Chapter 2

CLASSIFYING AND EVALUATING ARCHITECTURE DESIGN METHODS

Bedir Tekinerdoğan and Mehmet Akşit
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: {bedir, aksit}@cs.utwente.nl,
www: http://trese.cs.utwente.nl

Keywords: Software architectures, architecture design methods, problems in designing architectures

Abstract: The concept of software architecture has gained a wide popularity and is generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems. This chapter first gives a definition of architecture. Second, a meta-model for architecture design methods is presented. This model is used for classifying and evaluating various architecture design approaches. The chapter concludes with the description of the identified problems.

1. INTRODUCTION

Software architectures have gained a wide popularity in the last decade and they are generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems [1]. A common assumption is that architecture design should support the required software system qualities such as robustness, adaptability, reusability and maintainability [2][7]. Software architectures include the early design decisions and embody the overall structure that impacts the quality of the whole system.

For ensuring the quality factors it is generally agreed that, identifying the fundamental abstractions for architecture design is necessary. We maintain

that the existing architecture design approaches have several difficulties in deriving the right architectural abstractions. To analyze, evaluate and identify the basic problems we will present a survey of the state-of-the-art architecture design approaches and describe the obstacles in each approach.

The chapter is organized as follows. Section 2 provides a short background on software architectures in which existing definitions including our own definition of software architecture will be given. In section 3 a meta-model for software architecture design approaches will be given. This meta-model will serve as a basis for identifying the problems in our evaluation of architecture design approaches. In section 4 a classification, analysis and evaluation of the contemporary architectural approaches is presented. Section 5 refers to the related chapters in this volume. Finally, section 6 presents the conclusions and evaluations.

2. NOTION OF ARCHITECTURE

In this section we focus on the meaning of software architecture by analyzing the prevailing definitions as described in section 2.1. In section 2.2 we provide our own definition that we consider as general and which covers the existing definitions.

2.1 Definitions

Software architectures are high-level design representations and facilitate the communication between different stakeholders, enable the effective partitioning and parallel development of the software system, provide a means for directing and evaluation, and finally provide opportunities for reuse [7].

The term architecture is not new and has been used for centuries to denote the physical structure of an artifact [39]. The software engineering community has adopted the term to denote the gross-level structure of software-intensive systems. The importance of structure was already acknowledged early in the history of software engineering. The first software programs were written for numerical calculations using programming languages that supported mathematical expressions and later algorithms and abstract data types. Programs written at that time served mainly one purpose and were relatively simple compared to the current large-scale diverse software systems. Over time due to the increasing complexity and the increasing size of the applications, the global structure of the software system became an important issue [31]. Already in 1968,

Dijkstra proposed the correct arrangement of the structure of software systems before simply programming [15]. He introduced the notion of layered structure in operating systems, in which related programs were grouped into separate layers, communicating with groups of programs in adjacent layers. Later, Parnas maintained that the selected criteria for the decomposition of a system impact the structure of the programs and several design principles must be followed to provide a good structure [25][26]. Within the software engineering community, there is now an increasing consensus that the structure of software systems is important and several design principles must be followed to provide a good structure [12].

In tandem with the increasing popularity of software architecture design many definitions of architecture have been introduced over the last decade, though, a consensus on a standard definition is still not established. We think that the reason why so many and various definitions on software architectures exist is because every author approaches a different perspective of the same concept of software architecture and likewise provides a definition from that perspective. Notwithstanding the numerous definitions it appears that the prevailing definitions do not generally conflict with each other and commonly agree that software architecture represents the gross-level structure of the software system consisting of components and relations among them [7][1].

Looking back at the historical developments of architecture design we can conclude that similar to the many concepts in software engineering the concept of software architecture has also evolved over the years. We observe that this evolution took place at two places. First, existing stable concepts are specialized with new concepts providing a broader interpretation of the concept of software architecture. Second, existing interpretations on software architectures are abstracted and synthesized into new and improved interpretations. Let us explain this considering the development of the definitions in the last decade. The set of existing definitions is large and many other definitions have been collected in various publications such as [35], [28] and [34]. We provide only the definitions that we consider as representative.

> *"The logical and physical structure of a system, forged by all the strategic and tactical design decisions applied during development" [9]*

[1] Compare this to the parable of "the elephant in the dark", in which four persons are in a dark room feeling different parts of an elephant, and all believing that what they feel is the whole beast.

Hereby, software architecture represents a high-level structure of a software system. It is in alignment with the earlier concepts of software architecture as described by Dijkstra [15] and Parnas [26].

> *"We distinguish three different classes of architectural elements: processing elements; data elements; and connection elements. The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together. " [28]*

This definition explicitly considers the interpretation on the elements of software architecture. It is a specialization of the previous architecture definitions and represents the functional aspects of the architecture focusing basically on the data-flow in the system.

> *" ...beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design. " [17]*

This definition provides additional specialization of the structural issues.

> *"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." [18]*

This definition extends the previous definitions by including design information in the architectural specification.

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." [7]*

This definition abstracts from the previous definitions and implies that software architectures have more than one structure and includes the behaviour of the components as part of the architecture. The term component here is used as an abstraction of varying components [7]. This definition may be considered as a sufficiently good representative of the latest abstraction of the concept of software architecture.

2.2 Architecture as a concept

The understanding on the concept of software architecture is increasing though there are still several ambiguities. Architectures consist of components and relations, but the term components may refer to subsystems, processes, software modules, hardware components or something else. Relations may refer to data flows, control flows, call-relations, part-of relations etc. To provide a consistent and overall definition on architectures, we need to provide an abstract yet a sufficiently precise meaning of the components and relations. For this we provide the following definition of architecture:

> *Architecture is a concept representing a set of abstractions and relations and constraints among these abstractions.*

In essence this definition considers architecture as a *concept* that is general yet well defined. We think that this definition is general enough to cover the various perspectives on architectures. To clarify this definition and discuss its implications we will provide a closer view on the notion of concept.

A *concept* is usually defined as a (mental) representation of a category of instances [21] and is formed by abstracting knowledge about instances. The process of assigning new instances to a concept is called *categorization* or *classification*. In this context, concepts are also called *categories* or *classes*. There are several theories on concepts and classification addressing the notions of concepts, classes, instances and categories [24][33][27].

In the context of software architectures the architectural concepts are also abstractions of the corresponding domain knowledge. The content of the domain knowledge, however, may vary per architecture design approach.

Concepts are not just arbitrary abstractions or groupings of a set of instances but are defined by a consensus of experts in the corresponding domain. As such concepts are stable and well-defined abstractions with rich semantics. The definition thus enforces that each architecture consists of components that do not only represent arbitrary groupings or categories but

are semantically well defined. For a more detailed description of architecture as concept, we refer to chapter 3 of [36].

3. META MODEL FOR ARCHITECTURE DESIGN APPROACHES

In this section we provide a meta-model that is an abstraction of various architecture design approaches. We will use this model to analyze and compare current architecture design approaches, which we will describe in the subsequent section.

The meta-model is given in Figure 1. The rounded rectangles represent the concepts and the lines represent the association between these concepts. The diamond symbol represents an association relation between three or four concepts. Let us now describe the concepts individually.

The concept *Client* represents the stakeholder(s) who is/are interested in the development of a software architecture design. A stakeholder may be a customer, end-user, system developer, system maintainer, sales manager etc.

The concept *Domain Knowledge* represents the area of knowledge that is applied in solving a certain problem.

The concept *Requirement Specification* represents the specification that describes the requirements for the architecture to be developed.

The concept *Artifact* represents the artifact descriptions of a certain method. This is, for example, the description of the artifact Class, Operation, Attribute, etc. In general each artifact has a related set of heuristics for identifying the corresponding artifact instances.

The concept *Solution Abstraction* defines the conceptual representation of a (sub)-structure of the architecture.

The concept *Architecture Description* defines a specification of the software architecture.

In Figure 1, there are two quaternary association relations and one ternary association relation.

The quaternary association relation called *Requirements Capturing* defines the association relations between the concepts *Client*, *Domain Knowledge, Requirement Specification* and *Architecture Description*. This association means that for defining a requirement specification the client, the domain knowledge and the (existing) architecture description can be utilized. The order of processing is not defined by this association and may differ per architecture design approach.

The quaternary association relation called *Extracting Solution Structures* is defined between the concepts *Requirement Specification, Domain*

Knowledge, Artifact and *Solution Abstraction*. This describes the structural relations between these concepts to derive a suitable solution abstraction.

The ternary association relation *Architecture Specification* is defined between the concepts *Solution Abstraction*, *Architecture Description* and *Domain Knowledge* and represents the specification of the architecture utilizing these three concepts.

Various architecture design approaches can be described as instantiations of the meta-model in Figure 1. Each approach will differ in the ordering of the processes and the particular content of the concepts.

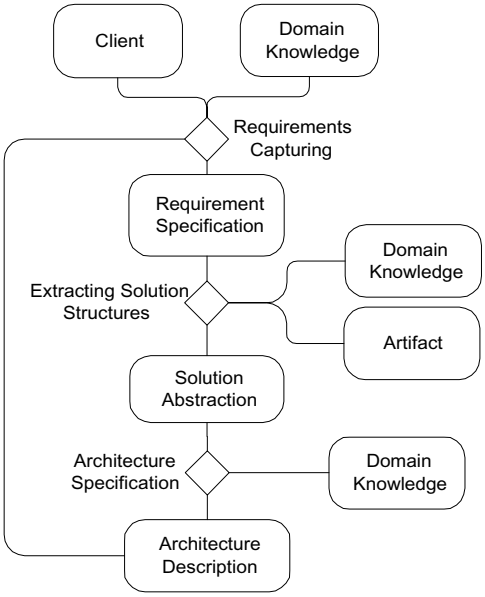

Figure 1: Meta-model for architecture design approaches

In the meta-model, the concept *Domain Knowledge* is used three times. Since this concept plays a fundamental role in various architectural design approaches we will now elaborate on this concept.

The term domain has different meanings in different approaches [36]. We distinguish between the following specialization of this concept: *Problem Domain Knowledge*, *Business Domain Knowledge, Solution Domain Knowledge* and *General Knowledge*.

The concept *Problem Domain Knowledge* refers to the knowledge on the problem from a client's perspective. It includes requirement specification documents, interviews with clients, prototypes delivered by clients etc. The concept *Business Domain Knowledge* refers to the knowledge on the

problem from a business process perspective. It includes knowledge on the business processes and also customer surveys and market analysis reports. The concept *Solution Domain Knowledge* refers to the knowledge that provides the domain concepts for solving the problem and which is separate from specific requirements and the knowledge on how to produce software systems from this solution domain. This kind of domain knowledge is included in for example textbooks, scientific journals, and manuals. The concept *General Knowledge* refers to the general background and experiences of the software engineer and also may include general rules of thumb. Finally, the concept *System/Product Knowledge* refers to the knowledge about a system, a family of systems or a product.

4. ANALYSIS AND EVALUATION OF ARCHITECTURE DESIGN APPROACHES

A number of approaches have been introduced to identify the architectural design abstractions. We classify these approaches as *artifact-driven*, *use-case-driven* and *domain-driven* architecture design approaches. The criterion for this classification is based on the adopted basis for the identification of the key abstractions of architectures. Each approach will be explained as a realization of the meta-model described in Figure 1.

4.1 Artifact-driven Architecture Design

We term artifact-driven architecture design approaches as those approaches that extract the architecture description from the artifact descriptions of the method. Examples of artifact-driven architectural design approaches are the popular object-oriented analysis and design methods such as OMT [30] and OAD [9]. A conceptual model for artifact-driven architectural design is presented in Figure 2. Hereby the labeled arrows represent the process order of the architectural design steps. The concepts *Analysis & Design Models* and *Subsystems* in Figure 2 together represent the concept *Solution Abstraction* of Figure 1. The concept *General Knowledge* represents a specialization of the concept *Knowledge Domain* in Figure 1.

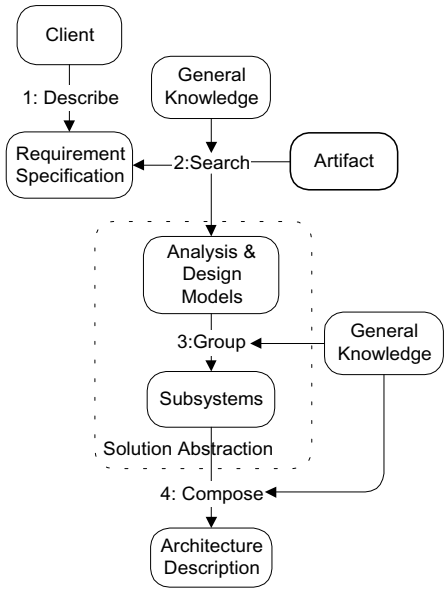

Figure 2: Conceptual model of artifact-driven architectural design

We will explain this model using OMT [30], which can be considered as a suitable representative for this category. In OMT, architecture design is not an explicit phase in the software development process but rather an implicit part of the design phase. The OMT method consists basically of the phases *Analysis, System Design,* and *Object Design*. The arrow *1:Describe* represents the description of the requirement specification. The arrow *2:Search* represents the search for the artifacts such as classes in the requirement specification in the analysis phase. An example of a heuristic rule for identifying tentative class artifacts is the following:

> **IF** an entity in the requirement specification is relevant
> **THEN** select it as a Tentative Class.

The search process is supported by the general knowledge of the software engineer and the heuristic rules of the artifacts that form an important part of the method. The result of the *2:Search* function is a set of artifact instances that is represented by the concept *Analysis &Design Models* in Figure 2.

The method follows with the *System Design* phase that defines the overall architecture for the development of the global structure of a single software system by grouping the artifacts into *subsystems* [30]. In Figure 2, this grouping function is represented by the function *3:Group*. The software

architecture consists of a composition of subsystems, which is defined by the function *4:Compose* in Figure 2. This function is also supported by the concept *General Knowledge.*

4.1.1 Problems

In OMT, the architectural abstractions are represented by grouping classes that are elicited from the requirement specification. We maintain that hereby it is difficult to extract the architectural abstractions. We will explain the problems using the example described in [30] on an Automated Teller Machine (ATM) which concerns the design of a banking network. Hereby, bank computers are connected with ATMs from which clients can withdraw money. In addition, banks can create accounts and money can be transferred and/or withdrawn from one account to another. It is further required that the system should have an appropriate recordkeeping and secure provisions. Concurrent accesses to the same account must be handled correctly.

The problems that we identified with respect to architecture development are as follows:

Textual requirements are imprecise, ambiguous or incomplete and are less useful as a source for deriving architectural abstractions

In OMT, artifacts are searched within the textual requirement specification and grouped into subsystems, which form the architectural components. Textual requirements, however, may be imprecise, ambiguous or incomplete and as such are not suitable as a source for identification of well-defined architectural abstractions. In the example, three subsystems are identified: *ATM Stations, Consortium Computer* and *Bank Computers*. These subsystems group the artifacts that were identified from the requirement specification. With respect to the transaction processing, the example only includes one class artifact called *Transaction* since this was the only artifact that could be discovered in the textual requirement specification. Publications on transaction systems show that many concerns such as scheduling, recovery deadlock management etc. are included in designing transaction systems [16][14][8]. Therefore, we would expect additional classes that could not be identified from the requirement specification.

Subsystems have poor semantics to serve as architectural components

In the given example, the component *ATM stations* represents a subsystem, that is, an architectural component. The subsystem concept

serves basically as a grouping concept and as such has very poor semantics[2]. For the subsystem *ATM stations* it is, for example, not possible to define the architectural properties, architectural constraints with the other subsystems, and the dynamic behavior. This poor semantics of subsystems makes the architecture description less useful as a basis for the subsequent phases of the software development process.

Composition of subsystems is not well-supported

Architectural components interact, coordinate, cooperate and are composed with other architectural components. OMT, however, does not provide sufficient support for this process. In the given example, the subsystem *ATM Stations, Consortium Computer* and *Bank Computers* are composed together, though, the rationale for the presented structuring process is performed implicitly. One could provide several possibilities for composing the subsystems. The method, however, lacks rigid guidelines for composing and specifying the interactions between the subsystems.

4.2 Use-Case driven Architecture Design

In the use-case driven architecture design approach, *use cases* are used as the primary artifacts for deriving the architectural abstractions. A *use case* is defined as a sequence of actions that the system provides for *actors* [22]. Actors represent external roles with which the system must interact. The actors and the use cases together form the *use case model*. The use case model is meant as a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers. The Unified Process [22], for example, applies a use-case driven architecture design approach. The conceptual model for the use-case driven architecture design approach in the Unified Process is given in Figure 3. Hereby, the dashed rounded rectangles represent the concepts of Figure 1. For example the concepts *Informal Specification* and the *Use-Case Model* together form the concept *Requirement Specification* in Figure 1.

The Unified Process consists of *core workflows* that define the static content of the process and describe the process in terms of activities, workers and artifacts. The organization of the process over time is defined by phases. The Unified Process is composed of six core workflows: *Business Modeling*, *Requirements*, *Analysis*, *Design*, *Implementation* and *Test*. These core workflows result respectively in the following separate models:

[2] In [3] this problem has been termed as subsystem-object distinction.

business & domain model, *use-case model*, *analysis model*, *design model*, *implementation model* and *test model*.

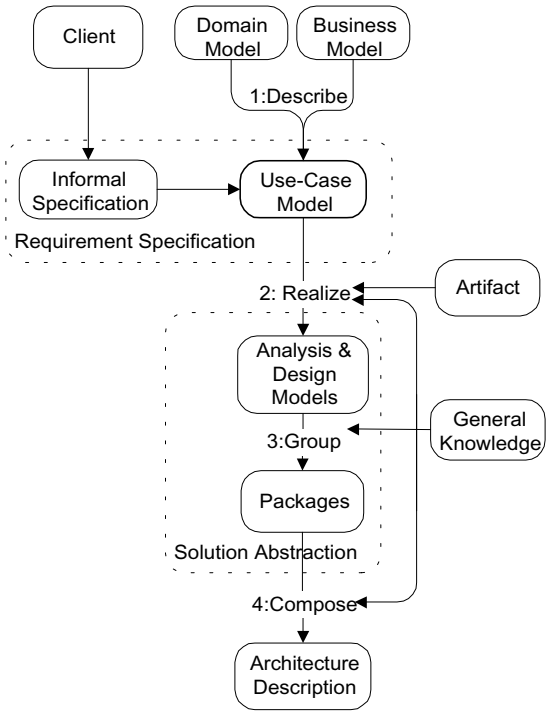

Figure 3: Conceptual model of use-case driven architectural design

In the requirements workflow, the client's requirements are captured as use cases which results in the use-case model. This process is defined by the function *1:Describe* in Figure 3. Together with the informal requirement specification, the use case model forms the requirement specification. The development of the use case model is supported by the concepts *Informal Specification*, *Domain Model* and *Business Model* that are required to define the system's context. The *Informal Specification* represents the textual requirement specification. The *Business Model* describes the business processes of an organization. The *Domain Model* describes the most important classes within the context of the domain. From the use case model the architecturally significant use cases are selected and *use-case realizations* are created as it is described by the function *2:Realize*. Use case realizations determine how the system internally performs the tasks in terms of collaborating objects and as such help to identify the artifacts such as

classes. The use-case realizations are supported by the knowledge on the corresponding artifacts and the general knowledge. This is represented by the arrows directed from the concepts *Artifact* and *General Knowledge* respectively, to the function *2:Realize*. The output of this function is the concept *Analysis & Design Models*, which represents the identified artifacts after use-case realizations.

The analysis and design models are then grouped into *packages* which is represented by the function *3:Group*. The function *4:Compose* represents the definition of interfaces between these packages resulting in the concept *Architecture Description*. Both functions are supported by the concept *General Knowledge*.

4.2.1 Problems

In the Unified Process, first the business model and the domain model are developed for understanding the context. Use case models are then basically derived from the informal specification, the business model and the domain model. The architectural abstractions are derived from realizations of selected use cases from the use case models.

We think that this approach has to cope with several problems in identifying the architectural abstractions. We will motivate our statements using the example described in [22, pp. 113] that concerns the design of an electronic banking system in which the internet will be used for trading of goods and services and likewise include sending orders, invoices, and payments between sellers and buyers. The problems that we encountered are listed as follows:

Leveraging detail of domain model and business model is difficult

The business model and domain models are defined before the use case model. The question raises then how to leverage the detail of these models. Before use cases are known it is very difficult to answer this question since use cases actually define what needs to be developed. In [22, pp. 120] a domain model is given for an electronic banking system example. Domain models are derived from domain experts and informal requirement specifications. The resulting domain model includes four classes: *Order*, *Invoice*, *Item* and *Account*. The question here is whether these are the only important classes in electronic banking systems. Should we consider also the

classes such as *Buyer* and *Seller*? The approach does not provide sufficient means for defining the right detail of the domain and business models[3].

Selecting architecturally relevant use-cases is not systematically supported

For the architecture description, 'architecturally relevant' use cases are selected. The decision on which use cases are relevant lacks objective criteria and is merely dependent on some heuristics and the evaluation of the software engineer. For example, in the given banking system example, the use case *Withdraw Money* has been implicitly selected as architecturally relevant and other use cases such as *Deposit Money* and *Transfer between Accounts* have been left out.

Use-cases do not provide a solid basis for architectural abstractions

After the relevant use cases have been selected they are *realized* which means that analysis and design classes are identified from the use cases. Use-case realizations are supported by the heuristic rules of the artifacts, such as classes, and the general knowledge of the software engineer. This is similar to the artifact-driven approach in which artifacts are discovered in the textual requirements. Although use cases are practical for understanding and representing the requirements, we maintain that they do not provide a solid basis for deriving architectural design abstractions. Use cases focus on the problem domain and the external behavior of the system. During use case realizations, transparent or hidden abstractions that are present in the solution domain and the internal system may be difficult to identify. Thus even if all the relevant use cases have been identified it may still be difficult to identify the architectural abstractions from the use case model. In the given banking system example, the use case-realization of *Withdraw Money* results in the identification of the four analysis classes *Dispenser*, *Cashier Interface*, *Withdrawal* and *Account* [22, pp. 44]. The question here is whether these are all the classes that are concerned with withdrawal. For example, should we also consider classes such as *Card* and *Card Check*? The transparent classes cannot be identified easily if they have not been described in the use case descriptions.

[3] Use cases focus on the functionality for each user of the system rather than just a set of functions that might be good to have. In that sense, use cases form a practical aid for leveraging the requirements. They are however less practical for leveraging the domain and business models.

Package construct has poor semantics to serve as an architectural component

The analysis and design models are grouped into package constructs. Packages are, similar to subsystems in the artifact-driven approach, basically grouping mechanisms and as such have poor semantics. The grouping of analysis and design classes into packages and the composition of the packages into the final architecture are also not well supported and are basically dependent on the general knowledge of the software engineer. This may again lead to ill-defined boundaries of the architectural abstractions and their interactions.

Applied heuristics are implicitly based on the structural paradigm and hinder to identify and define the fundamental abstractions of current large scale and diverse applications.

A close look at the Unified Process results in the interesting observation that the heuristics that are applied to identify the abstractions, which are needed to define the architectural components are essentially based on the traditional structural paradigm of software development. In this paradigm, data is processed by a set of functions resulting in some output data. The basic abstractions that define the architectural components in the Unified Process are the classes and packages. Classes are derived from the use case model by searching for entities that are needed for interfacing, control and information. Packages are derived from the problem domain and use cases that support specific business processes, require specific actors or use cases that are related via generalizations and extends-relations. Both in class identification and package identification, actually functional entities are searched that get some input data, process these and result some output data. This bias from the early period of software engineering, which largely dealt with defining systems for numerical applications, is not suitable anymore for identifying and defining the architectural abstractions of current large-scale, diverse systems.

4.3 Domain-driven Architecture Design

Domain-driven architecture design approaches derive the architectural design abstractions from domain models. The conceptual model for this domain-driven approach is presented in Figure 4.

Domain models are developed through a domain analysis phase represented by the function *2:Domain Analysis*. Domain analysis can be

defined as the process of identifying, capturing and organizing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [29]. The function *2:Domain Analysis* takes as input the concepts *Requirement Specification* and *Domain Knowledge* and results in the concept *Domain Model.* Note that both the concepts *Solution Domain Knowledge* and *Domain Model* in Figure 4 represent the concept *Domain Knowledge* in the meta-model of Figure 1.

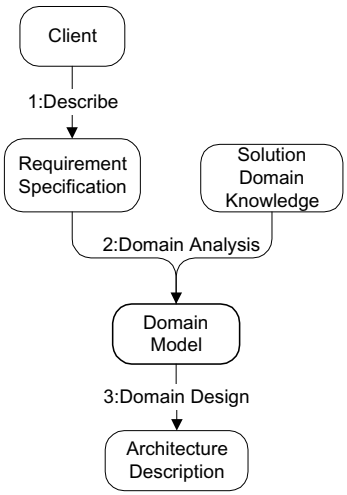

Figure 4: Conceptual model for Domain-Driven Architecture Design

The domain model may be represented using different representation forms such as classes, entity-relation diagrams, frames, semantics networks, and rules. Several *domain analysis* methods have been published, e.g. [19], [23], [29], [32] and [13]. Two surveys of various domain analysis methods can be found in [4] and [40]. In [13] a more recent and extensive up-to-date overview of domain engineering methods is provided.

In this chapter we are mainly interested in the approaches that use the domain model to derive architectural abstractions. In Figure 4, this is represented by the function *3:Domain Design.* In the following we will consider two domain-driven approaches that derive the architectural design abstractions from domain models.

4.3.1 Product-line Architecture Design

In the product-line architecture design approach, an architecture is developed for a *software product-line* that is defined as a group of software-

intensive products sharing a common, managed set of features that satisfy the needs of a selected market or mission area [11]. A *software product line architecture* is an abstraction of the architecture of a related set of products. The product-line architecture design approach focuses primarily on the reuse within an organization and consists basically of *the core asset development* and *the product development*. The core asset base often includes the architecture, reusable software components, requirements, documentation and specification, performance models, schedules, budgets, and test plans and cases [5], [6], [11]. The core asset base is used to generate or integrate products from a product line.

The conceptual model for product-line architecture design is given in Figure 5 The function *1:Domain Engineering* represents the core asset base development. The function *2:Application Engineering* represents the product development from the core asset base.

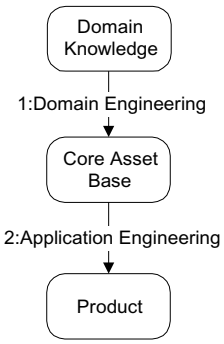

Figure 5: A conceptual model for a Product-Line Architecture Design

Note that various software architecture design approaches can be applied to provide a product-line architecture design. In the following section we will describe an approach that follows the conceptual model for product-line architecture design in Figure 5.

4.3.2 Domain Specific Software Architecture Design

The *domain-specific software architecture* (DSSA) [20][38] may be considered as a multi-system scope architecture, that is, it derives an architectural description for a family of systems rather than a single-system. The conceptual model of this approach is presented in Figure 6.

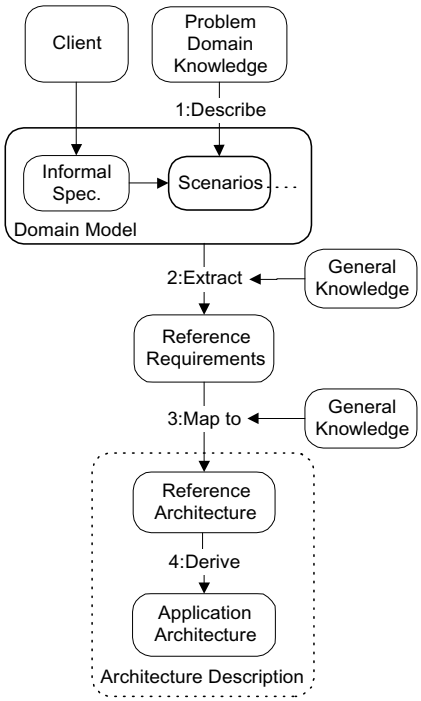

Figure 6: Conceptual model for Domain Specific Software Architecture (DSSA) approach

The basic artifacts of a DSSA approach are the *domain model*, *reference requirements* and the *reference architecture*. The DSSA approach starts with a domain analysis phase on a set of applications with common problems or functions. The analysis is based on *scenarios* from which functional requirements, data flow and control flow information is derived. The *domain model* includes scenarios, domain dictionary, context (block) diagrams, ER diagrams, data flow models, state transition diagrams and object models.

In addition to the domain model, *reference requirements* are defined that include functional requirements, non-functional requirements, design requirements and implementation requirements and focus on the solution space. The domain model and the reference requirements are used to derive the *reference architecture*. The DSSA process makes an explicit distinction between a *reference architecture* and an *application architecture*. A reference architecture is defined as the architecture for a family of application systems, whereas an application architecture is defined as the architecture for a single system. The application architecture is instantiated or refined from the reference architecture. The process of

instantiating/refining and/or extending a reference architecture is called *application engineering*.

4.3.3 Problems

Since the term domain is interpreted differently there are various domain-driven architecture design approaches. We list the problems for problem domain analysis and solution domain analysis.

- *Problem domain analysis is less effective in deriving architectural abstractions*

Several domain-driven architecture approaches interpret the domain as a problem domain. The DSSA approach, for example, starts from an informal problem statement and derives the architectural abstractions from the domain model that is based on scenarios. Like use cases, scenarios focus on the problem domain and the external behavior of the system. We think that approaches that derive abstractions from the problem domain, such as the DSSA approach, are less effective in deriving the right architectural abstractions. Let us explain this using the example in [37] in which an architecture for a theater ticket sales application is constructed using the DSSA approach. In this example a number of scenarios such as *Ticket Purchase*, *Ticket Return, Ticket Exchange*, *Ticket Sales Analysis*, and *Theater Configuration* are described and accordingly a domain model is defined based on these scenarios. The question hereby is whether the given scenarios fully describe the system and as such result in the right leverage of the domain model. Are all the important abstractions identified? Do there exist redundant abstractions? How can this be evaluated? Within this approach and generally approaches that derive the abstractions from the problem domain these questions remain rather unanswered.

- *Solution Domain Analysis is not sufficient*

There exist solution domain analysis approaches that are independent of software architecture design which provide systematic processes for identifying potentially reusable assets. As we have described before, this activity is called *domain engineering* in the systematic reuse community. Unlike system engineering and problem domain engineering, solution domain analysis looks beyond a single system, a family of systems or the problem domain to identify the reusable assets within the solution domain itself. Although solution domain analysis provides the potential for modeling the whole domain that is necessary to derive the architecture, it is not sufficient to drive the architecture design process. This is due to two reasons.

First, solution domain analysis is not defined for software architecture design per se, but rather for systematic reuse of assets for activities in, for example, software development. Since the area on which solution domain analysis is performed may be very wide, it may easily result in a domain model that is too large and includes abstractions that are not necessary for the corresponding software architecture construction. The large size of the domain model may hinder the search for the architectural abstractions. The second problem is that the solution domain may not be sufficiently cohesive and stable to provide a solid basis for architectural design. Concepts in the corresponding solution domain may not have reached a consensus yet and still be under development. Obviously, one cannot expect to provide an architecture design solution that is better than the solution provided by the solution domain itself. Therefore, a thorough solution domain analysis may in this case also not be sufficient to provide stable abstractions since the concepts in the solution domain themselves are fluctuating.

5. OVERVIEW OF THE ARCHITECTURE DESIGN APPROACHES IN THIS BOOK

The chapters in part 2 of this book are devoted to the architectural issues in software development. Chapters 5, 6 and 7 present architecture design methods. Chapter 8 proposes a set of basic abstractions to design various kinds of message-based architectures. Chapters 9 and 10 describe means to refine architectures into object-oriented software systems. In the following, we give a brief summary of these chapters from the perspective of architecture design.

Chapter 5 emphasizes the importance of the so-called non-functional properties in architecture design. By using a car navigation system design example, the chapter illustrates how the functional and non-functional requirements can be considered in a uniform manner. The functional design of the architecture is use-case driven, and therefore, confirms to the design model shown in Figure 3. The non-functional part of the design is, however, solution-domain driven and can be considered as an instantiation of the design model shown in Figure 4. For example, deadline and schedulability solution-domain techniques are used to analyze and design the non-functional characteristics of the architecture.

In chapter 6 by the help of two industrial design problems, a product-line architecture design method is presented. This is a domain-driven architecture design approach, which confirms the design models shown in figures 4 and

5. The architectural abstractions are derived by using both use-cases and solution domain abstractions.

In chapter 7 a synthesis-based architecture design method is presented. In this method, the functional requirements, which may be derived from the use cases, are first expressed in terms of technical problems. These problems are then synthesized towards solutions by systematically applying solution domain knowledge. This approach can be considered as a specialization of the domain-driven architecture design method shown in Figure 4.

In chapter 8 first various message-oriented interaction styles are analyzed. A set of basic abstractions is derived from the solution domains such message-oriented interaction models, delivery semantics and reliability concerns. Expressiveness of these abstractions are motivated by a number of examples.

Chapter 9 proposes a logic meta-programming language to capture and preserve the architectural constraints along the refinement process.

Chapter 10 proposes a technique called Design Algebra to analyze and refine various architecture implementation alternatives by using quality factors such as adaptability and performance.

6. CONCLUSION

In this chapter we have defined architecture as a set of abstractions and relations that form together a concept. Further, a meta-model that is an abstraction of software architecture design approaches is provided. We have used this model to analyze, compare and evaluate architecture design approaches. These approaches have been classified as *artifact-driven*, *use-case-driven* and *domain-driven* architecture design approaches. The criterion for this classification is based on the adopted basis for the identification of the key abstractions of architectures. In the *artifact-driven* approaches the architectural abstractions are represented by groupings of artifacts that are elicited from the requirement specification. *Use-case driven* approaches derive the architectural abstractions from use case models that represents the system's intended functions. *Domain-driven* architecture design approaches derive the architectural abstractions from the domain models. For each approach, we have described the corresponding problems and motivated why these sources are not optimal in identifying the architectural abstractions. We can abstract the problems basically as follows:

1. Difficulties in Planning the Architectural Design Phase

Planning the architecture design phase in the software development process is a dilemma[4]. In general architectures are identified before or after the analysis and design phases. Defining the architecture can be done more accurately after the analysis and design models have been determined because these impact the boundaries of the architecture. This may lead, however, to an unmanageable project because the architectural perspective in the software development process will be largely missing. On the other hand, planning the architecture design phase before the analysis and design phases may also be problematic since the architecture may not have optimal boundaries due to insufficient knowledge on the analysis and design models[5].

In artifact-driven architecture design approaches the architecture phase follows after the analysis and design phases and as such the project may become unmanageable. In the domain-driven architecture design approaches the architecture design phase follows a domain engineering phase in which first a domain model is defined from which consequently architectural abstractions are extracted. Hereby the architecture definition may be unmanageable if the domain model is too large. In the use-case driven architecture design approach the architecture definition phase is part of the analysis and design phase and the architecture is developed in an iterative way. This does not completely solve the dilemma since the iterating process is mainly controlled by the intuition of the software engineer.

2. Client requirements are not a solid basis for architectural abstractions

The client requirements on the software-intensive system that needs to be developed is different from the architectural perspective. The client requirements provide a problem perspective of the system whereas the architecture is aimed to provide a solution perspective that can be used to realize the system. Due to the large gap between the two perspectives the architectural abstractions may not be directly obvious from the client requirements. Moreover, the requirements themselves may be described inaccurately and may be either under-specified or over-specified. Therefore, sometimes it is also not preferable to adopt the client requirements.

This problem is apparent in all the approaches that we analyzed. In the artifact-driven approach the client requirements are directly used as a source

[4] In [3] this problem has been denoted as the *early decomposition* problem

[5] An analogy of this problem is writing an introduction to a book. To organize and manage the work on the different chapters it is required to provide a structure of the chapters in advance. However, the final structure of the introduction can be usually only defined after the chapters have been written and the complete information on the structure is available.

for identifying the architectural abstractions. The use-case driven approach attempts to model the requirements also from a client perspective by utilizing use case models. In the domain-driven approaches, such as the domain specific software architecture design approach (DSSA), informal specifications are used to support the development of scenarios that are utilized to develop domain models.

3. Leveraging the domain model is difficult

The domain-driven and the use case approaches apply domain models for the construction of software architecture. Uncontrolled domain engineering may result in domain models that lack the right detail of abstraction to be of practical use. The one extreme of the problem is that the domain model is too large and includes redundant abstractions, the other extreme is that it is too small and misses the fundamental abstractions. Domain models may also include both redundant abstractions and still miss some other fundamental abstractions. It may be very difficult to leverage the detail of the domain model.

This problem is apparent in domain-driven and the use-case driven approaches. In the domain-driven approaches that derive domain models from problem domains, such as the DSSA approach, leveraging the domain model is difficult because it is based on scenarios that focus on the system from a problem perspective rather than a solution perspective. In the use-case driven architecture design approach, leveraging the domain model and business model is difficult since it is performed before use-case modeling and it is actually not exactly known what is desired.

4. Architectural abstractions have poor semantics

A software architecture is composed of architectural components and architectural relations among them. Often architectural components are similar to groupings of artifacts, which are named as subsystems, packages etc. These constructs do not have sufficiently rich semantics to serve as architectural components. Architectural abstractions should be more than grouping mechanisms and the nature of the components and their relations, and the architectural properties, the behavior of the system should be described as well [10]. Because of the lack of semantics of architectural components it is very hard to understand the architectural perspective and make the transition to the subsequent analysis and design models.

5. Composing architectural abstractions is weakly supported

Architectural components interact, coordinate, cooperate and are composed with other architectural components. The architecture design

approaches that we evaluated do not provide, however, explicit support for composing architectural abstractions.

Acknowledgements

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, The Dutch Ministry of Economical affairs under the SENTER program, The Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project) and the AMIDST project. The participants of the AMIDST project are the Centre for Telematics and Information Technology institute, the Telematics institute and KPN Research.

References

1. G. Abowd, L. Bass, R. Kazman and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*, CA: IEEE Computer Society Press, pp. 81-90, May, 1994.
2. M. Akşit, K. Berg van den, L. Bergmans, P. Broek van den, A. Rensink and B. Tekinerdoğan, B. *Towards Quality-Oriented Software Engineering.* Chapter 1 in Software Architectures and Component Technology: The State of the Art in Research and Practice, M. Aksit (Ed.), Kluwer Academic Publishers, 2000.
3. M. Akşit and L. Bergmans. Obstacles in Object-Oriented Software Development. In *Proceedings OOPSLA '92*, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.
4. G. Arrango. Domain Analysis Methods. In *Software Reusability,* Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.
5. L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey. *Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
6. L. Bass, P. Clements, G. Chastek, S. Cohen, L. Northrop and J. Withey. *2nd Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
7. L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice*, Addison-Wesley 1998.
8. P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*, Morgan Kaufman Publishers, 1997.

9. G. Booch. *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc, 1991.

10. P. Clements. A Survey of Architectural Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, Paderborn, Germany, March, 1996.

11. P.C. Clements and L.M. Northrop. *Software Architecture: An Executive Overview*, Technical Report, CMU/SEI-96-TR-003, Carnegie Mellon University, 1996.

12. P. Clements, D. Parnas and D. Weiss. *The Modular Structure of Complex Systems*. IEEE Transactions on Software Engineering SE-11, 1, pp. 259-266, 1985.

13. C. Czarnecki. *Generative Programming*: *Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, PhD Thesis, Technical University of Ilmenau, 1999.

14. C.J. Date. *An Introduction to Database Systems*, Vol. 3, Addison Wesley, 1990.

15. E.W. Dijkstra. *The Structure of the`T.H.E.' Mulitprogramming System.* Communications of the ACM 18, 8 , pp. 453-457, 1968.

16. A.K. Elmagarmid (ed.) *Transaction Management in Database Systems*, Morgan Kaufmann Publishers, 1991.

17. D. Garlan and M. Shaw. *An Introduction to Software Architecture*. Advances in: Software Engineering and Knowledge Engineering. Vol 1. River Edge, NJ: World Scientific Publishing Company, 1993.

18. D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch: Why It's Hard to Build Systems Out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*. Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, pp. 170-185. , 1995.

19. H. Gomaa. *An Object-Oriented Domain Analysis and Modeling Method for Software Reuse.* In *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, January, 1992.

20. F. Hayes-Roth. Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program, 1994.

21. R.W. Howard. *Concepts and Schemata*: *An Introduction*, Cassel Education*,* 1987.

22. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

23. K. Kang, S. Cohen, J. Hess, W. Nowak and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

24. G. Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*, The University of Chicago Press, 1987.

25. D. Parnas. *On the Criteria for Decomposing Systems into Modules.* Communications of the ACM 15, 12 (December 1972): 1053-1058.
26. Parnas, D. *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering SE-2, 1: 1-9, 1976.
27. J. Parsons and Y. Wand. *Choosing Classes in Conceptual Modeling*, Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997
28. D.E. Perry and A.L. Wolf. *Foundations for the Study of Software Architecture.* Software Engineering Notes, ACM SIGSOFT 17, 4: 40-52, October 1992.
29. R. Prieto-Diaz and G. Arrango (Eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991.
30. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
31. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*, Englewood Cliffs, NJ: Prentice-Hall, 1996.
32. M. Simos, D. Creps, C. Klinger, L. Levine and D. Allemang. *Organization Domain Modeling (ODM) Guidebook*, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, http://www.synquiry.com, 1996.
33. E.E. Smith and D.L. Medin. *Categories and Concepts*, Harvard University Press, London, 1981.
34. Software Engineering Institute, Carnegie Mellon university, Web-site: http://www.sei.cmu.edu/architecture/, 2000.
35. D. Soni, R. Nord and C. Hofmeister. *Software Architecture in Industrial Applications*. 196-210. Proceedings of the 17th International ACM Conference on Software Engineering, Seattle, WA, 1995.
36. B. Tekinerdoğan. *Synthesis-Based Software Architecture Design*, PhD Thesis, Dept. Of Computer Science, University of Twente, March 23, 2000.
37. W. Tracz. *DSSA (Domain-Specific Software Architecture) Pedagogical Example*. In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, July 1995.
38. W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines. Technical Report*. ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.
39. Merriam Webster on-line Dictionary, http://www.m-w.com/cgi-bin/dictionary, 2000.
40. S. Wartik and R. Prieto-Díaz. *Criteria for Comparing Domain Analysis Approaches*. In International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, pp. 403-431, September 1992.