Chapter 8
Evaluation of Automotive Software Architectures

Abstract In this chapter we introduce methods for assessing the quality of software architectures and we discuss one of the techniques—ATAM. We discuss the non-functional properties of automotive software and we review the methods used to assess such properties as dependability, robustness and reliability. We follow the ISO/IEC 25000 series of standards when discussing these properties. In this chapter we also address the challenges related to the integration of hardware and software and the impact of this integration. We review differences with stand-alone desktop applications and discuss examples of these differences. Towards the end of the chapter we discuss the need to measure these properties and introduce the need for software measurement.

8.1 Introduction

Having the architecture in place, as we discussed in Chap. 2, is a process which requires a number of steps and revisions of the architecture. As the evolution of the architecture is a natural step, it is often guided by some principles. In this chapter we look into aspects which drive the evolution of the architectures—non-functional requirements and architecture evaluation methods.

During this process the architects take a number of decisions about their architecture—starting from the basic one on what style should be used in which part of the architecture and ending in the one on the distribution of signals over the car's communication buses. All of these evaluations lead to a better or worse architecture and in this chapter we focus on the question that each software architect confronts—*How good is my architecture?*

Although the question is rather straightforward, the answer to it is rather complicated, because the answer to it depends on a number of factors. The major complication is related to the need to balance all of these factors. For example, the performance of the software needs to be balanced with the cost of the system, the extensibility needs to be balanced with the reliability and performance, etc. Since the size of the software system is often large the question whether the architecture

© Springer Nature Switzerland AG 2021

M. Staron, *Automotive Software Architectures*,
https://doi.org/10.1007/978-3-030-65939-4_8

is optimal, or even good enough, requires an organized way of evaluating the architecture.

In Chap. 3 we discussed the notion of a requirement as a customer demand on the functionality of the software and the need for the fulfillment of certain quality attributes. In this chapter we dive deeper into the question—*What quality attributes are important for the automotive software architectures?* and *How do we evaluate that an architecture fulfills these requirements?*

To answer the first question we review the newest software engineering standard in the area of product quality—ISO/IEC 25023 (Software Quality Requirements and Evaluation—Product Quality, [ISO16b]). We look into the construction of the standard and focus on how software quality is described in this standard, with the particular focus on product quality.

To answer the second question about the evaluation of architectures, we look into one of the techniques for evaluating quality of software architectures—Architecture Trade-off Analysis Method (ATAM), which is one of the many techniques for assessing quality of software architectures.

So, let us dive deeper into the question of what software quality is and how it is defined in modern software engineering standards.

8.2 ISO/IEC 25000 Quality Properties

One of the main standards in the area of software quality is the ISO/IEC 25000 series of standards—Software Quality Requirements and Evaluation (SQuaRE) [ISO16a]. The standard is an extension of the old standard in the same area—ISO/IEC 9126 [OC01]. Historically, the view of the software quality concept in ISO/IEC 9126 was divided into a number of sub-areas such as reliability or correctness. This view was found to be too restrictive as the quality needs to be related to the context of the product—its requirements, operating environment and measurement. Therefore, the new ISO/IEC 25000 series of standards is more extensive and has a modular architecture with a clear relation to other standards. An overview of the main quality attributes, grouped into quality characteristics, is presented in Fig. 8.1. The dotted line shows a characteristic which is not part of the ISO/IEC 25000 series, but another standard—ISO/IEC 26262 (Road Vehicles—Functional Safety).

These quality characteristics describe various aspects of software quality, such as whether it fulfills the functions described by the requirements correctly (functionality) and whether it is easy to maintain (maintainability). However, for safety-critical systems like the software system of a car, the most important part of the quality model is actually the reliability part, which defines the reliability of a software system, such as *Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time* [ISO16b].

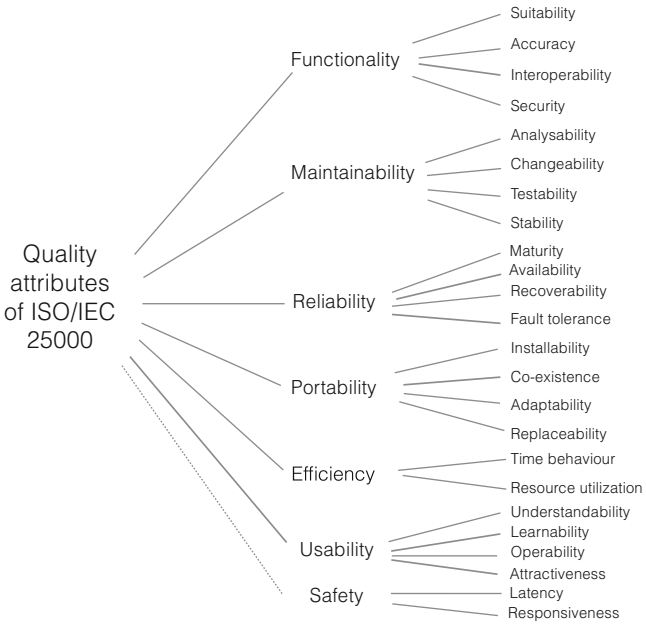


Fig. 8.1 ISO/IEC 25000 quality attributes

8.2.1 Reliability

Reliability of a software system in common understanding is the ability of the system to work according to the specification during a period of time [RSB+13]. This characteristic is important as car's computer system, including software, has to be in operation for years after its manufacturing. The ability to "reset" the car's computer system is very limited as it needs to operate constantly, controlling the powertrain, brakes, and safety mechanisms.

Reliability is a generic quality characteristics and contains four sub-characteristics as shown in Fig. 8.2—maturity, availability, recoverability and fault tolerance.

Maturity is defined as *degree to which a system, product or component meets needs for reliability under normal operation*. The concept defines how the software operates over time, i.e. how many failures the software has over time, which is often shown as a curve of the number of defects over time; see Fig. 8.2 from [RSM+13] and [RSB+16].

The figure shows that the number of faults discovered during the design and operation of the software system can have different shapes depending on the type of development, type of the functionality being developed and the time of the lifecycle of the software. The type of development (discussed in Chap. 3) determines how and when the software is tested and the testing determines the type of faults that are

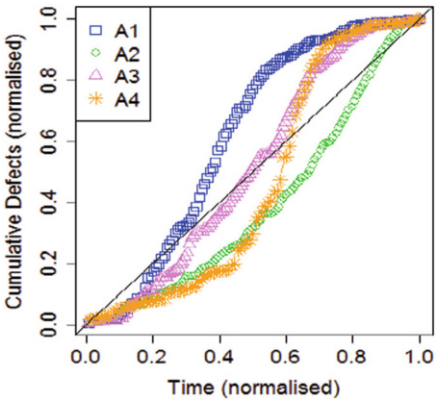


Fig. 8.2 Reliability growth of three different software systems in the automotive domain

discovered—e.g. the late testing phases often uncovers more severe defects, while the early testing phases can isolate simpler defects that can be fixed easily. Flattening of the curve towards the end of the development shows that the maturity of the system is higher as the number of defects found gets lower—the software is ready for its release and deployment.

Another sub-characteristic of reliability is the availability of the system, which is defined as *degree to which a system, product or component is operational and accessible when required for use*. The common sense of this definition is the ability of the system to be used when needed, which can be seen as a momentary property. High availability systems do not need to be available over time, all the time, but they need to be available when needed. This means that these systems can be restarted often and the property of "downtime" is not as important as for fault-tolerant systems which should be available all the time (e.g. 99.999% of the time, which is ca. 4 min of downtime per year).

Recoverability is defined as *Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system*. This quality property is often quoted in the research on self-* systems (e.g. self-healing, self-adaptive, self-managing) where the software itself can adjust its structure in order to recover from failure. In the automotive domain, however, this is still in the research phase as the mechanisms of self-* often should be formally proven that the transition between states is safe. The only exception is the ability of the system to restart itself, which has been used as a "last resort" mechanism for tackling failures.

Fault tolerance is defined as *degree to which a system, product or component operates as intended despite the presence of hardware or software faults*. This property is very important as the car's software consists of hundreds of software components distributed over tens of ECUs communicating over a few buses—something is bound to go wrong in this configuration. Therefore we discuss this property separately in the next section.

Fig. 8.3 Engine check control light indicating reduced performance of the powertrain, Volvo XC70

8.2.2 Fault Tolerance

Fault tolerance, or robustness is a concept of *the degree to which a computer system can operate in the presence of errors* [SM16]. Robustness is important as the software system of a car needs to operate, sometimes with reduced functionality, even if there are problems (or errors) during runtime.

A common manifestation of the robustness of the car is the ability to operate with reduced functionality when the diagnostics system indicates a problem with, for example, the powerline. In many modern cars the diagnostics system can detect problems with the exhaust system and reduce the power of the engine (degradation of the functionality), but still enable the operation of the car. The driver is only notified by a control lamp on the instrument panel as in Fig. 8.3.

As the figure shows, the software system (the diagnostics) has detected the problem and has taken action to allow the driver to continue the journey—which shows high robustness to failures.

8.2.3 Mechanisms to Achieve Reliability and Fault Tolerance

The traditional ways of achieving fault tolerance are often found on the lower levels of system design—hardware level. The ECUs used in the computer system can rely

on hardware redundancy and fail-safe takeover mechanisms in order to ensure the operation of the system in the presence of faulty component. However, this approach is often non-feasible in the car's software as the electrical system of the car cannot be duplicated and hardware redundancy is not possible. Instead, the designers of the software systems usually rely on substituting data from different sensors in order to obtain the same (or similar) information once one of the components fails.

One of the main mechanisms used in modern software is the mechanism of *graceful degradation*. Shelton and Koopman [SK03] define graceful degradation as *a measure of the system's ability to provide its specified functional and non-functional capabilities*. They show that a system that has all of its components functioning properly has maximum utility and "losing" one or more components leads to reduced functionality. They claim that "a system degrades gracefully if individual component failures reduce system utility proportionally to the severity of aggregate failures." For the architecture, this means that the following decisions need to be prioritized:

- No single point of failure—this means that no component should be exclusively dependent on the operation of another component. Service-oriented architectures and middleware architectures often do not have a single point of failure.
- Diagnosing the problems—the diagnostics of the car should be able to detect malfunctioning of the components, so mechanisms like heartbeat synchronization should be implemented. The layered architectures support the diagnostics functionality as they allow us to build two separate hierarchies—one for handling functionality and one for monitoring it.
- Timeouts instead of deadlocks—when waiting for data from another component, the component under operation should be able to abort its operation after a period of time (timeout) and signal to the diagnostics that there was a problem in the communication. Service-oriented architectures have built-in mechanisms for monitoring timeouts.

Prioritizing such decisions should lead to an architecture where a single failure in a component leaves the entire system operational and signals the need for manual intervention (e.g. workshop visit to replace a faulty component).

A design principle to achieve fault-tolerant software is to use programming mechanisms which reduce the risk of both design and runtime errors, such as:

- using static variables when programming—using static variables rather than variables allocated dynamically on the heap allows taking advantage of atomic write/read operations; when addressing a memory dynamically on the heap the read/write operation requires at least two steps (read the memory address, write/read to the address), which can pose threats when using multithreaded programs or interrupts.
- using safety bits for communication—any type of communication should include the so-called safety bits and checksums in order to prevent operation of software components based on faulty inputs and thus failure propagation.

The automotive industry has adopted the MISRA-C standard, where the details of the design of computer programs in C programming language [A+08], which has been discussed in more detail in the previous chapter.

However, since the architecture of the software is an artifact that is abstract and cannot be tested, the evaluation of the architecture needs to be done based on its description as a model and often manually.

8.3 Architecture Evaluation Methods

In our discussion of the quality of the system we highlighted the need to balance different quality characteristics against each other. This balancing needs to be evaluated and therefore we look into an example software architecture evaluation technique.

The goals behind evaluating architectures can differ from case to case, from the general understanding of the architectural principles to the exploration of specific risks related to software architectures. Let us explore what kinds of architecture analysis methods are the most popular today and why.

Techniques used for analysis of architectures, as surveyed by Olumofin [OM05]:

1. Failure Modes and Effects Analysis (FMEA)—a method to analyze software designs (including the architecture) from the perspective of risk of failures of the system. This method is one of the most generic ones and can come either in fully qualitative form (based on expert analysis) or as a combination of qualitative expert analysis and quantitative failure analysis using mathematical formulas for failure modelling.
2. Architecture Trade-off Analysis Method (ATAM)—a method to evaluate software architectures from the perspective of the quality goals of the system. ATAM, based on expert-based reviews of the architecture from the perspective of scenarios (more about it later in this chapter).
3. Software Architecture Analysis Method (SAAM)—a method which is seen as a precursor to ATAM is based on the evaluation of software architectures from the perspective of different types of modifiability, portability and extendability. This method has multiple variations, such as: SAAM Founded on Complex Scenarios (SAAMCS), Extending SAAM by Integration in the Domain (ESAAMI) and Software Architecture Analysis Method for Evolution and Reusability (SAAMER).
4. Architecture Level Modifiability Analysis (ALMA)—a method for evaluating the ability of the software architecture to withstand continuous modifications, [BLBvV04].

Fig. 8.4 Parking assistance camera showing the view behind the car while backing up, Volvo XC70

The above evaluation methods constitute an important method portfolio for software architects who need to make judgements about the architecture of the system before the system is actually implemented. It seems like a straightforward task, but in reality it requires skills and experience to be performed correctly.

An example of the need for skills and experiences is the evaluation of the performance of the system before it is implemented. When designing the software system in cars the performance of the communication channels is often a bottleneck—the bandwidth of the CAN bus is usually limited. Therefore adding new, bandwidth-greedy components and functions requires analysis of both the scenario of using the function in question and the entire system. A simple case is the function of providing a camera video feed from the back of the car when backing-up—used in the majority of premium segment cars today. Figure 8.4 shows this function on the instrument panel.

When adding the camera to the electrical system the amount of data transmitted from the back of the car to the front of the car increases dramatically (depending on the resolution of the camera, it could be up to 1Mbit/s). Since the data is to be transmitted in real time the communication bus must constantly prioritize between the video feed data and the signals from such sensors as parking assist sensors.

In this scenario the architects need to answer the question—will it be possible to add the camera component to the electrical system without jeopardizing such safety critical functions as park assist?

8.4 ATAM

ATAM has been designed as a response to the need of the American Department of Defense in the 1990s to be able to evaluate the quality of software systems in their early development stage (i.e. before the system is implemented). The origins of ATAM are at the Software Engineering Institute, in the publication of Kazman et al. [KKB+98]. The ATAM method, which can be used to answer this question is based on [KKC00]:

> The Architecture Tradeoff Analysis Method (ATAM) is a method for evaluating software architectures relative to quality attribute goals. ATAM evaluations expose architectural risks that potentially inhibit the achievement of an organization's business goals. The ATAM gets its name because it not only reveals how well an architecture satisfies particular quality goals, but it also provides insight into how those quality goals interact with each other and how they trade off against each other.

As stressed in the above definition, the method relates the system to its quality, i.e. non-functional requirements on its performance, availability, reliability (fault tolerance) and other quality characteristics of ISO/IEC 25000 (or any other quality model).

8.4.1 Steps of ATAM

ATAM is a stepwise method which is similar to reading techniques used in software inspections (e.g. perspective-based reading [LD97] or checklist-based reading [TRW03]). The steps are as follows (after [KKC00]).

Step 1: Present ATAM. In this step the architecture team presents the ATAM method to the stakeholders (architects, designers, testers and product managers). The presentation should explain the principles of the evaluation, evaluation scenarios and its goal (e.g. which quality characteristics should be prioritized).

Step 2: Present business drivers. After presenting the purpose of the evaluation, the purpose of the business behind this architecture is presented. Topics covered in this step should include: (1) the main functions of the system (e.g. new car functions), (2) the business drivers behind these functions and their optionality (e.g. which functions are to be included in all models and which should be optional), business case behind the architecture and its main principles (e.g. performance over extendability, maintainability over cost).

Step 3: Present architecture. The architecture should be presented in a sufficient level of detail to make the evaluation. The designers of the ATAM method do not propose a specific level of detail, but it is customary that the architects guide the reading of the architecture model—show where to start and where to stop reading the architecture model.

Step 4: Identify architectural approaches. In this step the architects introduce the architectural styles to the analysis team and present the high-level rationale behind these approaches.

Step 5: Generate quality attribute utility tree. In this step, the evaluation team constructs the system utility measure tree by combining the relevant quality factors, specified with scenarios, stimuli and responses.

Step 6: Analyze architectural approaches. This is the actual evaluation step where the evaluation team explores the architecture by studying the prioritized scenarios from step 5 and architectural approaches which address these scenarios and their corresponding quality characteristics. This step results in identifying architectural risks, sensitivity points, and tradeoff points.

Step 7: Brainstorm and prioritize scenarios. After the initial analysis of the architectural approaches is done, there is a lot of scenarios and sensitivity points elicited from the evaluation team. Therefore they need to be prioritized to guide the further analysis of the architecture. The 100 dollar technique, planning game and analytical-hierarchy-process are useful prioritization techniques at this stage.

Step 8: Analyze architectural approaches. In this step the team reiterates the analysis from step 6 with a focus on the highly prioritized scenarios from step 7. The result is again the list of risks, sensitivity points and trade-off points.

Step 9: Present results. After the analysis the team compiles and presents a report about the found risks, sensitivity points, non-risks and tradeoffs in the architecture.

The results of the analysis can only be as good as the input to the analysis, i.e. the quality of the architecture documentation (its completeness and correctness), the quality of the scenarios, the templates used in the analysis and the experience of the evaluation team.

8.4.2 Scenarios Used in ATAM in Automotive

ATAM is an extensible method which allows us to identify scenarios by the evaluation team, which is strongly encouraged. In this chapter we present a set of inspirational scenarios to guide the evaluation team. Our example set is based on the example set of scenarios presented by Bass et al. [BM+01] and in this chapter we present a set of scenarios important for the evaluation of automotive software. We present them in generic terms and in compact textual format. We group them according to quality characteristics, following the approach presented by Bass et al.

8.4.2.1 Modifiability

We start with the set of scenarios which date back to the origins of ATAM and address one of the main challenges for the work of the software architects—How extendable and modifiable is our architectural design?

It is worth noting that some of the scenarios impact the design (or the internal quality) of the product and some impact the external quality. The modifiability scenarios impact the internal quality of the product.

Scenario 1: A request arrives to change the functionality of the system. The change can be to add new functionality, to modify existing functionality, or to delete functionality [BM+01].

Scenario 2: A request arrives to change one of the components (e.g. because of a technology shift); the scenario needs to consider the change propagation to the other components.

Scenario 3: Customer wants different systems with different capabilities but using the same software and therefore advanced variability has to be built into the system [BM+01].

Scenario 4: New emission laws: the constantly changing environmental laws require adaptation of the system to decrease its environmental impact [BM+01].

Scenario 5: Simpler engine models: Replace the engine models in the software with simple heuristics for the low-cost market [BM+01].

Scenario 6: An additional ECU is added to the vehicle's network and causes new messages to be sent through the existing network. In the scenario we need to understand how the new messages impact the performance of the entire system.

Scenario 7: An existing ECU after the update adds a new message type: same messages but with additional fields that we are currently not set up to handle (based on [BM+01]).

Scenario 8: A new AUTOSAR version is adopted and requires update of the base software. We need to understand the impact of the new version in terms of the number of required modifications to the existing components.

Scenario 9: Reduce memory: During development of an engine control, the customer demands we reduce costs by downsizing the flash-ROM on chip (adapted from [BM+01]). We need to understand what the impact of this reduction is on the system performance.

Scenario 10: Continuous actuator: Changing two-point (on/off) actuators to continuous actuators within 1 month (e.g., for the EGR or purge control valve). We need to understand the impact of this change on the behavior of our models [BM+01].

Scenario 11: Multiple engine types in one car need to coexist: hybrid engine. We need to understand how to adapt the electrical system and isolate the safety-critical functions from the non-safety-critical ones.

8.4.2.2 Availability and Reliability

Availability and reliability scenarios impact the external quality of the product—allow us to reason about the potential defects which come from unfulfilled performance requirements (non-functional requirements).

Scenario 12: A failure occurs and the system notifies the user; the system may continue to perform in a degraded manner. What graceful degradation mechanisms exist? (based on [BM+01]).

Scenario 13: Detect software errors existing in third-party or COTS software integrated into the system to perform safety analysis [BM+01].

8.4.2.3 Performance

Performance scenarios also impact the external quality of the product and allow us to reason about the ability of the system to fulfill performance requirements.

Scenario 14: Start the car and have the system active in 5 s (adapted from [BM+01]).

Scenario 15: An event is initiated with resource demands specified and the event must be completed within a given time interval [BM+01].

Scenario 16: Using all sensors at the same time creates congestion and this causes loss of safety-critical signals.

8.4.2.4 Developing Custom Scenarios

It is natural that during an ATAM assessment the assessment group combines standard scenarios with custom ones. The literature about ATAM encourages us to create custom scenarios and use them in the evaluations, and therefore a few key points emerge which can help the development of scenarios.

Scenarios should be relevant to both the quality model's chosen/prioritized quality attributes and the business model of the company. It is important that the evaluation of the architecture be done in order to ensure that it fulfills the boundaries of product development. The BAPO model (Business Architecture Process and Organization, [LSR07]) from the evaluation of product lines can be used to make the link.

The criteria applied for the scenarios should be clear to the assessment team and the organization. It is important that all stakeholders understand what "good", "wrong", "insufficient", and "enough" mean in the evaluation situation. It is all too easy to get stuck in a detailed discussion of mechanisms used in the evaluation without the good support of measures or checklists.

When defining custom scenarios we can get help of the table with the elements presented in Fig. 8.5.

Aspect	Value
Source	The description of which architectural element initiates the scenario.
Stimulus	The stimulus signal or component of the scenario.
Artifact	Architectural elements which are affected by the scenario.
Environment	Description of the environment when this stimulus appears.
Response	Description of the expected outcome observed after the received stimulus.
Measure	Quantifiable measures that could help if the scenario is successful.

Fig. 8.5 Template for defining custom scenarios

Scenario ID	Unique ID of the scenario to identify it, later on used to link the scenario to the quality characteristics, and requirements
Stimulus	The stimulus in the scenario, i.e. what kind of event or activity of interest in the scenario. For example: Adding a new rear-view camera to the main CAN bus.
Response	The outcome of interest in the scenario. For example: Causes the congestion of signals on the bus and loss of safety critical signals from the parking assist sensors.
Requirement	The link of the scenario to the requirement(s) of the architecture, its performance or other non-functional characteristics.
Quality characteristics	The link of the scenario to one of the quality characteristics, e.g. modifiability, safety.
Textual version (optional)	Combining the stimulus and response into one sentence. For example: *"Adding a new rear-view camera to the main CAN bus can cause the congestion of signals on the bus and thus loss of safety-critical signals."*

Fig. 8.6 Template for the description of a scenario in ATAM

8.4.3 Templates Used in the ATAM Evaluation

The first template which is needed in the ATAM evaluation is the template to specify the scenarios. An example scenario template is presented in Fig. 8.6.

One of the templates, needed after the ATAM evaluation is completed, is the risk description template, which should be included in the results and their presentation. An example template is presented in Fig. 8.7.

Risk ID	Unique ID for the identification of the risk
Description	Detailed description of the risk, including the source of the risk.
Source / Sensitivity point	The description of the source of the risk. This field should include the reference to the element of the architecture which is the source of the risk in question. A detailed reference is important as it is needed for the assessment of the safety of the software system.
Impact	The description of the impact of the risk on the scenario, the quality characteristics of the system and ultimately the user of the system. For the risks related to the safety-critical functions of the system (e.g. when the ASIL level D is assigned to the source component), this impact should be related to the appropriate ASIL level requirements.
Severity	Severity of the risk, usually on the scale 1-5 from the least severe to critical.
Probability	The probability that this risk will manifest itself in the runtime system, usually on the scale 1-5 from the very unlikely to certain.

Fig. 8.7 Template for the description of risks found in ATAM

Another part of the results from ATAM is the set of sensitivity points which have been found in the architecture. A sensitivity point is defined by the Software Engineering Institute as

> a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response. Sensitivity points are places in a specific architecture to which a specific response measure is particularly sensitive (that is, a little change is likely to have a large effect). Unlike tactics, sensitivity points are properties of a specific system

A tradeoff template is presented in Fig. 8.8.

8.5 Example of Applying ATAM

Now that we have reviewed the elements of ATAM and its process, let us illustrate ATAM analysis using the example of placing the functionality related to a rear-view camera on the back bumper of the car. As we have just introduced ATAM in this chapter, let us start with the introduction of the business drivers.

Tradeoff ID	The ID of the tradeoff.
Quality characteristic 1	The first characteristic which is taking part of the trade-off.
Quality characteristic 2	The second characteristic which is taking part of the trade-off.
Sensitivity point	The sensitivity point in the software architecture where the trade-off decision takes place.
Tradeoff description	The description of the rationale and reasoning behind the trade-off. Here, the evaluation team should describe why this is trade-off identified and how the changes in the architecture to address one of the quality characteristics affect the other one.

Fig. 8.8 Template for the description of trade-offs identified after the ATAM analysis

8.5.1 Presentation of Business Drivers

The major business driver in this architecture is achieving a high degree of safety.

8.5.2 Presentation of the Architecture

First, let us present the function architecture of the car in Fig. 8.9.

Since we focus on camera functionality, we only include the major functions from the domains of active safety and infotainment. The functions presented in the figure represent the basic functions of braking and ABS in the active safety domain and the displaying of information on screens (both the main screen and the head-up display HUD).

Let us now introduce the simplistic architecture of the car's electrical system—i.e. the physical view of the architecture. The physical view is presented in Fig. 8.10.

In the example architecture we have two buses:

- CAN bus: connecting the ECUs related to the infotainment domain.
- Flexray bus: connecting the ECUs related to the safety domain and the chassi domain

We can also see the following ECUs :

- Main ECU: the main computer of the car, controlling the configuration of the car, initialization of the electronics and diagnostics of the entire system. The main ECU has the most powerful computing unit in the car, with the largest memory (in our example).

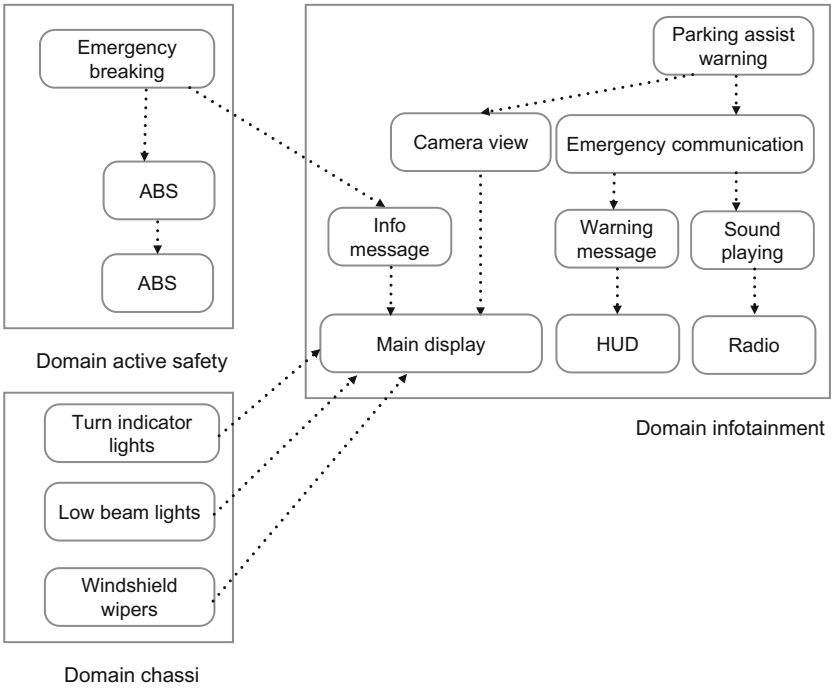


Fig. 8.9 Function dependencies in the architecture in our example

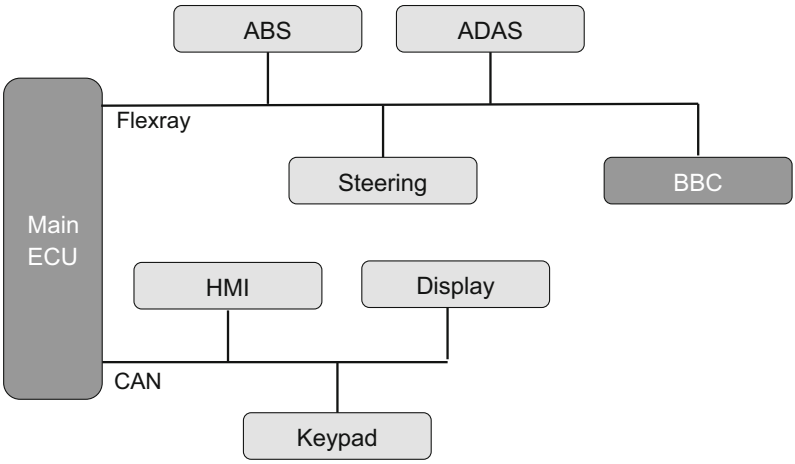


Fig. 8.10 Physical view of the architecture in our example

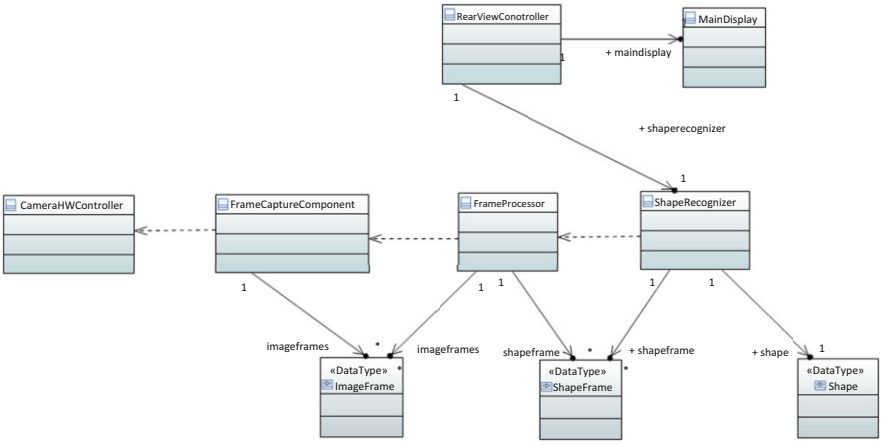


Fig. 8.11 Logical view of the architecture in our example

- ABS (Anti-locking Brake System): the control unit responsible for the braking system and the related functionality; it is a highly safety-critical unit, with only the highest safety integrity level software.
- ADAS (Advanced Driver Assistance and Support): the control unit responsible for higher-level decisions regarding active safety, such as collision avoidance by braking, emergency braking and skid prevention; it is also responsible for such functions as parking assistance.
- Steering: the control unit responsible for the steering functionality such as the electrical servo; it is also the controller of parts of the functions or parking assistant.
- BBC (Back Body Controller): the unit responsible for controlling non-safety critical functions related to the back of the car, such as adjusting of anti-dim lights, turning on and off of blinkers (back), and electrical opening of the trunk.

In the logical view of the architecture we focus on showing the main components used in the display of information and its processing from the camera unit, as we need them to perform the architecture analysis. Now let us introduce the logical architecture of the system in Fig. 8.11.

And finally let us show the potential deployment alternative of the architecture, where the majority of the processing takes place in the BBC node—as we can see in Fig. 8.12.

8.5.3 Identification of Architectural Approaches

In this example let us focus on the deployment of software components on the target ECUs. We also say that the physical architecture (hardware) does not change

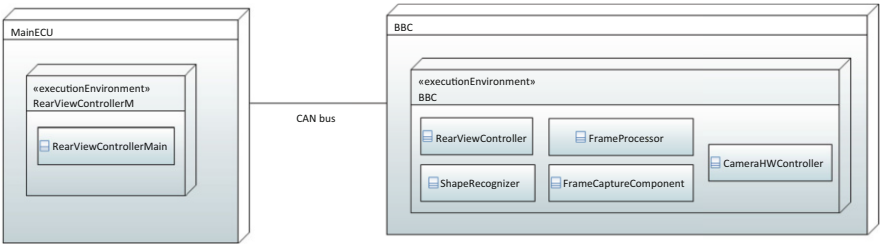


Fig. 8.12 The first deployment alternative in our example

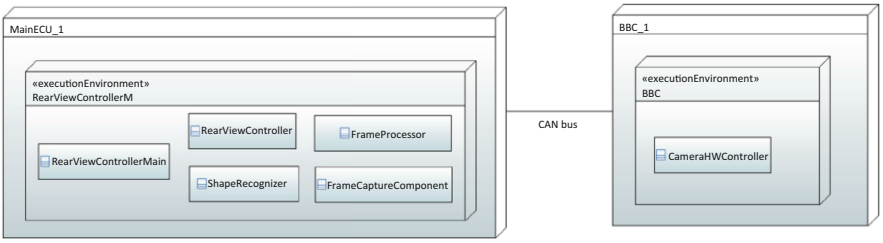


Fig. 8.13 The second deployment alternative in our example

and therefore we analyze the software aspects of the car's electrical system. As an alternative approach let us consider deploying all the processes on the main ECU instead of dividing the components between the Main ECU and the BBC. This results in the deployment as shown in Fig. 8.13. The dominant architectural style is pipes-and-filters as the processing of images is the main functionality here. The car's electrical system should support the advanced mechanisms of active safety (i.e. controlled by software) and should ensure that none of the mechanisms interfere with another one, jeopardizing safety.

In our subsequent considerations we look into these two alternatives and decide which one should be chosen to support the desired quality goals—i.e. what decision the architect should take given his quality attribute tree.

8.5.4 Generation of Quality Attribute Tree and Scenario Identification

In this example let us consider two scenarios which complement each other. We could naturally generate many more for each of the quality attributes presented earlier in this chapter, but we focus on the safety attribute—a scenario where there is congestion on the CAN bus when reverse driving and using a camera, and a scenario where we overload the main ECU when the video feed computations can interfere with other functions such as the operation of windshield wipers and low beam lights. We can use the scenario description template to outline the scenario in Fig. 8.14.

Aspect	Value
Source	Rear camera.
Stimulus	Camera feed.
Artifact	Main ECU, BBC ECU, CAN Bus.
Environment	Car in reverse driving.
Response	Process video data and show it on the display.
Measure	Video displayed in real time and no loss of safety signals from the parking sensors.

Fig. 8.14 Scenario described with its stimulus, response, environment and measure

Scenario ID	SC1: Congestion on the bus during reverse driving prevents safety-critical signals from reaching their destination.
Stimulus	The scenario is that during the reverse driving (backing up) of the car the video feed from the rear camera uses too much of the capacity and the communication bus is not able to relay (send) signals from the parking sensors. The main question to evaluate in this scenario is what kind of software deployment has the lowest influence on the safety of the car's software?
Response	• Analysis of the potential congestion for two architecture deployments. • List of constraints on the functionality for each of the solutions.
Requirement	"The architecture should allow the safety critical signals to be sent/received at any given point of time."
Quality characteristics	Safety: in this scenario we need to know that the particular architecture of the software does not cause congestions on buses and potential loss of signals.
Textual version (optional)	*When reversing the car, the video feed from the camera can reduce the ability of the parking sensors to send signals to the main ECU and therefore do not warn the driver about the potential collision.*

Fig. 8.15 Scenario of congestion on the communication bus

Let us also fully describe the first scenario as presented in Fig. 8.15.

In this scenario we are interested in the safety aspect of the reverse camera. We need to understand what kind of implications the video feed data transfer has on the capacity of the CAN bus which connects the BBC computer with the main ECU. We therefore need to consider both alternative architectural decisions—deployment of the video processing functionality on the BBC and the main ECU. We assume

Scenario ID	SC2: Overloading of the main processor during heavy weather conditions reduces the quality of the video feed.
Stimulus	The scenario is that during the heavy rain/snow condition where the main ECU is responsible for steering the windshield wipers, operating the lights and processing the video feed, the processing power of the ECU might not be enough to cope with all calculations The main question to evaluate in this scenario is, what kind of software deployment has the lowest influence on the performance of the car's software?
Response	• Analysis of the potential processing power for two architecture deployments. • List of constraints on the functionality for each of the solutions.
Requirement	"The car should provide the video feed from the rear-view camera during reverse driving in all weather conditions."
Quality characteristics	Performance: in this scenario we need to know that the particular architecture of the software does not cause overload of the computers and thus reduce the quality of the video feed.
Textual version (optional)	*When reversing in heavy weather conditions, the car's ECUs might be overloaded with computations and therefore not be able to handle all calculations related to the video feed processing.*

Fig. 8.16 Scenario of overloading of the main ECU

that none of the deployments result in adding new hardware and therefore do not influence the performance of the electrical system as a whole.[1]

We also can identify a scenario which is complementary to this one—see Fig. 8.16.

The reason for including both scenarios is the fact that they illustrate different possibilities of reasoning about deployment of functionality on nodes.

The quality attribute utility tree in our case consists of these two scenarios linked to two attributes—performance and safety. Both of these scenarios are ranked as high (H) in the utility tree, as shown in Fig. 8.17.

Now that we have the utility tree let us analyze the two architecture scenarios, and describe the trade-offs and sensitivity points.

[1]This assumption simplifies the analysis as we do not need to consider the physical architecture, but can focus only on the logical and deployment views of the architecture.

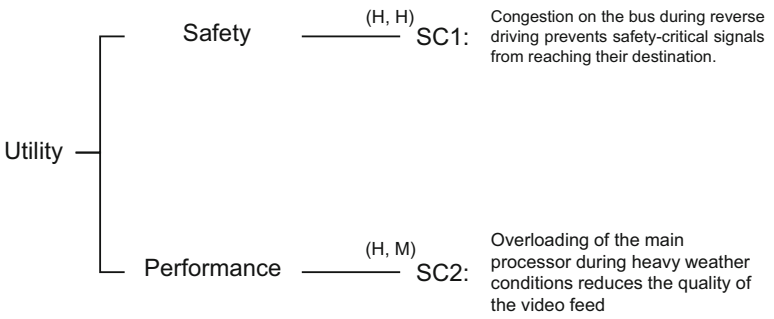


Fig. 8.17 Quality attribute utility tree

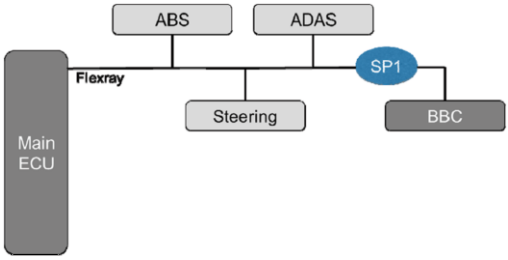
Risk ID	R1_S1
Description	The signals from the parking sensors cannot be transmitted over the bus. This causes the risk that the car does not stop before an obstacle and causes a collision.
Source / Sensitivity point	The sensitivity point is the Flexraybus between BBC and Main ECU as in the figure (SP1). 
Impact	ASIL C requirement: RQ1: The can should stop when detecting an obstacle in the range of 20 cm or less from the car. The impact on the user is that the car does not stop and therefore causes damage to property. It could also cause mild damage to the health of the passengers.
Severity	3
Probability	5 – it is very likely that during the reverse the safety signals and camera video feed coexist

Fig. 8.18 Risk description

8.5.5 Analysis of the Architecture and the Architectural Decision

Now we can analyze the architecture and its two deployments. In this analysis we can use a number of risks, for example the risk that the signal does not reach its destination. We can describe the risk using the template described in this chapter. The description is presented in Fig. 8.18.

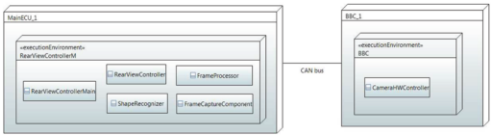
Scenario 5	Capture video during the reverse driving (backing up) of the car from the rear-camera and show it on the main display.		
Attributes	Safety.		
Environment	Car in reverse driving.		
Stimulus	Camera feed to be shown on the display.		
Response	Process video data and show it on the display.		
Architectural decisions	Sensitivity	Trade-off	Risk
Placing the processing of the video feed on the Main ECU	S1	T1	R1
Placing the processing of the video feed on BBC		T2	R2
Reasoning	The functioning of the main ECU is vital to the system (see sensitivity point S1) Safety versus lowered cost (see trade-off point T1) Safety requirement might be at risk due to heavy processing on Main ECU (see risk R1)		
Architecture diagram			

Fig. 8.19 Tabular summary of the example ATAM evaluation

Since the risk presented in Fig. 8.18 affects the safety of the passengers, it should be reduced. Reduction of this risk means that communication over the bus should not affect the safety-critical signals. Therefore the architectural decision is that priority should be given the deployment alternative 1—i.e. placing the processing of the video feed on the BBC ECU rather than on the main ECU.

The alternative means that the BBC ECU should have sufficient processing power to process the video in real time, which may increase the cost of the electrical components in the car. However, safety can allow the company to pursue its main business model (as described by the business drivers) and therefore balance the increased cost with increased sales of cars.

8.5.6 Summary of the Example

In this example we presented a simple assessment of a part of the software architecture for a car. The intention of this example is to provide an insight on how to think and reason when conducting such an assessment. In practice, the main purpose of an assessment like this one is all the discussions and presentations conducted by the assessment and the architecture teams. The questions, scenarios, prioritizations, and simply, brainstorming of ideas are the main point and benefit of the architecture. We summarize them in table presented in Fig. 8.19.

The ATAM procedure is defined for software architectures, but in the automotive domain the deployments of the software components and physical hardware architectures are tightly connected to the software—they both influence the software architecture and are influenced by the architecture (as this example assessment shows). Therefore, our advice is to always broaden the assessment team to include both software specialists and the hardware specialists—to cover the system properties of software architectures.

8.6 Further Reading

An interesting overview of scenario-based software architecture evaluation methods has been presented by Ionita et al. [IHO02]. Readers interested in a comparison between the methods are directed to this interesting article.

This article can be complemented by the work of Dobrica and Niemela [DN02], which focused on a more general overview and comparison of architecture evaluation methods.

A comprehensive work on the notion of graceful degradation has been presented by Shelton [She03, SK03] who discusses the notion of graceful degradation in the context of an example safety-critical system of an elevator, its modelling and measurement.

Readers interested in a wider view of the applicability of ATAM in other domains can look into the work of Bass et al. [BM+01], who analyzed the architecture evaluation scenarios of a number of safety-critical systems.

The original works of Bass and Kazman have been expanded to other domains and other quality attributes than the original few (modifiability, reliability, availability). An example of such extensions is presented by Govseva et al. [GPT01] and Folmer and Bosch [FB04].

In the automotive domain we often consider different car models as product lines with the equipment levels as product line members. For this kind of view on automotive software architectures one could find the extension of ATAM to capture product lines to be interesting [OM05].

Readers interested in further examples of architecture evaluations can be found in the article by Bergey et al. [BFJK99], who describe the experiences of using ATAM in the context of software acquisitions. The readers can also consider the work of Barbacci et al. [BCL+03].

8.7 Summary

Architecting is a discipline of high-level design which is often described in the form of diagrams. However, equally important to the design is the set of decisions taken when creating the architecture. These decisions delineate a set of principles which

designers have to follow in order to make sure that the software system fulfills its purpose.

Arriving at the right decisions is a process of combining the expertise of architects and the considerations of architects and designers. In this chapter we presented a method to elicit architectural decisions based on discussions between an external evaluation team and the architecture team—ATAM (Architecture Tradeoff Analysis Method). Through the assessments we can learn about the principles behind the architectural design and design decisions. We can learn about the alternative choices and why they are rejected.

In this chapter we focus on the "human" aspects of software architecture evaluation, which is by definition bound to be subjective to a certain degree. In the next chapter, however, we focus on the monitoring of the architecture quality given the set of information needs. This monitoring is done by conducting measurements and quantifying quality attributes discussed in this chapter.

References

A+08. Motor Industry Software Reliability Association et al. *MISRA-C: 2004: guidelines for the use of the C language in critical systems*. MIRA, 2008.

BCL+03. Mario Barbacci, Paul C Clements, Anthony Lattanze, Linda Northrop, and William Wood. Using the architecture tradeoff analysis method (ATAM) to evaluate the software architecture for a product line of avionics systems: A case study. 2003.

BFJK99. John K Bergey, Matthew J Fisher, Lawrence G Jones, and Rick Kazman. Software architecture evaluation with ATAM in the DoD system acquisition context. Technical report, DTIC Document, 1999.

BLBvV04. PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1):129–147, 2004.

BM+01. Len Bass, Gabriel Moreno, et al. Applicability of general scenarios to the architecture tradeoff analysis method. Technical report, DTIC Document, 2001.

DN02. Liliana Dobrica and Eila Niemela. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.

FB04. Eelke Folmer and Jan Bosch. Architecting for usability: a survey. *Journal of systems and software*, 70(1):61–78, 2004.

GPT01. Katerina Goševa-Popstojanova and Kishor S Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2):179–204, 2001.

IHO02. Mugurel T Ionita, Dieter K Hammer, and Henk Obbink. Scenario-based software architecture evaluation methods: An overview. *Icse/Sara*, 2002.

ISO16a. ISO/IEC. ISO/IEC 25000 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE). Technical report, 2016.

ISO16b. ISO/IEC. ISO/IEC 25023 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality. Technical report, 2016.

KKB+98. Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The architecture tradeoff analysis method. In *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*, pages 68–78. IEEE, 1998.

KKC00. Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for architecture evaluation. Technical report, DTIC Document, 2000.

LD97. Oliver Laitenberger and Jean-Marc DeBaud. Perspective-based reading of code documents at Robert Bosch GmbH. *Information and Software Technology*, 39(11):781–791, 1997.

LSR07. Frank Linden, Klaus Schmid, and Eelco Rommes. The product line engineering approach. *Software Product Lines in Action*, pages 3–20, 2007.

OC01. International Standard Organization and International Electrotechnical Commission. ISO IEC 9126, software engineering, product quality part: 1 quality model. Technical report, International Standard Organization/International Electrotechnical Commission, 2001.

OM05. Femi G Olumofin and Vojislav B Misic. Extending the ATAM architecture evaluation to product line architectures. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 45–56. IEEE, 2005.

RSB+13. Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen 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.

RSB+16. Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Wilhelm Meding. Analyzing defect inflow distribution and applying Bayesian inference method for software defect prediction in large software projects. *Journal of Systems and Software*, 117:229–244, 2016.

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.

She03. Charles Preston Shelton. *Scalable graceful degradation for distributed embedded systems*. PhD thesis, Carnegie Mellon University, 2003.

SK03. Charles Shelton and Philip Koopman. Using architectural properties to model and measure graceful degradation. In *Architecting dependable systems*, pages 267–289. Springer, 2003.

SM16. Miroslaw Staron and Wilhelm Meding. Mesram–a method for assessing robustness of measurement programs in large software development organizations and its industrial evaluation. *Journal of Systems and Software*, 113:76–100, 2016.

TRW03. Thomas Thelin, Per Runeson, and Claes Wohlin. An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering*, 29(8):687–704, 2003.