

Chapter 1

Introduction

Abstract Modern cars have evolved from mechanical devices into distributed cyber-physical systems which rely on software to function correctly. Starting from the 1970s the amount of electronics and software used has gradually increased from as little as one computer (Electronic Control Unit, ECU) to as much as 150 ECUs in 2015. The trend in the architecture, however, changes as companies look for ways to decrease the number of central computing nodes and connect them with the increased number of I/O nodes. In this chapter we provide an overview of the book and the conventions used in it and introduce the examples which we will use throughout. We describe the history of the automotive software anchoring the events in the evolution of the market of the electronics and software in modern cars. Towards the end of the chapter we also describe which directions can be pursued to deepen the knowledge of automotive software.

1.1 Software and Modern Cars

The introduction of software to cars opened up plenty of opportunities—from the optimization of cars' performance and to exciting infotainment features. Modern cars are full of electronics and the consumers are looking for car platforms which fully resemble software products. A good example of this kind of car is Tesla, which is known for innovations driven by software. The manufacturer is known for constantly pushing new versions of software to customers, providing them with new, exciting features almost every day.

The software intensive systems in modern cars provide plenty of new opportunities, but they also require more careful design, implementation, verification and validation before they can be released to users. And although the practices of software engineering include methods and tools able to fulfill the needs for safety and reliability of the automotive software, they must be applied in an automotive-specific manner to address these needs.

We could see the clear development of the automotive industry into a field less dominated by mechanical engineering but with a growing component of electronic and software engineering. We have seen the evolution of software from simple engine control algorithms of the 1970s to the advanced safety systems of the 2000s and the advanced connectivity of the 2010s. We can observe that the trends of using

the software is not going to decrease, but will increase and the amount of software used will continue to increase.

With the growing amount and importance of software in modern cars we can observe the increased need for professional software engineering. Rigorous processes of software engineering lead to higher quality software with complexity not higher than necessary and assuring that the software does not contribute to fatalities in the traffic conditions.

One of the practices of software engineering is the high-level design of software systems, also referred to as *software architecture*. The architecture of the software provides the designers with the possibility to prescribe how the software functions are distributed to software components and how the components are to interact with each other. Software architecting is usually done at the early stages of software development and serves as the basis for the allocation of software modules to components and the distribution (called *systemization*) of the functions to software components.

1.2 History of Software in the Automotive Industry

Although today it is a given that there is a lot of software in our cars, it was not like that at the beginning of the automotive industry. The first cars did not contain any electronics, which only entered the automotive market during the 1970s with the introduction of electronic fuel injection as a response to the demand for fuel efficiency [CC11].

In the 1970s the software in the cars was usually embedded deeply in the electronics in functions related to single domains—e.g., electronic fuel injection in the powertrain, electronic ignition in the electrical system or central locking. Since the use of electronics was scarce in that decade, the notion of functional safety did not relate to software and it was relatively easy to embed mechanisms for controlling the safety of the functions. The architectures of the software were usually monoliths which were not communicating with other parts of the software.

It was the 1980s that brought in such innovations as the central computers which could display basic telemetry of the vehicles—such as current fuel consumption, average fuel consumption and distance travelled. The ability to display the information to the drivers opened up new possibilities. On the embedded software front, software algorithms controlled new functions such as anti-lock brakes (ABS) and even electronic gearboxes.

The 1990s introduced even more consumer-visible electronics. The most notable innovation was in the infotainment domain and was the navigation system—or as it is commonly called, the GPS. Visualizing the information online required integration of important electronic components such as powertrain control computer, the dedicated GPS receiver and the infotainment display. The same decade introduced also more electronics and software in safety-critical areas such as ACC

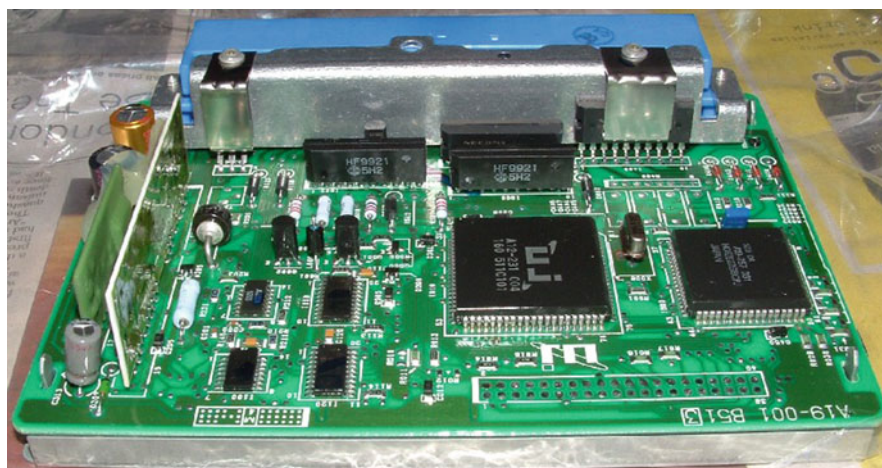


Fig. 1.1 Late 1990s JECS LH-Jetronic ECU for engine control

(Adaptive Cruise Control) which controlled the speed of a vehicle based on the speed of the vehicles in front. The introduction of this kind of functionality raised the important questions of liability for accidents caused by malfunctioning of software. The automotive software architecture used in the 1990s was more distributed and software became often recognized as important factor in innovation in the car industry. An example computer system is presented in Fig. 1.1.¹

This kind of development continued into the 2000s, when software started to dominate innovation in the car industry. It was also during the 2000s that the notion of advanced driver support systems was coined. The “advanced” referred to functions which integrated multiple computers in the car and made more “difficult” decisions for the driver. One of the most notable systems in this area was the City Safety system introduced by Volvo in its XC60 model [Ern13]. The system could stop the car from of speed under 50 kph when an obstacle appeared in front of it and the driver had no time to react. It was these kinds of systems that required more control over the complex interactions and prioritizations and therefore led to more advanced software architectures. The AUTOSAR standard was introduced to provide the possibility to communize solutions (where possible) and make it easy to change hardware platform with limited effort to adopt the software, and to enable easier sharing of the components between manufacturers and introduce a common “operating system” for the car’s computers [Dur15, DSTH14].

¹Author: RB30DE via Wikipedia <https://en.wikipedia.org/wiki/JECS>, under the Creative Commons License: <http://creativecommons.org/licenses/by-sa/3.0/>.



Fig. 1.2 2014 Audi TT infotainment unit

Finally, the 2010s introduced a completely new way of designing the electronics in cars [SLO10, RSB⁺13]. Departing from the distributed network of computers in a single car, this decade introduced the concepts of wireless cars, car-2-car communication, car-2-infrastructure communication and autonomous driving concepts. Many new actors appeared on the market where the car was no longer a final product, but a platform where new functions could be deployed even post-production. Examples of such cars are Tesla cars or Google's self-driving vehicle [Mar10]. It was also this decade that required more advanced control over the execution of software coming from different vendors for the possibility of adding new functionality to cars without the need for physically modifying the cars. An example of a focus area—infotainment—is presented in Fig. 1.2.²

Another example is the infotainment unit of Volvo XC90 as presented in Fig. 1.3.

In today's cars the size of the software grows to over 100 million lines of code according to Viswanathan [Vis15].

²Author: Audi, available at <https://en.wikipedia.org/wiki/JECS>, under the Creative Commons License: <http://creativecommons.org/licenses/by-sa/2.0/>.



Fig. 1.3 2016 Volvo XC90 infotainment unit

1.3 Trends Shaping Automotive Software Development

In 2007, Pretschner et al. [PBKS07] outlined the major trends in software development in automotive systems. This work has been a trendsetter since then and has foreshadowed the large increase in the amount of automotive software—in 2007 measured in megabytes and in 2016 measured in gigabytes. The five trends of automotive software systems presented by Pretschner et al. are:

- **Heterogeneity of software**—the software in modern cars realizes different functions in different domains. These domains range from highly safety-critical (e.g. active safety) to user experience-centered (e.g. infotainment). This means that the ways of specifying, designing, implementing and verifying the software vary among domains.
- **Distribution of labor**—the development of the software systems is often distributed between automotive OEMs (Original Equipment Manufacturers, like Volvo, BMW, and Audi) and suppliers. Suppliers are also often given an option to define their own way of working as long as they comply with the requirements of and contracts with the OEMs.
- **Distribution of software**—the automotive software system comprises a number of ECUs, and each of the computers has its own software which needs to cooperate with other ECUs to fulfill its functions. This entails more difficulty in coordination of the software and introduces more complexity.

- Variants and configurations—the globalized and highly competitive automotive market requires customizations of the same car based on the requirements of the country and the user. This means that the software in modern cars needs to be able to work in different countries without the need for recertification and, therefore the software needs to handle variants in multiple ways—both in the source code and also at runtime.
- Unit-based cost models—the competitive market means that the unit price of the car cannot be too high compared to the competition and therefore it is often the case that automotive OEMs optimize the hardware and software in such a way that unit costs remains low while the development costs can be higher.

A lot has happened since 2007 and the major trends in the automotive market today can be complemented with such trends as³:

- Connectivity and cooperation [BWKC16]—the ability to use internet functions through mobile networks enabled cars to connect to each other and/or to use information from the infrastructure to make decisions. Research projects in the area of intelligent transport systems explore such ideas as planning of the speed of a bus to minimize the need for braking for “red” when approaching intersections. The modern cars are expected to be able to connect to smartphones via bluetooth and to use internet features such as web browsers or music services.
- Autonomous functions [LKM13]—the ability of the car to brake, steer and autonomously take over from drivers entails a large amount of complexity in safety-critical systems, but is seen as “the next big thing” in the automotive sector. This also means that the verification and validation methods for software in cars will become even more stringent and even more advanced.

Autonomous driving scenarios are challenging because of the need to have an accurate and exact model of the physical surroundings of the car. This demand for the accuracy requires more sophisticated measurement equipment and therefore more data to process, more decision points, and in turn more complex algorithms. One piece of such measurement equipment which is used in autonomous driving is LIDAR, shown in Fig. 1.4.⁴

Figure 1.4 shows a LIDAR mounted on the roof of an autonomous car. The device provides a 360-degree view of the surroundings and allows the car’s software to find objects in the vicinity of the car. A LIDAR is often a complement to a RADAR, which is usually placed in the front of the vehicle. Figure 1.5 shows the picture of the radar ECU of a Volvo FH16 truck.

The production cars, however, do not have LIDARs yet, but take advantage of cameras placed in covered places. In Fig. 1.6 we can see the front camera of a Volvo XC90.

³Based on author’s own observations.

⁴Author: Steve Jurvetson; available at flickr.com, under the Creative Commons License: <http://creativecommons.org/licenses/by/2.0/>.



Fig. 1.4 Velodyne High-Def LIDAR



Fig. 1.5 Radar ECU in Volvo FH16 truck

It is interesting to observe the automotive software market today, and therefore we believe that this book will be of use to anyone who is interested in starting to get into automotive software engineering.



Fig. 1.6 Front camera in Volvo XC90

1.4 Organization of Automotive Software Systems

Over the years each car manufacturer (often referred to as an OEM, Original Equipment Manufacturer) developed its own way of organizing software systems with the diversity in pair of the diversity of car brands today. However, many of the car manufacturers design the software in a similar way—they use the V development model and a similar organization of the electrical (and software) systems into domains and subsystems. We can depict it in the model presented in Fig. 1.7.

In this view we can see that the electrical system is organized into domains, such as infotainment and powertrain. Each of these domains has a specific set of properties—some are safety-critical and some not, some are very user oriented and some are realtime and embedded. Each of these domains, however, is organized into subsystems which group a specific functionality (some OEMs call these subsystems simply “systems”) such as active safety, and advanced driver support and similar. These systems group a number of logical elements and realize the functionality, which is often grouped into functions. The functions are often called end-to-end functions, as they realize user functionality such as Adaptive Cruise Control, Line Departure Warning and Navigation from A to B.

The functions are realized by subsystems of the electrical system and they are orthogonal to the organization of subsystems, components and modules. Therefore we often see the concept of “functional architecture (view)”—describing the dependencies among functions.

Each subsystem contains a number of components which include smaller parts of software elements that realize parts of the functionality (e.g. such a part

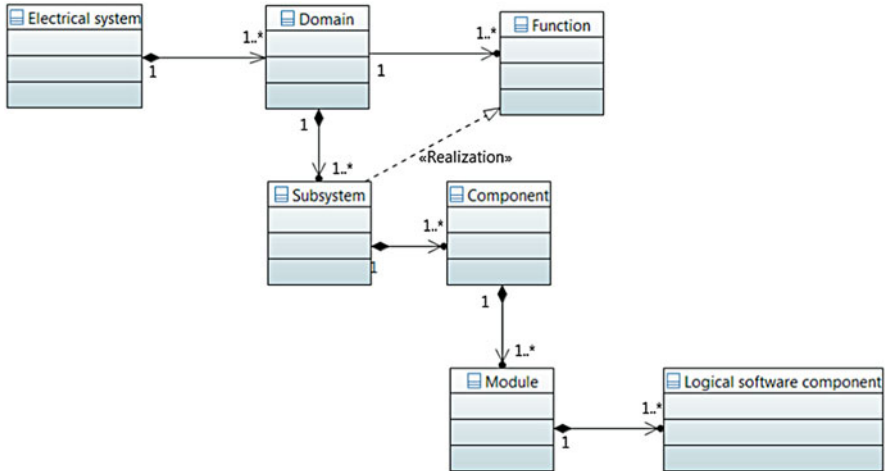


Fig. 1.7 Conceptual view of the organization of the software system

could be a message broker for an infotainment system). These components are organized into software modules, which are often source code files with a set of classes, methods and programming language functions. The groupings of these programming language functions or software classes are referred to as logical software components.

The term software architecture can be used in almost all levels of this hierarchy (except for the lowest one). We can talk about the EE architecture (Electrical System architecture) which describes the organization of software and hardware for the entire car. We can talk about an ECU architecture which describes the logical organization of software subsystems, components and modules in the ECU. Depending on the size and role of the ECU we could have modules, components or subsystems in the ECU [DNSH13].

The methods and techniques presented in this book can be applied at any of these levels.

1.5 Architecting as a Discipline

Software architecture is a kind of artifact in software development, but architecting is a full-fledged discipline with its own activities and tasks. It is quite often the case that software architects are perceived as more experienced than senior designers and are given a larger mandate to make decisions than software designers. In order to prevent confusion, let us briefly discuss the role of software architects in contrast to the designers and project managers. These two roles can be perceived as overlapping to some extent and therefore this comparison gets interesting.

1.5.1 Architecting vs. Project Management

Being a software architect means being in a role of a kind of technology leadership. The architects are the persons who lay the ground for the development of the entire system—in terms of general architectural styles, but also in terms of principles which guide the development of the system. Those principles form the boundaries within which the designers can make their choices. It is the role of the architect to ensure that these principles are followed during the entire lifecycle of the software system.

In some sense, setting the frames for the system design is a technical correspondent to setting the frames for the cost and scope of the project that develops the system. However, it is the responsibility of the project manager to set and monitor this project scope, schedule and cost. Therefore we contrast architecting as a technical correspondent to project management in Table 1.1.

Since the discipline of architecting is practices by technical experts, it is technical principles that are applied—how to create objects, send messages, deploy components onto ECUs. This means that the technologies and their characteristics are in focus. For example, the architects need to balance different quality characteristics with each other—performance vs. safety, maintainability vs. portability and others. Therefore the architects also focus on the quality and functionality—addressing such challenges as “how to enable video feeds over the Flexray network without adding new cables”. Finally the architects focus on the functionality and make sure that the electrical system of the car can realize the functionality given the constraints (e.g. weight of the cables, number of ECUs). All of these aspects make software architecting seem as technical product management.

In contrast to the technical management, we have project management, where the project leaders apply organizational theories to determine whether to work Agile or waterfall, or how to negotiate contracts, or how to measure the progress of the project. When applying the managerial and organizational theories the project leaders focus on the scope of the project—addressing the questions of whether a given functionality can be developed given the budget constraints of the project. The focus of the project leaders is on resources, on balancing cost and resources with the schedule of the project. All of these aspects can be seen as management of the project rather than management of the product.

Table 1.1 Architecting vs. project management

Architecting	Project management
Done by technical experts	Done by management experts
Technology in focus	Scope in focus
Focus on quality	Focus on cost
Focus on requirements	Focus on work products
Focus on solution	Focus on resources
Maximize functionality	Minimize cost

Table 1.2 Architecting vs. designing

Architecting	Designing
Making rules and decisions	Following rules and decisions
High level structures	Low-level structures
Holistic understanding	Specialistic understanding
Systems thinking	Software thinking
Documentation-oriented	Code and executable/detailed model-oriented
Modelling and analysis	Execution and testing

Both technical and project management need to work with one another as they develop the one and the same product! Humphrey [Hum96] in his book “Managing Technical People: Innovation, Teamwork and the Technical process” provides a number of useful guidelines on how to combine these two.

1.5.2 Architecting vs. Design

Similarly to contrasting the discipline of architecting to the discipline of project management, we can also contrast architecting to designing. We could observe from the previous contrast that technical product management is about setting principles for the work. The discipline of designing is all about following these principles in order to arrive at final software product. We present some of the differences in Table 1.2.

Software architecting, being the technical management of the system, sets the boundaries for the design in terms of principles, rules and decisions about how to design the system. An example of such a decision is the choice of the communication protocol between the ECUs and the number of ECUs in the system. It’s also about which standards to follow and why. Architecting, as we will see in this book, is a discipline operating at a high abstraction level—considering components (e.g. groups of software classes) and execution nodes. This requires a holistic understanding of the system—both the software and the underlying hardware used to execute the software or provide the software with data. This kind of a “systems thinking” makes the architects the core part of any software team because they understand the background of “why” things happen rather than just do things.⁵

The discipline of architecting is also very documentation-oriented—as the decisions, rules and principles need to be communicated, they also need to be explained and documented to lead to consistency and enforcement of rules. This happens often as a process of analysis and modelling of the system.

⁵Sinek in his book “Starting with Why: How Great Leaders Inspire Everyone to Action” [Sin11] presents a set of examples of how this works in practice.

In contrast, the discipline of designing is focused on realizing the principles, decisions and rules of the architecture in software code or an executable model. The high-level structure discussed in the architecture is now developed using lower-level structures—components using classes and blocks, ECUs using execution processes. This requires specialized knowledge and competence in the particular domain in question (e.g. the infotainment or powertrain). The design is focused on the software entities and their interaction with the underlying hardware, where the hardware is often given (or at least the specification of the hardware is given during the design of the software). This means that designing is focused on the code and executable/detailed models rather than on abstract analysis and modelling. It is also therefore the design that is the first activity where we discuss testing and execution, whereas in the architecture we talk about assessments and evaluations (a topic which we will return to in Chap. 6).

Similarly to the collaboration between the architects and the project managers, the architects need to collaborate closely with the designers in order to develop and deliver a software system which fulfills all the requirements and quality constraints.

1.6 Content of This Book

This book addresses one of the most fundamental aspects of engineering of software systems—software architectures. The architecture is a high-level design of a software system which enables the architects to distribute the functionality of the software system to multiple interacting components. The components are usually grouped into subsystems and domains which address a set of functional and non-functional requirements of the software system.

In this book we explore the concept of software architecture for modern cars which is intended for both novice and advanced software designers. This book is intended for two groups of audience—professionals working with automotive software who need to understand concepts related to automotive architectures, and students of software engineering or related programs who need to understand the specifics of automotive software to be able to construct cars or their components.

The idea to support the professionals came from the author's observations that the automotive industry requires an individual software engineer to be able to understand a variety of disciplines. Individuals working with the construction of car software or hardware need to understand their counterparts in order to be able to design safe, reliable and long-term solutions for the car industry. Software engineers need to understand how their software is to be integrated with other software from other vendors in order to be able to develop user functions, e.g. collision avoidance by braking.

The idea to support the students came from the observation that many of the graduates from software engineering programs require further education in order to understand such advanced concepts as software and systems safety, working with suppliers and distribution of software. During the author's years of working

with students it became evident that it is difficult to provide education in software engineering in general and also focus on specific aspects such as automotive software. This book addresses this challenge and is aimed at being both a reference book and a potential course book for software engineering programs.

This book is structured into independent chapters which can be read separately, although we recommend reading them in sequence. Reading the chapters in sequence allows us to follow the motivating example throughout the book and to gradually build up knowledge about automotive software architectures.

1.6.1 Chapter 2: Software Architectures

In this chapter we present the basics of software architecture in general as a recap for readers who are not familiar with architecting as a discipline, and towards the end of the chapter we describe the specificity of automotive software architectures.

In the beginning of the chapter we review the definitions of software architectures, define the types of view used in automotive software design and relate them to the architectural views in software engineering in general—the 4+1 architecture view model.

We gradually progress in the chapter to introduce elements important for automotive architectures, e.g., ECUs (Electronic Control Units), logical and physical components, functional architectures, and topologies for automotive architectures (physical and logical). We delve into the peculiarities of automotive software—embedded systems with large focus on safety and dependability.

1.6.2 Chapter 3: Automotive Software Development

In this chapter we describe and elaborate on software development processes in the automotive industry. We introduce the V-model for the entire vehicle development and we continue to introduce modern agile software development methods for describing the ways of working of software development teams. We also provide an overview of a tool which is used to keep the design data consistent—SystemWeaver by SystemIt.

In this chapter we discuss the specifics of automotive software development such as variant management, different integration stages, testing strategies and the methods used for these. We review methods used in practice and explain how they should be used.

1.6.3 Chapter 4: AUTOSAR Reference Model

In this chapter we continue on the topic of standardization and we discuss the current standardization efforts. We describe and discuss the AUTOSAR standard, which gets the most attention today in Europe and worldwide.

In the AUTOSAR standard we describe the main building blocks like software components and communication buses. We also describe the evolution of the standard from the perspective of the main concepts and their influence on the car industry.

Towards the end of the chapter we present the AUTOSAR reference architecture as described in the standard and discuss its evolution.

1.6.4 Chapter 5: Detailed Design of Automotive Software

In this chapter we continue to delve into the technical aspects of automotive software architectures and we describe ways of working when designing software within particular software components. We present the methods for modelling the functions using Simulink modelling and we show how these methods are used in the automotive industry.

Towards the end of the chapter we introduce the need for quality assessment of software architectures and the challenges related to assessment of the sub-characteristics of quality (the so-called “-ilities”).

1.6.5 Chapter 6: Evaluation of Automotive Software Architectures

In this chapter we introduce methods for assessing the quality of software architectures and we discuss ATAM. We discuss the non-functional properties of automotive software and we review the methods used to assess such properties as dependability, robustness and reliability. We follow the ISO/IEC 25000 series of standards when discussing these properties.

In this chapter we also address the challenges related to the integration of hardware and software and the impact of this integration. We review the differences with stand-alone desktop applications and discuss examples of these differences.

Towards the end of the chapter we discuss the need to measure these properties and introduce the need for software measurement.

1.6.6 Chapter 7: Metrics for Software Design and Architectures

In this chapter we describe the most commonly used metrics in software engineering in general and in automotive software engineering, e.g. lines of code, model size, complexity, and architectural stability or coupling [SHFMHNNH13]. In particular we present these metrics and their interpretation (what should be done, and why, based on the values of metrics). We discuss the use of metrics based on the international standard ISO/IEC 15939.

1.6.7 Chapter 8: Functional Safety of Automotive Software

In this chapter we elaborate on one of the most important issues related to software in modern cars—functional safety. We explore the safety-related concepts described in the international standard ISO/IEC 26262 and we describe how this standard is used in modern software development processes.

We explore such elements as verification and validation techniques mentioned in the standard and link them to the ASIL levels and efficiency of their applications.

In the chapter we describe how the standard is to be applied on the examples of the simple function.

1.6.8 Chapter 9: Current Trends in Automotive Software Development

We conclude the book with the outlook on the current trends in automotive software development and we introduce the emerging, disruptive technologies on the market that have the potential to change the automotive industry to become more software-oriented than it traditionally has been.

1.6.9 Motivating Examples in the Book

In this book we illustrate the concepts introduced in each chapter with a set of examples. Each chapter has its own examples which are dedicated to extrapolating the concepts described, and therefore:

- Chapter 2 contains a set of examples from different domains, e.g. infotainment, powertrain and active safety.

- Chapter 3 includes examples of requirements from AUTOSAR and requirements for opening the car from the chassi domain.
- Chapter 4 contains examples of the AUTOSAR models and their realization for communication between two ECUs.
- Chapter 5 includes examples of digitalization of an analog signal and the designing of the heating of a car's chassi from the Chassi domain.
- Chapter 6 contains examples of the parking assistance camera from the active safety domain.
- Chapter 7 contains examples of a real software (obfuscated) published as open source.
- Chapter 8 includes the example of a simple microcontroller demonstrating the different ASIL levels and architectural choices used to achieve these levels.

These examples do not constitute an entire software system of a car, as these systems are huge. As a reference, BMW in its talks at conferences showed the size of the electrical system to be about 200 ECUs, which includes all variants of its electrical system (meaning that there is no car with all 200 ECUs.⁶)

1.7 Knowledge Prerequisites

In order to understand the book one needs to understand how programming works. We do not require any specific programming skills, but it is good to know the basics of programming in C/C++ or Java/C#. It is also good to have the basic knowledge of the UML notation, especially the class diagrams.

We introduce topics from the automotive domain and we require no prior understanding of the domain nor any knowledge of software architecture.

For each chapter we provide pointers where the interested reader can find more information or where the necessary prerequisites can be obtained.

1.8 Where to Go Next

After reading this book you will be able to understand how to architect a software system for a modern car. You will also be prepared to understand the design principles guiding the development of software in modern cars and be able to understand the non-functional principles behind the design.

The next natural step is to follow your interest in the design of software systems. We recommend focusing on the principles of continuous integration and deployment, virtual verification and validation as well as advanced functional safety.

⁶Presentation from BMW at Elektronik i Fordon, Gothenburg, May 2016.

References

- BWKC16. Robert Bertini, Haizhong Wang, Tony Knudson, and Kevin Carstens. Preparing a roadmap for connected vehicle/cooperative systems deployment scenarios: Case study of the state of oregon, usa. *Transportation Research Procedia*, 15:447–458, 2016.
- CC11. Andrew YH Chong and Chee Seong Chua. *Driving Asia: As Automotive Electronic Transforms a Region*. Infineon Technologies Asia Pacific Pte Limited, 2011.
- DNSH13. Darko Durisic, Martin Nilsson, Mirosław Staron, and Jörgen Hansson. Measuring the impact of changes to the complexity and coupling properties of automotive software systems. *Journal of Systems and Software*, 86(5):1275–1293, 2013.
- DSTH14. D. Durisic, M. Staron, M. Tichy, and J. Hansson. Evolution of Long-Term Industrial Meta-Models - A Case Study of AUTOSAR. In *Euromicro Conference on Software Engineering and Advanced Applications*, pages 141–148, 2014.
- Dur15. D. Durisic. *Measuring the Evolution of Automotive Software Models and Meta-Models to Support Faster Adoption of New Architectural Features*. Gothenburg University, 2015.
- Ern13. Tomas Ernberg. Volvo’s vision 2020–‘no death, no serious injury in a volvo car’. *Auto Tech Review*, 2(5):12–13, 2013.
- Hum96. Watts S Humphrey. *Managing technical people: innovation, teamwork, and the software process*. Addison-Wesley Longman Publishing Co., Inc., 1996.
- LKM13. Jerome M Lutin, Alain L Kornhauser, and Eva Lerner-Lam MASCE. The revolutionary development of self-driving vehicles and implications for the transportation engineering profession. *Institute of Transportation Engineers. ITE Journal*, 83(7):28, 2013.
- Mar10. John Markoff. Google cars drive themselves, in traffic. *The New York Times*, 10(A1):9, 2010.
- PBKS07. Alexander Pretschner, Manfred Broy, Ingolf H Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007.
- RSB⁺13. Rakesh Rana, Mirosław Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Fredrik Törner. Increasing efficiency of iso 26262 verification and validation by combining fault injection and mutation testing with model based development. In *ICSOFT*, pages 251–257, 2013.
- SHFMHNNH13. Staron, M., Hansson, J., Feldt, R., Meding, W., Henriksson, A., Nilsson, S. and Höglund, C., 2013, October. Measuring and visualizing code stability—a case study at three companies. In *The International Conference on Software Process and Product Measurement*, (pp. 191–200). IEEE.
- Sin11. Simon Sinek. *Start with why: How great leaders inspire everyone to take action*. Penguin UK, 2011.
- SLO10. Margaret V String, Nancy G Leveson, and Brandon D Owens. Safety-driven design for software-intensive aerospace and automotive systems. *Proceedings of the IEEE*, 98(4):515–525, 2010.
- Vis15. Balaji Viswanathan. Driving into the future of automotive technology at genivi annual members meeting. *OpenSource Delivers*, online, 2015.

<http://www.springer.com/978-3-319-58609-0>

Automotive Software Architectures

An Introduction

Staron, M.

2017, XIX, 237 p. 150 illus., 107 illus. in color.,

Hardcover

ISBN: 978-3-319-58609-0