PRACE NAUKOWE POLITECHNIKI WARSZAWSKIEJ
z. 187 Elektronika 2013

Andrzej Zalewski

Instytut Automatyki i Informatyki Stosowanej

MODELLING AND EVALUATION OF SOFTWARE ARCHITECTURES

Rękopis dostarczono 16.04.2013 r.

Software architecture is an important development artefact, with substantial influence over the quality of a software system. This monograph presents the state of the art in modelling and evaluating software architectures, which are two closely related research areas influencing each other. Three main approaches to architectural modelling have been covered, i.e. models of software structure, architectural decisions, and models of architecture description. Semi-formal models, such as block diagrams models, UML, SysML and Archimate, are mainly used for modelling software structure. Architectural decisions capture the rationale underlying a given architectural design and the logic of the architecting process. The most important models for documenting architectural decisions have been discussed and compared: textual models, a comprehensive, flagship model by Zimmerman et al. extended with decision-making support, as well as the author's Maps of Architectural Decisions model, which has been tailored to the needs of documenting the evolution of rapidly and unpredictably evolving systems. Architectural patterns and tactics, which are closely related to architectural decisions, are also covered by this survey. The System Organisation Pattern is the author's proposition for the effective representation of top-level architecture of large-scale distributed systems, combining concepts of architectural patterns and architectural decisions. The models of architectural description focus on organising architectural information according to the stakeholders' concerns, captured by viewpoints. The monograph covers the most important developments in this area, i.e. ISO 42010:2011 standards, Kruchten's 4+1 views, Zachman's framework and recent developments regarding viewpoints. Architecture evaluation methods have evolved alongside architectural modelling. A new taxonomy of architecture evaluation methods, based on the method's applicability has been introduced, and two basic paradigms of architecture evaluation have been identified. Eighteen state-of-the-art architecture evaluation methods have been characterised according to a uniform description scheme. The Early Architecture Evaluation Methods, developed for the evaluation of large-scale system architectures at the inception stage of development, being the author's contribution to the research on architectural evaluation, was included in this survey. Such a comprehensive survey of architecture evaluation methods enabled the state of the art to be analysed, and a further research outlook to be drawn up.

1. INTRODUCTION

Software architecture has become a primary field of software engineering research in the recent twenty years. The wave of research on software architecture, which has intensified since the end of 20th century, can be perceived as a desire for yet another “silver bullet” [31] targeted at the growing complexity of software systems being developed nowadays. Its proposition can be summarised as:

- increasing the level of abstraction at which software systems are modelled and designed, by focusing the design activities on software structures at various levels of detail;
- analysing how these structures shape the properties of a software system, which should provide feedback on the quality of software design.

These ideas are naturally not entirely new, as systems structures and the properties that result from them have already been studied in general in the systems science of the 1960s and 70s (compare e.g. [136], [3], [68],) and in later work on modularisation (e.g. [9], [10]). Similarly, software structures and their influence on its properties have been studied since the advent of this discipline – compare, for example, [47], [48], [116], [114]. The legacy concepts of component (module) coupling and cohesion [161], [113] have become basic measures of the influence of software architecture on its maintainability and are still in use.

We can see that programming paradigms and software development methodologies (structured [162], [161] object-oriented [24], and service-oriented [5]) address the inherently “architectural” issue of software decomposition, i.e. define decomposition units (functions, classes/objects, services), ways of composing them into aggregate ones (hierarchies of functions, class inheritance, service orchestration), mechanisms of data exchange between software units (function invocation, message passing), and other means of organising software (source code modules, packages).

The concept of software architecture itself was born together with software engineering in the late 1960s [108, p. 22], [121, p. 9], [156]. Currently, there are numerous definitions of ‘software architecture’, emphasising its various aspects, e.g. [118], [60], [15] ISO/IEC/IEEE 24765:2010(E) [75], ISO/IEC/IEEE 42010:2011 [77]. Putting all of them together reveals that ‘software architecture’ comprises three basic elements:

- Structure – the building elements and their structure (relationships), rules governing the organisation, design and evolution of a software system;
- Behaviour – the logic of co-operation between the components;
- Motivation – the design intent and its rationale that shaped both structure and behaviour.

The importance of software architecture is that it may potentially influence all the non-functional quality attributes, or even place the development project in jeopardy. The influence of software architecture is felt throughout the entire lifecycle of the system. For example: over-complex software design can make changes very difficult and costly. As architecture defines an overall organisation of a software system, its changes require a substantial effort and may involve serious risks. Therefore, software architecture, although invisible, is a valuable asset.

The development of a software architecture requires architecture models to represent design concepts and evaluation methods that can be used in order to assess whether software architecture sufficiently supports significant requirements. The research on software architecture has been addressing both these concerns. This monograph focuses on architecture evaluation; however, the architecture has to be represented in some way in order to enable an analysis of its properties. Architectural models determine the scope of available analyses. Therefore, a survey of modelling approaches for software architecture has also been included. The book covers the following topics:

1. Software architecture modelling (chapter 2), which has been divided into three main areas: Architecture Modelling Languages (ADLs, section 2.1), architectural decisions (section 2.2) and models of an architectural description (section 2.3). The state of art in architectural modelling is discussed in section 2.4. The general conclusion of this chapter is that architectural modelling is dominated by informal and semiformal models. At the same time, the importance of aggregate concepts such as architectural styles, patterns or tactics for documenting and analysing architectures has also been emphasised;
2. Architecture Evaluation Methods (chapter 3) – this chapter comprises three main parts:
   a. sections 3.1–3.4 introduce basic concepts in order to prepare the ground for a precise and consistent presentation of architecture evaluation methods (sections 3.5–3.9). This includes: presentation of the paradigms of architectural evaluation (section 3.1), introduction of a taxonomy of architecture evaluation methods (sections 3.2 and 3.3) and definition of a uniform template for a description of architecture evaluation methods (section 3.4);
   b. sections 3.5–3.9 contain a survey of architecture evaluation methods, where the architecture evaluation methods have been grouped according to the taxonomy introduced in section 3.4;
   c. sections 3.10–3.11 are devoted to a discussion of the research achievements in the area of architectural evaluations and drafting an outlook for further research.

The uniform survey of all the state-of-the-art architecture evaluation methods is a unique value delivered by this monograph, because the results of research on architecture evaluation dispersed among many publications have been gathered, presented and discussed in a single work.

The author's original contribution to the field of software architecture research comprises:

1. Diagrammatic model of architectural decisions, called Maps of Architectural Decisions (MAD) [170], [167], which has been validated in an industrial setting (section 2.2.4);
2. The concept of the System Organisation Pattern [171], [169], comprising a set of architectural decisions defining the overall design of a large-scale distributed system (section 2.2.6);
3. The taxonomies of architecture evaluation paradigms (section 3.1) and architecture evaluation methods (section 3.3) based on the scope of the method's applications;
4. Early Architecture Evaluation Method (section 3.9.2) [169] – an original architecture evaluation method aimed at identifying major risks at the earliest stages of the development of large-scale distributed systems.

2. SOFTWARE ARCHITECTURE MODELLING

The purpose of this chapter is to summarise the achievements in architectural modelling to date, in order to provide a basis for the presentation and discussion of architecture evaluation methods. Therefore, its purpose is not to deliver a full, detailed presentation on architecture models, but rather to provide insights into their internal design and discuss their features.

The modelling of software architecture has so far been developed in three main directions: modelling software structures, documenting architectural decisions and organising architecture descriptions.

Research on modelling software structures (section 2.1) has focused on developing Architecture Description Languages (ADLs), which, as with other modelling languages (e.g. structured notations or UML), define modelling constructs and syntax rules that constrain the scope of uses and compositions of these constructs, as well as the semantics of the models built out of the modelling constructs.

The general advantage offered by Architecture Description Languages is that they explicitly represent software structures by exposing its building components and the relationships between them. However, they contain no information on what has motivated a given design (rationale), or how the architects arrived at it. This design intent and decision-making process, unless properly captured, evaporates as soon as software is implemented or the architects are gone. This deficiency is addressed by the concept of architectural decisions (section 2.2). The underlying concept is that software architecture can be modelled as a set of architectural decisions, whose superposition defines a given software architecture.

In addition to these two directions, models of organising architectural documentation have also been developed (section 2.3). The research challenge addressed here was to develop means of organising sets of architecture models, of whatever kind, so as to be able to extract the information needed by various stakeholders. The core achievements have been summarised by the recent standard ISO/IEC/IEEE 42010:2011, which promotes the organisation of architecture description around stakeholders, their concerns and viewpoints that capture these concerns. This idea has also influenced the newest ADLs, namely ArchiMate, which is compliant with the standard mentioned above.

2.1. MODELS OF SOFTWARE STRUCTURE

Models of software structure capture the following aspects of software architecture:

- Components – units of a system's organisation, which, for example, can be the unit of computation (procedures, classes, objects, applications or even a system consisting of many applications), data store, hardware and execution environment units, etc.;
- Interactions – represent the exchange of data between the components, e.g. queuing buffers, common data structures, sub-procedure invocations, communication protocols, message passing, etc. Interactions are usually represented by connectors;
- Compositions – show how higher level components are built out of lower level ones;
- Context – represent the business or technical environment that the modelled system interacts with. It can include: the components of a technical infrastructure (sensors, input devices), users, business processes and the organisation's structures.

Models of software structure have been developed since the advent of software engineering (comp. e.g. [114], [141]). They were later named Architecture Description Languages (ADL) [104] when the wave of research on software architecture intensified. The evolution of ADLs has gone towards bringing architecture modelling at a higher level of abstraction (compared to traditional object-oriented and structured notations), enriching the models of system context (business or technical) and including components of technical infrastructure on which the software runs.

State-of-the-art architectural modelling encompasses structured models, box models, UML and SysML and ArchiMate notation (sections 2.1.1–2.1.4). Structured models represent software architecture at a low level of abstraction and provide limited information on a system's context. The block diagrams are very flexible and can be used at every level of abstraction, from a top level coarse-grained view of domains and systems to the detailed software components [152]. At the same time they are intuitive, hence easy to learn and comprehend even by an untrained person. Therefore, block diagrams remain a popular means of representing software architecture, despite the lack of a formal syntax and the semantics and ambiguities that arise as a result.

This limitation has been addressed by semiformal models (UML, SysML) combining an increased level of modelling formalism with the flexibility of block diagrams. However, all the models mentioned so far abstract from the business context of a software architecture, which makes it difficult to organise a set of such models along with the stakeholders' concerns and viewpoints (compare section 2.3). The ArchiMate notation is a response to this challenge. It combines models of a system structure with the extensive models of the organisation being served by the modelled system.

2.1.1. Structured Models

The category of structured models includes two basic models that represent software at an architectural level, namely data flow diagrams (DFD) and structure charts that have been included in various modelling notation delivered by the structured methodologies, compare e.g. [162], [68], [113], 161]. Structured models capture the four basic entities that designate being an ADL:

1. Components – software components are represented both on data flow diagrams on a coarse-grained level (processes, transformations) and on structure chart on a fine-grained level (functions, procedures);
2. Interactions – are represented as data flows between software entities on data flow diagrams, and on structure charts as arguments passed between procedures;
3. Compositions – the processes (transformations) on data flow diagrams can be decomposed hierarchically, which captures the process's composition of "smaller" sub-processes; structure charts depict how higher level functionality is implemented by a hierarchy of more detailed ones;
4. Context – is represented as terminators on data flow diagrams. Terminators are external sources or sinks of data. Special context diagram is included in major structured methodologies such as Yourdon's or Ward-Mellor's method.

Structure charts are now obsolete, while data flow diagrams are still used to represent data flows between the processing units (components). Models of business processes, such as BPMN [111], are in fact modern data flow diagrams, and are widely used in business modelling.

2.1.2. Block Diagrams

Block diagrams are similar to data-flow diagrams, UML component diagrams or SysML internal block diagrams. As with fully-blown ADLs, they enable the modelling of:

1. Components – system components (domain, system, subsystem, application, application's components) are represented as boxes;
2. Interactions – lines or arrows between the boxes denote interactions, though its characteristics are implicit;
3. Compositions – boxes placed inside a larger one represent a composition out of a number of subcomponents;
4. Context – elements of system context can be represented using any symbols (e.g. boxes, circles).

However, the notation is informal and by no means uniform, which means that different architects usually draw block diagrams differently. An example of a box model is set out in figure 1.

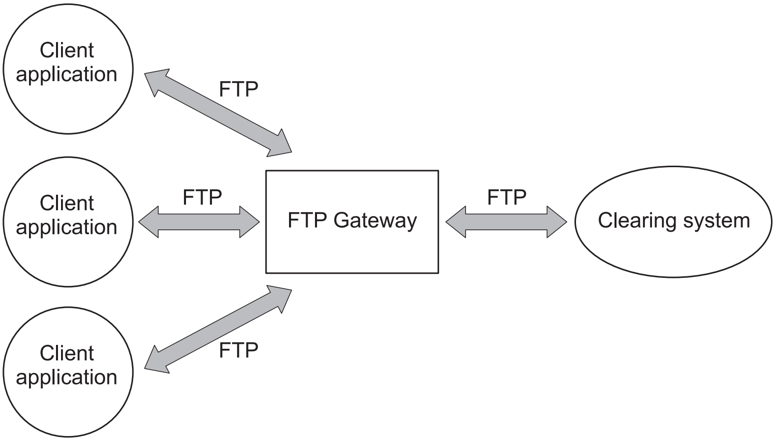

Figure 1. Example of a block diagram – interbank clearing system

2.1.3. UML and SysML

The Unified Modelling Language was originally developed in order to unify the modelling languages accompanying early object-oriented software development methodologies, such as methods by Rumbaugh, Booch and Jacobson. UML has grown into an industrial standard for software modelling. Its current version, No 2.4.1, developed by Object Modelling Group (OMG), has also become an international standard ISO/IEC 19505:2012 standard [72].

Many of the UML models can be used to capture the basic architectural entities:

1. Components – coarse-grained components (systems, applications, application components, services) can be presented in the component diagram; fine-

grained components (e.g. class member functions) can be presented in a composite structure diagram. Units of organisation of software systems are also captured by the package diagrams, which group software entities of any kind (classes, use cases etc.) according to the needs of the analyst.

2. Interactions – are captured in component diagrams as interfaces and connectors, and in composite structure diagrams as ports and connectors. Sequence and communication diagrams represent interactions between software entities (objects, components).
3. Composition – composite structure and component diagrams are a general means of modelling the composition of software system's entities – they define the internal structure of a given component (including class) and the services it delivers outside. The composition of software entities can also be captured by package diagrams, e.g. they can be used to define layers of software architecture.
4. Context – elements of a system context are captured as actors, allowing system context to be captured in use-case diagrams and models, or in activity or sequence diagrams.

Using UML for architecture modelling is generally challenging. UML has been specially developed as a model of object-oriented software. Therefore, it provides modelling concepts capturing various facets and peculiarities of object-oriented software, which *en masse* are not necessarily best suited to architectural modelling. This limitation is a result of inherent properties of UMLs. It is rather ineffective in capturing hierarchical structures, abstracts from business context, and is best suited for modelling design details.

The choice of UML models suitable for architecture modelling is, therefore, problematic. Legacy considerations in this respect can be found in [105]. More contemporary and practical guidelines can be found in [38], which provides rather limited guidelines on how to use UML for architecture modelling; for a summary see table 1, which assigns models to the types of architectural information (views).

Table 1. Summary of guidelines on how to use UML for documenting software architecture (based on [38])

Component of architectural description	Models (diagrams) used to represent a certain element of architectural description
Composition of software entities (modules, components, etc.)	Package, class, composite structure diagrams.
Interactions between components, interfaces	Component, object, composite structure diagrams.
Allocation of software entities to the execution environment	Deployment, package diagrams.
Behaviour models	Behaviour diagrams: activity, sequence, communication, timing, interaction overview, state machine, use-case.

Systems Modelling Language (SysML) [112] was developed by OMG as a UML profile. It has been crafted to be abstract from many details typical for the modelling of object-oriented software, while providing modelling constructs that are well-suited for the representation of a system's structure:

1. Components – are called blocks in SysML terminology. Blocks represent the components of a system's architecture (domains, systems, subsystems, applications, etc.) and are the main entities of SysML, which are included in the following models: block definition and internal block diagrams (see below);
2. Interactions – are modelled in internal block diagrams corresponding to the UML component diagrams and representing components and flows between them (flows of data, materials, electric current). Additionally, models of behaviour, namely activity, sequence, state machine and use case diagrams, can be used for modelling interaction;
3. Compositions – are modelled in block definition diagrams, using the same notation as UML class diagrams (conceptual). Component compositions are modelled as relationships between the blocks, similar to the relationships between classes in an UML class diagram, namely associations, generalisations, compositions, and aggregations. It is also possible to define multiplicities on the ends of an association – one component may contain a number of other components.
4. Context – internal block diagrams and use case diagrams can be used for modelling a system's context. The entities external to the modelled systems are represented as actors or blocks.

The most noticeable difference to UML is the lack of fully-blown data modelling capability: class and object diagrams are absent in SysML. Compared to UML, it also offers rather limited support for behaviour modelling: only activity, sequence, state machine and use case diagrams are available in SysML.

Package diagrams define packages of modelling artefacts and the relationships between them. Packages may contain: other packages, viewpoints, views, blocks and requirements. They may also be connected to the requirements and other models or model components, such as blocks and requirements. Package diagrams enable documentation to be organised in a way that is compliant with ISO/IEC 42010 (compare section 2.3.1). Package diagrams may be perceived as a kind of index of architectural information that enables navigation through it.

SysML provides two additional diagrams:

1. The requirements diagram captures:
   - the hierarchy of requirements relevant to a given system's design (general requirements are translated onto more detailed ones, and so on recursively);
   - the process of deriving requirements, i.e. how requirements are conceptually related to each other and how the analyst arrived at certain architecture.

The requirements diagram captures the context of a given requirement, i.e. related problems, the means of verifying whether a requirement has been fulfilled, its rationale, and refinement relationship.

2. The parametric diagram is supposed to integrate models of a system's behaviour and structure with engineering analysis models (e.g. in the case of control systems, models of a system's dynamics and optimal control algorithms).

2.1.4. ArchiMate

ArchiMate 2.0 [150] is an emerging (version 1.0 appeared in 2009) diagrammatic modelling language developed and promoted by The Open Group, which is becoming popular mainly among the enterprise architecture community. It attempts to include the concepts of service-oriented modelling into an ADL. Therefore, the concept of services is included in the notation. ArchiMate has been crafted to address the following main issues:

- Compliance with the ISO 42010:2011 standard, which was achieved by organising architectural models in eighteen predefined viewpoints capturing various kinds of architectural information (e.g. an organisation's structure, business processes, application usage, application deployment, service implementation) at various levels of detail (e.g. overview-level, technical details). Content patterns and examples of models are provided for all the viewpoints, along with modelling hints;
- Enabling modelling at a higher level of abstraction than competing notations (UML, SysML), which was achieved by including the concepts of service-oriented modelling [5] into the modelling notation, remaining detached from the details of the system's dynamics and by focusing on software architecture modelling at the application and component levels;
- Modelling organisational structures and processes, software systems and underlying technology as an integrated unity. This was achieved mainly by broadening the scope of architectural modelling by including the rich models of technical and organisational context into the scope of architectural modelling.

Architecture documentation in ArchiMate comprises three layers (table 2):

- Business (customers' perspective, business processes),
- Application (applications used in order to offer products and define business processes), and
- Technology (the computing, storage and network infrastructure needed for a system to run).

For each of these three layers, three aspects can be captured in ArchiMate models: structural (static structures), behavioural (a system's dynamics and interaction between the structural elements) and informational (link other concepts, especially behavioural, to the goals of an organisation) elements.

Table 2. Layers and components of architectural models of ArchiMate

Layers	Structural elements	Behavioural elements	Informational elements
Business	Business actor, business role, business collaboration, business interface, location, business object	Business process, business function, business interaction, business event, business service	Representation, meaning, value, product, contract
Application	Application component, application collaboration, application interface, data object	Application function, application interaction, application service.	–
Technology	Node, device, system software, infrastructure interface, network, communication path	Infrastructure function, infrastructure service	Artefact

The four basic components captured by the ADLs are represented in ArchiMate as follows:

- Components – component-connector modelling is included in the application structure viewpoint and both connectors (interfaces) and various types of components are included in the notation, compare table 2.
- Interactions – ArchiMate offers rather limited support for modelling behaviour details, especially when compared to UML or SysML. It focuses on capturing the collaborations between the modelling entities in abstraction from details of its dynamics. Interactions between software components are captured in the application co-operation and application behaviour viewpoints;
- Compositions – ArchiMate enables block diagram style composition modelling and also contains composition and aggregation relationships, which can be applied to any modelling object (e.g. application component, business process). Composition is captured in the application structure viewpoint;
- Context – ArchiMate includes models both of the business environment and of the technical infrastructure, which is a major advantage over the models described in sections 2.1.1–2.1.3. The participation of an application in a business process is captured in the application usage viewpoint; the relationships between software and infrastructure are modelled in the implementation and deployment viewpoint and in the infrastructure usage viewpoint.

The graphical symbols of the ArchiMate notations recall traditional UML, structured or box models. They are supposed to be intuitively understandable. Therefore, no formal semantics are defined, and there are no correspondence rules. This makes ArchiMate a lightweight architecture model, even when compared with the semi-formal UML or SysML. However, this fact can, paradoxically, foster its industrial adoption.

2.2. ARCHITECTURAL DECISIONS AND ARCHITECTURAL KNOWLEDGE

Architectural decisions [152], [78], [91] provide an approach to the modelling of software architecture alternative to the models of software structure (section 2.1). This alternative is founded on the following assumptions:

1. Architecture is a result of a decision-making process; therefore, software architecture can be represented as a set of architectural decisions made in order to develop a given architecture;
2. Architectural decisions resolve certain architectural issues by choosing the most suitable design.
3. These choices are made rationally, which implies that every architectural decision has its rationale, i.e. the motivation of the choices made.

The rationale is a component missing from the models of software structure and is usually considered tacit architectural knowledge that tends to vaporise with the passage of time or as the developers leave. However, not knowing the rationale hinders the transfer, the sharing and the reuse of knowledge on software architecture.

Let us also note that the idea of documenting design decisions and the rationale behind them can be traced back to efforts made in the 1960s and early 70s, aimed at an idealistic goal of defining and formalising the design process, for example compare [3], Issue-Based Information Systems (IBIS) [96] as well as models of design rationale discussed in [79].

Section 2.2.1 presents and discusses the concept and models of architectural decisions, section 2.2.2 discusses the use of architectural decisions as a carrier of architectural knowledge, section 2.2.3 presents an advanced model [172] for capturing architectural decisions, which is accompanied by the techniques of decision-making support. Section 2.2.4 presents the Maps of Architectural Decisions model [167], which is suitable for documenting the evolution of rapidly evolving systems. Architectural patterns and tactics, which are concepts closely related to architectural decisions, are discussed in section 2.2.5. System Organisation Pattern combining the concept of architectural patterns and architectural decisions into an effective means of documenting top-level architecture of large-scale systems is presented in section 2.2.6. Finally, the limitations of the existing approaches to architecture decision-making and capturing are discussed in section 2.2.7.

2.2.1. Concept of Architectural Decisions

Although the concept of architectural decision is fairly mature, it is not easy to find a precise definition in the existing literature. The seminal paper by Jansen and Bosch [78] defines architectural decisions as, "*a description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture*." This "compositional" definition reflects the expectation that software architecture could be synthesised as a sum (or rather a superposition) of a number of architectural decisions, which could be expressed by an equation: *software architecture* = $= ad_1 + ad_2 + \ldots + ad_n$.

In order to achieve this goal, a model linking architectural decisions with the fragments of component-connector models (Design Fragments) has been defined. The entire architecture was supposed to be composed out of those *Design Fragments*. However, this goal has so far turned out to be infeasible, namely neither the decisions nor the "+" operator have been formally defined. The unresolvable difficulty turned out to be the multifaceted nature of software architecture and the cross-cutting nature of architectural decisions. It makes it difficult to combine both of them in a strict and formalised manner. Although, this research direction has been abandoned, the components of architecture decision model by Jansen and Bosch do influence contemporary research, e.g. [167], [133].

Currently, an architectural decision is understood as a choice between a number of architectural (design) alternatives, aimed at resolving a certain architectural issue (problem), for example, "the selection of a communication mechanism between given services" (synchronous/asynchronous), "the selection of a persistence solution for a web application" (Hibernate/QLOR/Spring/OJB), and "the selection of an application server and its associated components".

The contents of architectural decisions have been featured in a number of papers [152], [78], [92], [170], [167], [154], [134]. Although, they use different wording and differ in some minor details, a generalised model of the content of architectural decisions can easily be derived from them. It includes the following attributes:

- Problem (issue) – problem that the architectural decision is expected to address;
- Options (positions, variants) – architectural solutions that are considered as possible solutions to a certain problem;
- Solution – the chosen option, i.e. the architectural solution;
- Rationale – explanation and reasoning underlying the choice made, i.e. it can explain identified trade-offs, indicate pros and cons (compare [78], [167]) or chosen properties of all the considered options [170];
- Decision context – it relates a given architectural decision to the information, which was taken into account, while making that decision, namely other ar-

chitectural decisions, artefacts, design constraints, design rules, requirements, artefacts and assumptions, compare [152], [78], [143];

- Implications (consequences) – an architectural decision can introduce new design constraints, introduce or modify existing requirements, require a review of certain other decisions, etc.;
- Technical fields – contain data necessary to manage decision-making, and organise sets of architectural decisions. This category includes such fields as status, author, time stamp, history and categories.

Further insight into the nature of architectural decisions provides classifications of architectural decisions. The most influential one of these has been introduced by Kruchten in [93]. It defines four classes of architectural decisions:

- Existence decisions – define a system's structure (systems, subsystems, applications, etc.) – compare [103] and behaviour;
- Bans or non-existence decisions – constrain a system's design by indicating that some structural elements or behaviour will not be included. For example, "the system is supposed to be a no-SQL one", "the system will not use asynchronous communication";
- Property decisions – define properties that architecture should account for, for example, "the system should cope with a varying load", or "the system should operate on poor-quality communication links". Hence, this category can be understood as encompassing decisions that introduce design rules, guidelines or constraints;
- Executive decisions – they are made by the management stakeholders, and hence they are usually not addressing directly any software components or their properties. To this category belong: process decisions (e.g. the choice of a development process, the choice of a change management scheme), technology and tool decisions (e.g. the choice of the implementation technology, the choice of the implementation environment, or the choice of the testing tools).

Another important taxonomy was introduced in [172]. It divides architectural decisions between the four levels, namely:

- Executive level – this is the same category as in Kruchten's taxonomy, e.g. preferred implementation or execution platform;
- Conceptual level – these are decisions that define architectural solutions included in a given architecture in abstract from the implementation details, e.g. using a message broker to gather and distribute messages between a number of systems;
- Technology level – this category captures the implementation choices, e.g. communication protocols, choice of interface technology (e.g. web service);
- Vendor asset level – this group includes decisions about the choice of vendors of the products that implement the technologies chosen at the technology-level, or about the configuration of these products, e.g. the choice of component's

vendor can serve as a general example, a more specific one can be the choice of an enterprise service bus vendor.

The most important weakness of Kruchten's taxonomy is its vagueness and ambiguity. In practice, property decisions may overlap with non-functional requirements; it may be difficult to distinguish between certain executive decisions and existence decisions (e.g. a decision that "the system will use Oracle® database" may be attributed to both the existence and executive class of architectural decisions). The classification by Zimmerman et al. is much more precise, as it is supposed to follow the top-down logic of the architectural decision-making process, nevertheless, the category of executive decisions still overlaps with other categories.

2.2.2. Capturing Architectural Knowledge with Architectural Decisions

Architectural knowledge comprises explicit and tacit knowledge. The former is documented as artefacts, defined by a development process (e.g. waterfall, Rational Unified Process) and created during the software development. Tacit architectural knowledge, in turn, is the reasoning and motivation underlying a given design. It tends to vaporise with the passage of time or as the staff leaves, if it is not captured as it is conceived.

By documenting architectural decisions, one captures both explicit and tacit architectural knowledge, namely architectural design and its rationale as well as the decision-making process that leads to a given architectural design. Architectural decisions integrate various kinds of architectural knowledge into a single entity. This observation triggered a wave of research on architectural knowledge management [6], in which architectural decisions play a pivotal role.

There are two general research challenges in the area of architectural knowledge management:

1. The representation of architectural decisions including their content and form of presentation (graphical, textual);
2. The organisation of large sets of architectural decisions in order to enable the storage, navigation, manipulation (adding, modifying, deleting), search and retrieving of the relevant architectural knowledge. Let us note that in order to document software architecture, a lot of architectural decisions usually have to be captured, e.g. in [173] almost 400 architectural issues, and hence, architectural decision kinds, have been identified.

The first of the above issues has been addressed by the introduction of various ways of representing architectural decisions:

- as pure text records, whose content complies with that presented in section 2.2.1,
- visualised in the form of diagrams, which should facilitate comprehension, compare e.g. [170], [167], [133], [45];
- linked to the relevant development artefacts, compare e.g. [35], [167].

The second of these two research challenges has been addressed by documenting the various relations between the architectural decisions or their components (options and chosen solutions) and by classifying architectural decisions according to a predefined scheme (compare section 2.2.1).

A reference point for classifications of relations that might exist between architectural decisions constitutes an influential classification by P. Kruchten presented in [93], [92]. It defines the following categories of relations:

1. Constrains – one decision constrains the other, e.g. the decision that "System A will use JBoss application server" constrains the decision about "the choice of an external, grid cache for system A" (the choice of JBoss application server implies that external, grid cache will have to belong to the JBoss family).
2. Forbids – one decision eliminates the other, e.g. "the system will use a synchronous communication" forbids "the services will use asynchronous communication";
3. Enables – one decision makes the other possible, e.g. "System A will communicate with B over a queuing system" enables "System A will send data to system B asynchronously";
4. Subsumes – one decision includes other decisions, e.g. "System A will be implemented in Java" subsumes "Component X, Y, Z of system A will be implemented in Java";
5. Conflicts with – two decisions are mutually exclusive, e.g. "module X will be implemented in Ada" conflicts with "module X will be implemented in Java".
6. Overrides – one decision cancels another, e.g. "subsystem A and B will share a relational database" overrides "subsystem A will store data in XML files only".
7. Comprises – means that a certain decision is actually composed of a number of sub-decisions, e.g. "configuration of communication between subsystems" decision can comprise "synchronous vs. asynchronous communication", "choice of communication protocol", "choice of communication library/api" decisions.
8. Is an Alternative to – two decisions address the same problem, but denote a different solutions, e.g. "Subsystems A and B will communicate asynchronously" and "Subsystems A and B will communicate synchronously" address the same issue but propose different solutions. Let us note that this is, in fact, rather a relation between considered options (potential solutions) not between architectural decisions alone.
9. Is Bound to – two decisions constraint each other, e.g., "component A will run on an application server" and "component A will be implemented in J2EE".
10. Is Related to – decisions are related to each other in some way other than those listed above.

The general problem is that the above relations are ambiguous and vague. It is often difficult to identify them in practice. A separate issue is that some of these relations are between the components of architectural decisions rather than between the decisions themselves – generally most of them could be defined as relations between the considered options ("is an alternative to", "conflicts with", "forbids", "enables") or between an option and another decisions (or vice versa) ("overrides", "enables", "forbids"). This is quite an ambiguous situation, which makes this classification more cognitive than practical.

Architectural decisions, their classifications as well as the relations between them, are the basic components of models of architectural knowledge and of the design of architectural knowledge management systems, which enable architectural knowledge to be produced, consumed and managed. Architectural knowledge management models and tools usually define their own classification schemes for architectural decisions and for the kinds of relations between architectural decisions; nevertheless, the above classification still remains a reference point. For a survey of the experimental tools, refer to [144], [6]; none of them has been adopted by the software industry so far.

The flagship model for architectural knowledge management with many additional features has been presented in section 2.2.3.

2.2.3. Extending Models of Architectural Decisions with Decision-Making Support

The ideas presented in section 2.2.2 gave rise to the development of elaborate methodologies for architectural decision-making and capturing architectural knowledge, supplementing the basic components of the models of architectural knowledge, described in sections 2.2.1 and 2.2.2, with the following elements:

- rules of model consistency;
- support for decision-making, possibly including the decision-making techniques [53]; and
- modelling guidelines.

A highly formalised model by Zimmerman et al. [172], being a major achievement in this field, has been presented below.

The model proposed in [172] comprises:

A. The taxonomy of architectural decisions;
B. Architecture decision model, which represents the architectural decisions and the relations between them and their components (detailed treatment is provided beneath);
C. The integrity constraints;
D. The support for architectural decision-making: the outcome instances mechanism and the production rules.

The taxonomy of architectural decisions proposed in [172] classifies architectural decisions against the four predefined refinement layers, which are supposed to reflect a top-down design approach:

- The executive decisions – the same as analogous Kruchten's category;
- The conceptual decisions – a selection of top-level architectural patterns (e.g. three-tier application, service-oriented architecture) and key technologies (e.g. Business Process Management System, Enterprise Service Bus);
- The technology decisions – a selection of detailed-level architectural patterns, design patterns [61] are typical examples;
- The vendor asset decisions – choice of vendors of system and software components (e.g. application server, commercial component libraries, middleware solution).

The model of an architectural decision comprises the following elements, making up the tree structure shown in figure 2:

- *Topic groups*, which represent closely related design concerns. Topic groups should be organised as tree-like top-down hierarchies. Topic groups provide means of organising architectural decisions in a logical and easy to comprehend way, which could be tailored to a certain application.
- *Issues*, which represent architectural (design) problems;
- *Alternatives*, which represent possible solutions to a given *issue*. The given alternative can only be a solution to a single issue.

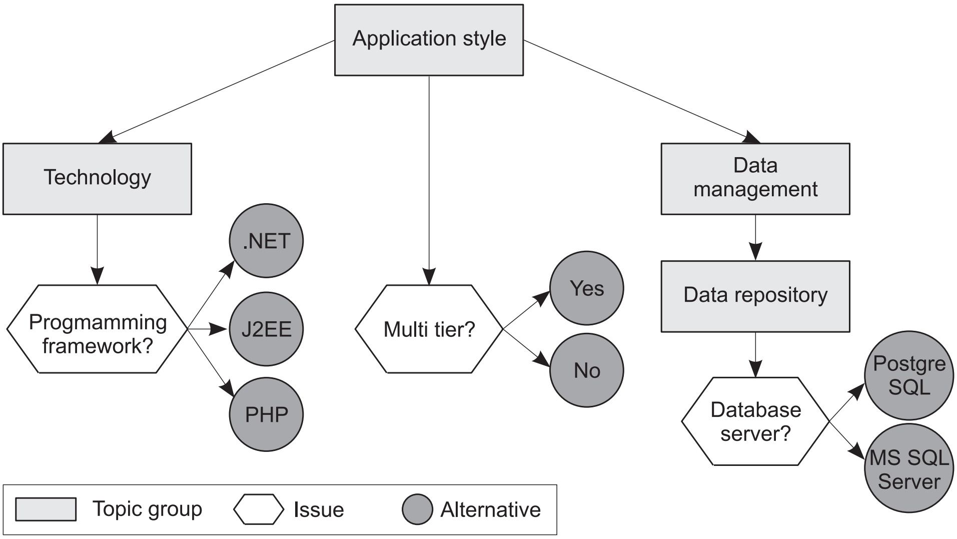

Figure 2. An example of architectural decision model.

The architectural decision model [172] is represented as a set of trees (a forest), representing architectural decisions. Trees (decisions) are additionally grouped according to the levels of abstraction they belong to. In general, an ordered set of such levels should be predefined by the architects. The four refinement levels described above can also be used for that purpose.

It has also been noticed in [172] that relations exist rather between the components of the models of architectural decisions than between decisions themselves. These relations set up links between models of architectural decisions. This has enabled the integrity constraints and the production rules to be defined. The latter support the decision-making process. The following relations have been defined:

- Between issues (problems):
  - the *influences* relation represents a cross-cutting concern between issues;
  - the *refinedBy* relation represents an issue that should be analysed at different levels of detail, i.e. *x refinedBy y* means that the details of issue *x* are represented by issue *y*, which has to belong to a lower, more detailed level of the architectural decision model;
  - the *decomposesInto* relation represents functional aggregation, i.e. the related issues should in fact define a single, decomposable, architectural problem. These issues have to belong to the same refinement level.
- Between alternatives (options)
  - *forces* – alternative *x forces y* means that selection of alternative *x* in one issue, enforces the choice of alternative *y* in another issue.
  - *isIncompatibleWith* – denotes that related alternatives exclude each other (are incompatible).
  - *isCompatibleWith* – means that two alternatives work together;
- Between other model entities:
  - the *triggers* relation between: alternative, issue and topic group – it is a causal relation, meaning that it captures a decision-making process, rather than logical dependencies. It means that the selection of a given alternative *a* triggers issue *i* together with the topic group *t*, to which issue *i* should belong. The *triggers* relation defines the decision-making paths that run through a number of decision trees;
  - the *hasOutcome* relation binds an issue and an outcome instance. An outcome instance is an entity representing a subset of alternatives that are considered for a given kind of issue. This enables already analysed issues to be reused, by connecting their alternatives (outcome instance) to other issues.

The model briefly described above has been accompanied by a set of **nine integrity constraints** that should preclude flawed structures of relations. The integrity rules are:

1. The relations *refinedBy* and *decomposesInto* exclude each other (the latter concerns alternatives belonging to the same refinement level, while the former concerns different refinement levels).
2. The issues bound by either *refinedBy* or *decomposesInto* relation must not be simultaneously in an *influences* relation, and vice versa. This would introduce a useless redundancy.

3. If an alternative *x forces y*, the other alternatives connected to the same issue as alternative *y* should be incompatible with *y*;
4. The relations defined between the alternatives (*forces*, *isCompatibleWith*, *isIncompatibleWith)* are mutually exclusive; one of them is supposed to exist between every pair of alternatives, *isCompatibleWith* is default one.
5. If issue *i* is *refinedBy* issue *j* or *decomposesInto* issue *j*, then all the alternatives connected with issue *i* trigger issue *j* (the *triggers* relation is induced by the relations *refinedBy* and *decomposesInto* and should be automatically included in the model, which is later needed for a decision making).
6. The *forces* relation between alternatives *x* and *y* means that alternative *x* triggers issue *i*, which contains alternative *x*.
7. Let us consider all the issues that are reachable from alternative *x* (*I*(*x*)) by following the triggers relations in the same way as the breadth-first search graph algorithm (starting from an issue x we arrive by the triggers relation at issue *i*, then starting from each of the alternatives connected with issue *i* we follow trigger relations to other issues, from these issues we repeat the earlier procedure adding issues found "on the way" to the set *I*(*x*)). For every issue *j* reachable from alternative *x*, there must be at least one alternative *a* attached to issue *j*, such that the *a isCompatibleWith i*, or exactly one alternative connected to issue *j* must be in a *forces* relation with alternative *x*.
8. Issue *j* triggered by alternative *x* that belongs to issue *i* must belong to a lower refinement level than *i*, or, if they belong to the same refinement level, issue *j* should be a successor of issue *i* according to the ordering relation (it defines the recommended reading sequence for architectural decisions, which comprise the architectural decision model).
9. If alternative *x* is chosen, only alternatives that are compatible or forced by alternative *x* are allowed to be chosen in other outcome instances of the same type (label). See more on outcome instances below.

Let us note that:

- these rules constrain the syntax not the semantics of the model, as they either preclude or impose a certain correspondence between the relations, but do not deal with the concrete content of issues and alternatives, which provide a carrier for the decisions' semantics;
- these rules can be verified automatically;
- rules Nos 5 and 6 ensure the automatic generation of the *triggers* relation; for more on the role of the triggers relation see below;
- rule No 7 is supposed to ensure the resolvability of the models, i.e. whatever alternative one starts from, one arrives at issues that have at least one alternative that is compatible with the starting issue;
- the model's authors observe that rule No 8 does not hold in many cases, as strictly top-down architecting is not always possible. For example, in many cases vendor-level decisions result straight from the executive decisions;

- there is no proof of the completeness of the above set of rules, i.e. it has not been proved that there cannot exist a faulty model that satisfies the integrity constraints.

In the model described above, **architectural decision-making** is a process of traversing issues following *triggers* relations, and choosing a resolution out of the alternatives resolving a certain issue. The method does not provide any support for making choices, which is left to the architects.

Decision-making starts from issues, called *entry points*, that do not participate directly in any *triggers* relation. Note that some triggers relations can be generated automatically from *refinedBy*, *decomposesInto* and *forces* relations (compare integrity rules Nos 5 and 6). All the entry points should be examined.

The method assumes that some issues may recur in the architecture. Such recurring issues should be addressed consistently, i.e. the sets of alternatives that resolve such issues should recur in all the recurring issues. Such a recurring, reusable set of alternatives is called *outcome instance* and can be bound to issues with a *hasOutcome* relation. Outcome instances that concern the same kind of issues have the same text labels (possibly denoting the kind of issue, e.g. "service interface type"). This way, such recurring issues can be treated coherently. Outcome instances may be in one of three statuses: "open" (assigned initially), "implied" (intermediate status, containing only one alternative, which has not been pruned), and "resolved" (none or one alternative has been chosen).

The decision is made by choosing an alternative according to the architect's knowledge and preferences. It results in pruning all the other alternatives connected to the considered issue. The following production rules provide for the automatic pruning of superfluous alternatives and for updating the statuses of outcome instances:

1. [*Pruning the incompatible alternatives*] Let a_1 and a_2 be alternatives to issue i, which are incompatible. If a_1 is chosen as a solution to an issue i (note: it means that the status of issue i is changed to "resolved"), then a_2 should be removed from all the outcome instances of i. As choices are made, more and more alternatives become incompatible with the alternatives already chosen. This rule expresses that there is no point in analysing alternatives that conflict with a decision already made.
2. [*Outcome propagation*] If alternative x *forces* y and x has been chosen, then alternative y should also become chosen, and other alternatives to y pruned. This choice should be propagated to all the instances of the outcome alternatives that contain alternative y. Note: alternative y may occur in a number of *outcome instances* of the same type. Outcomes affected by this procedure are called "*implied outcomes*", i.e. their status is changed from "open" to "implied".
3. [*Outcome instance status update*] If all the alternatives have been pruned from outcome instance p, or after pruning there remains only a single alternative,

then the status of such an outcome is changed to "implied". Such implied outcome instances should be reviewed by the architect in order to verify their technical soundness. This may result in revoking a previously made decision.

2.2.4. Maps of Architectural Decisions

The model by Zimmermann et al. captures and supports the architecture decision-making process in a highly formalised way, especially when compared to the textual models of architectural decisions [6], [152]. The fundamental precondition for this formalisation is that it is possible to enforce a certain order of architectural decision-making (in the case of [172] – top-down refinement). As observed in [167], such an orderly decision-making process is, in many practical cases, unachievable. Many modern organisations, including telecoms, belong to the class of "emergent organisations" whose systems are "subject to constant urgent change" [22]. Their evolution is random rather than a highly predictable, carefully deliberated process following an earlier established path, as in [64] or [172]. At the same time, architects working for emergent-organisations have a rather limited time for crafting architectural changes, which leaves little time for capturing architectural knowledge. These factors limit the applicability of elaborate models and methodologies, such as that of Zimmerman et al.

Maps of Architectural Decisions 2.0 model (MAD 2.0) was crafted in order to support the capture of architectural decisions as they are made while architecting changes to rapidly evolving systems. Its presentation has been based on paper [167].

MAD 2.0 works similarly to popular mind maps used to graphically present a problem structure. The model consists of two diagrams, Architecture Decision Relationship Diagrams (ADRD), and Architecture Decision Problem Maps (ADPM). ADRD represents the logic of the decision-making process – the diagram can be developed gradually, while ADPM models the internal structure of a single decision problem (issue). The notation's syntax and validity rules are presented below.

The **Architecture Decisions Relationship Diagram** (ADRD) is built out of just two basic elements (figure 3):

- **Decision problem** – represents the architectural issue being considered;

Attributes: problem name, problem description, status, creation date, resolution date, extended solution rationale.

States: *defined* – indicates a newly defined project, *being solved* – an ADPM for the problem has been created, but the problem has not yet been resolved, *solved*, *requires reassessment* – indicates that solution, or the occurrence of other problems, requires an already resolved problem to be reconsidered.

- **Connector** – in its basic form shows just that one problem led the architect to the one indicated by an arrow, in the form of a hexagon it indicates that the solution of a given problem constrains the possible solutions of the indicated problem ("constrains relation"). Two Decision problems can be connected only once.

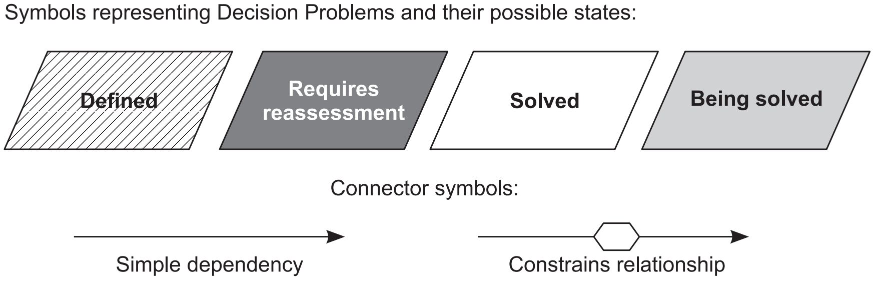

Figure 3. The Elements of an Architectural Decisions Relationship Diagram

Problems can group on the ADRD by surrounding some of them with a solid line – compare figure 5. Such a group is treated as a single problem. In this way, architects can indicate that problems concern a closely-related issue (e.g. define a domain solution).

The **Architecture Decision Problem Map** (ADPM) is a model, in which architectural decisions are actually captured. The elements of the ADPM diagram have been summarised in figure 4. The central element of a diagram is a single decision problem symbol (as in figure 3) representing the architectural issue (problem) being analysed. The other symbols are:

- **Solution – represents a single solution to the architectural problem being considered**
  Attributes: name, state, description, generated problems.
  States: *defined* – assigned immediately after creating an element; *feasible* – indicates a solution meeting all the requirements, *infeasible* – indicates a solution that does not meet some of the requirements (the two former states are assigned automatically), *chosen* – indicates the finally selected solution (assigned by the decision-maker).
- **Requirement – represents a requirement relevant to a given architectural** problem
  Attributes: *name*, *description*
- **Decision-maker** – represents a person or a group of people responsible for the resolution of a related architectural problem.
  Attributes: name, remarks
- **Pro or Con** – represents a single advantage or disadvantage of a given solution
  Attributes: name, description, state, related requirement (met or unmet requirement if the given element represents such a case)

State: *defined* – assigned immediately after creating a given Pro or Con element; *minor*, *medium*, *major* – declares the importance of a given advantage or disadvantage of a given solution.

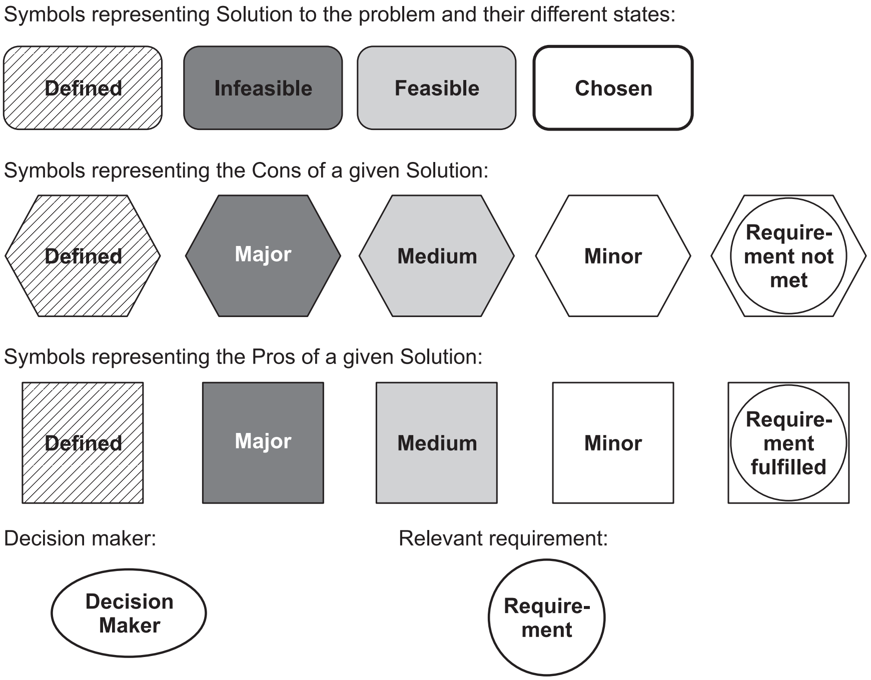

Figure 4. The Elements of the ADPM Diagram

MAD 2.0 models have been crafted to assist system architects in the same way as creating mind maps. This helps to capture architectural knowledge as it gradually comes to light while elaborating the architecture (i.e. decision-making). Examples of ADPM and ADRD models are presented in figure 5 and 6. Note that a tool supporting the capture of architectural decisions and binding them to appropriated parts of specifications of changes has also been developed and described in [167].

The syntax of the MAD 2.0 model is intuitive. The following syntax rules have been defined:

- Rule 1. Two decisions represented on ADRD may be unconnected or connected with just a single connector.
- Rule 2. Decision-maker, Requirement and Solution symbol on ADPM can only be attached to a Decision problem symbol.
- Rule 3. Pros and Cons symbols on ADPM can only be attached to a solution symbol.

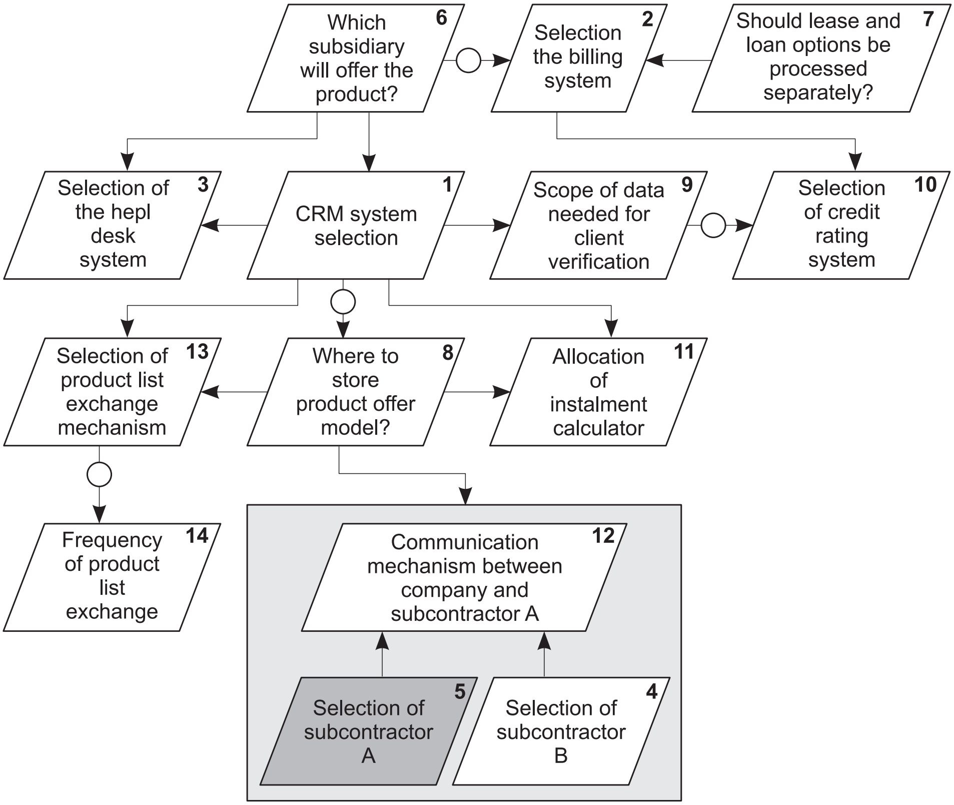
Which subsidiary will offer the product? 6
Selection the billing system 2
Should lease and loan options be processed separately? 7
Selection of the hepl desk system 3
CRM system selection 1
Scope of data needed for client verification 9
Selection of credit rating system 10
Selection of product list exchange mechanism 13
Where to store product offer model? 8
Allocation of instalment calculator 11
Frequency of product list exchange 14
Communication mechanism between company and subcontractor A 12
Selection of subcontractor A 5
Selection of subcontractor B 4

Figure 5. An example of an ADPM model

Although, MAD 2.0 models are semiformal, the syntax imposes a certain structure of the information representing architectural decisions that enables model validation:

- Rule 1. Every decision problem has to be resolved, i.e. one of the solutions chosen.
- Rule 2. Every solution has to be assessed in the context of every requirement relevant to the given problem. All these requirements have to be finally classified as either Pros (requirement met) or Cons (requirement not met) of a given solution.
- Rule 3. Every Pro (Advantage) or Con (Disadvantage) may be attached to a given solution only once.
- Rule 4. Only a single solution to a given problem can be in the "Chosen" state.
- Rule 5. Pros and Cons connected with a given solution cannot be mutually contradictory.
- Rule 6. Two solutions to a problem cannot designate virtually the same resolution to the same architectural problem.

Rules 1–4 can be verified automatically, while rules 5–6 can be verified with a model walkthrough.

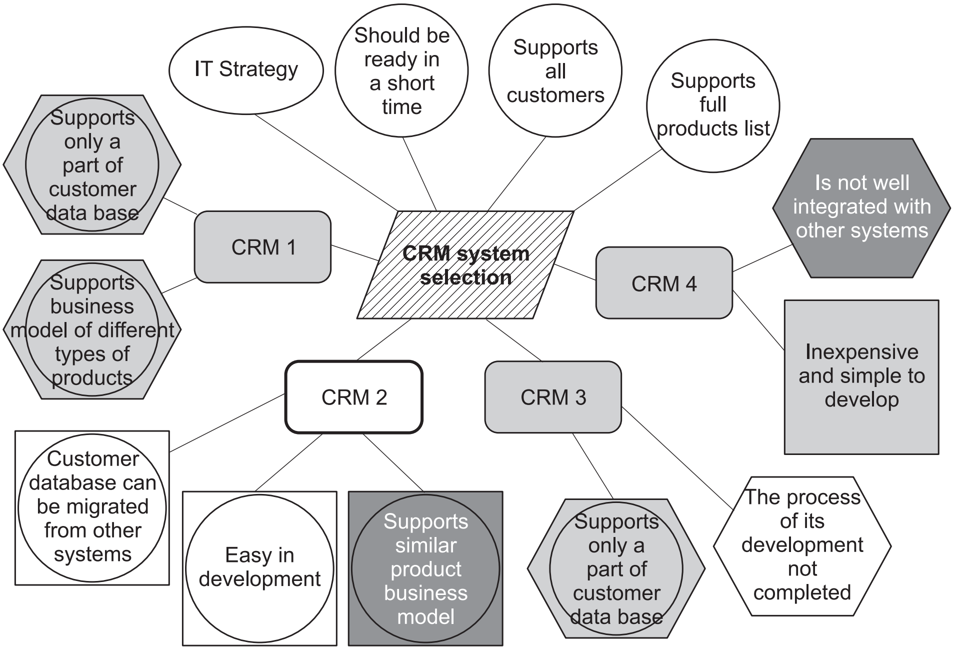
IT Strategy
Should be ready in a short time
Supports all customers
Supports full products list
Supports only a part of customer data base
CRM 1
CRM system selection
CRM 4
Is not well integrated with other systems
Inexpensive and simple to develop
Supports business model of different types of products
CRM 2
CRM 3
Customer database can be migrated from other systems
Easy in development
Supports similar product business model
Supports only a part of customer data base
The process of its development not completed

Figure 6. ADPM for the problem of CRM selection

When developing models supporting architectural decision-making, we face the classic dilemma: the more complicated the model, the less usable it is, and vice versa: we can increase usability by decreasing the complexity of the model, and at the same time decrease its expressiveness, namely the range of analyses it enables. Over-complicated architecture decision-making models create a complexity of their own, adding to the complexity of the overall systems construction, instead of supporting complexity control [168].

Table 3 contains a comparison of the Maps of Architectural Decisions model with textual representations of architectural decisions and model by Zimmerman et al. (section 2.2.3).

Semiformal diagrammatic modelling has turned out to provide the right balance between model usability and complexity. MAD 2.0 is certainly not overloaded with information, though it is still possible to verify some consistency/completeness rules. In terms of the level of formalism, MAD 2.0 stands between informal text models (e.g. [152], [65]), and the formalised model by Zimmerman et al. Although the range of information concerning architectural decisions and the decision-making process has been limited, the most important components of architectural knowledge are still preserved, i.e. the rationale of the decision, the considered solutions and their pros and cons.

Table 3. Models supporting architectural decision making – a comparison

	Textual models	MAD 2.0	Model by Zimmerman et al.
Model form	Text records	Diagrams, additional information stored in attributes of diagram elements	Graphs, certain elements accompanied with text attributes.
Level of formalism	Basic information structuring (fields of text records).	Syntax defined, simple consistency/completeness check.	Syntax defined, extended completeness/consistency check, decision-making consistency based on relations between ADs.
Information content	Issue, Decision, Status, Group, Assumptions, Constraints, Positions, Argument, Implications, Related decisions, Related requirements, Notes [3]	Decisions, two kinds of relations between ADs, problems (issues), possible solutions, pros and cons of every solution, chosen solution indicated, rationale.	Classification of architectural problems, possible solutions, pros and cons of every solution, chosen solution indicated, rationale.
Classifi-cation of ADs	No classification assumed, decisions can be grouped according to the architects' needs.	No classification assumed; decisions can be grouped according to the architects' needs.	Problems assigned to one of the following levels: Executive, Conceptual, Technology, Vendor Asset. Topic groups provide for an additional classification.
Relations between ADs	Does not assume any particular types of relations.	Only "leads to" and "constrains" relations.	Influences, refined by, decomposes into, forces, is incompatible with, is compatible with, triggers, has outcome.
Rationale modelling	Textual.	Diagrammatic, when necessary supported by additional textual explanations.	Textual.
Model analysis and verification	Manual walkthroughs only.	Limited to syntax enforcement, consistency/completeness check, with automated or manual walkthroughs.	Automatic verification of decision-making consistency, completeness and consistency check.

ADPM is similar to the rationale model of [133]. An assessment of candidate problem solutions by indicating their pros and cons seems intuitive and recalls the model proposed in [78].

MAD 2.0 does not assume any predefined classifications of architectural decisions, which is a reasonable decision if one considers the drawbacks of existing classifications, described in [168] and section 2.2.1 and 2.2.2.

Only two kinds of relations between ADs are available in MAD 2.0. This provides for a smooth, uninterrupted flow of the architecting process, as architects do not need to worry about which of the several kinds of relations to choose from, which often becomes a separate challenge in itself. The MAD 2.0 was accompanied by a tool similar to Knowledge Architect [6] – architectural decision models are linked to appropriate parts of requirements specifications.

Although MAD 2.0 has been motivated by the rapid, random changes typical for the evolution of systems supporting emergent organisations, it can also be used as a kind of light-weight architecture decision-making model for the initial architecture development.

2.2.5. Architectural Patterns

Architectural patterns are an idea complementary to architectural decisions. They are, in fact, generic architectural solutions, which resolve certain type of architectural problems [65]. Hence, architectural patterns define reusable solutions for recurring architectural issues.

The concept of design pattern has been originally developed in the civil engineering discipline. A comprehensive catalogue of urban and building engineering patterns, found in the famous book by Alexander et. al. [4], which is still cited by all contemporary books and many research papers on patterns in software engineering, introduces a scheme for pattern definition that comprises context, i.e. the conditions in which a problem occurs, the problem and the solution to that problem. These three basic elements can be supplemented with auxiliary information on such issues as the consequence of using a given pattern, possible implementations, and examples of applications.

Let us note that there are three almost synonymous expressions denoting the concept of patterns to be used in software design: design patterns, architectural styles, architectural patterns (compare, e.g. [65]).

Design patterns were introduced in the famous book by Gamma et al. [61] and are supposed to be used in detailed software design. The book comprises a collection and definition of patterns recurring in the design of object-oriented software. It includes the following categories of patterns: creational (abstract factory, builder, factory method, prototype, singleton), structural (adapter, bridge, composite, decorator, facade, flyweight, proxy) and behavioural (chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor).

Architectural patterns and styles are synonymous notions denoting design patterns that can be used at an architectural level. Definitions of architectural patterns include not only the description of a problem's context, the problem statement and solution to that problem, but also the possible influence of that solution on software properties. This enables the properties of software to be inferred from the patterns included in software architecture.

The comprehensive catalogue of architectural patterns can be found in [32], [33], [34], [131], [88]. A survey of this catalogue is beyond the scope of this monograph. Let us observe that the most important architectural patterns are: layers, pipes and filters, shared repository, active repository, blackboard, interceptor, model-view-controller, client-server, publish-subscribe, broker, and message queuing.

Architectural patterns are an effective means of capturing design knowledge, because they define both the structure and logic of a given part of the software architecture. This is a major advantage over the models of software structure (section 2.1), which usually requires a number of diagrams to capture both structure and processing logic. Architectural patterns increase the level of abstraction of architectural modelling by providing higher-level entities that enable to define, comprehend and analyse software architecture. This leads to the conclusion that both patterns and architectural decisions can effectively represent complex entities making up software architecture. However, the users of such a pattern-based architecture description have to possess proficient knowledge on patterns.

Architectural tactics (compare, for example, [16], [142]) are a concept similar to patterns. They denote a proven solution to a recurring issue connected with a given quality attribute, though they focus on defining mechanisms or guiding the design activities rather than indicating certain structures. Architectural tactics for modifiability (e.g. increase cohesion, reduce coupling), performance (e.g. prioritise events, introduce concurrency), security (e.g. detect intrusion, authenticate users), testability (e.g. executable assertions) and usability (e.g. cancel, undo) have been gathered in [16]. Let us note that the sets of architectural patterns and architectural tactics overlap to some extent.

2.2.6. System Organisation Pattern for Large-scale Distributed Systems

As observed in section 2.2.5 architectural decisions and patterns are very effective at capturing complex design entities. This advantage has motivated the development of the System Organisation Pattern for large-scale distributed systems [171], [169], which is presented below.

System Organisation Pattern defines a "back-bone" of a large-scale distributed system, which comprises a set of architectural decisions that define how a system is organised in terms of permanent data storage management, data communication, data input and output, coarse-grained modularisation and allocation within the organisational structure. It provides an efficient way of documenting top-level architectural design for large-scale distributed systems. System Organisation Pattern as every other architectural pattern can be defined in terms of context, problem and solution to that problem.

Context: The class of large-scale software systems can be distinguished by the following technical features:

- Strong geographical distribution – large-scale software is typically built at the commission of national or global public or private organisations.
- Large volumes of data being stored and processed.
- Large numbers of concurrent users being served – the number of users usually reflects the number of an organisation's employees, and typically exceeds 1000 users.
- Distribution of computational and data resources – results from the geographical distribution feature; the main concern here is the allocation of storage and processing to the locations and the organisation's units.
- Managing large distributed data resources – the distribution of data resources (distributed databases, local databases in thick-client architecture) usually require the resolution of issues of data synchronisation, transaction processing and data exchange between system units.
- Long- and short-distance communication solutions ensuring data flow between the system's units – communication mechanisms, protocols, message passing schemes, etc. (compare also [63])

Problem: Large-scale distributed systems consist of entities that recur throughout the system's structure. For example, if it has been decided to have local databases serving local client software for each company location, then this structure will be replicated in all the locations. Therefore, the basis for a detailed design of a large-scale distributed system is the identification of those entities, their allocation within a target organisational structure, ensuring the communication between them, as well as the organisation of permanent data storage to be used by those entities.

Solution: The following architectural decisions should be made in order to define the large-scale system's backbone, upon which a detailed design will be made:

- Decomposition into a set of subsystems/applications – defines a coarse-grained functional decomposition. Subsystems here are meant as single applications, or suites of applications accompanied by a common infrastructure (e.g. databases, application servers), prepared for a single functional domain. Each application or subsystem encompasses a portion of the system's functionality. The allocation of functionality is a crucial aspect of the decomposition onto a set of subsystems/applications.
- Geographical and organisational allocation of subsystems/applications – the structure of the system has to reflect the structure and geographical distribution of a target organisation. The System Organisation Pattern provides information on the organisation's units for which certain applications are provided, as well

as information on the geographical situation of these units and corresponding subsystems/ applications.

- Organisation of data input – defines the design and location of points where external data is entered into the system. This includes system interfaces, the location of remote access points, data entry (typing) points, scanning devices, sensors etc.
- Organisation of data storage and processing:
  - Data storage distribution – defines the distribution of permanent data resources (typically databases),
  - Data processing organisation – is basically about choosing the data processing architecture (client-server, multi-tier, thin/thick client).
  - Distributed data storage management – concerns the synchronisation of distributed databases, transmitting data onto remote systems, uploading data from a local data store to the main data store.
  - Transaction management – regards the selection of solutions ensuring transactional processing. This involves both short and long transactions, but short ACID (Garcia-Molina et al., 2001) transactions are offered by data base engines. However, this mechanism is not sufficient for long transactions requiring specialised advanced solutions, such as transaction monitors.
- Communications framework – defines the communication mechanisms between equivalent system entities (e.g. processes, applications) and communication protocols used to transmit data between system entities.

*

Such a set of decisions may be represented using the templates presented in [152], [65], or as an architectural narrative containing information prescribed by the above definition of the System Organisation Pattern (section "solution"). Let us consider, for example, the top level architectural description of the interbank clearing system [169] shown in figure 1:

This system was designed to deliver wire transfer services between banks, and has been operating in a clearing house company for more than 15 years. It consists of three subsystems (the first two subsystems had mirror copies to increase reliability):

1. Clearing processing system – processing wire transfer data.
2. FTP gateway – receiving packages of wire transfers data sent from client applications and providing access to them for the clearing processing system.
3. The client application – used to verify data before the clearing process, and to send and receive data from the FTP gateway.

The clearing processing system and the FTP gateway are located in the data centre of the clearing house, while client applications are installed in the banks' systems. The client application uploads wire transfer data, packaged in a text file and stamped with a hash, on to the FTP gateway, which in turn verifies the consistency of the received data and, if successful, access to the data is provided for the clearing processing systems. The clearing process works in sessions, therefore, the packages are processed by the clearing processing system at scheduled times. Repackaged wire transfer data, directed to corresponding banks (transfer addressees), is sent back to the FTP gateway. Finally, these packages are downloaded by the client application of the corresponding banks.

This example shows that the System Organisation Pattern efficiently captures top-level designs of large-scale distributed systems. This is achieved by defining a comprehensive set of architectural decisions that define the system's design at a uniform abstraction level. Such a model defines the top-level system entities (subsystems, applications etc.) and their placement in the geographic and organisational structure, the organisation and operation of the permanent data storage, as well as the communication mechanisms used to transfer data between system entities. These basic components of every distributed software system design define a platform upon which more detailed designs will be crafted. A method of evaluating the System Organisation Pattern is presented in detail in section 3.9.2.

2.2.7. Limitations of Architectural Decision-Based Modelling

The limitations of architectural decision-based modelling of software architecture have been examined in depth in [168]. The main observation was that, despite the role of architectural decisions as a carrier of architectural knowledge, architectural decision-making contributes minimally to overcoming a software system's complexity. This results mainly from:

- the limitations of the textual documentation of architectural decisions, i.e. incompleteness, inconsistency and ambiguity, as well as inefficiency in representing and sharing engineering concepts;
- the inherent complexity – architectural decisions may represent internally complex, often cross-cutting concepts; they may concern a single or a number of entities, and they may represent concepts belonging to various abstraction levels.

These properties make it difficult to define an unambiguous classification of architectural decisions and a set of such relations between architectural decisions or their components. This, in turn, hinders the effective organisation of large sets of architectural decisions.

Providing support for architectural decision-making and knowledge capturing also remains a challenge. Despite efforts to formalise the architecture decision-making process, such as [172], architecting remains a creative activity that remains resistant to formalisation efforts, especially in the case of modern highly complex and rapidly evolving software systems.

With the current state of the art, it is rather unimaginable that architecture documentation could consist only of architectural decisions – such a model consisting of hundreds of architectural decisions would collapse under its own weight, becoming useless. No wonder, then, that architectural decisions and architectural knowledge management still remain an area of intensive research that is yet to be implemented in every-day engineering practice.

Architectural patterns, combined with architectural decisions, may help to overcome some of the above limitations. Architectural patterns aggregate component structures and logic of their collaboration. This increases the level of abstraction at which software architecture is represented and comprehended. The concept of the System Organisation Pattern (section 2.2.6) takes advantage of architectural patterns' ability to effectively represent complex design entities. Its efficiency in representing the top-level architecture of large scale systems comes also from predefining categories of architectural decisions to be captured.

2.3. MODELS OF ARCHITECTURAL DESCRIPTIONS

Software architecture, as with any complex entity, can be seen from a number of perspectives. The research on architectural descriptions is concerned with organising architectural information according to the needs of architecture stakeholders. Views and viewpoints are the primary mean of organising architectural information. They are supposed to tailor the scope of architectural information and the way it is presented to the needs of the architecture stakeholders.

Although the software architecture research community attributes the origins of the research on viewpoints and views to a seminal paper by Perry and Wolf [118], it is interesting to observe that similar concepts had already been introduced in [126] (1977), and that some results of research on viewpoints were presented in 1992 [55].

The concept of stakeholders' viewpoints and the views of software architecture was first generalised by the standard IEEE 1471 "IEEE Recommended Practice for Architectural Description" and its contemporary successor international standard ISO/IEC/IEEE 42010:2011 "Systems and software engineering — Architecture description". This standard is presented in detail in section 2.3.1. The research on architectural description resulted in defining sets of viewpoints (sometimes called viewpoint frameworks), which can be seen as instantiations of the general architecture description model defined by ISO/IEC/IEEE 42010:2011 standard. These instantiations address the information needs of various stakeholders, hence, they define sets of various viewpoints.

Popular models of architecture description, such as, 4+1 Views by Kruchten [94], Zachman's framework [163] are presented in sections 2.3.2 and 2.3.3 respectively. The recent developments in the genre of viewpoint-based architectural description are presented in section 2.3.4. Let us note that many various sets of viewpoints are defined by architecture frameworks, e.g. DoDAF [50], Reference Model for Open Distributed Processing [71] etc. These frameworks, categorise architectural information, but do not recommend certain notations to be used for modelling software architecture from certain perspectives.

2.3.1. Standard ISO/IEC/IEEE 42010:2011

Different stakeholders may have different concerns, hence they may be interested in different aspects of software architecture (e.g. data flow, control flow, component interaction and composition, source code organisation, component-to-hardware allocation, etc.), and may need information at different levels of detail. This has been reflected by the standard of architectural description – ISO/IEC/IEEE 42010:2011. Architecture stakeholders can include the following groups of people: users, operators, acquirers, owners, suppliers, developers, builders and maintainers.

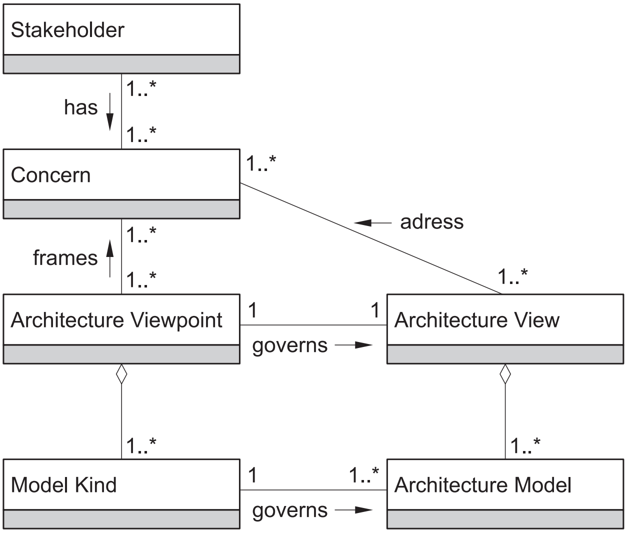

Figure 7. A fragment of the conceptual model of an architectural description, based on ISO 42010:2011

According to this standard, the architecture description should be organised around **architecture viewpoints** combining architecture views, concerns and model types (figure 7):

- **Concern** – is an abstract notion that denotes something of interest to one or more stakeholders, e.g. a system's property, a kind of information, etc. Every concern should be framed by at least one viewpoint. Concerns may be related to the system's goals, the adequacy of the architecture for the system's goals, the feasibility of the system's development project, the risks and impact on the stakeholders, or they may simply reflect the need for a certain kind of information (e.g. data exchanges between the system's sites, or the allocation of functions to the system's sites);

- **Architecture View** – is a model or a set of models expressing the architecture of a system from the perspective of specific system concerns;
- **Model types** – define available types of models that can be included in views.

Architecture viewpoint defines how to "construct, interpret and use architecture views in order to frame specific concerns." It should specify: concerns framed by this viewpoint; stakeholders whose concerns are framed by the viewpoint; model kinds used in this viewpoint; "languages, notations, conventions, modelling techniques, analytical methods and/or other operations to be used on models" for every model used in the viewpoint. The standard also includes a viewpoint template presented in [38].

Let us note that standard ISO 42010:2011 includes also the following notions:

- **Architecture framework**, i.e. modelling methodology for a specific domain and/or community of stakeholders; it should include at least one concern, stakeholder and viewpoint related to the concerns; architecture framework is, in fact, a set of architecture views together with their corresponding entities (concerns, model types, views, etc.);
- **Architecture description language (ADL)**, i.e. model kinds that frame certain concerns of the stakeholders, rules of correspondence between models of various kinds. Note that the standard does not recommend any specific modelling language to be used;
- **Architectural decisions and rationale** that are recommended to be included in an architectural description;
- **Architecture description element** is a generalisation of all the components of an architectural description, as defined by the standard (stakeholder, view, viewpoint, model, model kind).
- **Correspondence and correspondence rules** are a means of establishing the relations between architecture description elements and verifying their consistency. The standard simply defines these notions, but does not recommend any concrete form of their representation, or a consistency analysis.

2.3.2. Four Plus One Views of Software Architecture

In his famous paper "4+1 views of software architecture" (1995) [94], Kruchten proposed to represent software architecture using the views summarised in table 4.

Kruchten's framework reflects the practice of architecture modelling using UML, as it indicates models to be used for documenting software architecture.

The framework does not establish correspondence rules defining compliance between the included models.

Table 4. 4+1 views of software architecture by Kruchten

View	Stakeholder	Concern
Logical view	End users	Functionality
Development view	Programmers	Software development
Process view	System integrators	Performance, scalability, throughput
Physical view	System engineers	System topology, delivery, installation, telecommunication
Scenarios	Architect	Consistency between the models belonging to different views.

2.3.3. Zachman's Framework

Zachman's framework [163] categorises information describing software architecture according to a set of pre-established criteria, namely the subject of information and the level of scope related to the stakeholder who has an interest in that kind of information (compare the framework summary in tables 5 and 6). Let us observe that Zachman's framework was proposed back in 1987, while the famous paper by Perry and Wolf comes only from 1992. The framework has a number of versions, with the most current being No 3.0, called "The Enterprise Ontology".

Table 5. Zachman's framework: level of detail and their relevant stakeholders

View	Stakeholders	Concern
Scope, context	Executives	Identification. Lists.
Business concepts	Business Management	Definition. Business-level models.
System logic.	Architects	Representation. Models of system logic.
Technology physics	Engineers	Specification.
Tool components	Technicians	Configuration
Operations instances	Users	Instantiations

Table 6. Zachman's framework: categories of architectural information

Level of detail	Description
What	Concerns data that fuels the modelled system, including documents, data structures and databases.
How	Represents information processing from business processes to components implementing a certain functionality.
Where	Concerns a system's geography, topography and topology, including networks.
Who	Concerns organisations, roles and people.
When	Concerns timing properties.
Why	Concerns motivation underlying the related architectural information.

Zachman's framework should be tailored to the specific needs connected with a concrete modelling purpose. This means that the framework's users should chose information categories to be included in an architectural description, and the models that should be used as a career for each information category. It does not recommend any modelling notation and does not define any correspondence rules between various categories of information defined by the model. Zachman's framework, compared to Kruchten's 4+1 views, extends the strictly technical information on software architecture with broad information on system's context (organisational, business, motivational). This makes Zachman's framework suitable for modelling architectures of large-scale systems, while 4+1 views are adequate for capturing architecture of software applications.

2.3.4. Recent Achievements in Viewpoint-based Architecture Modelling

Views and viewpoints have become a separate area of interest for both the architecture community and practitioners. This research was aimed at extending the scope of information comprising an architectural description or focusing on certain specific issues. Let us consider some of the recent developments in the "viewpoint research":

- the inclusion of architectural decisions into an architectural description was proposed in [154]. A set of viewpoints representing architectural decisions has been defined. It includes the following viewpoints: decision details (contents – compare section 2.2.1), architectural decision relationship viewpoints (encompasses relations between architectural decisions – section 2.2.2); decision chronology (captures changes made to architectural decisions, and by the same their evolution), and decision stakeholder involvement viewpoints (reflect the roles of the stakeholders in the decision-making process);
- combining architectural decisions with the forces (factors) that influenced architects to make these decisions is the core concept of the viewpoint for forces on architectural decisions [153]. This viewpoint focuses on capturing the motivation underlying the architectural decisions;
- the variability viewpoint [146] focuses on architectural information regarding mechanisms supporting software variability, namely on variability points included in a software's design;
- the business goals viewpoint [37], which frames the business goals that drive an architecture. Business goals are represented as goal subject (stakeholder concerned with a goal), goal object (architectural entities to which the goal applies), environment (legal, social, technical, customer aspects concerning the goal), goal, goal measure and pedigree (origin of the goal). This is yet another approach to capturing the motivation underlying a software architecture. Naturally, business goals have to be appropriately associated with the relevant

models of software architecture representing architectures motivated by these goals. Let us note that business aspects of a system's architecture was already included in Zachman's framework [163].

A complete presentation of the multitude of views and viewpoints that have been developed in the past 20 years exceeds the scope of this monograph. Apart from the classic 4+1 views by Kruchten (section 2.3.2), Zachman's framework (section 2.3.3) and generic The Open Group Architecture Framework (TOGAF 9.1 [151]), and some domain architecture frameworks, most of the developments to date have not become pervasive in the industry.

2.4. DISCUSSION: STATE-OF-THE-ART ARCHITECTURAL MODELLING

Three waves of architectural modelling are presented in this chapter: models of software structure, architectural decisions, and models of architectural descriptions.

The models of software structure strive to represent software components and the relationships between them. Sufficiently expressive models could enable and support the analysis of architectural properties. If combined with certain viewpoints, they would enable an analysis of a system's properties, being the stakeholders' concerns. However, the practice of architectural modelling is dominated by semi-formal models, such as UML, SysML or Archimate. Each of these provides a uniform language in which different aspects of architectural design can be expressed. Therefore, they make it easier to document, present, understand and transfer architectural information between the architecture stakeholders. However, reasoning about the properties of architectures documented with these models can be done mainly on the basis of expert judgements. Model-based analysis can be performed with the appropriate formal models. However, early ADLs, many of which contained an underlying formal semantics, were abandoned in their infancy, while the semi-formal models are not amenable to model-based analysis, as they do not include formal models suitable for that purpose.

The above condition significantly constrains the analysability of software architectures, hinders the development of architecture evaluation methods (compare section 3.10), and limits the benefits resulting from the viewpoint-focused architecture descriptions.

Architectural decisions, supplement the explicit architectural knowledge, which can be represented with the models of software structure, with tacit knowledge on rationale and the logic of the architecting process. Although the content of architectural decisions seems to be agreed by the software architecture community, the means of managing large sets of architectural decisions are still not mature enough for industrial use. Architectural decisions are multifaceted and often deal with crosscutting-concerns. This hinders the development of unambiguous classifications of both architectural decisions and the relations between them.

Architectural decisions alone do not facilitate dealing with a software system's complexity [168]. On the contrary, they add to the software's complexity rather than minimising it. As a result, architectural decisions alone cannot currently be used for documenting software architecture, and their efficient use as a model of software architecture remains still an open research challenge. They can be used as an auxiliary modelling tool rather than as a carrier for large amounts of architectural information, especially if it should concern details of software design.

While ineffective in representing design details, architectural decisions turn out to be an effective means of capturing and explaining complex design entities, if combined with architectural patterns or focused on a predefined set of architectural decisions. The concept of the System Organisation Pattern confirms the above observation. It also indicates a promising direction of research, which could lead to the proposition of similar documentation schemes that could efficiently capture architectures of other kinds of systems and at other levels of detail.

A lot of research effort has been devoted to the development of the models of architecture description built upon the concept of viewpoints representing stakeholder concerns. The concept of a viewpoint-focused architecture description, initially promoted by the IEEE 1471 standard, seems to have gained appeal with both researchers and practitioners. The ISO 42010:2011, which is a refined version of IEEE 1471, has been followed by such popular developments as Archimate, TOGAF, and SysML. The general limitation of the hitherto research is that it delivers more and more categories of architecturally relevant information, without adequately extending the available models of software architecture. In this way, newer and newer viewpoints either define auxiliary information that has to be integrated with other models of software architecture, or provide for indexing or "zoom-in" capability to architectural information targeted at identifying and extracting specific information.

Certain recent developments seem to be addressing this very issue by integrating modelling notation with a set of viewpoints – compare Archimate (section 2.3.4).

The main research challenges for the near future are:

- further integration of research developments belonging to the three waves of architectural modelling, in order to deliver an integrated modelling framework that would organise models of software structure and architectural decisions around a set of viewpoints;
- Enhancing the expressive power of architectural models in order to enable an analysis of quality attributes. This could enable a model-based analysis of quality attributes such as performance or reliability, and would provide for

capturing such concerns as reliability, modifiability and reliability within the viewpoint-focused architecture descriptions;

- Developing correspondence rules defining the consistency conditions for the set of models comprising an architectural description. The lack of these rules is one of the traditional weaknesses of semi-formal modelling, even though such rules were included as an integral element of the model of architecture description of ISO 42010:2011.

3. ARCHITECTURE EVALUATION METHODS

The importance of software architecture results from its two basic features:

1. Software architecture influences many properties of a software system, including its quality attributes (compare the quality model of ISO/IEC 25010:2011 [73] or ISO 9126 [74] standards), its ability to compete with market competitors (see [11], [52]), its buildability [171], [169] and more;
2. Software architecture is a long-term asset. It is difficult, costly and risky to change the architecture of an already implemented system. Therefore, flawed software architecture usually remains unchanged until a major reconstruction takes place, which is similar to a vendor lock-in syndrome described in [166].

The purpose of architecture evaluation methods can be generally defined as being to analyse how software architectures influence software properties. Architecture evaluation methods are applied in order to verify whether a software architecture is suitable for a given system (its type, technology, goals, environment etc.) and does not contain flaws that may imperil the system's development or the achievability of the quality requirements, or that may impede maintenance and evolution.

The evaluation of software architecture can be of significant value for a system's stakeholders, especially if it is done early in the system's life cycle, when the architecture can easily be altered. This explains why architecture evaluation methods have become an area of an extensive research since the appearance of the Software Architecture Analysis Method (SAAM) [83] in 1994.

This chapter contains a critical study of the hitherto research in the area of architecture evaluation. First, the two basic paradigms (architecture verification and review) of architecture evaluation are discussed in section 3.1. The taxonomies of architecture evaluation methods are discussed in section 3.2. This leads to the proposition of an original taxonomy based on a method's application area in section 3.3. A uniform template for the description of architecture evaluation methods has been introduced in section 3.4, on the basis of existing comparison frameworks. Eighteen state-of-the-art architecture evaluation methods are presented in sections 3.6–3.9, preceded by a short presentation of the legacy architecture evaluation methods (section 3.5). This enables a discussion of the state of the art in section 3.10 and identified research opportunities in section 3.11.

3.1. PARADIGMS OF ARCHITECTURAL EVALUATION

Architecture evaluation methods, in many aspects, resemble the established methods of the software quality assurance, namely walkthroughs, reviews and audits, compare e.g. [130], [132], [107], [115], [145].

Two basic paradigms of architectural evaluation have emerged from the research carried out to date:

1. Assessing architecture against a set of relevant requirements (figure 8a) – architecture is analysed to show whether it sufficiently supports the relevant quality requirements.
2. Searching for architectural flaws (figure 8b) – are about searching for flaws included in software architecture.

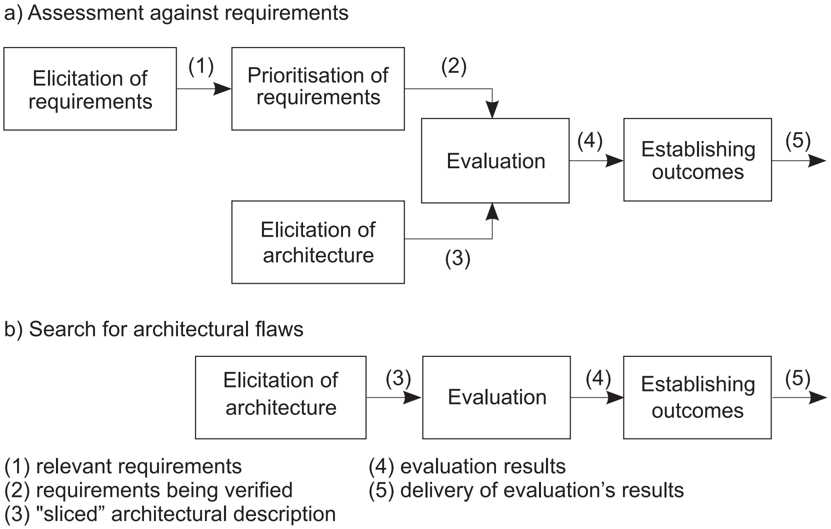
a) Assessment against requirements
Elicitation of requirements
(1)
Prioritisation of requirements
(2)
Evaluation
(4)
Establishing outcomes
(5)
Elicitation of architecture
(3)
b) Search for architectural flaws
Elicitation of architecture
(3)
Evaluation
(4)
Establishing outcomes
(5)
(1) relevant requirements
(2) requirements being verified
(3) "sliced" architectural description
(4) evaluation results
(5) delivery of evaluation's results

Figure 8. Architecture evaluation paradigms and the workflow resulting from them

The evaluation paradigm largely determines the internal design of the evaluation methods following that paradigm. The general assumption underlying the "assessing against requirements" paradigm is that software architectures can be neither objectively good, nor objectively wrong. Architecture suitable for one system may be inappropriate for another, for example: a complex, highly coupled modularity may hinder a software's maintainability, at the same time it can support its market position by making a software product more difficult to comprehend and be imitated by market competitors. Therefore, architecture evaluation should provide evidence that the software architecture is suitable for the business goals that the system should support. These may include concerns such as growth and continuity, meeting financial objectives or constraints, meeting personal objectives, responsibility to society, country or shareholders, managing market position, managing product quality and reputation etc., compare also the business goals viewpoint in [37].

In order to verify whether the architecture adequately supports business goals, they have to be expressed in terms of relevant requirements that can be verified by analysing a software's architecture. Most methods take into account certain non-functional requirements, such as usability, modifiability, performance, reliability and security. Therefore, the identification of architecturally significant requirements is the first step of an evaluation in the "assessment against requirements".

Let us note, that quality scenarios are the most popular way of representing quality requirements for an architectural evaluation. Quality scenario represents a quality requirement together with the conditions or the situation, in which it should be met. This is supposed to provide for an unambiguous and testable requirements specification. Architecture evaluation methods tailor the content of quality scenarios to their particular needs.

Knowledge of business goals is needed if requirements relevant for these goals are supposed to be elicited. Business goals may be explicitly identified (as in ATAM) prior to eliciting the requirements, or it may be assumed that the evaluation participants already have sufficient knowledge on business goals and reflect it in the elicited requirements.

The elicitation of requirements may produce a large number of requirements. Analysing all of them may be a daunting and time-consuming task. In order to make architecture analysis achievable within the allocated time, it is usually necessary to choose for analysis only a subset of the initially elicited requirements, which should be sufficiently representative for the needs of the architecture stakeholders. This is the purpose of the "prioritisation of requirements" step.

In order to analyse software architecture, it has to be presented in terms that enable reasoning about the influence exerted by the architecture on the software properties. The general challenge is to gather information defining software architecture on a level of detail suitable for a given analysis, to identify its parts that influence the analysed requirements, and to gather the information necessary for the evaluation techniques. This is the purpose of the "elicitation of architecture" task, which can be done on the basis of formal architectural documentation, requirements specifications or verbal descriptions. This activity is often aimed at identifying architectural approaches, patterns, tactics, or decisions included in software architecture. Let us note that, even if a formal architecture description exists, the identification of architectural patterns, tactics etc. included in software architecture requires some knowledge that is external to the architectural description. The existing architecture evaluation methods are not very prescriptive in this respect; the "sliced" form of an analysed architecture generally depends on the evaluation method, its goals and analysed properties.

The "evaluation" step is the very essence of every architecture evaluation method. In this step, *architecture evaluation techniques* are applied in order to assess the influence of the architecture on the analysed requirements. Naturally, this is the component that varies among the architecture evaluation methods.

Architecture evaluation techniques are analysis tools used within the workflow of the architecture evaluation method in order to establish how the architecture influences the software's properties. They have been classified in [2] as questioning techniques (e.g. scenarios, questionnaires, checklists) and measuring techniques. The latter include software metrics (compare e.g. [36], [67], [120], [124]), simulations, prototypes, experiments as well as model-based analysis techniques, such as queuing networks, rate monotonic analysis models [89], reliability analysis models and methods [159], concurrency models (e.g. CSP [70], LOTOS [100], and other process calculi, Petri Nets [80]) etc.

In many cases, the results produced by the architecture evaluation require some further processing before they can be presented to the evaluation audience, which is the purpose of the "establishing outcomes" step. This may be just about prioritising outcomes according to their importance (e.g. SAAM), relating them to the relevant business goals (e.g. ATAM) or transforming the values of metrics into the conclusion of an evaluation (e.g. TARA). Methods assume various forms of presenting results, e.g. report or visual presentation.

The "search for architectural flaws" paradigm (fig 8b) does not demand that requirements be elicited and prioritised before the actual analysis of the architecture. Therefore, such activities are absent in its workflow. However, in order to discover flaws in the evaluated architecture, it is necessary that information on the architecture be elicited in a form suitable for the evaluation, which is the role of the "elicitation of architecture" step. Its results are generally similar to those of a corresponding activity of the "assessment" paradigm. The evaluation is done by walking through an appropriately "sliced" architecture in search of flaws (compare, for example, PBAR). Detected flaws may impede some of the quality requirements, which is included in the evaluation's conclusion.

Finally, although the existing evaluation methods define a variety of evaluation processes, all of them include the overarching logic of architectural evaluation captured by the evaluation paradigms presented in this section.

3.2. TAXONOMIES OF ARCHITECTURE EVALUATION METHODS

Although there are more than thirty architecture evaluation methods, only a few taxonomies of such methods exist. Barcelos and Travassos [13] have classified architecture evaluation methods according to the evaluation techniques, they use. They defined categories of methods based on questioning techniques, on measuring techniques and on hybrid techniques, which follows the classification of evaluation techniques presented in [2].

Breivold and Crnkonvic [29] applied similar classification criteria and defined categories of experience-based, scenario-based and metric-based architecture evaluation methods. In a survey paper [90], only scenario-based and metric-based categories have been differentiated.

The general drawback of the evaluation technique-based classifications is that many architecture evaluation methods (e.g. ATAM, PASA, TARA, HoPLAA) use a number of architecture evaluation techniques, belonging to various categories.

A more elaborate classification scheme has been proposed by Roy and Graham [127], who first categorise architecture evaluation methods according to the development stage at which they are applicable, namely before implementation (early methods) or after implementation (late methods), and then, according to the evaluation techniques used by these methods. Hence, early methods have been classified as scenario-based and mathematical-model based, while late ones as metrics-based and tool-based. A controlled experiment-based category has been distinguished for both early and late methods applicable to architectural styles or design patterns. This taxonomy, apart from inheriting the weaknesses of evaluation technique-based classifications, has a further three limitations:

1. The taxonomy confusingly mixes architecture evaluation methods such as ATAM, SAAM and ALMA, with architecture evaluation techniques (model-based, metric-based); The methods belonging to the "search for architectural flaws" paradigm fall outside of the proposed classification. It is little wonder that they have been omitted in the taxonomy;
2. Early methods are generally also applicable to the implemented software, and some metric-based techniques (attributed to the "late" category), especially based on architecture-level metrics, can also be applied before implementation, e.g. to component-connector models.

Let us note that expressions such as "scenario-based", "lightweight", and "early", together with their default antonyms "non-scenario-based", "comprehensive" and "late" (used less frequently), provide for the most popular, common classifications of architecture evaluation methods. However, they are still vague. The vast majority of the architecture evaluation methods are scenario-based. Criteria indicating which methods are lightweight and which are not have not yet been defined (I try to introduce such in section 3.7). Finally, the term "early" is also not a precise one, as “much earlier” methods than most of the scenario-based ones can be defined, as has been shown in section 3.9.2.

3.3. APPLICATION AREA-BASED TAXONOMY OF ARCHITECTURE EVALUATION METHODS

In order to resolve the inherent ambiguities of the existing taxonomies, a novel taxonomy has been proposed (figure 9). It categorises the architecture evaluation methods according to the scope of their applications, dividing them into two basic classes:

- **General-purpose methods,** whose area of application is not restricted to any property or to any kind of software systems to be assessed;
- **Special-purpose methods,** which have been designed in order to be applied for the analysis of chosen quality attributes (attribute-specific methods subclass) or a given class of software systems (application domain-specific methods subclass).

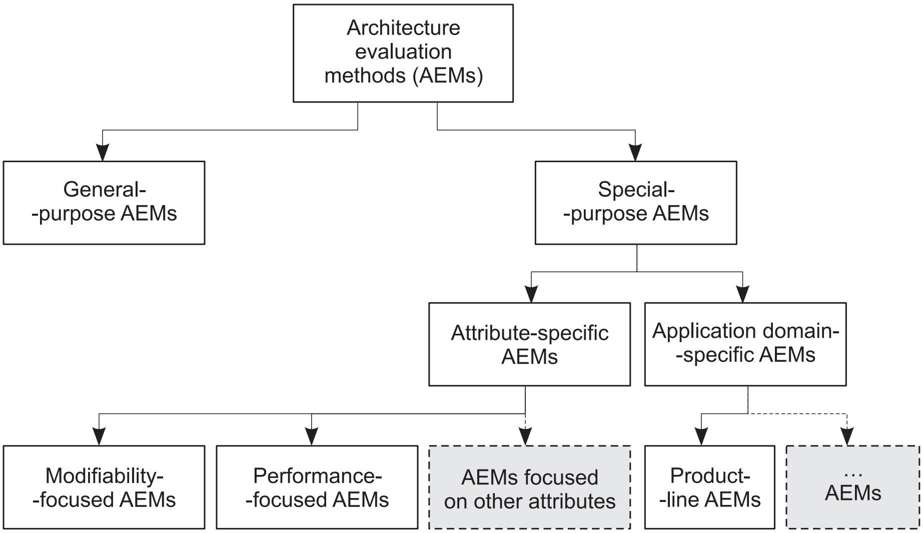
Architecture evaluation methods (AEMs)
General- -purpose AEMs
Special- -purpose AEMs
Attribute-specific AEMs
Application domain- -specific AEMs
Modifiability- -focused AEMs
Performance- -focused AEMs
AEMs focused on other attributes
Product- -line AEMs
… AEMs

Figure 9. Application area-based taxonomy of architecture evaluation methods

The taxonomy provides for an unambiguous classification of existing architecture evaluation methods. It is useful both for researchers and practitioners. It helps the latter to seek architecture evaluation methods that best suit their needs. The former can easily note the research opportunities, for example, it appears that the number of domain-specific methods does not correspond with the diverse kinds of software systems, and the choice of methods of analysing given quality attributes is limited to just one or two methods. A classification of architecture evaluation methods included in this survey has been shown in table 7. The evaluation paradigms define a dimension orthogonal to the categories presented above, which has also been shown in table 7.

Table 7. Architecture evaluation methods classified according to the application area-based taxonomy

Class	Subclass	"Assessment against requirements" methods	"Search for architectural flaws" methods
General-purpose	–	SAAM, SBAR, ATAM, Lightweight ATAM, CBAM, APTIA	PBAR, SHADD, TARA, ARID
Special-purpose	Attribute-specific	Usability: SALUTA Modifiability: SAAM'94 [83], SAAMCS, ESAAMI, ALPSM, ALMA Performance: PASA, CPASA Evolvability: SAAMER, AREA Modularity: ASAAM Reliability: SARAH	–
	Application domain-specific	HoPLAA	EAEM

3.4. CHARACTERISING ARCHITECTURE EVALUATION METHODS

Architecture evaluation methods are elaborate concepts that can be characterised by many properties. The sets of such features have already been proposed in papers [49], [8] and [84], which present comparison frameworks for architecture evaluation methods. They contain just flat lists of features concerning various aspects of architecture evaluation methods. The methods' properties included in the comparison frameworks have been grouped below according to the aspect of architecture evaluation methods they concern. Hence, the properties of architecture evaluation methods have been categorised as shown below:

1. Method design
   a. Evaluation goals – the purposes for which architecture evaluation methods can be applied;
   b. Method inputs:
      i Form of architectural description – form of architectural description needed for the analysis (e.g. formal, informal, particular ADL, etc.).
   c. Method internal logic:
      i. Examined properties – a list of properties (mainly quality attributes) subject to evaluation;
      ii. Evaluation process – defines the activities needed to perform an evaluation;

  iii. Evaluation participants – the roles of people that should participate in an evaluation and their participation in the appropriate steps of architecture evaluation;
  iv. Representation of evaluated requirements;
  v. Techniques for eliciting requirements;
  vi. Techniques for prioritising requirements;
  vii. Architecture evaluation techniques;
 d. Method outcomes – the evaluation results produced by a given method;
2. Method properties resulting from its design:
  i. Applicability stage – phases, at which an evaluation method can be applied;
  ii. Level of effort – the empirically measured level of effort required to perform an evaluation;
  iii. Reuse of knowledge – techniques supporting the reuse of knowledge and experiences gathered during earlier evaluations;
  iv. Support for non-technical issues (e.g. social, managerial);
3. External features:
 a. Tool support – the existence of tools supporting a given architecture evaluation method;
 b. Method maturity – a record of the method's applications.

The properties characterising method design have become the basis for a uniform description framework for architecture evaluation methods, which will be used in a survey presented in sections 3.6–3.9, while the properties resulting from method's design and external features will become a basis for the discussion of method's properties in section 3.10.

The uniform description framework for architecture evaluation methods comprises:

1. Method highlights – this section underlines the distinguishing features of a method, and presents the challenges that the method resolves.
2. Evaluation goals – the methods of eliciting the goals of the evaluation or (more commonly) listing the goals of the evaluation that are built-in into the method are presented here.
3. Examined properties – this section contains a list of software properties that are examined by a given architecture evaluation method. In most cases, these concern non-functional quality attributes such as usability, performance, reliability and modifiability (compare [16]).
4. Evaluation process – a description of the workflow of a method, comprising an ordered list of activities with a short description of each, if needed to comprehend the method logic.
5. Architecture description – this section indicates the form of the description of the evaluated architecture that is required by a given evaluation method.

6. Representation of evaluated requirements – architecture evaluation methods usually assess architecture against a set of requirements. These may take various forms, from informal statements to formalised quality scenarios of ATAM. In some cases, such as AREA (section 3.8.5), SARAH (section 3.8.6), some equivalents of the requirements are used in order to focus the analysis. Both are described in this section.
7. Techniques for eliciting requirements – the techniques for identifying architecturally relevant requirements or their equivalents (in the case of some methods) are presented in this section.
8. Techniques for prioritising requirements – in most cases, the scope of the architecture evaluation has to be limited in order to make it feasible in terms of time, effort and cost. This usually means that only requirements that exert a prevailing impact on the analysed properties have to be chosen for the evaluation. The prioritisation techniques provided by a method are presented in this section.
9. Architecture evaluation techniques – there are various techniques that can be used to analyse software properties at an architectural level. Architecture evaluation methods employ many of them, and often propose new ones. This section describes the details of the evaluation techniques, and presents how architecture is analysed in order to arrive at the outcomes of the evaluation.
10. Evaluation outcomes – types of outcomes that are produced by an evaluation method.
11. Method limitations, discussion – the limitations and peculiarities of a method.

3.5. NOTE ON THE LEGACY ARCHITECTURAL EVALUATION METHODS

The early developments in the area of architecture assessment have been comprehensively presented in a survey [49] covering the methods described in table 8. They have been included in this monograph for the sake of completeness of the presentation.

Table 8. Early developments in architecture evaluation

Method's acronym, full name	Analysed attributes, Short description
SAAM – Scenario-Based Architecture Analysis Method [83]	Initially [83] SAAM was used for analysing modifiability, later [39], SAAM was later extended onto any subset of the quality attributes. For details of the method see section 3.6.1.
SAAMCS – SAAM Founded on Complex Scenarios [98]	Flexibility (related to modifiability), scenario-based method, focuses on scenarios that are difficult to achieve, and therefore could cause major risks.

Table 8 cont.

Method's acronym, full name	Analysed attributes, Short description
ESAAMI – Extending SAAM by Integration in the Domain [106]	Mainly modifiability, scenario-based method which promotes the reuse of domain-specific knowledge captured with analysis templates (reusable fragments of architectures, concept similar to architectural patterns)
SAAMER – Software Architecture Analysis Method for Evolution and Reusability [101]	Evolution/reusability, scenario-based method with scenarios capturing the anticipated changes
SBAR – Scenario-Based Architecture Reengineering [21]	Multiple, uses different analysis techniques depending on the analysed attributes (e.g. mathematical modelling, simulation)
ALPSM – Architecture Level Prediction of Software Maintenance [20]	Maintainability, scenario-based method with scenarios capturing the anticipated changes enforced by the maintenance activities
SAEM – A Software Architecture Evaluation Model [51]	Multiple attributes; employs metric-based evaluation technique based on GQM

Most of these early developments have become the foundations for more mature evaluation methods, namely SAAM, SAAMCS, ESAAMI, SAAMER were later included into Architecture Trade-off Analysis Method (ATAM), ALPSM evolved into ALMA, and some have been left dormant (SAEM, SBAR).

3.6. SURVEY OF COMPREHENSIVE GENERAL-PURPOSE ARCHITECTURE EVALUATION METHODS

According to the taxonomy presented in section 3.3, general-purpose architecture evaluation methods are characterised by the following properties:

- They can be used in order to analyse any software architecture, i.e. they are not oriented on a chosen application's domain;
- They are supposed to be used for analysing architectural support for any subset of the quality attributes.

3.6.1. Software Architecture Analysis Method (SAAM)

Method highlights: SAAM is most probably the first architecture evaluation method compliant with the contemporary meaning of this term and with the assessment against requirements paradigm. The method was published several times, compare the original paper [83], and [86], [39]. Note that the word "scenario" was not used in the original paper on SAAM [83], though, SAAM was already a scenario-based method. The presentation of SAAM was based on the most recent account of the method presented in [39]. In SAAM scenarios are analysed by searching an architecture for components, which are responsible for achieving properties requirements represented by these scenarios.

Evaluation goals: the goals of SAAM-based analysis are not as carefully defined as in ATAM or ALMA. SAAM can serve various purposes, including identifying architectural weaknesses (requirements that are not sufficiently supported by the architecture) and the comparison of architectures.

Examined properties: mainly modifiability, though other attributes may also be analysed.

Evaluation process: The activities of SAAM's evaluation process comply with those described in section 3.1:

1. Develop scenarios;
2. Describe architecture;
3. Classify and prioritise scenarios;
4. Individually evaluate indirect scenarios – see "architecture evaluation techniques" section;
5. Assess scenario interactions – see "architecture evaluation techniques" section below;
6. Create an overall evaluation

Architecture description: SAAM assumes a component-connector model of software architecture, supplemented with necessary information regarding data and a system's dynamics, the latter in any understandable form (e.g. diagrams and text). In practice, the development of scenarios and architecture description can be performed iteratively because it is often necessary to tailor the architecture's description to the specific needs of a concrete assessment.

Representation of evaluated requirements: quality scenarios representing the stakeholders' expectations with regard to the anticipated use of a system, for example: porting to another operation system, expected changes to a system's functionality, or a change of the development environment. No structure has been imposed on the quality scenarios, they are just simple statements. In a broader understanding, scenarios express in brief the stakeholders' activities, anticipated changes and other properties that the system should support.

Techniques for eliciting requirements: brainstorming, possibly in a number of iterations.

Techniques for prioritising requirements: classifying scenarios as direct or indirect in order to provide insights into how they are supported by architecture, then cumulative voting of the participants and selecting the top 30% of the elicited scenarios for a detailed inspection. *Direct scenarios* are those that are already addressed by architecture, while *indirect scenarios* are quite the opposite: the architecture has to be modified in order to ensure that the indirect scenario is supported.

Architecture Evaluation Techniques: informal, namely analysing the architectural description aimed at showing how direct scenarios are met by architecture and identifying changes to the architecture required in order to obtain architectural support for the indirect scenarios (step 3). This is followed by an analysis of the interaction of indirect scenarios (step 4): two scenarios *interact* if the same component has to be changed in order to satisfy two or more indirect scenarios. Too many such interactions may indicate poor architectural design, i.e. poor functionality allocation resulting in low cohesion and high coupling. It may also indicate that the architecture has not been analysed at an appropriate level in terms of the scope, and that the architecture's description should probably be refined in order to perform an adequate evaluation.

Techniques for establishing outcomes: SAAM is rather sketchy in this respect. It has been suggested in [39] that scenarios should be weighted by their importance, where the measure can be the anticipated cost of changes, level of risk or some other criteria agreed by the stakeholders. The weighting is particularly useful when comparing two or more alternative architectures.

Evaluation outcomes: the findings established during the analysis, which may include: scenarios that are difficult or costly to become supported by architecture, risks connected with some scenarios, cost of modifications.

Methods validation: number of applications difficult to estimate, certainly exceeds ten.

Method limitations, discussion: SAAM, especially when compared to ATAM, offers an unsophisticated evaluation process. It directly addresses the participation of various stakeholders, as well as the need to identify and prioritise stakeholders' concerns. Evaluation techniques (a distinction between direct and indirect scenarios; scenario-interaction) are rather sketchy, meaning that the analysis relies heavily on the knowledge and experience of the participants.

3.6.2. Architecture Trade-Off Analysis Method (ATAM)

Method highlights: ATAM was first presented in 1998 [87]. The description of ATAM presented below is based on [39] and [16]. The prevailing goal shaping ATAM's "design" was the method's flexibility. It can be used for a variety of evaluation purposes (e.g. risk analysis, seeking architecture improvement opportunities, architecture comparison), which are neither presumed nor built-in into the method's design, but are established individually during every single analysis.

The general purpose of ATAM-based evaluation is to find out whether architecture adequately supports business goals driving architectural design. Therefore, although during an evaluation the viewpoints of various stakeholders are taken into account, business stakeholders' goals and concerns are treated with particular care. In fact, their concerns define a reference point for the entire evaluation. As stakeholders' concerns and goals may conflict with each other, ATAM provides techniques for explicitly managing (requirements prioritisation based on cumulative voting and utility tree) and resolving such conflicts (identifying trade-offs).

Evaluation goals: elicited from the stakeholders at the start of the evaluation (step 4).

Examined properties: non-functional quality attributes, for example: usability, modifiability, security, performance, availability, and testability. In fact, any property that is important for achieving business goals may be evaluated.

Evaluation process: a summary is contained in table 9.

Table 9. Summary of the steps in ATAM

Evaluation Step	Main participants	Goal	Outcomes
	Phase I	Presentation	
1. Present the ATAM	All stakeholders, ET[*]	To make the stakeholders aware of the ATAM evaluation process and its supporting techniques	–
2. Present the business drivers	All stakeholders, ET	To make the stakeholders aware of the business goals the system is developed for, as well as the architectural drivers (quality attributes shaping the architecture)	–
3. Present the architecture	Project decision makers, ET	To make the evaluation team familiar with the architecture, being evaluated	–
4. Identify the architectural approaches	Architects, ET	To identify architectural styles (patterns, tactics) included in an architecture design	List of architectural approaches/styles included in the architectural design
5. Generate the quality attribute tree	Project decision makers, ET	To establish the preferences of project decision makers concerning the quality attribute goals	Quality attribute tree. Initial list of prioritised quality scenarios
6. Analyse the architectural approaches	Architect, ET	To build up an initial overview of the relations between the software architecture and most important quality attribute goals. To identify initial analysis outcomes expressed in terms of risks, non-risks, sensitivity points and trade-off points	Initial lists of risks, non-risks, risk themes, sensitivity points, trade-off points
	Phase II	Testing	
7. Brainstorm and prioritise scenarios	All stakeholders, ET	To exploit the knowledge of all the stakeholders in order to get a broader and deeper view of the quality requirements	Final list of prioritised quality scenarios
8. Analyse the architectural approaches	Architect, ET	To extend and refine the outcomes achieved in step 6	Final lists of risks, non-risks, risk themes, sensitivity points, trade-off points
	Phase III	Reporting	
9. Present the results	All stakeholders.	To summarise the evaluation outcomes and relate them to the business drivers presented in step No 2	Evaluation outcomes presentation. Evaluation report

[*] *ET – Evaluation Team*

Architecture description: an architecture description is necessary for an ATAM-based evaluation. Architectural approaches, i.e. architectural tactics and patterns (compare section 2.2.5) used in software architecture should be identified during the evaluation (step 4).

Representation of the evaluated requirements: quality scenarios express the quality attributes in terms of concrete situations, which should be supported by software architecture. ATAM defines a precise template for documenting the quality scenarios, comprising the following elements:

- *Source* – the stakeholder, exerting a stimulus on an architecture;
- *Stimulus* – is a condition to which the architecture should respond;
- *Artefact* – is affected by a stimulus;
- *Environment – the* condition in which a stimulus occurs;
- Response – the response to/support for a given stimulus;
- Response measure – a measure of the desired response.

Let us consider, for example:

- a modifiability scenario: a group of users (source) wishes a new report (stimulus) to be included in a financial application (artefact) with its next build (environment). The architecture should enable such an extension to be implemented (response) within a week (response measure);
- a performance scenario: a user (source) requests the presentation of flight offers (stimulus), application (artefact) under normal operating conditions (environment) presents (response) a list of offered flights in no more than 30 seconds (response measure).

Techniques for eliciting requirements: although brainstorming isa requirements elicitation technique used in ATAM, the elicitation process is organised in a unique way. Requirements elicitation takes place after building up the participants' awareness of the business goals (step 2) and knowledge of the evaluated architecture (step 3). Quality scenarios elicitation is a two-phase process:

- in the "Preparation" phase (step 5), it is done with the project's decision makers (architect, customer, sponsor, the project's management, component designers) only. It results in building a utility trees with the quality scenarios in the leaves.

  The utility tree represents the internal structure of each of the quality attributes as it is understood by the project decision makers. General quality attributes, such as performance, reliability, are translated onto a set of more detailed properties, e.g. performance onto throughput and response time, reliability onto fault tolerance and availability. These more detailed properties may be translated onto even more detailed ones and so on. The depth of the utility tree is not limited formally, though it should obviously be kept reasonable if the tree is supposed to be comprehensible and useful. The development of a utility tree should go from general to specific.

- In the "Testing" phase (step 7), requirements are elicited from all the stakeholders, and a larger set of scenarios is elicited from all the stakeholders during a brainstorming session, which should foster the communication between stakeholders.

  The elicitation of scenarios is facilitated by introducing a classification of the quality scenarios and providing scenario templates for the essential quality attributes: usability, testability, interoperability, modifiability, performance and security [15], [16]. The scenarios have been classified into three basic groups:
  - Use-case scenarios – represents the ways in which stakeholder's are planning to use the system;
  - Growth scenarios – reflect the expected changes in the system's context and how the system should accommodate these changes. These changes may concern foreseen functionality modifications and the system's load;
  - Exploratory scenarios – scenarios of this kind are supposed to be kind of a stress test aimed at revealing the architecture's limitations. They should reflect extreme cases of changes to the system's operating condition, the system's mission, performance and reliability requirements. Their evaluation should discover more risks, sensitivity and trade-off points.

Techniques for prioritising requirements: the prioritisation of requirements is integrated with quality scenario elicitation, though different techniques are used in the "Preparation" and "Testing" phases:

- In the "Preparation" phase, the quality scenarios are prioritised in order to reveal their relative importance for the project decision makers. The advised technique is to assess scenarios along two dimensions: a scenario's importance for the system's success, and the difficulty of ensuring that the architecture supports a given scenario. The assessment should be done using a numeric scale or three-level ranking H/M/L (High/Medium/Low), which the ATAM's authors prefer because of its simplicity. The scenarios of a higher ranking express the requirements that are most important to the key project's stakeholders (high priority scenarios).
- In the "Testing" phase, the elicited quality scenarios should be prioritised using a cumulative voting [99] technique in which all the stakeholders take part. Comparing the priorities established in the "Preparation" and "Testing" phases can show whether the architects have a common understanding of design goals with the decision makers.

Techniques for assessing architecture: the general architecture analysis technique is the analysis of the influence on quality attributes inflicted by the architectural tactics and patterns included in a system's architecture, which should be done scenario-by-scenario. Both qualitative (e.g. experience-based reasoning) and quantitative (compare section 3.10.3) can be used. In fact, ATAM allows any architecture analysis techniques to be used if its choice is reasonable with regard to the goals and conditions of a given evaluation.

Evaluation outcomes: risks – architectural decisions that may negatively affect quality attributes, non-risks – safe architectural decisions, risk-themes – recurring kinds of risks, sensitivity points – decisions that may influence certain quality attributes, trade-off points – an architectural decision that affects more than one quality attribute at the same time and in the opposite directions (e.g. the choice of layers increases modifiability but reduces performance).

Method validation: an extensive evaluation record is available, compare [81], [54], [25], [139], [123] as well as evaluation examples in [39], [15]. The outcomes of 18 ATAM-based evaluations were summarised in [17], [18].

Method limitations, discussion: The Architecture Trade-off Analysis Method (ATAM) is perceived as the most sophisticated and the most mature architecture evaluation method. ATAM is a successor of SAAM, as it adopts and extends scenario-based evaluation paradigm. These extensions made to SAAM components have been depicted in table 10.

Table 10. How ATAM extends the concepts introduced by SAAM

	SAAM	ATAM
Addressing business goals	Unstructured, it was assumed that the participation of the stakeholders in scenario generation will ensure that business goals are reflected in the set of scenarios	Quality scenarios are supposed to translate business goals into the language of quality requirements captured with quality scenarios
Scenario structure	Unstructured scenarios briefly representing activities of the architecture's stakeholders	Quality scenarios extend the concept of SAAM's scenarios by providing a uniform quality scenario template.
Scenarios elicitation	Unstructured	Scenarios elicitation techniques – brainstorming facilitated by: • The quality scenario classification: use case scenarios, growth scenarios, exploratory scenarios; • The quality scenario templates for each of the quality attributes.
Architecture representation	Architecture is presented and comprehended in terms of components and connectors, though no modelling notation has been recommended	Architecture is presented and comprehended in terms of architectural tactics and patterns. No architecture model has been recommended, though many examples use some set of architectural views
Requirements (scenarios) prioritisation	Prioritisation with cumulative voting	Two-phase prioritisation: establishing quality attributes, which are of highest interest to the major stakeholders (decision makers) by the use of a utility tree, then juxtaposing the earlier outcomes with a broader view of all the stakeholders.

Apart from the important extensions to a scenario-based assessment paradigm, a distinguishing feature of ATAM is that it is the only method that tries to explicitly relate the elements of an architectural design to business goals driving that design. Despite its sophistication, numerous ATAM limitations are commonly recognised and triggered further research aimed at overcoming them. These limitations include [165]:

1. High level of effort (32-70 man-days, 2-6 weeks) and the cost of a fully-blown ATAM-based evaluation – compare [39];
2. The method does not indicate precisely how to relate the outcomes of an evaluation back to the business goals.
3. Quality scenarios used in ATAM effectively represent detailed requirements, but are often ineffective and even confusing when defining more general properties. For example, a work around is needed to represent a requirement that "the generation of reports should not interfere with the processing of transactions": Stimulus: transaction arrives; Source of stimulus: user; Artefact: system; Response: the system processes the transaction in no more than 2 seconds; Environment: the system (as a whole) is processing a report when the transaction arrives; Response measure: maximum transaction processing time. It is certainly not a straightforward way to put across the requirement to separate OLTP from OLAP.
4. The cumulative voting is known for its deficiencies, voting can give rise to numerous games played between the stakeholders in order to obtain preferred requirements priorities. It is observed in [99] that, in most cases, the method works well only once per project.
5. In the case of many real-world projects, a lack of architectural documentation together with the instability of the quality requirements make ATAM-based assessments impossible – compare [66];
6. Insufficient integration with the established development and project management practices. Such a time-consuming evaluation does not comply with the short cycles of agile methods' iterations. In the case of integrating ATAM with RUP, the elicitation of quality scenarios may overlap with the activities of the "Requirements" discipline of RUP.

3.6.3. Analytic Principles and Tools for the Improvement of Architectures (APTIA)

Method description: APTIA [85] is an extension to ATAM, aimed at extending ATAM-based analysis by an in-depth investigation of risk themes identified during ATAM analysis in order to identify design alternatives and to choose among them. The method promotes the use of quality attribute models in order to analyse architecture and propose alternatives on the basis of the outcomes of such a model-based analysis. The choice among the alternatives should be made by analysing cost and benefit related to the considered architectural options.

Evaluation goals: to identify and make choices among alternative architectural designs.

Examined properties: APTIA does not limit analysis to any presumed set of quality attributes.

Evaluation process: according to [85], APTIA comprises the following steps:

1. Perform an ATAM.
2. Determine the focus for analysis based on risk themes identified through ATAM.
3. Use quality attribute models related to the risk themes to understand the architecture.
4. Use insights gained from model-based analysis and design principles to propose alternatives.
5. Rank the alternatives based on costs/benefits.
6. Make design decisions.

Steps Nos 3–4 should be repeated for each of the considered quality attributes.

Architecture description: architectural description is needed to use APTIA as it *in fact* starts from performing an ATAM-based evaluation.

Representation of evaluated requirements: risk themes discovered during the ATAM provide the focus of an APTIA-based analysis.

Techniques for eliciting requirements: as with ATAM.

Techniques for prioritising requirements: not prescribed, left to the expertise of the evaluators.

Architecture Evaluation Techniques: a cost-benefit analysis of CBAM (compare section 3.8.8) is applied in order to rank the identified architectural alternatives; quality attribute models (e.g. Rate Monotonic Analysis [89], reliability engineering techniques [102], performance analysis techniques [12]), are promoted as a means of investigating how architecture affects the quality of the software.

Evaluation outcomes: identified architectural alternatives as well as cost and benefits related to those alternatives.

Method validation: the examples set out in [85] present an analysis of the performance with Rate Monotonic Scheduling [89] and variability with cost-benefit models.

Method limitations, discussion: APTIA combines the ATAM with analysis techniques for certain quality attributes in order to deepen the outcomes of an ATAM-based analysis. However, the evaluation process is rather loose, and the proper application of the techniques listed in [85] depends entirely on the expertise of the evaluator. The application of model-based architecture assessment techniques requires evaluators with skills in certain branches of mathematical modelling (e.g. stochastic modelling, Rate Monotonic Analysis, queuing models, reliability models, etc.). However, practitioners are rather reluctant to use advanced mathematical models during software development, as the survey [7] confirms.

3.7. GENERAL-PURPOSE LIGHTWEIGHT ARCHITECTURE EVALUATION METHODS

The limitations of comprehensive architecture evaluation methods, in particular ATAM, especially those directly perceived by users of the method as a high level of effort and cost, together with the lack of a convincing business case for the ATAM assessment [7], have triggered a wave of research on lightweight architecture analysis methods.

Although the criteria indicating which methods are lightweight and which are not have not yet been defined, they can be deduced from publications setting out proposals for such methods:

- Inexpensiveness in terms of time, effort, and cost;
- Minimum formality – the method provides for sketched evaluation steps rather than a formally defined evaluation process, such as in the case of ATAM;
- Minimum number of project stakeholders involved in the assessment;

Minimum scope of documentation needed in order to perform an evaluation, i.e. lightweight methods should not require a complete architecture description, nor a complete specification of requirements, and should not require other kinds of models. These methods should be able to accept partial documentation or to elicit the necessary information, *in situ*, during an evaluation.

3.7.1. Active Reviews for Intermediate Designs (ARID)

Method highlights: ARID [40], [39] refers to the classic concept of active design reviews [115], which is an established software quality assurance technique. The architecture is not evaluated in order to confirm its support for quality attributes and business goals, but in order to detect its weaknesses.

Evaluation goals: an assessment of an architecture's suitability through the discovery of architectural issues resulting from a software architecture, or its part.

Examined properties: ARID does not assume an analysis of any particular software properties, and they are not explicitly established during the analysis. The analyses focus on a set of properties represented by a set of quality scenarios.

Evaluation process:

Phase 1: Pre-meeting

Step 1: Identify reviewers

Step 2: Prepare design presentation

Step 3: Prepare seed scenarios (seed scenarios are presented by designers or the review facilitator to the evaluation participants in order to explain the concept of quality scenario; seed scenarios have not got to be used in the phase 2)

Step 4: Prepare for the review meeting

Phase 2: Review meeting

Step 5: Present ARID method

Step 6: Present design

Step 7: Brainstorm and prioritise scenarios

Step 8: Perform a review (evaluate architecture)

Step 9: Present conclusions

Architecture description: architecture is presented and discussed with the stakeholders; no particular form of architectural documentation has been assumed.

Representation of evaluated requirements: quality scenarios are the only carrier for the evaluated requirements, though no particular contents of quality scenarios are recommended.

Techniques for eliciting requirements: brainstorming.

Techniques for prioritising requirements: cumulative voting.

Architecture Evaluation Techniques: ARID is not very strict in this respect. Expert and experience-based reasoning are the main assessment techniques applicable in such an informal review.

Evaluation outcomes: list of issues (problems) connected with a given architecture;

Method validation: a pilot example presented in [40].

Method limitations, discussion: reviews can only reveal the existence of certain flaws in a software's architecture, but will not show that architecture adequately supports business goals. This, together with the assumption that ARID is supposed to be used for a single version of software architecture, enables the method's simplicity.

3.7.2. Pattern-Based Architecture Reviews (PBAR)

Method highlights: the Pattern-Based Architecture Reviews method was crafted in order to better suit production-focused projects than traditional ATAM-based architecture reviews [66]. Such projects are typical for many real-world organisations focusing on delivering operational software, rather than on developing extensive documentation and on strictly following a prescriptive development processes. They often use agile and lean software development methodologies [41], [42], which also limits the use of comprehensive evaluation methods.

Evaluation goals: the aim of such an evaluation is the identification of quality attribute issues – see the evaluation outcomes section below.

Examined properties: the analysis looks at the risks potentially affecting non-functional quality attributes.

Evaluation process: The evaluation with PBAR should be performed during a single, face-to-face meeting with the development team. This evaluation comprises the following five consecutive steps:

1. Establishing the most important quality requirements by analysing user stories and by the discussion with the developers;
2. Eliciting software architecture by discussing the software's structure with the developers;
3. Identifying architectural patterns;
4. Examining the influence of the identified architectural pattern on the quality attributes;
5. Identifying and discussing analysis outcomes, i.e. quality attribute issues.

Architecture description: no particular form of architectural documentation is assumed. Architecture is elicited during informal meetings, discussions and presentations with the development team. Architectural patterns included in software architecture have to be identified during the evaluation.

Representation of evaluated requirements: not prescribed.

Techniques for eliciting requirements: requirements are elicited informally, *in situ*, during an informal meeting with the development team.

Techniques for prioritising requirements: absent.

Architecture Evaluation Techniques: the architectural assessment is based on the known relationships between architectural patterns and the quality attributes (section 2.2.5). The evaluation is based on investigating the mismatches between architectural patterns included in the design and non-functional quality requirements. The assessment technique has not been formalised and requires knowledge on architectural patterns and their potential influence on software quality attributes. A common example of this is the negative influence of layers pattern on performance, or of the "pipes and filters" pattern on reliability (a filter is a single point of failure).

Evaluation outcomes: quality attribute issues, i.e. quality attributes that have not been properly addressed by the architecture, patterns that could have been included into the design, and conflicts between quality attributes and patterns included in the design**.**

Method validation: nine student capstone projects carried out with real customers.

Method limitations, discussion: PBAR bears all the features of the light-weight methods mentioned in section 3.7. The authors report that PBAR requires only 4 to 11 man-hours, a negligible amount when compared to ATAM's average of 32 man-days [39].

However, drawing conclusions with regard to the influence of patterns on quality attributes is not decisive in most cases, and in many cases is actually elusive and/or ambiguous. This can limit the applicability of pattern-based evaluation. For example, layers may impede software performance, though most contemporary internet applications follow this style; a message broker is a single point of failure, which does not hinder its popularity (compare, for example, enterprise service bus), but these performance and reliability risks are usually addressed by the appropriate system's design, namely by a resource contingency or redundancy respectively.

3.7.3. Tiny Architectural Review Approach (TARA)

Method highlights: TARA [157], [158] is supposed to be applicable in industry, where more complex evaluation methods and techniques often cannot be applied because of their intrinsic complication or the level of required participation by stakeholders. TARA relies on code analysis techniques and/or operational data, both requiring that software has already been implemented.

Evaluation goals: the goal of TARA evaluation is defined in [158] as "to establish how well suited a particular architecture is to supporting a set of key requirements."

Examined properties: functional (!) and non-functional requirements established individually for every assessment;

Evaluation process: TARA comprises seven steps:

1. System context and requirements – the evaluator builds his understanding of key requirements and context, in which the system exists,
2. Functional and deployment views – the evaluator builds his understanding of the system's context and design, developing or gathering a kind of architecture description;
3. Implementation analysis – assessment techniques are applied;
4. Requirements assessment – the results of the implementation analysis should be related to the identified requirements. Expert judgement is the main technique supposed to be applied here. Finally, levels of the evaluator's confidence in the system's ability to fulfil the requirements should be assessed for every identified requirement.
5. Identify and report findings – the report on the evaluation's findings is usually at least partially prepared in parallel with the evaluation activities (steps 3 and 4); however, in this step the outcomes of the evaluation should be gathered in a uniform and comprehensible form.
6. Deliver the findings and recommendations.

Architecture description: system context, functional and deployment structures have to be understood by the evaluator. The method does not assume any form of architecture description, though some notes or sketches can be used for that purpose.

Representation of evaluated requirements: a list of key functional and non-functional requirements.

Techniques for eliciting requirements: the practical advice was that an effective elicitation technique is to identify candidate requirements with the developers, then to validate and approve them with relevant stakeholders. Other techniques for eliciting requirements and sources of information on requirements (e.g. requirements specification documents) have not been excluded.

Techniques for prioritising requirements: integrated with requirement elicitation techniques, i.e. it is an expert choice made by an evaluator consulted with the stakeholders.

Architecture Evaluation Techniques: chosen as needed, no techniques are excluded in advance. The method that [158] promotes involves the use of automated code analysis techniques (module dependencies, size measures, code metrics (e.g. McCabe's metrics, comment to code ratio, etc.; see also [67]), test coverage. For software already deployed, information characterising software execution (e.g. event/incident/production logs).

Method validation: TARA has been used in an industrial context for a small number of exercises [158]**.**

Method limitations, discussion: The main limitation of TARA is its dependence on expert judgement with regard to drawing conclusions from the evidence gathered during the architecture analysis. TARA can be applied to already implemented software as long as code analysis techniques are to be used. This seems to be another important deficiency, as software architecture is hardly ever corrected when software is ready for deployment. Therefore, TARA's outcomes can only be applied in the maintenance phase.

3.7.4. Lightweight ATAM

Method highlights: The high cost of an ATAM-based evaluation results from the complicated evaluation procedure and extensive participation of stakeholders. Naturally, limiting both of these factors will diminish the total level of effort required for an evaluation, thereby minimising the cost of the evaluation. This idea has been put into force in Lightweight ATAM presented in [16], which should not require more than 4-6 hours. The method is supposed to be used internally by a development team that is already familiar with the system's architecture and goals. Lightweight ATAM is supposed to be used between full ATAMs, namely for intermediate evaluations of consecutive development increments. As it takes as an input most of the results of an ATAM evaluation (e.g. utility trees, quality scenarios), it is not a stand-alone lightweight architecture evaluation method such as ARID, PBAR or TARA.

Evaluation process: the evaluation process was created by eliminating or constraining the scope of ATAM's activities, which is shown in table 11.

Table 11. Comparison of ATAM and Lightweight ATAM (based on table 21.4 [16])

Phase	Inclusion/time	Comments
1. Present the ATAM	Omitted	The participants are supposed to be familiar with ATAM
2. Present business drivers	Short overview, 15 min	The participants are expected to be familiar with the system and its business goals
3. Present architecture	Short overview, 15 min	As above
4. Identify the architectural approaches	Short overview, 15 min	Supposed to be known to the architect, or in many cases identified already in the previous phase
5. Generate the quality attribute tree	Included, variable length, 0.5–1.5 hrs	Here, the existing utility tree and set of quality scenarios should be supplemented with newly emerged information
6. Analyse the architectural approaches	Included, 2–3 hrs	Mapping the highest-priority scenarios onto the architecture cannot be avoided, though it can encompass a limited number of scenarios, e.g. newly identified
7. Brainstorm and prioritise scenarios	Omitted	The scenarios should already have been included in step 5
8. Analyse architectural approaches	Omitted	This has already been done in step 6
9. Present results	Included, 30 min	The "increments" of risks, non-risks, sensitivity points and trade-offs, together with those already identified, should be reviewed and discussed

Evaluation goals, examined properties, architecture description, representation of evaluated requirements, techniques for eliciting requirements, techniques for prioritising requirements, Architecture Evaluation Techniques: generally the same as in ATAM. Most of the results of ATAM are reused by Light-weight ATAM.

Method validation: information not available.

Method limitations, discussion: the method can be used only as a supplement to full ATAMs, which means that it should be performed at intermediate development stages, where a full ATAM has not been planned, but some form of architecture evaluation is needed. The lower effort is achieved at the expense of a less exhaustive and objective evaluation, which reflects the obvious trade-off between the scope and depth of an evaluation.

3.7.5. Scenario-based Approach to Software Architectural Defects Detection (SHADD)

Method highlights: SHADD [135] reverses the approach that underlies the majority of the architecture evaluation methods. The evaluation starts by hypothesising about the possible problems, and then about defects that might be causing those problems. The hypotheses about defects included in the software architecture are validated by the analysis of software architecture.

Evaluation goals: the discovery of architectural defects.

Examined properties: the SHADD method is not aimed at evaluating any specific software quality attribute, or architectural support for that attribute.

Evaluation process: evaluation is performed according to a procedure comprising the following steps (the description presented below demystifies the original method's presentation in [135]):

1. Identification of problems – takes the form of a brainstorming session; each problem is characterised by its cost (loss caused by a given problem occurring) and the probability of an occurrence (initially assumed as 1);
2. Identification of hypothetical defects causing the problems identified in step No 1. The problems are analysed in order to identify defects that might cause them; the more severe problems are analysed first.
3. Verification of the identified defects, and carrying out corrective actions if needed.

   Hypothetical defects are validated as described in the subsection above, in descending order of severity. Corrective actions are undertaken or planned as needed. This, in turn, should cause the evaluation participants to reconsider their assessment of the probability of the problem occurring.
4. Check the termination criterion, finish or restart the analysis.

   The termination criterion is 'average problem severity', given by the formula:

   $$S = \frac{1}{\overline{\overline{P}}} \sum_{p \in P} c_p p_p$$

   where c_p and p_p denote the cost and probability of problem $p \in P$ occurring. If the average problem severity is above a presumed threshold (established individually for every assessment), the evaluation should continue. It was suggested that the least severe problems should be eliminated from the set of considered problems in consecutive iterations. Such a refined set of problems is subject to further evaluation, which restarts from step No 2.

Architecture description: has not been specified; the examples in [135] use a component-connector representation of software architecture.

Representation of evaluated requirements: the method does not address any specific software requirements. However, these requirements are implicitly reflected in the identified problems, which are a consequence of not meeting functional or non-functional requirements. Every problem is characterised by the probability and cost (damage) of its occurrence.

Techniques for eliciting requirements: the project stakeholders brainstorm the problems on the basis of their knowledge of the requirements and of the software architecture.

Techniques for prioritising requirements: as the requirements are not addressed directly by SHADD, the prioritisation concerns the problems and defects identified during the analysis. For problems, the prioritisation criterion is the problem's severity, which is a product of the cost and probability of the problem occurring $S(p) = c_p \times p_p$, where c_p and p_p are, respectively, the cost (loss) and the probability of problem *p* occurring.

The proposed prioritisation technique was indicated as based on a fuzzy relation [164], [160], i.e. the severity of defect *d* is given by the formula:

$$S(d) = \sum_{p \in P} c_p \mu_R(p, d)$$

where: $d \in D$, $p \in P$, *D* and *P* – are sets of all defects and problems, respectively, $\mu_R(p, d)$ – is a membership function whose values are from the interval <0, 1> indicating how much defect *d*, contributes to the existence of problem *p*, i.e. the greater the value, the more important is defect *d* for the existence of problem *p*.

Therefore, the severity of defect *d* can be understood as a share of defect *d* in costs that might be incurred due to the problems that are, to the extent $\mu_R(p, d)$, caused by defect *d*. It was indicated that the values of the membership functions should be defined by the participants of the evaluation.

Architecture Evaluation Techniques: SHADD assumes that problems may be caused by one or more defects. The members of the development team hypothesise about the defects included in the software architecture. The analysis focuses on verifying whether the hypothetical defects really exist within the software architecture. This is achieved by applying validation cases (e.g. tests, inspections, walkthroughs, etc.) on the relevant architectural components.

Evaluation outcomes: although not specified explicitly, the main outcome is a list of discovered architectural defects and the problems caused by them. This can be accompanied by information on corrective activities proposed or undertaken during the analysis.

Method validation: SHADD was illustrated by the example of a Parking Access Control System [135].

Method limitations, discussion: the method relies heavily on the knowledge and experience of the participants, who play the role of "architectural oracle". If the oracle fails, the analysis will deliver no value. It seems that in this way only small systems can be analysed, or in the case of large systems the analysis would have to be limited to the most important problems. In many cases it will turn out to be difficult to estimate the costs of failures, which makes the prioritisation mechanism difficult to use, or even useless.

3.8. ATTRIBUTE-SPECIFIC ARCHITECTURE EVALUATION METHODS

Special-purpose architecture evaluation methods focus the analysis on a chosen quality attribute. This specialisation should make them better fit to the specific needs of evaluating those quality attributes than general-purpose architecture evaluation methods. The list of attribute-specific architecture evaluation methods has been presented in section 3.3.

3.8.1. Architecture-Level Modifiability Analysis (ALMA)

Method highlights: ALMA [19] was specially designed for analysing architectural support for modifiability. The scenarios represent categories of changes that are expected to appear in the future. The impact of those changes is analysed. ALMA provides detailed guidance as to the elicitation of scenarios and analysis of the impact of changes, depending on the goal of the evaluation.

Evaluation goals: chosen at the beginning of the analysis from three options: maintenance cost prediction (to estimate the effort necessary to adapt a system to future changes), risk assessment (identify changes that might turn out difficult to carry out), software architecture comparison.

Examined properties: modifiability.

Evaluation process: The assessment process comprises the following phases:

1. Establish the goal of the evaluation – choose one of the three predefined goals: risk analysis, modification cost, architecture comparison.
2. Describe the architecture.
3. Elicit change scenarios
4. Evaluate change scenarios
5. Interpret the results – the results of the change scenario evaluation have to be interpreted in order to draw conclusions regarding the goal assumed at the beginning of the evaluation.

Architecture description: ALMA analysis is based on component-connector models of software architecture.

Representation of evaluated requirements: modifiability requirements are represented as a set of change scenarios, which should represent changes that are expected to happen in the foreseeable future. The change scenario structure has not been precisely defined. They have a textual form [19], for example “A new report based on accounting data”, ”Replacement or upgrade of an operating system”.

Techniques for eliciting requirements: change scenarios are extracted during interviews with the architecture stakeholders. There are no general rules on selecting the interviewed stakeholders, as the right choice depends both on the system's specifics and the goal of the analysis. The hints presented in [19] could be summarised as: the stakeholders chosen for an interview should be familiar with the sources of changes that are important for a given analysis (or even just be the initiators of those changes) as well as with the types of changes that may come from these sources. For example, if a system is developed for the mass market, a certain knowledge about the customers' needs and domain trends is necessary to effectively foresee changes. This knowledge can be obtained from the members of the marketing department rather than from the software developers or architects.

The two general elicitation techniques are top-down and bottom-up. Top-down elicitation starts from a chosen change category, and then stakeholders are prompted to deliver change scenarios falling into that category. In the bottom-up technique, the interviewed stakeholders come up freely with the change scenarios, which are subsequently categorised. The emergence of ever newer scenarios may also result in modifying the changes classification scheme. In both cases, the elicitation process is guided by the evaluator, which should aim at exploring the changes most relevant for the analysis.

Too many scenarios can make the evaluation too costly or time-consuming. ALMA addresses this issue by proposing two techniques for minimising the number of change scenarios that are subject to the analysis:

- clustering scenarios into equivalence classes – sets of similar (equivalent) scenarios represented as a single one, representative for the entire set. In this way, superfluous scenarios will not be analysed and the entire number of analysed scenarios will shrink;
- categorising changes – defining categories of changes of primary interest and using them to facilitate and focus the development of new change scenarios. The scenario categories may stem from the decomposition of a system's domain, e.g. for an accounting system the categories can reflect the variety of accounting areas, e.g. fixed assets, accounts receivable/payable, etc. The emergence of newer scenarios can enforce the modification of existing change categories. This technique is integrated with scenario elicitation techniques.

ALMA integrates the above activities with the elicitation techniques into a single change scenario elicitation approach. However, the entire approach has not been formalised into a precisely defined recipe, and the use of the components described above depends on the expert knowledge of the evaluators. The stop criteria for the elicitation process are that all the defined change categories have already been considered, and further scenario elicitation does not change the classification of changes.

Techniques for prioritising requirements: prioritisation criteria depend on the evaluation goals. In the case of a maintenance cost prediction, this should be the probability of the scenario occurring; in the case of a risk assessment goal, it is the level of modifiability risks (the most complex scenarios usually produce the highest risk); in the case of architectural comparison, the two previous criteria can be used, or scenarios that emphasise differences between the architectures. Prioritisation is used in step 3 as an integral part of scenarios elicitation in order to keep the number of evaluated scenarios at an analysable level.

Architecture evaluation techniques: the core of the ALMA architecture assessment is the architecture-level impact analysis, which should be performed for all the changes expressed by the change scenarios. Its aim is to identify the scope of changes to software architecture that will be needed in order to accomplish the change scenarios. However, the method offers impact analysis techniques tailored to the relevant goals of the analysis.

If ALMA is used for risk assessment, the assessment techniques are aimed at estimating and comparing the complexity of changes required to fulfil the scenarios. The proposed technique comprises four steps:

1. Determining the initiator of change;
2. Determining the impact level of every scenario. ALMA defines four impact levels for change scenarios: (1) no impact, i.e. the change scenario has already been provisioned for; (2) a single component is affected only; (3) several components are affected, and (4) the software architecture is affected by the scenarios, i.e. structural changes are necessary to implement scenario.
3. Establishing the parties that will have to be involved in change implementation. This concerns particularly owners of the changed components. Note that change implementation may require modifying external components (e.g. external systems interfaced with the modified software) and that software may include components owned by third-parties (in terms of intellectual rights).
4. Identifying the level of version conflict that could result from implementing a certain change. Three levels have been defined: (1) the scenario does not introduce different versions of the same component, (2) the scenario introduces different versions of the component, (3) the scenario's implementation causes version conflicts between the components.

Although it has not been directly indicated in [19], the proposed risk assessment approach is similar to the well-known equation: Level of risk = Probability of occurrence × Level of impact, in which 'level of impact' should be understood as the scenario's complexity. The scenario's complexity, in turn, is a product of the number of component owners involved in making the change and the level of version conflict. The probability of occurrence has to be discussed with the stakeholders. Change scenarios that pose a substantial modifiability risk, may indicate the need for architectural improvements.

If ALMA is performed for maintenance cost prediction, the impact analysis should indicate the existing components affected by a given scenario and new components that will have to be developed. The estimated maintenance effort *E* is given by the formula:

$$E = \frac{\sum_{s \in S} \left(P_M \sum_{c \in C(s)} w_s c_s + P_I \sum_{c \in N(s)} w_s s_s \right)}{N_{CS}} N_C$$

where:

P_M, P_{I} – productivity of modification (M) and implementation (I). It should be expressed e.g. in KLOC/Man-hour;
S – represents a set of scenarios;
C(*s*) – denotes a set of components that needs to be changed to support scenario *s*;
N(*s*) – denotes a set of components that has to be implemented in order to support scenario *s*;
N_{CS} – number of change scenarios;
N_C – number of changes expected to happen during an assumed period;
s_c – size of component changes/implementation expressed in comparable units;
w_c – weight of scenario *c*, i.e. the ratio of estimated number of changes represented by scenario *c* to the total number of changes represented by all the scenarios (note that this may be different than N_C). Therefore, w_c is a share of changes represented by scenario *c* in the total number of changes represented by all the scenarios.

The formula estimates the total effort needed to make all the foreseen changes. Note that each scenario may represent a number of foreseen changes of the same category (e.g. introducing a new data report). These numbers, which are subsequently included in the weights w_c, have to be estimated by the stakeholders.

Evaluation outcomes: depending on the evaluation's goal they could be: list of modifiability risks, estimated cost of foreseeable changes, ranking of compared architectures with regard to the foreseeable changes**.**

Method validation: two detailed, practical case studies presented in [97].

Method limitations, discussion: ALMA provides detailed guidance for analysing modifiability at an architectural level, which is a major advantage over the similar but more general and sketchy SAAM. The limitation of the method is that it does require some pre-development (pre-architecting) of changes when analysing the impact of changes. This may be unachievable in many practical cases, as architecting is usually even more challenging and time consuming than evaluation. Therefore, ALMA can be applied when changes have a rather limited and foreseeable impact.

3.8.2. Scenario-based Architecture Level Usability Analysis (SALUTA)

Method highlights: SALUTA [58], [59] facilitates verification of whether software architecture meets usability requirements, i.e. whether it adequately supports software usability. Usability is an intrinsically complex property that is defined by four sub-attributes (see section "examined properties" below). Usability scenarios represent cases of software use and the level of support for each of the usability attributes, which should be provided by the architecture. The most important part of SALUTA is the usability framework, connecting patterns (tactics) included in software architecture with usability sub-attributes affected by these patterns (for complete treatment, see [56]). These relationships provide a basis for architecture evaluation.

Evaluation goals: an assessment of architectural support for software usability or a comparison of software architectures in terms of their support for software usability.

Examined properties: *usability*. SALUTA assumes that usability combines four detailed software "usability attributes":

- Learnability – how easily users can learn to use the software's features;
- Efficiency of use – how quickly tasks can be performed by the users, i.e. what the users' performance will be when using the software (how many operations they can perform per unit of time);
- Reliability in use, error handling and recovery – how many errors users make while using the software, and how easy it is to recover from or correct them;
- User satisfaction – reflects the users' subjective opinions.

The above components of software usability have been identified on the basis of the definitions presented in research papers and standards (compare [56] for a comprehensive account).

Evaluation process: The evaluation process comprises four steps:

1. Create usage profile – comprises elicitation of usability scenarios and prioritising them in order to develop a usage profile representative for the usage of a given software and suitable for the purpose of the analysis;
2. Describe provided usability – identification of usability properties and patterns included in the software architecture.
3. Evaluate scenarios – assessing the level of architectural support for the elicited usability scenarios.
4. Interpret the results – the outcomes regarding the goals of the analysis are drawnup on the basis of the evaluated usability scenarios.

Architecture description: SALUTA requires that the usability patterns and usability properties (see below) included in a software architecture be identified. The method does not assume any concrete form of architectural description as the source of information about usability patterns and properties. The patterns and properties can be identified by analysing the software architecture, functional design documentation, interviewing the software architects, etc.

Representation of evaluated requirements: usability requirements are captured as usability scenarios, which represent cases of users' interactions with the analysed software system. Usability scenarios are modelled as a three-tuple comprising:

- role (the group of users performing the same duties);
- task (the activity made by users with the software; it reflects the software's functionality);
- context of use (the situation in which the software is used, e.g. use on a desktop computer, use on a mobile device, regular use or training use).

Techniques for eliciting requirements: the elicitation of usability scenarios should start by identifying users' roles, tasks and contexts of use. Then only valid tuples of the Cartesian product of the sets of users' roles, tasks and contexts should be included in the set of considered usability scenarios. Numbers of scale 1–4 should be assigned to the usability attributes for every elicited scenario. They should reflect the relative importance of the usability attributes for meeting usability requirements connected with a given usability scenario. The assignment is a matter of expert judgement and should be done together with the users.

Let us consider the following usability scenario: in an accounting system, the chief accountant performs the annual closing of accounts once a year. This consists of a number of steps that should receive final approval after verifying whether the profit-loss account and balance sheet are correct. The usability attributes for the scenario could be given the following values: satisfaction – 1 (low importance), learnability – 3 (significant importance, it should be easy to learn how to perform the annual closing), efficiency – 1 (low importance, the operation is done only once a year), reliability – 4 (the correctness of the annual closing is a very important feature).

Techniques for prioritising requirements: In order to make an evaluation feasible, it has been suggested to focus the architectural analysis only on scenarios that are representative for a system's use. No strict prioritisation technique has been indicated in the method's description [58]. The scenarios' priorities may depend on the goal of the analysis. The importance of the scenarios may reflect such things as the importance of certain scenarios to users, the frequency of performing certain tasks in certain modes, the evaluated differences between architectures. The analysts usually chose only the most important scenarios, i.e. those with the highest priority, for further analysis. The set of such carefully chosen scenarios has been referred to as the *usage profile*.

Techniques for Evaluating Architecture: the architecture analysis is made scenario-by-scenario. For every evaluated usability scenario, the sets of usability patterns and properties affecting a given usability scenario are identified. Identified usability patterns and properties can be related to the usability attributes by the use of the usability framework (figure 10) [58]. This captures the relationships between:

- usability attributes and the usability properties affecting those attributes,
- usability properties and usability patterns (they are in fact architectural tactics for usability, compare [16]) that could have been applied in order to implement certain usability properties.

The usability framework is the core of the SALUTA method, as it summarises the knowledge necessary for a usability evaluation of the architectural support. Let us note that the terms "efficiency" and "reliability" are used here as attributes of the interaction with the analysed system, and should not be understood as defined in ISO 9126.

Finally, the level of support for every usability attribute is subject to expert judgement based on:

1. Usability patterns/properties affecting a certain usability scenario;
2. Usability attribute priorities (importance levels) established while eliciting scenarios;
3. Knowledge of the architecture analyst.

The result of the scenario's evaluation may be expressed as a percentage of fulfilment, with a two-level (supported/not supported) or five-level (–, – –, –/+, +, ++) scale, depending on the range and level of detail of available information.

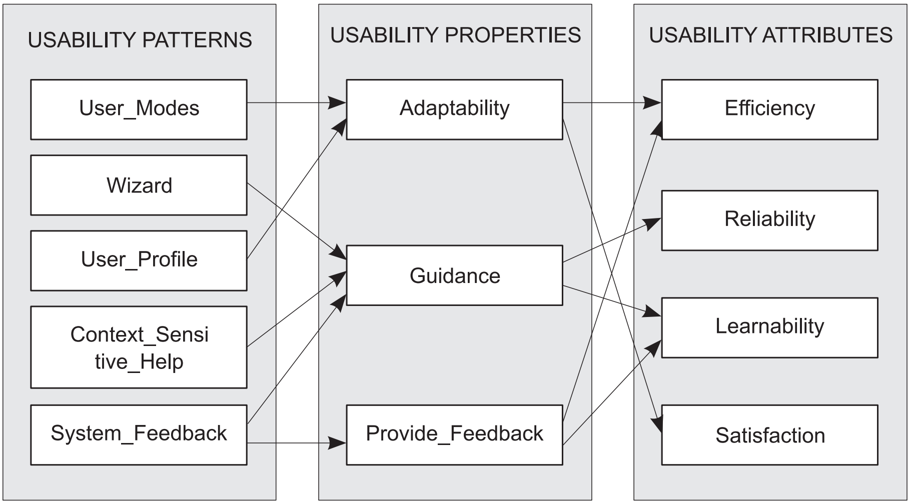

Figure 10. Fragment of the usability framework (prepared on the basis of [58])

Returning to the example of an accounting system, let us assume that the software architecture contains two usability patterns addressing the usability scenario, namely a wizard that guides through the closing process, and context sensitive help. Both these usability patterns provide guidance, increasing learnability and reliability, and have been identified while eliciting usability scenarios as attributes of the highest priority. In order to assess the level of architectural support for this usability scenario, a broader knowledge on the closing process is needed. For every chief accountant, it is obvious that the system should allow trial annual closings to be made, and to be undone. This important usability pattern has not been included in the software architecture, which is a major weakness. So the support for a given scenario can be assessed as partial (e.g. 50% of fulfilment or –/+).

Evaluation outcomes: have not been precisely specified or structured. In general, the outcomes depend on the goal of the analysis (usability support assessment or comparison of architectures). The outcomes of the evaluation have not been prescribed or limited by the method's authors. The evaluation may result in: information about the level of architectural support for the software usability, a list of components that require improvement, a list of scenarios usability patterns that should be included in order to ensure the right level of usability support.

Method validation: three real-world case studies [57].

Method limitations, discussion: SALUTA's usability framework is the main contribution of the method. It summarises knowledge on the architectural ramifications of software usability. It is both useful for architects, designers and evaluators. The usability framework later evolved into "usability tactics", compare [16]. As the brief example presented in this section reveals, a successful evaluation depends on the evaluator's knowledge and the appropriate participation of users, as the assessment technique defines the framework rather than a precise evaluation recipe.

3.8.3. Performance Assessment of Software Architecture (PASA)

Method highlights: PASA [155] is aimed at verifying that software architecture sufficiently supports performance. The architectural assessment is done by identifying patterns and anti-patterns that may impede performance or by applying model-based performance analysis techniques.

Evaluation goals: the goals of the analysis have not been explicitly defined, nor are they supposed to be identified or chosen during the analysis. The method identifies "potential areas of risk with respect to performance and other quality objectives" [155].

Examined properties: performance.

Evaluation process. The evaluation process of PASA comprises nine, almost self-explanatory, steps:

1. Process Overview – PASA is presented to the participants of the evaluation.
2. Architecture Overview – the architecture is presented to the participants.

3. Identification of Critical Use Cases
4. Selection of Key Performance Scenarios
5. Identification of Performance Objectives
6. Architecture Clarification and Discussion – the participants discuss the architecture in more detail in order to better understand its components determining the system's performance.
7. Architectural Analysis – is aimed at assessing whether architecture sufficiently supports the performance requirements and the discovery of architectural flaws that may impede the system's performance.
8. Identification of Alternatives – developing corrections to software architecture in order to correct flaws detected during the Architectural Analysis. The authors propose some kinds of corrective actions: architecture refactoring aimed at the removal of detected anti-patterns, the modification of instantiation of architectural patterns, in order to improve performance and/or modify interaction between the components.
9. Presentation of Results (and reporting).
10. Economic analysis – its purpose is to produce evidence of the value of every PASA-based evaluation to its participants. This evidence may comprise a summary of the costs and effort presented against the list of benefits of a PASA-based evaluation. The value of the benefits of the evaluation can be expressed in terms of the time and money that would have been required to correct discovered architectural flaws if they had not been found during the architectural evaluation.

Architecture description: PASA does not assume any concrete form of architecture documentation. It also accepts that architectural description may turn out to be incomplete and informal, so it may be necessary to elicit it directly from the developers.

Representation of evaluated requirements: quantitative performance requirements should be identified for every *key performance scenario*. Key performance scenarios are scenarios of interactions with a system, which might have a prevailing impact on the system's performance. These requirements may concern response time, throughput and constraints on resource utilisation.

Techniques for eliciting requirements: The use cases, which determine system performance, have been referred to as "critical". These critical use cases reflect the responsiveness of a system to the actions of the actors (users), or concern the use cases that are particularly vulnerable to a system's performance. As each critical use case may contain a number of alternative interaction scenarios, it is necessary to choose for the analysis those that have a prevailing impact on performance, for example alternatives that are rarely executed can be ignored. Performance requirements connected with the scenarios should be expressed in quantitative terms.

Techniques for prioritising requirements: selecting critical use cases and their most important scenarios, which can be done while eliciting the requirements.

Architecture Evaluation Techniques: the impact of the software architecture on performance can be assessed using one of the three proposed techniques:

1. Identifying architectural patterns included in the system's design and investigating their influence on the system's performance, this is a similar technique as in ATAM and PBAR (compare section 2.2.5).
2. Identifying performance anti-patterns, i.e. typical architectural structures that hinder performance [138], [43], [44] that might have been included in the system's design or implementation.
3. Applying quantitative performance analysis techniques – an extensive survey of such techniques can be found in [12].

Evaluation outcomes: detected performance issues (detected patterns or anti-patterns that may impede performance, bottlenecks, quality requirements not met) proposed architectural corrections.

Method validation: lack of precise information, in [155] it was said that PASA is a result of multiple assessments in several software domains over five years.

Method limitations, discussion: PASA requires from one intensive to several less-intensive weeks [155]. Evaluation staffing has not been precisely specified. The method is rather sketchy with regard to the evaluation process and analysis techniques.

3.8.4. Continuous Performance Assessment of Software Architecture (CPASA)

Method highlights: CPASA is a lightweight performance evaluation method that should be applicable in an agile development. The evaluation process of the PASA method (section 3.8.3) makes it most suitable for elaborative development processes such as waterfall or the Rational Unified Process [95]. However, the application of PASA in agile development processes is problematic. CPASA was developed in order to address the above issue. The method extends iterations of agile development processes with the evaluation of the architecture's impact on the system's performance. In this way, software architecture support for the performance requirements is permanently monitored.

Evaluation goals: the continuous assessment of the performance aspects of the system being developed by the use of an agile methodology.

Examined properties: performance.

Evaluation process: CPASA integrates a performance evaluation of software architecture into an agile development process. Performance evaluation tests are performed between every two consecutive iterations of the agile development process. If a performance problem is detected during the evaluation, then design refactoring should be undertaken in order to resolve the problem. The evaluation itself comprises two steps:

1. Construction of a performance model;
2. Solving the performance model.

Architecture description: mainly as UML component diagrams.

Representation of evaluated requirements: as in PASA (section 3.8.3).

Techniques for eliciting requirements: as in PASA (section 3.8.3).

Techniques for prioritising requirements: as in PASA (section 3.8.3).

Architecture Evaluation Techniques: the method promotes the extensive re-use of performance models built during earlier development cycles. In [119], performance analysis techniques and models such as Markov Chains, Queuing networks, Petri nets (incl. stochastic Petri Nets) and Process algebra (TIPP) have been mentioned as possible choices. The resolution of performance models built at preceding agile iterations, altered in order to reflect the changed conditions or automatically generated from UML models (UML-JMT tools based on the Extended Queuing Network model [1] were developed for that purpose; the SPEED tool [137] can also be used) is promoted as the main means of monitoring whether architecture keeps supporting the performance requirements. This is supposed to fit well into the concept of test-driven development.

Evaluation outcomes: identified performance issues.

Method validation: no information.

Method limitations, discussion: the method's success relies heavily on the re-use of performance models built earlier in the course of a project, as well as on tool support. The general challenge is the validity of parameters fed into those models, namely execution times or their distributions. If they can be measured directly on the running software, the application of CPASA makes little sense. If not, the accuracy of such estimations is generally questionable. This may undermine the trustworthiness of the outcomes of a performance analysis based on that model.

3.8.5. Architecture Evolvability Analysis (AREA)

Method summary: AREA [26], [27], [30] was designed to analyse whether software architecture adequately supports evolution. AREA assumes that evolution is a controlled and deliberate process, which does not hold true in every case; consider, for example, software evolution in emergent organisations in [22]). Evolvability is a multidimensional attribute that is defined by a set of sub-characteristics (e.g. analysability, changeability, extensibility). The main challenge addressed by the AREA method is that evolvability sub-characteristics usually have a different importance to different stakeholders. So, in order to evaluate software evolvability, it is necessary to start by defining what the evolvability of a given piece of software means. This leads to inevitable trade-offs that have to be achieved between the expectations of the stakeholders. AREA enables both qualitative and quantitative evaluations. In a qualitative evaluation, the consensus between the stakeholders is achieved through an informal discussion of the importance of requirements and of their influence on the evolvability of candidate architectural solutions. In a quantitative evaluation, the process of making these trade-offs is achieved through the use of Analytical Hierarchical Process (AHP) [128], [129] for requirements prioritisation and a comparison of architectural solutions.

Evaluation goals: identification of the weak parts of a system's architecture with regard to evolvability, defining evolvability requirements, choosing an architecture's modifications that best suit software evolvability.

Examined properties: evolvability. Evolvability has been defined as the ability of a software system to accommodate changing requirements. Let us note that maintainability [117], [74] is a similar notion, with a broader meaning than evolvability, as it defines the ability of a software to adapt to changes of any kind (e.g. error removal, functional extensions, hardware changes, etc.)

Evolvability is an aggregate of sub-characteristics that may vary among different domains. The main enablers of evolvability, according to [26], are: analysability, architectural integrity, changeability, extensibility, portability, testability, and, domain-specific attributes (e.g. timing properties in the case of hard real-time systems).

Evaluation process: AREA's evaluation process comprises four steps:

1. Elicit architectural concerns;
2. Analyse implications of change stimuli – identify the evolvability requirements;
3. Propose architectural solutions;
4. Assess the proposed architectural solutions and present the evaluation's outcomes.

Architecture description: AREA does not promote any concrete form of architectural description to be used. Instead, it has been indicated that analysis focuses on architectural constructs (i.e. fragments of architecture, such as components and systems) related to the identified problems, as well as on architectural solutions that should be developed during the analysis in order to respond to the evolvability requirements. *Architectural solutions* are equivalent to architectural decisions (section 2.2). They are defined by the following components:

- problem description (the weaknesses of the original architecture),
- new requirements the architecture should support,
- improvement solution resolving the problem
- rationale and architectural consequences of the proposed solution,
- estimated workload needed to carry out the solution.

The above information is gathered during the evaluation.

Representation of evaluated requirements: the evolvability requirements are captured as sentences, possibly accompanied by a list of possible refinement activities related to these requirements (compare also architectural tactics, which are a similar concept – section 2.2.5). It should be possible to assign the requirements to the appropriate evolvability sub-characteristics.

Techniques for eliciting requirements: the evolvability requirements should reflect change stimuli and architectural concerns that have to be identified prior to establishing the requirements (steps 1 and 2). Change stimulus is an event or a condition that may require changes to be made to software architecture. Change stimuli usually reflect the stakeholders' concerns relating to evolvability. Requirements are identified by analysing how change stimuli influence software architecture. Let us consider an example: the company has developed a traveling salesmen application for a certain PDA model. The company's management expects this system to become a flagship product sold to multiple clients, using various mobile devices and mobile operating systems. The evolvability requirements would certainly include the following:

R.1. Portability between various hardware platforms

- investigate the available hardware platforms;
- investigate the support for portability provided by the operating systems;
- introduce a hardware abstraction layer in the application.

R.2. Portability between operating systems

- investigate the operating systems used on mobile devices;
- introduce an operating system abstraction layer, implement the functionality as a layer above the abstraction layer.

Techniques for prioritising requirements: requirements prioritisation is made by a discussion between the evaluation participants (qualitative evaluation), or through the use of AHP (quantitative evaluation). In the case of a qualitative evaluation, it is recommended to agree on the prioritisation criteria first, then to prioritise the requirements.

Architecture Evaluation Techniques: in response to the evolvability requirements, architectural solutions, including architectural modifications or refactorings, should be proposed. Then their influence on evolvability sub-characteristics should be assessed, which is done either by discussing the solutions with the participants (qualitative evaluation), or through applying AHP with pair-wise comparisons of architectural solutions done by the individual participants (quantitative evaluation). In both cases, the participants should present their expert judgement on the architectural solutions. The assessed influence of an architectural solution on evolvability sub-characteristics, together with the requirements priorities, should guide the selection of the most suitable solutions.

Evaluation outcomes: prioritised architectural requirements; stakeholders' evolvability concerns; candidate architectural solutions; and the impact of the architectural solutions on evolvability. [26].

Method validation: at least one industrial case study [28].

Method limitations, discussion: AREA, as presented in [26], is in fact a method supporting software reengineering by facilitating the choice from among the alternative architecture modifications. Nevertheless, it should also be applicable for software architecture during the design stage. AREA requires moderate effort as it is typically performed during three half-day workshops, though with quite a broad participation of architecture stakeholders (architects, product manager, key software developers, and evaluator). The identification of stakeholders' concerns and combining them with change stimuli is a tricky part of the methods, which relies entirely on the expertise of the participants.

The quantitative version requires that participants accept that their opinions are subject to an automated calculation procedure, which derives the basis for formulating the outcomes of the evaluation. Co-operation between participants is a necessary precondition for a successful qualitative evaluation. In both versions of the method, the participants build their opinion on the influence of architectural solutions on the evolvability sub-characteristics using their own expertise.

3.8.6. Software Architecture Reliability Analysis Approach (SARAH)

Method highlights: SARAH [148] was designed to be used to analyse reliability as perceived by the users of consumer electronics devices. The reliability requirements are identified as failure scenarios associating components with sets of their possible failures. Failures are captured according to a failure domain model providing a catalogue of possible failures, errors and faults. FTA is used to model interactions between the failures. This enables critical components causing or facilitating the majority of failures to be identified, and reliability tactics to improve their reliability to be applied.

Evaluation goals: the evaluation is performed to obtain information necessary for system components to be modified in order to improve a system's ability to avoid or tolerate faults.

Examined properties: reliability, which is defined as "the ability to cope with failures" [148].

Evaluation process:

Phase 1. Definition

1.1. Describe the architecture
1.2. Develop failure scenarios (elicit reliability requirements)
1.3. Derive fault tree set (explore dependencies between failure scenarios)
1.4. Define severity values in the fault tree set (prioritise failure scenarios)

Phase 2. Analysis

2.1. Perform architectural level analysis
2.2. Perform architectural element level analysis
2.3. Report (summarising the outcomes and the information gathered during the analysis – domain model, failures scenarios, fault tree set, etc.)

Additionally, SARAH includes an “Adjustment” phase during which reliability tactics should be applied in order to correct the detected architectural flaws.

Architecture description: not prescribed, the example in [148] and [140] uses module or component views; generally coarse-grained level representations are preferable for a SARAH analysis as the method analyses all the identified architectural elements (components).

Representation of evaluated requirements: failure scenarios are “potential failures that could occur due to internal or external causes within a given context” [148]. Each failure scenario should be defined using the following attributes:

- Failure Identifier – a unique identifier for a failure scenario.
- Architectural element identifier – an acronym identifying the architectural element (component) affected by a given failure scenario;
- Fault, Error, Failure – represented according to the failure domain model (see below).

This above representation is compliant with the Failure Mode Effect Analysis.

Techniques for eliciting requirements: reliability requirements are derived in three steps:

1. Define the failure domain model relevant for the domain of the analysed system. Failure is defined by three features:
    - Fault – characterises the cause of an error, for example, a bug in a software or a hardware failure;
    - Error – characterises how an element fails, the state of the system causing a failure, for example data corruption, deadlock, livelock, etc.;
    - Failure – occurs when a service rendered by a system becomes in any way distorted. This attribute characterises a system malfunction, especially those visible to the users. The causal chain is: fault -> error -> failure.

The failure domain model provides space for available failure scenario descriptions by defining a vocabulary of attributes of faults, errors and failures, and possible values for those attributes. The domain model should be developed by systematically analysing the domain of the system being evaluated. A domain model for digital TV software presented in [148] is set out below (possible values of each of the attributes are given in brackets):

- *Faults* are characterised with the following attributes: “source” (internal/external/other element), “dimension” (hardware/software) and “persistence” (persistent/transient)
- *Errors* are characterised by “type” (data corruption/deadlock/wrong value/wrong execution path, etc.), “detectability” (detectable/undetectable) and “reversibility” (reversible/irreversible)
- *Failures* are characterised by “type” (timing/behaviour/presentation quality/wrong value) and “target” (user/other element).

2. Derive failure scenarios.
   Failure scenarios are identified for each of the architectural elements by the evaluation participants. SARAH does not provide any additional guidance in this respect.

Techniques for prioritising requirements: SARAH requirements prioritisation is performed during step No 1.3 “Derive fault tree set from failure scenarios” and No 1.4 “Define severity values in fault tree set” (the former is needed to perform the latter).

The failure of one component can cause the failure of another component and so forth. Fault trees should capture these cause-effect relationships between the failure scenarios. Failure scenarios affecting users directly should be identified in the entire set of scenarios. The roots of fault trees of interest should represent failures observable by users. The leaves of the fault trees define the root causes of failures.

Severity values are first assigned to the leaves of the fault trees, and then they are automatically propagated up the trees to their roots, using an algorithm presented in [148]. This way, the most severe failure scenarios observable by users are identified.

SARAH ranks failure scenarios severity using the following scale: 1 – very low (failure hardly noticeable by the users), 2 – low (failure noticeable but not annoying), 3 – moderate (annoying), 4 – high (loss of functionality), 5 – very high (system useless, basic functions unavailable, stops responding). It is also possible to use a more elaborate model of users’ perception of failures, thought this is treated as an external input to the method.

Architecture Evaluation Techniques: The analysis of architecture comprises the following two steps (compare the “evaluation process” subsection below):

1. Analysis at an architectural level – aims to identify components that have a prevailing impact on a system’s reliability. This is achieved by prioritising components according to the percentage of failures connected with a given component or according to the weighted percentage of failures, i.e. the share of severity of failures connected with a given element in the sum of all the failure severities.
2. Analysis at an architectural element-level – the most sensitive architectural elements identified in the previous steps are analysed to establish how they fail. This can be achieved by analysing the failure scenarios connected with a given architectural element in order to group them according to the failure domain model. This way, the most common faults, errors and failures can be identified. This, in turn, is necessary to adjust software architecture by applying suitable architectural tactics [16] chosen in order to increase the system’s reliability.

Evaluation outcomes: list of unreliable and sensitive components, types of failures that might frequently happen.

Method validation: single detailed practical example in [148] and [140].

Method limitations, discussion: although reliability can be analysed with ATAM, SARAH provides a much more detailed guidance on carrying out the reliability evaluation. The method assumes that the coupling between the components is low, which may be typical in the case of embedded software for home electronic devices. The components should also be coarse-grained to make a SARAH-based evaluation feasible. This allows for a manageable number of failure scenarios to be captured and for the causal logic of failures to be captured with fault trees. It is hard to imagine this method being applied for business software. The calculation of severity values on the basis of the severities of the root causes seems to be problematic: such a calculation certainly makes sense in the case of propagating fault probabilities up the fault tree, though severity levels can be assigned directly to the failures observable by users on the basis of knowledge of the system. In such a case, the propagation of severity values would not be needed. Another problem is how failures not directly affecting users can be assessed in terms of the disturbance they cause to the users. SARAH does not clarify that issue.

3.8.7. Aspectual Software Architecture Analysis Method (ASAAM)

Method description: ASAAM [149] is an extension to SAAM, which enhances SAAM by identifying aspects and the tangled components related to them. This indicates components that may require refactoring because overly complex components may seriously impede a software's modifiability.

Evaluation goals: verification of whether the architecture sufficiently supports the separation of concerns. This is achieved by identifying overly complex (tangled) components.

Examined properties: coupling and cohesion of software modules that may influence attributes such as modifiability, maintainability and evolvability.

Evaluation process: ASAAM extends the SAAM evaluation process in the following way:

- The step "Individual scenario evaluation and aspect identification" extends the SAAM step "Perform scenario evaluations" with techniques for the identification of crosscutting concerns represented by quality scenarios. Heuristic rules defined in [149] classify scenarios into groups of direct scenarios, indirect scenarios, aspectual scenarios and architectural aspects.
- The step "Scenario interaction assessment and component classification" extends the SAAM step "Reveal scenario interaction" with techniques that classify components into cohesive component, tangled components, composite components, or ill-defined component groups.
- ASAAM's evaluation process ends with the "Refactoring of architecture" step, during which the architecture should be modified in order to ensure the proper

separation of concerns. Techniques such as applying design patterns or aspect-oriented techniques are recommended for that purpose.

Architecture description: component-connector models, as in SAAM (section 3.6.1).

Representation of evaluated requirements: SAAM-style quality scenarios.

Techniques for eliciting requirements: as in SAAM.

Techniques for prioritising requirements: n/a.

Architecture Evaluation Techniques: ASAAM introduces a set of heuristic rules for classifying scenarios and components (see below). For more detail, see [149].

Evaluation outcomes:

- quality scenarios classified into groups of direct scenarios (supported by the existing architecture), indirect scenarios (requiring changes to one or more components in order to be supported), and aspectual scenarios (support for a given scenario is distributed among the components and cannot be refactored into one).
- software components classified into the following groups: cohesive (well-defined, supports semantically close scenarios), composite (consists of a number of cohesive components), tangled (supports the aspectual scenario directly or indirectly), ill-defined (supports semantically distinct scenarios and cannot be decomposed, or does not support the aspectual scenario);
- architectural aspects corresponding to the identified aspectual scenarios are also identified.

Method validation: a single case presented in [149].

Method limitations, discussion: the coupling and cohesion of software components has a strategic value as it generally determines how easy it will be to maintain and evolve software. The notion of semantically close or semantically distant is rather a vague one; therefore, the proper classification of components relies on a subjective assessment of the evaluator. The general limitation of ASAAM is that a set of quality scenarios enabling an effective analysis of software modularity would have to cover the entire software functionality. This is rarely achievable. In SAAM, scenarios mostly capture expected software modifications, but not the functional requirements already met.

The refactoring of architecture in order to eliminate all the tangled and ill-defined components will not always be possible, so a compromise will have to be achieved, which is not addressed by ASAAM.

3.8.8. Cost-Benefit Analysis Method (CBAM)

Method highlights: CBAM [82], [15] is actually a supplement to ATAM. It takes as an input a set of quality scenarios developed during ATAM. The outcome is information on the Return On Investment (ROI) for a number of considered architectural strategies. During the evaluation, those quality scenarios that are less important for achieving business goals are gradually eliminated as the analysis proceeds.

Evaluation goals: comparison of architectures (built-in) by the Return on Investment delivered by each of "architectural strategies".

Examined properties: as in ATAM, i.e. possibly all the important non-functional quality attributes;

Evaluation process: The method comprises 9 steps described in detail in [15].

Step 1. Scenarios gathered during the ATAM evaluation, and some added by the CBAM participants, are prioritised based on their ability to satisfy business goals. Out of the entire set, only one third should be left for the refinement in step 2.

Step 2. The levels of response measures are elicited by indicating worst-case, current, desired, best-case levels for each of the scenarios selected in step 1. These levels have to be expressed in quantitative terms, e.g. system response time: 100 ms, 50 ms, 20 ms, 10 ms for each of the mentioned levels respectively.

Step 3. The scenarios are prioritised by the stakeholders with a cumulative voting technique or just by agreeing the priorities during a discussion. Numbers of votes obtained by every scenario should be transformed into weights (range 0.0–1.0). The 1.0 weight should be assigned to the scenario that obtained the biggest number of votes. The stakeholders should base voting on the response levels of each scenario, which is desired by them. The votes given for each scenario should be totalled and the top 50% of the scenarios should be selected for step 4. The quality attributes that have turned out to be most important to the stakeholders should be listed.

Step 4. Utility is assigned for each of the response levels by consensus of the stakeholders. Worst-case level is assigned a utility of 0, "best-case" of 100; the utility values for "current" and "desired" levels are decided by the stakeholders.

Step 5. Architectural strategies should be developed for the scenarios. The expected attribute response levels for each scenario should be determined. A single strategy can influence multiple scenarios; hence the expected response levels should be estimated for all the scenarios affected by an architectural strategy.

Step 6. The expected utility value of an architectural strategy is estimated using interpolation.

Step 7. The total benefit of an architectural strategy should be calculated as a normalised sum of the differences between the expected utility of a strategy and the current utility overall the scenarios and quality attributes. The normalisation factors should be the numbers of votes that every quality scenario obtained in step 3.

Step 8. Architectural strategies are selected on the basis of ROI regarding cost constraints. ROI is calculated as a ratio of "Total benefit" to "Cost of implementing a strategy". ROI is a ranking factor for all the strategies. The strategies for which implementation exceeds cost constraints are rejected.

Step 9. Confirm the results with intuition. The results obtained after step 8 should be verified as to whether they are in line with the business goals and intuition. If not, reiterating steps 1-8 should be considered.

Architecture description: as in ATAM.

Representation of evaluated requirements: quality scenarios.

Techniques for eliciting requirements: CBAM is a supplement to ATAM, hence the scenarios generated during ATAM provide a basis for the analysis; more scenarios are added by additional brainstorming.

Techniques for prioritising requirements: cumulative voting technique or agreeing the priorities during a discussion.

Architecture Evaluation Techniques: the method relies on expert judgement, but still enforces an assessment of quality attributes in quantitative terms.

Evaluation outcomes: Return On Investment (ROI) for each of the considered architectural strategies

Method validation: several detailed examples presented, e.g. in [15], [82].

Method limitations, discussion: CBAM is based on expressing stakeholders' expectations and analysing system properties in quantitative terms. This is both a virtue and a vice. The quantitative results are unambiguous and supposedly more convincing. However, most numbers gathered during CBAM do not result from a measurement or model-based analysis, but express the intrinsically subjective opinion of the stakeholders. The authors observed that such a quantitative assessment procedure may sometimes lead to results that contradict the business goals, and proposed that the outcomes be verified intuitively (step 9).

3.9. APPLICATION DOMAIN-SPECIFIC ARCHITECTURE EVALUATION METHODS

3.9.1. Holistic Product Line Architecture Assessment (HoPLAA)

Method highlights: HoPLAA [109], [110] reorganises ATAM (section 3.6.2) in order to make it more suitable for evaluating product line architectures rather than single product architecture. The key observation underlying HoPLAA is that to evaluate product line architecture it is necessary to evaluate both the common architecture for the entire product line (known as the "*core architecture*") as well as the architectures of the individual products. The former should address the requirements common to all the products in a product line, while the latter should include both common and product-specific requirements. The main idea delivered by HoPLAA is to analyse the core architecture first (phase I), then to analyse the product architectures (phase II) to confirm whether these preserve the support for the common requirements while supporting the requirements specific to every individual product. HoPLAA is an extension of ATAM, so the description below emphasises only the differences between these two methods.

Evaluation goals: as in ATAM, i.e. established during the analysis.

Examined properties: HoPLAA can be applied in order to assess product line architecture against any quality attributes.

Evaluation process:

Phase I. Core Architecture (CA) evaluation

Step I.1. Present HoPLAA for phase I

Step I.2. Present the product line architectural drivers – business goals motivating the product line, its scope as well as commonality and variability of products, especially concerning quality goals.

Step I.3. Present the product line architecture – architects present the Core Architecture.

Step I.4. Identify architectural approaches

Step I.5. Generate, classify, brainstorm, and prioritise quality attribute scenarios.

Step I.6. Analyse architectural approaches/generic scenarios

Step I.7. Present results. A report summarising the outcomes of phase I should be delivered.

Phase II. Product Architecture (PA) evaluation

Step II.1. Present HoPLAA for phase I

Step II.2. Present the product architectural drivers

Step II.3. Present the product architecture

The presentation should concentrate on the parts of architecture that have been enhanced through the implementation of variation points.

Step II.4. Identify architectural approaches

Step II.5. Generate, classify, brainstorm, and prioritise quality attribute scenarios

Step II.6. Analyse architectural approaches/generic scenarios

Step II.7. Present the results. (A report summarising the outcomes of phase I should be delivered.)

Architecture description: HoPLAA requires a description of the core architecture and the product architectures as required by ATAM; in particular, the architectural approaches included in an analysed architecture should be identified.

Representation of evaluated requirements: quality scenarios represented as in the ATAM (source, stimulus, artefact, environment, response, response measure). A utility tree should also be developed, similarly to ATAM.

Techniques for eliciting requirements: quality scenarios are brainstormed in both phases of HoPLAA. In phase I, both scenarios that are common to the entire product line (called “*generic scenarios*”) as well as those specific to the individual products are elicited, while in phase II it is possible to add only the quality scenarios that are specific to the concrete products.

Techniques for prioritising requirements: in phase I, core architecture scenarios are ranked for the analysis according to their generality, significance and cost using a three-level scale: Low (1), Medium (2), High (3). The sum of these three numbers gives the priorities. For analysing product line architecture (phase II), scenarios should be ranked using any prioritisation technique, e.g. voting, cumulative voting or establishing priorities through discussion.

Architecture Evaluation Techniques: analysing architectural approaches with respect to the affected quality scenarios seems to be the main evaluation technique used in HoPLAA. Nevertheless, the method does not forbid the use of any other architecture evaluation technique, as with ATAM. Generic scenarios with the highest priority are inspected during the evaluation of the core architecture, while both generic and product specific scenarios should be analysed during the analysis of product line architectures. This allows confirmation that both common and product specific requirements are supported by the architectures of individual products.

Evaluation outcomes: the outcomes are generally similar to ATAM and include lists of architectural approaches, utility tree, generic scenarios, product-specific scenarios identified, areas of risks in the CA, non-risks, sensitivity points, trade-offs, and risk themes. These should be delivered for the core architecture, as well as for each of the product architectures. Additionally, evolvability points and evolvability guidelines should also be produced for the core architecture. An evolvability point is a sensitivity or trade-off point (compare section 3.6.2) that contains at least one variation point. An evolvability point denotes part of an architecture that can be defined individually for each product, and that can influence common quality requirements. Evolvability guidelines should indicate how to develop product architectures without violating common requirements.

Method validation: several practical examples [110], [109].

Method limitations, discussion: as HoPLAA extends ATAM, its weaknesses are similar to ATAM’s. They include the high level of effort required for such a comprehensive analysis, and the method’s intrinsic complexity, etc. (compare section 3.6.2 “Discussion”). However, such an assessment performed during the development phase is particularly valuable in the case of a product line, because its architecture will be re-used by a family of software products, which will be evolving over many years.

3.9.2. Early Architecture Evaluation Method (EAEM)

The presentation of the Early Architectural Evaluation Methods has been developed by reorganising parts of [169], which extended the evaluation method sketched out in [171].

Method highlights: the EAEM was designed to analyse architectures of large-scale distributed systems very early in the development lifecycle, i.e. in the inception phase of the Rational Unified Process. Such early architecture is represented as a set of architectural decisions defining the backbone of a large-scale system referred to as the System Organisation Pattern. This comprises functional decomposition (e.g. domain, systems and applications), geographical allocation, the organisation of data input, the organisation of distributed data storage and processing. As many practical examples show, these early architectural decisions often play a critical role in a development project. They may pose a substantial risk, which has to be properly managed in order to avoid the development project failing. The EAEM is aimed at identifying these prevailing risks. These are indicated by a Goal-Question-Metric (GQM) [14] model that is the core of the method. An evaluation process has been precisely defined in order to ensure that the evaluation is made in a disciplined way.

Evaluation goals: identification of significant risks.

Examined properties: buildability, performance, reliability.

Evaluation process: the evaluation process is summarised in table 12.

Table 12. Summary of the EAEM's evaluation process

	Input	Output	Roles involved	Description
Preparation	None or architectural documentation	Description of the System Organisation Pattern	Architects, evaluators	Elicitation of the System Organisation Pattern
Evaluation	Description of the System Organisation Pattern	Completed evaluation model	Architects, evaluators	Fill-in the GQM scheme according to the System Organisation Pattern.
Risk identification and mapping	Completed evaluation model, Description of the System Organisation Pattern	Risk list	Architects, evaluators	Identify risks by investigating the critical metric values.
Risk assessment and feedback	Assessed risk list	Assessed risk list	Evaluators, relevant project stakeholders (incl. architecture stakeholders, compare [16])	Assess risk severities and propose mitigation tactics.

Architecture description: EAEM evaluates the System Organisation Pattern, which has already been presented in section 2.2.6.

Representation of evaluated requirements: the System Organisation Pattern should serve at least the following three purposes:

- to avoid overly complex designs that might not be achievable by the development team – the goal of “complexity control”;
- to ensure that a system’s architecture is in line with the system’s context (e.g. the target organisation’s structure, the structure of the controlled installation) – the goal of “context adequacy”; and
- to provide a robust platform for the entire system being developed, i.e. one that will not threaten the performance and reliability of the system – the goal of “satisfactory performance and reliability”.

These goals universally concern all software systems, especially large-scale ones, and have to be achieved well before more detailed design goals become achievable. Therefore, in EAEM, the requirements are represented as goals of a Goal-Question-Metrics scheme, namely “complexity control”, “context adequacy”, “satisfactory performance and reliability”. System organisation pattern should enable the achievability of these goals, as they will have to be finally fulfilled by the detailed design.

Techniques for eliciting requirements: a fixed set of requirements built into the GQM scheme is evaluated in the EAEM.

Techniques for prioritising requirements: not needed, the GQM goals are equally important.

Techniques for assessing architecture: a Goal-Question-Metric model facilitates risk identification in EAEM. The goals reflect the enabling role of the System Organisation Pattern (compare section “representation of evaluated requirements” above). These questions, in turn, explore concrete architectural decisions in order to identify those that may expose the related goals to any substantial risk. Such decisions that may affect the achievability of any of the above goals are, in ATAM’s terminology, sensitivity points. Metrics assigned to the questions identify the level of risk that a certain architectural decision may cause. Scales for all the metrics have been defined: the metric values have been ranked in ascending order (from the worst to the best). Critical values for each of the metrics have been highlighted, as they indicate that there may be a critical design risk.

The GQM-based architecture evaluation model is presented below.

GOAL: Complexity Control

– *Decomposition into a set of subsystems/applications*

DSA.01. Are the specified subsystems/applications functionally consistent?

Metric: Level of cohesion.

Scale: Coincidental, Logical, Temporal, Functional.

Critical values: Coincidental, Logical.

Comment: The concepts of coupling and cohesion were proposed in the 1970s in [141], and in [161]. The cohesion scale adopts the scale proposed in [161]. The levels of coupling are defined as follows:

Coincidental – module (subsystem/application) comprises multiple, unrelated functions serving a variety of purposes; Logical – module contains a set of functions belonging to the same genre, but they serve various purposes; Temporal – module implements a set of functions whose execution is related in time (e.g. initiation functions); Functional – module serves only a single purpose (e.g. transmitting data from/to local subsystems).

DSA.02. Are the specified subsystems/applications sufficiently independent?
Metric: Level of coupling.
Scale: Content, Common, Control, Data, No coupling.
Critical values: Content, Common, Control.
Comment: The above coupling levels are an adaptation of the scale proposed in [113]. The consecutive coupling levels are defined as follows: Content – two modules (subsystems/applications) are directly accessing and changing each other's internal data sets; Common – two modules refer to the same common data sets; Control – one module steers the execution of the other one; Data – modules exchange data using a communication mechanism; No coupling – modules are independent of each other;

– *Data storage distribution*

DSD.01. Does the data storage distribution, together with functional decomposition, imply the need to transfer data between databases?
Metric: Level of database dependence;
Scale: High – there are data storages dependent on each other – on line synchronisation is needed; medium – data upload/download/ /transfer between databases has been provisioned for; Low – databases are independent or there is a single database, and so synchronisation between databases is not needed.
Critical values: High.

– *Data processing organisation*.

DPO.01. Is advanced local data storage needed for the client application?
Metric: Kind of client application.
Scale: Thick, Hybrid, Thin.
Critical values: Thick.
Comment: Only a thick client contains advanced local data storage (e.g. relational database), which can give rise to problems with long transactions and/or data synchronisation between local and central storages.

DPO.02. What is the duration of transactions necessary to manage the data storage of a client application?
Metric: Length of transactions.
Scale: Unknown, Long, Short.

Critical values: Unknown, Long.

Comment: This question encompasses the case of client storage in a multi-tier application.

DPO.03. Will 'central' data be uploaded/downloaded from/onto local applications/subsystems?

Metric: The existence of the need for local data upload/download.

Scale: Exists/Does not exist.

Critical values: Exists.

– *Management of distributed data storage.*

DSM.01. What is the confidence in the chosen database synchronisation/data upload solutions?

Metric: Level of confidence.

Scale: Low – the synchronisation will be implemented entirely by the development team; high – the synchronisation will base on proven commercial or proven open source solutions.

Critical values: Low.

– *Transaction management framework.*

TMF.01. What is the confidence in the selected long transaction management framework?

Metric: Level of confidence.

Scale: None –pp.problems with long transactions have been noticed, low – the long transaction management will be designed by the development team, high – proven solutions will be used (e.g. commercial transaction monitor)

Critical values: None, Low.

– *Communication framework.*

CF.01. Do the development technologies of choice provide a uniform communication framework ensuring data exchange between equivalent software units (e.g. processes, distributed objects, etc.)?

Metric: The existence of uniform communication mechanisms for the whole software.

Scale: No, Yes.

Critical values: No.

CF.02. Is the communication between the system entities uniform from the software developer's point of view?

Metric: Number of protocols that the developer has to be aware of?

Critical values: Greater than one.

GOAL: Context Adequacy

– *Geographical and organisational allocation of the subsystems/applications.*

GOA.01. Is the allocation of subsystems/applications to the organisation's units adequate for the organisation's structure?

Metrics: a) Number of organisational units where necessary functions are unavailable; b) number of organisational units to which superfluous applications/subsystems have been allocated.
Critical value: a) Any higher than 0; b) not specified.

GOA.02. Is the geographical allocation of subsystems/applications adequate for the geographical distribution of a given organisation?
Metrics: a) Number of locations where necessary functions are unavailable; b) number of locations where superfluous applications/subsystems have been allocated.
Critical value: a) Any higher than 0; b) not specified.

– *Organisation of data input*

ODI.01. Are the physical data input points (scanning devices, typing stations, sensors) placed at points nearest to the data sources?
Metrics: a) Time need to move the documents from source to the data input point; b) distance between source and data input points.
Critical value: To be established individually.

ODI.02. Is the electronic data input accessible to the users or systems entering data?
Metrics: Share of users or external systems that cannot input data electronically being adequately equipped.
Critical value: To be established individually.

GOAL: Satisfactory Performance and Reliability

– The questions below refer by analogy to the respective System Organisation Pattern components indicated above.

DSA.01. Defined earlier.

TMF.01. Defined earlier.

DSM.01. Defined earlier.

DPO.02. Defined earlier.

CF.03. Is the communication mechanism planned for distant data transfer capable of coping with a varying intensity of data load?
Metric: Level of suitability for a varying data load.
Scale: Weak – only synchronous communication has been planned; Strong – asynchronous communication, including queuing services, has been provisioned for.
Critical values: Weak.

CF.04. Does the communication mechanism planned for distant data transfer guarantee sufficient communication reliability?
Metric: Confidence in the communication mechanism; failure ratio; ratio of the number of negative application report to the total number of applications.

Critical value: Low, high (for confidence metric); to be established individually (other metrics).

ODI.03. Is the data entering the system accurate enough?
Metrics: Inaccuracy of data input devices; level of disturbances (e.g. noise) in the system's environment.
Scale: Low, medium, high (both metrics), unknown;
Critical value: High, unknown.
Comment: The question applies only to systems using scanning or sensor devices for data input.

ODI.04. Has the reliability of data input and transfer through the system's entities been accounted for?
Metrics: Number of data entry points, number of routes through which data is transferred from input devices to system entities.
Critical value: One (for both questions).

The critical value of a metric indicates a critical architectural decision that may endanger the achievability of related goals. In order to identify concrete risks, the influence of the critical architectural decision on related goals and other system elements should be considered. For example, the choice to implement a database synchronisation mechanism from scratch may turn out to be too complex to be properly implemented and tested within the project's schedule, which may endanger the goal of "complexity control". At the same time, there is a risk that the implemented mechanism may turn out to be defective, endangering the goal of "satisfactory performance and reliability", and that failures of database synchronisation may result in failures of the subsystems using this mechanism.

In general, the risks themes connected with the evaluated goals are typically:

- in the case of "Complexity control" – the ability of the project team to deliver features defined by the "sensitive" architectural decisions, or features that require those mechanisms;
- in the case of "Context adequacy" – access to the system not delivered to appropriate organisation entities (departments, geographic locations, or individuals), and problems with feeding external data into the system, e.g. inaccurate, distorted or partially missing data;
- in the case of "Sufficient performance and reliability" – failures of the mechanisms defined by the critical architectural decisions, or their inadequate performance.

Identified risks should be assessed in order to estimate the actual level of risk, which is determined by the product of the probability of occurrence and risk impact. The choice of scale is at the discretion of the organisation, though three-level scales like: low, medium, high, are usually the least confusing and easiest to use.

The risk level assessment has to be done on the basis of knowledge on the context of a project, e.g. the skills and experience of the development team, the experience of the maintenance organisation, the scope and schedule of the project, environmental factors (such as the level of electromagnetic noise), dependencies between the system's entities etc. For example, the probability that database synchronisation will not be completed within the project schedule can be high if the development team has no experience in this, and if there are short deadlines, which can be quite the opposite in the case of an experienced team.

Evaluation outcomes: list of risks and suggested mitigation tactics.

Method validation: seven real-life examples of large-scale systems – all the details are included in [169]. Two illustrate how the Early Architecture Evaluation Method works let us consider again an example of an interbank clearing system (section 2.2.6). Consecutive steps of the Early Architecture Evaluation Method applied have been presented in detail below.

Step 1. Preparation

The elicited description of the System Organisation Pattern has been summarised in table 13.

Table 13. Description of the System Organisation Pattern of interbank clearing system

Component of the System Organisation Pattern	Architectural Decision
Decomposition into a set of subsystems/applications	Clearing processing system – processes wire transfer data. FTP gateway – receives packages of wire transfers data sent from client applications and makes them accessible for the clearing processing system The client application – verifies the wire transfer data entered from the bank's systems before the clearing process, sends and receives data from the FTP gateway.
Data storage distribution	There are no distributed data stores. Each subsystem is equipped with its own database but only for its internal use.
Data processing organisation	Local subsystems follow hybrid client architecture.
Management of distributed data storage	Distributed data management is not needed.
Transaction management framework	Long transactions are not processed by the system.
Communication framework	Asynchronous communication between the client application and the FTP gateway over the FTP protocol. A similar situation occurs in the case of communication between the FTP gateway and the clearing system.
Geographical and organisational allocation of the subsystems/applications	The central subsystem is situated in the premises of the headquarters. Over 50 local subsystems are located in banks across Poland.
Organisation of data input	The data is automatically entered into local subsystems from the bank's systems.

Step. 2. Evaluation

The System Organisation Pattern presented in table 13 has been reflected in the completed GQM assessment model depicted in tables 14 and 15.

Table 14. Risk identification and mapping for the "complexity control" goal for interbank clearing system

Question	Metric	Metric Value	Risk
DSA.01. Are the specified subsystems/applications functionally consistent?	Level of cohesion	Functional	–
DSA.02 Are the specified subsystems/ applications sufficiently independent?	Level of coupling	Data	–
DSD.01 Is it necessary to transfer data between databases	Level of database dependence	Low	–
DPO.01 Is advanced local data storage needed for the client application?	Kind of client application	Hybrid	–
DPO.02 What is the duration of transactions necessary to manage the data storage of a client application?	Length of transactions	Short	–
DPO.03 Will 'central' data be uploaded/downloaded from/onto local applications/subsystems?	Existence of the need for local data upload/download	**Exists**	R.1. The data download/upload mechanism will not be implemented on time?
DSM.01 What is the confidence in the chosen database synchronisation/data upload solutions?	Level of confidence	High	–
TMF.01 What is the confidence in the selected long transaction management framework?	Level of confidence	N/A	–
CF.01 Do the development technologies of choice provide a uniform communications framework ensuring data exchange between equivalent software units (e.g. processes, distributed objects, etc.)?	Existence of uniform communication mechanisms for the whole software	Yes	–
CF.02 Is the communication between the system entities uniform from the software developer's point of view?	Number of protocols that the developer has to be aware of?	1	–

Table 15. Risk identification and mapping for "sufficient performance" goal (interbank clearing system)

Question	Metric	Metric's value	Risks
DSA.01. Are the specified subsystems/ applications functionally consistent?	Level of cohesion	Functional	–
TMF.01 What is the confidence in the selected long transaction management framework?	Level of confidence	N/A	N/A
DSM.01 What is the confidence in the chosen database synchronisation/data upload solutions?	Level of confidence	High	–
DPO.02 What is the duration of transactions necessary to manage the data storage of a client application?	Length of transactions	Short	–
CF.03 Is the communication mechanism planned for distant data transfer capable of coping with a varying intensity of data load?	Level of suitability for a varying data load	N/A	N/A
CF.04 Does the communication mechanism planned for distant data transfer guarantee sufficient communication reliability?	Confidence in the communication mechanism	High	–
ODI.03 Is the data entering the system accurate enough?	N/A	N/A	N/A
ODI.04 Has the reliability of data input and transfer through the system's entities been accounted for?	Number of data entry points	2 – FTP Gateway and emergency mechanism by an external web portal.	–

Step. 3. Risk identification and mapping

There is only 1 critical metric value in the completed architecture evaluation model. No wonder, the list of identified risks is rather short.

Step. 4. Risk assessment and feedback

Table 16 (risk register) contains only one risk item and its recommended treatment. Let us note that this recommends that this suggests that the considered interbank clearing system was developed on a sound System Organisation Pattern.

Table 16. Risk assessment and proposed treatment (interbank clearing system)

Risks	Probability/impact	Recommended risk mitigation tactic
R.1. Data download/upload mechanism will not be implemented?	Low/High	Contingency – ensure manual data exchange via FTP, or even hard media

Method limitations, discussion: The Early Architecture Evaluation Method does not require advanced architecture documentation, but it can be applied as soon as any form of System Organisation Pattern description is available. Thanks to that, it can be applied much earlier than the Architecture Trade-off Analysis Method, i.e. in the inception phase of the Rational Unified Process. In reality, the System Organisation Pattern is often defined at pre-project stages, and then it becomes a kind of architectural skeleton for the subsequently initiated development project. The method seamlessly integrates with project management by feeding assessed risks and recommended mitigation tactics into risk management procedures contained in every project management methodology, such as Prince2 or PMBoK.

ATAM is a general-purpose, business-goal driven architecture assessment method. Analysis goals, focus, examined properties and priorities are all established during the assessment, with the active involvement of the architecture stakeholders. This versatility is achieved at the expense of an elaborate assessment procedure, as well as the skills and knowledge needed to facilitate such a full analysis process.

The Early Architecture Evaluation Method does not impose a complicated assessment procedure, its context is limited to the System Organisation Pattern, and its goal is to identify substantial design risks. Achieving these general goals is a necessary precondition for meeting more concrete quality requirements supporting specific business goals. As business goals are not investigated throughout the analysis, it is assumed that there already exists a business case for the development of the system subject to the evaluation. This helps avoid the complexity generated by tracing business goals throughout the evaluation process, at the cost of limited versatility and scope of analysis.

The pattern-based assessment advocated in [66], overcomes many of the limitations of ATAM. However, two basic factors limit the application of their approach to the System Organisation Pattern: the first is that the assessment is carried out on a kind of implemented architecture prototype, known as a "walking skeleton", which is a rare case in the development of large-scale systems; the second is that the early, organising decisions often do not provide sufficient information on architectural patterns included in the design, which makes pattern-based reasoning impossible.

3.10. DISCUSSION: STATE OF ART AND PRACTICE IN ARCHITECTURE EVALUATION

This section summarises the survey of architecture evaluation methods presented in sections 3.6–3.9 and describes state of the art and practice. On this foundation, further research needed to overcome the limitations of the existing methods has been envisaged – section 3.11.

3.10.1. Evaluation goals, examined properties, form of architecture description

Architecture evaluation methods can be characterised according to many properties (compare section 3.4). This section summarises the survey with respect to important features of the presented methods.

Evaluation goals: architecture evaluation methods have been designed to serve the following goals:

- Risk analysis;
- Verification of architectural support for business goals;
- Comparison of architectures with respect to some characteristics;
- Prediction of the cost of modifications;
- Detecting architectural flaws or requirements not met.

Examined properties: in fact, the support for almost every non-functional quality requirement can be subject to the evaluation. A complete list of quality attributes with methods suitable for analysing them can be found in section 3.3.

Architecture description: architecture evaluation methods use various forms of architecture description – see table 17.

Table 17. Forms of architecture description used by the architecture evaluation methods

Form of architecture description	Method acronym
Architectural decisions	EAEM
Component-connector models	SAAM, ALMA, SARAH, COSAAM, ASAAM,
Full architecture description (views)	ATAM, APTIA, CBAM, Lightweight-ATAM, HoPLAA
Architectural patterns, tactics, approaches, solutions, constructs etc.	ATAM, PBAR, PASA, AREA, SALUTA, HoPLAA
No particular form of architectural documentation assumed, elicitation of architectural description during the analysis, partial description	AREA, ARID, PBAR, TARA, SHADD,

3.10.2. Architecturally Significant Requirements: Representation, Eliciting, Prioritising

Representation of evaluated requirements: scenarios are the main form of capturing architecturally-relevant requirements, though they may differ in detail, hence, the following types of scenarios have been defined:

- Non-formalised scenarios – SAAM, ASAAM, COSAAM, ALMA, AREA;
- Full quality scenarios, introduced by ATAM and used by its derivatives: APTIA, Lightweight ATAM, CBAM, HoPLAA;
- Failure scenarios – represent failures of the components described by fault, error and failure, compare SARAH in section 3.8.6;

- Usability scenarios – capture users, their tasks and context of use as well as expected levels of usability attributes, for details see: SALUTA, section 3.8.2;
- Performance scenarios – key performance scenarios defined with regard to the critical use cases and their most frequently used interaction alternatives.

Eliciting relevant requirements and prioritising them belong to the activities that have to be included in the evaluation methods representing the "assessment against requirements" paradigm (compare section 3.1). The majority of such methods employ brainstorming as a technique for eliciting requirements. In most cases requirements are represented as scenarios. Architecture evaluation methods facilitate elicitation by:

- Defining templates for the scenario's content and its structure, compare ATAM, HoPLAA, SALUTA, SARAH;
- Defining the structure of a domain model whose instantiation will be used for specifying requirements, compare SALUTA, SARAH;
- Providing appropriated guidance, which can take the form of:
  - patterns of the scenarios for specific quality attributes, compare ATAM;
  - indicating how and where to search for the quality scenarios, compare ALMA, PASA, HoPLAA.

An interesting observation seems to be that only PBAR assumes requirements specification as a possible source of the relevant requirements for an architecture evaluation. It is even more surprising if one considers that the majority of architecture evaluation methods are supposed to be applied when some form of a software design is ready. At such an advanced stage, the most important requirements should already have been identified.

The prioritisation techniques included in the architecture verification methods are:

- cumulative voting – SAAM, ATAM and its successors, namely HoPLAA, CBAM, APTIA,;
- classifying scenarios – SAAM (used together with cumulative voting), ALMA;
- analysing failure severities (SARAH);
- Analytical Hierarchical Process (AREA);
- By using some form of expert judgement (SALUTA, AREA, PASA).

None of the architecture evaluation methods employs Must-Should-Could-Won't have technique (MoSCoW) [99], [69] prioritisation technique, which is very popular in practice.

3.10.3. Architecture Evaluation Techniques

A careful analysis of the survey of architecture evaluation methods leads to the conclusion that they are generally not very prescriptive with respect to the analysis techniques recommended for analysing architecture's properties. In general, most of the employed analysis techniques facilitate gathering information necessary to be resolved by experience-based reasoning, or support some of the activities needed to reach evaluation's conclusion (compare, for example, the use of AHP in AREA method, section 3.8.5, or the description of techniques for architecture analysis in SALUTA). Table 18 contains a catalogue of analysis techniques employed by the methods presented in this monograph.

Table 18. Architecture evaluation techniques employed by the architecture evaluation methods

Technique for analysing architecture	Methods employing a given technique
Analysing component/component-concern dependencies	COSAAM, ASAAM
Analytical Hierarchical Process	AREA
Effort calculation model	ALMA
Experience-based reasoning	In fact: all the methods.
Fault Tree Analysis	SARAH
Goal Question Metric	SAEM, EAEM
Identification of parts of architectures responsible for certain properties	SAAM, ALMA, SHADD
Mathematical model-based analysis	CPASA (queuing networks)
Metrics	TARA
No detailed recommendation	ATAM, APTIA, ARID, Lightweight ATAM
Reasoning based on the identified anti-patterns or architectural bad smells based [62], [125]	PASA
Reasoning based on the influence of identified architectural patterns and tactics on quality attributes	PBAR, ATAM, Light-weight ATAM, APTIA
System/project log analysis	TARA
Usability framework	SALUTA

Although there are many models and methods for analysing software properties that can also be applied at an architectural level, evaluation methods seldom recommend them as suitable Architecture Evaluation Techniques. Nevertheless, there are methods, such as ATAM, that do not exclude the use of any analysis technique.

3.10.4. Properties of Architecture Evaluation Methods Resulting from their Design

Stage of applicability: table 19 contains a classification of architecture evaluation methods by the typical stage of their applicability.

Table 19. Typical stages at which architecture evaluation methods can be applied

Development stage	Methods, for which given development stage is a typically stage of application
Inception[(1)]	EAEM
Elaboration[(1)]	ARID, PBAR, ATAM, Lightweight-ATAM, SHADD, APTIA, PASA, CPASA, SARAH, SAAM, ASAAM, COSAAM, HoPLAA
Post-deployment	TARA, AREA

[(1)] phases of the Rational Unified Process [95]

Method validation: the architecture evaluation methods have been classified according to the available record of their validation in table 20. The following scale for the extent of validation record has been assumed:

- Scarce – there is at most a single example of a method's application, whose content may be sketchy;
- Small – there is at least one detailed example of a method's application, the total number of documented applications does not exceed three.
- Significant – there are more than four examples, at least some of them have been presented in detail;
- Extensive – there are more than 10 documented examples of a method's application.

Table 20. Validation of the architecture evaluation methods

Validation	Methods
Scarce	APTIA, SHADD, ASAAM, COSAAM, Lightweight-ATAM, SARAH, CPASA
Small	AREA, SALUTA, PASA, CBAM
Significant	ALMA, PBAR, EAEM
Extensive	ATAM, SAAM (together with its early variations – see section 3.6.1)

3.10.5. Architecture Evaluation: State-of-Practice

The state of industrial practice in the area of architecture evaluation has been surveyed in [7]. Data obtained from 86 software industry respondents revealed the following main facts:

- Architecture assessment methods developed by the research community are hardly recognised by practitioners (24% were aware of ATAM, 17% of PASA, 6% of ALMA);
- Architecture assessment methods are hardly ever used by practitioners (5% of the respondents declared the organisation is using ATAM and SAAM, 1% – PASA and ALMA). However, these methods are not used as out of the box tools, but the organisations' processes are derived from the academic methods;

- Most popular Architecture Evaluation Techniques are: experience-based reasoning (83%), prototyping (70%), scenarios (56%) and checklists (40%), while metrics (15%) and mathematical models (5%) are rarely employed in practical engineering;
- Architecture reviews are performed at the early stages of development by 80% of respondents, at middle stage – by 34%, at post-deployment stage – by 15%, at re-engineering stage – by 28%, for system acquisition – by 11%;
- The existence of a formal architecture review process was reported by 41% of the participants, while 56% indicated an informal review process.

Therefore, the obvious conclusion is that the research developments still remain far away from the industrial practice. According to the survey's authors, this condition is rooted in:

- The lack of a convincing business case showing the benefits and surplus resulting from architecture assessment;
- Problems with porting the architecture assessment methods onto industrial ground, supposedly caused by the lack of tool support, problems with including architectural evaluation into the established development processes as well as problems with know-how transfer.

3.11. STATUS AND PROSPECTS OF ARCHITECTURE EVALUATION

Since the first architecture evaluation method appeared in 1994, more than twenty five such methods have been developed. Newer and newer methods are still emerging. It is a noticeable fact that since 2010 alone, at least six new methods have been presented (AREA, TARA, PBAR, CPASA, SHADD, EAEM). The state of practice described in section 3.10.5 indicates that the research achievements are still far from shaping the industrial practice. Apart from the reasons for this condition, as presented in section 3.10.5, there are also important issues inherent to the internal design of the methods developed to date, which generally limit their usefulness and proliferation in practice:

1. Most of the architecture evaluation methods organise and facilitate an evaluation rather than providing models and techniques for drawing conclusions regarding the properties of a software architecture. These methods define the evaluation processes and support for their activities (mainly the elicitation and prioritisation of requirements); however, the very architectural analysis is done by the experience-based reasoning of the evaluator;
2. The "architectural wisdom" [169] needed to establish the outcomes of an evaluation is rarely "sold" together with the method (SALUTA and EAEM are exceptions to that general rule). In most cases, it has to be already in the pos-

session of some of the company's employees, or has to come with an external expert;

3. The existing architecture evaluation methods analyse architectures documented informally, or semi-formally at best. It implies that the architecture analysis has to be performed by an informal reasoning of the expert (evaluator) or other participants of the evaluation.
4. The proliferation of the use of informal architecture models hinders the use of model-based evaluation techniques, which can deliver decisive, objective conclusions.

In order to overcome the limitations of the existing architecture evaluation methods, further research is needed in the following directions:

- Reconstructing existing architecture evaluation methods or developing new ones that seamlessly integrate with the established software development methodologies, such as the Rational Unified Process, waterfall or agile;
- Developing "self-contained" architecture evaluation methods that deliver processes, models and the architectural wisdom needed to perform an evaluation. Architectural wisdom would have to be provided in the form of a knowledge base that could be used during an architecture evaluation;
- Integrating established mathematical models into architecture evaluation methods. The list of such models includes: queuing networks and other stochastic models (e.g. Markov chains), real-time analysis, temporal logic, formal systems enabling theorem proving or model checking, concurrency models (process calculi, Petri nets). This also means that formal models should become more popular in the area of architectural modelling;
- The development of architecture evaluation methods suitable for specific domains of applications, e.g. for service-oriented architectures, real-time systems or distributed systems. Let us note that the number of domain-specific methods is rather small, as section 3.3 shows. The same applies to attribute-specific methods – most attributes are addressed by just a single architecture evaluation method.

4. CONCLUSION

Brooks' famous paper on silver bullets [31] indicates that the general challenge of software engineering is to deliver means of coping with software complexity. Research on software architecture, which has been thriving since the early 1990s, is another effort to develop such means. This work takes an overview and discusses the state of the art in modelling and evaluating software architectures.

The survey of approaches to architectural modelling, presented in chapter 2, indicates clearly that not all developments in this area support that general purpose. Large sets of architectural decisions tend to produce complexity of their own, rather than help to manage the complexity of the design. Organising architecture descriptions around a set of architectural viewpoints brings the challenge of defining correspondence and ensuring consistency between models belonging to different views. At the same time, the usefulness of the comprehensive and elaborate documentation is generally questioned by the agile development community.

The concept of the System Organisation Pattern shows that architectural decisions can efficiently capture complex design concepts, while modelling languages such as SysML and ArchiMate still require broader validation in practice. The other issues raised above comprise challenges that will have to be addressed by future research.

Research on architecture evaluation methods was carried out in parallel with that on architectural modelling. As a result, more than 25 such methods have been developed during the last twenty years. Eighteen state-of-the-art architectural evaluation methods have been covered by this monograph, and an additional seven legacy architecture evaluation methods have also been briefly characterised. The taxonomy shows that new evaluation methods should be developed for specific domains of application, as well as for certain quality attributes (esp. performance and reliability). The Early Architecture Evaluation Method proves that such specialised methods can deliver substantial value by minimising the risk of the development failing. This could provide a convincing business case for a wider industrial adoption of architecture evaluation methods.

The survey and discussion of the state of the art in architecture modelling and evaluation, presented in this dissertation, reveals constant progress observable in this research domain. The author’s contribution was aimed at overcoming the limitations of the existing achievements and comprises:

- Diagrammatic model of architectural decisions, known as Maps of Architectural Decisions (MAD), suitable for documenting the evolution of rapidly evolving systems;
- The System Organisation Pattern shows how architectural decisions can tersely and comprehensibly represent architectures of large-scale distributed systems, facilitating the management of software complexity;
- The taxonomies of architecture evaluation paradigms and architecture evaluation methods;
- Early Architecture Evaluation Method – an original architecture evaluation method, applicable at the earliest development stages of large-scale distributed systems.

The research achievements in architectural modelling and architecture evaluation have not yet become pervasive for the software industry. However, the transition of information technologies into widespread use usually takes 15-20 years [122]. Hence, a wider adoption of the achievements of software architecture research is likely to be seen in the years to come. This hypothesis is partially confirmed by the increasingly widespread use of the word ‘architecture’ as being synonymous with ‘design’, by including the concepts of stakeholders’ concerns, viewpoints and views into the ISO 42010:2011 international standard of architecture description, as well as by the appearance of architecture modelling notation (ArchiMate) that are compliant with that standard.

BIBLIOGRAPHY

[1] Abdullatif A.A. and Pooley R.J.: UML-JMT: a Tool for Evaluating Performance Requirements. 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems. IEEE, 2009.

[2] Abowd G., Bass L., Clements P., Kazman R., Northrop L., Zaremski A.: Recommended Best Industrial Practice for Software Architecture Evaluation. CMU/SEI-96-TR-025. Carnegie Mellon University, 1997.

[3] Alexander Ch. Notes on the synthesis of form. Harvard University Press, Cambridge, Massachusetts, 1964.

[4] Alexander Ch., Ishikava S., Silverstein M., et al. A Pattern Language: towns, building, construction. Oxford University Press, New York, 1977.

[5] Arsanjani A., Ghosh S., Allam A., Abdollah T., Ganapathy S., Holley K.: SOMA: A method for developing service-oriented solutions. IBM Systems Journal, vol. 47, No 3, pp. 377-396. IBM 2008.

[6] Babar M.A., Dingsøyr T., Lago P., van Vliet H.: Architecture knowledge management. Theory and Practice. Springer-Verlag, Berlin Heidelberg, 2009.

[7] Babar M.A., Gorton I. Software Architecture Review: The State of Practice. Computer 42, No 7, pp. 26-32, IEEE 2009.

[8] Babar M.A., Zhu L., Jeffery R.: A framework for classifying and comparing software architecture evaluation methods. Australian Software Engineering Conference, 2004, pp. 309-318, IEEE 2004.

[9] Baldwin C.Y. and Clark B.K.. Design Rules, Vol. 1: The Power of Modularity. First Edition. The MIT Press, 2000. ISBN: 0262024667.

[10] Baldwin C.Y. and Woodard C. J.: Competition in Modular Clusters. Harvard Business School, 2007. (available online at http://hbswk.hbs.edu/item/5842.html)

[11] Baldwin C.Y., and Clark K.B. Sun wars: Competition within a modular cluster, 1985–1990. In D.B. Yoffie (Ed.), Competing in the age of digital convergence: 123-158. Harvard Business School Press. Boston, 1997.

[12] Balsamo S., Di Marco A., Inverardi P., Simeoni M.: Model-based performance prediction in software development: a survey. IEEE Transactions on Software Engineering, vol. 30, No 5, pp. 295-310, IEEE 2004.

[13] Barcelos R. and Travassos G.: Evaluation approaches for software architectural documents: a systematic review. In Ibero-American Workshop on Requirements Engineering and Software Environments (IDEAS'06), 2006.

[14] Basili V.R., Caldiera G., Rombach H.: The Goal Question Metric Paradigm in: J.J. Marciniak (Ed.), Encyclopedia of Software Engineering, John Wiley & Sons, Inc., New York, 1994, pp. 528-532.

[15] Bass L., Clements P., Kazman R.: Software Architecture in Practice, second edition. Addison-Wesley, 2003. ISBN 0-321-15495-9.

[16] Bass L., Clements P., Kazman R.: Software Architecture in Practice, third edition. Addison-Wesley, 2012. ISBN 0-321-81573-4.

[17] Bass L., Nord R., Wood W., Zubrow D., Ozkaya I.: Analysis of architecture evaluation data. Journal of Systems and Software, Sep. 2008, vol. 81, iss. 9, pp. 1443-1455; Elsevier, 2008.

[18] Bass L., Nord R., Wood W., Zubrow D.: Risk Themes Discovered Through Architecture Evaluations. TECHNICAL REPORT CMU/SEI-2006-TR-012 ESC-TR-2006-012. Carnegie Mellon University, 2006.

[19] Bengtsson P., Lassing N., Bosch J., van Vliet H.: Architecture-level modifiability analysis (ALMA). Journal of Systems and Software, Volume 69, Issues 1-2, 1 January 2004, pp. 129-147. Elsevier 2004.

[20] Bengtsson P.O. and Bosch J.: Architecture Level Prediction of Software Maintenance, Proc. Third European Conf. Software Maintenance and Reeng., pp. 139-147. IEEE 1999.

[21] Bengtsson P.O. and Bosch J.: Scenario-Based Architecture Reengineering, Proc. Fifth Int'l Conf. Software Reuse (ICSR 5), pp. 308-317. IEEE, 1998.

[22] Bennett K.H. and Rajlich V. T.: Software maintenance and evolution: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, pp. 73-87, ACM 2000.

[23] Bertalanffy L.. System Theory: Foundations, Development, Applications. George Braziller, Inc., New York, 1968.

[24] Booch G., Maksimchuk R.A., Engel M.W., Young B.J., Conallen J., Houston K. A.: Object-Oriented Analysis and Design with Applications. 3rd edition Addison-Wesley Professional, 2007. ISBN-10: 020189551X

[25] Boucke N., Holvoet T., Lefever T., Sempels R., Schelfthout K., Weyns D., Wielemans J.: Applying the architecture tradeoff analysis method (ATAM) to an industrial multi-agent system application. Report CW 431, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium (Dec. 2005).

[26] Breivold H.P., Crnkovic I., Larsson M.: Software architecture evolution through evolvability analysis. Journal of Systems and Software, Volume 85, Issue 11, November 2012, pp. 2574-2592. Elservier, 2012.

[27] Breivold H.P., Crnkovic I., Eriksson P.J.: Analyzing Software Evolvability. In: 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC '08, pp. 327-330. IEEE 2008.

[28] Breivold H.P., Crnkovic I., Land R., Larsson M.: Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study. The Third International Conference on Software Engineering Advances (ICSEA '08), pp. 205-213. IEEE 2008.

[29] Breivold H.P., Crnkovic I.: A Systematic Review on Architecting for Software Evolvability. 21st Australian Software Engineering Conference (ASWEC), 6-9 April 2010, pp. 13-22, IEEE 2010.

[30] Breivold H.P., Crnkovic I.: Analysis of Software Evolvability in Quality Models. 35th Euromicro Conference on Software Engineering and Advanced Applications, 2009. SEAA '09, pp. 279-282, 27-29 Aug. 2009.

[31] Brooks F.P.: No Silver Bullet – Essence and Accidents of Software Engineering. IEEE Computer 20 (4), pp. 10-19. IEEE 1987.

[32] Buschmann F., Henney K., Schmidt D.C.: Pattern Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, John Wiley&Sons, 2007. ISBN 0470059028.

[33] Buschmann F., Henney K., Schmidt D.C.: Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages, John Wiley&Sons, 2007. ISBN 0471486485.

[34] Buschmann F., Meunier R., Rohnert H., Peter S., Stal M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns. John Wiley&Sons, 1996. ISBN-10: 0471958697

[35] Capilla R., Zimmermann O., Zdun U., Avgeriou P., Küster J.M.: An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle. Proceedings of 5th European Conference on Software Architecture (ECSA 2011). LNCS, vol. 6903, pp. 303-318. Springer-Verlag, 2011.

[36] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering. vol. 20, iss. 6, pp. 476-493. IEEE 1994

[37] Clements P. and Bass L., 2010. The Business Goals Viewpoint. IEEE Software. Nov.-Dec. 2010, vol. 27, No 6, pp. 38-45. IEEE 2010.

[38] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Merson P., Nord R., Stafford J.: Documenting Software Architectures: Views and Beyond (2nd edition). Addison-Wesley Professional, 2010. ISBN 0321552687.

[39] Clements P., Kazman R., and Klein M.: Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002. ISBN 0-201-70482-X.

[40] Clements P.: Active Reviews for Intermediate Designs. CMU/SEI-2000-TN-009. SEI, Carnegie Mellon University, 2000.

[41] Cockburn A.: Agile Software Development: The Cooperative Game. 2nd edition (October 29, 2006). Addison-Wesley Professional, 2006. ISBN-10: 0321482751.

[42] Coplien J., Bjørnvig G.: Lean Architecture: For Agile Software Development. Wiley. Hoboken, NJ, USA, 2010. eISBN: 9780470665039

[43] Cortellessa V., Di Marco A., Eramo R., Pierantonio A., Trubiani C.: Digging into UML models to remove performance antipatterns. In Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (QUOVADIS '10), pp. 9-16. ACM, New York, NY, USA, 2010.

[44] Cortellessa V., Martens A., Reussner R., Trubiani C.: Towards the identification of "Guilty" performance antipatterns. In Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering (WOSP/SIPEW '10), pp. 245-246. ACM, New York, NY, USA, 2010.

[45] de Boer R.C., Lago P., Telea A., Van Vliet H. Ontology-driven visualization of architectural design decisions. Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture (WICSA/ECSA 2009), pp.51,60, IEEE 2009.

[46] de Silva L. and Balasubramaniam D.: Controlling software architecture erosion: A survey, Journal of Systems and Software, Volume 85, Issue 1, January 2012, pp. 132-151. Elservier, 2012

[47] Dijkstra E.W.: Letters to the editor: go to statement considered harmful. Communications of the ACM. Vol. 11, Issue 3 (March 1968), pp. 147-148, ACM 1968.

[48] Dijkstra E.W.: The structure of the \“THE\”-multiprogramming system. Communications of the ACM. Vol. 11, issue 5 (May 1968), pp. 341-346. ACM, 1968

[49] Dobrica L. and Niemelä E.: A survey on software architecture analysis methods. IEEE Transactions on Software Engineering, Jul. 2002, vol. 28, No 7, pp. 638-653. IEEE 2002.

[50] DoDAF 2.02. Department of Defense Architecture Framework (DoDAF), Version 2.0. Available on-line: http://dodcio.defense.gov/dodaf20.aspx. US Department of Defence, 2010.

[51] Dueñas J.C., de Oliveira W.L., de la Puente J.N.: A Software Architecture Evaluation Model. LNCS, vol. 1429, pp. 148-157. Springer-Verlag 1998.

[52] Ethiraj S. and Levinthal D.: Modularity and innovation in complex systems. Management Science, vol. 50, No 2, pp. 159-173, February, 2004.

[53] Falessi, D., Cantone, C., Kazman, R., Kruchten, P.: Decision-making techniques for software architecture design: A comparative survey. ACM Computing Surveys, vol. 43, iss. 4, October 2011.

[54] Ferber S., Heidl, Lutz P.: Reviewing product line architectures: experience report of ATAM in an automotive context. in: Revised Papers from the 4th Int. Workshop on Software Product-Family Engineering (PFE’01). LNCS Volume 2290, pp. 364-382. Springer-Verlag, 2002.

[55] Finkelstein A., Kramer J., Nuseibeh B., Finkelstein L., Goedicke M.: Viewpoints: A framework for integrating multiple perspectives in system development. Int. Journal of Software Engineering and Knowledge Engineering, 2(1): pp. 31-58, World Scientific, 1992.

[56] Folmer E., Bosch J.: Architecting for usability: a survey. Journal of Systems and Software, Volume 70, Issues 1–2, February 2004, pp. 61-78. Elsevier, 2004.

[57] Folmer E., Bosch J.: Case studies on analyzing software architectures for usability. 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005, pp. 206- 213. IEEE 2005.

[58] Folmer E., van Gurp J., Bosch J.: Software Architecture Analysis of Usability. Proc EHCI-DSVIS2004, Springer, LNCS, vol. 3425. Springer-Verlag 2005.

[59] Folmer E.: Software Architecture Analysis of Usability. Ph.D. thesis. Rijksuniversiteit Groningen. PrintPartners Ipskamp, Enschede, 2004. ISBN 90-367-2361-2.

[60] Gacek C., Abd-Allah A., Clark B., Boehm B.: On the Definition of Software System Architecture. Proceedings of the First International Workshop on Architectures for Software Systems. School of Computer Science, Carnegie Mellon University, 1995. [Available as Technical Report USC/CSE-95-TR-500, April 1995].

[61] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994. ISBN-10: 0-201-63361-2.

[62] Garcia J., Popescu D., Edwards G., Medvidovic N.: Identifying architectural bad smells. In Proc. 13th European Conf. on Software Maintenance and Reengineering (CSMR’09), pp. 255-258. IEEE, 2009.

[63] Garcia-Molina, H., Ullman, J.D., Widom, J.D.: Database Systems: The Complete Book. Prentice-Hall, Englewood Cliffs, 2001.

[64] Garlan D., Barnes J.M., Schmerl B., Celiku O.: Evolution styles: Foundations and tool support for software architecture evolution. Proceedings of WICSA/ECSA 2009, pp. 131-140. IEEE 2009.

[65] Harrison, N.B., Avgeriou, P., Zdun, U.: Using Patterns to Capture Architectural Decisions. IEEE Software, vol. 24, iss. 4, pp. 38-45. IEEE 2007

[66] Harrison, N., Avgeriou, P.: Pattern-Based Architecture Reviews, IEEE Software, vol. 28, iss. 6, pp. 66-71. IEEE 2011.

[67] Harrison, R., Counsell, S., Nithi. R.: An Evaluation of the MOOD Set of Object-Oriented Software Metrics. IEEE Transactions on Software Engineering. vol. 24, iss. 6, pp. 491-496. IEEE 1998.

[68] Hatley D.J., Pribhai L.A.: Strategies for real-time systems specifications. Dorset House, New York., 1987.

[69] Hatton S.: Choosing the Right Prioritisation Method. 19th Australian Conference on Software Engineering (ASWEC 2008), pp. 517-526, IEEE 2008.

[70] Hoare C.A.R.: Communicating sequential processes. Communications of ACM, vol. 21, issue 8, pp. 666-677. ACM, 1978.

[71] ISO/IEC 10746-1, 2, 3, 4. Open Distributed Processing – Reference Model: overview, foundations, architecture, architectural semantics. ISO/IEC 1998.

[72] ISO/IEC 19505. Information technology – Object Management Group Unified Modeling Language (OMG UML), Infrastructure, (ISO/IEC 19505-2:2012). Information technology – Object Management Group Unified Modeling Language (OMG UML), Superstructure (ISO/IEC 19505-2:2012).

[73] ISO/IEC 25010:2011. Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models. ISO/IEC 2011.

[74] ISO/IEC 9126-1, Software engineering – product quality – Part 1: Quality Model, first ed.: 2001-06-15.

[75] ISO/IEC/IEEE 24765:2010(E), 2010. Systems and software engineering – Vocabulary.

[76] ISO/IEC/IEEE 42010:2007, IEEE 1471. Systems and software engineering – Recommended practice for architectural description of software-intensive systems. ISO/IEC 2011, IEEE 2007.

[77] ISO/IEC/IEEE 42010:2011. Systems and software engineering – Architecture description. ISO/IEC 2011, IEEE 2011.

[78] Jansen A. and Bosch J.: Software Architecture as a Set of Architectural Design Decisions. Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pp. 109-120. IEEE 2005.

[79] Jarczyk A.P.J., Löffler P., Shipman F.M.: Design Rationale for Software Engineering: A Survey. In Proceedings of the 25th Hawaii International Conference on System Sciences, 1992, Vol. 2, pp. 577-586, IEEE, 1992.

[80] Jensen K.: Coloured Petri nets. Advances in Petri Nets 1986, Part I Proceedings of an Advanced Course Bad Honnef, pp. 248-299. Springer-Verlag, 1986.

[81] Jones L.G., Lattanze A.J.: Using the architecture tradeoff analysis method to evaluate a wargame simulation system: a case study. CMU SEI Technical Report CMU/SEI-2001-TN-022. Software Engineering Institute, Pittsburgh, PA (Dec. 2001).

[82] Kazman R., Asundi J., Klein M.: Quantifying the costs and benefits of architectural decisions. Proceedings of the Twenty-Third International Conference on Software Engineering, pp. 297- 306. IEEE, 2001,

[83] Kazman R., Bass L., Abowd G., Webb M.: SAAM: A Method for Analyzing the Properties of Software Architectures, Proc. 16th Int'l Conf. Software Eng., pp. 81-90, ACM 1994.

[84] Kazman R., Bass L., Klein M., Lattanze T., Northrop L.: A Basis for Analyzing Software Architecture Analysis Methods. Software Quality Journal, vol. 13, pp. 329-355. Springer 2005.

[85] Kazman R., Bass L., Klein M.: The essential components of software architecture design and analysis. Subscribed Journal The Journal of Systems & Software, Vol: 79, Issue: 8, August, 2006, pp. 1207-1216. Elsevier, 2006.

[86] Kazman R., G. Abowd, L. Bass, and P. Clements, 1996. Scenario-Based Analysis of Software Architecture. IEEE Software, pp. 47-55, Nov. 1996.

[87] Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., Carriere J.: The architecture tradeoff analysis method. Fourth IEEE International Conference on Engineering of Complex Computer Systems, 1998. pp. 68-78.

[88] Kircher M., Jain P.: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. John Wiley&Sons, 2004. ISBN 0470845252.

[89] Klein M., Ralya T., Pollak B., Obenza R., Harbour M.G.: Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluver Academic Publishers, 1993. ISBN 0-7923-9361-9.

[90] Koziolek H.: Sustainability evaluation of software architectures: a systematic review. QoSA-ISARCS 2011. ACM SIGSOFT 2011.

[91] Kruchten P., Capilla R., Dueñas J.C.: The Decision View's Role in Software Architecture Practice. IEEE Software, vol.26, No 2, pp.36-42. IEEE 2009.

[92] Kruchten P., Lago P., van Vliet H.: Building Up and Reasoning About Architectural Knowledge. QoSA 2006, LNCS 4214, pp. 43-58. Springer, 2006.

[93] Kruchten P.: An Ontology of Architectural Design Decisions in Software-Intensive Systems. In 2nd Groningen Workshop Software Variability (October 2004), pp. 54-61.

[94] Kruchten P.: The 4+1 View Model of Architecture. IEEE Software, Issue 6, Volume 12, pp. 42-50. IEEE 1995.

[95] Kruchten P.: The Rational Unified Process: An Introduction, Third Edition. Boston, MA: Addison-Wesley, 2004.

[96] Kunz W. and Rittel H. W. J.: Issues as elements of information systems. Working Paper No 131. Studiengruppe für Systemforschung, Heidelberg, Germany, 1970.

[97] Lassing N., Bengtsson P., van Vliet H., Bosch J.: Experiences with ALMA: Architecture-Level Modifiability Analysis, Journal of Systems and Software, Volume 61, Issue 1, pp. 47-57, Elservier, 2002.

[98] Lassing N., Rijsenbrij D., and H. van Vliet: Software Architecture Analysis of Flexibility, Complexity of Changes: Size Isn't Everything. Proc. Second Nordic Software Architecture Workshop (NOSA '99), pp. 1103-1581, 1999.

[99] Leffingwell D. and Widrig D.: Managing software requirements: A Use Case Approach, 2nd ed. Addison-Wesley, Boston, 2003.

[100] Logrippo L., Faci M., Haj-Hussein M.: An introduction to LOTOS: learning by examples. Computer Networks and ISDN Systems, Volume 23, Issue 5, February 1992, pp. 325-342, Elsevier.

[101] Lung C., Bot S., Kalaichelvan K., and Kazman R.: An Approach to Software Architecture Analysis for Evolution and Reusability. Proc. CASCON ‚97. ACM 1997.

[102] Lyu M.R. (editor). Handbook of Software Reliability Engineering. Handbook of Software Reliability Engineering. McGraw-Hill, 1996. ISBN 0070394008. (available on-line http://www.cse.cuhk.edu.hk/~lyu/book/reliability/)

[103] Malan R., Bredemeyer D.: Less is more with minimalist architecture. IT Professional, vol. 4, No 5, pp. 48, 46- 47, IEEE 2002.

[104] Medvidovic N. and Taylor R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, vol. 26, No 1, pp. 70-93, January 2000.

[105] Medvidovic N., Rosenblum D.S., Redmiles D.F., and Robbins J.E.:. Modeling software architectures in the Unified Modeling Language. ACM Trans. Softw. Eng. Methodol. 11, 1 (January 2002).

[106] Molter G.: Integrating SAAM in Domain-Centric and Reuse-Based Development Processes, Proc. Second Nordic Workshop Software Architecture (NOSA '99), pp. 1103-1581, 1999.

[107] Murali Ch.: Mastering Software Quality Assurance: Best Practices, Tools and Technique for Software Developers. J. Ross Publishing Inc., 2011. ISBN 978-1-60427-032-7.

[108] Naur P. and Randell B. (Eds.): Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO, 1969.

[109] Olumofin F.G. and Misic V.B.: A holistic architecture assessment method for software product lines. Journal of Information and Software Technology. Volume 49, Issue 4, pp. 309-323. Elsevier, 2007.

[110] Olumofin F.G. and Misic V.B.: Extending the ATAM Architecture Evaluation to Product Line Architectures", Technical report TR 05/02 Department of computer science, university of Manitoba Winnipeg, Manitoba, Canada R3T 2N2, June 2005.

[111] OMG: Business Process Model and Notation (BPMN). Version 2.0. Object Modeling Group, 2011.

[112] OMG: Systems Modeling Language (OMG SysML™), Version 1.3. Object Modelling Group 2012. (available on-line at http://www.omg.org/spec/SysML/1.3/).

[113] Page-Jones M.: The practical guide to structured system design. Second Edition. Prentice Hall, 1988. ISBN-0136907695.

[114] Parnas D.L., Darringer J.A.: SODAS and a methodology for system design. AFIPS Fall Joint Computing Conference, pp. 449-474. ACM, 1967.

[115] Parnas D.L., Weiss D.M.: Active design reviews: principles and practices. In Proceedings of the 8th international conference on Software engineering (ICSE '85), pp. 132-136. IEEE 1985.

[116] Parnas D.L.: On the Criteria To Be Used in Decomposing Systems into Modules. Commun. ACM 15(12): 1053-1058, 1972.

[117] Peercy D.E.: A Software Maintainability Evaluation Methodology. IEEE Transactions on Software Engineering, vol.SE-7, No 4, pp. 343- 351, July 1981.

[118] Perry D.E. and Wolf A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT. Software Engineering Notes vol. 17 No 4. ACM 1992.

[119] Pooley R.J.; Abdullatif A.A.L.: CPASA: Continuous Performance Assessment of Software Architecture. 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), pp.79-87. IEEE 2010.

[120] Ramachandran P., Adve, S.V., Bose P., Rivers J.A.: Metrics for Architecture-Level Lifetime Reliability Analysis. IEEE International Symposium on Performance Analysis of Systems and software, ISPASS 2008, pp. 202-212. IEEE 2008.

[121] Randell B. and Buxton J.N. (Eds.): Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO, 1970.

[122] Redwine S.T. Jr., Riddle W.E.: Software technology maturation. In Proceedings of the 8th international conference on Software engineering (ICSE ‘85), pp. 189-200. IEEE 1985.

[123] Reijonen V., Koskinen J., and Haikala I.: Experiences from scenario-based architecture evaluations with ATAM. In Proceedings of the 4th European conference on Software architecture (ECSA’10). LNCS, vol. 6285, pp. 214-229. Springer-Verlag 2010.

[124] Riaz M., Mendes E., Tempero E.: A systematic review of software maintainability prediction and metrics. In Proceedings of the 2009 3rd Symposium on Empirical Software Engineering and Measurement (ESEM ’09), pp. 367-377. IEEE 2009.

[125] Roock S. and Lippert M.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. John Wiley & Sons, 2005.

[126] Ross D.T. and Schoman Jr. K.E.: Structured analysis for requirements definition. IEEE Transactions on Software Engineering, SE-3(1), January 1977.

[127] Roy B., Graham T.: Methods for Evaluating Software Architecture: A Survey. Technical Report 545, Queen’s University at Kingston, Ontario, Canada, Kingston, 2008.

[128] Saaty T.L.: Fundamentals of Decision Making and Priority Theory with the Analytic Hierarchy Process. RWS Publications. Pittsburgh, 2000.

[129] Saaty T.L.: The Analytic Hierarchy Process: McGraw-Hill 1980.

[130] Sandler C., Myers G.J., Badgett T.: The Art of Software Testing. Second edition. John Wiley & Sons, 2004. ISBN: 0-471-46912-2.

[131] Schmidt D., Stal M., Rohnert H., Buschmann F.: Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. John Willey & Sons, 2000. ISBN 047160695.

[132] Schulmeyer G.G.: Handbook of Software Quality Assurance, Artech House Norwood, MA, USA, 2007.

[133] Shahin M., Liang P., Khayyambashi M.R.: Improving understandability of architecture design through visualization of architectural design decision. ICSE Workshop on Sharing and Reusing Architectural Knowledge, pp. 88-95. ACM 2010.

[134] Shahin M., Peng Liang, Khayyambashi M.R.: Architectural design decision: Existing models and tools. Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009., pp. 293-296.

[135] Sharafi S.M.: SHADD: A scenario-based approach to software architectural defects detection. Advances in Engineering Software, vol. 45, issue 1, March 2012, pp. 341-348. Elsevier, 2012.

[136] Simon H.A.: The Architecture of Complexity, Proceedings of the American Philosophical Society, vol. 106, No 6., pp. 467-482, Dec. 12, 1962.

[137] Smith C.U. and Williams L.G.: Performance Engineering Evaluation of Object-Oriented Systems with SPE*ED. Proceedings of the 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, pp. 135-154. Springer-Verlag, 1997.

[138] Smith C.U. and Williams L.G.: Software performance antipatterns. In Proceedings of the 2nd international workshop on Software and performance (WOSP ‘00), pp. 127-136. ACM 2000.

[139] Smith D., Merson P.: Using architecture evaluation to prepare a large web based system for evolution. Proc. Fifth IEEE International Workshop on Web Site Evolution, pp. 85-92. IEEE 2003.

[140] Sozer H., Tekinerdogan B., Aksit M.: Extending Failure Modes and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Level. In: Architecting dependable systems IV. LNCS 4615, pp. 409-433. Springer 2007.

[141] Stevens W.P., Myers G.J., Constantine L.L., 1974: Structured Design. IBM Systems Journal, vol. 13, pp. 115-139.

[142] Suntae K., Dae-Kyoo K., Lunjin L., Sooyong P.: Quality-driven architecture development using architectural tactics. Journal of Systems and Software, Volume 82, Issue 8, August 2009, pp. 1211-1231. Elsevier 2009.

[143] Szlenk M., Zalewski A., Kijas S.: Modelling architectural decisions under changing requirements. Proceedings of the Joint 10th Working Conference on Software Architecture & 6th European Conference on Software Architecture, pp. 211-214. IEEE Computer Society (2012).

[144] Tang A., Avgeriou P., Jansen A., Capilla R., and Babar M.A.: A comparative study of architecture knowledge management tools. Journal of Systems and Software. Vol. 83, Issue 3, pp. 352-370. Elsevier, 2010.

[145] Tang A., Kuo F.-C., Lau M.F., 2008. Towards Independent Software Architecture Review. LNCS, Volume 5292, pp. 306-313. Springer-Verlag 2008.

[146] Tekinerdogan B. and Sözer H., 2012: Variability viewpoint for introducing variability in software architecture viewpoints. In Proceedings of the WICSA/ECSA 2012, pp. 163-166. ACM, New York, NY, USA.

[147] Tekinerdogan B., Scholten F., Hofmann Ch., Aksit M.: Concern-oriented analysis and refactoring of software architectures using dependency structure matrices. In Proceedings of the 15th workshop on Early aspects (EA ‘09), pp. 13-18,. ACM, New York, NY, USA, 2009.

[148] Tekinerdogan B., Sozer H., Aksit M.: Software architecture reliability analysis using failure scenarios. Journal of Systems and Software, Volume 81, Issue 4, April 2008, pp. 558-575. Elsevier 2008.

[149] Tekinerdogan B.: ASAAM: aspectual software architecture analysis method. Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), pp. 5-14. IEEE, 2004.

[150] The Open Group: ArchiMate® 2.0 Specification. 2009-2012 The Open Group. Available on-line. The Open Group 2012.

[151] The Open Group: The Open Group Architecture Framework (TOGAF®) Version 9.1. Available on-line at http://pubs.opengroup.org/architecture/togaf9-doc/arch/.

[152] Tyree J., Akerman A.: Architecture Decisions: Demystifying Architecture. IEEE Software, Mar.-Apr. 2005, pp. 19-27.

[153] van Heesch U., Avgeriou P., Hilliard R.: Forces on Architecture Decisions – A Viewpoint. Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 101-110. IEEE 2012.

[154] van Heesch U., Avgeriou P., Hilliard R.: A documentation framework for architecture decisions. Journal of Systems and Software, Volume 85, Issue 4, pp. 795-820. Elsevier, 2012.

[155] Williams L., Smith C.: PASA^{SM}: A Method for the Performance Assessment of Software Architectures. Proceedings of the 3rd international workshop on Software and performance (WOSP ‘02), pp. 179-189. ACM, 2002.

[156] Wirth N., 2008, A Brief History of Software Engineering, Annals of the History of Computing, IEEE, vol. 30, No 3, pp. 32-39, July-Sept. 2008.

[157] Woods E.: Industrial Architectural Assessment Using TARA. 9th Working IEEE/IFIP Conference on Software Architecture (WICSA’11), pp. 56-65. IEEE, 2011.

[158] Woods E.: Industrial architectural assessment using TARA. Journal of Systems and Software, Volume 85, Issue 9, September 2012, pp. 2034-2047. Elsevier 2012.

[159] Yacoub S.M., Ammar H.H.: A methodology for architecture-level reliability risk analysis. IEEE Transactions on Software Engineering, vol. 28, No 6, pp. 529-547. IEEE 2002.

[160] Yen J, Langari R.: Fuzzy logic: intelligence, control, and information. Prentice-Hall, Upper Saddle River, NJ, 1999.

[161] Yourdon E., Constantine, L., 1979. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 1979. ISBN 0138544719.

[162] Yourdon E.: Modern Structured Analysis. Prentice Hall, 1988. ISBN 0135986249. (Polish edition also available)

[163] Zachman J.A., 1987. Framework for Information Systems Architecture. IBM Systems Journal, vol. 26, No 3, pp. 276-292, 1987.

[164] Zadeh L.A.: Fuzzy sets, Information and Control, Volume 8, Issue 3, June 1965, pp. 338-353. Elsevier 1965.

[165] Zalewski A. and Kijas S.: Feature-Based Architecture Reviews, ISAT 2012. Information Systems Architecture and Technology. Networks Design and Analysis. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 2012, ISBN 978-83-7493-702-3. pp. 81-96, 2012.

[166] Zalewski A. and Kijas S.: Towards the Competitive Software Development. 12th Conference on Product-Focused Software Process Improvement, PROFES 2011. LNCS, vol. 6759, pp. 103-112. Springer-Verlag 2011.

[167] Zalewski A., Kijas S., Sokołowska D.: Capturing Architecture Evolution with Maps of Architectural Decisions 2.0. ECSA 2011, Essen, Germany. LNCS, Volume 6903, pp. 83-96. Springer-Verlag 2011.

[168] Zalewski A., Kijas S.: Architecture Decision-Making in Support of Complexity Control. Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010, Proceedings. LNCS, vol. 6285, pp. 501-504. Springer-Verlag 2010.

[169] Zalewski A., Kijas S.: Beyond ATAM: Early Architecture Evaluation Method for Large-Scale Distributed Systems. Journal of Systems and Software. Journal of Systems and Software, Volume 86, Issue 3, March 2013, pp. 683-697. Elsevier 2013.

[170] Zalewski A., Ludzia M.: Diagrammatic Modeling of Architectural Decisions. Software Architecture, Second European Conference, ECSA 2008 Paphos, Cyprus, September 29–1 October, 2008 Proceedings. LNCS, vol. 5292, pp. 350-353. Springer-Verlag 2008.

[171] Zalewski A.: Beyond ATAM: Architecture Analysis in the Development of Large Scale Software Systems. Software Architecture, First European Conference, ECSA 2007 Aranjuez, Spain, September 24-26, 2007 Proceedings. LNCS, vol. 4758, pp. 92-105. Springer-Verlag 2007.

[172] Zimmermann O., Koehler J., Leymann F., Polley R., Schuster N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. Journal of Systems and Software, Volume 82, Issue 8, pp. 1249-1267. Springer 2009.

[173] Zimmermann O., Zdun U., Gschwind T., Leymann F.: Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. WICSA 2008, pp. 157-166, 18-21 Feb. 2008, IEEE 2008.

TABLE OF CONTENTS