reply and format the data for external use, e. g. writing it to the data pool. The RW handler for example, converts the raw wheel telemetry to SI units.

The two-step handling of equipment replies in device handlers separates the formal checking of the application layer protocol and the interpretation of the content.

To complete the concept of equipment communication with `DeviceHandlerBase`, the following part explains how sending commands to devices is actually initiated, and in which format. In total, there are three methods foreseen in `DeviceHandlerBase` to start interaction with the hardware device:

- **Automatic commanding in `MODE_NORMAL`:** In that mode, the device handler itself is responsible for creating a command for the device. Command parameters, e. g. the torque for a RW command, are taken from a variable of the data pool. Likewise, sensor measurements are written back to the data pool. A subclass can create such commands by implementing the `buildNormalCommand` method.
- **High-level commanding with the `HasActionsIF`:** By submitting an action to a device handler, other components can communicate with the device without knowledge of the device specific protocol. This is a step towards *virtualization* of devices, as proposed in CCSDS SOIS (Section 3.4.3).
- **Raw access with the `DeviceHandlerIF`:** Raw commands allow communication with the device in its native protocol, circumventing any form of translation and safety measures, e. g. range checks. This is intended for contingency situations and only available if the device handler is in its `MODE_RAW` mode.

The `DeviceHandlerBase` class provides the infrastructure for these three methods, and takes care of communication with other components. Since protocol conversion for high-level commanding is already implemented in the aforementioned adaption points, no further effort is necessary in a subclass. This also covers feature S.1.3 for value conversion.

For illustration, Figure 4.42 shows a sequence diagram of reading telemetry from a RW in `MODE_NORMAL`.

Equipment Monitoring

As identified in feature S.1.1, another important activity is equipment monitoring. A good start for monitoring is to check communication with a device. If, for example, no reply is received from a RW device, this is a first indication of an error.

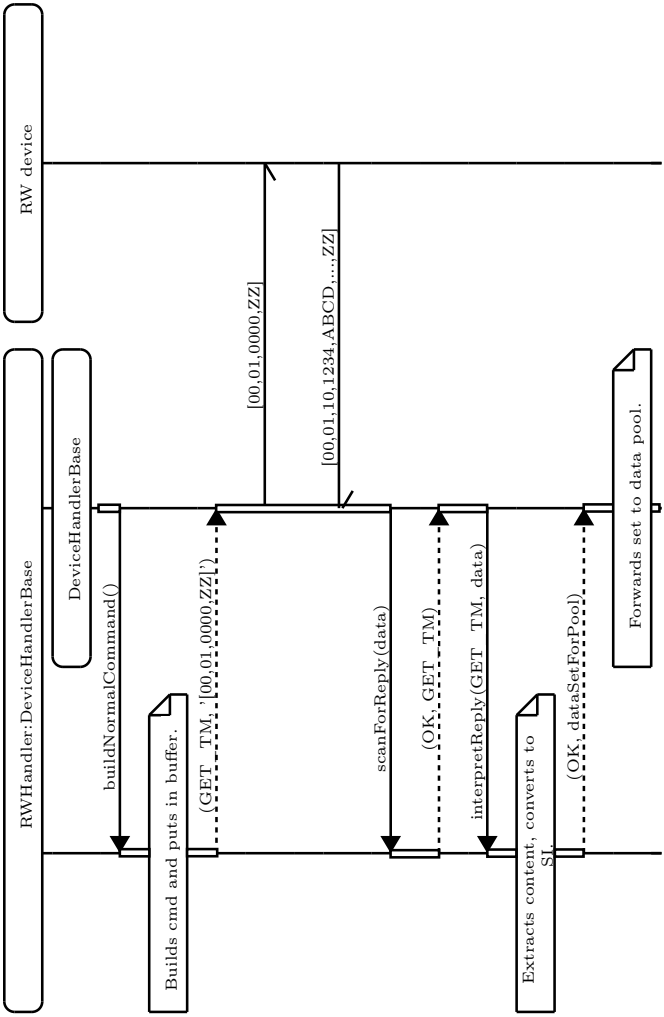

Figure 4.42.: Sequence diagram to show the interaction between `DeviceHandlerBase` and a subclass, here a RW handler, to create a command in `MODE_NORMAL`.

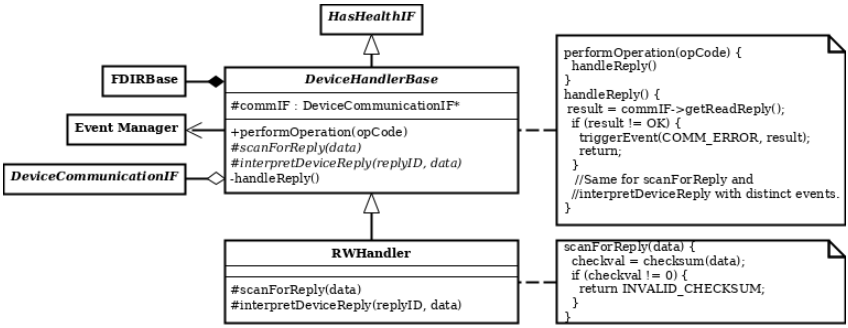

Figure 4.43.: Extract of `DeviceHandlerBase` with elements relevant for monitoring.

Since device handlers already provide the protocol and the generic structure of data exchange, it is possible to monitor device communication in the base class. This happens by checking the duration of reply reception and reacting on results of `scanForReply` and `interpretDeviceReply` from the subclass. Also, `DeviceHandlerBase` takes errors of the communication interface into account. For each such error, `DeviceHandlerBase` triggers a failure event, relieving subclass implementations from doing so. This communication monitoring is an essential part of device failure detection, since a broken device will likely produce incorrect or no replies.

Aside from detection of communication errors, device handler components are responsible for collecting and interpreting failure events related to their device. This is part of the distributed hierarchical FDIR concept of the FSFW and further described in Section 4.7.

The elements relevant for equipment monitoring of `DeviceHandlerBase` are shown in Figure 4.43.

`DeviceHandlerBase` also implements the `HasHealthIF` interface, so a device handler component represents the health state of its equipment. Being a common FSFW interface, changing the health state is possible by every component. In practice, however, a health state is adjusted by the component's FDIR or the ground segment, such as the health commanding service of *Flying Laptop*.

Equipment Representation

The remaining point found in the domain analysis is equipment representation (S.1.4), which is somewhat more vague than the others. However, as discussed

in Section 4.1.5, it is already covered by introducing device handlers in the first place.

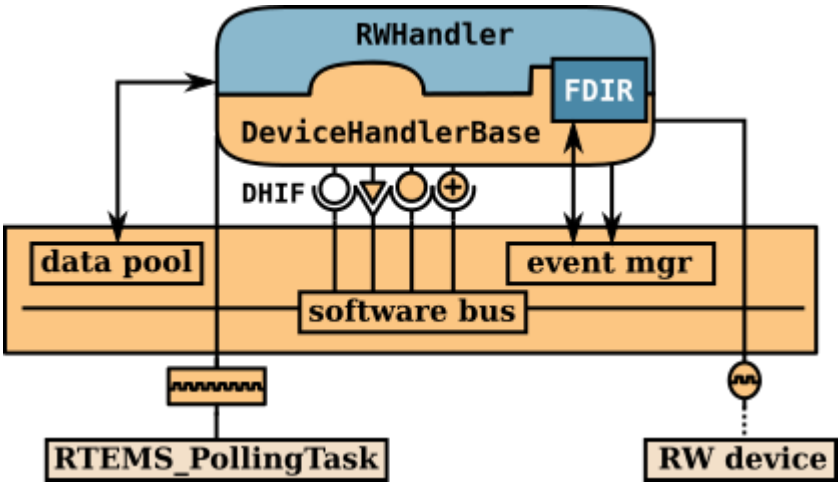

Figure 4.44.: Graphical representation of the RW device handler and its interaction with FSFW-Core elements.

As shown in Figure 4.44, each device handler represents its device in software with regards to the device mode, health and data exchange. It encapsulates specific behavior and protocols, so all devices share a uniform appearance to other components. Making use of `DeviceHandlerBase` supports commonality, as it already defines certain interfaces, e. g. mode and health, and implements a common usage.

With this approach, incidental complexity in other components can be avoided, which simplifies implementation of advanced functionality, such as fault and redundancy management of equipment.

Moreover, using `DeviceHandlerBase` allows rapid and straightforward implementation of a wide range of different device handlers. For *Flying Laptop*, for example, almost all device handler components introduced in Section 4.2.2 use `DeviceHandlerBase` as foundation.

4.5.2. Implementing Controllers - `ControllerBase`

Controllers are the central element of an embedded control system and likewise of FSW. They combine and monitor sensor values, apply control laws and

calculate output values for actuators. Thus, providing a template for controller implementation in the FSFW seems reasonable.

In practice, however, it is difficult to factor out commonalities between controllers to form a component template. This is due to the fact that every controller has a specific purpose which is very different form that of another controller. For instance, there are almost no functional commonalities between the thermal and attitude control component of *Flying Laptop*. For the same reason, no generic features of controllers were identified in the domain analysis (Section 3.7).

In effect, it is more useful to provide subsystem-specific *building blocks*, which can be used when needed, for controller components than defining a complex common base class such as `DeviceHandlerBase`.

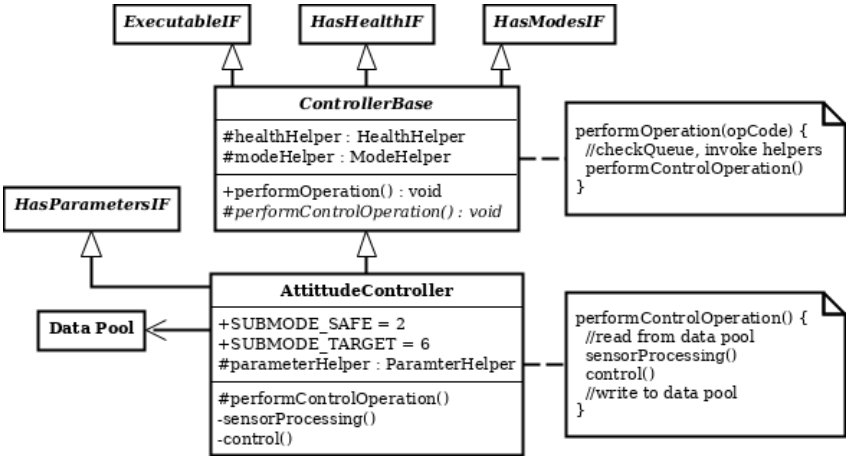

Figure 4.45.: Simplified class diagram of the `ControllerBase` template used by an attitude control component.

Nonetheless, a lightweight `ControllerBase` template is provided by the FSFW. Its main purpose is to identify a component as a controller and provide some common interfaces (see Figure 4.45).

As shown, the template provides the `HasModesIF` interface, as controllers may operate in different modes. Similar to `DeviceHandlerBase` the `ControllerBase` class pre-defines some modes and their semantics:

- `MODE_OFF`: Disables a controller, i. e. no calculation is performed and outputs are invalid. This is reasonable under certain circumstances, e. g. during ground tests.

- `MODE_ON`: All control algorithms are executed, but the output is set to invalid. This mode is intended for diagnostics mainly.
- `MODE_NORMAL`: Controller executes nominally. The submode determines the current control strategy.

For example, the attitude control component of *Flying Laptop* provides seven submodes in `MODE_ON` and `MODE_NORMAL`, e. g. safe mode or target pointing mode.

Controllers in the FSFW shall execute periodically and mainly independent from device handlers, with data exchange happening asynchronously through the data pool (see Section 4.4.5). Periodic execution is ensured by implementing the `ExecutableIF` and relying on the component execution infrastructure of the FSFW-Core (Section 4.4.2).

The component template also implements the `HasHealthIF`. As it is not very reasonable to talk about a "faulty" controller, the only allowed health state are `healthy` and `external_control`.

Building Blocks

The `ControllerBase` template, especially when compared to `DeviceHandlerBase`, is merely an empty hull for controller implementation, as it is not possible to find abstractions for domain-specific control problems.

To support such implementations, e. g. for thermal control, so-called *building blocks* are provided by the FSFW. They are an aid to assemble complex control components by providing out-of-the-box solutions for specific issues.

For instance, to support implementation of a thermal control component, the following building blocks are provided:

- `Heaters`: Represents a simple electric heater, which can be activated with a power switch.
- `TemperatureSensor`: This building block handles non-linear value conversion and range checks for analog temperature sensors.
- `ThermalModule`: Allows creation of different thermal control domains within a system.

Together, these building blocks cover typical needs of a thermal control software. Thus, implementing a thermal controller takes the form of assembling those building blocks according to system specifications, as illustrated in Figure 4.46.

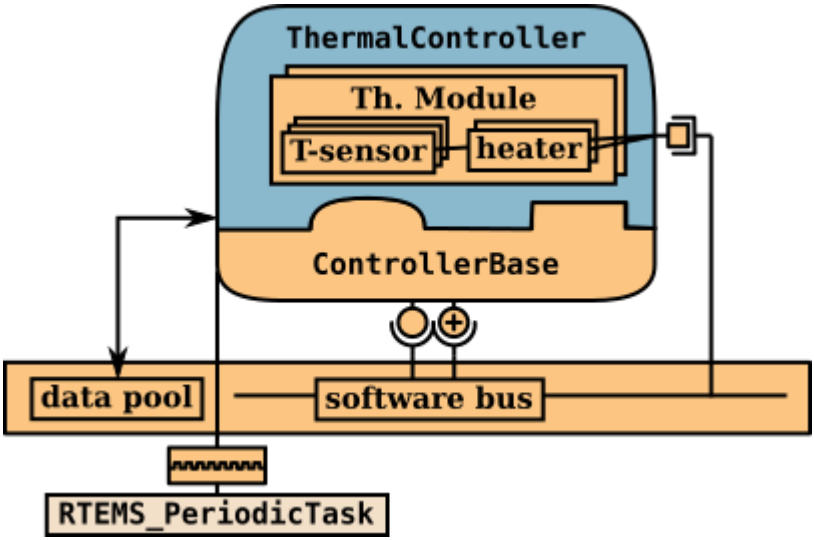

Figure 4.46.: Illustration of a thermal control component, created with `ControllerBase` and some building blocks for thermal control.

In a similar fashion, there are domain-specific building blocks for power control, such as a `PowerSensor`. It monitors current and voltage limits, calculates a power value and can be used for solar array or battery monitoring.

A more generic group of building blocks are *monitors*, which support checking of on-board variables. In most cases, monitoring is not a standalone activity, but embedded into dependent procedures. For example, an attitude control algorithm will check if the rate sensor inputs are within a certain range before progressing.

The monitoring building blocks of the FSFW support such embedded value checking in the form of a collection of monitoring classes (see Figure 4.47). These classes have the following properties:

- The basic `MonitorReporter` class provides the basis for reporting of out-of-range events, which can be enabled and disabled by adjusting parameters. Also, it defines a unique identifier for the monitor and the parameter it monitors.
- On top of that, `MonitorBase` implements checking a sample of the variable to monitor, as well as fetching that sample from the data pool, if needed. However, the type of check is an adaptation point for subclasses.

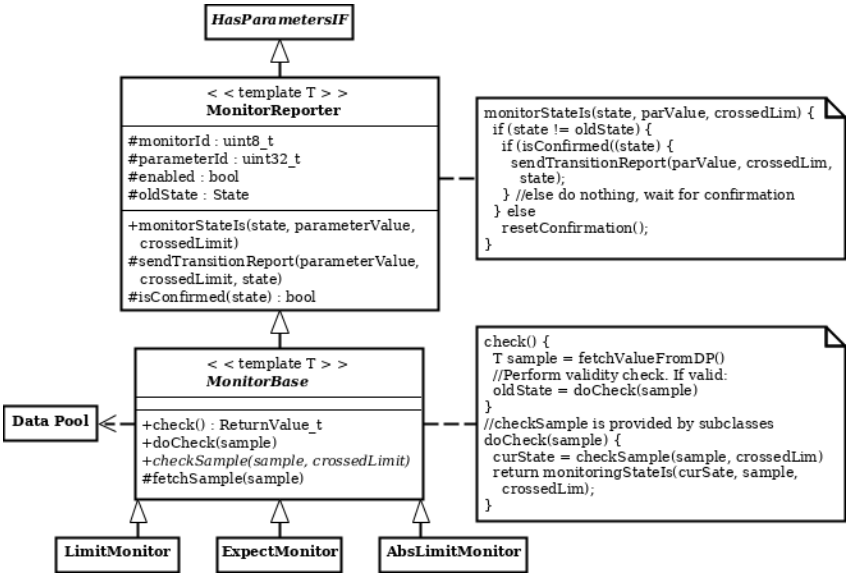

Figure 4.47.: Simplified class diagram of monitoring building blocks.

- The FSFW provides standard implementations for
  - `LimitMonitors` which check against a lower and upper limit.
  - `AbsLimitMonitors` which check the absolute value of a sample against a limit.
  - `ExpectMonitors` which check if a variable exactly has a certain value.
- The limit values for these monitors are adjustable via the `HasParameters-IF`.

Controller components can either use these classes directly, or create subclasses to modify the default behavior. The monitoring building blocks simplify implementation of monitoring needs within components and ensure that reporting of violations and adjustment of parameters is common for all monitors within the FSFW.

In combination, the `ControllerBase` template and the FSFW building blocks provide valuable support for the development of control components for a FSW. All but one control component described for *Flying Laptop* in Section 4.2.3 use the template as basis.

4.5.3. System, Subsystems and Assemblies - `SubsystemBase`

The goal of these component templates is to provide an easy implementation for subsystem and assembly components. These components aim to control the mode of other components and form the so-called mode tree of the system, as introduced in Section 4.2.4. The functionality was first described in [84].

Before explaining their implementation, an example shall detail their operation.

Example

For illustration, the ACS subsystem of *Flying Laptop* is depicted in Figure 4.48. On the image, the subsystem is in its safe mode configuration, using simple sensors for sun acquisition only[14].

For a transition to a fine-pointing ACS mode, it is necessary to enable all equipment of the subsystem. This could happen one-by-one with dedicated commands or ground procedures.

However, the mode tree allows doing so with a single mode command to the ACS subsystem component:

[14] The sun sensors are simple solar cells read by the PCDU and have no dedicated device handler component.

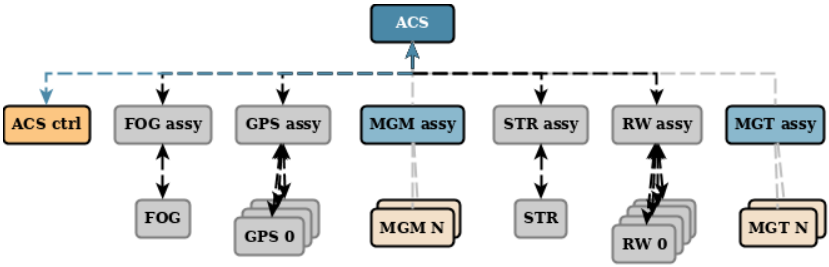

Figure 4.48.: The mode tree of the ACS subsystem of *Flying Laptop* in safe mode. On a transition to a pointing mode, first all assemblies are commanded (black lines), then the controller (blue line).

- The subsystem will check the mode command by looking up the intended mode, which is an identifier for a specific *mode sequence*.
- This mode sequence consists of a number of *mode tables*: One *target* and multiple *transition* tables. These specify which mode commands to issue to its children.
- In case of commanding the ACS subsystem to a pointing mode, the sequence consists of two transition tables, one to activate all equipment and one to change the controller mode. These tables are executed consecutively, so the controller will switch to pointing control only after all high-level devices have been activated.
- The first table directs the subsystem to send mode commands to assemblies. Each assembly forwards mode commands to the device handlers, taking into account health states and the redundancy scheme of the devices.
- Device handlers take care of the power up procedure of their device and perform an initial communication check.
- The device handler reports a successful mode change to the assembly, which, when all requested devices are available, reports a successful mode change to the subsystem.
- After confirmation of the mode change, the ACS subsystem will proceed with the current sequence, take the next table and command the controller to adjust its control strategy.
- In a final step, the subsystem uses the target table to check if all components are in their intended mode. If so, the transition was successful.

- The target table also defines the modes of all components to maintain the current mode. This makes it possible to, e. g. disable device with a transition table, but allow them to be enabled manually without jeopardizing the current mode.

The described mode change does not only work on subsystem level, but also in nested form for the entire system. This section describes the FSFW elements which allows the implementation of such a mode tree.

The `SubsystemBase` Template

`SubsystemBase` is the common template for all assembly and subsystem components (see Figure 4.49). It provides the basic functionality to build a mode tree, as it allows to register child nodes in a list and handle the execution of mode tables. To allow recursive composition for building the mode tree, it implements the `HasModesIF` itself[15]. The `HasHealthIF` is implemented as well, but used to indicate `EXTERNAL_CONTROL` only, similar to its use in `ControllerBase`.

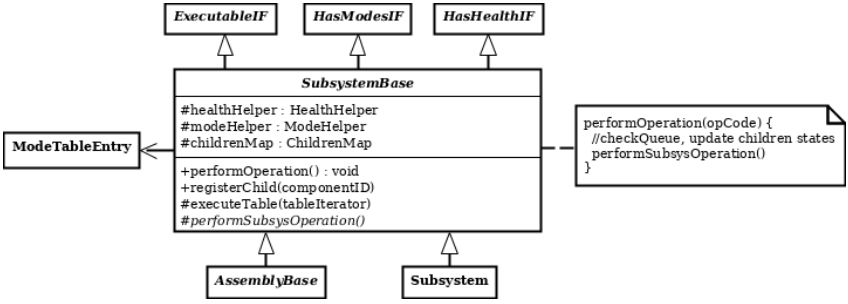

Figure 4.49.: Simplified class diagram of the `SubsystemBase` template.

A *mode table* is a list with component identifiers, as well as modes and submodes to command to (see Table 4.2). Execution of mode tables works as following:

- For each entry, it is checked if the component is registered.
- Then, the component's health state is checked. If it is `faulty`, it is commanded to `MODE_OFF`, no matter what the initial command was.
- If it is healthy and commandable, i. e. not in health state `EXTERNAL_CONTROL`, the intended mode command is created.
- If the component is already in the right mode, the command is discarded. Otherwise, it is sent via the software bus.

[15] This is the implementation of a classic design pattern of [60], the `composite` pattern.

Component ID	Mode	Submode
MGT_ASSEMBLY	MODE_NORMAL	0
MGM_ASSEMBLY	MODE_NORMAL	0
RW_ASSEMBLY	MODE_NORMAL	TORQUE
STR_ASSEMBLY	MODE_NORMAL	0
FOG_ASSEMBLY	MODE_NORMAL	0
GPS_ASSEMBLY	MODE_ON	0

Table 4.2.: Extract of a mode table of a transition of the ACS subsystem of *Flying Laptop*

A remarkable point is that the entire commanding relies only on the `HasHealthIF` and `HasModesIF` interfaces. Thus, it is an example on how abstraction in the form of interfaces simplifies high-level commanding.

`SubsystemBase` is an abstract class, aside from setting up a mode tree and executing mode tables, it does not provide an implementation on what to do with these features. The functionality of the base class is exploited in its subclasses, `AssemblyBase` and `Subsystem`. The former is a component template to implement assemblies of device components, the latter a complete component representing entire subsystems. These are described following.

The `AssemblyBase` Template

Almost all spacecraft fly with some sort of redundant equipment which is available in case of a device failure, as servicing a spacecraft is very difficult and expensive. It is easy to instantiate multiple device handler components for redundant devices. However, there needs to be an element that monitors the state of such equipment and handles reconfiguration if necessary. These efforts could be put into controllers, but this increases complexity of a component that has enough essential complexity anyway.

Instead, the FSFW provides the `AssemblyBase` template, which allows implementing assembly components to manage redundant equipment. As a subclass of `SubsystemBase`, it has a mode itself, which represents the mode of the device handler components registered as child nodes (see Figure 4.50). Registered device handlers are typically of identical type, e. g. four RW components.

The `AssemblyBase` template monitors mode and health state of its children and checks availability of devices on every detected change. `AssemblyBase` does not implement any redundancy logic by itself, but provides adaptation points for implementations to do so. Since most monitoring activities rely on mode and health state only and are therefore generic, it is sufficient for subclasses to

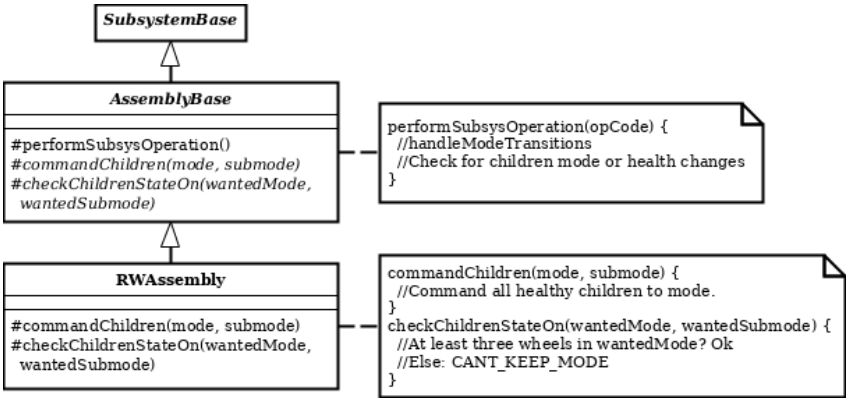

Figure 4.50.: Simplified class diagram of the `AssemblyBase` template.

provide the check logic when active (`checkChildrenStateOn`) and the transition logic to change the mode (`commandChildren`).

Thus, the RW assembly component will implement a three-out-of-four logic for the reaction wheels, which works as following:

- In case a mode change to "active", i. e. `MODE_NORMAL` is requested, the assembly commands all healthy RW device handlers to that mode.
- After the transition, the assembly checks if at least three of the four wheels are available.
- If so, it reports a successful mode change to the subsystem component.
- If this is not the case, it will disable all other wheels and report to the ACS subsystem that the mode can't be maintained.
- The same logic applies in case a health or mode change is detected by the assembly. If, for example, more than one RW handler's health state is `faulty`, the assembly will disable all wheels and report accordingly.

As described, assembly components manage redundancies by relying on mode and health states of their children. They do, however, not change the health state of children themselves, e. g. by analyzing events. Thus, assemblies are responsible for failure recovery, but neither for failure detection nor for failure identification.

Still, the approach simplifies the design of system mode management and FDIR, which are freed of knowing every detail of component redundancy handling. Admittedly, control algorithms need to be aware of available sensors and actuators anyway, but only as an input information to select an appropriate control

law. The use of assemblies and subsystems in FDIR design is described in Section 4.7.

The `Subsystem` component

With a mode tree of the system or a subsystem, such as that in Figure 4.48, it becomes apparent that the mode of a subsystem or the system as a whole is defined by all modes of its direct children. With this main idea, it is possible to define a generic machine capable of observing and changing the mode of arbitrary child nodes.

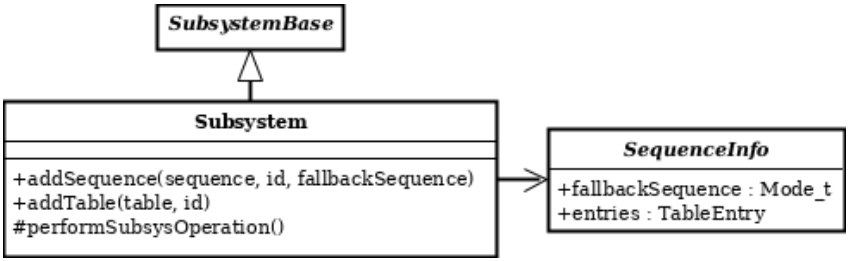

Figure 4.51.: Simplified class diagram of the `Subsystem` component.

This machine is encoded in the `Subsystem` component. It extends the facilities of `SubsystemBase` to manage a number of mode tables and *mode sequences*. As introduced in the above example, these sequences describe the steps necessary to reach a certain mode. The steps themselves consist of a reference to a list of transition tables, like that shown in Table 4.2, as well as a potential wait time before executing the next table. In addition, each sequence has an identifier, which is equivalent to one mode of the subsystem, and a reference to a fallback sequence. This is illustrated in Figure 4.52.

By executing the tables one-by-one, an ordered mode transition of the entire subsystem is achieved.

The *fallback sequence* becomes relevant in case a transition failed or the component can't keep its current mode, e. g. when a child node autonomously changed its mode. This is further described below.

The subsystem component is no template, as it is not necessary to implement any adaptation points for instantiation. Instead, it is sufficient to configure the pre-defined component by registering child nodes and defining mode tables and sequences. Still, it is possible to create subclasses of `Subsystem` to extends its functionality, e. g. to add autonomous mode changes on certain on-board events. For example, the *Flying Laptop* top-level `System` node is a `Subsystem`

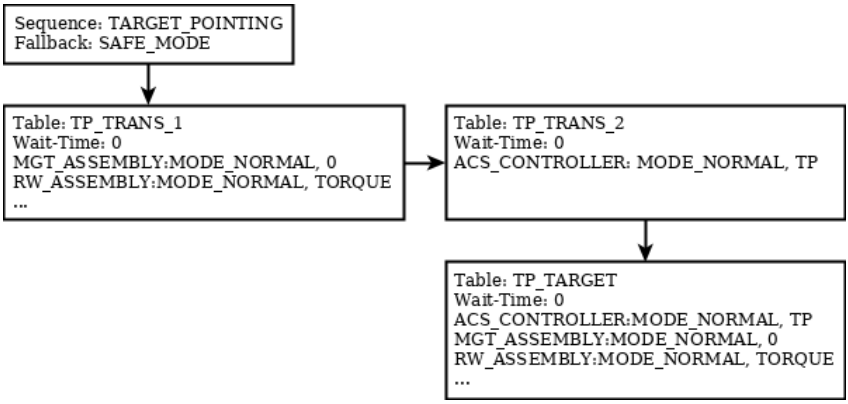

Figure 4.52.: Example of a mode sequence for a transition to an ACS target pointing mode

component extended to handle system-wide failures such as a low battery state of charge.

Fallback Transitions

Fallback transitions come into play as soon as something goes wrong. For example, the ACS subsystem would react with a fallback to ACS safe mode if the RW assembly were not available anymore.

In such cases, the `Subsystem` component looks up the fallback sequence and executes it like any other sequence. These are typically less demanding than the original sequence, and therefore there is an increased chance of successful execution. For illustration, Figure 4.53 shows an extract of the system-wide modes with fallback references from *Flying Laptop*.

This is an important feature of failure recovery, as it allows creating hierarchies of modes with increasing functionality, where fallback transitions to lower-functionality modes are possible. In case of equipment relevant for the current mode becomes faulty, and no more redundancies managed by the assembly are available, the subsystem will trigger a fallback mode transition, which will again trigger a fallback of the higher level subsystem or system node, until the system reaches a stable configuration[16]. Thus, reconfiguration happens in a bottom-up manner and stops automatically at the lowest level of containment.

[16] The final safe mode transition must be recursive and always successful, as is the case for safe mode in *Flying Laptop*

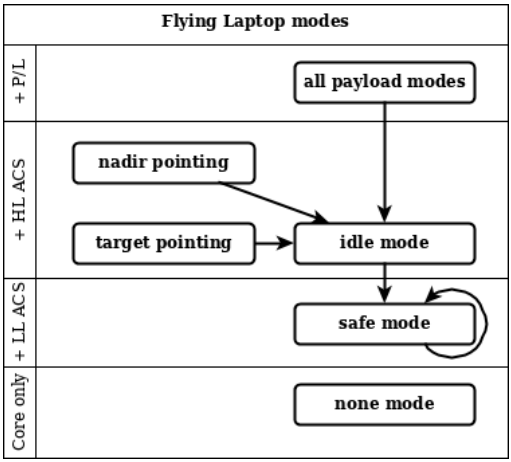

Figure 4.53.: Extract of *Flying Laptop* modes with arrows indicating fallback modes. None mode can only be commanded manually. From [55].

The FSFW FDIR concept relies on this feature for recovery actions, as further explained in Section 4.7.

In summary, `SubsystemBase` and its subclasses `AssemblyBase` and `Subsystem` are good examples where abstraction, encapsulation and generalization is use- and successful. This is mainly due to the fact that they rely on the rather abstract framework interfaces `HasModesIF` and `HasHealthIF`, which enforce uniformity. Thus, the benefits which `SubsystemBase` components bring to operators and FDIR designers of a spacecraft are a result of the abstraction introduced by the FSFW.

4.5.4. Summary

The component templates introduced in this section unify and simply creation of components for a FSW. Those templates allow internal code reuse, as their code is shared among many different components. For example, almost all components of the *Flying Laptop* software are implemented using a component template.

Inheritance is a good way to group components into categories, as the templates form a dedicated type hierarchy. This categorization simplifies describing and arguing about implemented components, as it is possible and correct to talk of a property of e. g. ‘all device handlers of the system”. The approach chosen is that of a white-box framework for components (see Section 2.3.2).

Component templates, especially `SubsystemBase` and its subclasses are good examples of the utilization of abstraction, encapsulation and generalization. Relying on the common `HasModesIF` and `HasHealthIF` interfaces, which enforce uniformity, reduces effort for spacecraft operators and component designers.

4.6. The FSFW PUS Framework

A standardized communication protocol such as PUS allows to provide generic implementations for certain protocol features avoiding implementing the same protocol over and over again in every mission.

Thus, the FSFW provides a PUS framework, which in principle has the same role as other such frameworks, e. g. OBOSS (Section 3.6), but is compatible to the overall design of the FSFW.

The PUS framework provides building blocks and component templates to implement a space link stack like that of *Flying Laptop* described in Section 4.2.5. Thus, it provides supporting elements for the following OSI layers:

- Handling of TC frame acceptance of the data link layer in software.
- Distributing space packet TCs and returning TM packets for downlink, i. e. routing packets in the on-board networking layer
- Implementing service components for the application layer.

4.6.1. Data Link Layer - Channeling Frames

The CCSDS TM and TC data link layer is responsible for correct transmission of multiple virtual channels (VCs) over a single space link (see Section 3.4.1).

The FSFW provides an implementation for TC transfer frame reception, including distribution to different VCs and the entire Communications Operation Procedure-1 (COP-1) reception engine. It is provided in the form of a building block and allows free configuration of number and IDs of virtual channels. After extraction, the frame content, typically a space packet, is forwarded to a configurable entity on a per-VC basis[17].

The framework does not support dedicated handling of TM frames. Due to the simple protocol, these frames are often handled in hardware encoder boards, reducing the need to deliver a software solution. However, an implementation can be added if necessary.

[17] More precisely, packets are distributed depending on the Multiplexer Access Point (MAP) channel, of which multiple are transferred within one VC

Example

For *Flying Laptop*, the COP-1 frame acceptance building block is instantiated in the CCSDS device handler component, of which two instances manage each of the hot redundant CCSDS-Boards (see Figure 4.54). Each board is configured to accept frames of a dedicated virtual channel, so command packets are not duplicated and the ground station can decide which board to use for decoding.

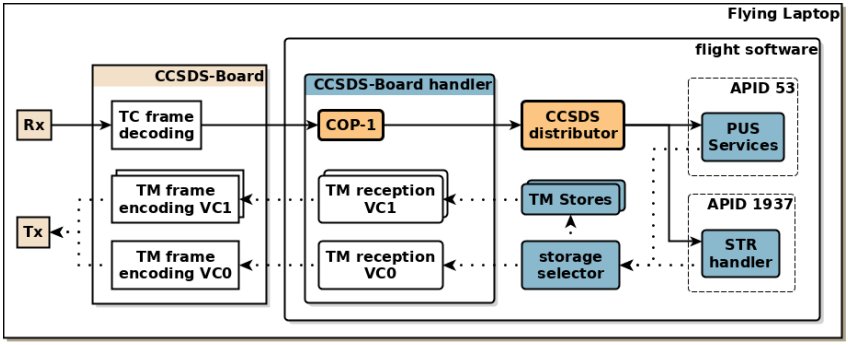

Figure 4.54.: TC frame reception and packet distribution, as well as TM packet forwarding in the *Flying Laptop* setup. Orange elements with thick borders are provided by the PUS framework.

As TM frame generation happens in the CCSDS-Board hardware, the CCSDS device handler simply serves as the receiving end for TM packets within the software. It provides one access point per VC and is responsible for the correct forwarding of these packets to the CCSDS-Board. In *Flying Laptop* the TM VCs are used for live TM on VC 0 and different types of stored telemetry on the other VCs (Figure 4.54).

4.6.2. Networking Layer - Distributing Space Packets

The FSFW supports space packet routing on-board the spacecraft with a dedicated software component called `CCSDSDistributor` (Figure 4.55). It is the first entity to receive incoming telecommands packets after decoding and extraction from TC frames.

The distributor is designed to forward space packet to individual software components. To actually receive packets, components need to provide a dedicated interface called `AcceptsTelecommandsIF` and register at the `CCSDSDistributor`, which consequently forwards all space packets containing the respective application process identifier (APID). Passing the packets between components happens

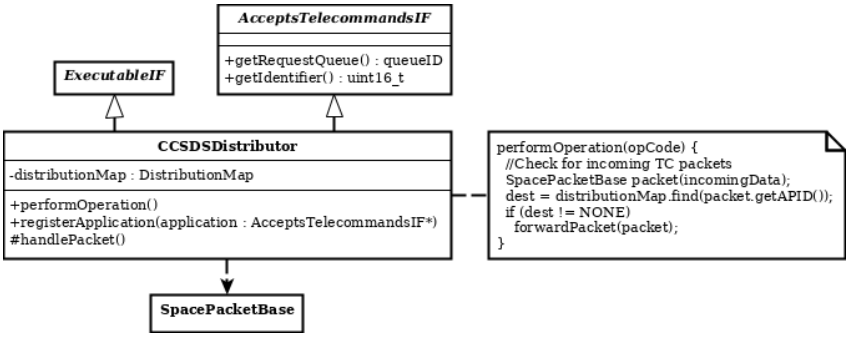

Figure 4.55.: Simplified class diagram of the `CCSDSDistributor` component.

with the same mechanisms as those of the FSFW software bus, i. e. asynchronously via messages, but on dedicated communication lines.

Routing of TM packets is even more simple, as they are always forwarded to a single downlinking entity. However, to allow on-board storage, it is often convenient to route all TM packets to a storage component first, which stores selected packets and routes all packets to the live downlink. The routing scheme for *Flying Laptop* is shown in Figure 4.54.

This routing component, together with space packet representation and distribution methods, form the on-board networking layer provided by the FSFW. The modular setup makes adding APIDs or introducing new TM routes a mere configuration issue.

Space Packets

To simplify handling of space packets, it is convenient to wrap the content in an object providing getters and setters for various packet fields. The `SpacePacketBase` class of the FSFW provides such an interface. To avoid costly copying of data, it operates on a data stream, directly accessing the underlying data. It is the basis for subclasses used for the packet utilization standard (PUS) application protocol, as seen in Figure 4.56. As these subclasses indeed are space packets of a specific type, inheritance is a natural way to define the dependency. Using the base class simplifies access to and creation of space packets.

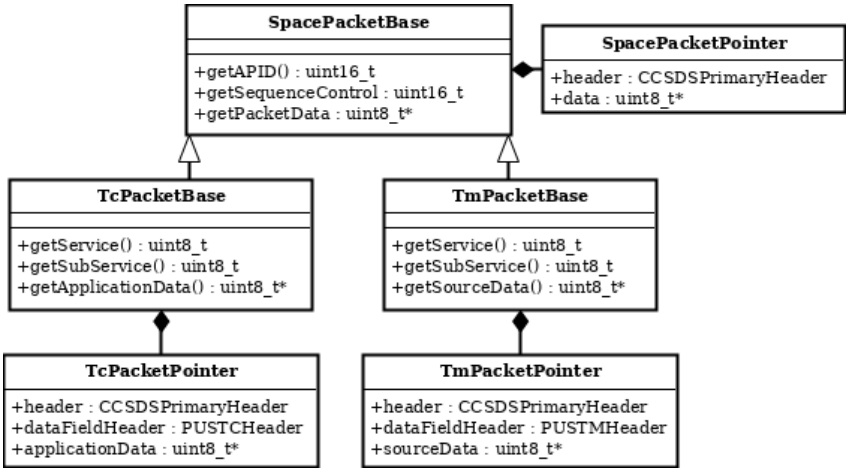

Figure 4.56.: Simplified class diagram of `SpacePacketBase` and its subclasses. or clarity, only getters of packet fields are shown.

4.6.3. Application Layer - Providing PUS Services

The FSFW PUS framework provides component templates to implement standalone and gateway service components. These templates are `PUSServiceBase` and `CommandingServiceBase`, which are described following. There are, however no complete components provided, neither for standard service types, nor for FSFW-specific gateways components. In some cases, such as the housekeeping service, this would in principle be possible, but the mission-specific tailoring, e. g. which optional field of a specific request to use, is difficult to bring in a generic form.

`PUSServiceBase`

The basic activities of a PUS service are as following:

- Receiving a request in form of a TC packet.
- Decoding the content and performing some immediate action, if necessary.
- Generating verification reports as requested by the standard.
- In some cases, returning a data report in form of a TM packet in response.
- Producing TM report packets periodically, such as housekeeping packets.

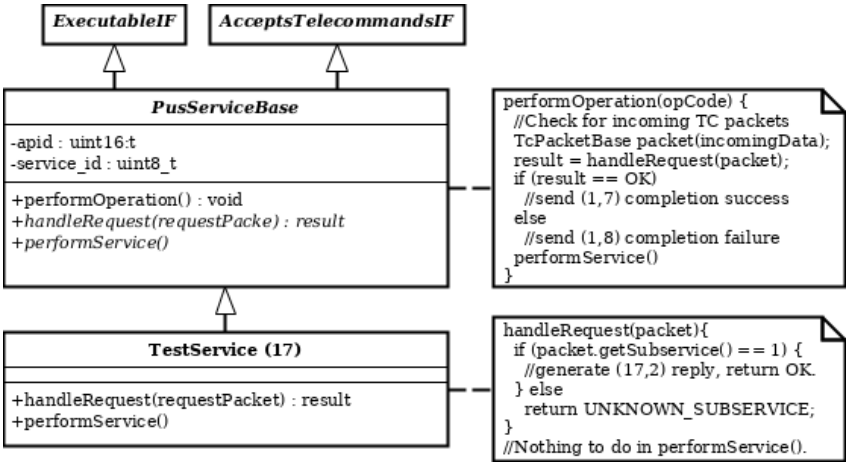

Figure 4.57.: Simplified class diagram of the `PUSServiceBase` component template.

To implement such a service, the FSFW provides the `PUSServiceBase` component template (see Figure 4.57). It reduces service implementation to a minimum by performing all generic steps of TC handling, including registration for packet routing, as well as generation of verification reports.

Specific service implementations can extend the basic functionality with two adaptation points, which are `handleRequest` to actually handle an incoming request and `performService` to support periodic activities of the service.

Even though the model is rather simple, it is a viable basis for a lot of service implementations. For example, all standalone services of *Flying Laptop* shown in Section 4.2.6 use this template as basis. Figure 4.57 shows the implementation of a simple test service component.

`CommandingServiceBase`

`CommandingServiceBase` is a component template to implement service gateways as described in Section 4.1.6. Thus, it is responsible for forwarding PUS requests to other software components as internal software bus messages and vice versa.

Regarding interface inheritance and handling of PUS requests, it is similar to `PUSServiceBase`, but has additional capabilities to communicate with other components using messages over the software bus. Thus, the adaptation points for component implementations are different:

- `isValidSubservice`: Check if a TC request is acceptable.
- `getMessageQueueAndComponent`: Check existence and obtain address of the intended target component.
- `prepareCommand`: Prepare a message for the software bus based on the incoming TC request content.
- `handleReply`: Handle incoming replies from components, eventually create data reports and complete the transaction.

This template is the basis for all gateway components of *Flying Laptop* (Section 4.2.6).

To illustrate its functionality, the interaction between the template and the implementation of a function service, which forwards action commands to components is shown in Figure 4.58.

In the example, the ground segment dispatches a TC to the attitude control component to adjust the pointing target. This message is received by the function service component, which uses the `CommandingServiceBase` template to handle the bulk of the PUS protocol.

4.6.4. Summary

The FSFW PUS framework can be used for FSFW-based flight software implementations which use the common CCSDS / ECSS PUS protocol stack. It eases frame reception, as well as on-board packet handling and distribution.

Also, it provides templates to implement PUS service components, either in the form of standalone or gateway services. These component templates simplify implementation of specific services. Alternatively, reuse of existing service components is possible, even though they are not provided by the framework itself.

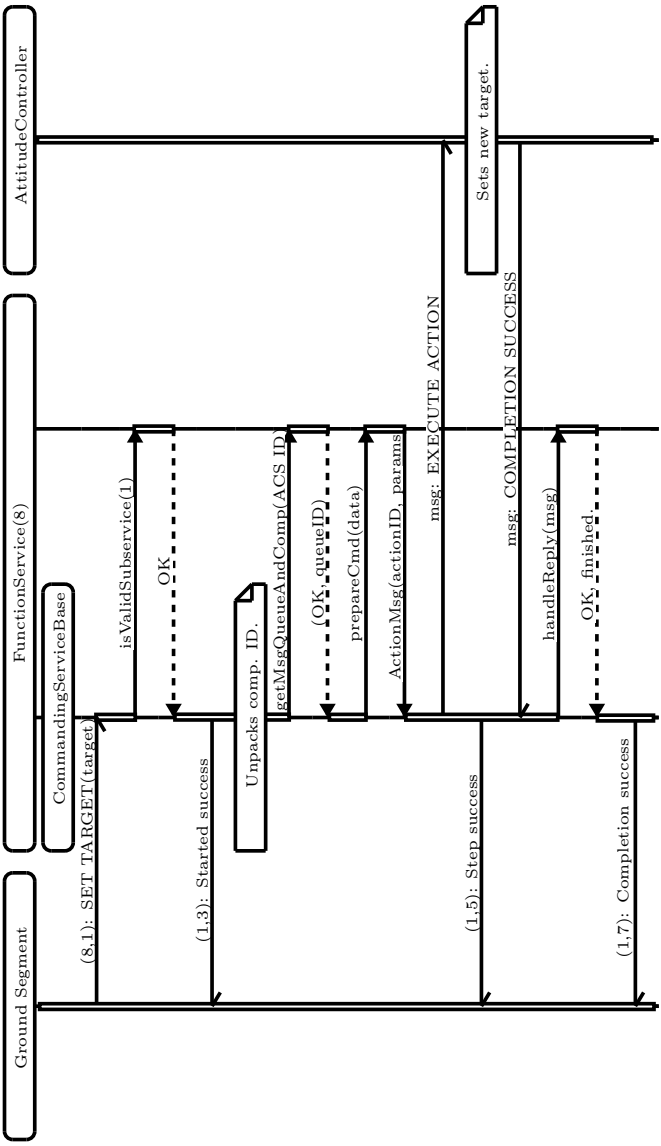

Figure 4.58.: Sequence diagram to show the interaction between `CommandingServiceBase` and a subclass, here the function service, to forward an action invocation to the ACS control component.

4.7. Fault Management

A distinctive feature of spacecraft in comparison to other machines is the uniqueness and remoteness of their operational environment. Typically, they are built in precisely controlled clean room conditions under direct control of engineers, only to be released into the harsh environment of open space, where direct access is virtually impossible. This has a strong influence on system design and typically dictates a spacecraft to carry redundant sensor and actuator equipment to compensate for arbitrary failures.

These redundancies are quite useless, however, if they are not activated and used in potentially hazardous situations. As remote control is often intermittent and slow, this results in autonomy requirements for fault management software, as described in Section 3.5.3. This failure detection, isolation and recovery (FDIR) software must ensure spacecraft availability, or at least survival, in case of critical equipment failures.

The Flight Software Framework supports implementing FDIR software: As every hardware equipment is represented by a dedicated device handler component, or some building block within a control component, the goal of the FDIR design of the FSFW is to identify the faulty component and adjust the component's or building block's health state. The component itself, or an associated assembly or subsystem component then reacts on this change in health state. The system's main design goal is to handle faults as local as possible, ideally within the faulty component itself.

This section describes the interaction of FSFW interfaces, core features and component types to ensure a seamless integration of failure detection, isolation and recovery in a FSFW implementation.

4.7.1. Example

To illustrate the functionality, parts of the attitude control subsystem (ACS) of *Flying Laptop* serve as example, namely the MGM sensors and the RW actuators. These FDIR concepts have been devised and described in detail in [109].

All higher modes of the ACS subsystem require the RW assembly to be available in `MODE_NORMAL`, but they are not needed for safe mode. The MGM assembly must be available at all times[18]. As described in Section 3.3.2, all equipment is connected to the IO-Board, which is controlled by the OBC's processor board.

The three-out-of four redundancy allows the RW assembly to remain operational even if an arbitrary wheel fails, but it always tries to keep all healthy

[18] In higher ACS modes, the magnetic field information is needed for RW desaturation.

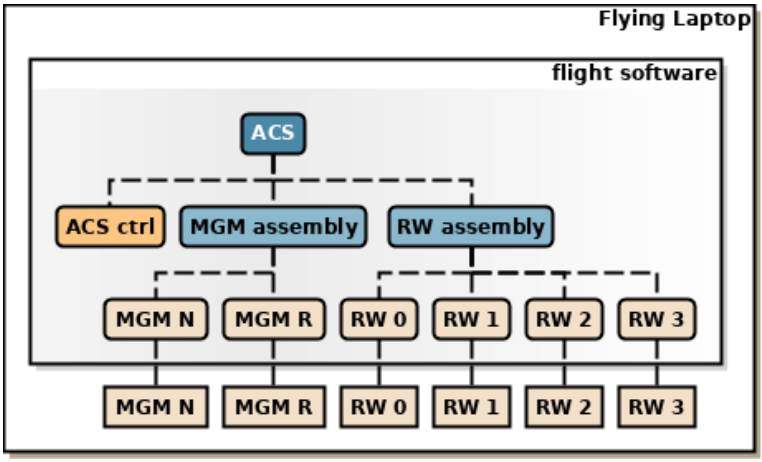

Figure 4.59.: Extract of the *Flying Laptop* ACS subsystem mode tree with MGM and RW assemblies.

wheels active. It uses the `AssemblyBase` template as basis. Each RW device is represented by its own RW device handler component, which have the RW assembly as mode tree parent. As described in Section 4.5.1, the RW device handler is based on the `DeviceHandlerBase` template.

To improve measurement accuracy, the MGM assembly keeps both MGMs active by default, but remains available with one faulty MGM as well. The MGM devices are polled for measurements with a simple command, which is manged by a device handler component for each MGM. Both the assembly and the device handler component are based on their respective component template.

4.7.2. Fault Detection

A prerequisite for autonomous FDIR software is that faults are properly detected. In the FSFW, failure detection takes place either in the device handler component itself, or in some assembly or controller component monitoring sensor variables [19]. In [109], Section 5.5, these are the stages two and three of "the three stages of device failure detection". Any detected fault is reported using the FSFW-Core event distribution mechanism.

If the device handler component template is used (see Section 4.5.1), monitoring of equipment communication is built-in. If the device itself is capable of error

[19] Monitoring building blocks as described in Section 4.5.2 can be used for this purpose.

reporting, the device handler component additionally translates these error reports into on-board event messages. For example, the FOGs used in *Flying Laptop* are capable of reporting a number of hardware errors.

Other equipment failures can be detected based on sensor values. These values may be coming from the equipment itself, or are associated values, such as power or temperature of the equipment. They are best monitored not in the device component, but in the sensor or actuator monitoring parts of controller components. This is more simple, as controllers process the values anyway, but also more reasonable, as cross-checks and comparisons of hot redundant sensor values are possible.

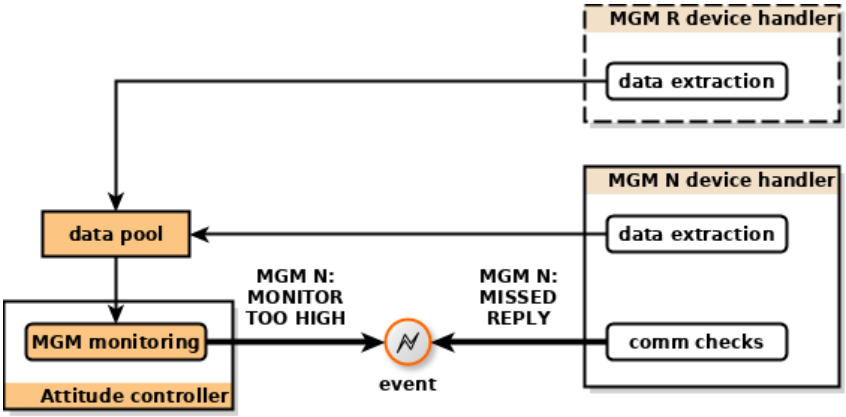

Figure 4.60.: Illustration of failure detection for a MGM device.

For example, measurements of the MGM device are checked in the attitude control component against expected minimum and maximum field values, which depend on the orbit of the mission. Also, a comparison between values from MGM 1 and 2 takes place if both are available. An important point is that the reporter ID of a failure event is always that of the potentially faulty component: If, for instance, the attitude controller detects an out-of-range magnetic field value of MGM 1, the reporter ID of the issued event is set to `MGM_1`, and not `ACS_CONTROLLER`. Failure detection for such a sensor is illustrated in Figure 4.60.

4.7.3. Failure Isolation

Failure isolation is twofold: The first part is *identification*, which is about finding the root cause of a failure based on detected and reported faults. In

technical systems, one needs to take into account the possibility of common cause failures, where a failure in multiple components is caused by a single fault, e. g. if a faulty power supply equipment drives multiple components.

Second, the actual failure *isolation* are the measures to be taken to avoid infection of other parts of the system. An example is to ignore sensor values or shut down faulty equipment.

Failure Identification

To illustrate the issue of failure identification, one of the failure cases of *Flying Laptop* is used: As described in Section 11.1 of [109], the attitude controller performs different checks on the MGM measurements. While an out-of-range check of a single device is directly attributable, this is impossible for a comparison of MGM N and R. So for failure identification, the first case would result in some isolation activity, whereas the second won't, as the system can not identify the root cause of the fault by itself. In such cases, ground segment intervention is necessary.

It gets even more tricky in case of missed replies of the MGM devices (or any other device). As all equipment communication is routed over the IO-Board, a failure of the board would result in `MISSED_REPLY` events of all currently active device handlers.

The FSFW supports failure identification with a so-called `FailureIsolationBase` building block (see Figure 4.61). It is an abstract base class, which is intended for use as part of other components in general and device handlers components in particular.

Its main intention is to collect all events assigned to its owner, to evaluate them and ultimately, decide on corrective actions for failure isolation. To do so, it subscribes for all events with its owners reporter ID at the FSFW-Core event manager and prepares incoming events for evaluation. The evaluation itself depends on the type and properties of the owning component, therefore, users of this abstract building block must implement the `eventReceived` method as main adaptation point.

To handle issues such as the faulty IO-Board of the above example, `FailureIsolationBase` provides a failure confirmation mechanism: It allows components to request confirmation from failure isolation building blocks of other components, as well as handling and responding to such requests. Communication takes place with modified event messages, which are exchanged between the building blocks using a dedicated `ConfirmsFailuresIF` interface.

This mechanism helps resolving failure identification in the above example:

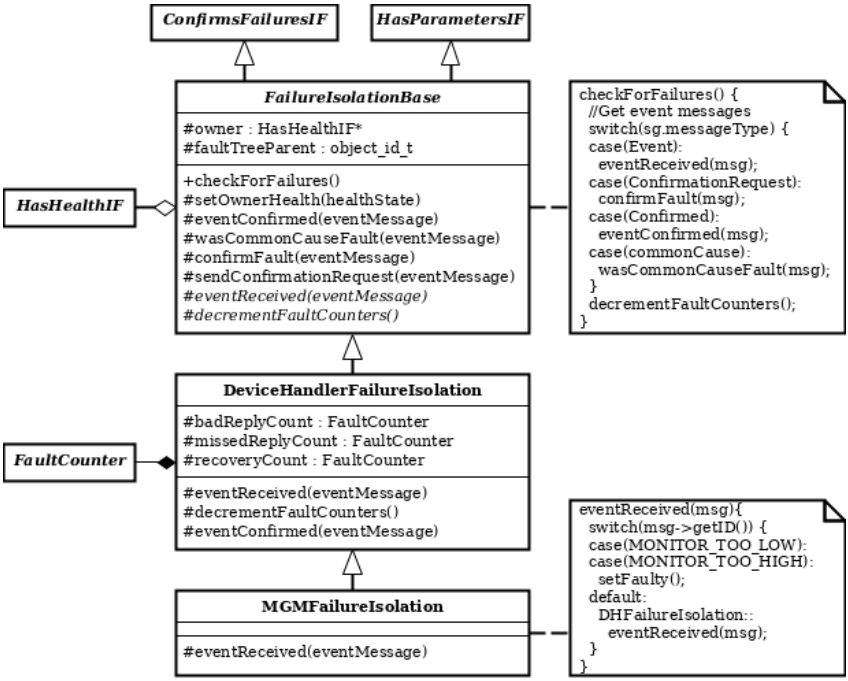

Figure 4.61.: Simplified class diagram of `FailureIsolationBase` class, the `DeviceHandlerFailureIsolation` subclass, and a specialization for failure isolation of MGM devices.

- In case of a `MISSED_REPLY`, the failure isolation building block of a MGM device handler first issues a confirmation request to the failure isolation block of the IO-Board device handler.
- Depending on its own state and other incoming confirmation requests, the IO-Board device handler's failure isolation has two options:
  - Confirm the fault, i. e. returning that it is caused by the MGM device itself.
  - Returning that a common cause fault was identified. This means a broken IO-Board was detected and most likely the MGM device is not broken.
- Depending on the result of the confirmation request, the MGM failure isolation either initiates isolation activities or ignores the fault.

This strategy avoids wrong association of faults due to common cause failures.

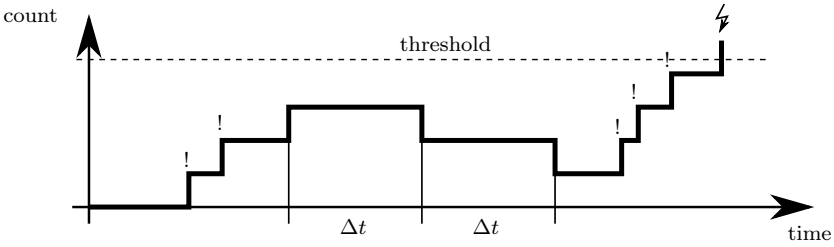

Figure 4.62.: Diagram showing the use of the `FaultCounter` building block. From [109].

Failure events can either indicate a change of state, e. g. a fuse triggered, or not, as in case of a lost device reply. In the latter case, the failure isolation blocks needs to make sure isolation does not react overly sensible on glitches [109]. The FSFW provides so-called `FaultCounter` building blocks for that purpose. They have two adjustable parameters, a threshold value and a decrement interval Δt. As depicted in Figure 4.62, an adjustable amount of incidents can be ignored in a certain time range. `FailureIsolationBase` does not use any fault counters directly, but is prepared for using them by providing a `HasParametersIF` for parameter adjustment from ground and the `decrementFaultCounters` adaptation point.

Failure Isolation

If conditions are met to unambiguously assign a failure to a certain component, i. e. the failure cause is identified, failure isolation takes place.

In the FSFW this happens by adjusting the heath state of a component (or building block). Useful options are either `faulty`, to disable the equipment and/or ignore sensor values, or `needs recovery` to initiate a power cycle of a device.

No other action for failure isolation is necessary, as associated assembly and subsystem components will autonomously react on the changed health state and perform the programmed recovery actions, as detailed in section Section 4.7.4 below.

`DeviceHandlerFailureIsolation`

Even though failure causes and reactions are typically unique for every device, there are some common failure modes for different equipment. For example, both MGMs and RWs should be disabled if the temperature is out of the operational range.

This is even more true if the device handler component template is used as basis. Due to the generic communication monitoring, it generates a common set of events for every device. This facilitates the implementation of a default equipment failure isolation building block, which is provided with `DeviceHandlerFailureIsolation`. As shown in Figure 4.61, it inherits from `FailureIsolationBase` and implements the `eventReceived` method.

Within this method, it handles all common events regarding power, thermal and communication of a device. To avoid being overly sensible, it uses fault counters to count missed and malformed replies before adjusting the health state of its owning component.

With the exception of a few severe events, e. g. too high power consumption, it first tries to reboot the device using the `needs recovery` state, before marking the device `faulty`[20]. Also, it uses the failure confirmation mechanism to check if communication errors stem from a failure in a dedicated interface component, such as the IO-Board of *Flying Laptop*. For illustration, the entire source code of the `eventReceived` method is shown in Section A.2.

This building block further simplifies FDIR implementations, as all common forms of failure handling are already programmed. Variations and extensions are possible by creating subclasses of `DeviceHandlerFailureIsolation`. For example, the MGM device handler extends the default failure isolation block by adding an isolation action, which triggers if an value out-of-range event is detected.

[20] Another `FaultCounter` is used to avoid indefinite reboot cycles.

4.7.4. Failure Recovery

The missing piece for a complete FDIR implementation are means for failure recovery. Most of these are, in fact, already described in Section 4.5.3. Assembly components react on health states of device handlers, i. e. they command `MODE_OFF` in case a component is `faulty` and perform a power cycle if it is set to `needs recovery`[21]. In addition, they implement redundancy type-specific logic to check if the current mode can be kept, eventually by activating redundant equipment.

This is illustrated with the RW assembly of the above example:

- In case of a health change of one of the RWs to `needs recovery`, the RW assembly will power cycle the device without changing its own mode.
- If successful, the failure is handled and no further actions are required.
- If the problem persists, the same failure events will cause the RW component to change its health state to `faulty`.
- The assembly shuts down the wheel and checks if at least three wheels are available.
- If so, the assembly retains its mode and the failure is handled.
- If not, the assembly shuts down all wheels, changes its mode to `MODE_OFF` and reports this step, e. g. by issuing a `CANT_KEEP_MODE` event.
- These reports trigger the parent subsystem, i. e. the ACS subsystem component to check its mode: If the RW assembly is needed in the current mode (which is likely), it will trigger an autonomous system fallback as described in Section 4.5.3.

In effect, multi-staged device failure recovery is already implemented in the system mode tree.

As the whole is often greater than the sum of its parts, there may be some failure cases which are not attributable to a single device, but need direct treatment on system level, e. g. a low battery state or high rotation rates. For such cases, failure isolation building blocks within other components, such as controllers or subsystem components can handle dedicated events and perform reactive measures directly.

In *Flying Laptop*, for example, the top level `System` component reacts on low battery state events by disabling all non-essential loads, which happens by commanding the `Payload` subsystem component to `MODE_OFF`.

[21] They do so by commanding `MODE_OFF` and `MODE_ON`, consecutively.

4.7.5. Summary

This chapter showed the concepts of fault management implementation with the FSFW. It shows that a hierarchical FDIR, which is desirable for reasons of decomposition and separation of concern [61], harmonizes well with the component-based approach. It allows not only to create a unit and system level FDIR, but also multiple intermediate subsystem levels for failure handling.

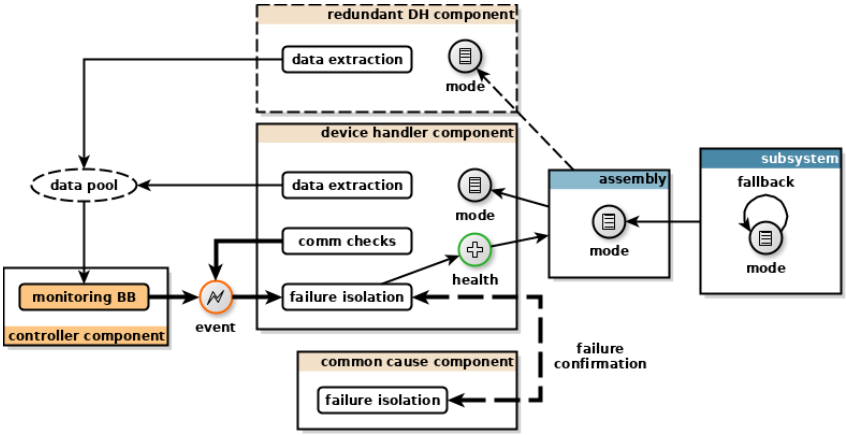

Figure 4.63.: Illustration of the interaction of FDIR elements of the FSFW. Thick lines indicate events or confirmation messages, thin lines indicate software bus messages or data pool access.

In addition, the strict separation of failure detection, isolation and recovery enables a clean FDIR design, which mainly utilizes standard, built-in functionality of components, such as modes and health, as well as recovery features of assemblies and subsystems. The only additional implementation effort is required for failure isolation, which is alleviated by the building blocks and default implementations the FSFW provides. Utilizing those, a complete FDIR implementation comes together like the pieces of a puzzle, as shown in Figure 4.63.

Moreover, the generic form of this implementation simplifies interaction with ground operators, as reactions, even of complex failure scenarios, follow a common concept. Also, operators can interact with the FDIR implementation, e. g. by setting device component health states manually in cases where on-board identification fails. Changing the health state by ground command triggers the same on-board reactions as an autonomous failure detection, which also supports ground testing of fault management software.

The FDIR implementation of the FSFW supports all fault management-related FSW features identified in the domain analysis in Section 3.7.

5. Evaluation

This chapter evaluates to what extent the original hypothesis has been achieved:

> Utilizing modern software engineering techniques improves development of complex embedded software in general, and spacecraft flight software in particular.

First of all, it is indeed possible to utilize these techniques to develop, implement, test, and operate a spacecraft FSW: The small satellite *Flying Laptop*, whose FSW is based on the FSFW developed in this thesis, was launched on July 14$^{\text{th}}$, 2017 and is performing extraordinary well since then. To date, there were no unexpected issues related to the general architecture of the FSFW. The first test image was received five days after launch (see Figure 5.1).

This on-orbit proof of concept of the FSFW shows that a component framework architecture with SOA-like communication for embedded systems is not just a vision in the ivory tower of a University, but works in a real-world deployment.

Thus, the following evaluation focuses on describing scenarios in which the FSFW architecture proved particularly useful, not only during development, but also during system tests and operations of the spacecraft.

5.1. Developing a FSFW-Based Software

The component-based approach followed in the FSFW ensures *separation of concerns*, as every functionality is clearly assigned to a dedicated component. Thus, developers can focus on implementing individual components, without the need to fear interference from their colleague's work. For example, there were typically little issues regarding software integration during the development of the *Flying Laptop* software and bugs in new components were well encapsulated.

Due to interaction between components by well-defined mechanisms, coupling is loose, and it is simple to replace individual components with stubs, e. g. for unit testing. Moreover, by hedging the inflationary definition of service interfaces and instead defining a limited common set (Section 4.3), the often-encountered incidental complexity of a general-purpose SOA implementation was avoided.

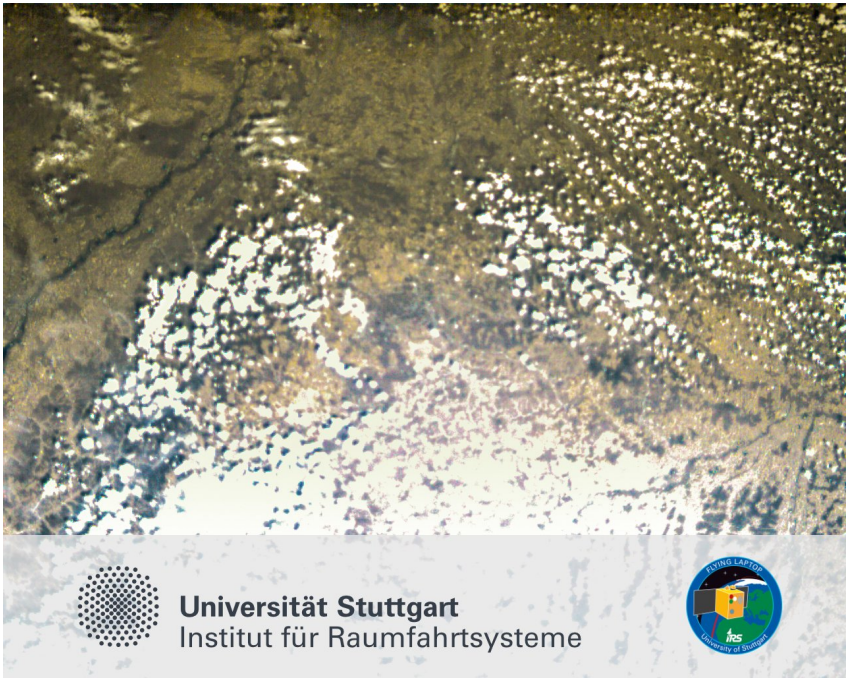

Figure 5.1.: First image acquired from *Flying Laptop*, showing southern Germany. Taken on July 19$^{\text{th}}$, 2017.

Thus, one finding of this thesis is that *domain-specific*, but *component-generic* interfaces support developers and improve reuse of software components.

Also, by providing component templates and common building blocks, the FSFW design fosters *internal reuse* of code, as they are applicable for many different components of a project. For example, 54 of the 57 components from *Flying Laptop* - introduced in Section 4.2 - use one of the component templates of the FSFW as basis. For device handler components this means that most code to handle equipment is shared by using the component template, reducing implementation effort to a minimum.

This is not only convenient for developers, but improves overall code quality: Shared code reduces the need for duplicated functionality and allows new components to be based on well tested foundations.

These templates provide enough flexibility to cover uncommon extensions: The TTC subsystem from *Flying Laptop*, for example, is based on the `Subsystem` class, but was extended to serve as a controller to manage automatic mode transitions of the subsystem, e. g. to activate transmitters on an incoming ground segment signal.

Still, the most important aspect of developing with the FSFW is that it provides a software architecture. This architecture guides developers on how to create the overall FSW, and by providing the fundamental FSW-specific infrastructure for execution and communication, ensures that programmers can focus on the essential complexity of mission-specific applications.

5.2. Spacecraft Testing with the FSFW

The component-based approach also provides advantages for spacecraft testing:

First, it is possible to perform system tests early with rather incomplete versions of a flight software, as each component can operate independently. For example the *Flying Laptop* flatsat campaign was performed with a software that provided only the most relevant PUS services and device handler components, some of which were being completed just in time for the tests. However, as every component is an encapsulated unit, the tests were representative enough to validate the electrical interfaces and device handler components themselves.

Moreover, the flexibility to exchange software components helps adjusting the system to different testbeds. For example, the *Flying Laptop* software was ported to an evaluation board without SpaceWire lines to overcome a testing bottleneck in the final development campaign. This was managed by replacing the CCSDS-Board and IO-Board components with simple bridging components,

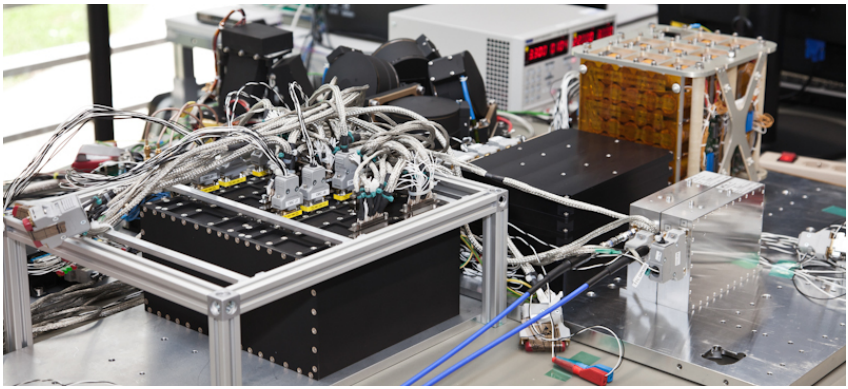

Figure 5.2.: Flatsat setup of *Flying Laptop*.

which routed space-ground, as well as equipment communication over available Ethernet links.

As a last point, the flexible mode concept of the FSFW as described in Section 4.5.3 eases system tests, as it is possible to quickly bring the entire system into a specific system mode with a single command. Moreover, the FSFW allows to configure each component manually by commanding individual controller and device handler modes.

5.3. Operating a Spacecraft with the FSFW

A FSFW-based software helps operators to get a consistent view of the system as all interactions happen with components conforming to common interfaces [84]. This also ensures that every telemetry value and every event is bound to a certain component and thus is automatically ordered.

Moreover, using component templates as basis ensures that the "look-and-feel" of every component is similar, avoiding unpleasant surprises for operators.

For example, instead of memorizing particular start-up procedures of individual equipment of the spacecraft, operators can rely on the mode abstraction introduced by the `DeviceHandlerBase` component template (Section 4.5.1). Also, each device handler component issues a common sequence of events, e. g. during a mode change, which makes it easier to understand the system's behavior as a ground operator.

For *Flying Laptop*, the "simple" operability due to the FSFW allowed operating the spacecraft by undergraduate students as staff shortly after commissioning. The basics for satellite operations were taught in a one-semester lecture.

Another feature to simplify operations is the FDIR concept of the FSFW: Due to the hierarchical isolation of failures, ideally on assembly level, mission operations can continue even in the presence of arbitrary equipment faults, as the system can recover from most faults without a mode switch.

For example, almost every higher-mode ACS sensor on *Flying Laptop* gets stuck due to radiation from time to time, but such errors are handled by the on-board FDIR, e. g. by power-cycling equipment, without interrupting the mission. The system proved so robust that first night-time, and later week-end and public holiday operations were fully automated to reduce operator workload.

Common interfaces and components also pay off for on-board maintenance. The `HasParametersIF` interface ensures that parameter manipulation to fine-tune the system happens in a common, type-safe manner, and due to the assignment of parameters to components, addressing is obvious.

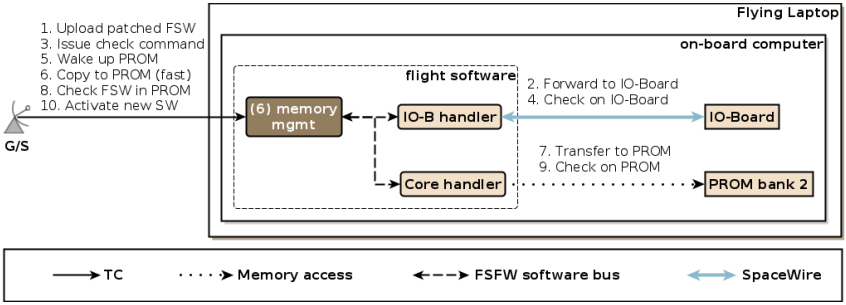

Figure 5.3.: Illustration of the software upload process of *Flying Laptop*, using standard PUS memory service commands and a custom memory copy subservice, in combination with the `HasMemoryIF` interface.

Also, the `HasMemoryIF` facilitates moving software patches around the system: For example, this interface allowed introducing a memory copy subservice in the PUS memory service in *Flying Laptop*, which is capable of moving raw memory from one component implementing `HasMemoryIF` to another. In effect, FSW patches can be uploaded to the IO-Board memory first and can be checked there, before performing a fast copy to the critical boot memory (see Figure 5.3).

5.4. Software Reuse with the FSFW

Another main goal of the FSFW, aside from loose coupling to facilitate development, testing and operations, is improving software reuse. Software reuse shall avoid developing almost, but not entirely identical applications over and over again.

And indeed, using the FSFW enhances both internal and external software reuse:

- **Internal reuse** is improved by identifying and eliciting recurring functionality and refactoring this code in an abstract fashion. The goal is to minimize code duplication, by providing common building blocks and component templates, as described in Section 5.1 above.
- **External reuse** is supported by the three software layers of the FSFW (Section 4.1.4):
  - **Component layer:** By ensuring components interact with other components, RTOS features and equipment via standardized interfaces only, reusing entire components is a matter of configuration. This allows deploying well-written components in different contexts.
  - **FSFW-Core layer:** The FSFW-Core itself is the basis for FSW reuse, as it holds the individual pieces together. Reuse of components is possible only by utilizing the FSFW-Core.
  - **Platform abstraction layer:** This layer, in the form of interfaces for the underlying RTOS and drivers, ensures that the FSFW-Core and all applications are independent of the execution platform. Thus, as soon as RTOS and device driver wrappers are available, an entire FSFW implementation is portable between different systems.

The applicability of the above concepts has been shown in parts by porting the *Flying Laptop* software to different execution platforms for software testing, e. g. an FPGA-based development board, but also a common Linux computer. In this example, most components are identical to those of the *Flying Laptop* software, but the wrappers and drivers in the platform abstraction layer are replaced.

Moreover, the Flight Software Framework is the basis for the flight software of the Flexible LEO Platform (FLP2), a spin-off from the original *Flying Laptop* design by Airbus Defence and Space for modular small satellite missions [54]. In this setting, the flexibility of the FSFW can be fully exploited, e. g. by providing replaceable software components to support various propulsion options of the platform.

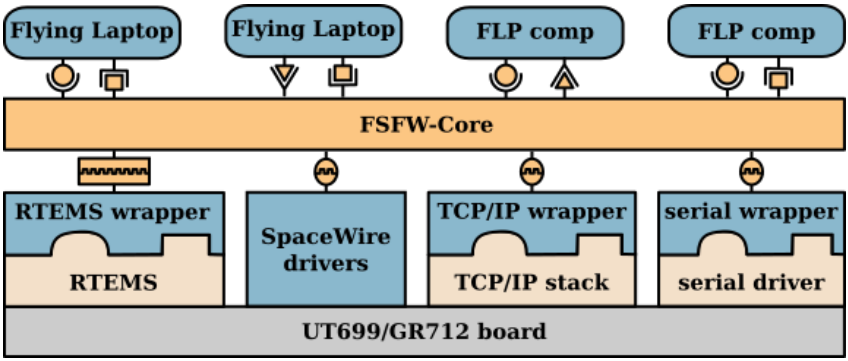

Figure 5.4.: Reuse of the FSFW in the Flexible LEO Platform. It utilizes legacy components from *Flying Laptop* where appropriate, but requires additional interface drivers, as well as dedicated components, e. g. for propulsion systems.

Existing components from *Flying Laptop* can be reused for the platform by means of component configuration. For example, the RW device handler component is highly configurable to the mission on construction, e. g. by setting operational limits and specific identifiers, such as data pool IDs. The software stack for the FLP2 is shown in Figure 5.4.

5.5. Towards an FSFW-based Software Product Line

An important aspect for reusability is whether the FSFW actually provides suitable features for more than just the *Flying Laptop* mission. Thus, it is necessary to check which of the features of a generic spacecraft FSW identified in Chapter 3 are actually supported by the FSFW.

As shown in the feature tracing tables in Appendix B.2, the FSFW supports most of the generic features identified in Section 3.7. Some missing elements, such as additional device interfaces, or low-level space link protocols are simple to be added on demand. Other functionality, such as file system support or another application layer for the space link, as well as execution on distributed systems requires additional effort. These open issues are addressed in Chapter 5.7.

To form a *flight software product line*, a relevant aspect is ease of use for software developers as using a framework as basis will only pay off if it actually simplifies work.

This is the goal of the FSFW in general and of component templates in particular. In their current form, these elements form a white-box framework (see Section 2.3.2), which is a well understood technique for framework implementations.

Still, the wide range of supported generic FSW features indicate that the FSFW is a good fit for many types of space missions. Also, it lays the foundation for better forms of reuse than simple "clone-and-own" strategies and may indeed support flight software development of many space missions to come, even though, for now, the simple drag-and-drop software solution remains a vision on the horizon.

5.6. The FSFW as Real-Time Embedded Software

Due to the design of the FSFW-Core for component communication and execution (Section 4.4), the FSFW is well-suited for embedded systems with regards to real-time capabilities.

However, memory and performance overhead are relevant factors for space missions (see Section 2.4.3), as high-performance processors and memory are expensive due to the harsh space environment. Thus, it is reasonable to evaluate the resource demand introduced by using the FSFW. A comparative analysis would require performance data of FSW from comparable space missions. Unfortunately, such data was not available. Therefore, only the resource demands of the *Flying Laptop* sofware are presented.

Regarding performance, an important quantitative result is that the *Flying Laptop* software, utilizing a 5 Hz attitude control cycle and executing 57 components in 24 tasks, runs on a 33 MHz LEON3-FT microprocessor with about 35% of processor utilization. This is a good indication that using the FSFW does not produce unacceptable performance overhead. Also, it shows that it was a good decision to focus performance optimizations on core elements of the system, such as an efficient implementation of the software bus. A more detailed performance analysis, albeit of an early version of the FSFW, can be found in [11].

For memory utilization, a relevant metric is the code size introduced by the FSFW, as critics of OOP often complain about so-called *code bloat*. As shown in Table 5.1, the FSFW does introduce significant overhead in code size of about 1 MB when compared to an empty RTOS. Still, the increase in code size of about 1.5 MB to a full FSW implementation is rather moderate.

Due to this overhead, using the FSFW may be unattractive for very simple deployments, e. g. on small CubeSats. For more complex use-cases, the advantage of internal code reuse comes into play, which allows building complex

Image	Code Size (B)
RTEMS sample	206 320
Empty FSFW/RTEMS	1 218 624
Flying Laptop software	2 710 736

Table 5.1.: Code size of a simple RTEMS example, an almost empty FSFW deployment and the *Flying Laptop* image. All images are compiled with the RTEMS g++ compiler with full optimization (O3).

applications with little additional code. In *Flying Laptop*, the entire FSW, with execution code, stacks and heap, runs on 8 MiB of RAM.

Thus, even though performance and memory utilization were regarded secondary to functionality aspects and code readability[1], the FSFW is well suited for the computing hardware of current space missions.

5.7. Improvements and Open Issues

Despite the operational success of the FSFW in *Flying Laptop*, which proved its general applicability to develop operational FSW, some known open issues exist in the current implementation. Also, there is room for improvements and extensions. These points are addressed in this section.

5.7.1. Incomplete SOA Implementation

In the FSFW in its current form, there is an imbalance between implementing and calling a common interface over the software bus.

To provide an interface to other components, it is sufficient to implement the adaptation points in the form of method calls, and to instantiate a helper class (c. f. Section 4.4.3), which handles the incoming message and invokes an interface. For example, providing a changeable mode is realized by implementing the `HasModesIF` interface and forwarding incoming messages to a mode helper class. In effect, the helper takes the role of a skeleton in a SOA implementation (Section 2.3.4). Thus, only few details of the underlying implementation are known to a component implementer.

In contrast, there is no support for calling another component's interface: Callers uses the middleware directly. For instance, commanding a mode change requires creating a message, looking up the receiver's address and putting it into

[1] This is in accordance with the first rule of code optimization: *Don't.* [6]

the middleware manually, with the unpleasant consequence that many implementation details of the framework are exposed to the caller. In other words, there is no such thing as a SOA proxy available in the FSFW so far.

Some experiments to close that conceptual gap have been conducted: The FSFW offers a `CommandsActionsIF` and a `CommandActionHelper` to invoke actions of other components. They interact in a similar fashion as normal interfaces and helper classes, but for the commanding side of the interaction, and therefore hide the messaging details from a component implementation. This concept could be transferred to all common interfaces of the FSFW in future upgrades.

5.7.2. Improving `DeviceHandlerBase` implementation

The `DeviceHandlerBase` component template is an important element of the FSFW, which shows that some abstract form of device handling is possible in spacecraft systems (see Section 4.5.1). Still, the functionality of `DeviceHandlerBase` is provided by a large, monolithic base class. Thus, using the template is a all-or-nothing issue.

A better separation of features and a more modular design, e. g. using building blocks for device communication monitoring or power switching, may be more desirable, to avoid instantiating unneeded functionality, but also from a code maintenance perspective. Also, modularization could facilitate adding new functionality, such as power and thermal monitoring.

5.7.3. More Common Interfaces

A number of iterations took place to identify the current set of common interfaces defined in the FSFW. They finally are an adequate basis to control and monitor small satellite systems.

However, to add new features to components of the FSFW, additional interfaces become necessary. Some ideas for such interfaces are presented here.

Persistence

Due to the radiation environment in space, occasional reboots of computers, including the main OBC, must be taken into account. In such cases, it is important to restore the correct state of the system, e. g. for control parameters and health states.

Thus, in a component-based software, it would be ideal to allow loading and saving of the inner state of components in the form of a common interface.

Such an interface is missing in the presented FSFW implementation and would be a good extension for upcoming releases[2].

Distribution of Periodic Data

In its current form, the data pool works reasonably well and provides a good decoupling in time of sensor/actuator and control components. Also, asynchronous access of a data reporting entity, such as the PUS housekeeping service, works quite well.

On the downside, however, the global accessibility violates the encapsulation concept of the FSFW, less because it makes the variable accessible at any time, but more so because it allows other component to change the variable at will, i. e. it is not owned by a single component. Also, the shared memory concept makes accessing specific samples of a variable difficult, as a reader may always miss a sample due to varying timing conditions[3].

A potential better fit to the FSFW architecture would be some well-designed publish-subscribe mechanism on the software bus. Components could publish data or data sets to subscribers and the subscription mechanism could include the requested sampling frequency, e. g. 5 Hz or `ALL`.

This would allow fine-grained control of data exchange between device handler, controller, housekeeping, and diagnostic components, but also make the current data pool implementation obsolete. Therefore, such a change would require redesign of all existing components, as well as new common interfaces for publishing and receiving periodic data.

5.7.4. Handling the Space Link Protocol

Being the only way to interact with a spacecraft, the space link protocol is very important for FSW design. However, as explained in Section 4.1.6, the FSFW has a dedicated internal protocol, which are the messages on the software bus Section 4.4.3.

In the current solution, translation takes place on service level, after incoming commands were distributed according to their address. This approach allows dedicated translation of each telecommand to an optimized protocol on the software bus. Also, standalone components, which accept telecommands without translation, are possible.

However, the concept has some drawbacks (see Figure 5.5): As services are supposed to operate in a dedicated address space, defined by the application

[2] Some unpublished experiments on that topic have been performed by Steffen Gaisser at the IRS.

[3] This is an aliasing effect.

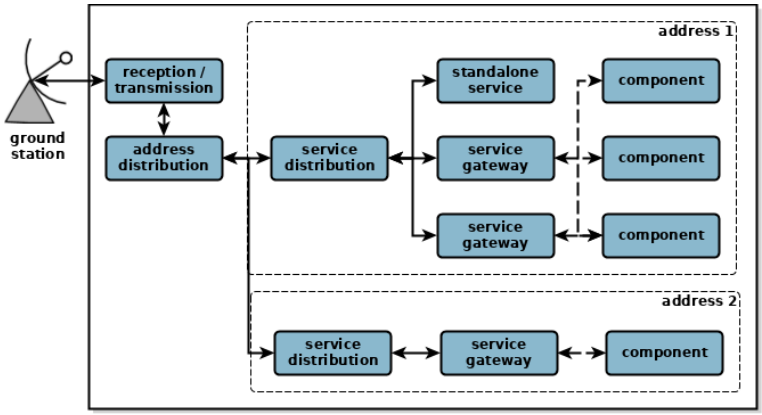

Figure 5.5.: The current space link translation in the FSFW. For each address space, dedicated service gateway components are required. Solid lines indicate TMTC communication, dotted lines messages on the internal software bus.

process identifier (APID), all components behind the gateway belong to that address. Thus, TC and TM packets need to contain an additional component address to allow gateways to identify the command destination. Moreover, for each application process, dedicated instances of all needed service components are required, which is rather costly. For instance, the flight software of *Flying Laptop* utilizes a single APID for all components.

A possible alternative to the current approach would be to establish some form of end-to-end communication between the ground segment and software components, which can for example be realized by using the APID as a component address. This would require changes in the FSFW PUS framework, but existing controller and device handler components work without changes.

5.7.5. Supporting Distributed Computing

Even though FSFW components are well encapsulated and designed for independent execution, building distributed systems is currently not supported. This has some technical background, as certain interaction, such as system initialization, as well as the data pool, work via shared memory in a single address space.

Adjusting the FSFW to handle these issues may be possible, so components could be distributed on several independent nodes, but there is a high risk to

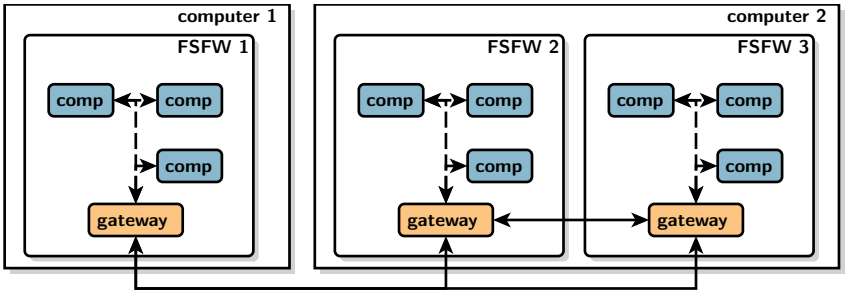

Figure 5.6.: Deploying the FSFW on multiple computers or partitions, using gateways for communication. Solid lines are network communication, dotted lines represents IPC on the FSFW software bus.

introduce a lot of incidental complexity for functionality which is required only by few deployments of the FSFW.

Therefore, a simpler approach would be to run multiple independent instances of the FSFW and introduce to gateways that forward inter-component communication to some on-board network, as illustrated in Figure 5.6. This approach would also work for deploying the FSFW on multiple partitions in one computer. This concept has been successfully implemented for the FLP2 platform at Airbus Defence and Space, where two instances of the FSFW run in the multiprocessor environment of a GR712 system-on-chip.

5.7.6. Evolving a Blackbox Framework

The FSFW is a white-box framework (see Section 2.3.2), i. e. creating components from templates happens by subclassing. This is a well understood technique, but not ideal with regards to information hiding: Creating subclasses always requires a certain amount of insight into structure and dynamic behaviour of base classes and therefore some expertise and training.

This effort could be reduced by evolving the FSFW to a black-box framework, where interaction between components and the framework only happens via interfaces, as illustrated in Figure 2.5 of Section 2.3.2.

However, these techniques are much less investigated than those for white-box frameworks. Therefore, this is rather a promising direction of research than a simple update of the current design. Still, some ideas to use C++ metaprogramming techniques to generate component “containers” are found in Appendix A.3.

6. Summary

To counter the challenges introduced by ever more complex flight software, the Flight Software Framework (FSFW) was developed in the frame of this thesis.

It uses selected software engineering techniques to assist the development of FSW for space missions. The software engineering techniques applied are:

- A *framework* approach, to provide a domain specific tool set to developers to speed up implementation.
- *Software components*, to improve separation of concerns and allow reuse on component basis.
- Communication techniques similar to that in *service-oriented architecture*, to ensure loose coupling between components.

With these concepts, the FSFW forms a component framework, which keeps essential complexity manageable by enforcing strict separation of concerns due to independent components. Also, it avoids incidental complexity by focusing on domain specific needs of software development with the framework elements provided.

To ensures that the FSFW not only covers the need of a specific space mission, but may serve as the basis for a *software product line* for very different satellites, an extensive *domain analysis* is part of this thesis. Its outcome is a *feature tree*, which structures generic requirements for any robotic space software, divided in component and system management, as well as operations and autonomy.

The FSFW covers most features in this tree by providing the following elements:

- A set of *common interfaces*, which reflect functionality a component has or offers to other components. These interfaces are the basis for inter-component communication.
- The *FSFW-Core* allows real-time compatible execution of, and communication between components. Also, it provides abstractions for the underlying execution platform and on-board networks.
- A set of *component templates*, which form a framework to implement typical component types, such as device handlers and controllers.

- The FSFW PUS framework, which supports designing a space link protocol stack to utilize common CCSDS and ECSS protocols.

With these elements, the FSFW is designed to improve software reuse, internally by sharing code with component templates and building blocks and externally by reusing the entire FSFW-Core and components on different missions. This is a key factor to produce high-quality flight software with reduced development effort.

Moreover, the common interfaces of the FSFW introduce a layer of abstraction and generalization, which simplifies the design of complex features, such as hierarchical failure detection, isolation and recovery.

Within this thesis, the framework was not only designed as a big laboratory experiment, but actually used to implement software for the small satellite *Flying Laptop*. The spacecraft performs well in orbit since its launch in July 2017. The testing, integration and operations campaign provided invaluable real-life feedback not only for the *Flying Laptop* software itself, but also for the overall framework design.

In conclusion, the flight software domain analysis in this thesis and the Flight Software Framework itself will hopefully provide a small contribution to master software development of future space missions.

Indeed, the prospects for further use are good: The FSFW is continuously improved at the University of Stuttgart and serves as baseline for planned satellite projects there. Furthermore, Airbus Defence and Space applies the framework on its new *Flexible LEO Platform (FLP2)*.

A. Example Code Listings

A.1. Object-Oriented Programming Examples

Listing A.1: Class and object example

```
1
2  class A {
3  public:
4    //class A constructor
5    A() : value(0) {}
6    //another class A constructor
7    A(int start) : value(start) {}
8    //a method
9    int getValuePlusOne() {
10     return ++value;
11   }
12 private:
13   //an attribute
14   int value;
15 }
16
17 int main() {
18   //create object "myA" as instance of class "A"
19   A myA;
20   //call a method of myA
21   myA.getValuePlusOne();
22   std::cout << myA.getValuePlusOne() << std::endl //prints "2"
23   myA.value = 10; //compile error: "value is private"
24   //create another object "myOtherA"
25   A otherA(3);
26   std::cout << otherA.getValuePlusOne() << std::endl //prints "4"
27 }
```

Listing A.2: Inheritance and interface example

```
1
2  class A; //declaration of class A as in listing A.1
3
4  class PrintIF {
5  public:
6    //C++ expects a virtual destructor
7    virtual ~PrintIF() {}
8    virtual void print() = 0;
9  };
10
11 class C {
12 public:
13   void print() {
14     std::cout << "Class C" << std::endl;
15   }
16 };
17
18 class B : public A, public PrintIF {
19 public:
20   //Implements method print
21   void print() {
22     std::cout << "Class B has a ";
23     myC.print();
24   }
25 private:
26   C myC;
27 };
28
29 int main() {
30   //create object "myB" as instance of class "B"
31   B myB;
32   std::cout << myB.getValuePlusOne() << std::endl //prints "1"
33   myB.print(); //prints "Class B has a Class C"
34   myB.myC.print() //compile error: "myC is private"
35
36   //obtain a reference to myB
37   PrintIF* printable = &myB;
38   printable->print(); //prints "Class B has a Class C"
39   printable->getValuePlusOne(); //compile error
40 }
```

A.1.1. PlacementFactory

Listing A.3: Full source code of `PlacementFactory` implementation

```
1
2  #include <framework/storagemanager/StorageManagerIF.h>
3  #include <utility>
4
5  class PlacementFactory {
6  public:
7    PlacementFactory(StorageManagerIF* backend) :
8        dataBackend(backend) {
9    }
10   template<typename T, typename ... Args>
11   T* generate(Args&&... args) {
12     store_address_t tempId;
13     uint8_t* pData = NULL;
14     ReturnValue_t result = dataBackend->getFreeElement(&tempId,
           sizeof(T),
15         &pData);
16     if (result != HasReturnvaluesIF::RETURN_OK) {
17       return NULL;
18     }
19     T* temp = new (pData) T(std::forward<Args>(args)...);
20     return temp;
21   }
22   template<typename T>
23   ReturnValue_t destroy(T* thisElement) {
24     //Need to call destructor first, in case something was
           allocated by the object
25     //(shouldn't do that, however).
26     thisElement->~T();
27     uint8_t* pointer = (uint8_t*) (thisElement);
28     return dataBackend->deleteData(pointer, sizeof(T));
29   }
30 private:
31   StorageManagerIF* dataBackend;
32 };
```

A.2. DeviceHandlerFailureIsolation::eventReceived

The following code listing shows the complete implementation of the `eventReceived` method of the `DeviceHandlerFailureIsolation` building block. As shown, failure isolation boils down to implementing a switch case over all possible events. Code comments provide rationales for the resulting reactions.There are four reactions implemented:

- **Do nothing**: In case the event is information only, or can't be handled on-board, it is ignored.
- **Increment a fault counter using** `incrementAndCheck`: If only multiple event occurrences indicate a fault, fault counters are used for confirmation. The method returns `true` if the threshold was reached, resulting in another reaction.
- **Get confirmation from a potential "common cause" component**: Using the `sendConfirmationRequest` method, other components can be asked for confirmation. The reply to this confirmation request is handled in dedicated `eventConfirmed` and `wasCommonCause` methods.
- **Initiate a recovery with** `handleRecovery`: The method sets its owner's health state to `needs recovery`. Within the method call, another counter is checked to avoid infinite reboots of devices. If the counter has reached its threshold, the health state is set to `faulty`.
- **Mark the component** `faulty`: In some cases, the component's health state is set to `faulty` directly, as a reboot will not resolve the situation.

With these explanations and the comments in the code, understanding the reaction on different failure events is quite straightforward.

Listing A.4: DeviceHandlerFailureIsolation::eventReceived

```
1  ReturnValue_t
2    DeviceHandlerFailureIsolation::eventReceived(EventMessage* event
         ) {
3    if(isFdirInActionOrAreWeFaulty(event)) {
4      return RETURN_OK;
5    }
6    ReturnValue_t result = RETURN_FAILED;
7    switch (event->getEvent()) {
8    case HasModesIF::MODE_TRANSITION_FAILED:
9    case HasModesIF::OBJECT_IN_INVALID_MODE:
10     //We'll try a recovery as long as defined in MAX_REBOOT.
11     //Might cause some AssemblyBase cycles, so keep number low.
12     handleRecovery(event->getEvent());
13     break;
14   case DeviceHandlerIF::DEVICE_INTERPRETING_REPLY_FAILED:
15   case DeviceHandlerIF::DEVICE_READING_REPLY_FAILED:
16   case DeviceHandlerIF::DEVICE_UNREQUESTED_REPLY:
17   case DeviceHandlerIF::DEVICE_UNKNOWN_REPLY: //Some DH's generate
         generic reply-ids.
18   case DeviceHandlerIF::DEVICE_BUILDING_COMMAND_FAILED:
19     //These faults all mean that there were stupid replies from a
           device.
20     if (strangeReplyCount.incrementAndCheck()) {
21       handleRecovery(event->getEvent());
22     }
23     break;
24   case DeviceHandlerIF::DEVICE_SENDING_COMMAND_FAILED:
25   case DeviceHandlerIF::DEVICE_REQUESTING_REPLY_FAILED:
26     //The two above should never be confirmed.
27   case DeviceHandlerIF::DEVICE_MISSED_REPLY:
28     result = sendConfirmationRequest(event);
29     if (result == HasReturnvaluesIF::RETURN_OK) {
30       break;
31     }
32     //else
33     if (missedReplyCount.incrementAndCheck()) {
34       handleRecovery(event->getEvent());
35     }
36     break;
37   case StorageManagerIF::GET_DATA_FAILED:
38   case StorageManagerIF::STORE_DATA_FAILED:
39     //Rather strange bugs, occur in RAW mode only. Ignore.
40     break;
41   case DeviceHandlerIF::INVALID_DEVICE_COMMAND:
42     //Ignore, is bad configuration. We can't do anything in flight
           .
43     break;
44   case HasHealthIF::HEALTH_INFO:
45   case HasModesIF::MODE_INFO:
46   case HasModesIF::CHANGING_MODE:
47     //Do nothing, but mark as handled.
48     break;
49     //****Power*****
50   case PowerSwitchIF::SWITCH_WENT_OFF:
51     result = sendConfirmationRequest(event, powerConfirmation);
52     if (result == RETURN_OK) {
53       setFdirState(DEVICE_MIGHT_BE_OFF);
54     }
```

```
55        break;
56      case Fuse::FUSE_WENT_OFF:
57        //Not so good, because PCDU reacted.
58      case Fuse::POWER_ABOVE_HIGH_LIMIT:
59        //Better, because software detected over-current.
60        setFaulty(event->getEvent());
61        break;
62      case Fuse::POWER_BELOW_LOW_LIMIT:
63        //Device might got stuck during boot, retry.
64        handleRecovery(event->getEvent());
65        break;
66        //****Thermal*****
67      case ThermalComponentIF::COMPONENT_TEMP_LOW:
68      case ThermalComponentIF::COMPONENT_TEMP_HIGH:
69      case ThermalComponentIF::COMPONENT_TEMP_OOL_LOW:
70      case ThermalComponentIF::COMPONENT_TEMP_OOL_HIGH:
71        //Well, the device is not really faulty, but it is required to
              stay off as long as possible.
72        setFaulty(event->getEvent());
73        break;
74      case ThermalComponentIF::TEMP_NOT_IN_OP_RANGE:
75        //Ignore, is information only.
76        break;
77        //*******Default monitoring variables. Are currently not used
              .*****
78  //    case DeviceHandlerIF::MONITORING_LIMIT_EXCEEDED:
79  //      setFaulty(event->getEvent());
80  //      break;
81  //    case DeviceHandlerIF::MONITORING_AMBIGUOUS:
82  //      break;
83      default:
84        //We don't know the event, someone else should handle it.
85        return RETURN_FAILED;
86    }
87    return RETURN_OK;
88  }
```

A.3. Container Generation For A Black Box Framework

This is a compiling demonstration of how to use template metaprogramming to instantiate containers that automatically instantiate and call helper classes, depending on the interfaces the component implements. This technique may serve as an idea to enhance the FSFW to form a black box software framework.

Listing A.5: Example Code to generate framework containers.

```
1  #include <iostream>
2  #include <type_traits>
3
4  using namespace std;
5
6  //Definitions for interface A
7
8  //A allows some callback to the FW
9  class ACallbackIF {
10 public:
11   virtual ~ACallbackIF() {
12   }
13   virtual void doCallback() = 0;
14 };
15
16 class AfunctionalityIF {
17 public:
18   virtual ~AfunctionalityIF() {
19   }
20   virtual void setCallbackForA(ACallbackIF* callback) = 0;
21   virtual void doA() = 0;
22 };
23
24 class AHelper: public ACallbackIF {
25 public:
26   void helpA(AfunctionalityIF* user) {
27     cout << "Helping A.." << endl;
28     user->doA();
29   }
30   void doCallback() {
31     cout << "Calling back AHelper. Hello!" << endl;
32   }
33 };
34
35 //Using SFINAE
36 template<class T, typename = void>
37 class ContainerA {
38 public:
39   void doAction(T* user) {
40   }
41   void initializeUser(T* user) {
42   }
43 };
44
45 template<class T>
46 class ContainerA<T,
47     typename enable_if<is_base_of<AfunctionalityIF, T>::value>::
         type> {
48 public:
```

```
49    AHelper aHelper;
50    void doAction(T* user) {
51      aHelper.helpA(user);
52    }
53    void initializeUser(T* user) {
54      cout << "... for A impl." << endl;
55      user->setCallbackForA(&aHelper);
56    }
57  };
58
59  //Definitions for interface B
60
61  class BfunctionalityIF {
62  public:
63    virtual ~BfunctionalityIF() {
64    }
65    virtual void doB() = 0;
66  };
67
68  class BHelper {
69  public:
70    void helpB(BfunctionalityIF* user) {
71      cout << "Helping B.." << endl;
72      user->doB();
73    }
74  };
75
76  template<class T, typename = void>
77  class ContainerB {
78  public:
79    void doAction(T* user) {
80    }
81  };
82
83  template<class T>
84  class ContainerB<T,
85      typename enable_if<is_base_of<BfunctionalityIF, T>::value>::
            type> {
86  public:
87    BHelper bHelper;
88    void doAction(T* user) {
89      bHelper.helpB(user);
90    }
91  };
92
93  //The complete container, aware of all
94  //existing interfaces.
95
96  template<typename T>
97  class Container {
98  public:
99    T instance;
100
101   ContainerA<T> containerA;
102   ContainerB<T> containerB;
103
104   void initialize() {
105     cout << "Initializing container..." << endl;
106     containerA.initializeUser(&instance);
107   }
108
```

```
109     void doAction() {
110       containerA.doAction(&instance);
111       containerB.doAction(&instance);
112     }
113   };
114
115   class SmallComponent: public BfunctionalityIF {
116   public:
117     void doB() {
118       cout << "Small: B " << endl;
119     }
120   };
121
122   class LargeComponent: public AfunctionalityIF, public
          BfunctionalityIF {
123   public:
124     void doA() {
125       cout << "Large: A " << endl;
126       callbackForA->doCallback();
127     }
128     virtual void setCallbackForA(ACallbackIF* callback) {
129       callbackForA = callback;
130     }
131     void doB() {
132       cout << "Large: B " << endl;
133     }
134     void doC() {
135       cout << "Large: C " << endl;
136     }
137   private:
138     ACallbackIF* callbackForA;
139   };
140
141   class NoneComponent {
142   public:
143     void doNothing() {
144     }
145   };
146
147   int main() {
148     Container<LargeComponent> myContainer;
149     myContainer.initialize();
150     myContainer.doAction();
151     cout << "Size of large container: " << sizeof(Container<
            LargeComponent> )
152          << endl;
153
154
155     Container<SmallComponent> myContainer2;
156     myContainer2.initialize();
157     myContainer2.doAction();
158     //Helpers for IF A are not included.
159     cout << "Size of small container: " << sizeof(Container<
            SmallComponent> )
160          << endl;
161
162     Container<NoneComponent> myContainer3;
163     myContainer3.initialize();
164     //Doesn't do anything at all.
165     myContainer3.doAction();
```

```
166     //There's a single pointer overhead, as an instance of
            NoneComponet is created.
167     cout << "Size of none container: " << sizeof(Container<
            NoneComponent> ) << endl;
168 }
169 }
```

B. Domain Analysis Details

B.1. Historical Domain Analysis Results

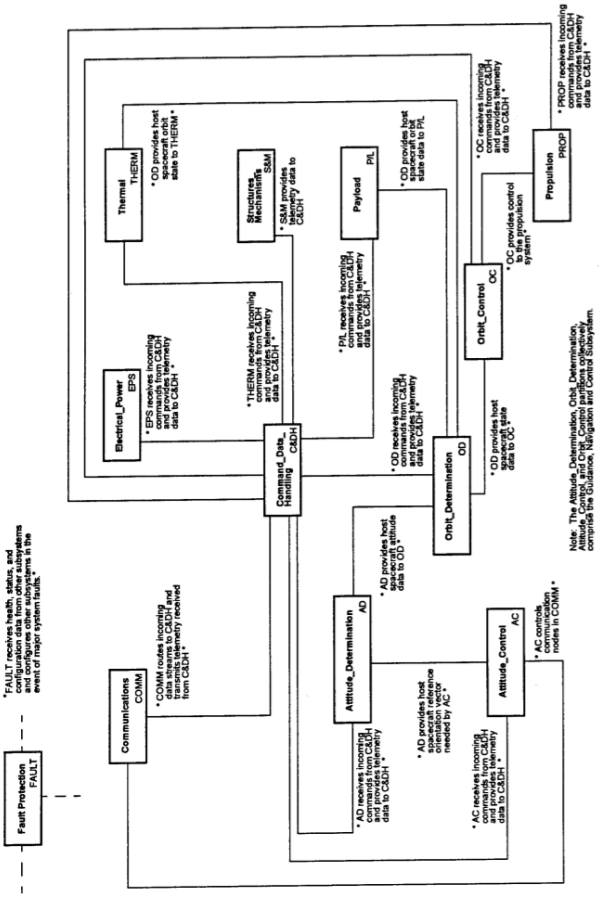

Figure B.1.: Summary of the generic subsystem relationships found in a 1995 domain analysis for spacecraft FSW [57].

B.2. Feature Tracing Tables

The first set of tables in Section B.2.1 ensures a backtrace from the synthesized FSW features identified in the domain analysis in Chapter 3 to the original sources, e. g. features identified in a CCSDS standard.

The second set of tables in Section B.2.2displays which of these synthesized features are actually implemented or supported in the FSFW of this thesis.

B.2.1. Backtrace Tables

Component Management

ID	Name	FLP	CCSDS	ECSS	FSW
Component execution					
C.1.1	Cyclic execution	2			
C.1.2	Execution control				3
C.1.3	RTOS abstraction				1
C.1.4	Schedulability			3	
C.1.5	Hardware abstraction	20			
C.1.6	Distributed computing	21			
C.1.6a	- IPC gateway				7
C.1.7	Off-the-shelf components				8
Component support					
C.2.1	Inter-process communication		20		
C.2.1a	- Data distribution	3	18		
C.2.1b	- Event distribution		9	11	
C.2.2	Run-time configuration data				4
C.2.2a	- File management			15	
C.2.3	Time management		22	7	
C.2.4	Action Execution	1			
C.2.5	Component Modes	5			
Maintenance					
C.3.1	Resource monitoring			23	
C.3.2	Memory scrubbing				5

Table B.1.: Synthesis of generic FSW requirements for component management.

System Management

ID	Name	FLP	CCSDS	ECSS	FSW
Equipment					
S.1.1	Equipment monitoring	9			
S.1.2	Equipment access		14	12	
S.1.2a	- Commanding	7			
S.1.2b	- Data acquisition	7	17		
S.1.3	Value conversion		19		
S.1.4	Equipment representation		15		
S.1.5	Equipment modes	8	16		
Subsystems					
S.2.1	Redundancy management	12			
S.2.2	Health states	11			
S.2.3	Subsystem representation	16			
S.2.4	Subsystem modes			18	
S.2.4a	- Mode transitions			18.4	
On-Board Communication					
S.3.1	Communication Layering	6	33		
S.3.2	Subnetwork access		13		
S.3.2a	-Packet based		31		
S.3.2b	-Memory based		32		
S.3.3	Network types				
S.3.3a	- Mulitdrop buses		23		
S.3.3b	- Point-to-point		24		
S.3.3c	- Homogeneous networks		25		
S.3.4	Interface types				
S.3.4a	- MIL-STD-1553B		26		
S.3.4b	- SpaceWire		27		
S.3.4b	– RMAP			4	
S.3.4b	– CCSDS packet transport			5	
S.3.4c	- CAN		28		
S.3.4d	- Ethernet		29		
S.3.4e	- Embedded serial buses	18			

Table B.2.: Synthesis of generic FSW requirements for system management.

Operations

ID	Name	FLP	CCSDS	ECSS	FSW
Control					
O.1	Commandability		12		
O.1.1	Common commanding	15			
O.1.2	Action commanding		12.1		
O.1.3	Parameter access	4	12.2		
O.1.4	Memory access			14	
O.1.5	Mode commanding			18.1, 18.2	
O.1.6	Critical commands			16	
O.1.7	Authentication and authorization			17	
Monitoring					
O.2	Observability		13		
O.2.1	Event reporting		13.1		
O.2.2	Housekeeping reporting		13.2		
O.2.3	Activity reporting		11		
O.2.4	OB monitor reporting	10	13.3		
Space link					
O.3.1	Space link abstraction				2
O.3.2	Data Link Layer				
O.3.2a	- CCSDS TM frames		1		
O.3.2a	- CCSDS TC frames		2		
O.3.2a	– COP-1		2.1		
O.3.2b	- AX.25	19			
O.3.3	Networking layer				
O.3.3a	- Space packets		3		
O.3.3a	– Space packet routing		3.1		
O.3.3b	- Encapsulation		4		
O.3.4	Higher layers				
O.3.4a	- PUS space packet binding			6	
O.3.4b	- MO MAL space packet binding		7		
O.3.4b	MAL interaction patterns		6		
O.3.5	Time Codes		5		

Table B.3.: Synthesis of generic FSW requirements for operations.

Autonomy

ID	Name	FLP	CCSDS	ECSS	FSW
Autonomous operations					
A.1.1	Command scheduling	13			
A.1.1a	- command injection			8	
A.1.1b	- command storage			9	
A.1.2	Information storage		10		
A.1.2a	- File store		21.1		6
A.1.2b	- Packet store		21.2	10	
A.1.3	Mission goal representation			19	
A.1.4	Procedure execution engine			24	
Fault management FDIR					
A.2	Survivability	14			
A.2.1	On-board monitoring	10	13.3		
A.2.2	Failure reaction			25	
A.2.3	Hierarchical FDIR			20	

Table B.4.: Synthesis of generic FSW requirements for on-board autonomy.

B.2.2. Feature Tracing Tables of the FSFW

Component Management

ID	Name	Impl.	Remark
Component execution			
C.1.1	Cyclic execution	**Yes**	`PeriodicTaskIF`
C.1.2	Execution control	No	Not possible to start, stop, restart tasks
C.1.3	RTOS abstraction	**Yes**	FSFW-Core OSAL
C.1.4	Schedulability	**Yes**	See Appendix C.2.2
C.1.5	Hardware abstraction	**Yes**	FSFW-Core OSAL and `DeviceCommunicationIF`
C.1.6	Distributed computing	No	See Section 5.7.5
C.1.6a	- IPC gateway	No	Subfeature of C.1.6
C.1.7	Off-the-shelf components	**Yes**	Possible e. g. for DH components
Component support			
C.2.1	Inter-process communication	**Yes**	FSFW-Core
C.2.1a	- Data distribution	**Yes**	software bus, data pool
C.2.1b	- Event distribution	**Yes**	event manager
C.2.2	Run-time configuration data	**Yes**	Container building blocks
C.2.2a	- File management	No	May be useful, not tried yet.
C.2.3	Time management	**Yes**	FSFW-Core clock IF
C.2.4	Action Execution	**Yes**	`HasActionsIF`
C.2.5	Component Modes	**Yes**	`HasModesIF`
Maintenance			
C.3.1	Resource monitoring	No	FSFW-Core error reporting implemented, but no continuous monitoring of SW resources.
C.3.2	Memory scrubbing	No	Mission-specific memory scrubbing implemented, but no generic functionality.

Table B.5.: Implemented component management features of the FSFW.

System Management

ID	Name	Impl.	Remark
Equipment			
S.1.1	Equipment monitoring	**Yes**	Monitoring building blocks and `DeviceHandlerBase`
S.1.2	Equipment access	**Yes**	`DeviceHandlerIF` for low-level, `HasActionsIF` for "virtual" access
S.1.2a	- Commanding	**Yes**	`DeviceHandlerBase`
S.1.2b	- Data acquisition	**Yes**	`DeviceHandlerBase`
S.1.3	Value conversion	**Yes**	`DeviceHandlerBase`
S.1.4	Equipment representation	**Yes**	`DeviceHandlerBase`
S.1.5	Equipment modes	**Yes**	`DeviceHandlerBase`, `HasModesIF`
Subsystems			
S.2.1	Redundancy management	**Yes**	`AssemblyBase`
S.2.2	Health states	**Yes**	`HasHealthIF`
S.2.3	Subsystem representation	**Yes**	`SubsystemBase`
S.2.4	Subsystem modes	**Yes**	`SubsystemBase`, `HasModesIF`
S.2.4a	- Mode transitions	**Yes**	`SubsystemBase` and `Subsystem`
On-Board Communication			
S.3.1	Communication Layering	**Yes**	`DeviceCommunicationIF`
S.3.2	Subnetwork access	**Yes**	`DeviceCommunicationIF`
S.3.2a	-Packet based	**Yes**	`DeviceCommunicationIF`
S.3.2b	-Memory based	**Yes**	`HasMemoryIF`
S.3.3	Network types		
S.3.3a	- Mulitdrop buses	No	Not needed yet, should fit with `FixedTimeslotTaskIF`
S.3.3b	- Point-to-point	**Yes**	SpaceWire implementation
S.3.3c	- Homogeneous networks	**Yes**	Tested with TCP/IP stack
S.3.4	Interface types		
S.3.4a	- MIL-STD-1553B	No	Not needed yet.
S.3.4b	- SpaceWire	**Yes**	Custom driver
S.3.4b	– RMAP	**Yes**	Custom driver
S.3.4b	– CCSDS packet transport	No	Not needed yet.
S.3.4c	- CAN	No	Not needed yet.
S.3.4d	- Ethernet	**Yes**	For testing purposes, no qualified implementation.
S.3.4e	- Embedded serial buses	No	Not needed yet.

Table B.6.: Implemented system management features of the FSFW.

Operations

ID	Name	Impl.	Remark
Control			
O.1	Commandability		
O.1.1	Common commanding	**Yes**	Common interfaces
O.1.2	Action commanding	**Yes**	`HasActionsIF`
O.1.3	Parameter access	**Yes**	`HasParametersIF`
O.1.4	Memory access	**Yes**	`HasMemoryIF`
O.1.5	Mode commanding	**Yes**	`HasModesIF`
O.1.6	Critical commands	No	Hardware high-priority commands used.
O.1.7	Authentication and authorization	No	Not deemed necessary.
Monitoring			
O.2	Observability		
O.2.1	Event reporting	**Yes**	Event distribution, service implementation
O.2.2	Housekeeping reporting	**Yes**	Data pool, service implementation
O.2.3	Activity reporting	**Yes**	PUS TC verification
O.2.4	OB monitor reporting	**Yes**	Supported by monitoring building blocks
Space link			
O.3.1	Space link abstraction	**Yes**	FSFW-Core is independent
O.3.2	Data Link Layer		
O.3.2a	- CCSDS TM frames	No	Not necessary, easily extensible.
O.3.2a	- CCSDS TC frames	**Yes**	FSFW PUS framework
O.3.2a	– COP-1	**Yes**	FSFW PUS framework
O.3.2b	- AX.25	No	Not needed yet.
O.3.3	Networking layer		
O.3.3a	- Space packets	**Yes**	`SpacePacket` classes
O.3.3a	– Space packet routing	**Yes**	`TcDistributor`, interfaces
O.3.3b	- Encapsulation	No	Not needed yet.
O.3.4	Higher layers		
O.3.4a	- PUS space packet binding	**Yes**	FSFW PUS framework
O.3.4b	- MO MAL space packet binding	No	See Section 5.7.4
O.3.4b	MAL interaction patterns	No	See Section 5.7.4
O.3.5	Time Codes	**Yes**	`CCSDSTime` library

Table B.7.: Implemented operations features provided by the FSFW.

Autonomy

ID	Name	Impl.	Remark
Autonomous operations			
A.1.1	Command scheduling	**Yes**	FSFW PUS framework
A.1.1a	- command injection	**Yes**	`TCDistributor`
A.1.1b	- command storage	**Yes**	`StorageMangerIF`, `PoolManager`
A.1.2	Information storage	**Yes**	
A.1.2a	- File store	No	No file system support yet.
A.1.2b	- Packet store	**Yes**	PUS-based
A.1.3	Mission goal representation	No	
A.1.4	Procedure execution engine	No	
Fault management FDIR			
A.2	Survivability	**Yes**	FSFW FDIR framework
A.2.1	On-board monitoring	**Yes**	monitoring building blocks
A.2.2	Failure reaction	**Yes**	FSFW FDIR framework
A.2.3	Hierarchical FDIR	**Yes**	FSFW FDIR framework

Table B.8.: Implemented autonomy features provided by the FSFW.

C. FSFW Details

C.1. *Flying Laptop* Mode Tree

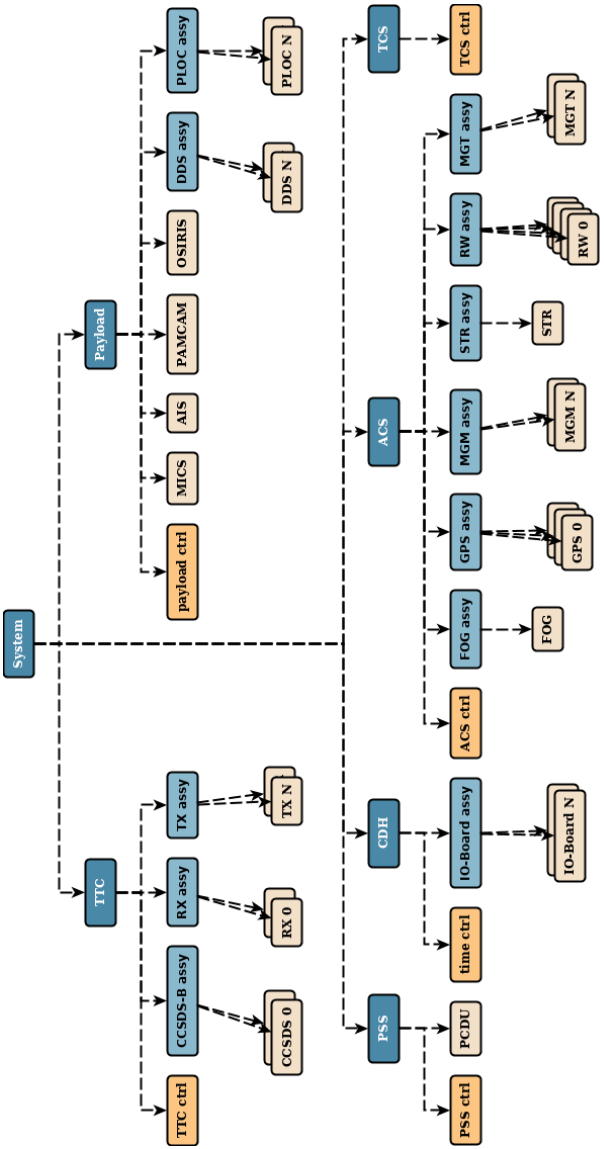

Figure C.1.: Mode tree of *Flying Laptop*.

C.2. More FSFW-Core Features

This appendix sums up some technical implementation details of the FSFW core.

C.2.1. Large Data Distribution

As described in Section 4.4.3, data distribution happens mainly via messages. However, the queue implementations in most RTOSs use allocated slots of messages for queuing. Thus, the maximum message size is a critical parameter for memory utilization, especially if the number of message queues reaches multiple hundreds. Therefore, messages are kept as small as possible.

To enable forwarding of large data chunks, e. g. for a software patch, the message queue mechanism is supplemented with a shared memory mechanism for large data transfer. Its basic idea is to reserve a certain amount of space in a shared memory and use messages to pass the address of the memory between components. Thus, the address works as a token, with the component holding the token responsible for the data. With this mechanism, copying of data is eliminated and locking reduced to the reservation and deletion of data.

Even though the idea allows very efficient distribution of both small and large amounts of data, there is some risk in this concept. First, creating multiple copies of the address ID token must be avoided, otherwise, components may operate on invalid data. Moreover, as deletion of data is in the responsibility of components, there is a risk for memory leaks due to erroneous implementations. Therefore, accessing the shared memory should ideally be handled entirely within the well-tested FSFW-Core, e. g. in helper classes, and not be accessible by users of the framework.

The shared memory is effectively implemented using a container introduced in Section 4.4.8, enhanced by locking mechanisms for thread safety.

C.2.2. Schedulability Analysis

In real-time systems there is always a strong focus on schedulability, which means that provisions are taken to make sure no unfortunate combination of task execution exists such that one tasks misses its intended deadline. Even though spacecraft are rather soft than hard deadline systems, i. e.there is no immediate danger to the system if a deadline is missed, a schedulability analysis is often mandatory for spacecraft (see e. g. [42], section 5.8.3.11).

As already stated in Section 4.4.2, the scheduling policy for the FSFW depends on the RTOS used. Still, in [13] was a concept devised for the common fixed priority preemptive or rate monotonic scheduling (RMS), as e. g. provided by

RTEMS. In summary, one needs to calculate the overall processor utilization for each task, taking into account interruption by higher priority tasks and blocking due to shared resources. This is reflected in the formula (from [95]):

$$\forall i, 1 < i < n, \sum_{k \in hp(\tau_i)} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$$

This formula needs to be calculated for any given task τ_i in the system. $hp(i)$ denotes all tasks with higher priority than τ_i, C_i is the worst-case execution time (WCET) of a task, T_i its period and B_i the worst-case blocking time, which is the longest interval a task with a lower priority than τ_i holds a common shared resource. The right-hand side of the inequation determines a upper limit for processor utilization and converges to $\approx 69\%$. To correctly include PST implementations into the analysis, one needs to convert it into a number of periodic tasks with same priority but shorter period than the original polling task.

However, the difficulty of performing a schedulability is not the mathematics, but to determine reasonable numbers for the WCETs C_i and blocking times B_i. This is especially difficult with large message queues, as the types of incoming messages often have a large impact on task duration. The FSFW does currently not provide any built-in features for component execution time measurement, however, an appropriate extension would be useful for schedulability analysis.

Bibliography

[1] C. Atkinson, C. Bunse, H.-G. Gross, and C. E. Peper. *Component-Based Software Development for Embedded Systems*. Springer-Verlag Berlin Heidelberg, 2005.

[2] AUTOSAR. AUTOSAR standard specification release 4.2.2. Technical report, AUTOSAR development partnership, 2015.

[3] M. F. Barschke and K. Gordon. TUBiX20 - A generic systems architecture for a single failure tolerant nanosatellite platform. In *Proceedings of the 65th International Astronautical Congress*, Toronto, Canada, October 2014.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001.

[6] R. Bemrose. The rules of code optimization, 2007. [https://blogs.msdn-.microsoft.com/audiofool/2007/06/14/the-rules-of-code-optimization/].

[7] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] Bright Ascension. generationOne flight software development kit. Product Sheet 1, Bright Ascension Ltd., Aug. 2017. [http://www-.brightascension.com/wordpress/wp-content/uploads/2017/08/Flight-SDK-Datasheet.pdf; accessed Dec 21, 2017].

[9] F. P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] H. Bruyninckx. V model from structured systems design, 2008. [https://upload.wikimedia.org/wikipedia/commons/f/f9/V-model.svg; accessed July 6, 2016].

[11] N. Bucher. Integration of attitude control functions into the on-board software for the small satellite Flying Laptop. Diploma thesis, IRS, Universität Stuttgart, 2012.

[12] N. Bucher, U. Mohr, B. Bätz, K.-S. Klemich, S. Klinkner, and J. Eickhoff. Functionial verification of the small satellite Flying Laptop. In *Small Satellites for Earth Observation of the IAA*, number IAA-B11-0214. IAA, Apr. 2017.

[13] B. Bätz. Development of scheduling design and equipment handling algorithms for the software of the small satellite Flying Laptop. Diploma thesis, IRS, Universität Stuttgart, 2011.

[14] B. Bätz, U. Mohr, R. Witt, N. Bucher, and J. Eickhoff. Applying dynamic development methods to satellite software: The software framework for the FLP micro-satellite platform. In *4S Symposium*. ESA, 2014.

[15] F. Böhringer. *Nutzlasten des universitären Kleinsatelliten Flying Laptop - Entwicklung, Aufbau und Qualifikation*. PhD thesis, Universität Stuttgart, 2014.

[16] CCSDS. Space packet protocol. Recommendation for Space Data System Standard 133.0-B-1, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Sept. 2003.

[17] CCSDS. Encapsulation service. Recommendation for Space Data System Standard 133.1-B-2, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Oct. 2009.

[18] CCSDS. Communications Operations Procedure-1. Recommendation for Space Data System Standard 232.1-B-2, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Sept. 2010.

[19] CCSDS. Mission Operations services concepts. Informational Report 520.0-G-3, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Dec. 2010.

[20] CCSDS. TC synchronization and channel coding. Recommendation for Space Data System Standard 231.0-B-2, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Sept. 2010.

[21] CCSDS. Time code formats. Recommendation for Space Data System Standard 301.0-B-4, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Nov. 2010.

[22] CCSDS. TM synchronization and channel coding. Recommendation for Space Data System Standard 131.0-B-2, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Aug. 2011.

[23] CCSDS. Spacecraft Onboard Interface Services. Informational Report 850.0-G-2, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Dec. 2013.

[24] CCSDS. Mission Operations Common Object Model. Recommended Standard 521.1-B-1, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Feb. 2014.

[25] CCSDS. Mission Operations MAL space packet transport binding and binary encoding. Recommended Standard 524.1-B-1, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Aug. 2015.

[26] CCSDS. TC space data link protocol. Recommendation for Space Data System Standard 232.0-B-3, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Sept. 2015.

[27] CCSDS. TM space data link protocol. Recommendation for Space Data System Standard 132.0-B-2, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Sept. 2015.

[28] CCSDS. Mission Operations monitor & control services. Draft Recommended Standard 522.1-R-4, Consulative Committee for Space Data Systems (CCSDS), Newport Beach, CA, Apr. 2017.

[29] V. Cechticky, A. Pasetti, and W. Schaufelberger. The adaptability challenge for embedded control system software. In *IFAC World Congress*, pages 155–160, Prague, Czech Republic, July 2005.

[30] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[31] S. DeCarlo. The Fortune 500 list. *Fortune*, 2015.

[32] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[33] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, Oct. 1972.

[34] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. Microservices: Migration of a mission critical system. *CoRR*, abs/1704.04173, 2017.

[35] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. *SIGPLAN Not.*, 31(10):306–323, Oct. 1996.

[36] D. L. Dvorak. NASA study on flight software complexity. Technical report, National Aeronautics and Space Administration, 2009.

[37] ECSS. Ground systems and operations — Telemetry and telecommand packet utilization. Space Engineering ECSS-E-70-41A, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Jan. 2003.

[38] ECSS. Interface and communication protocol for MIL-STD-1553B data bus onboard spacecraft. Space Engineering ECSS-E-ST-50-13C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Nov. 2008.

[39] ECSS. Space segment operability. Space Engineering ECSS-E-ST-70-11C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, July 2008.

[40] ECSS. Spacecraft discrete interfaces. Space Engineering ECSS-E-ST-50-14C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, July 2008.

[41] ECSS. SpaceWire – Links, nodes, routers and networks. Space Engineering ECSS-E-ST-50-12C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, July 2008.

[42] ECSS. Software. Space Engineering ECSS-E-ST-40C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Mar. 2009.

[43] ECSS. Spacecraft on-board control procedures. Space Engineering ECSS-E-ST-70-01C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Apr. 2010.

[44] ECSS. SpaceWire - CCSDS packet transfer protocol. Space Engineering ECSS-E-ST-50-53C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Feb. 2010.

[45] ECSS. SpaceWire - remote memory access protocol. Space Engineering ECSS-E-ST-50-52C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Feb. 2010.

[46] ECSS. Satellite attitude and orbit control system (AOCS) requirements. Space Engineering ECSS-E-ST-60-30C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Aug. 2013.

[47] ECSS. ECSS document tree - Disciplines, 2014. [http://ecss.nl/standards/ecss-document-tree-and-status/; accessed Nov 14, 2017].

[48] ECSS. CANbus extension protocol. Space Engineering ECSS-E-ST-50-15C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, May 2015.

[49] ECSS. Ground systems and operations — Telemetry and telecommand packet utilization. Space Engineering ECSS-E-ST-70-41C, European Cooperation for Space Standardization (ECSS), Noordwijk, The Netherlands, Apr. 2016.

[50] J. Eickhoff. *Simulating Spacecraft Systems*. Springer Aerospace Technology, 2009.

[51] J. Eickhoff. *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*. Springer Aerospace Technology. Springer Berlin Heidelberg, 2011.

[52] J. Eickhoff. *A Combined Data and Power Management Infrastructure*. Springer Aerospace Technology. Springer Berlin Heidelberg, 2013.

[53] J. Eickhoff. Multifunctional controller for a satellite with reconfiguration functions for an on-board computer and disconnection of electrical consumers in case of a fault. European Patent EP2 665 205 A2, Nov. 2013.

[54] J. Eickhoff. Flexible LEO platform - Modularity for small missions. Product Sheet 1, Airbus Defence and Space, Nov. 2017.

[55] J. Eickhoff (Ed.). *The FLP Microsatellite Platform: Flight Operations Manual*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[56] ESA. SAVOIR on-board software reference architecture training material. Handbook SAVOIR-HB-001, European Space Agency, July 2015.

[57] A. Flanegan and M. Benjamin. Satellite domain analysis for reusable software architecture for spacecraft. Final Report PL-TR–95-1007, Defense Technical Information Center, Fort Belvoir, VA, Dec. 1995.

[58] J. Fuegi and J. Francis. Lovelace and Babbage and the creation of the 1843 'notes'. *Annals of the History of Computing*, 2003.

[59] K. Fujisaki and O. Garnatz. AUTOSAR Trends around the Globe. *AUTOSAR Open Conference*, 2015.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[61] R. Gessner, B. Kösters, A. Hefler, R. Eilenberger, J. Hartmann, and M. Schmidt. Hierarchical FDIR concepts in S/C systems. In *Space OPS 2004 Conference*, Montreal, Quebec, Canada, 2004.

[62] H. Goedvolk, A. S. van Schijndel, V. van Swede, R. Tolido, and D. Rijsenbrij (Ed.). An outline about the design, development and deployment of ICT systems in the 21st century, 2000. [http://home.kpn.nl/daanrijsenbrij/progx/eng/index.htm; accessed October 26, 2016].

[63] G. Grillmayer. *An FPGA based Attitude Control System for the Micro-Satellite Flying Laptop*. PhD thesis, Universität Stuttgart, 2009.

[64] P. Hagel. *Modular Software Architecture for FPGA based Payload Module Computers (working title)*. PhD thesis, Universität Stuttgart, exp. 2018.

[65] D. Herity. C++ in embedded systems: Myth and reality. *EE Times India*, 1998.

[66] D. Herity. Modern C++ in embedded systems - part 1: Myth and reality, 2015. [http://www.embedded.com/design/programming-languages-and-tools/4438660/1/Modern-C–in-embedded-systems—Part-1–Myth-and-Reality; accessed November 17, 2016].

[67] G. Hohpe. Software service engineering - Architect's dream or developer's nightmare? In *Dagstuhl Seminar Proceedings*, 2009.

[68] G. J. Holzmann. Landing a spacecraft on Mars. *IEEE Software*, 30(2):83–86, March 2013.

[69] IEEE. Standard for information technology–Portable operating system interface (POSIX(R)) base specifications, issue 7. Technical report, IEEE, Sept 2016.

[70] ISO/IEC. Information technology - Open Systems Interconnection - basic reference model: The basic model. Technical report, ISO/IEC, Vernier, Geneva, Switzerland, 1994.

[71] ISO/IEC. Terminoligy and concepts for SOA. Reference Architecture for Service Oriented Architecture (SOA RA) 18384-1, ISO/IEC, Vernier, Geneva, Switzerland, June 2016.

[72] ISO/IEC/IEEE. Systems and software engineering - Vocabulary. Technical report, ISO/IEC/IEEE, Vernier, Geneva, Switzerland, 2010.

[73] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1988.

[74] A. Jung and J.-L. Terraillon. Faster, later, softer: Cordet–An on-board software reference architecture. In *Workshop on Spacecraft Flight Software (FSW-10)*. European Space Agency ESTEC, Software Systems Division., 2010.

[75] K.-S. Klemich, N. Bucher, M. Böttcher, J. Braun, S. Hilpert, S. Klinkner, and J. Eickhoff. A ground segment for small satellite operations in a university context combining professional and custom software tools. In *SpaceOps 2016 Conference*, pages 3505–3517. AIAA, May 2016.

[76] G. Konecny. Small satellites – A tool for earth observation. In *ISPRS Archives*, 2004.

[77] C. Kormanyos. *Real-Time C++ - Efficient Object-Oriented and Template Microcontroller Programming.* Springer, 2013.

[78] G. Lautenschläger. FDIR experience at Airbus. In *9th ESA Workshop on Avionics, Data, Control and Software Systems - ADCSS*. ESA, 2015. Presentation Slides.

[79] M. Lengowski. *Entwicklung mechanisch/thermischer Architekturen und innovativer Strukturelemente im Rahmen zweier Satellitenmissionen des Stuttgarter Kleinsatellitenprogramms*. PhD thesis, Universität Stuttgart, 2013.

[80] A. T. Manes. SOA is dead; long live services, 2009. [http://apsblog-.burtongroup.com/2009/01/soa-is-dead-long-live-services.html].

[81] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[82] D. McComas, S. Strege, and J. Wilmot. core Flight System (cFS): A low cost solution for SmallSats. In *Annual Small Satellite Conference*. American Institute of Aeronautics and Astronautics and Utah State University, 2015.

[83] U. Mohr. Initial operation of the on-board computer engineering model of the small satellite Flying Laptop. Student thesis, IRS, Universität Stuttgart, Oct. 2011.

[84] U. Mohr, B. Bätz, S. Klinkner, and J. Eickhoff. Aspects of applying an object-oriented design to a small satellite flight software. In *DASIA*. Eurospace, 2015.

[85] S. Montenegro. RODOS: Network-Centric Core Avionics. Overview Version 05, DLR, Nov. 2008.

[86] NASA. core Flight System (cFS) - Background and overview. Training Slides 1, Goddards Space Flight Center, July 2017. [https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf; accessed Dec 07, 2017].

[87] P. Naur and B. Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division*. NATO, 1969.

[88] Object Management Group. OMG Unified Modeling Language TM (OMG UML), superstructure. Technical report, OMG, 2011.

[89] A. Pasetti. *Software Frameworks and Embedded Control Systems*. Springer-Verlag, Berlin, Heidelberg, 2002.

[90] P&P Software. The OBS framework project, 2016. [www.pnp-software.com/ObsFramework/; accessed October 20, 2016].

[91] R. D. Rasmussen. Thinking outside the box to reduce complexity in NASA flight software. *NASA Study on Flight Software Complexity*, May 2009.

[92] E. S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[93] Rockwell-Collins. Reaction wheel assembly RSI 01-5/28 with integrated wheel drive electronics. Technical Description 2, Rockwell-Collins, Oct. 2000. internal use.

[94] K. Schwaber and J. Sutherland. *Scrum Guide*. scrum.org, 2011.

[95] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, Sept. 1990.

[96] D. E. Simon. *An Embedded Software Primer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[97] F. Steinmetz. *Realisierung des Thermalkontrollsystems für einen Kleinsatelliten*. PhD thesis, Universität Stuttgart, 2016.

[98] C. Szyperski. *Component Software: Beyond Object-oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[99] TERMA. On-board operations software. Product Sheet rev 2, TERMA A/S Space, Jan. 2012.

[100] J.-L. Terraillon, A. Jung, P. Arberet, S. Montenegro, A. Rossignol, G. Garcia, A. I. Rodriguez, S. Mazzini, P. Hougaard, S. Fowell, et al. Space on-board software reference architecture. In *DASIA 2010 Data Systems In Aerospace*, volume 682, 2010.

[101] Tomayko, J.E. and United States. NASA. Scientific and Technical Information Division. *Computers in Spaceflight: The NASA Experience*. NASA contractor report. National Aeronautics and Space Administration, Scientific and Technical Information Division, 1988.

[102] UBM Tech. 2014 embedded market study - then, now: What's next? *EE|Live*, 2014.

[103] A. Uryu. *Development of a Multifunctional Power Supply System and an Adapted Qualification Approach for a University Small Satellite*. PhD thesis, Universität Stuttgart, 2013.

[104] T. Vardanega, G. Caspersen, and J. S. Pedersen. A case study in the reuse of on-board embedded real-time software. In *International Conference on Reliable Software Technologies*, pages 425–436. Springer, 1999.

[105] B. Wallace. A hole for every component, and every component in its hole, 2010. [http://existentialprogramming.blogspot.com/2010/05/hole-for-every-component-and-every.html; accessed July 22, 2016].

[106] S. Walz. *Nutzlast des Kleinsatelliten Flying Laptop*. PhD thesis, Universität Stuttgart, 2012.

[107] E. Webb. Ethernet for space flight applications. In *Proceedings, IEEE Aerospace Conference*, volume 4, pages 4–1927–4–1934 vol.4, 2002.

[108] Wikipedia community. Reference architecture. [http://en.wikipedia.org-/wiki/Reference_architecture; accessed Dec 06, 2017].

[109] R. Witt. *Failure Detection, Isolation and Recovery Capabilities for the University Small Satellite Flying Laptop*. PhD thesis, Universität Stuttgart, 2015.

[110] H. Wörn and U. Brinkschulte. *Echtzeitsysteme*, volume 1 of *eXamen-press*. Springer Verlag, Institut für Prozessrechentechnik, Automation und Robotik, Universität Karlsruhe (TH), 1 edition, Februar 2005.

[111] F. Zeiger, M. Schmidt, and K. Schilling. A flexible extension for pico-satellite communication based on orbit operation results of UWE-1. In *Proceedings of the 57th International Astronautical Congress*, Valencia, Spain, Oct. 2006.

[112] O. Zeile. *Entwicklung einer Simulationsumgebung und robuster Algorithmen für das Lage- und Orbitkontrollsystem der Kleinsatelliten Flying Laptop und PERSEUS*. PhD thesis, Universität Stuttgart, 2012.