Chapter 3
Automotive Software Development

Abstract 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 start by describing the beginning of all software development—requirements engineering—and we describe how requirements are perceived in automotive software development using text and different types of models. We discuss the specifics of automotive software development such as variant management, different integration stages of software development, testing strategies and the methods used for these. We review methods used in practice and explain how they should be used. We conclude the chapter with discussion on the need for standardization as the automotive software development is based on client-supplier relationships between the OEMs and the suppliers developing components of vehicles.

3.1 Introduction

Software development processes are at the heart of software engineering as they provide *structure and rigor* to the practices of developing software [C+90]. Software development processes consist of phases, activities and tasks which prescribe what actors should do. The actors can have different roles in software development such as software construction designers, software architects, project managers and quality managers.

The software development processes are organized in phases where the focus is on a specific part of software development. Historically these phases include:

1. requirements engineering—the phase where ideas about the functions of the software are created and broken down into requirements (atomic pieces of information about what should be implemented)
2. software analysis—the phase where the system analysis is conducted and high-level decisions about the allocation of functionality to the logical part of the system are made
3. software architecting—the phase where the software architects describe the high-level design of the software including its components and allocate them to computational nodes (ECUs)

M. Staron, *Automotive Software Architectures*,
DOI 10.1007/978-3-319-58610-6_3

4. software design—the phase where each of the components is designed in detail
5. implementation—the phase where the design for each component is implemented in programming languages relevant for the design.
6. testing—the phase where the software is tested in a number of different ways, for example through unit and component tests.

These phases are often done in parallel as modern software development paradigms postulate that it is best to design, implement and test software iteratively. However, the prevalent software development model in the automotive industry is the so-called V-model where these phases are aligned to a V-shaped curve, where the design phases are on the left-hand side of the V and the testing phases are on the right-hand side of the V.

3.1.1 V-Model of Automotive Software Development

The V-model is illustrated in Fig. 3.1. This model is prescribed by international industry standards for development of safety-critical systems, like the ISO/IEC 26262 [ISO11].

In the figure, we also make a distinction between the responsibilities of OEMs (vehicle manufactures) and those of their suppliers. This distinction is important as it is often the phase where the handshaking between the suppliers and OEMs takes place, and therefore the requirements are used during the contract negotiations. In this context a detailed, unambiguous and correct requirements specification prevents potentially unnecessary costs related to the changes in requirements caused by misunderstandings between the OEMs and suppliers.

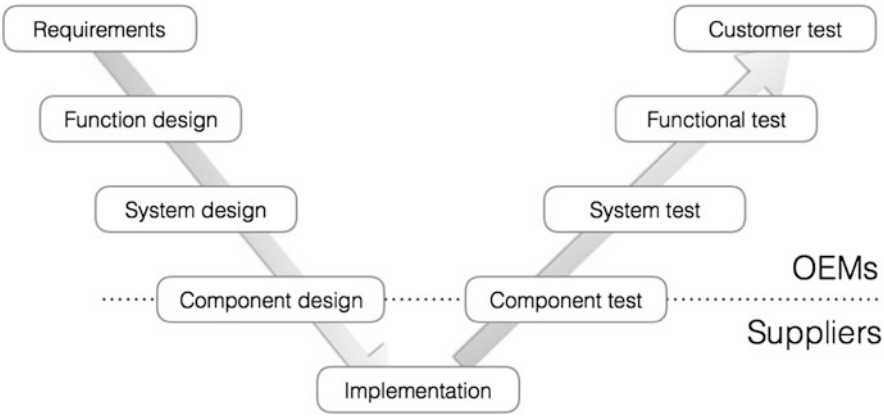

Fig. 3.1 V-shaped model of software development process in automotive software development

In the remainder of this chapter we go through the requirements engineering phase and the testing phase. The analysis and architecture phase are included in the next chapter while the detailed design phase is included in the latter part of the book.

3.2 Requirements

Requirements engineering is a discipline of vehicle development on the one hand and on the other hand a subdomain of software engineering and an initial phase of the software development lifecycle. It deals with the methods, tools and techniques for eliciting, specifying, documenting, prioritizing and quality assuring the requirements. The requirements themselves are very important for the quality of software in multiple senses as the quality is defined as *"The degree to which software fulfills the user requirements, implicit expectations and professional standards."* [C+90].

Requirements engineering in the automotive sector is increasingly about the software since the software is the source of the innovations. According to Houdek [Hou13] and a report about the innovation in the car industry [DB15], the number of functions in an average car grows much faster than the number of devices, with the number of systematic innovations growing faster than the individual innovations. The systematic innovations are systems of software functions rather than individual functions.

Therefore the discipline of requirements engineering is more about engineering than it is about innovation.

The length of an automotive requirements specification is in the range of 100,000 pages for a new car model according to Houdek, based on his study at Mercedes-Benz [Hou13], with ca. 400 documents of 250 pages each at the lowest specification level (component specifications), which are sent over to a large number of suppliers (usually over 100 suppliers, one for each ECU in the car).

Weber and Weisbrod [WW02] showed the complexity and size of requirements specifications in the automotive domain based on their experiences at Daimler-Chrysler. Their large software development projects can have as many as 160 engineers working on a single requirement specification and producing over 3 GB of requirements data. Weber and Weisbrod describe the process of requirements engineering in the following way: "Textual requirements are only part of the game – automotive development is too complex for text alone to manage." This quote reflects the state-of-the-practice of requirements engineering—that the requirements form only one part of the construction database. However, let's look at how the requirements are specified in the automotive domain. Similar challenges of linking requirements to other parts of the construction database can be also found in our previous studies in [MS08].

The requirements are often defined as *(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented*

representation of a condition or capability as in (1) or (2) [C+90]. This definition stresses the link between the user of the system and the system itself, which is important for a number of reasons:

- Testability of the system—it should be clear how a requirement should be tested, e.g. what is the usage scenario realized by the requirement?
- Traceability of the functionality to design—it should be possible to trace which parts of the software realize the requirement in order to provide safety argumentation and enable impact/change management
- Traceability of the project progress—it should be possible to get an overview of which requirements have already been implemented and which are still to be implemented in the project

It is a very technical definition for something that is intuitively well known—a requirement is a way of communicating what we, the users, want in our dream car. In this sense it seems that the discipline of requirements engineering is simple. In practice, working with requirements is very complex as the ideas which we, users, have need to be translated to one of the millions of components of the car and its software. So, let's look at how the automotive companies work with our requirements or dreams.

We talk about software requirements engineering because the automotive industry has recognized the need to move innovation from the mechanical parts of the car to the electronics and software. The majority of us, the customers, buy cars today because they are fast (sporty), safe or comfortable. In many cases these properties are realized by adjusting the software that steers the parts of modern cars. For example we can have the same car with a software package that makes it extremely sporty—look at Tesla's "Insane" acceleration package or Volvo's Polestar performance package. These represent just two challenges which lead to two very important trends in automotive software requirements engineering:

1. Growing amount of software in contemporary cars—as the innovation is driven by software, the amount of software and its complexity grow exponentially. For example the amount of software in the 1990s was a few megabytes of binary code (e.g. Volvo S80) and today reaches over one gigabyte, excluding maps and other user data (e.g. Volvo XC90 of 2016).
2. Safety requirements posed by such standards as ISO 26262—as software steers more parts of the car, there is a larger probability that it can interfere with our driving and cause accidents and therefore it has to be safety-assured just like the software in airplanes and trains. The contemporary standard for functional safety (ISO/IEC 26262, Road vehicles—Functional safety) prescribes methods and processes to specify, design and verify/validate the software.

Automotive software requirements engineering therefore requires rigid processes for handling the construction of software for a car and therefore is very different from other types of software requirements engineering, such as for telecom or web design.

This chapter takes us through the theory of requirements engineering in automotive development by looking into two types of requirements—textual specifications and models used as requirements. It also helps us to explore the evolution of requirements engineering in automotive software development to finally draw on current trends and challenges for the future.

3.2.1 Types of Requirements in Automotive Software Development

When designing software for a car, the designers (who are often referred to as constructors) gradually break down the requirements from car level to component level. They also gradually refine them from textual requirements to models of behaviour of the software. This gradual refinement is due to the fact that the requirements have to be sent to Tier 1 suppliers for development and therefore should be as detailed as possible to enable their validation. In the automotive domain we have a number of tiers of suppliers:

- Tier 1—suppliers working directly with OEMs, usually delivering complete software and hardware subsystems and ECUs to the OEMs
- Tier 2—suppliers working with Tier 1 suppliers, delivering parts of the sub-products which are then delivered by Tier 1 suppliers to the OEMs; Tier 2 suppliers usually do not work directly with OEMs, which makes it even more important for the requirements to be detailed so that they can be correctly broken down by Tier 1 suppliers for Tier 2.
- Tier 3—suppliers working with Tier 2 suppliers, similarly to Tier 2 suppliers working with Tier 1 suppliers. Usually silicon vendors who deliver the hardware together with the drivers.

In this section we describe these different types of requirements, which can be found in these phases.

3.2.1.1 Textual Requirements

AUTOSAR is a great source of inspiration for research in automotive software development, and therefore let us look at the requirements in this standard—they are mostly textual. We use the same template as AUTOSAR for specifying requirements to provide an example of a requirement for keyless entry to the vehicle, as presented in Fig. 3.2.

The structure of the requirement is quite typical for requirements in general—it contains the description, the rationale and the use cases. So far we do not see anything specific. Nevertheless, if we look at the sheer size of such a specification—

REQ-1: Keyless vehicle entryt

Type	Valid
Description	It should be able to open the car with an RFID key
Rationale	The cars of our brand should all have the possibility to be opened using keyless solution. The majority of our competitors have an RFID sensors in the car that opens and starts the car based on the proximity of the designated driver who has the RFID sender (e.g.a card).
Use case	Keyless start-up
Dependencies	REQ-11: RFID implementation
Supporting material	---

Fig. 3.2 An example AUTOSAR requirement

over 1000 pages—we can see that we might confront issues; so let's discuss the kind of issues we can discover.

Rationale The textual requirements are used when describing high-level properties of cars. These types of requirements are mostly used in two phases—the requirements phase, when the specification of the car's functionality at a high level takes place, and at the component design phase, where large software requirements specification documents are sent to suppliers for development (although the textual requirements are often complemented by model-based requirements).

Method Specifying this kind of requirement rarely happens from scratch. Textual requirements are often specified based on models (e.g. UML domain models) and are intended to describe details of the inner workings of software systems. They are often linked to verification methods describing how the requirements should be verified—e.g. describing the test procedure for validating that the requirement is implemented correctly. Quite often it is the suppliers who do the verification, as many requirements demand specific test equipment to test their implementation. If this is the case, the OEMs choose a subset of requirements and verify them to check the correctness of the verification procedure on their side.

Format The text for the requirement is specified in the format which we can see in Fig. 3.2—tables with text. This format is very good if we can specify the details, but they are not very good when we want to communicate overviews and provide the context for the requirements. For that we need other types of requirements—use cases or models.

3.2.1.2 Use Cases

In software engineering the golden standard for specifying requirements is using use cases as defined by Jacobson, together with his Objectory methodology, in the

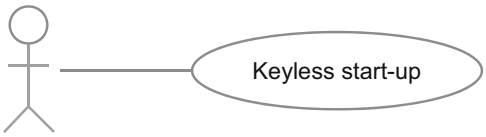

Fig. 3.3 An example use case specification with one use case

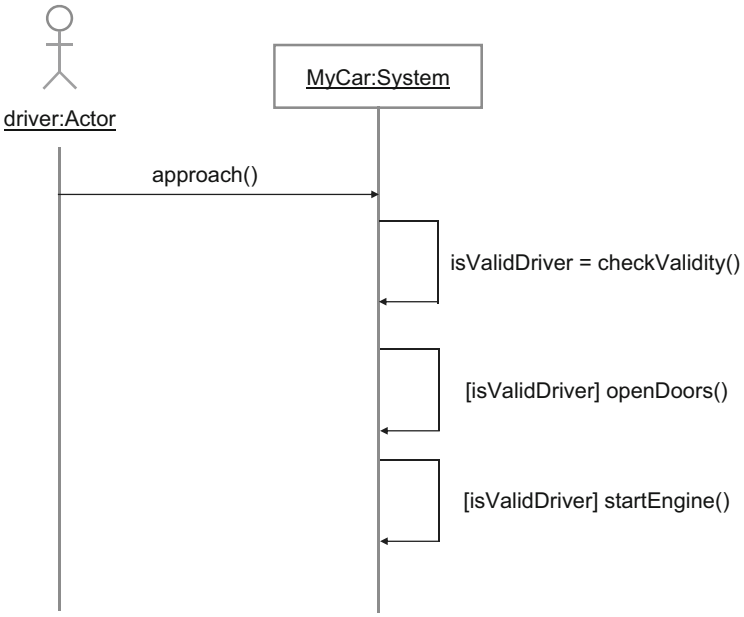

Fig. 3.4 An example specification of a use case using the message sequence charts/sequence diagrams

1990s [JBR97]. The use cases describe a course of interaction between an actor and the system under specification, for example as shown in Fig. 3.3, where the actor interacts with the car in the use case "Keyless start-up". The corresponding diagram (called the use case diagram in UML) is used to present which interactions (use cases) exist and how many actors are included in these interactions.

In the automotive industry this kind of requirements specification is the most common when describing the functions of the vehicles and their dependencies. It is used to describe how the actors (drivers or other cars) interact with the designed vehicle (the system) in order to realize a specific use case. This kind of specification is often described using the sequence diagrams of UML and we can see an example of such a specification in Fig. 3.4.

Rationale The use case specifications provide a high-level overview of the functionality of the designed system, such as a car, and therefore are very useful in the early phases of vehicle development. Usually these early phases are the functional

design (use case diagrams) and the beginning of the system design (use case specifications).

Method Using the high-level descriptions of product properties, the functional designers break down these properties into usage scenarios. These usage scenarios provide a way to identify which of the functions (use cases) are of value to the customers and which are too cumbersome.

Format These kinds of specifications consist of three parts—(1) the use case diagram, (2) the use case specification using a sequence diagram, and (3) the textual specification of a use case detailing the steps of the interaction using somewhat structured natural language.

3.2.1.3 Model-Based Requirements

One method to provide more context to the requirements is to express them as models. This kind of representation can be done in two types of formalisms—UML-like models and Simulink models. In Fig. 3.5 we present an excerpt of a Simulink model for an ABS system from [Dem] and [RSB+13a].

The model shows how to implement the ABS system, but the most important property is that the model shows how the algorithm should behave and therefore how it should be verified.

Rationale Using models as requirements has been recognized by practitioners, and in an automotive software project up to 23% of models are used as requirements

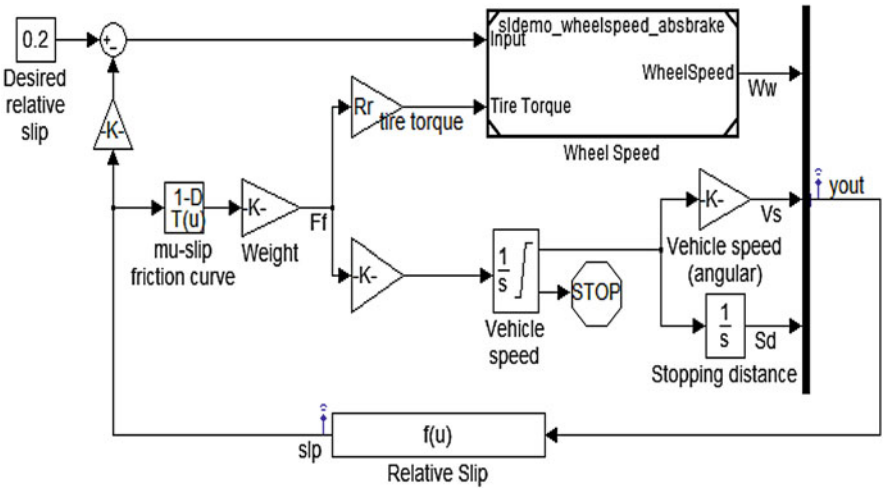

Fig. 3.5 An example Simulink model which can be used as a requirement to describe how to implement the ABS system

according to our previous studies [MS10b] and [MS10a]. According to the same studies, up to 13% of effort is spent in the software project to design these kinds of requirements.

Method The simulation models used for requirements engineering are often used as part of the process of system design and function design, where the software and system designers develop algorithms that describe how functions in modern cars are to be realized. These models can be automatically translated to C/C++ code using code generation, but it is rather uncommon. The reason is that these models describe entire functions which are often partitioned into different domains and spread over multiple components. Quite often these kinds of requirements are translated into textual specifications, shown in the previous subsection.

Format The models are expressed using Simulink or a variation of statechart such as Statemate or Petri nets. These simulation models detail the functions described in the use cases by adding the system view of the interaction—the blocks and signals. The blocks and signals represent the realization of the functionality in the car and are focused on one function only. These models are often used as specifications which are then detailed and often used to generate the source code automatically.

3.3 Variant Management

Having a good database of requirements and construction elements is key to success in automotive software engineering. This is dictated by the fact that the automotive market is based on *variability*—i.e. the locations in the product (software) where it can be configured. As customers we expect the ability to configure our car with the latest and greatest features of hardware, electronics and software.

There are two basic kinds of variability mechanisms in automotive software:

- Configuration—when we configure parameters of the software without modifying its internal structure. This kind of variability is often seen in the non-safety critical functions such as engine calibration or in configuring the availability of functions (e.g. rain sensors).
- Compilation—when we change the internal structure of the software, compile it and then deploy on the target ECU. This kind of variability is used when we need to ensure that the software always behaves in the same way, for example the availability of the function for collision avoidance by breaking.

In this section we explain the basics of these two mechanisms.

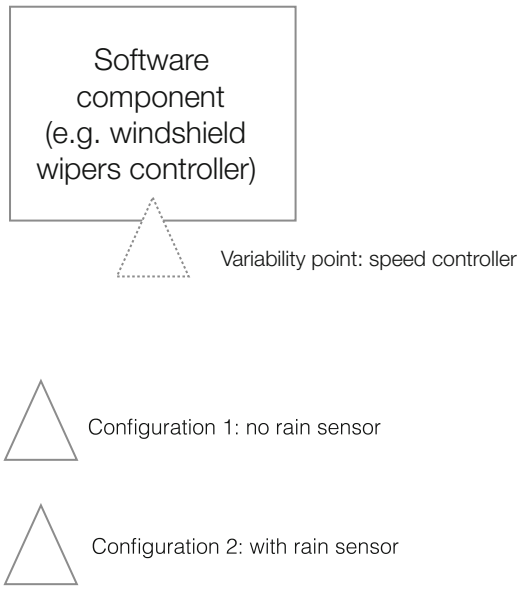

Fig. 3.6 Variability through configuration

3.3.1 Configuration

Configuration is often referred to as runtime variability as changing the software can be done after the software is compiled. Figure 3.6 presents the conceptual view of this kind of variability.

In Fig. 3.6 we can see that we have one variant of the software component (rectangle) with one variability point (the dotted line triangle) which can be configured using two different configurations—without the rain sensor and with the rain sensor. This means that we compile the code for the software component once and then use two different configuration files when deploying the software.

The configuration as a variability mechanism has important implications for the designers of the software. The main implication is that the software has to be tested using multiple scenarios—i.e. the software designers need to be able to prevent use of the software component with invalid configurations.

3.3.2 Compilation

The compilation as a variability mechanism is fundamentally different as it results in a software component which cannot be modified (configured) after its compilation, during runtime. Therefore it is an example of so-called *design time variability* as the designers must decide during design which variant is being developed. This is

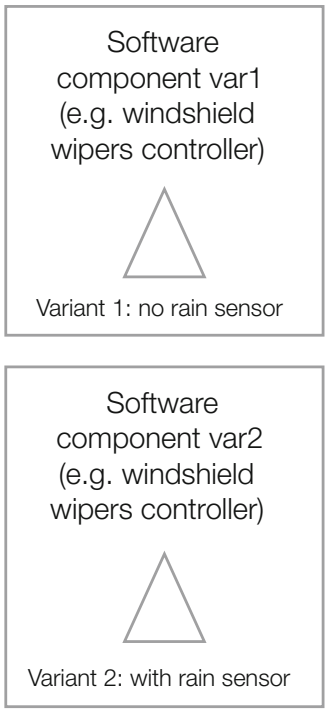

Fig. 3.7 Variability through compilation

conceptually shown in Fig. 3.7 where we can see two different versions of the same component—with and without the rain sensor.

As Fig. 3.7 suggests, there are two different code bases for the software component—one with and one without the rain sensor. This means that the development of these two variants can be decoupled from each other, but that also means that the designers have to maintain two different code bases at the same time. This parallel maintenance means that if there are defects in the common code then both code bases need to be updated and tested.

The main advantage of this kind of variability mechanism is the assurance that the code is not tampered with in any way after the compilation. The code can be tested, and once deployed there is no way that an incorrect configuration can break the quality of the component. However, the main disadvantage of this type of variability management mechanism is the high cost of maintenance of the code base—parallel maintenance.

3.3.3 Practical Variability Management

Both of the above variability management mechanisms are used in practice. Compile time variability is used when the software is an integral part of an ECU

whereas configuration is used when the software can be calibrated to different types of configurations during deployment (e.g. configuration on the assembly line, calibration of the engine and gearbox depending on the powertrain performance settings).

3.4 Integration Stages of Software Development

On the left-hand side of the V-model the main type of activity is refinement of requirements in multiple ways. On the right-hand side of the model the main activity type is integration followed by testing.

In short, integration is the activity where software construction engineers integrate their code with the code of other components and with the hardware. In the first integration stages the hardware is usually simulated hardware in order to allow for unit and component testing (described in Sect. 3.5). In the later integration phases the software code is integrated together with the target hardware, which is then integrated into a complete electrical/electronic system of the car (Table 3.1).

Figure 3.8 shows an example software integration of software modules and components into an electrical system. What is important to notice is the fact that the integration steps (vertical solid black lines) are not synchronized as the development of different software modules is done at different pace.

In practice this figure is even more complicated, as the integration plan is often a document with several dimensions. Each integration cycle (which is what we show in Fig. 3.4) is done several times during the project. First, the so-called basic

Table 3.1 Types of integration

Type	Description
Software integration	This type of integration means that two (or more) software components are put together and their joint functionality is tested. The usual means of integration depend on the what is integrated—it can be merging of the source code if the integration is on the source code level; it can be linking of two binary code bases together; or it can be parallel execution to test interoperability. The main testing techniques are unit and component testing, described in Sect. 3.5
Software-hardware integration	This type of integration means that the software is integrated (deployed) to the target hardware platform. In this type of integration, the focus is on the ability of the complete ECU to be executed and the main testing type is component testing, described in Sect. 3.5
Hardware integration	This type of integration means that the focus is on the integration of the ECUs with the electrical system. In this type of integration the focus is on the interoperability of the nodes and basic functionality, such as communication. The testing related to this type of integration is system testing

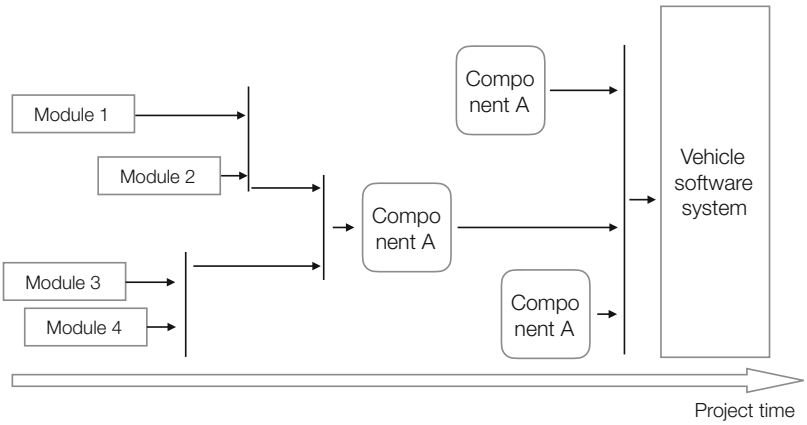

Fig. 3.8 Software integration with integration steps

software is integrated (functionality like the boot code, and communication) and then higher level functionality is added, according to the functional architecture as described in Chap. 2.

3.5 Testing Strategies

Requirements engineering progresses from higher abstraction levels towards more detailed, lower abstraction levels. Testing is the opposite. When the testers test the software they start from the most atomic type of testing—unit testing—where they test each function and each line of code. Then they gradually progress by testing entire components (i.e. multiple units linked together), then the entire system and finally each function. Figure 3.9 shows the right-hand side of the V-model with a focus on the testing phases.

In the coming subsections we look deeper into the testing phases of the automotive software.

3.5.1 Unit Testing

Unit test is the basic test, which is performed on individual entities of software such as classes, source code modules and functions. The goal of unit testing is to find defects related to the implementation of atomic functions/methods in the source code.

The basic scheme of unit testing is the creation of automated test cases which combine individual methods with the data that is needed to achieve the needed

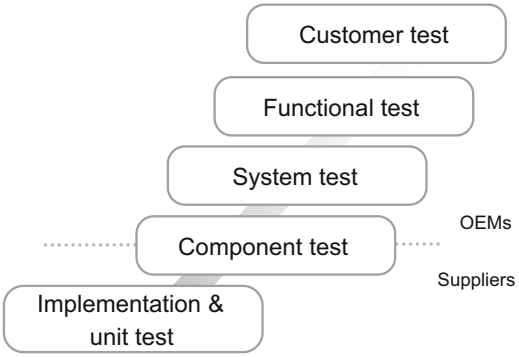

Fig. 3.9 Testing phases in automotive software development

```
1  using System;
2  using Microsoft.VisualStudio.TestTools.UnitTesting;
3  using WindshieldSimulator;
4
5  namespace WindshieldTest
6  {
7      [TestClass]
8      public class BasicSuite
9      {
10         // unit test method
11         [TestMethod]
12         public void TestCreationInitialState()
13         {
14             // arrange
15             WindshieldWiper pWiper;
16
17             // act
18             pWiper = new WindshieldWiper();
19
20             // assert
21             Assert.AreEqual(pWiper.Status,
22                             position.closed,
23                             "Initial status should be /closed/");
24         }
25     }
26 }
```

Fig. 3.10 Example unit test for testing the status of windshield wiper module

quality. The result is then compared to the expected result, usually with the help of assertions. An example of a unit test is presented in Fig. 3.10.

The unit test presented in Fig. 3.10 is a test for correctness of the creation of object "WindshieldWiper"—a unit under test (UAT). This particular test code is written in C# and in practice the test code can be written in almost any programming language. The principles, however, are the same for all unit tests.

Of interest for our chapter are lines 14–23, as they contain the actual test code. Line 15 is the arrangement line which prepares (sets up) the test case—in our example it declares a variable which will be assigned to the object of the class WindshieldWiper. Line 18 is the actuation line which executes the actual test code—in our example creates the object of the WindshieldWiper class.

The most interesting are lines 21–23 since they contain the so-called assertion. The assertion is a condition which should be fulfilled after the execution of the test code. In our example the assertion is that the status of the newly created object (line 21) is "closed" (line 22). If it is not the case, then the error message is logged in the testing environment (line 32) and the execution of the new test cases continues.

Unit testing is often perceived as the simplest type of testing and is most often automated. Frameworks like CppUnit, JUnit or Google test framework can orchestrate the execution of unit tests, allowing us to quickly execute the entire set of tests (called test suites) without the need for manual intervention.

Automated unit tests are also reused in several ways, for example to create nightly regression test suites or to create the so-called "smoke testing" where testers randomly execute test cases to check whether the system exposes random behavior.

It is also important to notice that reuse of test cases needs to be accompanied by the methods to prioritize test cases, e.g. by identifying risky areas in source code [ASM+14] or focusing on code that was changed since the last test run [KSM+15, SHF+13]. It is also important to trace the test process in the context of software reliability growth [RSM+13, RSB+13b].

We can also see that if the test case finds a problem (fails), then troubleshooting is relatively simple—we know which code was executed and under which conditions. This knowledge allows the testers to quickly describe where the defect is or even suggest how to fix it.

3.5.2 Component Testing

This is sometimes also called *integration testing*, as the goal of this type of testing is to test the integrations, i.e. links, between units of code within one of many components. The main characteristic which differentiates component tests from unit tests is that in component testing we use stubs to simulate the environment of the tested component or the group of the components. This is illustrated in Fig. 3.11.

In contrast to unit tests, component tests focus on the interaction between the stubs and the component under test. The goal of this type of testing is to verify that the structure and behavior of the interfaces is implemented correctly.

We should also mention that the number of stubs in the system decreases as the development project progresses. With the progress of the development, new components are designed and they replace the stubs. Hence the nickname of this type of testing—"integration testing".

In automotive systems this type of testing is often done by simulating the environment using either models (the so-called Model-In-the-Loop or MIL testing)

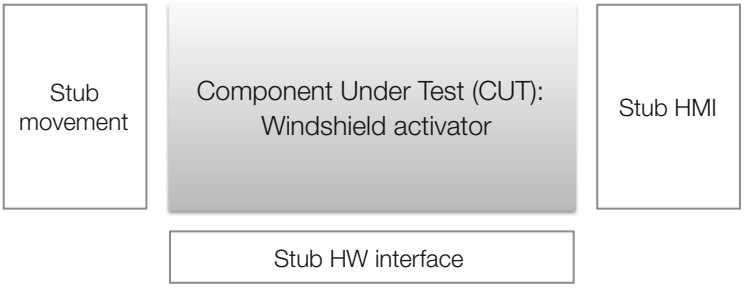

Fig. 3.11 Component under test with the simulated environment

or hardware simulators (the so-called Hardware-In-the-Loop or HIL testing). An example of equipment for HIL testing is presented in Fig. 3.12.

Figure 3.12 shows a testing rig from dSpace, which is widely used in the automotive industry to test components by simulating the external environment.

Since the environment of the components is simulated, the non-functional properties of the components are often hard to test or require very detailed simulations. The very detailed simulations, however, also tend to be very costly.

3.5.3 System Testing

System testing is the phase of testing when the entire system is assembled and tested as a whole. The focus of system testing is on checking whether the system fulfills its specifications in a number of ways. The system testing focuses on verifying the following aspects:

1. functionality—testing whether the system has the functionality as specified in the requirements specification
2. interoperability—testing whether the system can connect to the other systems which are designed to interact with the system under test
3. performance—testing whether the system under test performs within the specified limits (e.g. timing limits, capacity limits)
4. scalability—testing whether the system's operation scales up and down (e.g. whether the communication buses operate with 80 and 120 ECUs connected)
5. stress—testing whether the system operates correctly under high load (e.g. when the maximum capacity of the communication buses is reached)
6. reliability—testing whether the system operates correctly during a specific period of time
7. regulatory and compliance—testing whether the system fulfills legal and regulatory requirements (e.g. carbon dioxide emissions)

Fig. 3.12 HIL testing rig—Image source: dSPACE GmbH. Copyright 2015 dSPACE GmbH—reprinted with permission

System testing is usually the first testing phase when the above aspects can be tested and therefore it is usually the most effective way of testing. However, it is also very costly way of testing and very inefficient, as fixing the defects found in this phase requires frequent changes in multiple components.

In the automotive software this type of testing is often done using the so-called "box cars"—the entire electrical system being set up on tables without the chassi and the hardware components.

Test ID	T0001
Description	Test of the basic function of windshield wipers. The goal of the test is to verify that the windshield wipers can make one sweep with the engine turned off.
Action/Step	**Expected result**
Start ignition	Battery icon on the dashboard lit red; windshield wipers are in the position “closed”.
Push the windshield wipers level one step forward	The windshield wipers start to move to the position “open”
Wait 20 seconds	The windshield wipers go back to the position “closed”
Turn off ignition	All icons on the car’s dashboard turn off; windshield wipers are in position “closed”.

Fig. 3.13 Example of a functional test

3.5.4 Functional Testing

The functional testing phase focuses on verifying that the functions of the system work according to their specification. They correspond to the functional requirements in the form of use cases and are quite often specified according to the use cases. Figure 3.13 presents an example of a functional test—specified as a table.

What is important in this example is the specification, which is similar to the specification of a use case—the description of the action (step) on the left-hand side together with the expected outcome on the right-hand side. We can also observe that the functional test does not require the knowledge of the actual construction of the system under test (SUT), which led to the nickname of these tests as “black-box testing”.

We should not focus on the simplicity of the example because functional testing is often the most effort-intensive type of testing. It is often done in a manual manner and requires sophisticated equipment to conduct.

Examples of sophisticated functional test cases are safety test cases where OEMs test their safety systems. To be able to test such a function, car manufacturers need to recreate the situation where the system could be activated and check whether it was activated. They also need to recreate the situation when it should not be activated and test that it was not activated.

When the functional test fails, it is rather difficult to find the defect, as the number of construction elements which take part in the interaction can be quite large—in our example the failure of the functional test case could be caused by anything from mechanical failure of the battery to design defect in the software. Therefore functional testing is often used after the other tests are conducted to validate functionality rather than to verify the design.

3.5.5 *Pragmatics of Testing Large Software Systems: Iterative Testing*

As the electrical system of contemporary cars is very complex, OEMs often apply concepts of interactive testing to their development. Concept of iterative testing means that the functionality of the software is divided into levels (as prescribed by the functional architecture described in Chap. 2) and the functions are tested using unit, component, system and functional testing per layer. This means that the basic functionality such as booting up of the electronics, starting up of the communication protocols, running diagnostics, etc. are tested first and the more advanced functions such as lighting, steering, and braking are tested later, followed by more advanced functions such as driver alerts.

3.6 Construction Database and Its Role in Automotive Software Engineering

All these types of requirements need to come together somehow and that's why we have the process and the infrastructure for requirements engineering. Let us start with the infrastructure—usually named the design or construction database. In the light of work of Weber and Weisbrod [WW02] it is called the common information model. Figure 3.14 shows how this design database is used. The construction

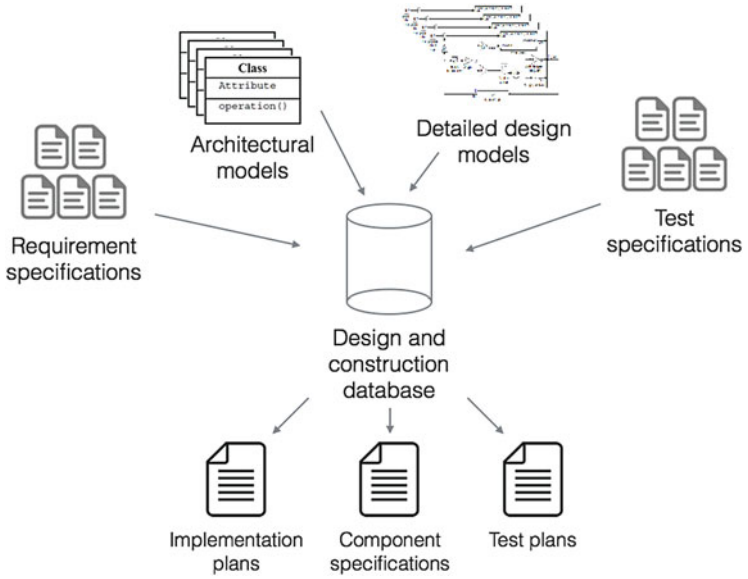

Fig. 3.14 Design database

database contains all elements of the design of the electrical system of the vehicle—components, electronic control units, systems, controllers, etc. The structure of such a database is hierarchical and reflects the structure of the vehicle. Each of the elements in the database has a set of requirements linked to it. The requirements are also linked to one another to show how they are broken down. Such a database grows over time and is version-controlled as different versions of the same elements can be used in different vehicles (e.g. different year models of the same car or different cars).

An example of such a system is described by Chen et al. [CTS+06] and has been developed by the company Systemite, which specializes in databases for vehicle construction. Such a database structures all the elements of the construction of the electronics of the vehicle and links all artifacts to the construction elements. An example of a construction element is the engine's electronic control unit, and all the functions that use this control unit are linked to it.

Such a database usually has a number of views which show the required set of details—functional view, architectural view, topological view and software components' view. Each view provides the corresponding entry point and shows the relevant elements, but the database is always in a consistent state where all the links are valid.

The database is used to generate construction specifications for different actors. For each supplier that delivers an ECU, the database generate the set of all requirements which are linked to the ECU and all models which describe the behaviour of the ECU. Sometimes, depending on the situation, the documentation contains even the simulation models for the functions which are to be included in the ECU.

One of the commercial tools available on the market which is used as a construction database is the tool SystemWeaver provided by Systemite. The main strength of such a tool is the ability to link all elements together. In Fig. 3.15 we can see how the requirements are linked to the software architecture model. On the left-hand side we can see that the requirements are part of an element (e.g. "Adjust speed" as part of the "Adaptive cruise control"), and on the right-hand side another requirement visualized as a diagram.

Such tools provide specific views, for example listing all requirements linked to a specific function as shown in Fig. 3.16. As part of that view we can see that the text is complemented with figures which allow the analysts to be more specific when specifying requirements and allow the designers to understand the requirements better.

The ability to link the elements from different views (e.g. requirements and components) and provide a graphical overview of these elements allows the architects to quickly perform change impact analyses and reason about their architectural choices. Such a dynamic creation of views is very important when assessing architectures (e.g. during ATAM assessments). An example of such a view is one showing a set of architectural components used in realization of a specific user function, as shown in Fig. 3.17.

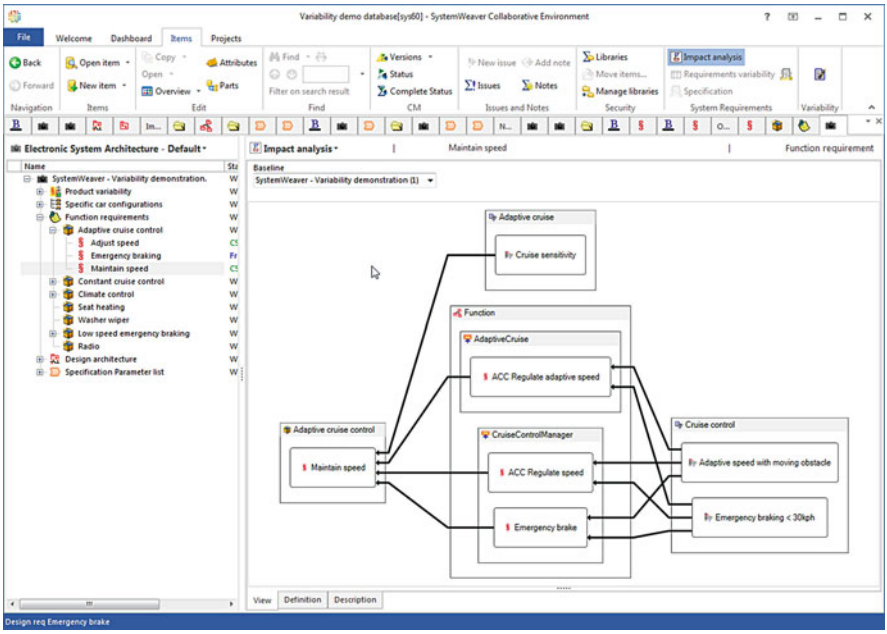

Fig. 3.15 Design database linking requirements to architectural elements. Copyright 2016, Systemite—reprinted with permission

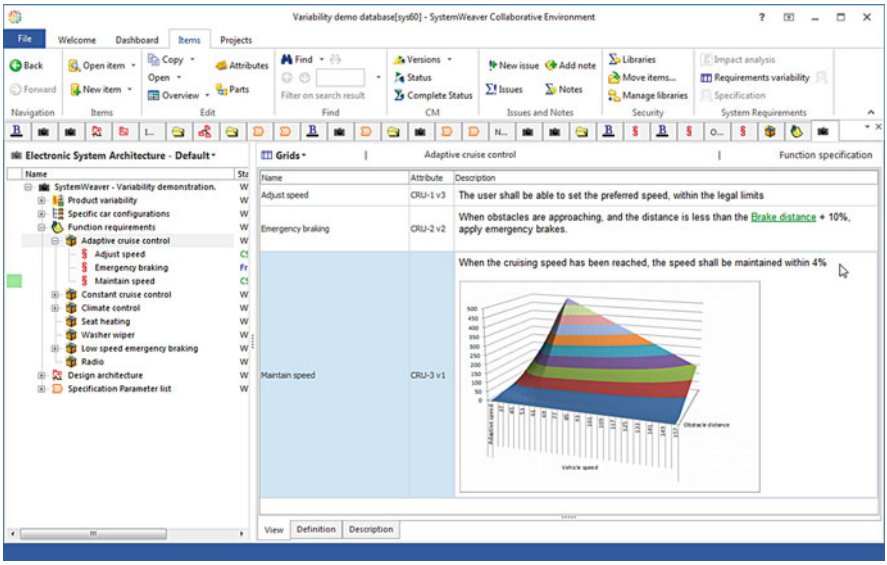

Fig. 3.16 Design database listing requirements for a specific function. Copyright 2016, Systemite—reprinted with permission

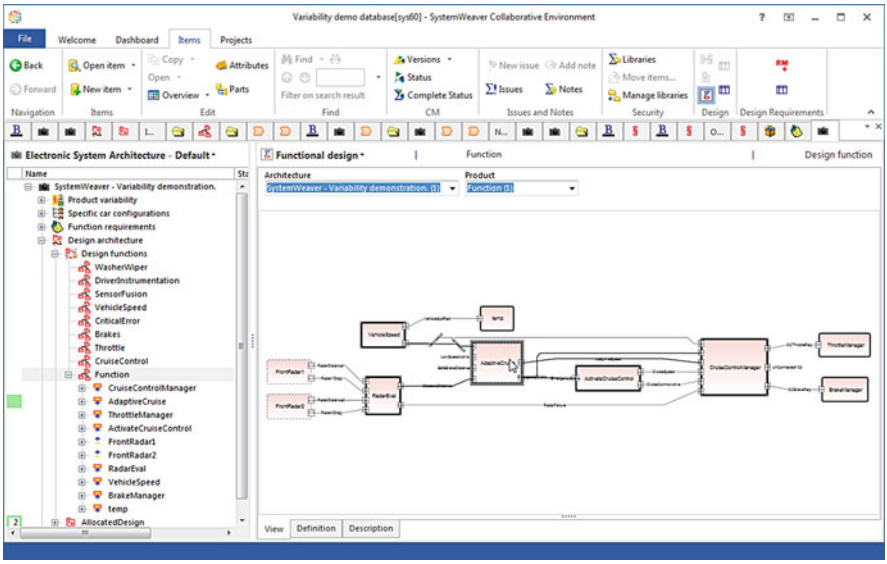

Fig. 3.17 Design database showing architectural components used when designing a specific function. Copyright 2016, Systemite—reprinted with permission

The system construction database can also help us in linking requirements to test cases during the test planning phase—as shown in Fig. 3.18.

It can also assist us in tracking the progress of testing—Fig. 3.19. Since the number of requirements is so large in automotive systems, tracking the progress of whether they are tested is also not trivial. Therefore a unified view is needed where the project can track the test cases that are planned to cover certain requirements, as well as those that they were executed and what the result of the execution was.

The construction database and modelling tool provide the project teams with a consistent view on their software system. In the case of software architectures this tool allows us to link together all the views presented in Chap. 2 (such as physical, logical, and deployment) and therefore avoid unnecessary work to keep documents in a steady and consistent state. Most of the tools available for this purpose provide the possibility to handle multiple parallel versions and baselines, which is essential in the development of automotive software.

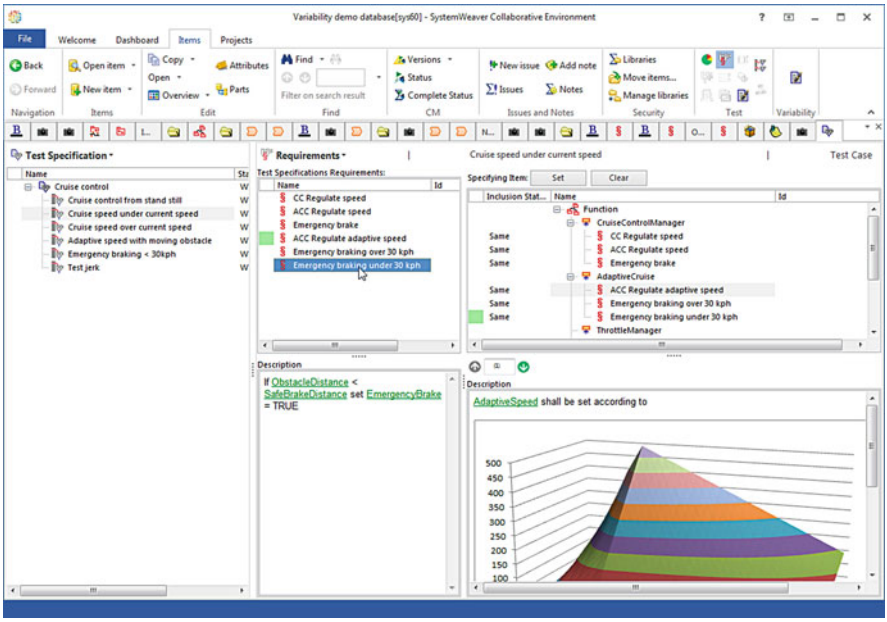

Fig. 3.18 Linking test cases to requirements. Copyright 2016, Systemite—reprinted with permission

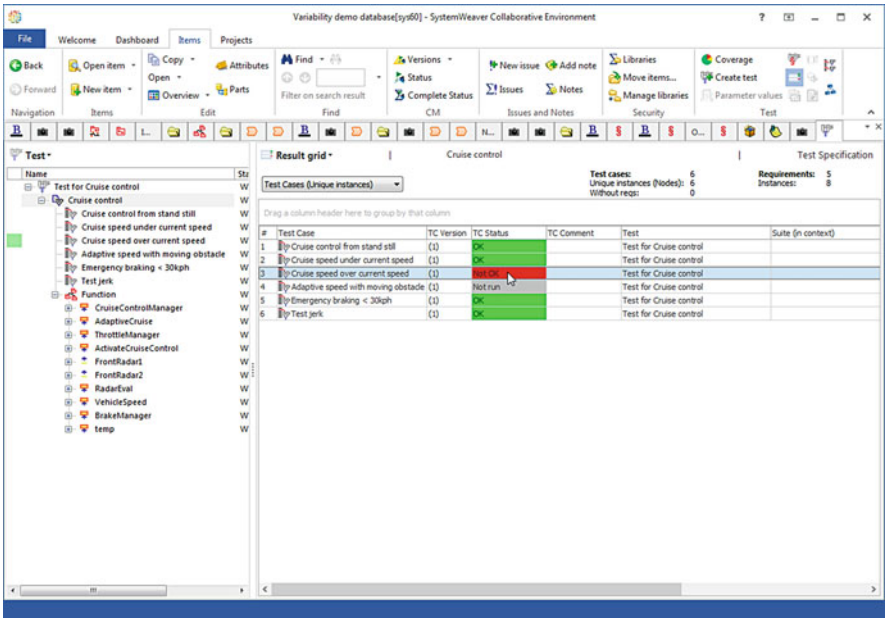

Fig. 3.19 Tracking test progress. Copyright 2016, Systemite—reprinted with permission

3.7 Further Reading

In this chapter we outlined the practical aspects of automotive software development from a bird's eye perspective. Interested readers can turn to more literature in the area to dive deeper into details.

For the automotive software processes we recommend the book by Schäuffele and Zurawka [SZ05], which presents a classical view on automotive software development, starting from low-level processor programming and moving on to advanced functionality development.

The classical paper by Broy [Bro06] describing the challenges in automotive software engineering is the next step to understanding the dynamics of automotive software engineering in general. This reading can be complemented by the paper by Pretschner et al. [PBKS07], where the focus is on the future of automotive software development.

Readers interested in the management of variability in general should explore the work of Bosch et al. [VGBS01, SVGB05] or [BFG+01]. The work is based on software product lines, but applies very well to the automotive sector. This can be complemented with more recent developments in this area—software ecosystems and their realization in the automotive sector [EG13, EB14].

Otto et al. [Ott12] and [Ott13] presents a study on requirements engineering at Mercedes-Benz, where they classified over 5800 requirement review protocols to their quality model. Their results showed that textual requirements (or natural language requirements as they are called in the publication) are prone to such problems as inconsistency, incompleteness and ambiguity—with about 70% of defects in requirements falling into these categories. In the light of this article we can see the need for complementing the textual requirements with more context, provided by use case models, user stories and use cases.

Törner et al. [TIPÖ06] presented a similar study but of the requirements at Volvo Car Group. In contrast to the study of Otto et al. [Ott12], these authors studied the use case specifications and not the textual requirements. The results, however, are similar, as the main types of defects are missing elements (correctness in Otto et al.'s model) and incorrect linguistics (ambiguity in Otto et al.'s model).

Eliasson et al. [EHKP15] described further experiences from Volvo Car Group where they explored challenges with requirements engineering at large in a mechatronics development organization. Their findings showed that there is a lot of communication in parallel to the requirements specification. The stakeholders in the requirements specification frequently mentioned the need to have a good network in order to specify the requirements correctly. This indicates the challenges described previously in this chapter that the requirements need more context than is usually provided in just the specification (especially the textual specification).

Mahally et al. [MMSB15] identified requirements to be the main barriers and enablers in moving towards Agile mechatronics organizations. Although today OEMs try to move towards fast development of mechatronics and reduce the cycle time by using Agile software development approaches, the challenges are that we

do not know upfront whether a requirement requires the development of electronics or is only a software requirement. According to Mahally et al. that kind of problem needs to be solved, and based on the prediction of Houdek [Hou13] this kind of issue might be coming to an end as device development flattens out and most of the requirements become software requirements. Similar challenges were presented by Pernstål et al. [PGFF13] who found that requirements engineering is one of the top improvement areas for automotive OEMs. The ability to communicate via requirements was also an important part.

At Audi, Allmann et al. [AWK+06] presented the challenges in the requirements communication on the boundary between the OEMs and their suppliers. They identified the need for better communication and the challenges of communicating through textual representations. They recognized the need for tighter partnerships as there is an inherent deficiency in communicating through requirements—transferring knowledge through an intermediate medium. Therefore they recommended integrating systems to minimize knowledge loss via transfer of documents.

Siegl et al. [SRH15] presented a method for formalizing requirements specifications using the Time Usage Model and applied it successfully to a requirements specification from one of the German OEMs. The evaluation study showed an increase in test coverage and increased quality of the requirements specification.

At BMW, Hardt et al. [HMB02] demonstrated the use of formalized domain engineering models in order to reason about the dependencies between requirements in the presence of variants. Their approach provided a simplistic, yet powerful, formalism and its strength was industrial applicability.

A study of the functional architecture of a car project at BMW and the requirements linked to the functions by Vogelsang and Fuhrmann [VF13] showed that 85% of functions are dependent on one another and that these dependencies cause a significant number of problems in software projects. This study shows the complexity of the functional decomposition of the vehicle's design and the complexity of its description.

At Bosch, the longitudinal study of a 5-year project by Langenfeld et al. [LPP16] showed that 61% of defects in requirements come from the incompleteness or incorrectness of the requirements specifications.

One of interesting trends in requirements engineering is the automatization of tasks of requirements engineers. One of such tasks is the discovery of non-functional requirements. This task is based on reading the specifications of functional requirements and identifying phrases which should transform into non-functional requirements. A study on the automation of this task has been conducted by Cleland-Huang et al. [CHSZS07]. The study showed that the automated classification of requirements could be as good as 90%, but at this stage cannot replace the manual classifiers.

3.7.1 Requirements Specification Languages

A model for requirements traceability [DPFL10] DARWIN4Req has been proposed to address the challenges related to the ability to follow the requirements' lifecycle. The model allows us to link requirements expressed in different formalities (e.g. UML, SysML) and connect them to one another. However, to the best of our knowledge, the model and the tool have not been adopted on a wider scale yet.

EAST-ADL [DSLT05] is an architecture specification language which contains elements to capture requirements and link them to the architectural design. The approach is similar to that of SysML but with the difference that there is no dedicated requirements specification diagram. EAST-ADL has been demonstrated to work in industry; however, it is not a standard for automotive OEMs yet. Mahmud [MSL15] presented a language ReSA that complements the EAST-ADL modelling language with the possibility to analyze and validate requirements (e.g. basic consistency checks).

For non-functional requirements in the domain of safety, Peraldi [PFA10] has proposed another extension of the EAST-ADL language which allows for increased traceability of requirements and their linking to non-functional properties of the designed embedded software (e.g. Safety).

Mellegård and Staron [MS09] and [MS10c] conducted an empirical study on the impact of using hierarchical graphical requirements specification on the quality of change impact assessment. For this purpose they designed a requirements' specification language based on the existing formalism—Requirements Abstraction Model. The results showed that the graphical overview of the dependencies between requirements introduces significant improvement [KS02].

Finally, the use of models as core artifacts in software development in the automotive domain has been studied in the context of MDA (Model-Driven Architecture) [SKW04a, SKW04b, SKW04c]. The important aspect is the evolution of models throughout the lifecycle.

3.8 Summary

Correct, unambiguous and consistent requirements specifications are foundations for high-quality software in general and in the automotive embedded systems in particular. In this chapter we introduced the most common types of requirements used in this domain and provided their main strengths.

Based on the current state of evolution of automotive software we could observe three trends in requirements engineering for automotive embedded systems—(1) agility in requirements specification, (2) increased focus on non-functional requirements and (3) increased focus on security as a domain for requirements. Towards the end of the chapter we also provided an overview of the requirements practices at some of the vehicle manufacturers (Mercedes Benz, Audi, BMW and

Volvo) based on documented experiences at these companies. We have also pointed out a number of directions for further reading for the interested.

In our future work we plan to review the requirements engineering practices in the main automotive OEMs and identify their differences and commonalities.

References

ASM+14. Vard Antinyan, Miroslaw Staron, Wilhelm Meding, Per Österström, Erik Wikstrom, Johan Wranker, Anders Henriksson, and Jörgen Hansson. Identifying risky areas of software code in agile/lean software development: An industrial experience report. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 154–163. IEEE, 2014.

AWK+06. Christian Allmann, Lydia Winkler, Thorsten Kölzow, et al. The requirements engineering gap in the oem-supplier relationship. *Journal of Universal Knowledge Management*, 1(2):103–111, 2006.

BFG+01. Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In *International Workshop on Software Product-Family Engineering*, pages 13–21. Springer, 2001.

Bro06. Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006.

C+90. IEEE Standards Coordinating Committee et al. IEEE Standard glossary of software engineering terminology (IEEE Std 610.12-1990). los alamitos. *CA: IEEE Computer Society*, 1990.

CHSZS07. Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. Automated classification of non-functional requirements. *Requirements Engineering*, 12(2):103–120, 2007.

CTS+06. DeJiu Chen, Martin Törngren, Jianlin Shi, Sebastien Gerard, Henrik Lönn, David Servat, Mikael Strömberg, and Karl-Erik Årzen. Model integration in the development of embedded control systems-a characterization of current research efforts. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1187–1193. IEEE, 2006.

DB15. Jan Dannenberg and Jan Burgard. 2015 car innovation: A comprehensive study on innovation in the automotive industry. 2015.

Dem. Simulink Demo. Modeling an anti-lock braking system.

DPFL10. Hubert Dubois, Marie-Agnès Peraldi-Frati, and Fadoi Lakhal. A model for requirements traceability in a heterogeneous model-based design process: Application to automotive embedded systems. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 233–242. IEEE, 2010.

DSLT05. Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST–ADL – An architecture description language. In *Architecture Description Languages*, pages 181–195. Springer, 2005.

EB14. Ulrik Eklund and Jan Bosch. Architecture for embedded open software ecosystems. *Journal of Systems and Software*, 92:128–142, 2014.

EG13. Ulrik Eklund and Håkan Gustavsson. Architecting automotive product lines: Industrial practice. *Science of Computer Programming*, 78(12):2347–2359, 2013.

EHKP15. Ulf Eliasson, Rogardt Heldal, Eric Knauss, and Patrizio Pelliccione. The need of complementing plan-driven requirements engineering with emerging communication: Experiences from Volvo Car Group. In *Requirements Engineering Conference (RE), 2015 IEEE 23rd International*, pages 372–381. IEEE, 2015.

HMB02. Markus Hardt, Rainer Mackenthun, and Jürgen Bielefeld. Integrating ECUs in vehicles-requirements engineering in series development. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 227–236. IEEE, 2002.

Hou13. Frank Houdek. Managing large scale specification projects. In *Requirements Engineering foundations for software quality, REFSQ*, 2013.

ISO11. ISO. 26262–road vehicles-functional safety. *International Standard ISO*, 26262, 2011.

JBR97. Ivar Jacobson, Grady Booch, and Jim Rumbaugh. The objectory software development process. *ISBN: 0-201-57169-2, Addison Wesley*, 1997.

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.

KSM+15. Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. Supporting continuous integration by code-churn based test selection. In *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, pages 19–25. IEEE Press, 2015.

LPP16. Vincent Langenfeld, Amalinda Post, and Andreas Podelski. Requirements Defects over a Project Lifetime: An Empirical Analysis of Defect Data from a 5-Year Automotive Project at Bosch. In *Requirements Engineering: Foundation for Software Quality*, pages 145–160. Springer, 2016.

MMSB15. Mahshad M Mahally, Miroslaw Staron, and Jan Bosch. Barriers and enablers for shortening software development lead-time in mechatronics organizations: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 1006–1009. ACM, 2015.

MS08. Niklas Mellegård and Miroslaw Staron. Methodology for requirements engineering in model-based projects for reactive automotive software. In *18th ECOOP Doctoral Symposium and PhD Student Workshop*, page 23, 2008.

MS09. Niklas Mellegård and Miroslaw Staron. A domain specific modelling language for specifying and visualizing requirements. In *The First International Workshop on Domain Engineering, DE@ CAiSE, Amsterdam*, 2009.

MS10a. Niklas Mellegård and Miroslaw Staron. Characterizing model usage in embedded software engineering: a case study. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 245–252. ACM, 2010.

MS10b. Niklas Mellegård and Miroslaw Staron. Distribution of effort among software development artefacts: An initial case study. In *Enterprise, Business-Process and Information Systems Modeling*, pages 234–246. Springer, 2010.

MS10c. Niklas Mellegård and Miroslaw Staron. Improving efficiency of change impact assessment using graphical requirement specifications: An experiment. In *Product-focused software process improvement*, pages 336–350. Springer, 2010.

MSL15. Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. ReSA: An ontology-based requirement specification language tailored to automotive systems. In *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pages 1–10. IEEE, 2015.

Ott12. Daniel Ott. Defects in natural language requirement specifications at Mercedes-Benz: An investigation using a combination of legacy data and expert opinion. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 291–296. IEEE, 2012.

Ott13. Daniel Ott. Automatic requirement categorization of large natural language specifications at Mercedes-Benz for review improvements. In *Requirements Engineering: Foundation for Software Quality*, pages 50–64. Springer, 2013.

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.

PFA10. Marie-Agnès Peraldi-Frati and Arnaud Albinet. Requirement traceability in safety critical systems. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 11–14. ACM, 2010.

PGFF13. Joakim Pernstål, Tony Gorschek, Robert Feldt, and Dan Florén. Software process improvement in inter-departmental development of software-intensive automotive systems – A case study. In *Product-Focused Software Process Improvement*, pages 93–107. Springer, 2013.

RSB+13a. Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Fredrik Törner. Improving fault injection in automotive model based development using fault bypass modeling. In *GI-Jahrestagung*, pages 2577–2591, 2013.

RSB+13b. Rakesh Rana, Miroslaw Staron, Claire Berger, Jorgen Hansson, Martin Nilsson, and Fredrik Torner. Evaluating long-term predictive power of standard reliability growth models on automotive systems. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 228–237. IEEE, 2013.

RSM+13. Rakesh Rana, Miroslaw Staron, Niklas Mellegård, Christian Berger, Jörgen Hansson, Martin Nilsson, and Fredrik Törner. Evaluation of standard reliability growth models in the context of automotive software systems. In *Product-Focused Software Process Improvement*, pages 324–329. Springer, 2013.

SHF+13. Miroslaw Staron, Jorgen Hansson, Robert Feldt, Anders Henriksson, Wilhelm Meding, Sven Nilsson, and Christoffer Hoglund. Measuring and visualizing code stability – A case study at three companies. In *Software Measurement and the 2013 Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013 Joint Conference of the 23rd International Workshop on*, pages 191–200. IEEE, 2013.

SKW04a. 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.

SKW04b. Miroslaw Staron, Ludwik Kuzniarz, and Ludwik Wallin. A case study on industrial MDA realization – Determinants of effectiveness. *Nordic Journal of Computing*, 11(3):254–278, 2004.

SKW04c. Miroslaw Staron, Ludwik Kuzniarz, and Ludwik Wallin. Factors determining effective realization of MDA in industry. In K. Koskimies, L. Kuzniarz, Johan Lilius, and Ivan Porres, editors, *2nd Nordic Workshop on the Unified Modeling Language*, volume 35, pages 79–91. Abo Akademi, 2004.

SRH15. Sebastian Siegl, Martin Russer, and Kai-Steffen Hielscher. Partitioning the requirements of embedded systems by input/output dependency analysis for compositional creation of parallel test models. In *Systems Conference (SysCon), 2015 9th Annual IEEE International*, pages 96–102. IEEE, 2015.

SVGB05. Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

SZ05. Jörg Schäuffele and Thomas Zurawka. *Automotive software engineering – Principles, processes, methods and tools*. 2005.

TIPÖ06. Fredrik Törner, Martin Ivarsson, Fredrik Pettersson, and Peter Öhman. Defects in automotive use cases. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 115–123. ACM, 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.

VGBS01. Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

WW02. Matthias Weber and Joachim Weisbrod. Requirements engineering in automotive development-experiences and challenges. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 331–340. IEEE, 2002.