

# A Case Study on using Microservice Patterns in an Embedded System

*Microservice architectures, initially a consequence of devOps organization, are nowadays the most fashionable architecture for enterprise or web scale applications. By separating functionalities into small, easily understandable, and maintainable parts, connected by a clearly defined interface, they provide several desirable characteristics. A microservice can be scaled, developed or updated independently from the others since the implementation of a service is by nature opaque. Organizing teams along services, the individual team can be small and communication overhead reduces to the absolute possible minimum, i.e., the interface. These are the same characteristics that electronic components have, esp. ICs, the building blocks of practically every electronic system in use today. It is therefore even more surprising that this architecture pattern is not used to the same extent by electronics engineers, especially in embedded systems. This paper presents a case study of dividing an embedded system in several small – and easy – components, each of them running on a small and simple system, all of which are connected by simple interfaces, in this case serial connections.*

**Keywords:** *Microservice architecture, embedded systems, architecture patterns*

## Introduction

Microservices are independently releasable services, that encapsulate a business relevant functionality and provide a well-defined interface to access it over the network (Newman, 2021). The key difference to Service Oriented Architectures (SOA) is, that the individual services are deployable separately. They run as separate processes, typically inside of a container or virtual machine. Access is typically through a REST-API. The implementation is completely language or platform agnostic. They are smaller than the services of a service-oriented architecture and used as building blocks to construct applications. Microservices are a child of cloud centric application development where the distribution, deployment and scalability of services is key. To allow that, it is important, that services are self-contained, that means especially, don't use shared state, often in form of a shared database, and be - as much as possible - independent of other services. That means, that microservices can be replaced or replicated without affecting other microservices in the system (Sommerville, 2021). Achieving this goal seems to be easier, when a microservice is related to a single business function (ibid). Obviously, there is a strong relation between microservices and Domain Driven Design, a method described in (Evans, 2003). Especially the bounded contexts of DDD do form a natural boundary for microservices.

Of course, there are ideas and publications about using microservices in embedded systems, see for example (Reglin, 2020) but most of them deal with the case of running many or all services on a single computer, often even in the

1 same image. This would be considered a monolith in normal systems. Other  
2 authors like (Rabault, 2022) go farther and use the same technology  
3 (containers, standard interface protocols like REST etc.) in embedded systems,  
4 because the problems in embedded software engineering are very similar to  
5 those of the rest of the world: moving targets require flexibility and redesign of  
6 the system. With embedded systems that often means changing hardware and  
7 starting over from scratch. While it is possible to transfer the ideas and  
8 technologies of microservice architectures to embedded systems at least for  
9 some domains, see for example (Dobaj, 2018) about industrial internet of  
10 things, in many applications the complexity and the resources needed for using  
11 these techniques are not available because of cost, size or energy constraints.

12 Even in those cases, where these resources are available, the distributed  
13 and independent nature of microservices brings additional complexity with it  
14 such as complex deployment processes, complex error detection and handling,  
15 complex debugging, communications failures, requiring increased monitoring,  
16 due to unexpected performance problems etc. Of course, all these problems can  
17 be solved, see for example (Gartziandia, 2021), since they apply to every use of  
18 microservices. But it must be considered, if the gain in flexibility is worth the  
19 price of complexity, especially in embedded systems where the flexibility is  
20 often limited by the hardware. We do not attempt to solve this problem in  
21 general, our contribution is to provide some insights from a real-life project  
22 using modest hardware.

23 There are other attempts like luos<sup>1</sup> running on ESP32 MCUs (luos.io)  
24 which try to implement something like services but without the overhead,  
25 which are interesting to follow in their progress.

26 An alternative to implement something like services is to use a real-time  
27 operating system like Free RTOS or ROS running several processes that  
28 communicate via standard inter-process communication techniques. This is the  
29 classic attempt to integrate different functions in an embedded system. From an  
30 architectural point of view, this is a conventional monolithic implementation of  
31 real-time requirements. It allows to run different tasks on the same hardware.  
32 The price is the (sometimes hidden) complexity of concurrent programming  
33 with limited resources and critical sections and paths.

34 We will use a different approach based on first principles of software  
35 engineering and keep things as simple as possible.

36 In this case study we will describe a project, that experienced exactly the  
37 challenges mentioned above. The case-study is a small sized prototypical  
38 example. The total effort is split equally between mechanical engineering and  
39 software development. By using ideas from the microservice world, we could  
40 avoid restarting from scratch even when faced with major changes in the  
41 requirements. Instead of using REST, containers, Kubernetes and all the other  
42 attractive current solutions, our approach relies on classic approaches of  
43 electrical and software engineering and simple hardware components. Clearly

---

<sup>1</sup><https://www.luos.io/>

1 defined interfaces help to maintain lower complexity while tackling the profits  
2 described above.

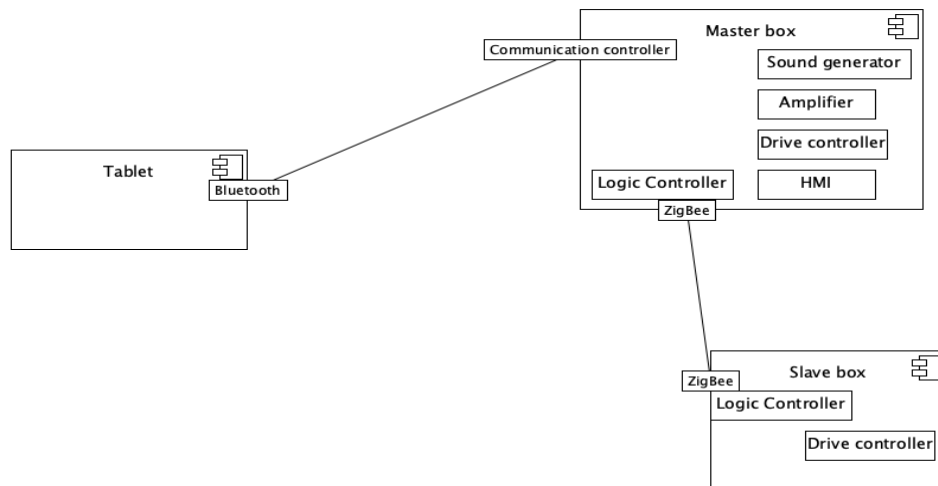
3 In the next section we will describe the project on a high-level view and  
4 derive our architecture from general best practices of software engineering. By  
5 analyzing the problems (and their solutions) that occur during development we  
6 will highlight the key findings of this case study.

## 9 Background

11 The project covered in this paper was to build the control system for a  
12 high-end audio speaker with electromechanical components (drives, sensors  
13 etc.), controlled by an App with added measurement functionality for  
14 automatic room adaption. The technical details were unclear at the beginning  
15 of the project since it was a research project started in 2019. The system was  
16 designed for a life span of more than 10 years and to be built in small  
17 quantities, typical for the price tag of more than hundred thousand Euro.

18 The structure of the system is relatively simple and shown in Figure 1.

20 **Figure 1.** *Components of the system*



21  
22 The user controls the system either via an App running on a tablet or  
23 smartphone that sends commands to the master box via Bluetooth. The master  
24 box communicates with the slave box via ZigBee.

25 The development was done by a small team of 4 persons for the hard- and  
26 software, and 1 person for mechanical engineering.

27 Development was explorative and the requirements volatile. Software and  
28 Hardware (electronics and mechanics) had to be developed independently in  
29 parallel. The pandemic required individuals to work separately due to contact  
30 limitations.

31 The ideas guiding the development of the architecture were:

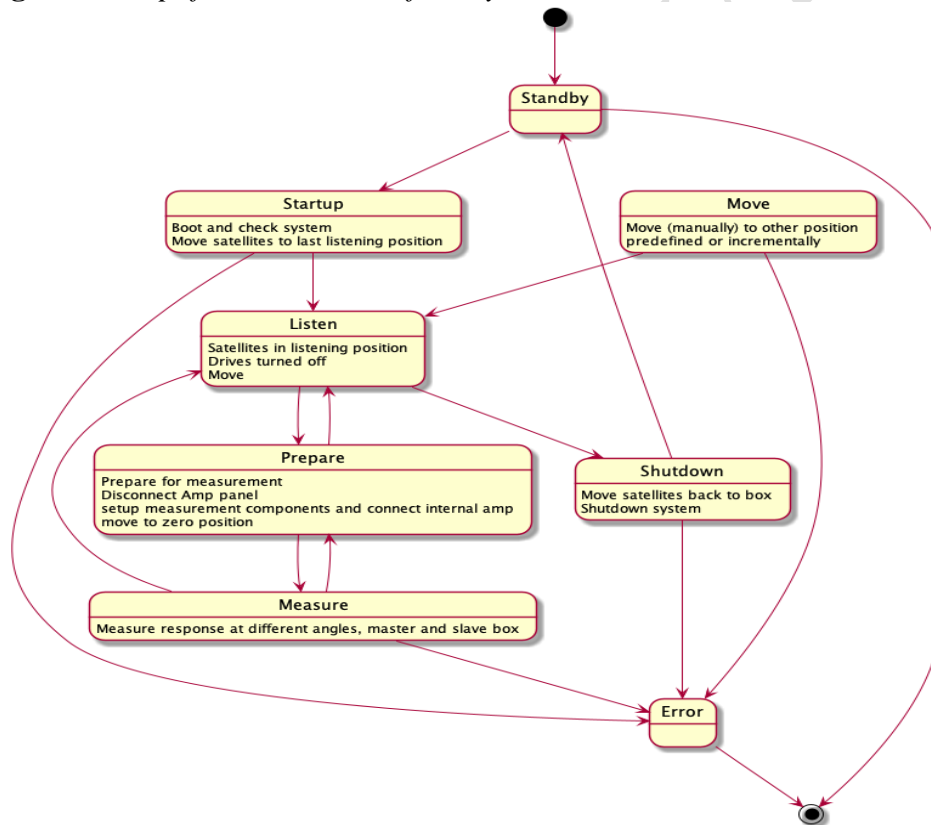
- 32  
33 - Maximize flexibility,

- 1 - loosely coupled components,
- 2 - low complexity,
- 3 - keep every important functionality on an independent, simple
- 4 controller, instead of using a complex central embedded system with a
- 5 real-time operating system,
- 6 - communication via simple (ASCII-text) messages transmitted via a
- 7 serial line (UART), and
- 8 - process control for complex actions at one, isolated point, instead of
- 9 being distributed over the system.

10  
11 The guiding metaphor was to implement “hardware objects”  
12 communicating via messages (Details see Section 3).

13 The simplified state chart of the system showing the possible flow of  
14 events is shown in Figure 2. The coordination of the activities described here is  
15 done by the “Logic Controller” component, see Figure 3.

16  
17 **Figure 2.** *Simplified state chart of the system*



18  
19  
20  
21  
22  
23  
24  
25

The components of one box are shown in Figure 3.  
Measurement mode is completely controlled from the tablet. The app is written in Java for Android.  
In measurement mode the satellites are moved to their starting position first. Then a few measurements are performed for fixed steps. Each step consists of the following tasks: 1) The test signal is played for a fixed time 2)

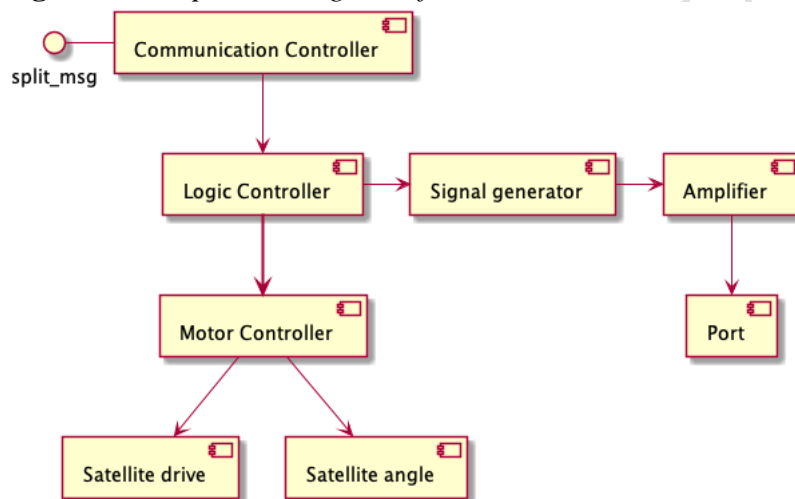
1 Audio is captured and analyzed. Steps 1 and 2 are repeated several times. If the  
2 difference between the measurements is too large, the measurement has to be  
3 repeated, due to issues with the equipment.

4 Before every measurement the system tests, if the box is disconnected  
5 from the external amplifier. If yes, the amplifier is activated, and the playback  
6 of the test signal is started. As soon as the tablet signals, that the signal was  
7 detected and the measurement has finished, the playback is stopped. If this  
8 message is not received in a predefined time, the measurement mode is  
9 terminated, the amplifier deactivated, and the error is signaled to the user.

10 At the end of the measurement cycle the sound generator and the amplifier  
11 are deactivated and the box is connected to the external amplifier port. The box  
12 is in listening mode, this is signaled to the user.

13 The tablet isn't needed for listening mode. While in listening mode, the  
14 satellites can be adjusted via the tablet in manual mode. Manual mode is  
15 activated from the App.

16  
17 **Figure 3.** Component diagram of one box



18  
19 Each box (component) in Figure 3 represents a separate controller unit.

20 The communication controller processes the incoming message and  
21 forwards them to the logic controller.

22 The logic controller interprets incoming messages and creates commands  
23 for the sound generator, the amplifier and its control, motor control and object  
24 recognition. The logic controller monitors the condition and state of the system  
25 and starts error handling if it detects problems.

26 The logic controller is designed to be as easy as possible to allow it to (re-)  
27 start as fast as possible to react to incoming messages. During listening it can  
28 be put in sleep mode to reduce EMI problems. It will be activated by the other  
29 components when needed.

30 The sound generator, amplifier control and relays are needed only in  
31 measurement mode and can be deactivated during listening or manual mode.  
32

1 Logic controllers and motor controllers are Arduino compatible teensy  
2 boards. Both motor controls are autonomous, in the sense that they receive and  
3 process commands like “move to this state”.

4 The motivation for this architecture is to need as few electronic  
5 components active while listening to music. All unnecessary components can  
6 be turned off or at least in sleep mode while listening. Ideally all digital  
7 components are turned off while hearing, but this requires a method for  
8 reactivating the control system, when the tablet should be used.

## 11 **Applying Microservice Patterns to Embedded Systems Design**

12  
13 There are many ways to use the central ideas of Microservices in  
14 embedded systems. But at the very core of those ideas is the idea of  
15 encapsulation and information-hiding. These are very old and broad ideas:  
16 Compared with the definition of UNIX-characteristics by (McIlroy, 1978)  
17 “Make each program do one thing well. To do a new job, build afresh rather  
18 than complicate old programs by adding new "features.", the parallels to the  
19 Open Closed Principle and the characteristics of microservices are obvious.

20 Typical characteristics of Microservices (Fowler, 2014):

- 21
- 22 1. Communicating with lightweight mechanisms
- 23 2. Using services as components, explicit interfaces
- 24 3. Smart endpoints and dumb pipes (esp. no ESB)
- 25 4. Running in its own process
- 26 5. Built around business capabilities
- 27 6. Design for failure
- 28 7. Evolutionary Design
- 29 8. May be written in different programming languages
- 30 9. Use different data storage technologies
- 31 10. Independently deployable
- 32 11. Products not projects (“you built it, you run it”)
- 33 12. Decentralized Governance
- 34 13. Decentralized Data Management
- 35 14. Infrastructure Automation
- 36

37 Numbers 9-14 are organizational and/or infrastructure aspects which are  
38 either irrelevant for embedded systems or at least for this project, so we will  
39 ignore them. Number 8 is obviously nice to have and, since part of the project  
40 is an App written in Java, was a requirement for the project anyway. Since we  
41 won’t cover the development process in this case study, number 7 is beyond  
42 the scope of this case study. Number 6 is obviously an important part of  
43 functional safety in embedded systems and made up a large part of the motion  
44 control in this project but is too specific in detail to be covered here. Number 5  
45 needs some adaption, because business capabilities are not a common concept  
46 in embedded systems, but if we interpret them as functional groups, then we

1 used exactly that for structuring the components (see figure 3). We interpreted  
2 “running in its own process” (number 4) as running on its own node to reduce  
3 complexity. Numbers 1-3 and 5 (in our interpretation) are the most interesting  
4 aspects and we will cover them in the next sections in more detail.

5 Key for the successful application of services is being able to change some  
6 implementation (or operational) aspect without affecting other service or  
7 clients. This is a very old goal of software engineering regarding components  
8 independent of the implementation mechanism (for example modules, objects,  
9 or services) is used.

10 The importance of loosely coupled modules (as they typically called and  
11 implemented components back then), described by low coupling is at the  
12 foundation of systematic program design, see for example (Yourdon, 1979):  
13 “The more that we must know of module B in order to understand module A,  
14 the more closely connected A is to B. The fact that we must know something  
15 about another module is a priori evidence of some degree of interconnection  
16 even if the form of the interconnection is not known.” In fact, we don’t want to  
17 know anything about component B when using component A. A more modern  
18 view (using objects) of that is the Dependency inversion principle, the “D”-Part  
19 of SOLID (Martin, 2003).

20 The same goes for high cohesion or “intramodular functional relatedness”  
21 (Yourdon, 1979, p. 95), the more modern variant being the Single  
22 Responsibility Principle, the “S”-Part of SOLID.

23 Loose coupling and high cohesion are essential for the independence of  
24 services. The same goes for high cohesion, see (Sommerville, 2021, p.165).

25 Both can be achieved or at least supported by being dependent only on  
26 abstract interfaces and not on implementation (details). That means numbers 2  
27 and 4 of our list of characteristics follows direct from basic principles of  
28 software engineering that are well tested since over 40 years.

29 High cohesion is not immediately on our list of characteristics. But  
30 Number 5 gives at least a hint to the right direction: Deciding, what a single  
31 microservices does based on the business context in the Domain Driven Design  
32 perspective leads to high cohesion.

33 An important factor leading to the recent success of Microservices is  
34 probably late binding (extreme loose coupling), a very popular concept.

35 A general definition of “late binding” relates to fixing the value of a  
36 variable or address of a function not at compile time but at run time. A good  
37 explanation including implementation in ANSI-C is given in (Schreiner, 1994).

38 For our purpose, we interpret late binding as: The resolution of the name  
39 of service being called is done at the latest moment possible – when it is called  
40 - not before that. That gives us the flexibility to change many properties of a  
41 service at run time, like replication or sharding it or using a different  
42 implementation etc. That means, the caller doesn’t have to know much about a  
43 service: the name of the service and the interface being used. When called, a  
44 request is made, that sends something looking like a message to the callee. The  
45 nice thing (and maybe the reason, why it works so well) is, that the typical  
46 implementation of a microservice uses REST-calls via HTTP. To find the

1 service, DNS is used to resolve the address of the server given in the URL.  
2 That fits nicely to the definition of object-oriented programming by Alan Kay  
3 (Kay, 2003). Compared with the techniques used before (SOAP via whatever,  
4 XML over ESB) these protocols easily qualify as lightweight mechanisms (our  
5 number 1 of characteristics). Number 3 also supports this point.

6 We started with a list of characteristics of microservices and discussed,  
7 which of them apply to embedded systems. We found that some of them are  
8 not applicable (or relevant) in this area. Others are basically rules of solid  
9 software design. In the discussion we were not very explicit about the  
10 “lightweight communication mechanism”, we will cover that in the next  
11 section.

### 14 **Consequences for the architecture of the system under investigation**

16 At the begin of the project we discussed intensively the pros and cons of  
17 using a central powerful controller with a real time operating system  
18 (FreeRTOS or ROS in this case) as a platform. The pros are obvious, see for  
19 example (Lethaby, 2013): using a well-tested platform for scheduling and  
20 communication makes life much easier. Explicit prioritization of tasks and  
21 guaranteed response times make life much easier and therefore it is industry  
22 standard to use a RTOS in such projects. There are several smaller cons like  
23 additional complexity from an additional platform, more resources required to  
24 run the OS and the additional effort of having to maintain the platform over a  
25 long lifespan (10-20 years), which is even harder when using the open- source  
26 variants like ROS which are changing quickly. But the main point, which led to  
27 the decision not to use a RTOS is: if we separate the “business” tasks not only  
28 in separate processes but on separate nodes, we didn’t have to schedule  
29 anything, because on every node (except the App which runs on the Android  
30 OS, so it isn’t relevant here) there is only one task with a well-defined flow of  
31 events.

32 By splitting the system in independent “hardware services” (every service  
33 has its own hardware), we don’t have any parallel activities running on one  
34 node except the communication, especially receiving messages from the other  
35 node. And this problem can be solved by a simple interrupt service routine.  
36 The main reason for this being possible, is the use of smart motor drives which  
37 receive a trajectory and control the acceleration and deceleration of the drives  
38 independently. So, controlling the drives (two axes) was also implemented by a  
39 “hardware service”, which we bought as an off-the-shelf component.

40 Using a sperate controller for each hardware service might look like a  
41 waste of resources at the first glance. But considering the history of  
42 components in electronic engineering it does not seem that eccentric: In  
43 electronic engineering it is common to use components like operational  
44 amplifiers in analog circuit design or logic gates (or combinations of them like  
45 PALs or FPGAs) as building blocks of larger components. The interfaces, in  
46 this case mechanical cases, pinouts, supply voltages or logic levels are highly



1 standardized. Only this way of reusing existing designs in form of components,  
2 which are separately packed and connected in the appropriate way to realize a  
3 required function made the quick innovation cycles possible we know today.  
4 Even though that approach uses more resources (transistors in this case), the  
5 gain in productivity and reliability is worth it.

6 So, the first important decision was to use hardware services: one service  
7 per device. That materializes independent services, built around business  
8 capabilities, platform independence and explicit interfaces (together with the  
9 communication mechanism discussed next).

10 The second question was, how the services should communicate,  
11 “communicating with lightweight mechanisms” being number one on our list  
12 of characteristics of microservices. There are several possibilities, all of them  
13 commonly used. Starting with the standard techniques in all computer systems,  
14 (wired) Ethernet and Wifi which are commonly used for the communication  
15 between complex systems in bigger (number and size) systems like industrial  
16 shopfloor automatization. We didn’t need the power of these and want to avoid  
17 the resource requirements of these computer network techniques. Communication  
18 between individual controllers with low bandwidth requirements in industrial  
19 (near) real time systems is often implemented with CAN (Controller Area  
20 Network), for example in automotive applications. We used that to  
21 communicate with the drive controls.

22 Board level communication is typically done via serial protocols like SPI  
23 (Serial Peripheral Interface) for higher bandwidth requirements (for example  
24 for DAQ-systems), I2C for lower bandwidth.

25 We didn’t need the performance of SPI and I2C is not good at transmitting  
26 variable length messages, so we took the easiest approach: simple ASCII-text  
27 messages for communication between all components using UART (Universal  
28 Asynchronous Receiver/Transmitter). It requires only two wires and is  
29 available on all microcontrollers. Additionally, the data paths are fixed in this  
30 point-to-point topology, so we don’t even need addressing.

31 All messages are of the form <command> {parameter}. Typically, the  
32 commands have only one parameter, in some cases two or three but not more.  
33 So, the individual messages are short (commands are one or two characters  
34 each). That means, buffering messages does not require extensive memory in  
35 case a controller cannot process a message immediately.

36 An important advantage of using UART-communication is, that it can be  
37 implemented transparently over many other protocols like Bluetooth, ZigBee  
38 or USB, all of them we used in the project.

39 The second important decision was to implement communication via  
40 messages, transmitted via simple protocol (ASCII over UART).

## 41 42 **Conclusions**

43  
44 We traced back our architecture to solid principles of software architecture  
45 and drew the connection to the characteristics of microservices. Transferring  
46 these ideas to embedded systems is possible even without sticking to the

1 technical implementation (and the overhead), microservice architectures  
2 typically use.

3 The independent, parallel development of the individual components  
4 (App, process control, motor control) with mockups for the missing parts  
5 worked well. Using messages transmitted via serial lines added some  
6 complexity (buffering to avoid losing messages) but allowed very simple  
7 debugging using just terminal emulation and/or simple scripts. Redesigning the  
8 complete external communication from Bluetooth/ZigBee to Wifi and the  
9 topology from master/slave to fully connected proved to be very simple (only a  
10 few days of work).

### 11 12 13 **References**

- 14  
15 Dobaj, Jürgen, Iber, Johannes, Krisper, Michael, Kreiner, Christian, A Microservice  
16 Architecture for the Industrial Internet-Of-Things, Proceedings of the 23rd  
17 European Conference on Pattern Languages of Programs (EuroPLOP '18).  
18 Association for Computing Machinery, New York, 2018
- 19 Evans, Eric, Domain Driven Design: Tackling Complexity in the Heart of Software,  
20 Addison Wesley, 2003, p. 161
- 21 Fowler, Martin, Microservices, 2014, Available online [https://martinfowler.com/artic  
22 les/microservices.html](https://martinfowler.com/articles/microservices.html)
- 23 Gartziaandia, Aitor, Microservice-based performance problem detection in cyber-  
24 physical system software updates, Proceedings of the 43rd International  
25 Conference on Software Engineering: Companion Proceedings (ICSE '21). IEEE  
26 Press, 147–149, 2021.
- 27 Kay, Alan, Mail exchange about what OOP is, available online at [https://www.purl.  
28 org/stefan\\_ram/pub/doc\\_kay\\_oop\\_en](https://www.purl.org/stefan_ram/pub/doc_kay_oop_en)", 2003
- 29 Lethaby, Nick, Why Use a Real Time Operating System in MCU Applications, Texas  
30 Instruments Whitepaper, 2013, available online [https://www.ti.com/lit/wp/spr  
31 y238/spry238.pdf?ts=1681631181658](https://www.ti.com/lit/wp/spry238/spry238.pdf?ts=1681631181658)
- 32 Martin, Robert C, Agile Software Development, Principles, Patterns and Practices,  
33 Prentice Hall, 2003
- 34 McIlroy, M. D., Pinson, E. N., Tague, B. A., Unix Time-Sharing System: Foreword".  
35 The Bell System Technical Journal Vol. 57, No. 6, Bell Laboratories, 1978
- 36 Newman, Sam, Building Microservices, O'Reilly, 2021, p. 1
- 37 Rabault, Nicolas, How microservices enhance agility in embedded systems  
38 development, Embedded.com, available online [https://www.embedded.com/how-  
39 microservices-enhance-agility-in-embedded-systems-development/](https://www.embedded.com/how-microservices-enhance-agility-in-embedded-systems-development/), 2022
- 40 Reglin, Frank, Verteilte Embedded Systems mit leichtgewichtigen Microservices  
41 realisieren, Elektroniki Praxis, available [https://www.embedded-software-  
42 engineering.de/verteilte-embedded-systems-mit-leichtgewichtigen-microservices-  
43 realisieren-a-165e68421d1b0bda5c9e5f33ac1a7c42/](https://www.embedded-software-engineering.de/verteilte-embedded-systems-mit-leichtgewichtigen-microservices-realisieren-a-165e68421d1b0bda5c9e5f33ac1a7c42/), 2020
- 44 Schreiner, Axel-Tobias, Object-Oriented Programming With ANSI-C, Hanser, 1994,  
45 p.15
- 46 Sommerville, Ian, Engineering Software Products, Pearson, 2021.
- 47 Yourdan, Edward, Structured Design: Fundamentals of a Discipline of Computer  
48 Program and Systems Design, Prentice Hall, 1979, p. 76
- 49