Chapter 2
Software Architectures: Views and Documentation

Abstract Software architecture is the foundation for automotive software design. Being a high-level design view of the system it combines multiple views on the software system, and provides the project teams with the possibility to communicate and make technical decisions about the organization of the functionality of the entire software system. It allows also us to understand and to predict the performance of the system before it is even designed. In this chapter we introduce the definitions related to software architectures which we will use in the reminder of the book. We discuss the views used during the process of architectural design and discuss their practical implications.

2.1 Introduction

As the amount of software in modern cars grows we observe the need to use more advanced software engineering methods and tools to handle the complexity, size and criticality of the software [Sta16, Für10]. We increase the level of automation and increase the speed of delivery of software components. We also constantly evolve software systems and their design in order to be able to keep up with the speed of the changes in requirements in automotive software projects.

Software architecture is one of the cornerstones of successful products in general, and in particular in the automotive industry. In general, the larger the system, the more difficult it is to obtain a good quality overview of its functions, subsystems, components and modules—simply because of the limitations of our human perception. In automotive software design we have more specific challenge, related to the safety of the software embedded in the car and the distribution of the software—both distribution in terms of the physical distribution of the computing nodes and distribution of the development among the car manufacturers and their suppliers.

In this chapter we discuss the concept of software architecture and explain it with the examples of building architectures. Once we know more about what constitutes software architecture, we go into the details of different views of software architecture and how they come together. We then move on to describing the most common architectural styles and explain where they can be seen in automotive software. Finally we present the ways of describing architectures—the

© Springer International Publishing AG 2017
M. Staron, *Automotive Software Architectures*,
DOI 10.1007/978-3-319-58610-6_2

architecture modelling languages. We end the chapter with references to further readings for readers interested in more details.

2.2 Common View on Architecture in General and in the Automotive Industry in Particular

The concept of architecture is well rooted in our society and its natural association is to the styles of buildings. When thinking about architecture we often recall large cathedrals, the gothic and modern styles of churches, or other large structures. One of the examples of such a cathedral is the "Sagrada Familia" cathedral in Barcelona with its very characteristic style.

However, let us discuss the concept of the architecture with a considerable smaller example—let us take the example of a pyramid. Figure 2.1[1] presents a picture of the pyramids in Gizah.

The form of the pyramid is naturally based on a triangle. The fact that it is based on a triangle is one of the architectural choices. Another choice is the type of the triangle (e.g. using the golden number 1.619 as the ratio between the slant height to half the base length). The decision is naturally based on mathematics and illustrated

Fig. 2.1 All Gizah pyramids: a picture represents an external view of the product

[1]Author: Ricardo Liberato, available at Wikipedia, under the Creative Commons License: https://creativecommons.org/licenses/by-sa/2.0/.

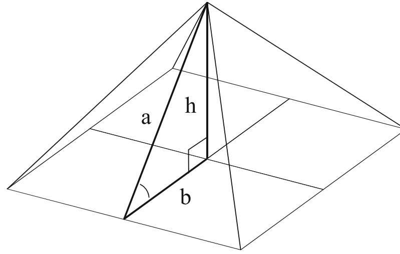

Fig. 2.2 Internal view of the architecture of a pyramid

Fig. 2.3 Volvo XC 90, another example of the external view of the product

using one of the views of the pyramid—call it an early design blueprint as presented in Fig. 2.2.

Figure 2.2 shows the first design principles later on used to detail the design of the pyramid. Instead of delving deeper into the pyramid construction, let us now consider the notion of architecture and software architecture in the automotive industry.

One obvious view of the architecture of the car is the external view of the product, as with the view of the pyramid (Fig. 2.3).[2]

[2]Author: Albin Olsson, available at Wikipedia, under the Creative Commons License: https://creativecommons.org/licenses/by-sa/4.0/deed.en.

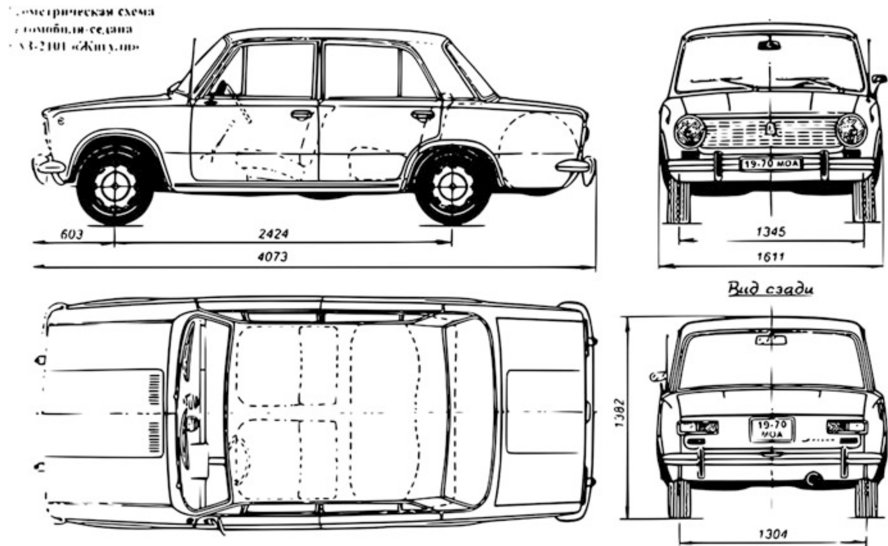

Fig. 2.4 A blueprint of the design principles of a car

We can observe the general architectural characteristics of a car—the placement of the lights, the shape of the lights, the shape of the front grill, the length of the car, etc. This view has to be complemented with a view of the internal design of the car. An example of such a blueprint is presented in Fig. 2.4.[3]

This blueprint shows the dimensions of the car, hiding other kinds of details. Another blueprint can be a view of the electrical system of a car, its electronics and its software.

2.3 Definitions

Software architecting starts with the very first requirement and ends with the last defect fix in the product, although its most intensive period is in the early design stage where the architects decide upon the high-level principles of the system design. These high-level principles are documented in the form of a software architecture document with several views included. We could therefore define the software architecture as the high-level design, but this definition would not be just. The definition which we use in this book is:

[3]Figure source: pixbay.com.

Software architecture refers to the high-level structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system

The definition is not the only one, but it reflects the right scope of the architecture. The definition comes from Wikipedia (https://en.wikipedia.org/wiki/Software_architecture).

2.4 High-Level Structures

The definition presented in this chapter ("Software architecture refers to the high-level structures of a software system...") talks about "high-level structures" as a means to generalize a number of different entities used in the architectural design. In this chapter we go into details about these structures, which are:

1. Software components/Blocks—pieces of software packaged into subsystems and components based on their logical structure. Examples of such components could be UML/C++ classes, C code modules, and XML configuration files.
2. Hardware components/Electronic Control Units—elements of design of the computer system (or platform) on which the software is executed. Examples of such elements include ECUs, communication buses, sensors and actuators.
3. Functions—elements of the logical design of the software described in terms of functionality, which is then distributed over the software components/blocks. Examples of such elements are software functions, properties and requirements.

All of these elements together form the electrical system of the car and its software system. Even though the hardware components do not "belong" to the software world, it is the often the job of the architect to make sure that they are visible and linked to the software components. This linking is important from the process perspective—it must be know which supplier should design the software for the hardware. We talk more about the concept of the supplier and the process in Chap. 3.

In the list of high-level structures, when introducing functions, we indicated the interrelation between these entities—"functions distributed over the software components". This interrelation leads us to an important principle of architecting—the use of views. An architectural view is *a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders* [RW12].

One could see the process of architecting as a prescriptive design, the process continuous as the design evolves. Certain aspects of design decisions influence the architecture and are impossible to know a priori—increased processing power required to fulfill late function requirements or safety-criticality of the designed system. If not managed correctly the architecture has a tendency to evolve into a descriptive documentation that needs to be kept consistent with the software itself [EHPL15, SGSP16].

2.5 Architectural Principles

The second part of the definition of the software architecture ("...the discipline of creating such structures...") refers to the decisions which the software architects make in order to set the scene for the development. The software architects create the principles by defining such things as what components should be included in the system, which functionality each component should have (but not how it should be implemented—this is the role of the design discipline, which we describe in Chap. 5) and how the components should communicate with each other.

Let us consider the coupling as an example of setting the principles. We can consider an example of a communication between the component representing the controller of the windshield wipers and the component representing the hardware interface to the small engine controlling the actual windshield wiper arm. We could have a coupling in one way, as presented in Fig. 2.5.

In the figure we can see that the line (association) between the blocks is directed from WindshieldWiper to WndEngHW. This means that the communication can only happen in one way—the controller can send signals to the hardware interface. This seems logical, but it raises challenges when the controller wants to know the status of the hardware interface without pulling the interface—it is not possible as the hardware interface cannot communicate with the controller. If an architect sets this principle then this has the consequences on the later design, such as the need for extra signals on the communication bus (pulling the hardware for the status).

However, the software architect might make another decision—to allow communication both ways, which is shown in Fig. 2.6.

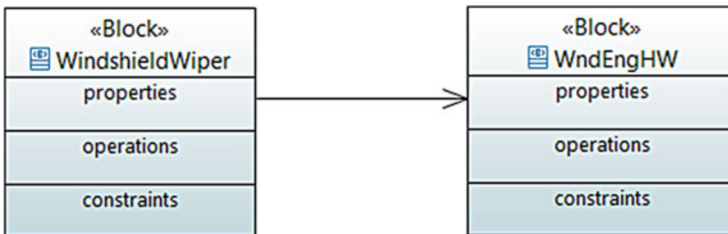

Fig. 2.5 An example principle—unidirectional coupling between two blocks

Fig. 2.6 An example principle—bidirectional coupling between two blocks

The second architectural alternative allows the communication in both ways, which solves the challenges related to pulling the hardware interface component for the status. However, it also brings in another challenge—tight coupling between the controller and the hardware interface. This tight coupling means that when one of these two component changes, the other should be changed (or at least reviewed) as the two are dependent on one another.

In the reminder of this chapter we discuss several of such principles when discussing architectural styles.

2.6 Architecture in the Development Process

In order to put the process of architecting in context and describe the current architectural views in automotive software architectures, let us first discuss the V-model as shown in Fig. 2.7. The V-model represents a high-level view of a software development process for a car from the perspective of OEMs. In the most common scenario, where there is no OEM in-house development, component design and verification is usually entirely done by the suppliers (i.e., OEMs send empty software compositions to the suppliers, who populate them with the actual software components).

The first level is the functional development level, where we encounter two types of the architectural views—the functional view and the logical system view. Now, let us look into the different architectural views, their purpose and the principles of using them. When discussing the views we also discuss the elements of these views.

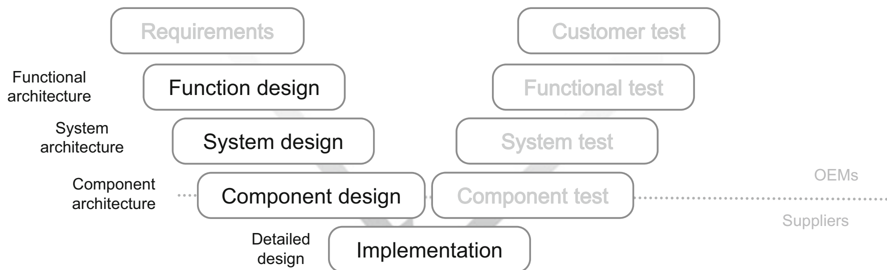

Fig. 2.7 V-model with focus on architectural views and evolution

2.7 Architectural Views

As we show in the process when starting with the development from scratch, the requirements of or ideas for functions in the car come first—the product management has the ideas about what kind of functionality the car should have. Therefore we start with this type of the view first and gradually move on to more detailed views on the design of the system.

2.7.1 Functional View

The functional view, often abbreviated to *functional architecture*, is the view where the focus is on the functions of the vehicle and their dependencies on one another [VF13]. An example of such a view is shown in Fig. 2.8.

As we can see from the example, there are three elements in this diagram—the functions (plotted as rounded-edge rectangles), the domains (plotted as sharp-edged rectangles) and the dependency relations (plotted as dashed lines), as the functions can depend on each other and they can easily be grouped into "domains" such as Powertrain and Active Safety. The usual domains are:

1. Powertrain—grouping the elements related to the powertrain of the car—engine, engine ECU, gearbox and exhaust.
2. Active Safety—grouping the elements related to safety of the car—ADAS (Advanced Driver Assistance Systems), ABS (Anti-lock Braking System) and similar.

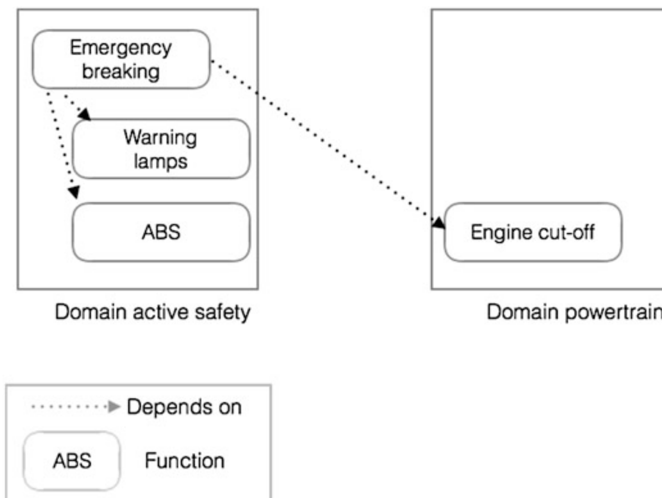

Fig. 2.8 Example of a functional architecture—or a functional view

3. Chassi and body—grouping the elements related to the interior of the car—seats, windows and other (which also contain electronics and software actuators/sensors).
4. Electronic systems—grouping the elements related to the functioning of the car's electronic system—main ECU, communication buses and related.

In modern cars the number of functions can reach more than 1000 and is constantly growing. The largest growth in the number of functions is related to new types of functionality in the cars—autonomous driving and electrification. Examples of functions from the autonomous driving area are:

1. Adaptive Cruise Control—basic function to automatically keep a distance from the preceding vehicle while maintaining a constant maximum velocity.
2. Lane Keeping Assistance—basic function to warn the driver when the vehicle is crossing the parallel line on the road without the turn indicator.
3. Active Traffic Light Assistance—medium advanced function to warn the driver of a red light ahead.
4. Traffic Jam Chauffeur—medium/advanced function to autonomously drive during traffic jam conditions.
5. Highway Chauffeur/pilot—medium/advanced function to autonomously drive during high-speed driving.
6. Platooning—advanced function to align a number of vehicles to drive autonomously in a so-called platoon.
7. Overtaking Pilot—advanced function to autonomously drive during an overtake situation.

These advanced functions build on top of the more basic functionality of the car, such as the ABS (Anti-lock Braking System), warning lights and blinkers. The basic functions that are used by the above functions can be exemplified by:

1. Anti-lock Braking System (ABS)—preventing the car from locking the brakes on slippery roads
2. Engine cut-off—shutting down the engine in situations such as after crash
3. Distance warning—warning the driver about too little distance from the vehicle in front.

The functional view provides the architects with the possibility to cluster functions, and distribute them to the right department to develop and to reason about these kinds of functionality. An example of such reasoning is the use of methods such as the Pareto front [DST15].

2.7.1.1 How-To

The process of functional architecture design starts with the development of the list of functions of the vehicle and their dependencies, which can be documented in block diagrams, use case diagrams or SysML requirements diagrams [JT13, SSBH14].

Once the list and dependencies are found, we move to organizing the functions to the domains. In the normal case these domains are known and given. The organization of the functions is based on how they are dependent on each other with the principle that the number of dependencies that cross-cut the domains should be minimized. The result of this process is the development of the diagram as shown in Fig. 2.8.

2.7.2 Physical System View

Another view is the system view on the architecture, usually portrayed as a view of the entire electrical system at the top level with accompanying lower-level diagrams (e.g. class diagrams in UML). Such an overview level is presented in Fig. 2.9. In this view we could see the ECUs (rounded rectangles) of different sizes placed on two physical buses (lines). This view of the architecture provides the possibility to present the topology of the electrical system of the car and provides the architects with a way to reason about the placement of the computers on the communication buses.

In the early days of automotive software engineering (up until the late 1990s) this view was quite simple and static as there were only a few ECUs and a few communication buses. However, in the modern software design, this view gets increased importance as the number of ECUs grows and the ability to give an overview becomes more important. The number of communication buses also increases and therefore the topologies of the components in the physical architectures have evolved from the typical star topologies (as in Fig. 2.9) to more linked architectures with over 100 active nodes. The modern view on the topology

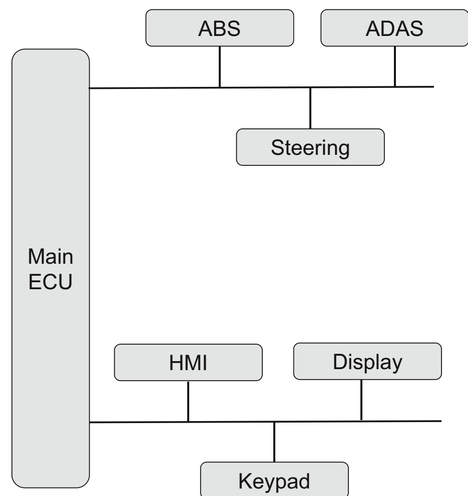

Fig. 2.9 Example of a system architecture—or a system view

also includes information about the processing power and the operating system (and its version) of each ECU.

2.7.2.1 How-To

Designing this view is usually straightforward as it is dictated by the physical architecture of the car, where the set of ECUs is often given. The most important ECUs are often predetermined from the previous projects—usually the main computer, the active safety node, the engine node, and similar. A list of the most common ECUs present in almost all modern cars is (https://en.wikipedia.org/wiki/Electronic_control_unit):

- Engine control unit (EnCU)
- Electric power steering control unit (PSCU)
- Human-machine interface (HMI)
- Powertrain control module (PCM)
- Telematic control unit (TCU)
- Transmission control unit (TCU)
- Brake control module (BCM; ABS or ESC)
- Battery management system

Depending on the car manufacturer, the other control modules can differ significantly. It is also the case that many of the additional control units are part of the electrical system, meaning that they are included only in certain car models or instances, depending on the customer order.

2.7.3 Logical View

The focus of the view is on the topology of the system. This view is often accompanied by the logical component architecture as presented in Fig. 2.10. The rationale behind the logical view of the system is to focus solely on the software of the car. In the logical view we show which classes, modules, and components are used in the car's software and how they are related to each other. The notation used for this model is often UML (Unified Modelling Language) and its subling SysML (Systems Modelling Language).

For the logical view, the architects often use a variety of diagrams (e.g. communication diagrams, class diagrams, component diagrams) to show various levels of abstraction of the software of the car—usually in its context. For the detailed design, these architectural models are complemented with low-level executable models such as Matlab/Simulink defining the behaviour of the software [Fri06].

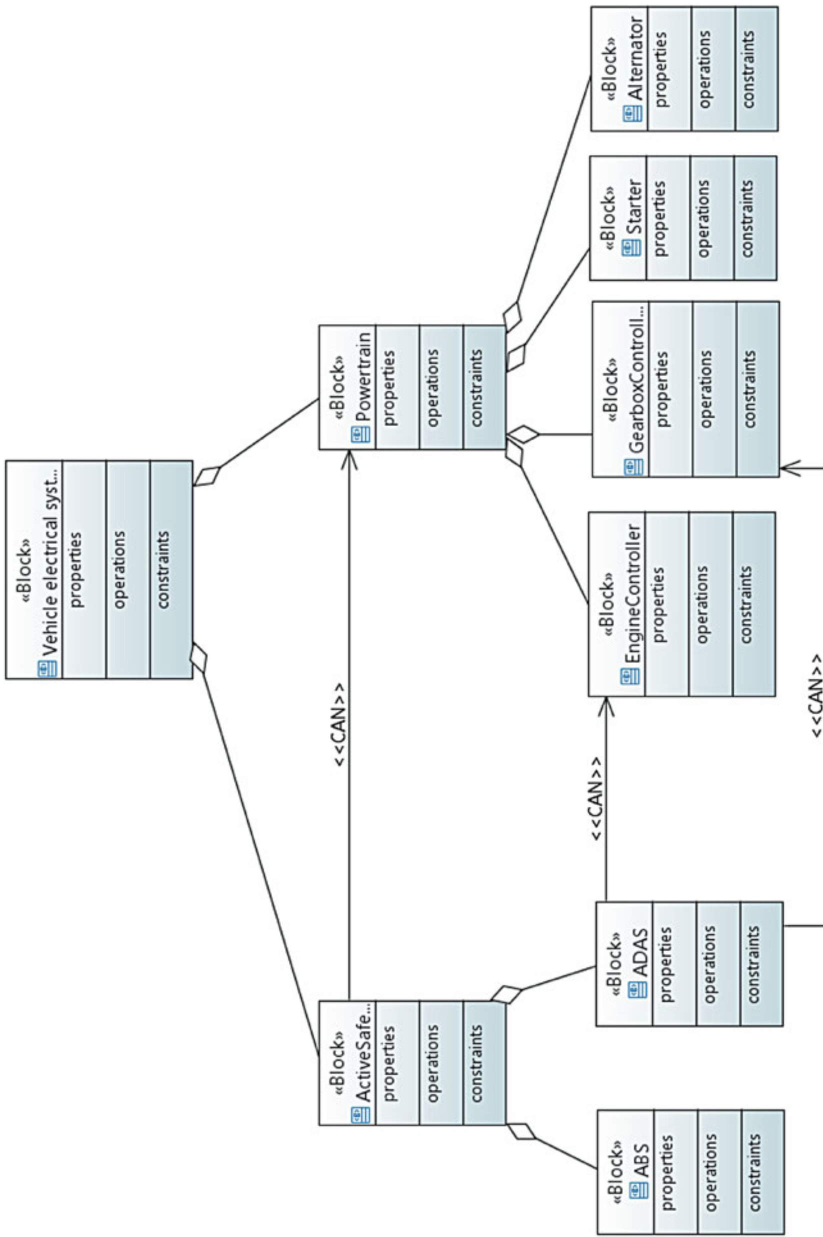

Fig. 2.10 Example of a logical view—a UML class diagram notation

2.7.3.1 How-To

The first step in describing the logical view of the software is to identify the components—these are modelled as UML classes. Once they are identified we should add the relationships between these components in the form of associations. It is important to keep the directionality of the associations correct as these will determine the communication between the components added during the detailed design.

The logical architecture should be refined and evolved during the entire project of the automotive software development.

2.7.4 Relation to the 4+1 View Model

The above-mentioned three views presently used in automotive software engineering evolved from the widely known principles of 4+1 view architecture model presented in 1995 by Kruchten [Kru95]. The 4+1 view model postulates describing software architectures from the following perspectives:

- logical view—describing the design model of the system, including entities such as components and connectors
- process view—describing the execution process view of the architecture, thus allowing us to reason about non-functional properties of the software under construction
- physical view—describing the hardware architecture of the system and the mapping of the software components on the hardware platform (deployment)
- development view—describing the organization of software modules within the software components
- scenario view—describing the interactions of the system with the external actors and internal interactions between components.

These views are perceived as connected with the scenario view overlapping the other four, as presented in Fig. 2.11, adapted from [Kru95].

The 4+1 view model has been used in the telecommunication domain, the aviation domain and almost all other domains. Its close relation to the early version of UML (1.1–1.4) and other software development notations of the 1990s contributed to its wide spread and success.

In the automotive domain, however, the use of UML is rather limited to class/object diagrams and therefore this view is not as common as in the telecommunication domain.

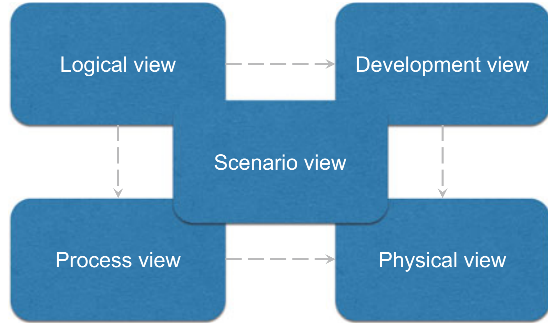

Fig. 2.11 4+1 view model of architecture

2.8 Architectural Styles

As the architecture describes the high-level design principles of the system, we can often observe how these design decisions shape the system. In this case we can talk about the so-called architectural styles. The architectural styles form principles of software design in the same way as building architecture shapes the style of a building (e.g. thick walls in gothic style).

In software design we distinguish between a number of styles in general, but in the automotive systems we can only see a number of those, as the automotive software has harder requirements on reliability and robustness than, for example, web servers. Therefore some of the styles are not applicable.

In this section, let us dive deeper into architectural styles and their examples.

2.8.1 Layered Architecture

This architectural style postulates that components of the system are placed in a hierarchy on top of each other and function calls (API usage) are made only from higher to lower levels, as shown in Fig. 2.12.

We can often see this type of layered architecture in the design of microcontrollers and in the upcoming AUTOSAR standard where the software components are given specific functions such as communication. An example of this kind of architecture is presented in Fig. 2.13.

A special variant of this kind of style is the two-tier style as presented by Steppe et al. [SBG+04], with one layer for the abstract components and the other one for the middleware details. One example of middleware can be found in Chap. 4 in the description of the AUTOSAR standard. Examples of the functionality implemented by the middleware are logging diagnostic events, handling communication on the buses, securing data and data encryption.

An example of such an architecture can be seen in the area of autonomous driving when dividing decisions into a number of layers, as shown in Fig. 2.14 extended from [BCLS16].

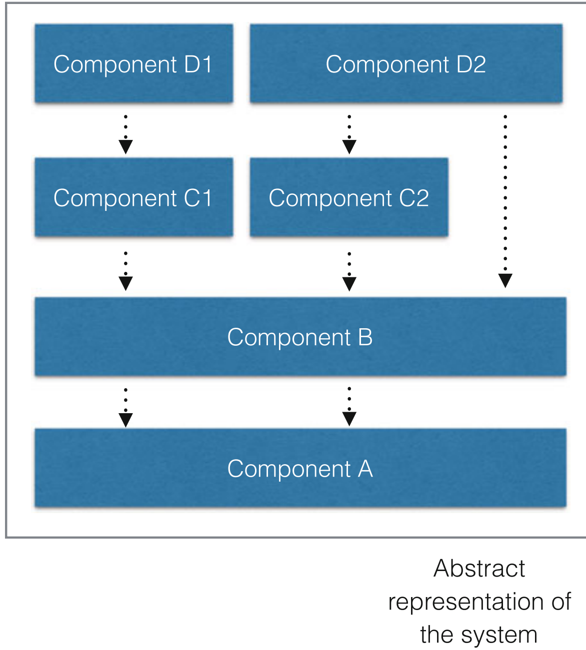

Fig. 2.12 Layered architectural style—*boxes* symbolize components and *lines* symbolize API usage/method calls

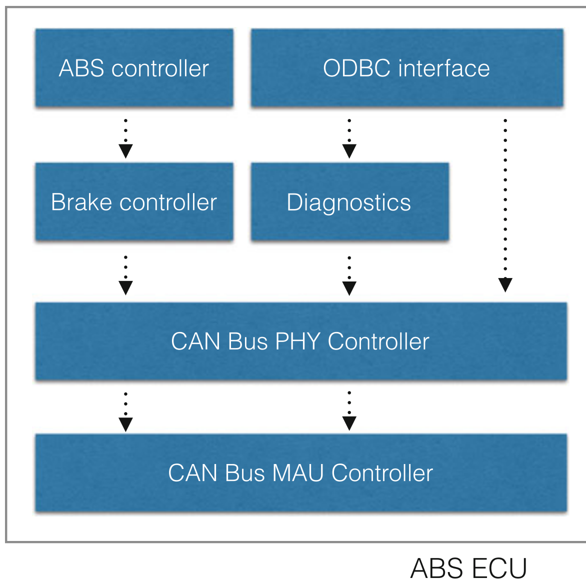

Fig. 2.13 An example of a layered architecture

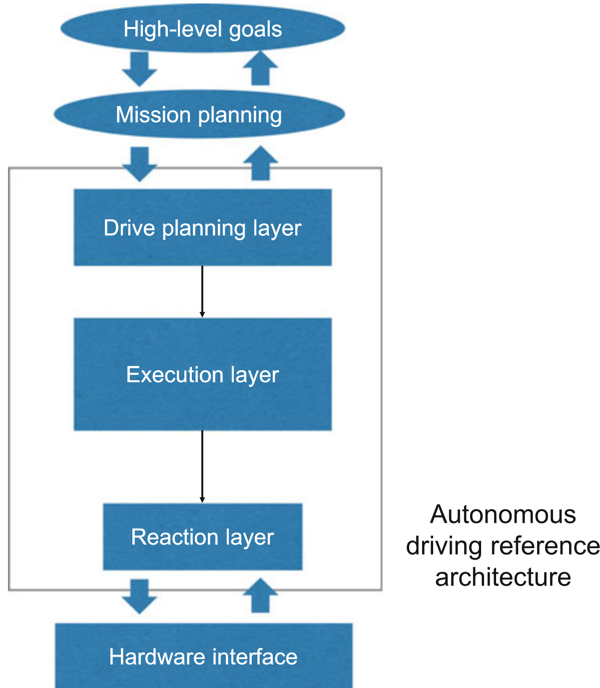

Fig. 2.14 Layered architecture example—decision layers in autonomous driving

In this figure we can see that the functionality is distributed in different layers and the higher layers are responsible for mission/route planning while the lower levels are responsible for steering the car. This kind of modular layered architecture allows the architects to distribute competence into the vertical domains. The blue arrows indicate that this architecture is abstract and that these layers can be connected either directly or indirectly (i.e. there may be other layers in-between).

We quickly realize the this kind of architectural style has limitations caused by the fact that the layers can communicate only in one way. The components within the same layer are often not supposed to communicate. Therefore, there is another style which is often used—component-based.

2.8.2 Component-Based

This architectural style is more flexible than the layered architecture style and postulates the principle that all components can be interchangeable and independent of each other. All communication should go through well-defined public interfaces

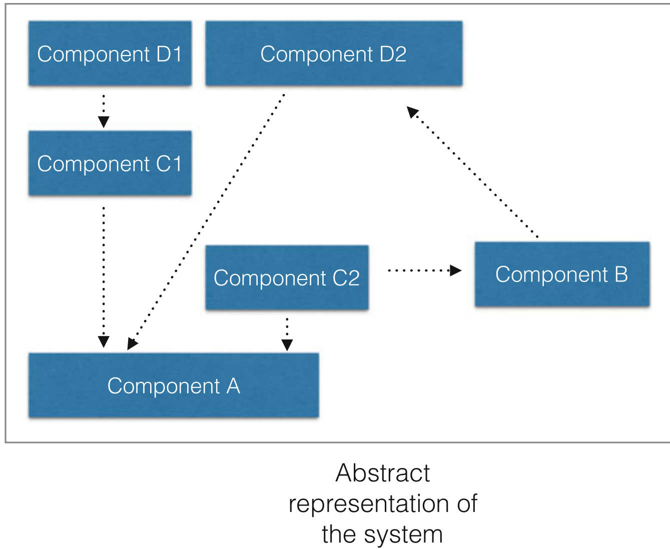

Fig. 2.15 Component-based architectural style

and each component should implement a simple interface, allowing for queries about which interfaces are implemented by the component. In the non-automotive domain this kind of architecture has been populated by Microsoft in its Windows OS through the usage of DLLs (Dynamic Linked Libraries) and the IUnknown interface.

An abstract view of this kind of style is presented in Fig. 2.15.

The component-based style is often used together with the design-by-contract principle, which postulates that the components should have contracts for their interfaces—what the API functions can and cannot do. This component-based style is often well suited when describing the functional architecture of the car's functionality.

In contemporary cars we can see this architectural style in the Infotainment domain, where the system is divided into the platform and the application layer (thus having layered architecture), and for the application layer all the apps which can be downloaded onto the system are designed according to component-based principles. These principles mean that each app can use another one as long as the apps have the right interface. For example a GPS app can use the app for music to play sound in the background without leaving the GPS. As long as the music app exposes the right interface, it makes no difference to the GPS app which music app is used.

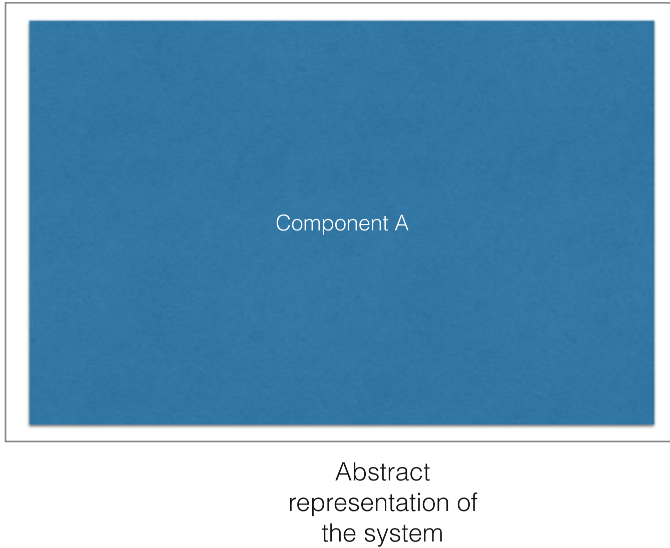

Fig. 2.16 Monolithic architectural style

2.8.3 Monolithic

This style is the opposite of that of component-based architecture as it postulates that the entire system is one large component and that all modules within the system can use each other. This style is often used in low-maturity systems as it leads to high coupling and high complexity of the system. An abstract representation in shown in Fig. 2.16.

The monolithic architecture is often used for implementing parts of the safety-critical system, where the communication between components needs to be done in real time with as little communication overhead as possible. Typical mechanisms in the monolithic architectures are the "safe" mechanisms of programming languages such as use of static variables, no memory management and no dynamic structures.

2.8.4 Microkernel

Starting in the late 1980s, software engineers started to use microkernel architecture when designing operating systems. Many of the modern operating systems are built in this architectural style. In short, this architectural style can be seen as a special case of the layered architecture with two layers:

- Kernel—a limited set of components with the higher execution privileges, such as task scheduler, memory manager, and basic interprocess communication

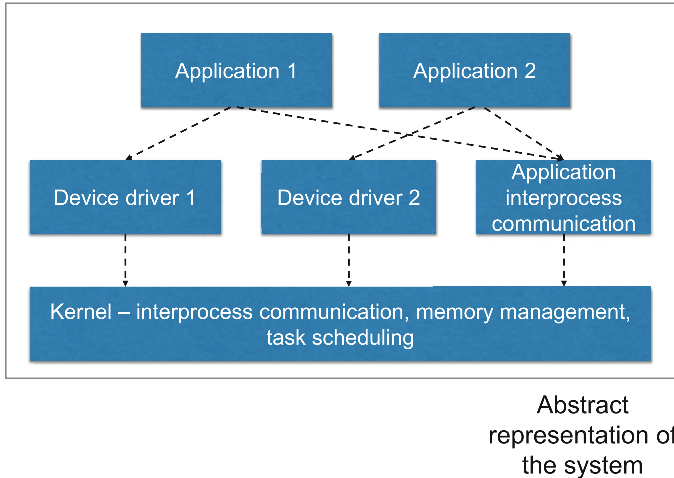

Fig. 2.17 Microkernel architectural style

manager. These components have the precedence over the application layer components.

- Application—components such as user application processes, device drivers, or file servers. These components can have different privilege levels, but always lower than that of the kernel processes.

The graphical overview of such an architectural style is show in Fig. 2.17.

In this architectural style it is quite common that applications (or components) communicate with each other over interprocess communications. This type of communication allows the operating system (or the platform) to maintain control over the communications.

In the automotive domain, the microkernel architecture is used in certain components which require high security. It is argued that the minimality of the kernel allows us to apply the principles of least privilege, and therefore remain in control of the security of the system at all times. It is also sometimes argued that hypervisors of the virtualized operating systems are build according to this principle. In the automotive domain the use of virtualization is currently in the research stage, but seems to be very promising as it would allow us to minimize the costs of hardware while at the same time retain the flexibility of the electrical system (imagine all cars had the same hardware and one could only use different virtual OSs and applications for each brand or type of car!).

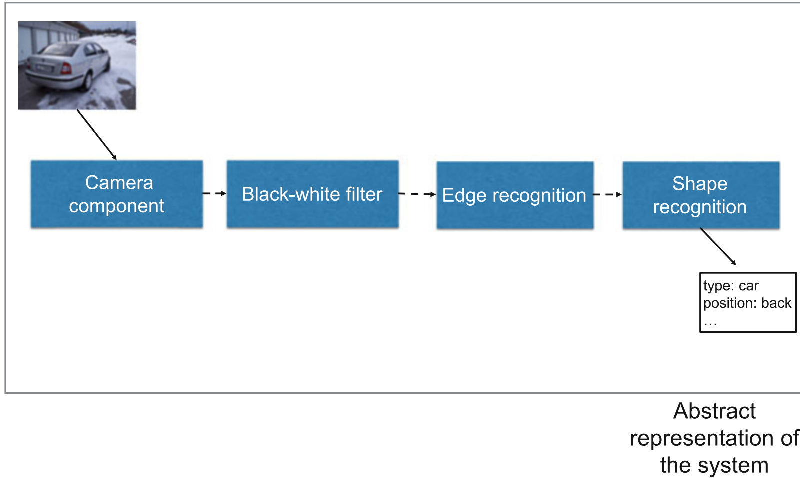

Fig. 2.18 Pipes and filters architectural style

2.8.5 Pipes and Filters

Pipes and filters is another well-known architectural style which fits well for systems that operate based on data processing (thus making its "comeback" as Big Data enters the automotive market). This architectural style postulates that the components are connected along the flow of the data processing, which is conceptually shown in Fig. 2.18.

In contemporary automotive software, this architectural style is visible in such areas as image recognition in active safety, where large quantities of video data need to be processed in multiple stages and each component has to be independent of the other (as shown in Fig. 2.18) [San96].

2.8.6 Client–Server

In client-server architectural style the principles of the design of such systems prescribe the decoupling between components with designated roles—servers which provide resources upon the request of the clients, as shown in Fig. 2.19. These requests can be done in either the pull or the push manner. Pulled requests mean that the responsibility for querying the server lies with the client, which means that the clients need to monitor changes in resources provided by the server. Pushed requests mean that the server notifies the relevant clients about changes in the resources (as in the event-driven architectural style and the published subscriber style).

In the automotive domain, this style is seen in specific forms like publisher-subscriber style or event-driven style. We can see the client-server style in such

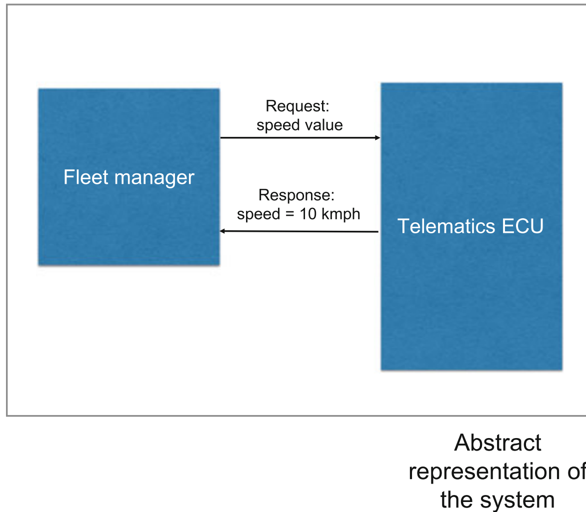

Fig. 2.19 Client-server architectural style

components as telemetry, where the telematics components provide the information to the external and internal servers [Nat01, VS02].

2.8.7 Publisher–Subscriber

The publisher–subscriber architectural style can be seen as a special case of the client–server style, although it is often perceived as a different style. This style postulates the principle of loose coupling between providers (publishers) of the information and users (subscribers) of the information. Subscribers subscribe to a central storage of information in order to get notifications about changes in the information. The publisher does not know the subscribers and the responsibility of the publisher is only to update the information. This is in clear contrast to the client–server architecture, where the server sends the information directly to a known client (known as it is the client that sends the request). The publisher–subscriber style is illustrated in Fig. 2.20.

In automotive software, this kind of architectural style is used when distributing information about changes in the status of the vehicle, e.g. the speed status or the tire pressure status [KM99, KB02]. The advantage of this style is the decoupling of information providers from information subscribers so that the information providers do not get overloaded as the number of subscribers increases. However, the disadvantage is the fact that the information providers do not have control of which components use the information and what information they possess at any given time (as the components do not have to receive updates synchronously).

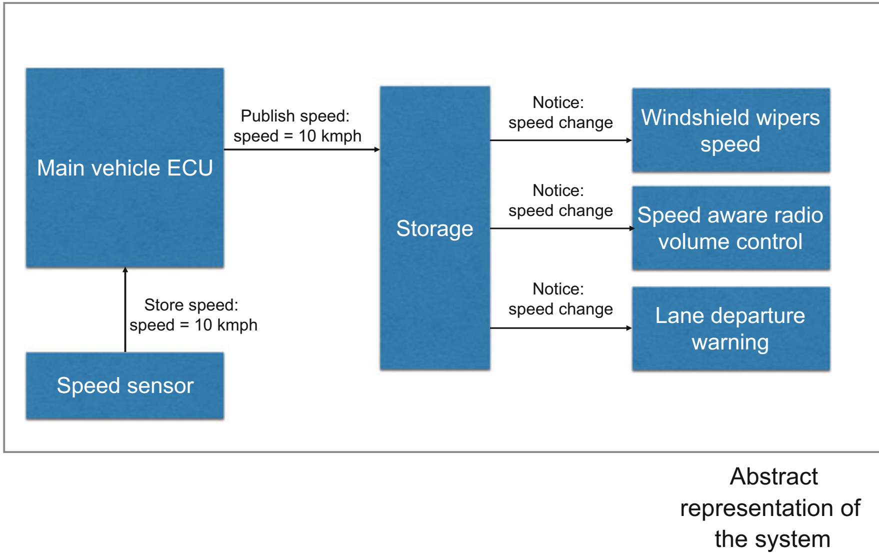

Fig. 2.20 Publisher–subscriber architectural style

2.8.8 *Event-Driven*

The event-driven architectural style has been popularized in software engineering together with graphical user interfaces and the use of buttons, text fields, labels and other graphical elements. This architectural style postulates that the components listen for (hook into) the events that are sent from the component to the operating system. The listener components react upon receipt of the event and process the data which has been sent together with the event (e.g. position of the mouse pointer on the screen when clicked). This is conceptually presented in Fig. 2.21.

The event driven architectural style is present in a number of parts of the automotive software system. Its natural placement with the user interface of the infotainment or the driver assist systems (e.g. voice control), which is also present in the aviation industry [Sar00] is obvious. Another use is diagnostics and storage of the error codes [SKM+10]. Using Simulink to design software systems and using stimuli and responses, or sensors and actuators, shows that event-driven style has been incorporated.

2.8.9 *Middleware*

The middleware architectural style postulates the existence of a common request broker which mediates the usage of resources between different components. The

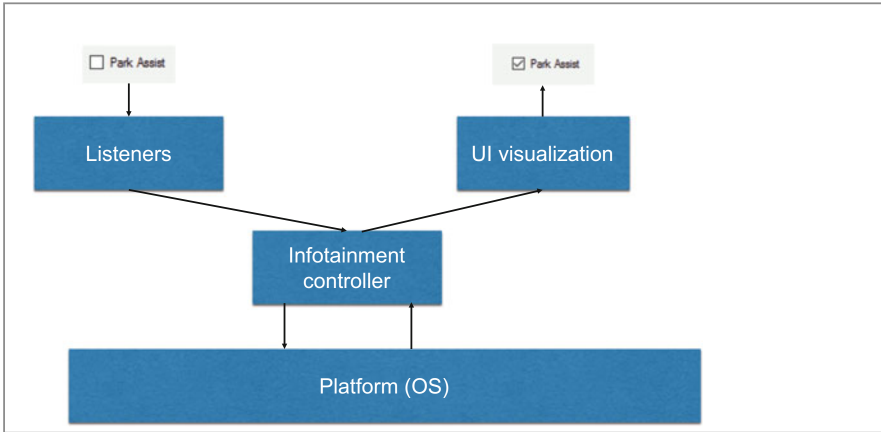

Fig. 2.21 Event-driven architectural style

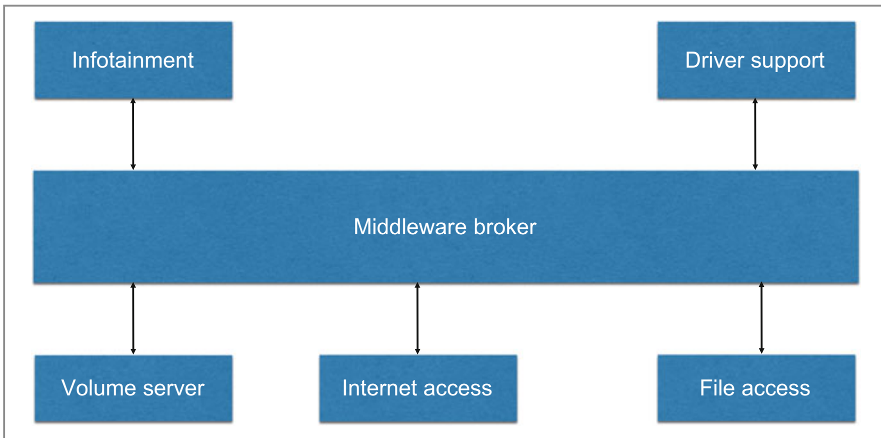

Fig. 2.22 Middleware architectural style

concept has been introduced into software engineering together with the initiative of CORBA (Common Object Request Broker Architecture) by Object Management Group [OPR96, Cor95]. Although the CORBA standard itself is not relevant for the automotive domain, its principles are present in the design of the AUTOSAR standard with its meta-model to describe the common elements of automotive software. The conceptual view of middleware style is shown in Fig. 2.22.

In automotive software, the middleware architecture is visible in the design of the AUTOSAR standard, which is discussed in detail later on in this book. The usage of middleware becomes increasingly important in automotive software's mechanisms of adaptation [ARC+07] and fault tolerance [JPR08, PKYH06].

2.8.10 Service-Oriented

Service-oriented architectural style postulates loose coupling between component using internet-based protocols. The architectural style puts emphasis on interfaces which can be accessed as web services and is often depicted as in Fig. 2.23.

Here the services can be added and changed on-demand during the runtime of the system.

In automotive software, this kind of architecture style is not widely used, but there are areas where the on-demand or ad hoc services are needed. One examples is vehicle platooning which has such an architecture [FA16], and is presented in Fig. 2.24.

Since vehicle platooning is done "spontaneously" during driving, the architecture needs to be flexible and needs to allow vehicles to link to and unlink from each other without the need to recompile or restart the system. The lack of available interfaces can lead to change in the vehicle operation mode, but not to disturbance in the software operation.

Now that we have introduced the most popular architectural styles, let us discuss the languages used to describe software architectures.

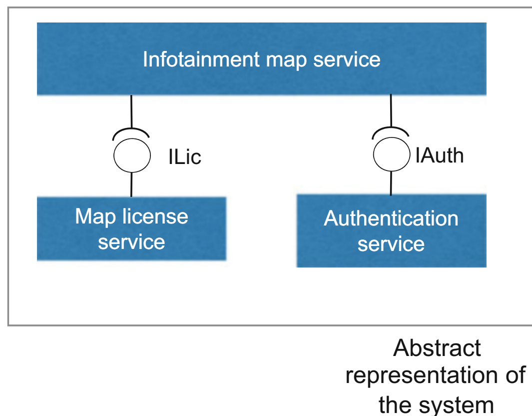

Fig. 2.23 Service-oriented architectural style

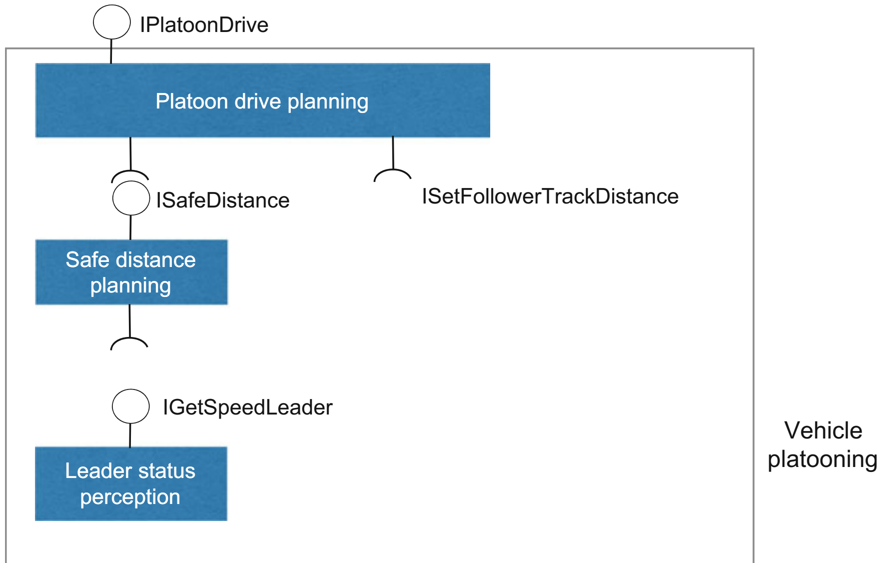

Fig. 2.24 An example of a service-oriented architecture—vehicle platooning

2.9 Describing the Architectures

In this book we have seen multiple ways of drawing architectural diagrams depending on the purpose of the diagram. We used the formal UML notation in Fig. 2.10 when describing the logical components of the software. In Fig. 2.8 we used boxes and lines, which are different from the boxes and lines used in Figs. 2.12, 2.13, 2.14, 2.15, 2.16, 2.17, 2.18, 2.19, 2.20, 2.21, and 2.22. It all has a purpose.

By using different notations we could see that there is no unified formalism describing a software architecture and that software architecture is a means of communication. It allows architects to describe the principles guiding the design of their system and discuss the implications of the principles on the components. Each of these notations could be called ADL—Architecture Description Language. In this section we introduce the most relevant ADLs which are available for software architects, with the focus on two formalisms—SysML (Systems Modelling Language, [HRM07, HP08]) and EAST-ADL [CCG+07, LSNT04].

2.9.1 SysML

SysML is a general-purpose language based on Unified Modelling Language (UML). It is built as an extension of a subset of UML to include more diagrams

(e.g. Requirements Diagram) and reuse a number of UML symbols with the profile mechanism. The diagrams (views) included in SysML are:

- Block definition diagram—an extended class diagram from UML 2.0 using stereotyped classes to model blocks, activities, their attributes and similar. As the "block" is the main building block in SysML, it is reused quite often to represent both software and hardware blocks, components and modules.
- Internal block diagram—similar to the block definition diagram, but used to define the elements of a block itself
- Package diagram—the same as the package diagram from UML 2.0, used to group model elements into packages and namespaces
- Parametric diagram—diagram which is a special case of the internal block diagram and allows us to add constraints to the elements of the internal block diagram (e.g. logical constraints on the values of data processed).
- Requirement diagram—contains user requirements for the system and allows us to model and link them to the other model elements (e.g. blocks). It is one of the diagrams that adds a lot of expressiveness to SysML models, compared to the standard Use Case diagrams of UML.
- Activity diagram—describes the behaviour of the system as an activity flow.
- Sequence diagram—describes the interaction between block instances in a notation based on MSC (Message Sequence Charts) from the telecommunications domain.
- State machine diagram—describes the state machines of the system or its components.
- Use case diagram—describes the interaction of the system with its external actors (users and other systems).

An example of a requirement diagram is presented in Fig. 2.25 from [SSBH14].

The diagram presents two requirements related to each other (Maximum Acceleration and Engine Power) with the dependency between them. Blocks like the "6-Cylinder Engine" are linked to these requirements with the dependency "satisfy" to show where these requirements are implemented.

As we can quickly see from this example, the requirements diagram can be used very effectively to model the functional architecture of the electrical system of a car.

The block diagram was presented when discussing the logical view of the architecture (Fig. 2.10) and it can be further refined into a detailed diagram for a particular block, as shown in Fig. 2.26.

The diagram fulfills a similar purpose as the detailed design of the block, which is often done using the Simulink modelling language. In this book we look into the details of Simulink design in Chap. 6.

The behavioral diagrams of SysML are important for the detailed design of automotive systems, but they are out of the scope of this chapter as the architecture model is supposed to focus on the structure of the system and therefore kept on a high abstraction level.

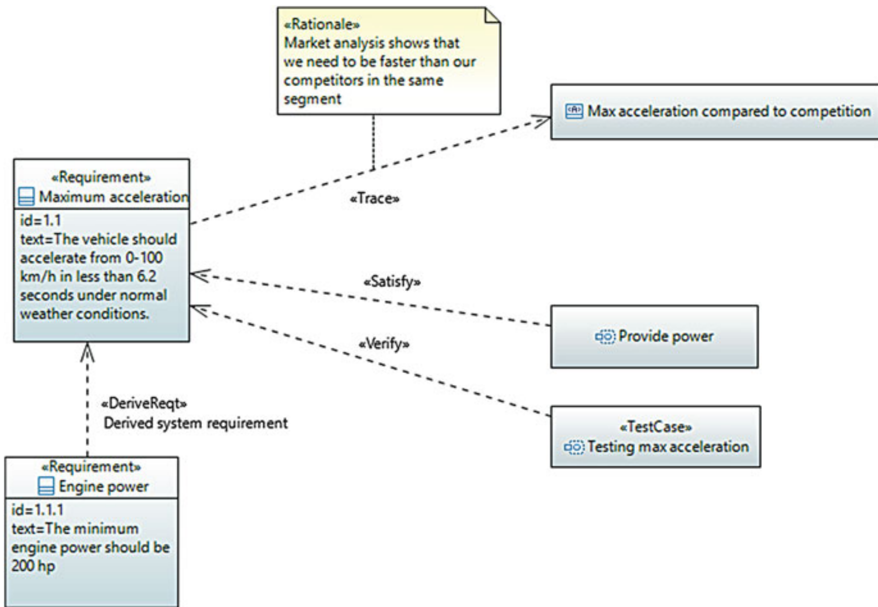

Fig. 2.25 Example requirements diagram

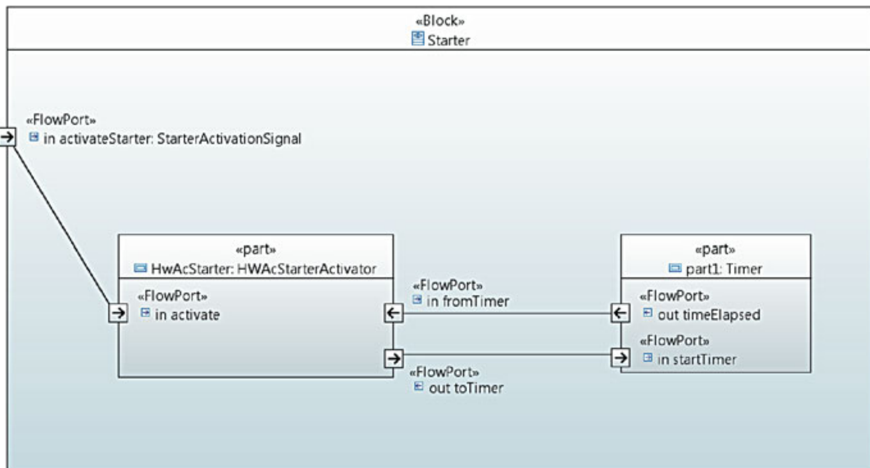

Fig. 2.26 Internal block diagram

2.9.2 EAST ADL

EAST ADL is another modelling language based on UML which is intended to model automotive software architectures [CCG+07, LSNT04]. In contrast to SySML, which was designed by an industrial consortium, EAST ADL is the result

of a number of European Union-financed projects which included both research and development components.

The principles of EAST ADL are similar to those of SySML in the sense that it also allows us to model automotive software architecture in different abstraction levels. The abstraction levels of EAST ADL are:

- Vehicle level—architectural model describing the vehicle functionality from an external perspective. It is the highest abstraction level in EAST ADL, which is then refined in the Analysis model.
- Analysis level—architectural model describing the functionality of the vehicle in an abstract model, including dependencies between the functions. It is an example of a functional architecture, as discussed in Sect. 2.7.1.
- Design level—architectural model describing the logical architecture of the software, including its mapping to hardware. It is similar to the logical view from Fig. 2.10.
- Implementation level—detailed design of the automotive software; here EAST ADL reuses the concepts from the AUTOSAR standard.

The vehicle level can be seen as a use case level of the specification where the functionality is designed from a high abstraction level and then gradually refined into the implementation.

Since EAST ADL is based on UML, the visual representation of models in EAST ADL is very similar to the models already presented in this chapter. However, there are some differences in the structure of the models and therefore the concepts used in SySML and EAST ADL may differ. Let us illustrate one of the differences with the requirements model in Fig. 2.27.

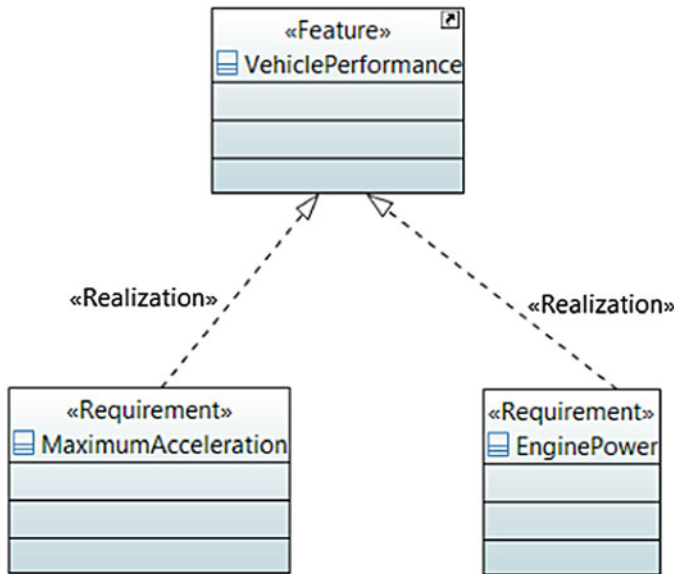

Fig. 2.27 Feature (requirements) diagram in EAST ADL

The important difference here is the link of the requirement—in EAST ADL the requirements can be linked to Features, a concept which does not exist in SySML.

In general, EAST ADL is a modelling notation more aligned with the characteristics of the automotive domain and makes it easier to structure models for a software engineer. However, EAST ADL is not as widely spread as SySML and therefore not as widely adopted in industry.

2.10 Next Steps

After the architecture is designed in the different diagrams, it should be transferred to the product development database and linked to all the other elements of the electrical system of the car. The product development database contains the design details of all software and hardware components, the relationships between them and the deployment of the logical software components onto the physical components of the electrical system.

2.11 Further Reading

The architectural views, styles and modelling languages, discussed in this section, are the most popular one used in the software industry today. However, there are also others, which we encourage the interested reader to explore.

Alternative modelling languages which are used in industry are the UML MARTE profile [OMG05, DTA+08]. The MARTE profile has been designed to support modelling of real-time systems in all domains where they are applicable. Therefore there is a significant body of knowledge from using this profile, including executable variants of it [MAD09].

Readers interested in extending modelling languages can find more information in our previous work on language customization [SW06, SKT05, KS02, SKW04] and the way in which these extension can be taught [KS05].

An interesting review of future directions of architectures in general has been conducted by Kruchten et al. [KOS06]. Although the review was conducted over a decade ago, most of its results are valid today.

2.12 Summary

In this chapter we presented the concept of software architecture, its different viewpoints, and its architectural styles and introduced two notations used in automotive software engineering—SySML and EAST ADL.

An interesting aspect of automotive software architectures is that they usually mix a number of styles. The overall style of the architecture can be layered architecture within an ECU, but the architecture of each of the components in the ECU can be service-oriented, pipes and filters or layered. A concrete example is the AUTOSAR architecture. AUTOSAR provides a reference three layer architecture where the first "application" layer can implement service-oriented architecture, the second layer can implement a monolithic architecture (just RTE) and the third, "middleware", layer can implement component-based architecture.

The reasons for mixing these styles is that the software within a modern car has to fulfill many functions and each function has its own characteristics. For the telematics it is the connectivity which is important and therefore client-server style is the most appropriate. Now that we have discussed the basics of architectures, let us dive deeper into other activities in automotive software development, to understand why architecture is so important and what comes before and next.

References

ARC+07. Richard Anthony, Achim Rettberg, Dejiu Chen, Isabell Jahnich, Gerrit de Boer, and Cecilia Ekelin. Towards a dynamically reconfigurable automotive control system architecture. In *Embedded System Design: Topics, Techniques and Trends*, pages 71–84. Springer, 2007.

BCLS16. Manel Brini, Paul Crubillé, Benjamin Lussier, and Walter Schön. Risk reduction of experimental autonomous vehicles: The safety-bag approach. In *CARS 2016 workshop, 4th International Workshop on Critical Automotive Applications: Robustness and Safety*, 2016.

CCG+07. Philippe Cuenot, DeJiu Chen, Sebastien Gerard, Henrik Lonn, Mark-Oliver Reiser, David Servat, Carl-Johan Sjostedt, Ramin Tavakoli Kolagari, Martin Torngren, and Matthias Weber. Managing complexity of automotive electronics using the EAST-ADL. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 353–358. IEEE, 2007.

Cor95. OMG Corba. The common object request broker: Architecture and specification, 1995.

DST15. Darko Durisic, Miroslaw Staron, and Matthias Tichy. Identifying optimal sets of standardized architectural features – a method and its automotive application. In *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 103–112. IEEE, 2015.

DTA+08. Sébastien Demathieu, Frédéric Thomas, Charles André, Sébastien Gérard, and François Terrier. First experiments using the UML profile for MARTE. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 50–57. IEEE, 2008.

EHPL15. Ulf Eliasson, Rogardt Heldal, Patrizio Pelliccione, and Jonn Lantz. Architecting in the automotive domain: Descriptive vs prescriptive architecture. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 115–118. IEEE, 2015.

FA16. Patrik Feth and Rasmus Adler. Service-based modeling of cyber-physical automotive systems: A classification of services. In *CARS 2016 workshop, 4th International Workshop on Critical Automotive Applications: Robustness and Safety*, 2016.

Fri06. Jon Friedman. MATLAB/Simulink for automotive systems design. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 87–88. European Design and Automation Association, 2006.

Für10. Simon Fürst. Challenges in the design of automotive software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 256–258. European Design and Automation Association, 2010.

HP08. Jon Holt and Simon Perry. *SysML for systems engineering*, volume 7. IET, 2008.

HRM07. Edward Huang, Randeep Ramamurthy, and Leon F McGinnis. System and simulation modeling using SysML. In *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pages 796–803. IEEE Press, 2007.

JPR08. Isabell Jahnich, Ina Podolski, and Achim Rettberg. Towards a middleware approach for a self-configurable automotive embedded system. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 55–65. Springer, 2008.

JT13. Marcin Jamro and Bartosz Trybus. An approach to SysML modeling of IEC 61131-3 control software. In *Methods and Models in Automation and Robotics (MMAR), 2013 18th International Conference on*, pages 217–222. IEEE, 2013.

KB02. Jörg Kaiser and Cristiano Brudna. A publisher/subscriber architecture supporting interoperability of the can-bus and the internet. In *Factory Communication Systems, 2002. 4th IEEE International Workshop on*, pages 215–222. IEEE, 2002.

KM99. Joerg Kaiser and Michael Mock. Implementing the real-time publisher/subscriber model on the controller area network (can). In *2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 1999*, pages 172–181. IEEE, 1999.

KOS06. Philippe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. *IEEE software*, 23(2):22–30, 2006.

Kru95. Philippe B Kruchten. The 4 + 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

KS02. Ludwik Kuzniarz and Miroslaw Staron. On practical usage of stereotypes in UML-based software development. *the Proceedings of Forum on Design and Specification Languages, Marseille*, 2002.

KS05. Ludwik Kuzniarz and Miroslaw Staron. Best practices for teaching uml based software development. In *International Conference on Model Driven Engineering Languages and Systems*, pages 320–332. Springer, 2005.

LSNT04. Henrik Lönn, Tripti Saxena, Mikael Nolin, and Martin Törngren. Far east: Modeling an automotive software architecture using the east adl. In *ICSE 2004 workshop on Software Engineering for Automotive Systems (SEAS)*, pages 43–50. IET, 2004.

MAD09. Frédéric Mallet, Charles André, and Julien Deantoni. Executing AADL models with UML/MARTE. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 371–376. IEEE, 2009.

Nat01. Martin Daniel Nathanson. System and method for providing mobile automotive telemetry, July 17 2001. US Patent 6,263,268.

OMG05. UML OMG. Profile for modeling and analysis of real-time and embedded systems (marte), 2005.

OPR96. Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: Common Object Request Broker Architecture*. Prentice Hall PTR, 1996.

PKYH06. Jiyong Park, Saehwa Kim, Wooseok Yoo, and Seongsoo Hong. Designing real-time and fault-tolerant middleware for automotive software. In *2006 SICE-ICASE International Joint Conference*, pages 4409–4413. IEEE, 2006.

RW12. Nick Rozanski and Eóin Woods. *Software systems architecture: Working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

San96. Keiji Saneyoshi. Drive assist system using stereo image recognition. In *Intelligent Vehicles Symposium, 1996., Proceedings of the 1996 IEEE*, pages 230–235. IEEE, 1996.

Sar00. Nadine B Sarter. The need for multisensory interfaces in support of effective attention allocation in highly dynamic event-driven domains: the case of cockpit automation. *The International Journal of Aviation Psychology*, 10(3):231–245, 2000.

SBG+04. Kevin Steppe, Greg Bylenok, David Garlan, Bradley Schmerl, Kanat Abirov, and Nataliya Shevchenko. Two-tiered architectural design for automotive control systems: An experience report. In *Proc. Automotive Software Workshop on Future Generation Software Archtiecture in the Automotive Domain*, 2004.

SGSP16. Ali Shahrokni, Peter Gergely, Jan Söderberg, and Patrizio Pelliccione. Organic evolution of development organizations – An experience report. Technical report, SAE Technical Paper, 2016.

SKM+10. Chaitanya Sankavaram, Anuradha Kodali, Diego Fernando Martinez, Krishna Pattipati Ayala, Satnam Singh, and Pulak Bandyopadhyay. Event-driven data mining techniques for automotive fault diagnosis. In *Proc. of the 2010 Internat. Workshop on Principles of Diagnosis (DX 2010)*, 2010.

SKT05. Miroslaw Staron, Ludwik Kuzniarz, and Christian Thurn. An empirical assessment of using stereotypes to improve reading techniques in software inspections. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

SKW04. Miroslaw Staron, Ludwik Kuzniarz, and Ludwik Wallin. Case study on a process of industrial MDA realization: Determinants of effectiveness. *Nordic Journal of Computing*, 11(3):254–278, 2004.

SSBH14. Giuseppe Scanniello, Miroslaw Staron, Håkan Burden, and Rogardt Heldal. On the effect of using SysML requirement diagrams to comprehend requirements: results from two controlled experiments. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 49. ACM, 2014.

Sta16. Miroslaw Staron. Software complexity metrics in general and in the context of ISO 26262 software verification requirements. In *Scandinavian Conference on Systems Safety*. http://gup.ub.gu.se/records/fulltext/233026/233026.pdf, 2016.

SW06. Miroslaw Staron and Claes Wohlin. An industrial case study on the choice between language customization mechanisms. In *Product-Focused Software Process Improvement*, pages 177–191. Springer, 2006.

VF13. Andreas Vogelsanag and Steffen Fuhrmann. Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 267–272. IEEE, 2013.

VS02. Pablo Vidales and Frank Stajano. The sentient car: Context-aware automotive telematics. In *Proceedings of the Fourth International Conference on Ubiquitous Computing*, pages 47–48, 2002.