Chapter 3

Classification and Evaluation of Software Architecture Design Approaches

3.1 Introduction

Since in order to speak, one must first listen, learn to speak by listening.
- Rumi

In the last decade 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 [Clements & Northrop 96]. A common assumption is that architecture design should support the required software system qualities such as robustness, adaptability, reusability and maintainability [Aksit et al. 00][Bass et al. 98]. Software architectures include the early design decisions and embody the overall structure that impacts the quality of the whole system. In the literature no consensus is reached yet for software architecture terminology, representations and architecture design approaches [Clements & Northrop 96] and several open problems have still to be solved. In this chapter we will focus on software architecture design approaches. 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 motivate the obstacles in each approach.

The chapter is organized as follows. Section 3.2 provides a short background on software architectures in which existing definitions including our own definition of software architecture will be given. In section 3.3 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 3.4 a classification, analysis and evaluation of the contemporary architectural approaches is presented. Finally, section 3.5 presents the conclusions and evaluations.

3.2 Notion of Software Architecture

In this section we focus on the meaning of software architecture by analyzing the prevailing definitions in section 3.2.1. In section 3.2.2 we provide our own definition that we consider as general and covers the existing definitions.

3.2.1 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 [Bass et al. 98].

The term architecture is not new and has been used for centuries to denote the physical structure of an artifact [Webster 00]. 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 size of the applications, the global structure of the software system became an important issue [Shaw & Garlan 96]. Already in 1968, Dijkstra proposed the correct arrangement of the structure of software systems before simply programming [Dijkstra 68]. 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 [Parnas 72][Parnas 76]. 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 [Clements et al. 85].

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 [Bass et al. 98][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 fronts. 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

[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.

been collected in various publications such as [Soni et al. 95], [Perry & Wolf 92] and [SEI 00]. We provide only the definitions that we consider as representative.

[Booch 91]:
The logical and physical structure of a system, forged by all the strategic and tactical design decisions applied during development

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 [Dijkstra 68] and Parnas [Parnas76]. The first variations of structure of architectures start to appear.

[Perry & Wolf 92]:
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.

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.

[Garlan & Shaw 93]:

...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.

This definition provides additional specializations of the structural issues.

[Garlan et al. 95]:
The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

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

[Bass et al. 98]:
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.

This definition abstracts from the previous definitions and implies that software architectures have more than one structure and includes the behavior of the components as part of the architecture. The term component here is used as an abstraction of varying components [Bass et al. 98]. This definition may be considered as a sufficiently good representative of the latest abstraction of the concept of software architecture.

3.2.2 Architecture as a concept

The understanding on the concept of software architecture is increasing though there are still several shortcomings. 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 [Howard 87] 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 [Lakoff 87][Smith & Medin 81][Parsons & Wand 97].

In the context of software architectures the architectural concepts are also abstractions of domain knowledge sources. The content of the domain sources, however, may vary per architecture design approach. We will elaborate on this topic in the following sections.

In general, three different views on the notion of concept are distinguished: the *classical view*, the *prototype view* and the *exemplar view*. The *classical view*[2] holds that all instances of the concept must share all the *defining* properties that are considered *necessary* and *sufficient* to define the concept. In other words, an instance must have all of the defining properties to be an instance of the concept and additionally, if an instance has at least the defining properties it is sufficient to denote it as an instance

[2] The classical view dates back to the philosophical works of Plato and Aristotle. Plato defined the notion of *forms*, which were defined as stable, immutable and ideal descriptions of things. Aristotle continued the research on classification and his work led to the classical view on categorization and concepts. [Lakoff 87]

of the concept. In the *prototype view*[3] the concept is not described by *defining* properties but rather by *characterizing* properties, features that instances tend to have but need not to have. Basically the view proposes that a concept should be represented by some measure of central tendency of some instances, which is described by a *prototype*. A prototype is defined as an instance that has all the properties of the central tendency and as such is a highly typical instance or idealization. The *exemplar view* of concepts is quite different from the classical and the prototype view since hereby a concept does not represent an abstracted set of defining features or as a measure of a central tendency. The theory does not require abstraction of instances at all. Instead concepts are represented through *exemplars*. An exemplar is a specific instance of a certain category, which is used to represent the category.

Given the different views on concepts the question here is then which of the view of concepts is suitable for architectures. Basically, each view has its advantages and disadvantages and can be applied for solving a particular category of problems [Stillings et al. 95]. The classical view can be best applied for representing well-defined concepts. The prototype view and exemplar view on the contrary can be best applied in the early phases of concept formation in which specific instances are discovered first and are later generalized. Accordingly, we may apply the prototypical and exemplar view in the early phases of architecture design and the classical view may be applied to define the stable architectural abstractions at later stages of the architecture design in which the knowledge on instances and concepts has got mature.

The definition has also implications for the structuring of concepts in software architectures. Two widely known structures are *taxonomies* and *partonomies* [Howard 87]. Taxonomies are usually represented by tree-like structures whereby the top-level concepts include the lower level concepts. Each taxonomy has both a horizontal and a vertical dimension. The vertical dimension represents the level of abstraction and the horizontal dimension represents mutually exclusive categories at the same abstraction level. A particular abstraction level that is called the *basic-level* defines the useful abstractions [Rosch et al. 76]. *Partonomies* define structures in which concepts are related to each other through part-whole relations rather than class inclusion. Taxonomies and partonomies are the basic well-known structures, however, other structuring mechanisms that include various relations between concepts, such as associations, may also be applied.

Since an architecture is a structure of concepts and each concept may represent structures themselves, the definition implies that an architecture may have different structures. In the simple case the architecture consist of a set of concepts that can be considered as 'atomic' and do not have an internal structure. For large software systems, however, it is necessary to define the architecture from various

[3] The prototypical view has emerged from the philosophical treatments of Wittgenstein who maintained that for most concepts meaning is determined not by definition but by family resemblance [Wittgenstein 53].

perspectives such as, for example, from a logical view, the process view, the development view and the physical view [Kruchten 95]. Therefore, at the highest abstraction level the software architecture may consist of concepts that each represents different architectural views.

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.

3.3 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.

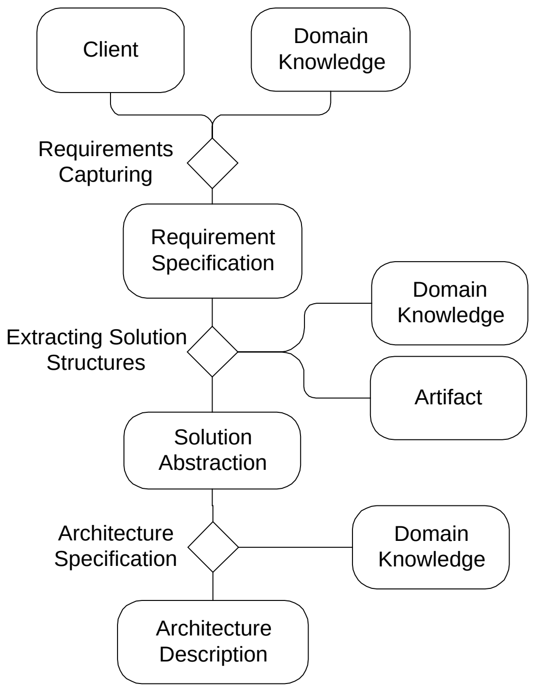

Figure 1. Meta-model for architecture design approaches

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. We will elaborate on this concept in the next sub-section.

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

In the Figure there is a ternary association relation between the concepts *Client*, *Domain Knowledge* and *Requirement Specification*. This association means that for defining a requirement specification both client and the domain knowledge are utilized. The order of processing is not defined by this association and may differ per architecture design approach.

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.

There is a quaternary association relation 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.

Finally, the concept *Architecture Description* defines a specification of the software architecture. There is a ternary association relation between the concept *Solution Abstraction*, *Architecture Description* and *Domain Knowledge*. The association relation is named *Architecture Specify* and represents the specification of the architecture utilizing the 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.

3.3.1 Domain Knowledge

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. We distinguish between the following specializations of this concept: *Problem Domain Knowledge*, *Business Domain Knowledge, Solution Domain Knowledge* and *General Knowledge*. This classification of domain knowledge concepts is shown in Figure 2.

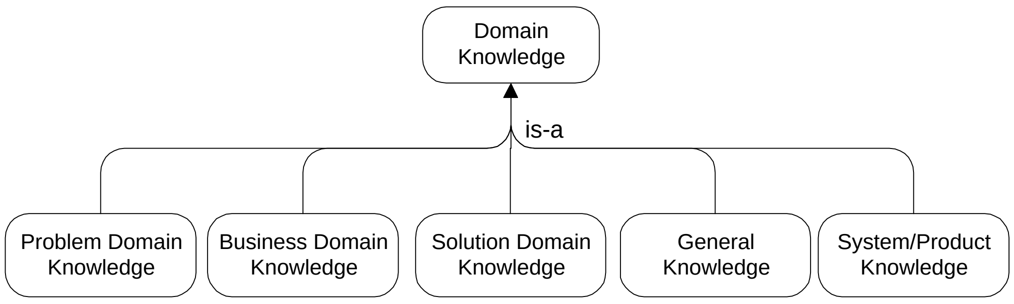

Figure 2. Different specializations of the concept *Domain Knowledge*

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.

The concept *System/Product Knowledge* refers to the knowledge on a system, a family of systems or a product.

3.4 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*, *domain-driven* and *pattern-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.

3.4.1 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 [Rumbaugh et al. 91] and OAD [Booch 91]. A conceptual model for artifact-driven architectural design is presented in Figure 3. Hereby the labeled arrows represent the process order of the architectural design steps. The concepts *Analysis & Design Models* and *Subsystems* in Figure 3 together represent the concept *Solution Abstraction* of Figure 1. The concept *General Knowledge* represents a specialization of the concept *Knowledge Domain* in Figure 1.

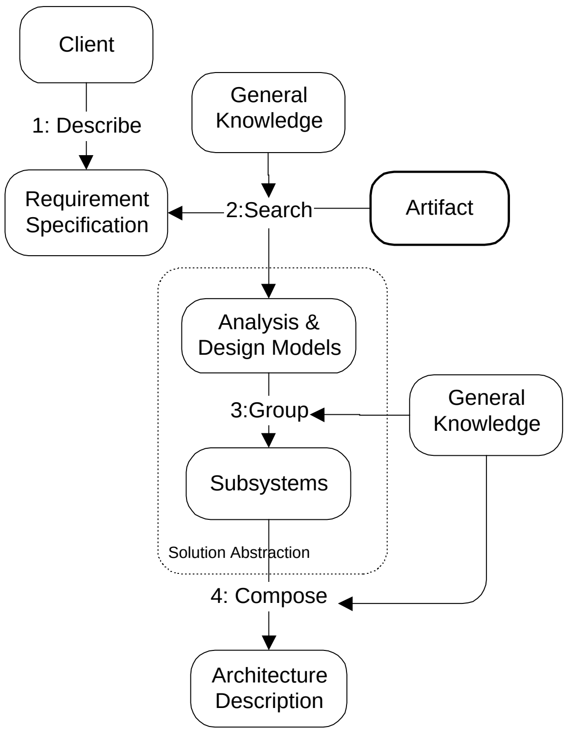

Figure 3. Conceptual model of artifact-driven architectural design

We will explain this model using OMT [Rumbaugh et al. 91], 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 [Rumbaugh et al. 91] 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 3.

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* [Rumbaugh et al. 91]. In Figure 3 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 3. This function is also supported by the concept *General Knowledge.*

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 OMT on an Automated Teller Machine (ATM) which concerns the design of a banking network [Rumbaugh et al. 91]. 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 B*ank Computers*. These subsystems group the artifacts that were identified from the requirement specification. 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 [Elmagarmid 91][Date 90][Bernstein & Newcomer 97]. 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[4]. 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

[4] In [Aksit & Bergmans 92] this problem has been termed as subsystem-object distinction.

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 provides, however, no sufficient support for this process. In the given example, the subsystem *ATM Stations*, *Consortium Computer* and B*ank Computers* are composed together, though, the rationale for the presented structuring process is performed implicitly. One could provide several possibilities for composing the subsystems, though, the method lacks rigid guidelines for composing and specifying the interactions between the subsystems.

3.4.2 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* [Jacobson et al. 99]. 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 [Jacobson et al. 99] 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 4. 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: *business & domain model*, *use-case model*, *analysis model*, *design model*, *implementation model* and *test model*.

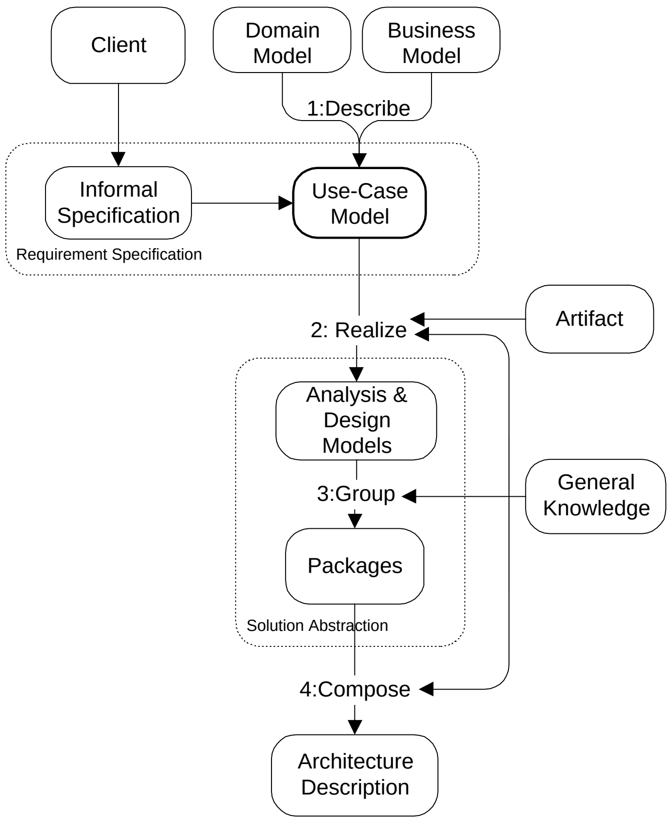

Figure 4. Conceptual model of use-case driven architectural design

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 4. 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 set 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*.

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 [Jacobson et al. 99, 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 is to be developed. In [Jacobson et al. 99 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 resulted 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[5].

- *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.

[5] 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.

- *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 realization 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* [Jacobson et al. 99, 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.

- *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.

3.4.3 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 5.

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 [Prieto-Diaz & Arrango 91]. 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 5 represent the concept *Domain Knowledge* in the meta-model of Figure 1.

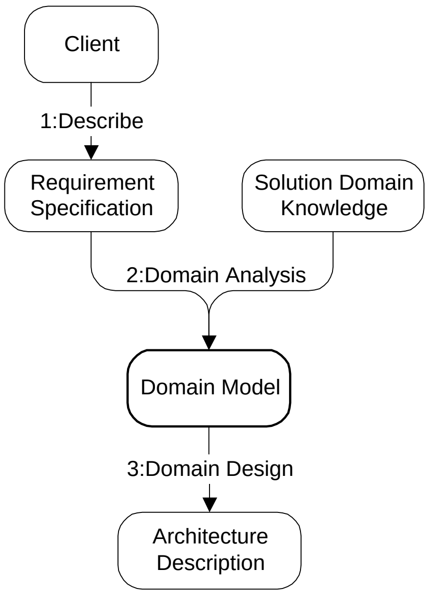

Figure 5. Conceptual model for Domain-Driven Architecture Design

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. [Gomaa 92], [Kang et al. 90], [Prieto-Diaz & Arrango 91], [Simos et al. 96] and [Czarnecki 99]. Two surveys of various domain analysis methods can be found in [Arrango 94] and [Wartik & Prieto-Diaz 92].

In this chapter we are mainly interested in the approaches that use the domain model to derive architectural abstractions. In Figure 5, 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.

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 [Clements & Northrop 96]. 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 [Bass et al. 97a], [Bass et al. 97b], [Clements & Northrop 96]. 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 6. 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.

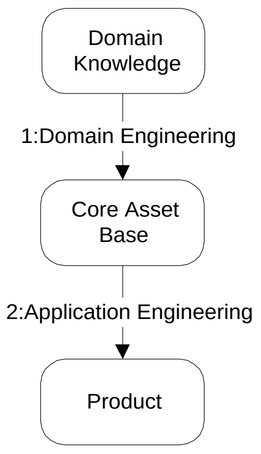

Figure 6. A conceptual model for a Product-Line Architecture Design

Note that various software architecture design approaches can be applied to provide a product-line architecture design. In the following section we will describe the DSSA approach that follows the conceptual model for product-line architecture design in Figure 6.

Domain Specific Software Architecture Design

The *domain-specific software architecture* (DSSA) [Hayes-Roth 94][Tracz & Coglianese 92] 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 7. 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) diagram, ER diagram, data flow models, state transition diagrams and object model.

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*.

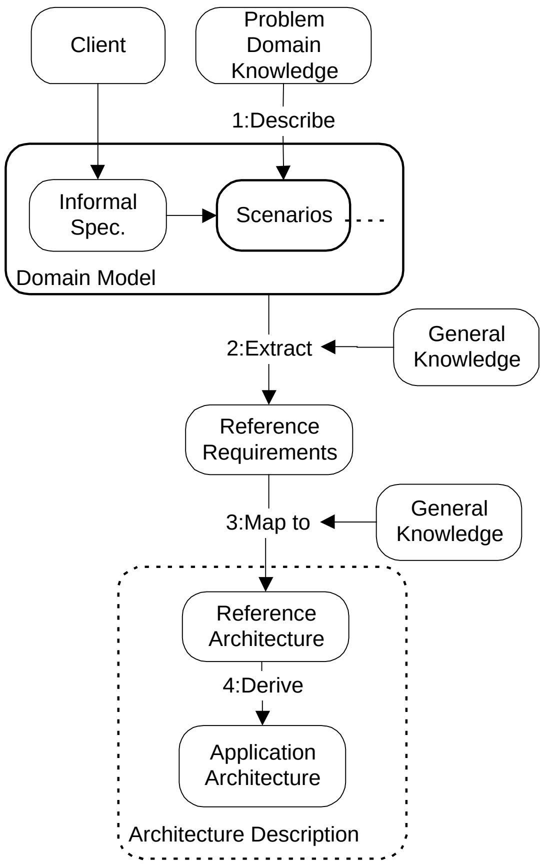

Figure 7. Conceptual model for Domain Specific Software Architecture (DSSA) approach

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 abstraction 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 [Tracz 95] 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 may not have reached a consensus yet and the area may 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. 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.

3.4.4 Pattern-driven Architecture Design

Christopher Alexander's idea on pattern languages for systematically designing buildings and communities in architecture [Alexander 79] has been adopted by the software community and led to the so-called software *design patterns* [Gamma et al. 95]. Similar to the patterns of Alexander, software design patterns aim to codify and make reusable a set of principles for designing quality software. The software design patterns are applied for the design phase, though, the software community has started to define and apply patterns for the other phases of the software development process. At the implementation phase patterns or idioms [Coplien 92] have been defined to map object-oriented design to object-oriented language constructs. Others have defined patterns for the analysis phase in which patterns are applied to derive analysis models [Fowler 96]. Recently, patterns have also been applied at the architectural analysis phase of the software development process [Buschmann et al. 99]. Architectural patterns are similar to the design patterns but focus on the gross-level structure of the system and its interactions. Sometimes architectural patterns are also called *architectural styles* [Shaw & Garlan 96]. An architectural pattern is not the architecture itself, as it is often mistaken, but rather it is just an abstract representation at the architectural level [Abowd et al. 94] [Bass et al. 98].

Pattern-driven architecture design approaches derive the architectural abstractions from patterns. Figure 8 depicts the conceptual model for this approach.

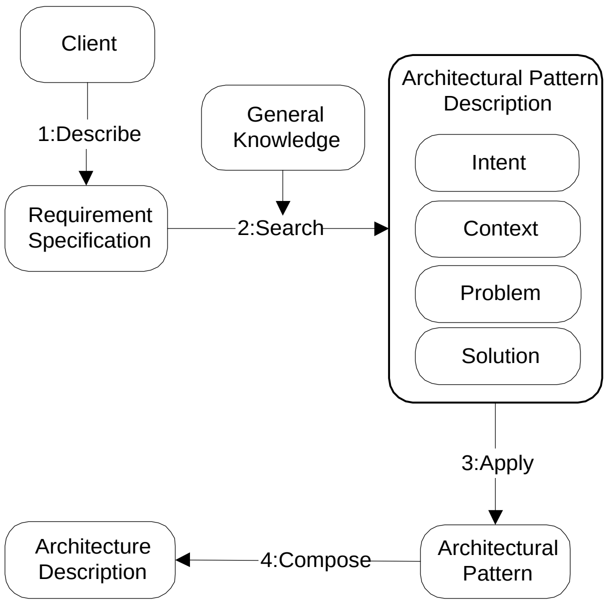

Figure 8. Conceptual Model for a Pattern-Driven Architecture Design

The concept *Requirement Specification* represents a specification of a problem that may be solved using a pattern. The function *Search* represents the process for searching a suitable pattern for the given problem description and is supported by the concept *General Knowledge*.

The concept *Architectural Pattern Description* represents a description of an architectural pattern. It consists mainly of four sub-concepts[6]: *Intent*, *Context*, *Problem*, and *Solution*. The concept *Intent* represents the rationale for applying the pattern. The concept *Context* represents the situation that gives rise to the problem. The concept *Problem* represents the recurring problem arising in the context. The concept *Solution* represents a solution to the problem in the form of an abstract description of the elements and their relations. For the identification of the pattern the intent of the available patterns is scanned. If the intent of a pattern is found relevant for the given problem then the context description (*Context*) is analyzed. If this also matches the context of the given problem, then the process follows

[6] There are other sub-concepts but we consider these four sub-concepts as important for the identification of the architectural abstractions.

with the function *3:Apply*. Thereby the sub-concept *Solution* is utilized to provide a solution to the problem. The concept *Architectural Pattern* represents the result of the function *3:Apply*. Finally, the function *4:Compose* represents the incorporation of the architecture pattern to the architecture description.

Problems

The pattern-driven architecture design approach is included as a sub-process in several architectural design approaches. Although architectural patterns are useful for building software architectures, the current approaches do not provide sufficient support for the selection of patterns, the application of these patterns and their composition to the architecture. We will describe these problems in the following:

- *Pattern base may not be sufficient for dealing with the wide range of architectural abstractions*

For a pattern-driven architecture design approach it is required that a sufficient base of patterns is available to support the design of software architectures. Currently, patterns have been catalogued in different publications such as [Buschmann et al. 99], [Shaw & Clements 97], [Gamma et al. 95], [Pree 95] and [Shaw 95]. Although, these catalogs provide practical vehicles for software architecture design, they do not and cannot cover all the problem areas for which architectures need to be developed. The reason for this is that architectures are composed of concepts representing abstractions from a particular domain and patterns define certain arrangements of these concepts and relations that are useful in solving recurring problems. Since there are numerous concepts and relations in the domain area, there are in principle also numerous architectural abstractions and accordingly numerous patterns. Consequently when utilizing a particular pattern catalogue, suitable patterns may be missing for a particular architecture design problem. In such cases it would be useful to provide means for generating new patterns for coping with novel but recurring problems.

- *Selecting patterns is merely based on the general knowledge and experience of the software engineer*

To ease the selection and manage and improve the understanding of patterns, patterns with common characteristics are usually classified into same groups. The classification criteria may differ per approach. For example, in [Shaw & Clements 97] and [Shaw 98] architectural patterns are classified according to the control and data interactions among architectural components. In [Buschmann et al. 99] patterns are classified into *problem categories*, grouping patterns addressing common problems. Sometimes, together with the categorization of the patterns a set of rules of thumb for choosing an architectural pattern is given as well. In [Shaw & Clements 97], for example, heuristic rules are given having the general form "If your problem has characteristic X then consider architectures with characteristic Y". An example of such a heuristic rule is, "If your problem can be decomposed into sequential stages, consider batch sequential or pipeline architectures". Despite of these classifications and heuristic rules it may happen that different alternative patterns are possible. Current approaches do not provide explicit support for prioritizing and balancing these alternative patterns. This is usually based on the experience and general knowledge of the software engineers. Therefore, this impedes the pattern-lookup process and as such the identification of the architectural abstractions.

- *Applying patterns is not straightforward and requires thorough analysis of the problem*

Once a pattern is selected the application of it is also not straightforward. A pattern is considered as a kind of template consisting of components and relations that must be matched with the concepts and concept relations identified in the problem domain. Examples of architectural patterns are *Pipes and Filters, Layering, Repositories, Interpreter,* and *Control* [Shaw & Garlan 96]. Assume for example, that for a given problem the architectural pattern *Pipes and Filters* is selected. In the *Pipes and Filters* architectural pattern the components are the *Pipes* and the *filters* represent the connecting relations between the components. *Filters* are components that receive input streams, do some processing and provide some output. *Pipes* transmit the output stream of one filter to the input stream of another filter. Important questions in applying this pattern to the problem are: Which concepts should be represented as *Pipes*; which concepts should be represented as *Filters*; how should be the structuring of *Pipes* and *Filters* etc. Currently, there is no serious support for this matching process and pattern application is also based on the experiences and general knowledge of the software engineer.

- *Composing patterns is not well-supported*

For developing software architectures usually several individual patterns need to be composed. Patterns are generally not independent and reveal several relationships with each other. Specifying the patterns independently will not reflect these interdependencies. In [Buschmann et al. 99] patterns are collected and organized into problem categories providing a problem-oriented view in selecting and applying patterns. Systematic approaches with explicit guidelines for composing patterns, however, is missing[7]. Assume that after the problem analysis it appears that the patterns *Layering*, *Repositories*, and *Pipes and Filters* need to be composed in the architecture. In the *Layering* pattern the architecture components are represented through *layers* and the connectors are the protocols that determine the interactions between the layers. The *Repository* pattern consists of a shared data structure and a set of independent set of components that access the shared data structure. How should we compose these three patterns? Which should be the basic pattern? Why? What are the dependencies? Current pattern-driven approaches lack to provide satisfactory answers for these questions because patterns are specified independently.

[7] In architecture design, Alexander introduced the concept of *pattern language* that defines the structure and the mutual arrangement of the patterns as an integrated whole [Alexander 79].

3.5 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*, *domain-driven* and *pattern-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. *Pattern-driven* architecture design approaches attempt to develop the architecture by selecting architectural patterns from a pre-defined pattern catalogue. 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[8]. 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[9].

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

[8] In [Aksit & Bergmans 92] this problem has been denoted as the *early decomposition* problem

[9] 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.

design phase and the architecture is developed in an iterative way. This solves the dilemma to some extent but the problems remain partially 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 and pattern-driven approaches the client requirements are directly used as a source 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, for example, 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 [Clements 96]. 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.

3.6 References

[Abowd et al. 94] Abowd, G.; Bass, L.; Kazman, R.; & Webb, M. *SAAM: A Method for Analyzing the Properties of Software Architectures*, 81-90. Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, May 16-21, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Aksit et al. 00] Aksit, M., Bergmans, L., Berg van den K., Broek, van den P., Rensink, A., Noutash, A., & Tekinerdogan, B. *Towards Quality-Oriented Software Engineering*, to be published in Software Architectures and Component Technology: The State of the Art in Research and Practice, M. Aksit (Ed.), Kluwer Academic Publishers, January 2000.

[Aksit & Bergmans 92] Aksit M. & Bergmans L. *Obstacles in Object-Oriented Software Development*, Proceedings OOPSLA ’92, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.

[Alexander 79] Alexander C., Ishikawa S., & Silverstein M. *A Pattern Language.* New York City: Oxford University Press, 1979.

[Arrango 94] Arrango, G. *Domain Analysis Methods.* In *Software Reusability*, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

[Bass et al. 97a] Bass, L., Clements, P., Cohen, S., Northtop, L. & Withey, J. *Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bass et al. 97b] Bass, L., Clements, P., Chastek, G., Cohen, S., Northrop, L.,. & Withey, J. *2nd Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bass et al. 98] Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice*, Addison-Wesley 1998.

[Bernstein & Newcomer 97] Bernstein, P.A., & Newcomer, E. *Principles of Transaction Processing*, Morgan Kaufman Publishers, 1997.

[Booch 91] Booch, G. *Object-Oriented Design with Applications*, The Benjamin / Cummings Publishing Company, Inc, 1991.

[Buschmann et al. 99] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1999.

[Clements 96] Clements, P. *A Survey of Architectural Description Languages*, Proceedings of the 8th International Workshop on Software Specification and Design, Paderborn, Germany, March, 1996.

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

[Clements et al. 85] Clements, P.; Parnas, D.; & Weiss, D. *The Modular Structure of Complex Systems.* IEEE Transactions on Software Engineering SE-11, 1, pp. 259-266, 1985.

[Coplien 92] Coplien, J.O. *Advanced C++ -Programming Styles and Idioms,* Addison-Wesley, Reading, MA, 1992.

[Czarnecki 99] Czarnecki, C., *Generative Programming*: *Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models,* PhD Thesis, Technical University of Ilmenau, 1999.

[Date 90] Date, C.J. *An Introduction to Database Systems*, Vol. 3, Addison Wesley, 1990.

[Dijkstra 68] Dijkstra, E.W. *The Structure of the'T.H.E.' Mulitprogramming System*. Communications of the ACM 18, 8 , pp. 453-457, 1968.

[Elmagarmid 91] Elmagarmid, A.K. (ed.) *Transaction Management in Database Systems,* Morgan Kaufmann Publishers, 1991.

[Fowler 96] Fowler, M. *Analysis Patterns : Reusable Object Models*, Addison-Wesley, 1996.

[Gamma et al. 95] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns, Elements of Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[Garlan & Shaw 93] Garlan, D. & Shaw, M. *An Introduction to Software Architecture*. Advances in: Software Engineering and Knowledge Engineering. Vol 1. River Edge, NJ: World Scientific Publishing Company, 1993.

[Garlan et al. 95] Garlan, D., Allen, R., & Ockerbloom, J. *Architectural Mismatch: Why It's Hard to Build Systems Out of Existing Parts*, 170-185. Proceedings, 17th International Conference on Software Engineering. Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, 1995.

[Gomaa 92] Gomaa, H. *An Object-Oriented Domain Analysis and Modeling Method for Software Reuse.* In *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, January, 1992.

[Hayes-Roth 94] Hayes-Roth, F. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*, 1994.

[Howard 87] Howard, R.W. *Concepts and Schemata*: *An Introduction,* Cassel Education, 1987.

[Jacobson et al. 99] Jacobson, I., Booch, G., & Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

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

[Kruchten 95] Kruchten, Philippe B. *The 4+1 View Model of Architecture*. IEEE Software, Vol 12, No 6, pp. 42-50, November 1995.

[Lakoff 87] Lakoff, G. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind,* The University of Chicago Press, 1987.

[Parnas 72] Parnas, D. *On the Criteria for Decomposing Systems into Modules.* Communications of the ACM 15, 12 (December 1972): 1053-1058.

[Parnas 76] Parnas, D. *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering SE-2, 1: 1-9, 1976.

[Parsons & Wand 97] Parsons, J., & Wand, Y. *Choosing Classes in Conceptual Modeling*, Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997

[Perry & Wolf 92] Perry, D.E. & Wolf, A.L. *Foundations for the Study of Software Architecture.* Software Engineering Notes, ACM SIGSOFT 17, 4: 40-52, October 1992.

[Pree 95] Pree, W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[Prieto-Diaz & Arrango 91] Prieto-Diaz, R., & Arrango, G. (Eds.). *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, Los Alamitos, California, 1991.

[Rosch et al. 76] Rosch , E., Mervis, C.B., Gray, W.D., Johnson, D.M., and Boyes-Braem, P. *Basic objects in natural categories.* Cognitive Psychology 8: 382-439, 1976.

[Rumbaugh et al. 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[SEI 00] Carnegie Mellon Software Engineering Institute, Web-site: http://www.sei.cmu.edu/architecture/, 2000.

[Shaw 95] Shaw, M. *Making Choices: A Comparison of Styles for Software Architecture.* IEEE Software 12, 6 27-41, November, 1995.

[Shaw 98] Shaw, M. *Moving from Qualities to Architectures: Architectural Styles*, in: L. Bass, P. Clements, & R. Kazman (eds.), Software Architecture in Practice, Addison-Wesley, 1998.

[Shaw & Clements 97] Shaw, M., & Clements, P. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*, Proc. COMPSAC97, 1st Int'l Computer Software and Applications Conference, August, 1997.

[Shaw & Garlan 96] Shaw, M. & Garlan, D. *Software Architectures: Perspectives on an Emerging Discipline*,. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[Simos et al. 96] Simos, M., Creps, D., Klinger, C., Levine, L., & Allemang, D. *Organization Domain Modeling (ODM) Guidebook*, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, http://www.organon.com, 1996.

[Smith & Medin 81] Smith, E.E., & Medin, D.L., *Categories and Concepts*, Harvard University Press, London, 1981.

[Stillings et al. 95] Stillings, N.A., Weisler, S.E., Chase, C.H., Feinstein, M.H., Garfield, J.L., & Rissland, E.L., *Cognitive Science: An Introduction.* Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.

[Soni et al. 95] Soni, D., Nord, R., Hofmeister, C. *Software Architecture in Industrial Applications.* 196-210. Proceedings of the 17th International ACM Conference on Software Engineering, Seattle, WA, 1995.

[Tracz & Coglianese 92] W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines. Technical Report*, ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.

[Tracz 95] Tracz, W. *DSSA (Domain-Specific Software Architecture) Pedagogical Example.* In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, July 1995.

[Wartik & Prieto-Diaz 92] Wartik, S., & Prieto-Díaz, R. Criteria for Comparing Domain Analysis Approaches. In International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, pp. 403-431, September 1992.

[Webster 00] Merriam Webster on-line Dictionary, http://www.m-w.com/cgi-bin/dictionary, 2000.

[Wittgenstein 53] Wittgenstein, L. *Philosophical investigations*, Macmillan, New York, 1953.

Table of Contents