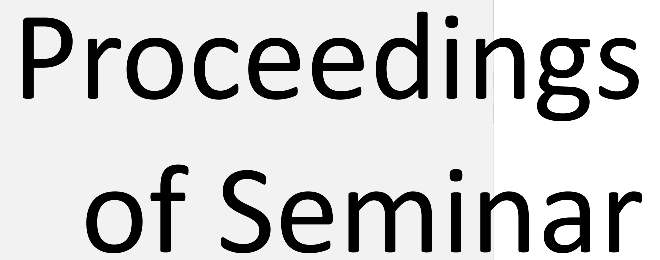

Proceedings of Seminar

Software Architecture Representation and Evaluation

2013

Editors: Horst Lichter
Ana Nicolaescu

SWC Software Construction

RWTH AACHEN UNIVERSITY

SOFTWARE CONSTRUCTION

RWTH AACHEN UNIVERSITY

Software Architecture Representation and Evaluation

— 13 - 14 February March 2013 —

Advanced Seminar in Computer Science

Research Group Software Construction
Prof. Dr. rer nat. H. Lichter
Winter Term 2012 / 2013

Contents

1 Enterprise Architecture Management **1**

1.1 Introduction 2

1.2 Enterprise Architecture 3

1.3 Frameworks 12

1.4 Conclusion and Future Work 18

Bibliography 19

2 Software Architecture Representation **21**

2.1 Motivation and Overview 22

2.2 Basic concepts 24

2.3 View-based models for the software architecture description . . . 28

2.4 Reference Architecture 33

2.5 Architecture Description Methods 35

2.6 Basic Architectures 39

Bibliography 41

3 Component-based development and architecture **43**

3.1 Introduction 44

3.2 Components 46

3.3 Component models 50

3.4 Conclusion 59
Bibliography 60

4 Model-driven Architecture 63

4.1 Introduction 64
4.2 Basic concepts 65
4.3 The principles of Model-Driven Architecture 68
4.4 Model-Driven Architecture in practice 74
4.5 Related work 78
4.6 Conclusion 80
Bibliography 80

5 Software Architecture Evolution 83

5.1 Introduction 84
5.2 Background 85
5.3 Related Works 88
5.4 Software Architecture Evolution 92
5.5 Conclusion 96
Bibliography 97

6 Software Architecture Reconstruction 101

6.1 Introduction 102
6.2 Software Architecture Reconstruction Goals 104
6.3 Architecture Reconstruction Process 105
6.4 SAR Inputs 109
6.5 Reconstruction Techniques 111
6.6 SAR Outputs 113
6.7 Model Driven Architecture Reconstruction 114

6.8 Summary and Future Research Opportunities 117

Bibliography 117

7 Extraction of the static view of software architectures 127

7.1 Introduction 128

7.2 SAR Approaches and Frameworks 129

7.3 Recovery Algorithms 135

7.4 Tools Overview 139

7.5 Visualization Techniques 145

7.6 Conclusion 147

Bibliography 148

8 Extraction of the Dynamic View of Software Architectures 151

8.1 Introduction 152

8.2 General Information 152

8.3 Dynamic View Extraction Techniques 153

8.4 Dynamic View Extraction in Practice 156

8.5 Case Study 158

8.6 Summary 163

Bibliography 164

9 Architecture evaluation 169

9.1 Introduction 172

9.2 Background 172

9.3 Benefits and costs of evaluation 177

9.4 Methods of SA evaluation 180

9.5 Conclusion 190

Bibliography 191

Chapter 1

Enterprise Architecture Management

Shobhit Sharda

Contents

1.1 Introduction 2
1.2 Enterprise Architecture 3
1.2.1 What is Enterprise Architecture 3
1.2.2 Enterprise Architecture Facets 5
1.2.3 Modelling in Enterprise Architecture 7
1.3 Frameworks 12
1.3.1 Enterprise Architecture Frameworks(EAF) 12
1.3.2 TOGAF 14
1.4 Conclusion and Future Work 18
Bibliography 19

Abstract: Today's business world has become highly competitive. It has shifted from notebooks, papers and pens to computers. As a consequence companies are being forced to adopt modern technologies and proper planning in order to successfully achieve its business objectives, goals, vision, organizational structures and business strategies. The current scenario of business strategies comes with plenty of challenges and issues which are sensitive for any company's growth. From the requirement specifications of a customer to the fully implemented product, there are numerous steps that have to be taken into account. Thus, it becomes an absolute necessity that the company should adopt a systematic approach to manage its business processes. Thus, in this paper I have introduced the concept of "Enterprise Architecture" which is a solution to the above mentioned concern. It is an emerging idea responsible for the high success rate of various companies. I have also discussed about TOGAF, a well known enterprise architecture framework.

1.1 Introduction

Generally, enterprise applications are extremely complex which are used by many users. Multiple teams are involved in its development and they are often deployed on heterogeneous systems which are spread over various environment. Thus, in order to have a complete view of a company, its application landscape, its business process, its business strategies and its technical infrastructure, we need to express its various aspects, domains and their relations in a meaningful manner which can be understood by all the stakeholders. To understand the complexity of any application landscape or organization it is extremely important to analyse its architecture description. But a big question lies on the meaning of the term "architecture"? We have often heard this term in relation to the concepts of buildings and constructions. Here different aspects like bedroom, kitchen, dining, and bathroom are put together by an architect to create a master plan which is then used by engineers and builders to construct a particular building. We adopt a similar approach when it comes to designing an enterprise.
In section 1.2 of this paper, I will discuss about the basic concept of enterprise architecture, its importance and benefits and architecture process. I will also discuss various enterprise architecture facets and their relation to enterprise architecture. Later in this section, we will see different aspects of modelling applied in enterprise architecture. I have examined how Unified Modelling Language, planningIT and few other methods can be used to model enterprise architecture.
In section 1.3, I will introduce basic features of Enterprise Architecture Framework (EAF) and different roles that are described by EAF. Later in this section I will be focusing on various aspects of TOGAF including its Architecture Development Method (ADM) which is the core of TOGAF. This section is followed by conclusion and future work.

1.2 Enterprise Architecture

1.2.1 What is Enterprise Architecture

Definition

A "*Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principle guiding its design and evolution*"[Lan99]. It is often observed that most of the stakeholders are more interested in the end result due to architecture than understanding the architecture itself. However, it is the responsibility of the architect that all stakeholders perceive the architecture. An architect should be able to explain his architecture to all the stakeholders irrespective of their backgrounds.
We refer an enterprise as "*Any collection of organizations that has a common set of goals and/or a single bottom line*"[Lan99] is referred to as an 'enterprise'. Now when architecture is discussed at an enterprise level it becomes 'enterprise architecture'. We can define an Enterprise Architecture as "*a coherent whole of principles, methods, and models that are used in the design and realisation of an enterprise's organisational structure, business processes, information systems, and infrastructure*" [Lan99].

Importance and Benefits

It is observed that we often change the architecture because of the changing needs, environmental changes or introducing new technologies which are more effective and helpful in achieving the business objectives. We should always keep in mind that enterprise architecture has to be designed in such a way that it should be open to accept the changes at any given point of time. Thus, with well-understood and properly documented enterprise architecture, the organization can respond quickly to the changing needs. Moreover, well-documented enterprise architecture will act as a ready reference which enables the organization to assess the impact of the changes on various enterprise architecture components. With this we can also make sure that the each of the components operates smoothly.[oHEA11]
A sensible architecture always provides an enterprise with complexity management, technical resource oversight, knowledge management, IT visibility, reduction in impact of staff turnover, faster adaptability, operating procedures improvement and decision making. Enterprise architecture also allows all the stakeholders to view different aspects of information from different perspectives. This will thereby help the organization in optimizing the business goals. Using enterprise architecture, an organization can organize and structure their enterprise in an efficient way.[SK07]

Architecture Process

We can consider architecture both as a product and process. The product guides managers and system developers to design business process and build the applications respectively, which are in accordance with the business objectives and policies. The effect of process is much more than just product creation. The level of awareness among the stakeholders will be much higher. Moreover, it is important to maintain the architecture once it is created because there is a continuous change in business strategies.
The architecture process contains (figure 6.3) several steps. Ideas are gathered during the initial steps. These ideas are normally noted down with the help of whiteboards or PowerPoint. Then comes the design phase where the ideas are converted into the useful models. Once the models are approved, the implementation is carried away thus resulting in application landscape. This phase is called the use phase. The overall application landscape is then maintained in management phase. There might be some ideas which came later and have to be incorporated. To accomplish this, the existing application landscapes are again analyzed and the previous steps are again adopted, thereby completing the loop.[Lan99]

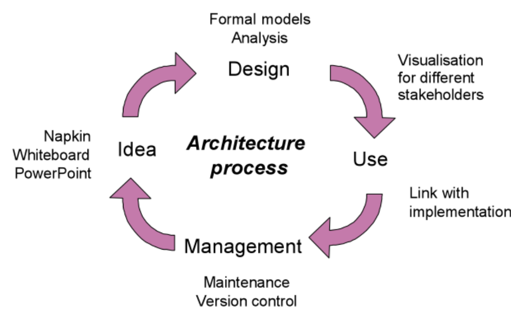

Figure 1.1: Architecture Process[Lan99]

We should note that, product creation process, and policy and planning process are the two aspects of architecture process. It is architecture process which is responsible for the integral technical aspects of product creation process and integral technical vision and synergy in the policy and planning process. Moreover, architecture process is responsible in maintenance of the consistency throughout the system ranging from the requirement to implementation and verification. It is estimated that the system architecture team, who are the owners of the architecture process, spend about 80% of their time in product creation process. The remaining time is devoted to policy and planning process. It is also observed that the small amount of time is spent on technology and people management as there are lots of people involved.[Mul00]

1.2.2 Enterprise Architecture Facets

Normally, we include 4 different facets of Architecture in Enterprise Architecture Facets. These are used in an organization on day to day basis. These facets have their own architecture, specifications, benefits and end results. They are:

- Software Architecture
- Business Architecture
- Infrastructure Architecture
- Information Architecture

Software Architecture:"*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them*"[BCK03]. Software Architecture defines software elements and the relation between those elements. Elements can either be private or public. They use interfaces to interact among themselves. We can reveal an interesting fact from the definition that any computing system based on software will have software architecture. We can show every system as elements and very well define the relation between them. But no matter how well we divide the system into elements, the software will still have architecture. Moreover, for a set of elements we often propose numerous architecture plans. But the difference between these variants is the way the various quality attributes should look like, when employing a certain solution.
We observe that the evolution of the software architecture is mainly due to the design principles of designers and their actions while working on a system. Designers always try to extract the commonalities among several systems or within a single system. Generally, in an organization we see that the team of designers partition whole project into smaller implementation units and these are then given to the developer teams. Each of these units is meant to carry a specific functionality and might be dependent on any other units. They are generally carried out on the basis of weekly assignments. Depending on the size of projects these units may further be divided into subunits.
However, we have to remember that there is lot more than just designing architecture. Efficient data structures, high end and modern technologies to be used, interface to those data structures and performance of the system are all architectural concern which we have to take into account while designing the software architecture for a system.

Business Architecture:Here, I have described some terms that are relevant for this section. A **business model** is nothing but the business plans that describes how and organization can make profits. It also includes company's strategies to compete today's business.[OST05] A **business strategy** states

the direction in which a business should pursue and necessary steps that should be adopted in order to achieve the business goals.[Wis13] A **business process** is a collection of activities which are carried to accomplish a certain organizational goal.[Rou05]
"*A business architecture is a part of an enterprise architecture related to architectural organization of business and the documents and diagrams that describe that architectural organization*"[IAS12]. We use the term Business Architects to refer the person who are responsible in building the business architecture. Business architecture is mainly used in terms of modelling approaches. Business architecture links business and IT and strategy and operations. This link is achieved using the concept of business model. For successful business, business architects in companies are engaged in designing business models. They often give less importance to the business strategies. It is observed that a good architecture consists of business models which allow the modelling of any organizational entity together with its multi-dimensional organizational views. Many companies use business architecture to combine business strategies on one hand and roles, business processes, information and behaviour on the other hand. We should understand that business architecture is authoritative for the structural responsibilities and can be used by one organization, several organizations or one part of an organization.

We must note that the main element of business architecture is business domains. Business domains are the clusters of coherent functions and objects thus defining the meaningful responsibilities. Business functions are the set of operations that are performed regularly to achieve some specific business goals. We use business concepts to define the target audience, features and benefits of the end product, description and size of the target market, strategies on implementing the product within the target market and a detailed study of how and how much revenue and profits can be generated. Overall we can say that the business architecture contains: a layout of business domains, business functions and business concepts and high level business process.

Infrastructure Architecture: We have to admit that the infrastructure architecture is a core part of any enterprise architecture. It holds the structure and behaviour of the technological infrastructure of any application landscape. We use it to define various aspects of hardware configurations like client and server nodes, the application that can run on them, the services that can be offered to them, the necessary protocols and the network interfaces that connect both of them. Using infrastructure architecture, we can also state some of the important variants like performance, storage, resilience and backup.[IAS12]

We know that Information and communication technology (ICT) enable business. ICT cannot survive without infrastructure and most businesses cannot exist without infrastructure. Of course we cannot say that infrastructure alone is responsible for the automation or computerization to achieve business objectives as most of the business logic lies under software application which is responsible for the automation of a system. Thus, during the early ages, infras-

tructure was never given much importance but gradually as more complex and faster hardwares were invented, there was a strong need to focus on infrastructure. This embarked the rapid development and deployment of application landscape.[Mic12]

Information Architecture:" *Information Architecture is the study of and application of structuring data and information and defining user interactions based on the flow of this data*"[Zar11]. It helps enterprise to retrieve, store, edit, add and delete the information in an efficient manner that is relevant so that they can achieve their business goals. Being more specific and restricting ourselves only to internet data, we all know that there is a vast amount of information available over internet which is expanding in a drastic manner. Thus, it is obvious that there can be number of ways by which we can seek information. Hence it is a challenge for the architects to design the application landscape in such a way so that all the possible ways of information retrieval and processing can be incorporated.
For example: from our day to day life it is clear that we generally fail to shop completely not because we did not have the interest but because we could not completely navigate through the website. Thus, it becomes crucial for the information architect to group and categorize the information which is easily accessible by anyone. They should keep in mind to define the levels of interaction and the hierarchy of the information. Moreover the navigational schemes should be dynamic in such a way that not only we can find what we are searching for but to get back as well. The navigation should be flexible enough to provide multiple ways of reaching particular information. From many studies, I have found that the websites with high redundancy of links have a higher success rate in terms of users that remain on the website. In general, information management system whether a website or record keeping or any other means of keeping any kind of data, which does not include information architecture risks of losing their customers thus affecting the business goals[Zar11]

1.2.3 Modelling in Enterprise Architecture

Unified Modelling Language (UML)

"*The UML is a graphical language for visualizing, specifying and constructing the artefacts of a distributed object system*"[Kob98]. UML is relatively simple yet powerful. It is a general purpose language and supports many modelling approaches. This language depends on many small core concepts which we can easily learn and apply. These core concepts are combined and extended thus helping modellers to define the application landscape or complex systems over a wide range of domains. These core concepts includes numerous constructs and diagram techniques that are useful for architectural modelling.[Kob98]
Overall we can classify UML diagrams in three categories[Amb04]:

- 1. **Behaviour diagrams:** Using these kind of diagram we can explain the behavioural features of a business process. This category includes activity diagrams, state machine diagram and use case diagrams.
- 2. **Interaction diagrams:** We can consider this as a subset of behaviour diagrams which focus on the interaction among various objects. This category includes communication (or collaboration) diagram, interaction overview diagram, sequence diagram and timing diagram.
- 3. **Structure diagrams:** We use these kinds of diagrams to depict the elements of a specification which are independent of time. This category includes class diagram, composite structure diagram, component diagram, deployment diagram, object diagram and package diagram.

It is important to note that UML diagrams are not the only way which we use to model various enterprise architecture facets. There are various other methods that can be used for the same purpose. Let us now have a closer look on how these facets can be modelled keeping UML as main focus.

Software Architecture: We can sometime divide software architecture as structural (static) software architecture and dynamic software architecture. Structural (static) software architecture defines the structure of all its software elements in a complex system or application landscape like class hierarchy, class library structure and relationship (inheritance, association, aggregation etc) among classes. The UML diagrams like class diagram, component diagram, deployment diagram and object diagram etc can fulfil these purposes. On the other hand, a dynamic software structure defines the behaviour of the complex system or application landscape like collaboration, interaction, activity and concurrency. The UML diagrams like use case diagram, sequence diagrams, activity diagrams and state machine diagrams can fulfil these purposes.

However, sometimes during the initial phase, simple box-line diagrams are used to describe different components of software architecture and relationship between those components. We can also observe that the lines are generally associated with arrows to show the flow and sequence. Figure 6.6 shows an example of box-line diagram for an online shopping business. After browsing, the customer selects and put the items in the shopping cart. After checking out, the system examines customer's credit record, update the inventory and notifies the shopping department. They then process the order.[QFTX09]
Alternatively, we can use Architecture Description Language (ADL) to describe software architecture. It is a formal specification language which has well defined syntax and semantics. It has an ability to decompose components, combine components and define interfaces of components.[QFTX09]

Business Architecture: Business process focuses on the set of activities and it is important that these activities are well defined and understood by all the

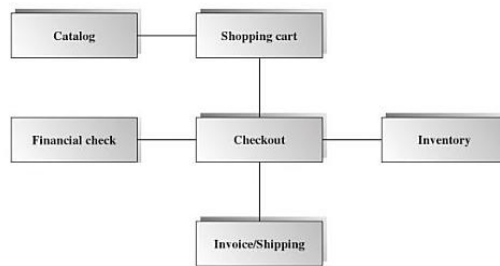

Figure 1.2: An example of Box-Line-Diagram[QFTX09]

stakeholders. UML activity diagram is thus used to graphically define these set of activities. Thus, among the entire thirteen UML diagram, activity diagram is the most important UML diagram that is used for business architecture. Using activity diagram, we can show the objects consumed, used, or produced by an activity, indicate the responsible person for an activity and we can also specify the relationship and dependencies between the activities.
However, in many companies, overall structure of an organization (which includes divisions, departments, sections etc), structures of resources, the products or the services, profits and loss data are modelled and represented with the help of organizational charts and descriptions and not using any UML diagrams. Moreover, we can model a business process using Business Process Model and Notation (BPMN). We can use BPMN to define the standards that are used to design and model business process that should be understood by to all the stakeholders. Business process diagrams that are based on flow chart technique (similar to activity diagram) are used under BPMN. [EP00]
An example of BPMN diagram is shown in figure 6.7 which describes the process between Bank and Customer. Customer orders cheques using an order form. On receiving the order request from the customer, it prints the cheques and mails the cheques to the customer via mail. Customer later receives the cheques.[IBM13]

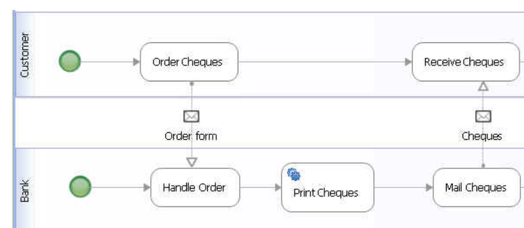

Figure 1.3: An Example of BPMN [IBM13]

Infrastructure Architecture: We have already seen that one of the key features of infrastructure architecture is to focus on the hardware deployment and interaction between them. Thus we can use deployment diagram and interaction overview diagram. Deployment diagram is used because it represents the execution architecture of the systems. It includes nodes, hardware or soft-

ware execution of complete application landscape and any middleware that connects them. Interaction overview diagrams are used because it describes the control flow within the business process.[Amb04]
However, in most of the companies, network diagram is preferred over any UML diagram for infrastructure architecture. Network diagrams are easy to understand compared to deployment diagrams. With network diagrams we can quickly understand the surrounding and the environment thus making it easy to deploy the hardware. An example of network diagram is depicted in figure 6.4. Here three computers are connected to a switch which is connected to the server. The server is connected to a router which is then connected to the Internet. These computers may lie in different geographic locations.

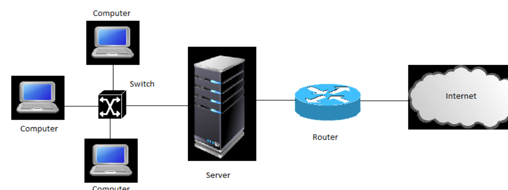

Figure 1.4: An Example of Network Diagram

Information Architecture: As we know, information architecture focuses on structuring data and information and defines user interaction. Here, we generally adopt use case diagrams to describe the scenarios in which users can interact with different systems in an application landscape. The sequential flow of user's interaction can be described with the help of sequence diagram. While user is interacting, the overall system state keeps changing. State machine diagrams can be used to describe the change of state. The data flow is sometime represented using activity diagram.
Site diagrams are alternative solutions to manage the content, its navigation and its interaction with the users. They help in developing the information hierarchy and hence it is easy for all the stakeholders to understand the organizational concepts. Generally during the first meeting, a rough site diagram is produced in which business analysts work upon and at the end of every meeting the revised diagram becomes the official result of the meeting. Thus it evolves as the plan evolves .Moreover, information architecture uses various data modelling techniques to achieve this for data representation and structuring.[LH13]

PlanningIT

"*PlanningIT is an application for enterprisewide IT planning and management*"[Nuc10]. With the help of this application, IT decision-makers can document and analyze their environment and can then collaborate with each other while making

investment decisions. It is capable of examining all the management tasks. With a given set of information, it creates a repository. Reports and graphs can be generated which will represent the necessary information from the repository. Easy to use query builder feature of planningIT allows us to launch customized data analysis. A collaboration feature of planningIt allows us to work with entire team on enterprise architecture.[Her08]
Moreover, business relationship management tools help people in both IT and lines of business to collaborate in assessing the IT supports and demands. IT planning functionality helps in creating the target architectural design. It includes the project prioritization and risk assessments thus minimizing the risk and cost of the project. Risk management tools help us to evaluate the existing architecture and controls thereby determining and minimizing the risk associated with the project. Financial management tools helps in budget fixing and analysis the project to minimize the cost of the enterprise management. Architecture management functionality evaluates the performance of the current architecture using some metrics and domain models. This will help us in identifying the projects that are duplicative and conflicting in nature.[Nuc10]

The planningIT provides a comprehensive meta-model that fulfils the key components of enterprise architecture management, demands to budget, strategy management, collaboration and governance. The simplified planningIT meta-model contains Business layer, Application layer, Physical layer and Reference Data. It is often seen that, for any project even if we start with a very small project, we will soon end up at a point where we have to extend, reshape and redesign the meta-model so that the requirements can be met.[Cam09]
The figure (figure 1.5) shows an example of information flow between the application in the center and other application in the application landscape. Here the flow of information is categorized in 3 types.

- 1. Incoming arrows which means that the information is received by other application.

- 2. Outgoing arrows which means that the information is passed to other application.

- 3. Bidirectional arrows which mean that the information is exchanged both the ways.

The arrows in the figure can be interpreted in many ways. Here it represents the frequency of information exchange between applications which could be monthly, daily, on demand or at any intervals.

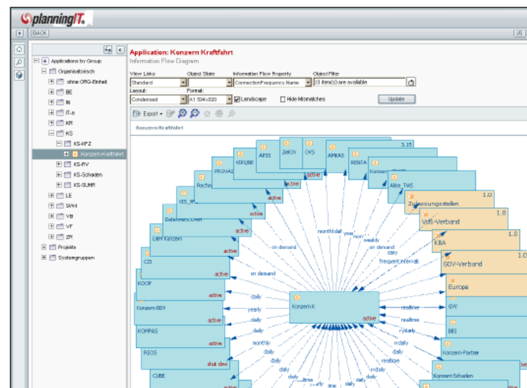

Figure 1.5: planningIt[Ser]

1.3 Frameworks

1.3.1 Enterprise Architecture Frameworks(EAF)

"*An Enterprise Architecture Framework (EAF) maps all of the software development processes within the enterprise and how they relate and interact to fulfil the enterprise's mission*"[UM06].We can define enterprise architecture with respect to EAF as "*Enterprise architecture is an integrated and holistic vision of a system's fundamental organization, embodied in its elements(people, process, applications and so on), their relationships to each other and to the environment, and principles guiding its design and evolution*" [SK07].Many organizations, both government and private have adopted such frameworks to fulfil their business needs. EAF details the methods that are useful in designing the application landscape. These methods are defined in terms of building blocks and they are equally responsible to bring these blocks together.

However, some rules and regulations have to be followed while implementing the systems. EAF also contains the list of these rules and regulations. It also covers the complete architectural dimensions. They play an important role in documentation and facilitate enterprise planning and problem solving. An interesting feature to note is, EAF are language independent in nature and provide a generic concepts. All the above mentioned features help stakeholders to communicate easily thereby simplifying the development of the architecture.[SK07] In general EAF are decomposed into three layers[SK07]:

- 1. The **business layer** that defines the business entities and the way entities interact with each other.
- 2. The **application layer** that determines the data elements and software applications. They are also responsible to support business layer.

- 3. The **technology infrastructure layer** that constitutes the hardware platforms and the communication infrastructure. This supports the application layer.

Each of these layers contains many domains that reflect an organization's information, behavioural and structural aspects. We can define architectural aspects like process, product, information and application architectures. For example, process domain being part of business layer describes business processes.[SK07]

Roles in EAF

There are some roles defined by EAF in developing the application landscape. They are:

- 1. **Chief Enterprise Architect** acts as enterprise architecture repository's owner. As they interact directly with chief information officer, they are also called systems champion. They have to be up to date with the new technologies, standards and methodologies that are being used. Planning of the whole project is also carried by them and important decisions like scheduling tasks, allocating resources and monitoring the progress of the systems is also taken care by them.
- 2. **Enterprise business architect** is responsible to analyze and document the business processes, scenarios and information. They are also responsible in monitoring the allocated resources.
- 3. The role of **enterprise IT architect** is to analyze and document the system's data flow and internal and external interfaces. They also should ensure the overall quality factors like availability, scalability and recoverability of the application landscape. In addition, they are responsible for developing, designing and evaluating the current and proposed architectural model.
- 4. **Infrastructure architect** documents and analyzes the application landscape environments, network communications, operating systems and middleware components.
- 5. **System architect** on the other hand collaborates with the enterprise IT architects and is responsible in choosing the suitable frameworks which is in accordance with the application landscape quality.
- 6. **Data architects** are responsible in developing data architecture. "***Data architecture*** *is a formal description and mapping of the plan and structure of data assets used to support organizational goals*"[Man13]. The

main component of data architecture includes databases and data flow. Thus, data architects analyze and design the database related components. They form the policies related to data management, storage and access.

Some of the most successful frameworks that are used in specific areas are[UM06]:

- 1. **The Zachman Framework** consists of six perspectives or views: Planner, Owner, Designer, Builder, Subcontractor, and User and it deals with the six basic questions: what, how, where, who, when and why. It does not focus on sequence, process or implementation but it make sure that all the views are well established.
- 2. **Department of Defense Architecture Framework (DODAF)** consists of three main views: Operational, System and Technical Standards. The fourth view is called ”All View” which links the other three views. It can compare and integrate families of systems, systems of systems and can ensure the interoperation and interaction between architectures.
- 3. **Federal Enterprise Architecture Framework (FEAF)** was developed by US Federal Chief Information Officers (CIO) council. Its focus is to develop, maintain and facilitate integrated system architecture and its main goal is to organize and promote sharing of information for the entire federal government.
- 4. **Treasury Enterprise Architecture Framework (TEAF)** was developed in order to map the interrelationships among the organizations to manage IT resources effectively. Thus its main goal is to facilitate, integrate, share information, and exploit common requirements across the enterprises.
- 5. **The Open Group Architecture Framework (TOGAF)** is discussed in detail in the following section.

1.3.2 TOGAF

Introduction to TOGAF 9

TOGAF stands for The Open Group Architecture Framework. We can use this framework to manage enterprise architecture. It contains a detailed method and set of tools. It is freely available and can be used by an organization which wants to manage its enterprise architecture. TOGAF allows architecture to be consistent. They consider both the current requirements and the future need to attain the business objectives.
Who will benefit on using TOGAF?: ”*Any organization undertaking, or planning to undertake, the design and implementation of a enterprise architecture*

for the support of mission critical business application will benefit from use of TOGAF" [Gro09]. Moreover any organization that wants to manage information would want to use TOGAF. With the help of TOGAF, an organization can define their business structures and business processes that are to be exchanged between enterprises. TOGAF ensures an open system implementation. Finally the definition of TOGAF is: "*TOGAF provides the methods and tools for assisting in the acceptance, production use and maintenance of an enterprise architecture. It is based on the iterative process model supported by best practices and a reusable set of existing architecture assets*" [Gro09].

Architecture in Context of TOGAF

With respect to TOGAF context, we can have two meanings of architecture[Gro09]:

- 1. Architecture is a formal description or a detailed plan of the systems that will guide the implementation process.
- 2. Architecture is a set of the components, defines the relationship among these components and contains certain principles and guidelines that govern their design and evolution over time.

Moreover there are four architecture domains that are described by TOGAF. They are[Gro09]:

- 1. A **Business Architecture** which is responsible for business strategies, governance, organization and key business process.
- 2. The **Data Architecture** which is responsible for the structure of organization's logical and physical data and data management resources.
- 3. The **Application Architecture** serves as a blueprint for each application to be deployed. It also defines the interaction and relationships between them.
- 4. The **Technology Architecture** is responsible for the logical software and hardware capabilities. These are necessary for the deployment of business, data and application services. IT infrastructure, middleware, networks, communications etc are the subset of technology architecture.

TOGAF and IT Strategy

We can define a strategy as a specific plan of action that is designed to achieve specific goals. The basic set of IT strategy can be categorized into set of 5. They are:

- **Application Strategy:** It contains all the required guidelines on how we can deal with an IT application landscape of an organization. It also describes how business strategies are supported using IT application.
- **Integration Strategy:** Generally, we develop a set of application which is integrated to get the final application. The integration strategy defines these instructions using which integration is carried. It also describes the instructions on how can we connect the application landscape with outside world.
- **Infrastructure Strategy:** We can decide various infrastructure factors including global scope, security, special solutions under this strategy.
- **Service Strategy:** It is really important to discuss how a customer is going to get the services offered.
- **Sourcing Strategy:** The point of discussion here is what to outsource? , what to produce? , and work with single or multiple providers.

These IT strategies are the backbone of any successful business process. However, as far as TOGAF is concerned, it does not detail anything regarding business strategies and IT Strategies.[Wol09]

Architecture Development Method(ADM)

ADM is the core of TOGAF and is responsible in developing enterprise architecture. It combines various TOGAF elements and other architectural concepts to meet the business and IT needs of an organization. There are several phases (figure 6.2) within an ADM. They are described as follows[Gro09]:

- 1. The **Preliminary Phase** defines initial preparation required to meet the business objectives. This include recognizing all the stakeholders, identify the elements and define their scope, defining people's responsibility to design the architecture, define the framework, methodologies and tools that are going to be used etc.
- 2. **Phase A: Architecture Vision** is responsible to validate the business principles, define scope, find relevant stakeholders and their concerns and objectives, define key business requirements, articulate architecture vision, create plan that address scheduling, resourcing, financing, communication, risks, constraints, assumption, dependencies etc, obtain the necessary approvals etc.
- 3. **Phase B: Business Architecture phase** develop the business architecture that describes architecture vision, product and service strategy

and organizational, functional, process, information and geographic aspects of the business environment. It also helps in selection and development of architectural viewpoints which will allow architect to demonstrate how the stakeholder concerns are addressed in business architecture. It also allows the selection of tools and techniques that should be in accordance with the selected viewpoints.

- 4. **Phase C: Information Systems Architectures phase** is responsible in identifying and defining the applications and data that support an enterprise's business architecture. These data and applications should be understood by all the stakeholders.

- 5. **Phase D: Technology Architecture phase** maps the applications and data components described in phase C to the set of technological components representing software and hardware components. It will also have the implementation and migration planning. It also supports the cost assessment for particular migration scenarios.

- 6. **Phase E: Opportunities And Solutions phase** is responsible to review the target business objectives and capabilities, reviews the enterprise's current state to decide if the changes can be incorporated or not.

- 7. **Phase F: Migration planning phase** will finalize the implementation and migration plans. It checks if these plans are in accordance with the various frameworks that are in use. It prioritizes all the work packages, projects and building blocks assigning business values to each. It creates, evolve and monitor implementation and migration plan in detail. With respect to the agreed implementation approach, architecture vision is finalized.

- 8. **Phase G: Implementation Governance phase** governs and manages overall implementation and deployment process. It ensures if the deployment is successful.

- 9. **Phase H: Architecture Change Management phase** assess the performance of the architecture and recommend the necessary changes, if any. It assesses the changes to frameworks and gives the procedures for incorporating any changes to the new architecture.

- 10. **ADM Architecture Requirements Management phases** uses ADM to examine the process of managing architecture requirements. After identifying the requirements they are stored and fed into and out of the relevant.

These phases are iterative in nature both between phases and within phases. These phases are further divided into numerous steps to accomplish business process. Certain kind of output is generated after every phase which can be modified in an early phase or later phase.

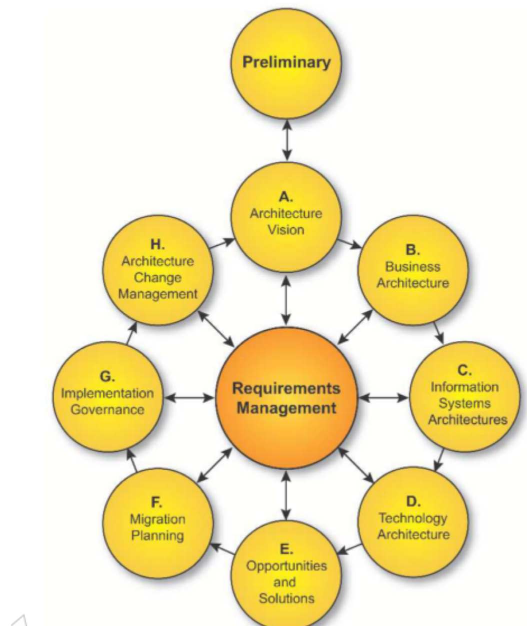

Figure 1.6: Architecture Development Cycle
[Gro09]

1.4 Conclusion and Future Work

Modern era of business is growing with much faster rate. In the present scenario, computers are the backbone that is used in optimizing the cost and performance to a level that is satisfactory and impressive to the users of the system. Enterprises have become much more organized than before and are focused on delivering high quality application landscape. In this paper I have described various concepts of Enterprise Architecture and its importance to the enterprise. Moreover we have also discussed different architecture facets and their roles. As modelling play an important role in designing architecture, some of the key features of modelling are also depicted in this paper. Enterprise Architecture Framework (EAF) is another area that we have focused on. One among EAF is TOGAF developed by Open Group. Architecture Development Methods and the benefits of TOGAF are also described in this paper. However, a detailed research can be done to identify the issues associated with TOGAF and the modelling techniques. Moreover, there are many frameworks available and thus the scope of this paper can be extended by including the detailed comparison between various frameworks. This will give a better understanding to enterprises and will give companies an option to choose the best framework depending on their business needs.

Bibliography

[Amb04] S.W. Ambler. *The object primer: Agile model-driven development with UML 2.0*. Cambridge University Press, 2004.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[Cam09] Adrian Campbell. `http://iea.wikidot.com/planningit-meta-model`. July 2009.

[EP00] H.E. Eriksson and M. Penker. *Business modeling with UML*. John Wiley & Sons, 2000.

[Gro09] The Open Group. *TOGAF Version 9*. Van Haren Pub, 2009.

[Her08] Wolfgang Herrmann. `http://www.computerwoche.de/a/so-finden-sie-das-richtige-eam-tool,1868156,4`. July 2008.

[IAS12] An Association of all IT Architects IASA. `http://www.iasaglobal.org/iasa/Resources.asp`. December 2012.

[IBM13] IBM. `http://pic.dhe.ibm.com/infocenter/rsahelp/v8/index.jsp?topic=/com.ibm.xtools.bpmn.diagram.doc/topics/c_bpmndiag.html`. January 2013.

[Kob98] C. Kobryn. Modeling enterprise software architectures using uml. In *Enterprise Distributed Object Computing Workshop, 1998. EDOC'98. Proceedings. Second International*, pages 25–34. IEEE, 1998.

[Lan99] Marc Lankhorst. *Enterprise Architecture at Work: Modelling, Communication and Analysis*. Springer, December 1899.

[LH13] PatrickJ. Lynch and Sarah Horton. `http://webstyleguide.com/wsg3/3-information-architecture/4-presenting-information.html`. January 2013.

[Man13] Information Management. `http://www.information-management.com/channels/data-architecture.html`. January 2013.

[Mic12] Microsoft. `http://msdn.microsoft.com/en-us/library/bb402960.aspx`. December 2012.

[Mul00] G. Muller. The system architecture process, 2000.

[Nuc10] Nucleus Research Inc. GUIDEBOOK ALFABET PLANNINGIT, 2010.

[oHEA11] National Institue of Health Enterprise Architecture. https://enterprisearchitecture.nih.gov/pages/what.aspxg. December 2011.

[OST05] ALEXANDER OSTERWALDER. http://www.businessmodelalchemist.com/2005/11/what-is-business-model.html. November 2005.

[QFTX09] K. Qian, X. Fu, L. Tao, and C. Xu. *Software architecture and design illuminated*. Jones & Bartlett Learning, 2009.

[Rou05] Margaret Rouse. http://searchcio.techtarget.com/definition/business-process. September 2005.

[Ser] Generali Informatik Services. Enterprise architecture management (eam) bei der generali deutschland informatik services (gdis) ein erfahrungsbericht. May.

[SK07] H. Shah and M.E. Kourdi. Frameworks for enterprise architecture. *IT Professional*, 9(5):36–41, 2007.

[UM06] L. Urbaczewski and S. Mrdalj. A comparison of enterprise architecture frameworks. *Issues in Information Systems*, 7(2):18–23, 2006.

[Wis13] WiseGEEK. http://www.wisegeek.com/what-is-a-business-strategy.htm. January 2013.

[Wol09] Wolfgang W Keller. TOGAF 9 Quick Start Guide for Enterprise Architects, 2009.

[Zar11] E. Zaroff. A case for designing information architecture around business goals & strategies. *Review of Business Information Systems (RBIS)*, 11(4):77–90, 2011.

Chapter 2

Software Architecture Representation

Rittika Bhattacharyya

Contents

2.1 Motivation and Overview 22
2.2 Basic concepts 24
2.3 View-based models for the software architecture description . 28
2.3.1 4+1 view-point model 29
2.3.2 Siemens 4 View Model 30
2.3.3 Avgeriou and Zdun models 32
2.4 Reference Architecture 33
2.5 Architecture Description Methods 35
2.5.1 Unified Modeling Language (UML) 35
2.5.2 Architecture Definition Language(ADL) 37
2.6 Basic Architectures 39
2.6.1 Multimedia architectures 39
2.6.2 Peer to Peer Architecture 40
Bibliography 41

Abstract: With the recent developments in the software industry, software companies have to delivered large number of projects in short amount of time yet successfully. So it is important to do proper documentation, architectural models should be considered, reference architectures should be well studied and architectural description methods must be followed properly.The primary reason for this is not the end result of the implementation or the lines of code but on the overall structure. Architectural representation is one of the primary goals for the development of a successful system. In this paper we strive to achieve and describe the various ways we could achieve the above mentioned facts.

2.1 Motivation and Overview

Today's world is mainly concerned with product development. Product development and its sale involves huge amount of money. So it is very important to have a perfect plan that would lead to a successful delivery of a product, i.e., a good architectural design that would make a huge impact on the sale of the product. So the idea is to have a good architecture and follow these architectural guidelines to develop a good successful product. We could never imagine building our house without an architect, could we? Similarly how can we ever imagine or hope to think that a software product could be delivered successfully without software architecture being carefully considered. This is because Software Architecture acts as a blueprint for the system and the project developing it.

"Software architecture deals with abstraction, decomposition and composition, and style and aesthetics. It also deals with the design and implementation of software's high-level structure"as said by Philippe B.Kruchten [Kru95]. So it is very important to understand how the system will be structured, deployed and understand its behavior.

This paper is divided into five chapters.So later in this paper we see the importance of software architecture and try to understand its main concepts. Important definitions and their and their concepts are explained.It is seen how these concepts are implemented and architecturally represented.Needs for architectural description and aspects of software architecture are also spoken about.The next major section in this paper are the view models.Three view models have described elaborately.The first is Kruchten's 4+1 view model.All the five views of this model are discussed in length.Then we have Siemens 4 View Model.And later Avgeriou and Zdun model is explained.At the end of this section it is seen how these models differ to each other.The next section is the Reference Architecture.Here the concept of Reference architecture is explained.Few examples are described and its importance is also talked about.The next section is on UML and ADLs.Description,importance and usability of both UML and ADLs are explained.It has been seen how these architectural description methods play an important role.At the end of this paper

two basic architectures have been discussed.They are the Multimedia architectures and Peer to Peer Architecture.

1. Definition of Software Architecture
   "*Architecture*" comes from architectura a Latin word. Generally the word Architecture often refers to building architectures. But Architecture in its broader sense has far more coverage. Here in this paper Architecture would generally mean Software Architecture. Software Architecture can be described in so many different ways. This paper describes Software Architecture to be a blueprint for how the system is structured, works and behaves and how is the system deployed in its various environments.

   A system often consists of a lot of components and each component is designed in such a way that they have a specific relation to each other.Components are functional and reusable blocks having application specific services. Components are described in more details late in this paper.

   Though there are a lot of famous Software Architecture definitions, the definition by Bass, Clements, and Kazman, fits best to my understanding."The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements details having to do solely with internal implementation are not architectural" [ea].

2. Importance of Software Architecture

   (a) A correct architecture paves the way for a successful system and a wrong architecture might lead to an absolute disaster. So it is very important to consider every minute detail before laying the first stones for the architectural design. The earliest design decisions play a very critical role in the successful development of a project. These decisions are hardest to make as a lot of things have to be considered like the cost, the time, the schedule etc. When these decisions are finally made, it becomes more difficult to decide whether these decision considered are correct. Because once these decisions are taken and work has begun, it would be the very difficult to change. This is the period where constraints are decided. The structure of the system is made and system qualities are predicted. So failing to consider these key scenarios could put the developing system into incomprehensible long term trouble.

   (b) Specific scenarios with its specific requirements should be given key preference. Systems should be developed with both the user goal and also keeping the business in mind.

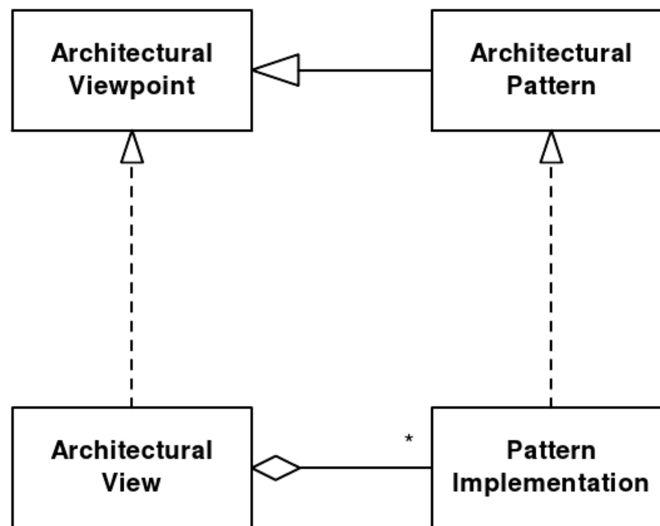

Figure 2.1: Classification of architectural patterns according to views [AZEE]

(c) Another reason why architecture is important is because it enables evolutionary prototyping.

(d) It bridges the communication gap between stakeholders. It helps in giving more accurate schedule and cost estimates.

(e) The complexity of problems is divided which aids in better understanding of the problem. Architectural design always plays a strong function in defining the success of a complex software system

2.2 Basic concepts

1. Basic Architectural Concepts

   Software Architecture also helps in the clear understanding between stakeholders. If we consider large scale architectures, the concept of complexity plays a major role. So the question now is what actually a large scale architecture is? The easiest way would be to say that large scale architectures are architectures that have more complexity associated with them. Some projects can be really big. Big projects introduce large amount of complexities which can sometime go beyond design, algorithms and data structures. Minor changes in requirements could lead to major changes in the project. Modules in the system are related to each other and sometime have overlapping functionality. It might happen that changes made to a particular model triggers some other functionality of a different module. So good choices based on the functional and

non-functional requirements have to made. So a small change in a business plan could make a big difference to the underdevelopment of the system. The developing system might not be quick enough to adapt to the changes.

So we see that there are both technical and business complexities. The challenge is how we could bring it together to solve the overall complexity and deliver a successful project. Software terminology like View, Viewpoint, Stakeholder, Concern, and Structure are the key features of this paper. They have been elaborately described later in this paper.

2. Definitions of a software architecture description: Definition and Basic notions.

   A Software Architecture Description is not just an artful description of design but has far more depth to it. It is a set of descriptive models grouped into views. Each view emphasizes certain architectural aspects that are useful to different stakeholders and for different purposes [PW92]. There are many types of views. Each view represents some kind of goal. It would be therefore wrong to focus on one specific view to describe the overall system. Developing a good system would require the system to have many quality attributes like extensibility, reliability, security, usability, fault-tolerant, maintainability etc. Therefore we can say that the quality attribute that is very important for you as well as the other stakeholders in the system's development should be the choice of what views to document. It is a disadvantage that architecture cannot be fully represented by a single view. But looking at it in a different way, we could also say that each view on its own respect is a strength as it ignores all the other aspects. So the current problem at hand is more scalable and tractable. 2. Definition: ”A view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders” [Arc]. Views represent certain aspects of the system, each addressing a set of related concerns.A view usually corresponds to a given stakeholder's viewpoint. Definition: ”A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views” [Arc]. A viewpoint on the other hand suppresses the major details of the system and provides a more simplified version that is easy to understand. For example, the viewpoint model for a fire alarm focuses on fire concern only. It is involved with a specific concern that if there is fire there would be an alarm. This viewpoint model contains those elements that are related to security from a more general model of a system. A viewpoint is able to point out the to the different system stakeholders their respective concerns.These concerns are addressed, which help in the modeling of languages and modeling of techniques that are used to create views.So an architectural

description is made up of views. It could have a single view or multiple views depending on the system. Each of these views comprise one or more models that conforms to a viewpoint. One or more of the viewpoints is selected and each of which covers one or more stakeholder concerns."Views are specified by viewpoints" [ICC06].

3. Creating an architectural description: There are many ways of creating an architectural description. One way to create an architectural description is to compose a set of views which address the stakeholders and their respective concerns and which also models few aspects of the system.The customers play a crucial role. They will be the future "users" of the system. Their roles and their architectural concerns should be thoroughly discussed. Other stakeholders include the developers of the system, the maintainers of the system, and the acquirers of the system. Inconsistencies among the views should also be meticulously recorded. A rationale for the architecture is lastly created. It should explain the extent to which the concerns of stakeholders are covered. Additionally The IEEE 1471 recommends that architectural descriptions include Summary, Scope, Context, Glossary, References etc. Therefore it is quite hard to achieve it.

For an existing system, the architectural description sometimes need to be reverse engineered, especially if architectural description does not exist for the system. At the end care should be taken to see whether they were satisfied, and whether the time limit to decided at the beginning was exceeded, and check that the quality was rightly measured i.e. quality attributes can be expressed as system concerns if they are defined explicitly.

3. Needs and uses of Software Architectures descriptions:

   When starting to design an architecture for the system, it is often seen that that there are a lot of questions to be considered. The easiest way would be to design a model that encompasses all the answers to the questions we have in our mind. But this would lead to a complex, incomprehensible model. The stakeholders would also disapprove of such a model as they would have to struggle and try to understand the complex model and find out their interest. This is not desirable. Also to achieve a model that satisfies all these is practically impossible. We need to represent the model in a comprehensible and simple manner. We also need to address the stakeholder issues in such a way that they are satisfied. One commonly used approach is to break the problem into a number of pieces and try solving them one at a time. The architecture is then broken down into different views. These views are separate but could at times have related concerns.

4. Different aspects of Software Architecture

   Functional requirements generally tell us how the system would react to a particular input or in some specific situation. However, the factors that

Figure 2.2: Classification of architectural patterns according to views [CMR+03]

really influence the way the final architecture should look like are actually the non-functional requirements, such as: re usability, comprehensibility, functionality, stakeholder concerns, technological constraints,performance to name a few.

Nonfunctional requirements are not related to system behavior but to constraints such as time, reliability,response time, process, standards and many more. Sometimes nonfunctional requirements are more critical than functional requirements. These requirements should be fulfilled else it might render the system to be useless.

The non-functional requirements can be addressed at various levels of the system's architecture, such as:

(a) Application Architecture:
It is very important to keep in mind that partitioning of responsibilities should always be done. There is a very high chance that concerning responsibilities might get mixed up. For example business logic should be separated from the details of presentation

(b) Data Architecture:
It is very important to maintain a correct order of who owns which data in an organization.
Data Architecture defines the rules which governs the way the data is collected and stored in an organization. Depending on the organization the data can be stored in a single place often referred to as "capture-all" system whereas in other cases the data is stored in separate places depending on it's type. The rules also define how

the data is retrieved depending on the size of the organization. In some cases the data retrieval is automated where there are new systems which capture all activity data in a single place. In other cases the data may be retrieved manually from various sources [IAT].

(c) Technical Architecture:
It is a good practice to choose what technology will be used to support the system [May05].

2.3 View-based models for the software architecture description

In Today's Hi-tech world one way to describe a complex architecture is to use a single, overcrowded model that encompasses all functionality and features as a whole. But this proves to be ineffective, as it is difficult to understand and maintain. A more effective way would be to break the architecture into different views addressing each view as a separate entity. The new challenge is to keep these separated but interrelated views consistent one with another.

What a is view-based model?
We have already seen that a view-based model is an important part for software architecture. Typically a view corresponds to the different concerns of a given group of stakeholders. In a lot of research papers a number of models have been described that prescribe what set of views best describe a given system. In this paper the three basic view models that have been proposed before are explained in detail. They are the 4+1 view-point model, Siemens 4 View Model and Avgeriou and Zdun model.

Advantages of employing such view based models are important for the development of software architecture. This is because they are created based on id-of their authors, best-practices and on the long experience of their authors. Every model has a specific set of views which is relevant for that model only. So not all view is important for all systems or stakeholders. Every model tries its maximum to address most of the possibly relevant concerns. But none of them are able to provide full coverage . So an "optimal set of viewpoints"' was devised that "'providing complete coverage and had a greater coverage than any of the individual viewpoint models" [May05].

2.3.1 4+1 view-point model

Description
Architects struggle to portray all architectural features together on one document i.e. the "blueprint". They try to pen down the gist of the to be developed system all at once. To approach this problem Kruchten has developed the 4+1 model [Kru95]. The model used five concurrent views to address specific set of concerns to different stakeholder in the system. Each view is a "'Blueprint"'. It uses its own notation. These five views are as follows

1. **Logical View** is concerned about the functional requirements of the End-users in the system. It is mainly concerned with the services that it should provide to its customers. The developers are mainly concerned with this view. The logical view can be modeled in UML using sequence diagram.

2. **Process View** "captures the concurrency and synchronization aspects of the design" [Kru95]. To model the process view, in UML, the activity diagram can be used.

3. **Physical View** depicts the correlation of software onto hardware and its reflection to their distributed aspects. Here process and software modules are mapped onto hardware. In every new iteration new scenarios are modeled. This continues until the system becomes stable. This happens when no new process, sub-process or inter-process are found. The physical view is a system engineer's main focus. To model the Physical View, in UML, deployment diagrams can be used.

4. **Development View**'s main focus is on the different modules and submodules. Software Managers and programmers are mostly associated with development view and its minute details. It is a layered topology. To model the Development View, in UML, component diagram can be used.

5. **Use case or Scenario based View** considers different scenarios. System consistencies and validation are done. Notations are similar to logical view. This fifth view is the essence of the 4+1 view model and it is mandatory. Here the functionality of the system with its users and external interfaces are described.

Usability [00]

1. Scenarios view is for Understandability

2. Logical View is for Functionality

3. Process View is for Performance especially for programmers

4. Implementation View is for Software management

5. Deployment View is for System topology, Delivery, Installation

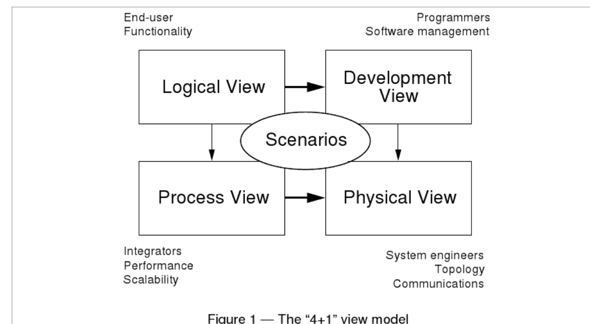

Figure 2.3: Flowchart representation of View and Viewpoints [Kru95]

2.3.2 Siemens 4 View Model

Description

1. Founder
   Siemens Four Views approach to software architecture was adapted from Hofmeister.

2. Concept
   Siemens 4 view model is also used for architectural documentation. This view model uses four views. These views are the **Conceptual view, Module view, Code view and Execution view.** For each view, a set of tasks that need to be performed are given. A very important fact in this model is that all views have *global analysis* as the first common task. The next tasks, namely the *central design* and the *final design* are specific to each view. purpose of global analysis is to rectify problematic issues from the very beginning. It is used in early identification of issues so that strategies could be formulated to solve the problems. Activities of global analysis include

   (a) Analyze Factors
       Helps in identification of factors that influence the system architecture Factors include Organizational factors, Management Factors, Technical Factors etc. These factors are then further refined until proper and correct analysis is done.

(b) Identify Issues
Issues that influence the architecture should be resolved. All sorts of influencing factors are listed and considered. Strength and weakness of each factor is consider and proper techniques are applied to resolve them.

(c) Develop Strategies
Strategies is the process by which a solution to problem is found. Here strategies are formulated to address the above issues and an attempt is made to identify related strategies.

Usability The purpose of Conceptual View helps in the documentation of both structure and behavior. Structure include Configurations of components, connectors, ports, roles and protocols. Behavior includes State diagrams and Sequence diagrams. So this view describes the architecture in terms of domain elements. Here functionality is also designed. Activities of Conceptual View include the following

1. Global analysis: Further evaluation done on system architecture.
2. Central design tasks: Conceptual components and connectors, are created and global evaluation is done.

The purpose of Module view is that it helps in the top level grouping of activities. here the system is decomposed into layers. Module view activities include:

1. Global analysis: Again further evaluation of design decisions.
2. Central design tasks: Subsystems, Layers, Modules are defined. These are then allocated with conceptual elements and then again global evaluation is performed.
3. Final design task: Modules interfaces are designed.

The purpose of Execution view depicts the run-time view of the system. It is the mapping of modules and components to run time images. This help in building Communication paths. Execution view activities include:

1. Global evaluation:Further evaluation done again.
2. Central design tasks: Here the run time entities like tasks, processes, threads are defined. Communication paths are established and later global evaluation is done as before.
3. Final design task: Here Resource allocation is done. Resource usage and performances are key factors of the execution view.

The purpose of the code view is to define intermediate components and show their organization and dependencies with each other. It shows the mapping of modules from the Module View to source components and also the mapping of run-time entities from the Execution View to deployment components. Here similar activities are performed like the above view. This model was also designed to reduce complexity of large systems aimed to aid in separation of concerns as mentioned in the 4+1 view model

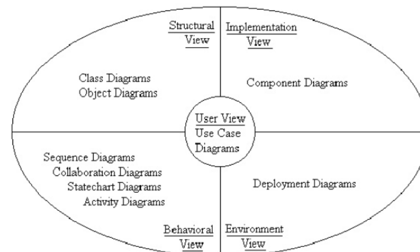

Figure 2.4: Different Diagrams and views supported by UML [Mal00]

2.3.3 Avgeriou and Zdun models

Description

1. Concept
   This model is an attempt to find a pattern language that is a super-set of ”existing architectural pattern collections and categorizations” It mainly focuses on the establishment of a relationship between the patterns. A pattern is considered to be an architecture pattern if it covers the overall system and not just few parts of the subsystem. A classification of architectural patterns are stated with regards to the views previously stated in the 4+1 view model. This architectural pattern schema is based on architectural views concepts. We already know what a view and a view point is from the other view models. So we directly come to Architectural Pattern. Architectural Pattern concentrates on the relationship between different elements that help in solving a particular problem.It could be considered to be the ”specialization of a viewpoint since it provides specialized semantics to the types of elements and relationships,as well as constraints upon them” [AZEE].

2. Aim of Avgeriou and Zdun models model and its Usability
   They have strived to conjugate existing approaches of architectural pat-

terns into a pattern language. This would help practitioners in finding a single comprehensive source of patterns.

3. Comparison with other models
   This model is quite unique in my opinion. Unlike the other view model this model does not stress on stakeholder issues and separation of concern and the interactive development. Avgeriou and Zdun model has tried to make a common reference platform that would enable developers to follow a single comprehensive source of patterns.

2.4 Reference Architecture

Definition and a brief description: Many a time it has been seen that an architect spends immense amount of time pondering over architectural design. The only reason could be improper documentation. So we could say documenting their work could build the perfect reference for future work. ”Reference architecture for a domain captures the fundamental subsystems and relationships that are common to the existing systems in that domain” [GG06]. This means that we can use this as a reference or as a template for future use. It is used like a template for designing new systems in any place where reuse is necessary. Reference architectures aid in better understanding of the system. It can used to educate amateur software architects . Software developers who are new to a particular system can break the system into different blocks and concentrate on the problematic block rather than the whole system. Reference architecture also helps in improvement of the communication between different software developers as one developer explains a particular concept to another developer who was unable to understand that part. Having reference architecture during the maintenance and design phase helps a lot in proper understanding of systems. It can also show the faults between different design models. Lastly it can help in re-engineering new models as well.

Few examples of reference architectures are given below.

1. Enterprise Reference Architecture Enterprise Architecture is model based.It helps in increasing the consistency and agility while maintaining the changes in the enterprise. The enterprise reference architecture is a framework that was developed to evaluate existing frameworks for enterprise integration.

   According to [Doc03]the characteristics of EAR are:

   (a) Service-oriented
       Application are broken down into small parts called services.These

services are accessed by systems and other applications based on similar functionality. ”Service-orientation is a design paradigm comprised of a specific set of design principles. The application of these principles to the design of solution logic results in service-oriented solution logic. The most fundamental unit of service-oriented solution logic is the service”c.

(b) Event-driven
An event is a change in state that lead to a particular action being triggered. Architectural designs have components and services coupled together. So with the help of event driven functions we get more responsiveness from the system. The underlying systems should support them. These small changes could make a big difference in the business world. So the system should be well equipped to adapt to it. Event-driven design and development allows easier development and maintenance of large distributed applications and services involving unpredictable and asynchronous occurrences. It also allows reassembled and reconfigured existing applications.

(c) Able to support assembly and integration
For better management applications are broken down to small functional part which are essential for the system. This was previously achieved with help of codes done by developers. This was really expensive. Now ”process management technology” can be used instead at a lower cost.

(d) Aligned with life-cycle support processes The life-cycle development is a very important part for the developing system. A system undergoes constant iterative evaluation. The life cycle phases includes the Requirements Analysis, Architectural Design, Code Construction and Unit Testing, System Testing and the Maintenance phase. During these stages the models undergoes constant iteration. Few well known like cycle models include the Waterfall model,Spiral model and many more.

(e) Able to leverage existing applications and infrastructure Nothing could be better that reuse of existing technology.

2. Organization for the Advancement of Structured Information Standards (OASIS) OASIS is a reference architecture for building SOA applications. SOA plays a very important role when designing systems that are meant for transaction like e-business, end to end transaction etc. Here the business application is broken down into small pieces or block. These blocks are called services. They are well defined business functionality. These services are independent of applications and the computing platforms on which they run with the help of SOA. These services can later be reused for a different need completely. Reconfigurable architecture that are dynamically provided by SOA enable enterprises to respond quickly and flexibly to market changes which ultimately helps in the betterment of business.

2.5 Architecture Description Methods

2.5.1 Unified Modeling Language (UML)

Description:
Unified Modeling Language (UML) is quite popular with the computer world. It has become one of the most popular standards for modeling object-oriented software. UML is one of the most popular modeling languages. It makes use of a set of notations. It can use rectangles, boxes, lines to create models of systems. A model can be said to be an abstraction of the systems. Models also help in the analysis and design procedure of software systems. "The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems" [UML]. It is a graphical language. It utilizes most of the Object Oriented concepts like Abstraction, Composite object, Polymorphism, Encapsulation and Dynamic Binding.

Different types of UML diagram [QE03].
UML 2 is very comprehensive. It is a language and not just a diagrammatic notation. It analyses the problems and designs a solution with the help of a model. It is very important to create a model as it captures all important application aspects while abstracting the rest. Construction of the model helps in Analysis, Specification, Code-Generation, Design, Visualization and Testing of the system as well. UML can be divided into different Diagram Types according to its functionality.

1. Structural Diagrams
   These diagrams generally focus on the static aspects of the software system. It is also called the static model as it does not change with time. Here different types of classes are defined for understanding the proper working of the system and its implementation.

2. Behavioral Diagrams
   These diagrams generally focus on dynamic aspects of the software system. It captures how object interact with each other and their reaction with respect to each other. It represents the time-independent or dynamic behavior of the system. It makes use of Use-case, Interaction, State Chart, and Activity Diagrams.

Comparison between various UML models

1. Structural Diagrams include

   (a) Class Diagram
   It represents the static structure of the system and on how it behaves. It consists of a set of classes and their relationships and dependencies. It describes interface to the class. It uses concepts of generalization, association and aggregation.

   (b) Object Diagram
   It shows the instance of a class of the system at a point in time.It can also be called the instance diagram as it shows the instance rather than the class itself. An object diagram undergoes continuous changes as development proceeds.

   (c) Component Diagram
   It shows the logical groupings of elements and their dependencies on software components.

   (d) Deployment Diagram
   It is a set of computational resources (nodes) that host each component. It also maps the software implementation components on the hardware like printers, scanners etc.

2. Behavioral Diagrams

   (a) Use Case Diagram
   Use Case Diagram depicts the set of ”Use Cases” that the users can use. This case consists of high-level behaviors of the system. The Use Case partitions the systems into a number of transactions in such a way that each transaction performs some useful actions from the user’s point of view. These transactions might involve single message or multiple messages being delivered between system and users. The external entities are called actors .Use Case Diagram represents all the functional requirements of the system.

   (b) Sequence Diagram
   It shows the interaction of objects in a two-dimensional chart. The chart is read from top to bottom. The interacting object is shown on top as boxes attached to a vertical dashed line. Inside the box the name of the interacting object is stated with a colon .The colon separates the name of the object from the class and of names are underlined. It focus on time ordering of messages.

   (c) Collaboration Diagram
   It focus on structural organization of objects and messages. It bring together both the static and dynamic behavior of the system as its represents the combined information taken from class, sequence, and use case diagrams

   (d) State Chart Diagram
   It is used to depict the changes made by an object in its lifetime. It describes best the behavior of an object over several use case

executions. However several objects collaborating together cannot be appropriately depicted by State Chart Diagram. State Chart Diagram is based on Finite state machine (FSM).So every finite state corresponds to an object being modeled. So an object undergoes a change only when a single specific change occurs.

(e) Activity Diagram
It shows flow of control between activities. It represents activities or process which may or may not correspond to the methods of classes. It is similar to procedural flow charts.

Importance of using UML

With the help of UML life of software engineers became far easier. It could be used to design details without it being too complex or incomprehensible. One very important fact about UML is that any kind of application can be designed with the help of UML. It is not platform or language dependent. It is also hardware independent. So working with UML gives a wide scope for development. UML can be also used to build complex systems as the different diagrams helps in different activities of the developing system. UML also helps in the modeling of a Transactional, Real-time, and Fault-Tolerant systems in a natural way. UML is a most useful method for visualization and documenting software systems. It can also be used for modeling middle-ware. So we see that UML was a boon to software industry. There are also many disadvantages of UML. Many times UML is confused with a visual programming language though it is actually a visual modeling language. There are still no specifications for modeling of user interfaces.

2.5.2 Architecture Definition Language(ADL)

Description

Architecture Definition Language is the formal language that can be especially used to represent architectures of software intensive systems. It shifts the focus from lines of code to modeling notations that are used to support architecture-based development. ADL has gained a lot of attention in large system development industries. It is a computer language approach for the representation of architectures. ADLs are able to address the problems of informal representations. ADLs also help in the early analysis and feasibility testing of the design decisions [AG94]. We could say ”an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module” [MT97].

Characteristics of ADLs ADLs are quite different from requirement languages. This is because requirement languages describe the problem space whereas ADLs are rooted in solution space. The requirements are often partitioned into behavioral blocks that are easy to represent. ADLs focus more on component representation.

Important properties that ADLs

1. Minimum level of abstraction and encapsulation
2. Competency to represent connectors and components.
3. Effectively provide common architectural styles
4. Help in achieving architecture creation, refinement and validation.

There are different types of ADLs. Some of the most popular ADLs are

1. AADL (SAE standard)
2. Avionics ADL
3. Wright (developed by Carnegie Mellon)
4. Acme (developed by Carnegie Mellon)
5. XADL (developed by UCI)

ACME is one of the most common used ADLs. It was developed by Carnegie Mellon. Acme provides a fundamental basis for description and assessment of architectures of component based systems. It does not help in the formulation of models that states the behavior of a system. It is used to design and then understand a system with high level of abstraction. The goal of Acme was to provide a common tool that could handle the interchange between different architectural design tools and their architectural descriptions. Sometimes Acme can be expressed by component and connector architectural view. Components in Acme are the basic building block for the description of the system. So Acme defines its component types and their properties and then checks for constraints in architectural style. It also performs advanced analysis.

Important elements of an ADL are Component, Connector and Configuration. As seen from above that although there is a considerable difference in capabilities between the different ADLs, they share the same concept. The concept of Acme includes:

1. Components: These are the computational elements. Components may be used to describe elements of a variety of different computational models at varying levels of abstraction

2. Connectors: A Connectors is like an abstract ”mortar” between components. They represent the interaction between different components.

3. Systems: It is the representation of components and the connectors.

4. Ports: These are interface points for components

5. Roles: These are interface points for connectors [Kom].

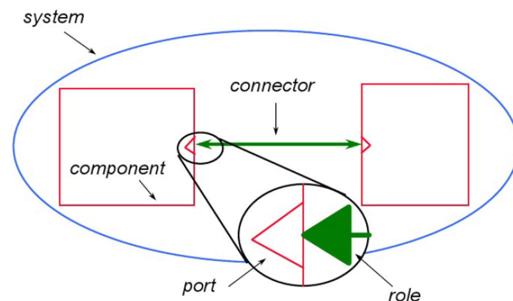

Figure 2.5: Block representation [GS]

2.6 Basic Architectures

2.6.1 Multimedia architectures

This type of architecture is designed to transfer multimedia data like audio and video files efficiently ensuring a predetermined quality of service. The main issues that these type of architecture deals with are as follows :

1. It should provide support for a continuous and seamless media such as audio, video and animation. This requires continuous data transfer over a long period of time.

2. Meeting the demand for the Quality of Service is of utmost importance in transferring multimedia data. This architecture helps in the classification of the contents into static and dynamic QoS management.

3. The architecture is designed in such a way that multimedia access and retrieval is done in a most efficient manner. Below we give a diagrammatic representation of real time Multimedia Architecture for mainly three types of systems namely Standalone, Server and Networked. This type of architecture is a layered architecture where the services of the lower layers are utilized by the higher layers.

2.6.2 Peer to Peer Architecture

This architecture is very different from the well known client server architecture. In this type of architecture all the components act as a client and a server at the same time. There are no central servers and the resources can be accessed directly from the other peers. The main reason for developing such an architecture is limited scalability , extensibility and reliability of the client-server architecture. This type of architecture is highly robust against failure of any node.

The architecture is scalable in terms of resources and computing power. Thus a decentralized self organizing system having a decentralized usage of resources helps this P2P architecture to deal with the weaknesses. This architecture is based on an overlay network which is used for indexing the nodes so that they are independent from the existing network topology. In this type of architecture if any node knows about the location about any other node , then there exists an edge between these two nodes in the overlay network. Thus in an P2P architecture the nodes are connected to one another though they may not be connected physically. This architecture can be broadly divided into two groups :

1. Structured
   The indexing of the nodes are done using distributed indexing and the connections in the overlay network is always fixed . Any terminal entity can be removed without loss of functionality. Examples of such architecture is Chord and CAN.

2. Unstructured : In this type of architecture the connections between the nodes are not fixed. The nodes connect in an Ad hoc manner and uses Flooding search for finding the nodes in the network. There is a flooding based search until the requested node is located. Once it is located, the nodes get connected in the overlay network though there may not be any connection in physical network.

   The Unstructured P2P architecture can be classified into the following categories. These are :

   (a) Centralized P2P - All features of P2P were included, Central entity provides the service by storing the the indexes of all the nodes in the architecture.

(b) Pure P2P - Any terminal entity can be removed without loss of functionality. There are no central entities to store the indexes of the nodes.

(c) Hybrid P2P - In this type of architectures there are dynamic central entities. All the features of P2P are included and any terminal entity can be removed without the loss of functionality.

Bibliography

[00] 0. Uml an overview. *0*, 0.

[AG94] Robert Allen and David Garlan. Beyond definition use architectural interconnection. 1994.

[Arc] Software Systems Architecture.

[AZEE] Paris Avgeriou and Uwe Zdun. Architectural patterns revisited, a pattern language. *2006*, IEEE.

[CMR+03] J. Champeau, F. Mekerke, E. Rochefort, et al. Patterns, aspects and views in mda process. 2003.

[Doc03] Doculabsi. Planning and building anarchitecture that lasts:the dynamic enterprise reference architecture. 2003.

[ea] Addison-Wesley Clements et al. Views and beyond,2nd edition.

[GG06] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, 2006.

[GS] David Garlan and Mary Shaw. An introduction to software architecture.phi learning pvt. ltd.

[IAT] Data architecture.

[ICC06] Stig Larsson Ivica Crnkovic and Michel Chaudron. Tahiti, french polynesia. 2006.

[Kom] Andrew Kompanek. Modeling a system with acme.

[Kru95] Philip B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[Mal00] Rajib Mall. *Fundamentals of Software Engineering*. 2000.

[May05] Nicholas May. A survey of software architecture viewpoint models. *IEEE*, 2005.

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. 1997.

[PW92] Dewayne E. Perry and Alexander L. Wolf. The 4+1 view model of architecture. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 1992.

[QE03] Terry Quatrani and UML Evangelist. Introduction to the unified modeling language. *Tech. rep., Rational Software, IBM*, 2003.

[UML] UML. Omguml q and a.

Chapter 3

Component-based development and architecture

Johannes Dohmen

Contents

3.1 Introduction 44
3.1.1 Motivation 44
3.1.2 Outline 45
3.2 Components 46
3.2.1 Basic concepts 47
3.2.2 Service provider and service consumer 47
3.2.3 Composition and binding 48
3.2.4 Development phases and binding types 49
3.3 Component models 50
3.3.1 Basic concepts 50
3.3.2 Architecture description languages (ADLs) 51
3.3.3 Concrete component frameworks 54
3.4 Conclusion 59
Bibliography 60

Abstract: Software components promise many advantages over reusable artifacts of traditional software development like functions or classes. Still component-based development is not widely adopted. This paper explains the main concepts of components and explores different approaches of component-based development through ADLs and component frameworks and provides a survey of their advantages and disadvantages.

3.1 Introduction

Almost from the very beginning software engineering had to struggle with the complexity that arises from numerous, non-trivial requirements a software product normally has to fulfill. The key to lower complexity in software engineering is decomposition, which should be considered while designing and implementing a software product. At first glance the introduction of high level programming languages facilitate decomposition in the sense of breaking down an large entity into smaller parts. High level programming languages provide functions and, in object-oriented languages, classes to encapsulate and by thus hide concrete implementations of specific aspects of the software product. This enables software engineers to fragment software systems into smaller parts which are easier to design, implement and test. But software components are not only sub-parts of software systems! The vital aspect of software components is their ability to be *composed* in order to create new products. As Szyperski et al. explicate to the point:

> ”Components are for composition.” [SGM02]

The composition of components introduces the need for components to publish not only their provided services but also their required services. Furthermore composition requires some kind of ”wiring” between the components without changing the components itself. In order to gain maximum flexibility components should not be bound to a specific programming language (required by a component framework or other components) and be distributable across a network.

3.1.1 Motivation

Software components as reusable software entities, which can be assembled into new software products, promise great advantages over traditional software development. Doug McIlroy proclaimed this already at the celebrated NATA conference of 1968 in Garmisch where the now famous term Software crisis was coined:

> . . . yet software production in the large would be enormously helped

> by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors. [Mci69]

Advantages for software development hoped to be achieved by software components include (without being limited to them):

Increased reuse
In theory components are the perfect software units for reuse. As components state clearly what services they provide and request, they can be used in many different composed software systems. This allows to implement and improve both the software system and the component independently. Components that are independent from programming languages and feature network transparency have much more capabilities in comparison to other software units like classes or modules.

Increased quality
With components being used in many software systems it is reasonable to expect a higher quality compared to software subsystems or parts like classes, procedure etc. which are only used in one software system. The improved component quality should increase the overall quality of the software system.

Shorter time-to-market
Using components instead of implementing each part of a software system from scratch saves development time. Generally this should lead to a shorter time-to-market for new products. Software modules like libraries, frameworks and so on have already achieved this effect. But components are expected to increase such savings greatly.

3.1.2 Outline

In the following section components are introduced using metaphors and a basic but widely accepted definition. It will be explained why it is useful to regard components from different viewpoints. In the third section component models are presented as the base of components. Afterwards six component models are examined, which forms the main part of this paper. Those component models are divided into component models in architecture description languages (Acme, UML) and concrete component frameworks (CORBA, JavaBeans, Java Enterprise Beans, COM). The presented component models and the current situation of component based development are reviewed and evaluated in the last section.

3.2 Components

One popular metaphor used to describe a software components is screws and nuts. The basic idea is to combine screws and nuts with other components to build more advanced objects. This metaphor emphasizes the importance of interfaces which form the abstraction from the tangible screws and nuts in a real scenario. The definition of the thread used forms the simplest interface for a screw-nut combination as shown in Figure 3.1. Actual screws and nuts from

Figure 3.1: A screw with a matching nut attached [Wik12]

different vendors and countries, even if consisting of different material, can be combined as long as both meet the requirements of the thread standard. Another metaphor might be of more value as the screw and nut metaphor ignores what the components contribute to the resulting object and what which dependencies to other components exist. Software components can also be compared to integrated circuits (ICs) with their input and output connectors (often called pins). Such ICs contain internal logics which require external data from its input pins in order to provide a service which is accessible at its output pins. The documented pin layout of an IC describes the input and output pins as shown in Figure 3.2. The presence of the input/output pins require some kind of *wiring* to connect an IC to the rest of the system in a sensible way. In a

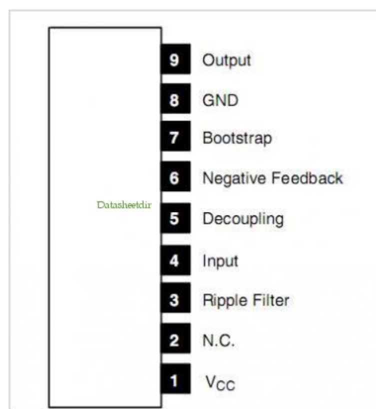

Figure 3.2: A pinout of an (audio) microchip. [dz812]

broader sense ICs both require and provide services.

3.2.1 Basic concepts

The prevalent definition by Szyperski et al. is a good starting point to advance from metaphors to the characteristics of software components:

> ”A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [SGM02]

This definition contains many important aspects of components already indicated by the two metaphors. First a component is a *software unit* and as such it can be designed and implemented largely independent from the rest of the system. It contributes to the system through its *provided services* (the ”specified interfaces”) and specifies its remaining dependencies as *required services* (the ”explicit context dependencies”) while both types of service form a contract to the residual software system. Moreover a component is a *deployable unit* which is used (together with other components) through *composition* to form a software system. Lastly it is explicitly stated that components do not have to be developed by the company which utilizes them but can instead be acquired from ”third parties”.

In summary a component provides services and requests services through interfaces. Moreover it is:

- an architectural unit
- a deployable unit
- a run-time entity

The ability to provide and request services is explored in the following. Subsequently the binding of components originating from the provided and requested services will be explained. The presentation of a life cycle which respects the components' trinity as architectural units, deployable units and run-time entities concludes this section.

3.2.2 Service provider and service consumer

As mentioned before components can provide and request services. When providing a service component A acts as a server to component B which by requiring the service acts as a client. It is possible that A additionally requests services from B which then leads to inverted roles. The assignment of server and client roles depend on the particular requests and are therefore not static.

Provided services
A component facilitates functionality to its clients through its provided services. This services are accessible through one or several interfaces. These interfaces can be seen as analogous to class interfaces but on the (conceptually higher) level of components. Actually class interfaces are used to publish the provided services in some component models (see Section 3.3).

Required services
A component may rely on functionality provided by other components as services accessible through their interfaces. In contrast to the aforementioned provided services programming languages in general do not provide a mechanism to specify required services or interfaces directly. Therefore, among other reasons, components are often shipped inside component containers which add the missing mechanism.

At most times a software component is seen as a black box that only reveals its provided and required services. Therefore a component is usually drawn as a (rectangle) shape with a distinct illustration of its services (commonly depicted as some kind of plugs) as shown in Fig. 3.3.

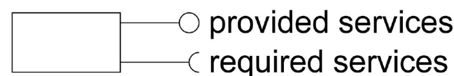

Figure 3.3: A software component. [LW07]

3.2.3 Composition and binding

The ultimate reason for component use as aforementioned is composition. In order to compose a software system from components it is necessary to bind their required and provided services in a sensible way. The required and provided services of each component are described through incoming respectively outgoing interfaces. These interfaces form the connectors which enable connections between different components in order to form a composite. Fig. 3.4 shows (in a slightly different notation as in Fig. 3.3) the relationship between components, interfaces and connections. An incoming interface of a required service must be connected to an outgoing interface which provides at least the required functionality (but possibly more). The act of establishing connections between components is called binding and can take place in different phases of the component life cycle.

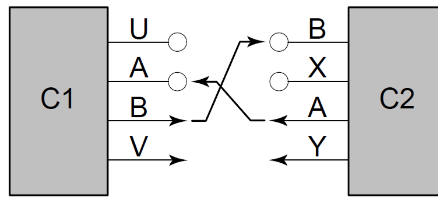

Figure 3.4: Connections, outgoing and incoming interfaces. [SGM02]

3.2.4 Development phases and binding types

Component life cycle respect that components are units in the design, the deployment and run-time phase. Lau and Wang [LW07] propose a life cycle that focuses on the binding of components in the different phases (as shown in Fig. 3.5).

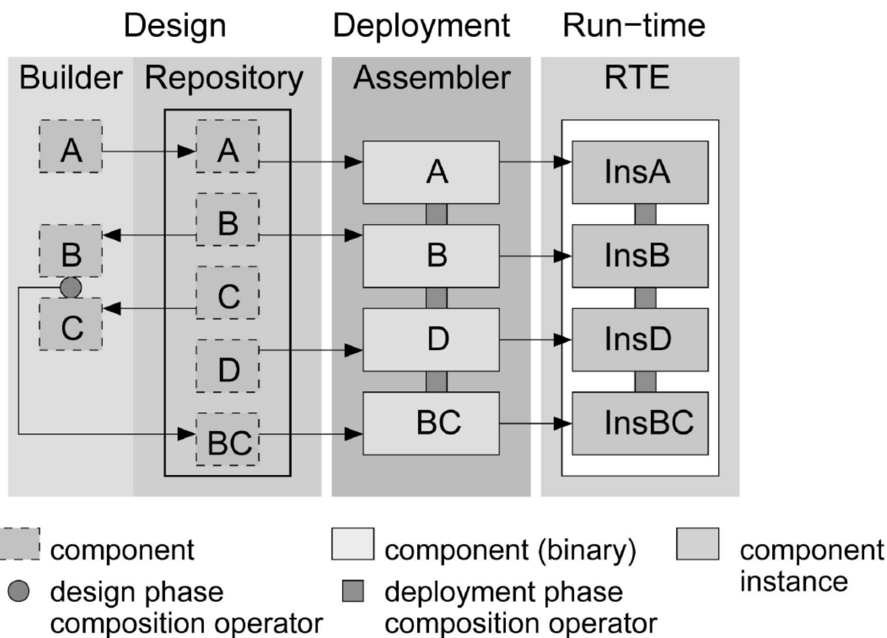

Figure 3.5: An idealized component life cycle. [LW07]

The repository holds all available (developed or acquired) components either in source-code or in a compiled binary. Two different tools are used in the design phase: The builder is used to create a composite component from the available components in the repository at this early phase. These new components are then stored in the repository. The assembler accesses the repository to retrieve the needed components, and compiles them if necessary, in the following deployment phase. The assembler than can wire the components in

a reasonable way. The component can connect themselves during run-time using functionality provided by the environment, this is the last phase of the life cycle. In this idealized life cycle composition of components can occur in all three phases.
Binding is the act of connecting components and as such the central operation of composition. It is imperative to distinguish in which phase the binding between components happens. The most static approach is to bind components in the design phase, which is also called *wiring*. This gives the opportunity to check whether all required services of the components are bound in a sensible way. However this approach is also the least flexible one. The next type of binding takes place in the deployment phase. Here an assembly tool is used to (often graphically) connect the components, which is referred to as *visual component assembly*. The last approach is to not really bind the components at all but to bind outgoing and incoming interfaces. This approach demands an run-time environment which sets up connections between components with matching interfaces or intercepts communication attempts and forwards them to all components with the matching interface. This very flexible approach is called *connection-oriented binding*.

3.3 Component models

The theoretical foundation to build component frameworks is a component model. In order to simplify the introduction of components the definition and description of components from Section 3.2 did not rely on component models. Many component frameworks still use only an implicit component model as the need of component models was not initially identified. Nevertheless component models are required to exactly describe a component as well as to implement component frameworks like COM, EJB and others. Component models generally fulfill two tasks, as stated by Heineman and Council:

> ”A component model operates on two levels. First, a component model defines how to construct an individual component.
> ...
> Second, a component model can enforce global behavior on how a set of components in a component-based system will communicate and interact with each other.” [CH01]

3.3.1 Basic concepts

Since component models make the theoretical base it is reasonable to define component models first and then define components in the context of component models.

Definition component model

”A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.” [CH01]

Definition component with regard to its model

”A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.” [CH01]

The first definition states that the binding as the mechanism for interaction and composition is part of the component model. The implementation of a component model also has to provide some kind of environment to execute the components. Such an environment can include additional services like a naming or directory service to identify components by name or attributes or a service to provide transactional contexts for the components. The second definition emphasizes that components have to respect the standard the component model has defined. It also notes the core characteristics (deployable, composable) of components in general as already discussed in the first section.

3.3.2 Architecture description languages (ADLs)

Architecture description languages are used to describe and develop software system architectures on a very high level. They usually support components as the main entities to model systems (or subsystems). Most ADLs share common concepts also referred to as their ontology. According to Garlan et al. such ontologies generally comprise of components, connectors, systems, properties, constraints, styles [GMW00].

Acme

Acme Project claims that Acme is a ”simple, generic software architecture description language (ADL) that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools” [Acm13]. Acme was initially started to support the interchange of architectures between (other) ADLs and their respective tools. But over the time Acme has grown into a feature-rich ADL itself. Acme features both a graphical annotation and a definition language as its syntax. Fig. 3.6 shows a most simple example of the graphical annotation. Acme supports an overall of seven core entities:

Components

The basic building blocks which store data and contain business logic.

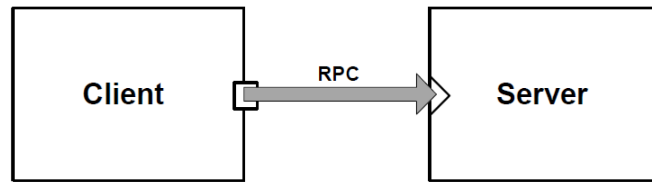

Figure 3.6: Simple Client-Server Diagram. [GMW00]

Ports
The interfaces for provided and requested services of a component.

Connectors
The connections between the components through which the connected components can communicate and interact.

Systems
The entities of the highest-level. Systems are build from components and their connectors.

Roles
The interfaces of the connectors. They define the participants of the interaction executed by the connector.

Representations
Architecture descriptions of a lower level. Every component or connector can have an *internal* architecture description. This enables a hierarchical description of the system including individual descriptions of its parts and subparts.

Rep-maps
Mappings of the internal ports of the individual representations into the external ports of component or connector.

Fig. 3.7 shows a *system* composed by *components* which interact through a *connector*. Additionally the *role* of the connector at the receiving end is depicted.

Listing 3.3 shows the same architecture definition as shown in Fig. 3.7 but given in Acme's formal language:

Apart from the aforementioned entity types Acme also supports properties that can be appended to all presented types. These properties are not interpreted by Acme but may contain valuable information for users or tools. Fig. 3.8 depicts entities enhanced with properties and representations.

Acme features a set of tools: AcmeStudio is an eclipse-based graphical design tool for software architectures based on the Acme annotation. Acme-Web is

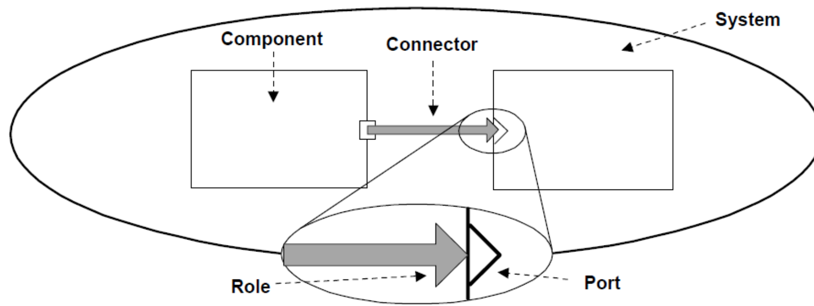

Figure 3.7: Elements of an Acme Description. [GMW00]

Listing 3.1: Simple Client-Server System in Acme. [GMW00]

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles {caller , callee} }
  Attachments : {
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee }
}
```

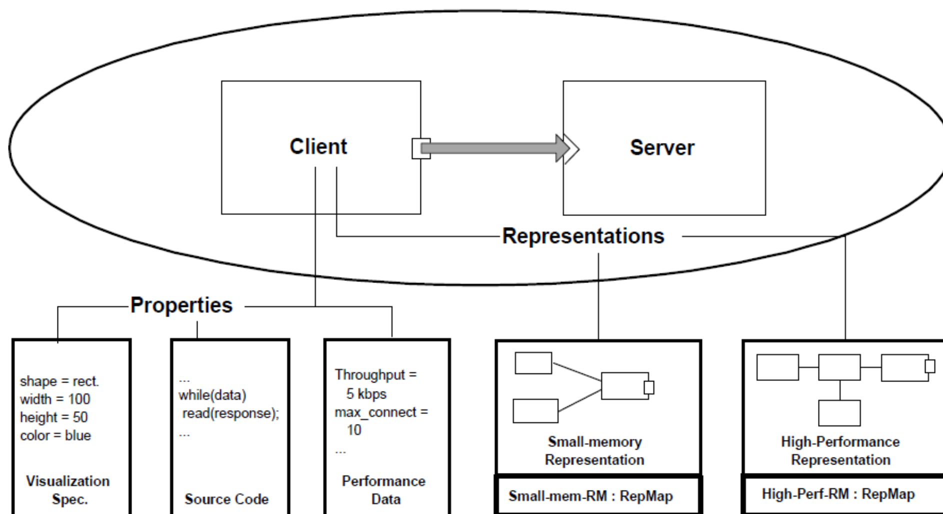

Figure 3.8: Representations and Properties of a Component. [GMW00]

a visualization tool which generates from a definition in the Acme language a graphical representation viewable with a standard web browser. AcmeLib offers a parser for Acme definitions and access to entities from a parsed definition.

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is currently the dominant domain-independent modeling language. Since its first release in the late 1990s UML contains components and component diagrams. That makes UML a promising candidate to function as an ADL as well. Using UML for defining architectures has two major advantages:
First, most people involved in software development (either academic or industrial) "speak" UML, i.e. they know at least the common UML features and annotations. That relieves the potential consumers from the necessity of learning a new modeling language.
Second, as UML is so widespread, it offers a very good tool support. This would enable system architects to use the same tools for modeling entities on a higher levels (architecture definitions) and lower levels (e.g. class diagrams).

However, despite the fact that UML "is a de facto standard general modeling language for software developments" [RKJ04] UML does not provide the complete feature set that ADLs commonly provide. UML's component diagrams are static in the sense that component instances cannot be replaced and connectors cannot be changed at run-time. Also connectors cannot have complex semantics or structures which are often needed in ADL definitions as the connector may implement a specific protocol. Furthermore UML is too flexible regarding components as they can consist of many types (e.g. classes) which is not desirable for the top-level view of ADLs. Components should be restricted thusly to contain only other components. Since version 2.0 UML provides abilities to customize the language in an accurate way. Roh et al. describe their attempt in [RKJ04].

3.3.3 Concrete component frameworks

CORBA

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) to enable development of distributed software components independently from programming languages and platforms. CORBA is designed in an object-oriented manner (even if it supports procedural languages like C). Interaction between components is therefore done through invoking methods on (possibly remote) objects. By

this a client-server behavior is formed in which the client calls methods of an object residing on the server. The most important part of CORBA is the Object Request Broker (ORB) which is responsible for carrying requests over the border of the requesting component to the requested component (and carrying the result back to the requester). The ORB therefore transforms the request call of its (client) object to the inter-ORB protocol (IIOP) and sends the request to the ORB of the (server) object, where it is transformed back to be intelligible to the (server) object. Fig. 3.9 shows the generalized situation of two CORBA objects with the client on the left and server on the right side. Note that the two objects may reside in different processes, can be written in different programming languages or even be distributed over a network.

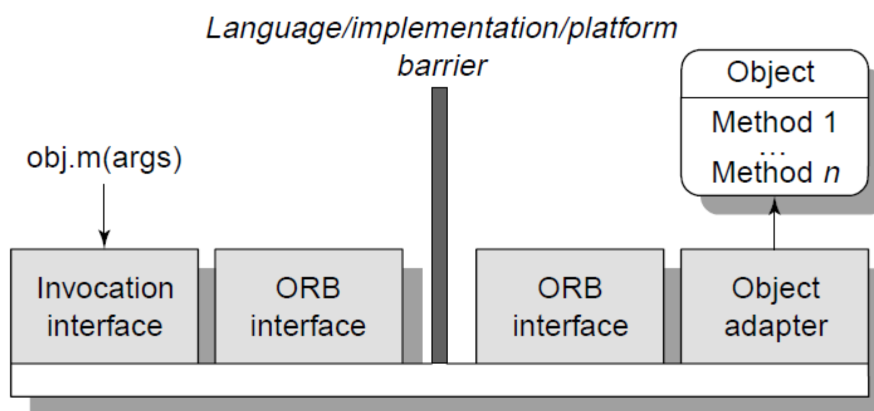

Figure 3.9: CORBA overview [SGM02]

JavaBeans

JavaBeans form an intraprocess component model for the Java Technology. JavaBeans explicitly support manipulation from tools after implementation and before deployment as the definition extracted from the JavaBeans Specifications reveals:

> ”A Java Bean is a reusable software component that can be manipulated visually in a builder tool.” [TMPL^{+}00]

In contrast to the aforementioned CORBA model JavaBeans are neither language independent nor platform independent, as JavaBeans can only be written in Java and need the Java Virtual Machine (JVM) as their run-time environment[1]. JavaBeans expose their properties through a naming conven-

[1] It can be argued that JavaBeans *are* platform independent as they require only the JVM and no particular OS or hardware

tion for the getter/setter method pair which controls the property. (The JavaBeans Specifications calls this naming convention "Design Patterns for Properties" [TMPL+00] which is misleading to they say the least.) JavaBeans use

Listing 3.2: Naming convention to expose a property. [SGM02]

```
public <PropertyType> get<PropertyName> ();
public void set<PropertyName> (<PropertyType> a);
```

a common event source/event listener model which is based on the Observer design pattern [GHJV95]. Both properties and event source/listeners can be introspected and manipulated by (graphical) tools. This allows for changing the properties of JavaBeans and setting up connections between different JavaBeans (wiring them) by an assembly tool before deployment.

Enterprise JavaBeans

Enterprise JavaBeans (EJBs) form a distributed component model for the Java Technology. Beside the name and the fact that both technologies support only Java, JavaBeans and EJBs have nothing in common (in particular EJBs are not the successors of JavaBeans). Unlike JavaBeans EJBs are not wired through

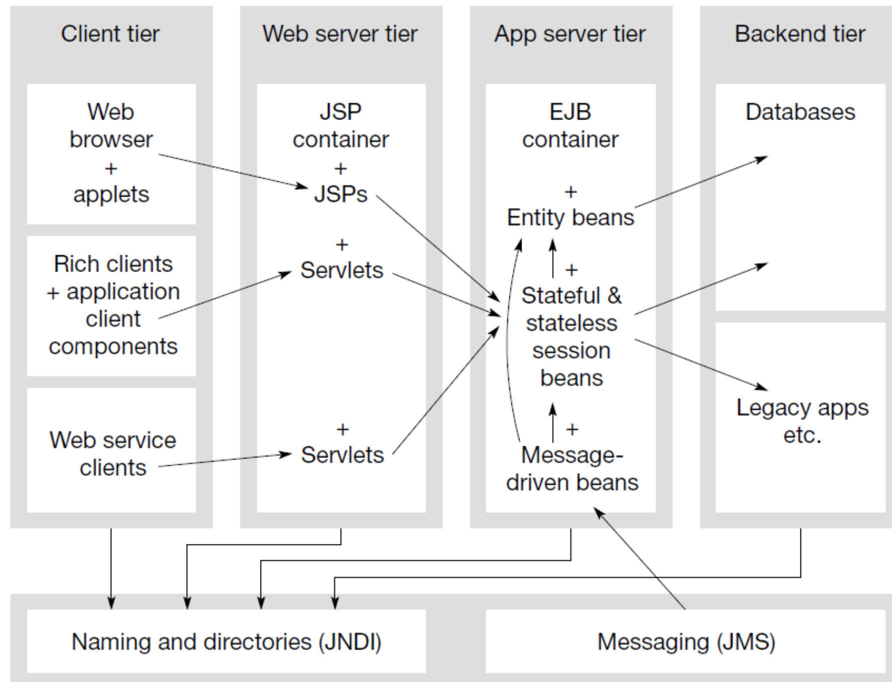

Figure 3.10: Architectural overview of J2EE [SGM02]

an assembly tool. Instead components make up their connections on their

own. The degree of composability of a developed EJB depends on the implementation and not on the component model. However EJBs support contextual composition, that is ”the automatic composition of component instances with appropriate services and resources” [SGM02]. Every EJB specifies its required components, services and its context. The run-time environment than guaranties that the EJB will be only executed in a context that matches its requirements. This context is implemented as a container around each EJB and communication between EJBs is solely done through the containers and not EJB instances. This enables additional functionality like network-transparency, so that EJBs can be distributed over a network. Other functionalities include support for concurrency, transactions and persistence, all introduced by the component model and achieved by its implementation. Two (formerly three) general types of EJBs are distinguishable: Session bean, Message-driven bean and Entity bean (deprecated).

Session Bean Session beans are divided three subtypes: Stateless, Stateful and Singleton. As the name suggests Stateless Beans have no (observable) state. This allows the run-time environment to both recycle Stateless Beans and execute different instances of the same Stateless Bean in parallel. Stateful Beans carry a state which changes through invocation of its methods. Concurrent execution is prohibited and concurrent accesses are simply serialized and executed consecutively. A Singleton Bean allows only the initiation of one instance (as the name indicates). Singleton Beans can control how concurrency on single instances is managed.

Message-driven Bean A Message-driven Bean is used for asynchronous communication. Message-driven beans cannot be instantiated by other beans or otherwise looked up. Instead a Message-driven Bean registers itself for some messages and when receiving such, processes them. If only Message-driven Beans are used for interaction in the software system it forms a message oriented architecture (MOA). Message driven architectures are very flexible but not appropriate for systems featuring an UI.

Entity Bean (deprecated) Entity Beans feature the representation of objects in (normally relational) databases. They provide access to the common CRUD-operations (Create, Read, Update, Delete) of databases. However Entity Beans were deprecated through version Enterprise JavaBeans 3.0 in 2006. Today persistence should easily be achieved by using the Java Persistence API (which itself is not bound to EJBs).

In summary both JavaBeans and Enterprise JavaBeans are language dependent component models. While JavaBeans allow wiring through assembly tools, Enterprise JavaBeans provide containers that wrap the EJBs and guarantee their requested context. EJBs also have access to high-level functionalities provided by the run-time environment including concurrency and transactions, which ease implementation.

Microsofts COM/DCOM/COM+

In contrast to the Object Management Group (OMG) Microsoft (MS) does not rely on global standards for its component models. Instead Microsoft releases own implementations of its proprietary Component Object Model (COM). COM serves as the base for more specific component models like object linking and embedding (OLE) or ActiveX. Szyperski has summarized the basics of COM as the following:

> "COM is a binary standard – it specifies nothing about how a particular programming language may be bound to it. COM does not even specify what a component or an object is. It neither requires nor prevents the use of objects to implement components. The one fundamental entity that COM does define is an interface." [SGM02]

COM is completely language independent as it only uses the binary form of an interface. In order to enable a programming language to use COM only a binding for such interfaces must be provided. In COM a component must implement COM's base interface (named IUnknown) but can implement an arbitrary number of additional interfaces. Microsoft also provides its own interface definition language (IDL) which is simply named IDL. Microsoft COM's IDL can be used with Microsoft IDL compiler (MIDL) to create COM-enabled binaries. As COM is a binary standard the usage of IDL is not mandatory but eases the usage of COM. The definition of the IUnknown interface in COM's IDL is as follows. Obviously the interface requires the implementation of three

Listing 3.3: IDL definition of the IUnkown interface. [SGM02]

```
[ uuid(00000000-0000-0000-C000-000000000046) ]
interface IUnknown {
  HRESULT QueryInterface([in] const IID iid, [out, iid_is(iid)] IUnknown iid);
  unsigned long AddRef();
  unsigned long Release();
}
```

methods: The second and third method provide automatic memory management through cooperative reference counting[2]. The first method is used to determine whether a given component has implemented a specific interface.

Microsoft has continuously improved and expanded COM and renamed it COM+ in the year 2000. Most notably Microsoft added the possibility to distribute components over a network by adding a transparent network layer to COM which is than referred to as Distributed COM (DCOM).

[2] Each component is asked to increase the reference counter of another component before using it and decrease the counter when the other component is no longer needed.

In summary (D)COM(+) is a language independent and distributed component model based on a binary standard. As COM and its extensions are proprietary standards hold by Microsoft it is mainly used on Microsoft Windows platforms (although open-source and proprietary implementation for other platforms do exist).

3.4 Conclusion

This paper examines how the usage of components can contribute to software development. Components should be based on a component model. In fact they are always, but the component model might only be implicitly defined. Six different component models were presented. The first two Acme and UML are architecture definition languages which serve the goal to define the architecture of a software system on the top level. While Acme was designed to interchange definitions from different ADLs it has matured to an ADL itself. Acme has some tool support but seems not to have a widespread usage in the industry. UML in contrast has widespread usage, but it is not perfectly usable to define architectures through components. Since version 2 UML provides customization abilities and there are attempts to adopt UML in order to transform it into an ADL (with conforming component support). The other four component models do not enable system definition through components but enable the implementation of components. From these four CORBA is the only global standard. It defines communication between components through proxies called Object Request Brokers (ORBs). Although CORBA is a quite mature approach (with its first important release of CORBA version 2 in 1997) and seems promising as an open standard backed by many major software development companies it suffers from many problems. CORBA is already seen as a vanishing technology by some people, e.g. Michi Henning, who worked on CORBA as a member of the OMG's architecture board and as an ORB implementer, consultant and trainer:

> ”Overall, however, CORBA's use is in decline and it cannot be called anything but a niche technology now.” [Hen06]

The remaining three component models are all platform respectively language dependent standards forged by single companies. Microsoft's COM and its successors enables components written in different programming languages to communicate and interact. However Microsoft only publishes implementations for its own operating system Windows. Despite the availability of implementations for other OSes it only plays a major role on Windows systems. The last two component models JavaBeans and Enterprise Java Beans were invented by Sun Microsystems (now part of Oracle) and are bound to the Java Technology. Nevertheless they are both platform independent in the sense that they

can be used on all platforms which have an implementation of the Java Virtual Machine. JavaBeans were designed to form graphical components which can be wired together in (graphical) assembly tools. In contrast Enterprise JavaBeans were designed to allow components to require a satisfying context which is provided in run-time through containers that wrap the individual component. While JavaBeans can reside both on the client and server side, EJBs can only be executed inside a server.

Both Acme and UML provide wiring through their graphical annotations. In addition Acme provides wiring through its textual definition language. The remaining component models presented differentiate regarding component binding. JavaBeans support connection-oriented programming (wiring) through graphical tools. EJB, COM and CORBA all support (static) wiring through global identifiers which normally is done in the implementation phase.

In conclusion component based development is still promising. On the one hand ADLs like Acme or the adoption of the widespread UML can be used to define systems by component composition. On the other hand tangible component frameworks like COM, CORBA, JavaBeans and EJB enable the component implementation. However there are still severe drawbacks for components: There is currently no widely accepted ADL which leads to lesser tool support for component based system architecture. It also forces system architects to learn several ADLs or leaves them with UML as the least common denominator. Graphical wiring during assembly is only possible for specific components (most namely graphical components, e.g. implemented with JavaBeans). Hard coding the component connections during implementation costs much of the flexibility components promised and contextual composition introduces possibilities of error (like composition mismatches). Furthermore development of a component is most likely much costlier than development in the traditional way. Although it is hard to measure Szyperski et al. suspect:

> ”As a rule of thumb, most components need to be used three times before breaking even.” [SGM02]

Component based development is (currently) not the ultimate solution to all software development problems as it sometimes praised. It is however a sensible approach for some software systems and it is a promising item for further research.

Bibliography

[Acm13] Acme Project. Acme - the acme architectural description language and design environment. `http://www.cs.cmu.edu/~acme/`, January 2013.

[CH01] Bill Councill and George T. Heineman. Definition of a software component and its elements. In George T. Heineman and William T. Councill, editors, *Component-based software engineering*, chapter Definition of a software component and its elements, pages 5–19. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[dz812] dz863.com. Nte1362integrated circuitaudio power amp, 5.5w. http://www.dz863.com/pinout-810646863-NTE1362/, 2012. [Online; accessed 21-January-2013].

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Foundations of component-based systems. chapter Acme: architectural description of component-based systems, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[Hen06] Michi Henning. The rise and fall of corba. *Queue*, 4(5):28–34, June 2006.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709 –724, oct. 2007.

[Mci69] Doug Mcilroy. Mass-produced Software Components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Proceedings of Software Engineering Concepts and Techniques*, pages 138–155. NATO Science Committee, January 1969.

[RKJ04] Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture modeling language based on uml2.0. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 663 – 669, nov.-3 dec. 2004.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Prentice Hall, 2002.

[TMPL+00] Enterprise Team, Vlada Matena, Eduardo Pelegri-Llopart, Mark Hapner, James Davidson, and Larry Cable. *Java 2 Enterprise Edition Specifications*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Wik12] Wikipedia. Screw — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Screw&oldid=527938384, 2012. [Online; accessed 21-January-2013].

Chapter 4

Model-driven Architecture

Hongyu Chen

Contents

4.1 Introduction 64
4.2 Basic concepts 65
4.2.1 Definition in Model-Driven Architecture 66
4.2.2 Related Standards 66
4.3 The principles of Model-Driven Architecture 68
4.3.1 Models in Model-Driven Architecture 68
4.3.2 Transformations between models 69
4.4 Model-Driven Architecture in practice 74
4.5 Related work 78
4.5.1 Domain Specific Languages 78
4.5.2 Architecture Reconstruction 78
4.6 Conclusion 80
Bibliography 80

Abstract: Model-Driven Architecture (MDA) has been used in software system development more than 10 years. MDA includes two main parts : one is Model, the other is Transformation between models. This paper introduces the basic concepts of MDA and illustrates the transformations between models, we also compare them in generally. An example of MDA is given to make it easy understanding. Related work about MDA is also introduced. Finally, we summarize the advantages and disadvantages of MDA.

4.1 Introduction

In 2001, the Object Management Group (OMG) introduced the Model-Driven Architecture (MDA).

What is the Model-Driven Architecture? Model-Driven Architecture is an approach using modeling languages to develop software systems. In figure4.1, I illustrate the processes of software development with the traditional way and the MDA approach.
With the traditional way, we generally get requirements from customers, then managers and developers analyze the requirements and build relevant models for the future software. The next step is to determine which platform should be chosen, and design each module depending on the analysis documents and platform. After the preliminary design, developers should write code to implement these modules. As well known, test and debug are necessary after coding. The final step is to deploy software systems.

The right part of figure4.1 presents the process used when employing an MDA approach. From the picture we can see clearly that the main parts of MDA are models and the transformations between models. The models in MDA include Computation Independence Model (CIM), Platform Independence Model (PIM), Platform Specific Model (PSM) and Code which can also be called Implementation. Generally the transformations are:

- From CIM to PIM
- From PIM to PSM
- From PSM to CODE
- From PIM to CODE directly

CIM is the same as Requirements part, because its main function is to model requirements and show the business environment where the system will operate; A PIM plays a role similar to the traditional Analysis part, because it gets information from CIM and enriches it to form a detailed view of a system by

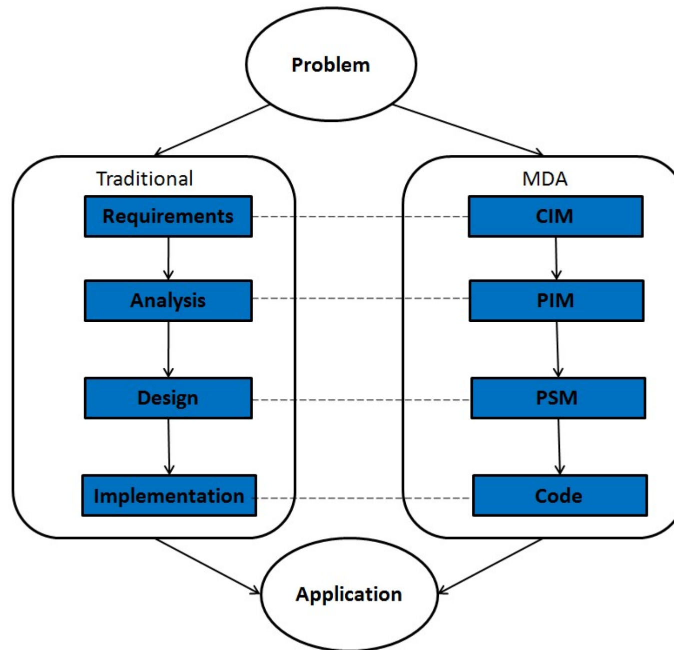

Figure 4.1: Development with traditional way and MDA

employing various modelling languages. A PSM is corresponding to the Design part, because in a PSM, the platform should be decided and technical details should be contained. Finally, the Code is the same as Implementation part.

This paper's purpose is to give an overview of the MDA standard and is divided in 4 chapters, as follows: Basic concepts, The principles of Model-Driven Architecture, Model-Driven Architecture in practice, Related work and Conclusion.

4.2 Basic concepts

Figure4.2 is the logo of MDA. It illustrates which standards the MDA uses.

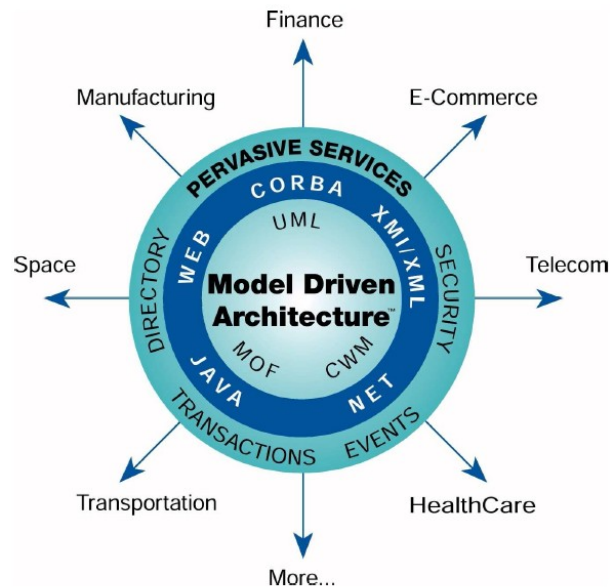

Figure 4.2: OMG's Model Driven Architecture. [OMG]

4.2.1 Definition in Model-Driven Architecture

Model-Driven

"MDA is an approach to system development, which increases the power of models in that work. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification."[MM+03]

Architecture

"The architecture of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors."[MM+03]

4.2.2 Related Standards

As mentioned in Introduction part, MDA includes many important standards defined by OMG. In the following part, I give these definitions of standards and how they support the MDA, then I explain the reason why MDA uses these standards. Finally, the relations between are given.

UML

”The Unified Modeling Language (UML) is a standard modeling language for visualizing, specifying, and documenting software systems.”[MM+03] It was created by the Object Management Group. Now we often use the version of UML 2.0. In the MDA, both PIMs and PSMs will be defined using UML profiles.

MOF

”MOF provides the standard modeling and interchange constructs that are used in MDA. Other standard OMG models, including UML and CWM, are defined in terms of MOF constructs. This common foundation provides the basis for model/metadata interchange and interoperability, and is the mechanism through which models are analyzed in XMI. MOF also defines programmatic interfaces for manipulating models and their instances spanning the application lifecycle.” [OBD+01]

CWM

”Common Warehouse Metamodel (CWM) is the OMG data warehouse standard. It covers the full life cycle of designing, building and managing data warehouse applications and supports management of the life cycle. ”[OBD+01] In the context of MDA, the CWM specification is especially useful for legacy integration.

XMI

”XML Metadata Interchange XMI is a standard interchange mechanism used between various tools, repositories and middleware.”[OBD+01] So any UML model can be represented in this XML-based format.

Relevance of the used standards

Model-Driven Architecture is based on the models and model transformations. So it needs standards to create models and transformation specifications.

- For the model specification, UML is a popular and well-defined language for model developing.

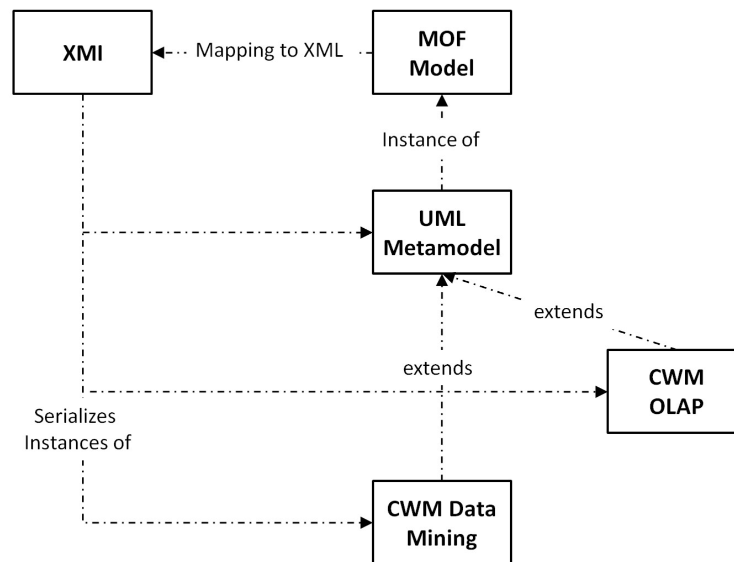

Figure 4.3: Relations between standards. Modified from [Joh]

- MOF metamodels, UML metamodel, CWM metamodel and interchanges between them are always used to transform from PIM to PSM.
- XMI is used to map the MOF to the XML.

So these standards are necessary and play important roles in MDA. Using these standards can help an architect to employ an MDA approach. Figure4.3 shows the relations between standards as mentioned before.

4.3 The principles of Model-Driven Architecture

4.3.1 Models in Model-Driven Architecture

As mentioned in Introduction part, models specified by MDA are: CIM, PIM, PSM and Code. In the next, I will give more details about each of these models.

CIM

Computation Independent Model (CIM) is a simple representation which models requirements for software systems and shows the business environments

in which it will operate. The CIM plays a similar role in MDA, as the requirements elicitation and analysis play in a traditional development process. [MM+03] offers more information about this.

PIM

Platform Independent Model (PIM) is a detailed view of a system. A PIM will contain information from CIM, in other words, transforms the requirements into elements of PIM. One important rule is that every element in a PIM can be traced to the requirements in a CIM. The information can be business functionality and behaviour, but not shows technical details of platforms which will be used. So it means the PIMs do not decide the Platform. So from a PIM, we can get different PSMs by transforming [MM+03].

PSM

Platform Specific Model (PSM) is a detailed view of a system. It includes not noly the information provided by a PIM, but also the technical details (such as Java methods, primary key in database) based on a specific platform which will be used to develop the system. In other words, in PSM, the platform is determined and the technical details of the platform should be illustrated in this PSM. The PSM should be specific and contains enough information to generate code. [MM+03]

Code

In the final step of the development is the transformation of PSMs to code. Code is the lowest level in a system development. Most of basic code is generated from models using MDA tools automatically. The complex code describing algorithm or others should be written manually.

4.3.2 Transformations between models

Model transformation is the process of converting one model to another model of the same system [MM+03]. In MDA, the model transformation means the transformations between these four models. Figure4.4 shows the basic transformations in MDA. They are CIM-to-PIM, PIM-to-PSM, PSM-to-Code, and a special one, PIM-to-Code. [OBD+01] also introduces the PIM-to-PIM, PSM-to-PSM and PSM-to-PIM.

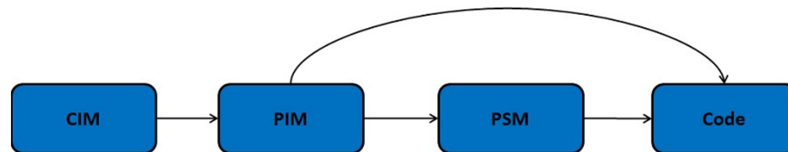

Figure 4.4: Model transformation in MDA

CIM-to-PIM

[NSN08] introduced that ”CIM presents specification of the system at problem domain level and can be transformed into elements of PIM”. In other words, CIM-to-PIM transformation is a process in which the requirements in CIM are transformed into the elements of PIM and described by modeling languages such as UML. So in this transformation, the most important thing is that we should make sure that every requirement is illustrated in PIM and based on modeling languages rules. [ZMZY05] shows a method named ”A Feature-Oriented Component-Based Approach” for transformation from CIM-to-PIM. ”It uses the feature model (consisting of a set of features and relationships between features) to structure requirements in CIM, and use the software architecture (consisting of a set of components and interactions between components) to organize elements at the PIM level.”

PIM-to-PSM

Transformation from PIM to PSM is the main part of MDA. Figure4.5 describes an example of a PIM and its transformation into three different PSMs. That means a PIM can be transformed into different PSMs. [OBD+01] [MM+03] show that mapping plays a key role in PIM-to-PSM transformation.
What is the mapping? A mapping is an approach which provides rules and techniques used to transform from a PIM into a PSM. There are two main kinds of mapping in MDA. One is Model Type Mapping, the other is Model Instance Mapping. The difference between them is explained below.
I merged some pictures from [MM+03] to form a new drawing, Figure4.6 which shows the mechanism of PIM-to-PSM by mappings. In figure4.6, the actions are Marking and Mapping. The green path shows the *Model Instance Mapping* approach for PIM-to-PSM transformation. It means the model elements in the PIM are marked by marks. These marks represent a concept in the PSM and

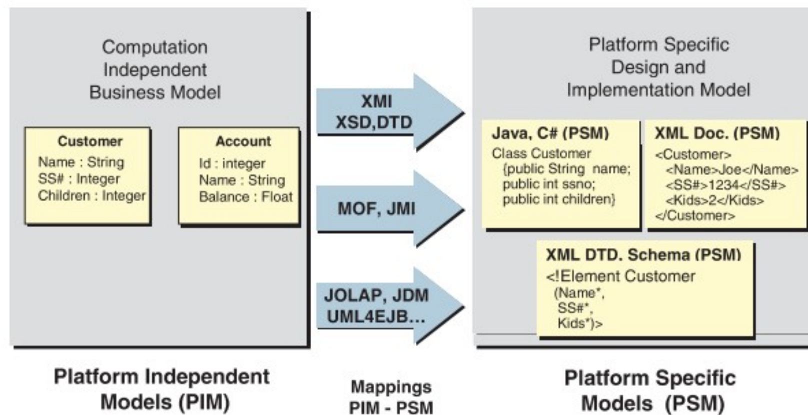

Figure 4.5: An example of PIM to PSM mappings [Bro04]

indicate how the elements in PIM is to be transformed. Then PIM becomes to "marked PIM". After mapping, the marked PIM is transformed to a PSM. On the anther hand, the red path shows the *Model Type Mapping* approach for PIM-to-PSM transformation. It means that the types described in the PIM are specified in models expressed using types from a PSM language directly. In Model Type Mappings, metamodel mapping is a specific example, the types of model elements in the PIM and the PSM are both specified as MOF metamodels [MM+03]. There are also many other type mappings. They all provide rules and/or algorithms to PIM-to-PSM transformation. Figure4.6 can also show the Combined Type and Instance Mapping. In this way, the advantages of using Model Type Mapping and Model Instance Mapping are combined.

As introduced in [MM+03], mappings can also include templates: "A template specifies a particular kind of transformation using parameterized models or patterns of model elements that are used as rules in model type mappings and model instance mappings." Sometimes when we mark the PIM, some additional information is also added to guide the transformation. For example, some information describes the quality of service in the model. I merge the Pattern applications and additional information to form a new picture. Figure 4.7 shows how PIM-to-PSM transformation works with pattern and additional information. The green line shows the Model Instance Mapping with additional information and patterns. In the process from PIM to Marked PIM, the additional information and patterns can be used. Furthermore, from Marked PIM to PSM, they can also be used to guide the transformation. The red line shows the Model Type mapping with patterns.
Figure 4.8 shows the table summarized by me to compare Model Type Mapping and Model Instance Mapping.

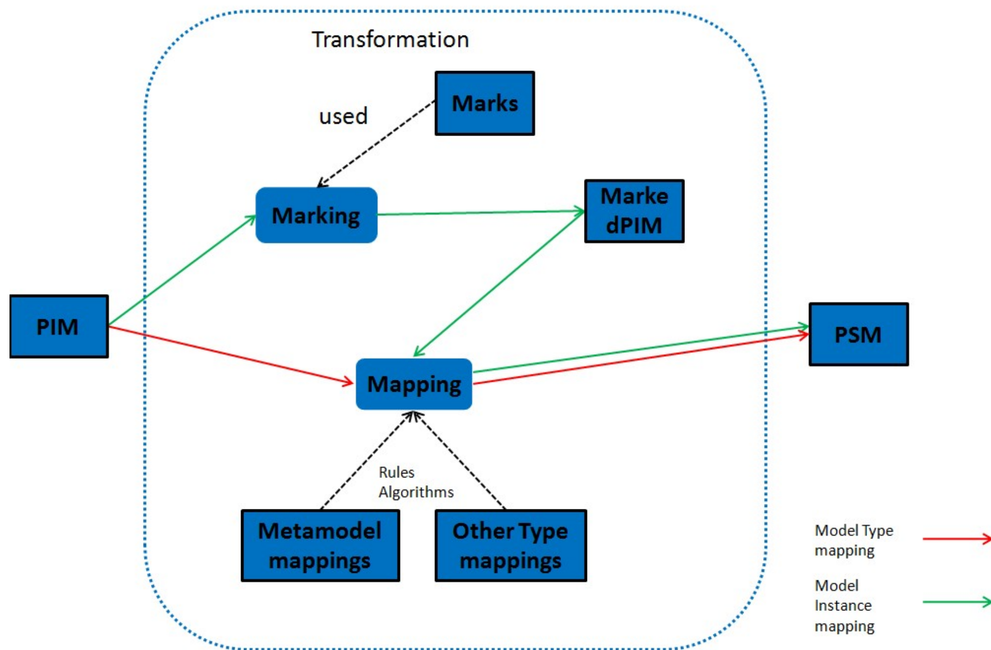

Figure 4.6: PIM-to-PSM transformation by mappings

PSM-to-Code and PIM-to-Code

In figure 4.4, I have illustrated that there are two transformations to code. One is PIM-to-Code, the other is PSM-to-Code. PIM-to-Code is a direct transformation without producing PSMs. In this transformation, the PIM must includes enough information to be transformed to code. It is not often used in software development.

PSM-to-Code is a familiar transformation to generate code. In PSMs, the information about requirements and technical details for the platform are specified. So what developers should do is using tools to generate code automatically or write/fix codes manually by following the platform language semantics. Figure4.9 illustrates an example of the transformation from PSM to code. In the picture, a PIM transforms to three kinds of PSM including J2EE, .NET and CORBA. Then each PSM is transformed to its own language. J2EE generates JAVA code, .NET generates C# code, CORBA generates C++ code. Transformation from a PSM to code also is supported by Model-To-Code transformation [KWB03]. There are two approaches for Model-To-Code [CH03].

- Visitor-Based Approaches

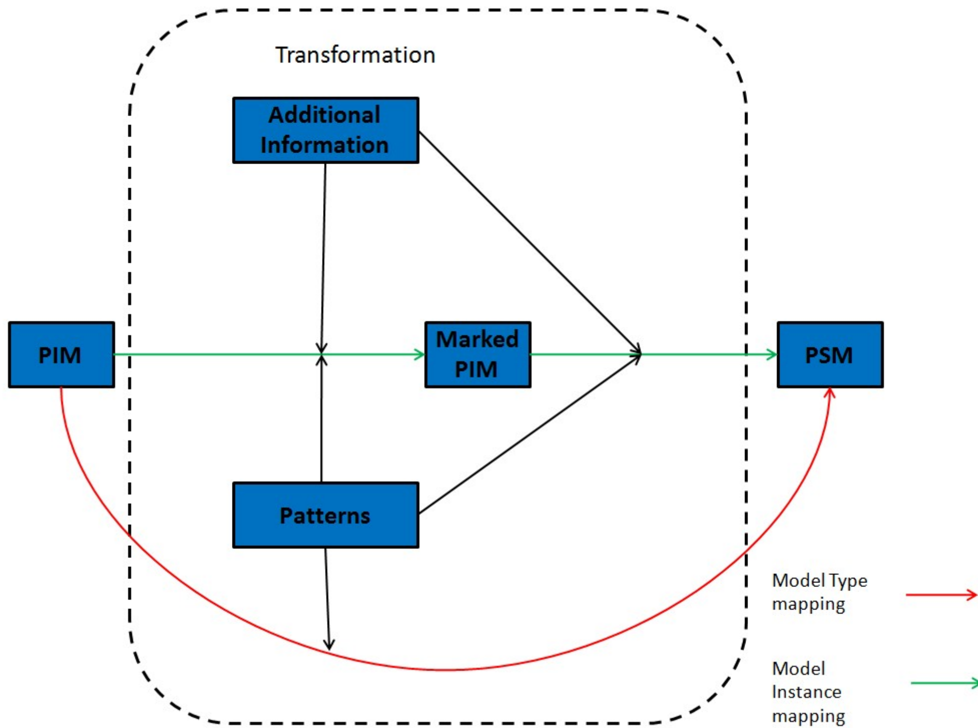

Figure 4.7: PIM-to-PSM transformation by pattern and additional information. Modified from [MM+03]

	Marks	Additional Information	Rules/ Algorithms	Patterns
Model Type Mapping	X	O	O	O
Model Instance Mapping	-	-	X	O

X ----- must use
O ----- may use
— ----- no use

Figure 4.8: Compare Model Type Mapping with Model Instance Mapping

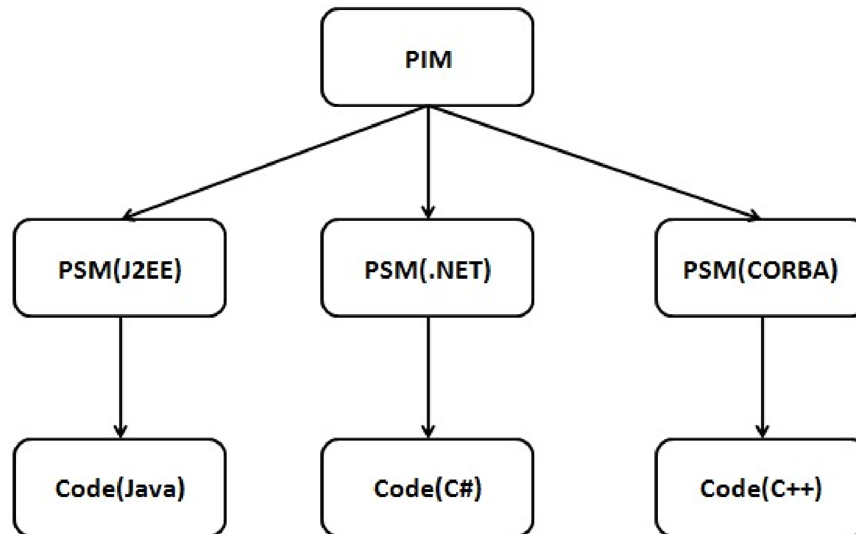

Figure 4.9: Transformation

- Template-Based Approaches

4.4 Model-Driven Architecture in practice

[MF05] gives an example about CIM-PIM-PSM using OptimalJ. The example is summarized below:

Create CIM

First, based on the textual requirements specified by the stakeholders which can be found in [MF05], the following CIM-relevant model has been proposed as fig4.10 which shows the workflow.

CIM-to-PIM

The following part is the transformation from CIM to PIM. As mentioned, it needs model languages to create a PIM. In this case, PIM was created using UML 2.0 and Rational Rose 2003. Through analyzing the CIM, it defined six

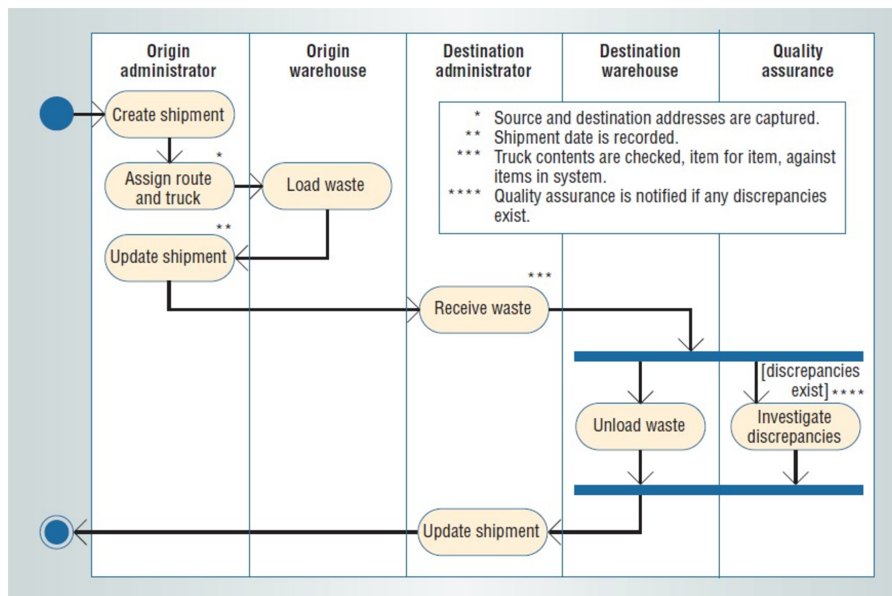

Figure 4.10: CIM of a web-based application to track chemical shipments [MF05]

classes and illustrated the n-to-n or 1-to-n relations. The important thing is that every element in the PIM must be traced to CIM. Figure4.11 is the PIM transformed from CIM.

PIM-to-PSM

In this case, the target platform is J2EE. So it used OptimalJ to generate PSM from PIM. Figure4.12 shows the EJB-based PSM. "This model includes platform-specific details, such as the addition of two finder methods in the ShipmentItem class, one by key and another by shipment, not present in the PIM." Figure4.13 illustrates a platform-specific database model transformed from the PIM by OptimalJ. This model includes database-specific attributes and relationships. "For example, the DeliveryTruck relation shows that it has a primary key named ID and that it participates in a many-to-many relationship with Shipment. The model also lists specific data types for each relation." Because of the requirements, this case choosed the JAVA platform as PSM target. But if the requirements change to use .Net as the target platform, we should use OptimalJ to generate a PSM using .Net rules.

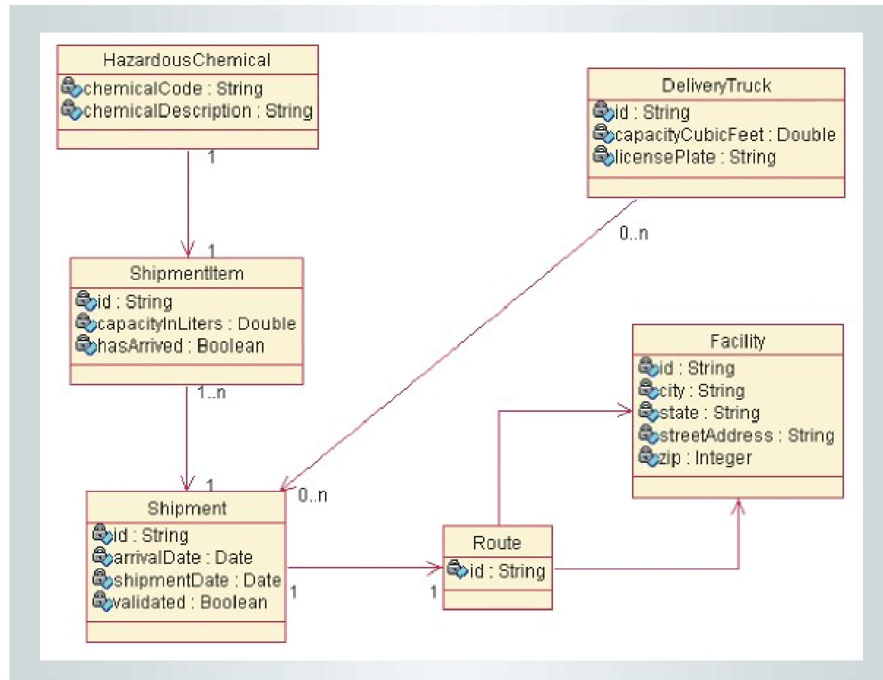

Figure 4.11: Platform independent model in Rational Rose 2003 [MF05]

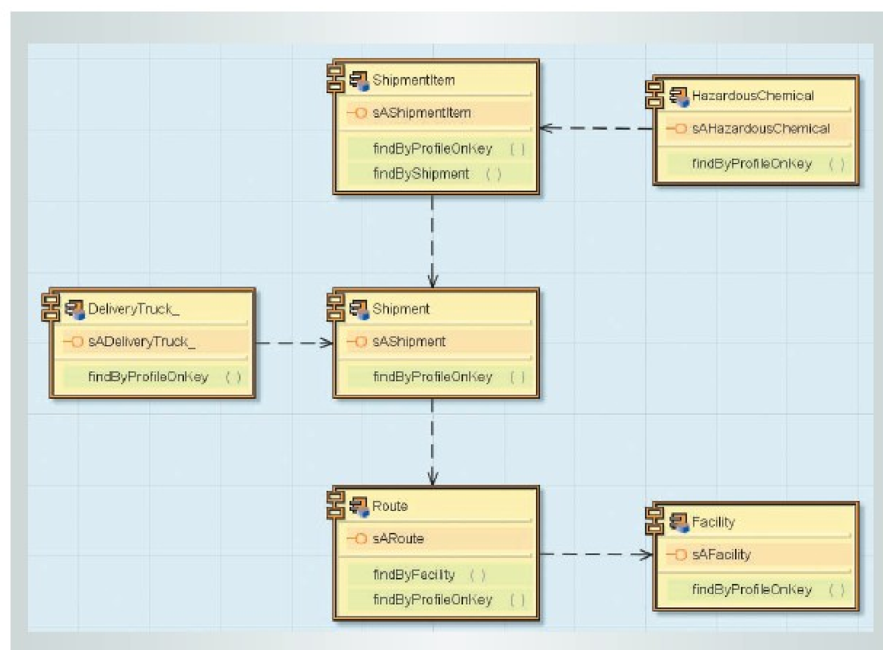

Figure 4.12: Platform-specific EJB model in OptimalJ [MF05]

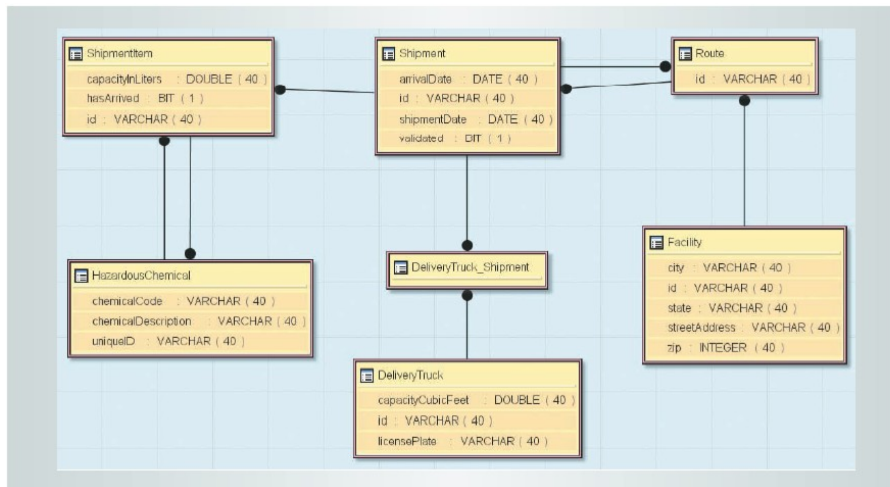

Figure 4.13: Platform-specific database model in OptimalJ [MF05]

PSM-to-Code

The final step is to transform PSM to Code. It is not complex. Most code would be generated from PSM by tools such as OptimalJ automatically. While some code should be modified by developers manually, because it is too complex to be defined in models.

Tools for MDA Now there are many tools including open source tools and Commercial tools for MDA. I list some popular Open source tools:

- OpenMDX, which supports code generation towards J2EE and .Net.
- Kermeta, which is built as an extension to Eclipse EMF.
- MOFScript, which based on one of the OMG MOF Model to Text Transformation submissions and metamodels/models in EMF.

Some popular commercial tools including:

- OptimalJ
- ArcStyler
- MCC (Model Component Compiler)

4.5 Related work

4.5.1 Domain Specific Languages

Definition

As mentioned in [VDKV00], ”a domain-specific language (DSL) is a small, usually declarative corresponding to the language that offers expressive power focused on a particular problem domain”. A DSL is a specialized and problem-oriented language. It forces on describing a domain of knowledge accurately.

Relation to MDA

A CIM is sometimes called a domain model [MM^{+}03]. A DSL is specific to a given domain, so DSLs make contribution to CIM. [LJJ07] said ”In model-driven engineering, a Domain-Specific Language (DSL) is a specialized language, which, combined to a transformation function, serves to raise the abstraction level of software and ease software development.” In MDA approach, PIMs use model languages to abstract software systems and then are transformed to PSMs. DSLs are languages which can make contribution to PIMs not only on system abstraction but also on transformation from PIMs to PSMs. Especially the UML, has been used in MDA generally.

4.5.2 Architecture Reconstruction

Definition

[Kos09] said ”Software architecture reconstruction is the form of reverse engineering in which architectural information is reconstructed for an existing system. Those information is gathered from the source, the system’ execution, available documentation, stakeholder interviews, and domain knowledge.” In other words, Architecture reconstruction is a reverse engineering from the implementation. Depending on [Riv02], Architecture reconstruction can be conducted in the following phases :

- Developers map the high level concepts with the source code or implementation.

- Depending on the mapping, developers build a model of the system. The model of the system is just at a very low level of abstraction.

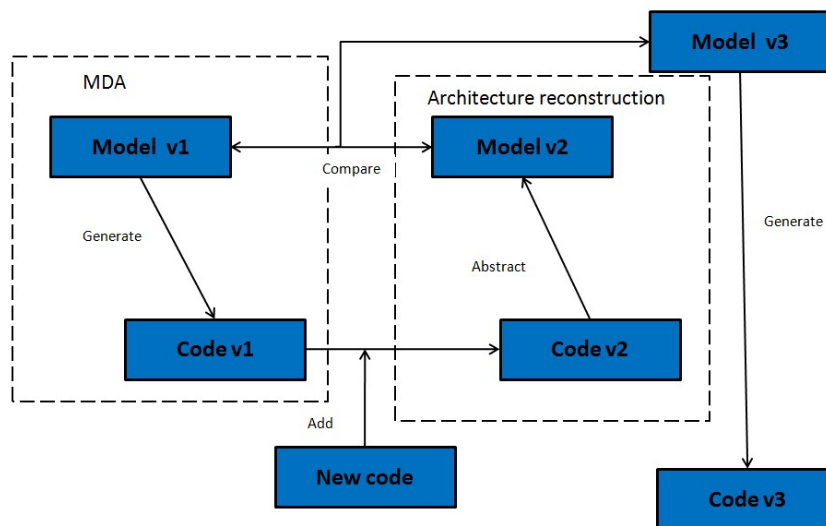

Figure 4.14: Relation between MDA and Architecture reconstruction [MF05]

- Enrich the model with domain specific knowledge that will lead to a high level view of the system. From the enriched model, we can get a global understanding of the system.

Relations to MDA

In MDA approach, we also need Architecture reconstruction. During the development, the requirements or other documents always have to be changed. What we should do is to change CIM and PIM, change the models to adapt the new plan. But it is complex and waste resource if we rebuild the whole models since some small requirements changes. So we can use Architecture reconstruction to make it easy. Figure4.14 illustrates the relation between MDA and Architecture reconstruction. In MDA part, we can generate *Code v1* from *Model v1*. When some new requirements are added, we can write the new code first, then merge them with *Code v1* and get *Code v2*. In Architecture reconstruction part, we abstract the *Code v2* to form *Model v2*. Finally, we compare *Model v2* with *Model v1* and form *Model v3*. For this system, that is enough. While when we do some new systems similar to it, we can generate *Code v3* from *Model v3* directly.

4.6 Conclusion

MDA has been developed for more than ten years. There are many advantages using MDA to develop software systems. Firstly, it increases application re-use and reduces the development cost. For example, if a software system has been developed in J2EE platform, and now the same or another customer wants to deploy the same system on a .Net platform, we can use tools to transform the same PIM to another PSM suitable for the .Net platform. In this way, the time and cost for developing CIM and PIM can be saved. Secondly, it makes the cross-platform interactive easy. Because one PIM can be transformed to several PSMs depending on the requirements.

Now from many examples we find that MDA has succeeded in large, distributed, industrial software development, but in small business and small project, MDA has not been used widely. There is a big disadvantage using MDA. That is the time invested in the models. There is a tradeoff between grasping the code and program directly and invest more time in the models. Describing everything at the model level can be very time consuming and hard. For example, describing a search algorithm in a model is much harder that just implementing it. So the tradeoff needs to be analysed carefully beforehand. In the future, I think MDA should be developed towards small projects and make the development process more easy.

Bibliography

[Bro04] Alan W Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327, 2004.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.

[Joh] John D. Poole. Model-Driven Architecture: Vision, Standards And Emerging Technologies.

[Kos09] R. Koschke. Architecture reconstruction. *Software Engineering*, pages 140–173, 2009.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. Mda explained. the practice and promise of the model driven architecture, 2003.

[LJJ07] B. Langlois, C.E. Jitia, and E. Jouenne. Dsl classification. In *OOPSLA 7th Workshop on Domain Specific Modeling*, 2007.

[MF05] T.O. Meservy and K.D. Fenstermacher. Transforming software development: an mda road map. *Computer*, 38(9):52 – 58, sept. 2005.

[MM+03] J. Miller, J. Mukerji, et al. Mda guide version 1.0. 1. *Object Management Group*, 234:51, 2003.

[NSN08] O Nikiforova, U Sukovskis, and V Nikulsins. Principles of Model Driven Architecture for the task of study program development. *Joining Forces in Engineering . . .* , 2008.

[OBD+01] Architecture Board Ormsc, Carol Burt, Desmond Dsouza, Keith Duddy, William El Kaim, William Frank, Sridhar Iyengar, Joaquin Miller, Jeff Mischkinsky, Jishnu Mukerji, Jon Siegel, Richard Soley, Sandy Tyndal, Axel Uhl, Andrew Watson, and Bryan Wood. Model Driven Architecture (MDA) Document number ormsc / 2001-07-01. pages 1–31, 2001.

[OMG] OMG. http://www.omg.org/mda/index.htm.

[Riv02] C. Riva. Architecture reconstruction in practice. In *Proc. Working Conf. on Software Architecture (WICSA)*, 2002.

[VDKV00] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[ZMZY05] W. Zhang, H. Mei, H. Zhao, and J. Yang. Transformation from cim to pim: A feature-oriented component-based approach. *Model Driven Engineering Languages and Systems*, pages 248–263, 2005.

Chapter 5

Software Architecture Evolution

Afshin Ghanizadeh

Contents

5.1 Introduction 84
5.1.1 Problem Statement 84
5.1.2 Research Motivation 84
5.1.3 Research Outline 84
5.2 Background 85
5.2.1 Software Architecture 85
5.2.2 Basic Concepts 86
5.2.3 Architectural Views and Viewpoints 87
5.2.4 Level of Abstraction 88
5.3 Related Works 88
5.3.1 Software Evolution 88
5.3.2 Architecture Recovery 90
5.3.3 Model-Driven Architecture 90
5.3.4 Dynamic Evolution of Software Architecture 91
5.4 Software Architecture Evolution 92
5.4.1 Introduction 92
5.4.2 Model-Driven Software Evolution 94
5.4.3 Dynamic Software Evolution 95
5.5 Conclusion 96
Bibliography 97

Abstract: Most systems are built to be used for a long time and will be considered as long-lived systems. Due to uncertainty of market, social and economic, most systems will be updated over a long period of time. This study is concerned with the design of different architectures and dynamic evolvable softwares.

5.1 Introduction

5.1.1 Problem Statement

Evolution of software systems is considered as inevitable changes of software specifications and is a natural process.Every evolution can take place at different level of abstraction of software like architectural level. It is important to design a software with ease of extension and it depends on how well the structure of the software is built.

5.1.2 Research Motivation

In this study, we are going to explore the relationship between evolution and architecture. For this purpose, some architecture evolution models are described in details like Architecture-Driven Software Development, Model-Driven Software Evolution, and Dynamic Software Evolution.

5.1.3 Research Outline

The problem statement is explored in the first chapter of this study. In the second chapter, some backgrounds on software evolution, Architecture Recovery, Model-Driven Architecture, and Dynamic Evolution of Software Architecture are provided. Software evolution architecture will be studied as an inevitable process in chapter 4. Finally, report is concluded in chapter 5.

5.2 Background

5.2.1 Software Architecture

Definition

Several definitions about software architecture have been proposed in the last decade. The first definition explained here is proposed by Perry and Wolf [PW92] which is one of the most widely extended and accepted definitions:

Software Architecture={Elements, Form, Rationale}
Elements capture the system's building blocks, which can be of three types: processing elements, data elements and connecting elements.
Form captures how the (architectural) elements are organized in the architecture, by means of weighted properties and relationships. That is, the form captures how the elements are composed (i.e. The architecture configuration), the characteristics of their interactions, and their relationship with their operating environment.
Rationale captures the motivation for the choice of an architectural style, the choice of elements, and the form. That is, the system designer's intent, assumptions, choices, external constraints, selected design patterns, and other information that is not easily observable from the architecture.

Practical characterizations of this definition were later provided by Taylor, Medvidovic, and Dashofy [TMD09] using What, How and Why questions:

Elements help to answer the What questions about the architecture: What are the elements of a system? What are their primary purpose and the services that they provide?
Form helps to answer How questions about the architecture: How is the architecture organized? How are the elements composed to accomplish the system's key task? How are the elements distributed?
Rationale helps to answer Why questions about the architecture: Why are particular elements used? Why are they combined in a particular way? Why is the system distributed in a given manner?

Another definition is given by ANSI/IEEE Standard 1471-2000:

Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principle governing its design and evolution.

One of the most recent definitions comes from the software architecture book written by Taylor et. al [TMD09]:

A software system's architecture is the set of principal design decisions made

about the system. Design decisions encompass every aspect of the system under development, including: system structure, functional behavior, interaction, nonfunctional properties and implementation. Principal is a term that implies a degree of importance and topicality that grants a design decision architectural status, that is, that makes it an architectural design decision (i.e. It impacts system's architecture).

This definition is the most abstract definition among aforementioned definitions which gives the architecture a central role in development of a system. Therefore, architecture not only includes the structure of a system, but describes main functional behavior, the non-functional properties and the technology. Despite different issues in each definition, all of them are mainly concerned with behavior and structure. Both terms is specified using Architectural Description Language (ADL).

5.2.2 Basic Concepts

Some concepts that are common in all ADLs are presented in this section.

Component

A component is a computational element with a high level of encapsulation that allows users to structure the functionality of a software system.
A software component is an architectural entity that [TMD09]:

1. encapsulates a subset of the systems's functionality and/or data.
2. restricts access to that subset via an explicitly defined interface.
3. has explicitly defined dependencies on its required execution context.

Connector

A connector is an architectural element tasked with effecting and regulating interactions among components. [TMD09]
Connectors are the locus of relations among components. They mediate instructions but are not ”things” to be hooked up. Each Connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc. [SG96]

Configuration

An architectural configuration is a set of specific association between the components and connections of a software system's architecture.[TMD09]

System

Systems represent architectural configurations that are made of connectors and components built in a hierarchical way. The Concept of system differs from configuration in that a system is a building block that can be reused in several software systems, whereas a configuration defines the structure of a specific system. [CS11]

Port

Ports are points where architectural elements can interact with other elements of a software.

5.2.3 Architectural Views and Viewpoints

In order to record an architecture, architectural views can be described as descriptive groups. In fact, representing an entire system based on a group of issues constitutes a view. In order to create and use a view, specific conventions are required, which can be specified as a viewpoint.

The viewpoint determines the languages to be used to describe the view and any associated modeling methods or analysis techniques to be applied to these representations of the view. [PGJ05] In order to fully describe a software, it is greed that various views and viewpoints are essential. Studies have been done that all categorize different view models. For example, [HNS00] suggested views including: conceptual architecture, module interconnection architecture, execution architecture, and code architecture.

Bass *et al.*[BCK03], on the other hand, proposed Three categories of architectural views:

Module Views, which contain elements that can describe the implementation units of the whole software system.
The second one is *components and connector views*, whose elements are computational units like clients, servers, databases which are called the components, and communication instructions such as remote procedure calls, named pipes, and sockets that constitute the connectors.

Allocation Views are the third category which contains elements that are components, connectors, modules, and environmental resources.
There are three categories of relationships used in views regularly modules, which are units of functionality in implementing softwares:

The first type of relationship is is-part-of, which defines a part/whole relationship between a submodule and a module.

The second type, depends-on, defined a relationship of dependency between two different modules. This type is commonly used in the beginning stages of the design.

"is-a" is the third type of relationship which defines a relationship of generalization between a particular modules and general modules, which can be related to as the parent of the first module, which is often called the child.

5.2.4 Level of Abstraction

There are various levels of abstraction for software implementation views. These levels often depend on concerns that are to be represented.

1. Architecture: The first level includes architectural viewpoints which indicate the parts of the system that are related to the architecture.
2. Design: The design aspects of the softwares can be described using design viewpoints which are often represented using models of source code which include all the details.
3. Code: The syntactic structure of the source files and classes are represented by the code viewpoints.AST (Abstract Syntax Tree) is commonly used to model the syntactic structures.
4. Source Text: Source files, build commands, configuration files and other documents make up part of the implementation of the system.

5.3 Related Works

5.3.1 Software Evolution

Lehman and Belady's Laws of Software Evolution [LB85] work concluded that a more complex and a more resource consuming system will be established as the system evolves. Besides that, successful systems become less useful over time in that environment. Many design patterns, specially all of the patterns in

the original Design Patterns book [VHJG95] are meant to increase flexibility, however they become more costly in terms of complexity. Rules that moderate system growth and evolution were established by software architectures [SG96], however, some changes can break the assumptions of an architectural style.

Several approaches analyze the influence these changes have on evolving software system. The influence of changes on the maintainability of software systems by defining a set of measurements or depict the complexity of the calls were analyzed by [BM99]. Gold and Mohan [GM03] defined a framework to understand the conceptual changes in an evolving system .

Analysis at the level of entities in a meta-model was suggested by [ZWDZ04]. Their main concern was priding a mechanism that warns programmers about the consequences of the changes being made.

Later, Ying et al. applied data mining techniques to the change history of the code base to identify change patterns [YMNCC04]. The Hipikat [CM03] approach was introduced by Cubranic and Murphy . Hipikat used project information to provide recommendations for a modification task. Project information includes different sources, including the source code versions, modification task reports, newsgroup messages, email messages, and documentation.

Fenton and Ohlsson report was based on an experiment with two commercial software systems where they tested a range of basic software engineering hypotheses relating to: The Pareto principle of distribution of faults and failures [FO00]. No evidence to support the hypothesis that size and complexity of modules are good predictors of either fault-prone or failure-prone modules were found. Their results showed that the most fault-prone prerelease models are among the least fault-prone postrelease, while, the most fault-prone postrelease are among the least fault-prone prerelease.

Lanza et al. introduced the Evolution Matrix by taking into account different releases of a system [Lan01] that represent the history of classes. The history of changes in software systems to detect the hidden dependencies between modules was analyzed by [GHJ98]. Their analysis was at the file level rather than dealing with the real code and considered release and version information of software units. The concept of logical coupling was extended by Fischer et al., they also defined a filtering mechanism and a data scheme for such an integration in [FPG03]. Their data scheme is the initial version of the Release History Database (RHDB).

Based on CVS data Krajewski et al. discovered change couplings: developers checking in and out files within certain periods of time and the relationship of these files discovered dependencies that are difficult to detect by other means and pointed to several bad code smells [FB99] by means of visualizations using JGraph [GJK03].

5.3.2 Architecture Recovery

Research on architecture recovery spans a wide area of activities: approaches, such as Bookshelf [FHK+97], and Dali [KC99]follow the Extract-Abstract-View Metaphor described in [RW02]. They focus on the creation of condensed high-level views to facilitate program understanding. Most tools differ in the underlying fact extraction technique, in the methods and details of fact representation, and in the analysis and visualization techniques.

Murphy and Notkin proposed a reconstruction technique based on reflexion models [MNS95]. The user starts with a structural high-level view model that is mapped against the source code. The result of the mapping is a reflexion model that shows the differences between the developer's high-level and the recovered model. Koschke and Simon have extended the original reflexion models to hierarchical architecture models [KS03].

Extracting architectural properties from large open source systems such as the Mozilla system has been addressed by Godfrey et al. [GL00]. Their work relied on PBS which is a reverse engineering workbench containing the Relational Algebra tool Grok.

Riva proposed a view-based architecture reconstruction approach named NIMETA [Riv04].Similar to Krikhaar the approach is based on relational algebra. NIMETA emphasizes the scrupulous selection of architectural concepts and architecturally significant views that are reflecting the stakeholders' interests.

Other works concentrate on diverse coupling metrics: in [BDW99] Briand et al. discuss a unified framework for coupling measurement in object-oriented systems based on source model entities. Based on these metrics they verified in [BWL99] the coupling measurements on file level using statistical methods and change coupling information based on a "ripple effects".

In terms of analysis of evolution history data, Zimmermann et al. inspected release history data of several software systems for change coupling between source code entities [ZWDZ04]. They conclude that augmentation of architectural data with evolutionary information could reveal new otherwise hidden dependencies between source code entities. Even though a number of other work used release history data as well, a detailed evaluation of the correlation between source model entities and the properties of change coupling is still missing.

5.3.3 Model-Driven Architecture

a Architectural documentation and models must be distinguished in order to properly use the automated software engineering tasks of UML for documen-

tation

An Architecture must be defined that contains the high-priority decisions regarding the software design in order to develop a software design.

Architectural Descriptors containing models and architectural views are part of the Model-Driving Architecture Framework.

Models, that can be used for tasks like code generation and analysis, often follows MetaModels, which along with viewpoints address a specific group of concerns.

In Model-Driven architecture, models and views re usually connected using UML diagram. Views are created from the same models as the source code using the connection between the view and models. In this way, views become model driven.

A metamodel for a viewpoint can be defined based on an ADL that is about a related concern. A viewpoint description can be used and the metamodel relations and elements are generated. Metamodels are normally created very easily due to the simplicity of the ADL syntaxes.

The mapping between the metamodel and UML is specified and implemented using model transformation languages. In order to transform an architectural model to a UML model, existing architectural model mappings to UML can be specified as model transformations.

The UML diagram doesn't represent the architectural model precisely, but it is suitable for communication purpose. This is why UML is so commonly used for architectural documentation in industry.

5.3.4 Dynamic Evolution of Software Architecture

The diversity and complexity of the existing software systems have made software architecture more important.

Dynamic Software Architectures are the architecture descriptions that include the descriptions of the both fixed and changing parts. This means that the dynamic software architectures can react to specific requirements or events by run-time reconfiguration of its components and connections.

If the structure of the system is changed, the dynamism is of the dynamic reconfiguration type, and if the types that compose the structure are changed, it is of the dynamic evolution architectural types.

Dynamic reconfiguration is the first evolution type that is about run-time changes in the software structure. According to Endler and Wei [EW92], dynamic re-

configuration is the changing part of an application while it operates.

Magee and Kramer [MK96] define dynamic reconfiguration as a feature of ADLS used to describe and model software systems with a dynamic structure.

Dynamic Evolution of architectural types, which is the second type of dynamisism discussed in this report, can be used to change the type of an element of architecture and its instances at runtime. This type is essentially required to create open systems or to update highly-available systems.

Dynamic Evolution of architectural types can introduce architectural types and links, can remove already existing architectural types, and can modify such architectural types at run-time. Instances of the above-mentioned modification include updating a component type or a connector type and the migration of the state of all the instantiations of an architectural type.

It is important to note that the dynamic reconfiguration acts at the configuration level, which the dynamic evolution of architectural types acts at the type level. In specific cases such as changing the topology of a software, both kinds of dynamism are required together.

5.4 Software Architecture Evolution

5.4.1 Introduction

Software Maintenance vs Software Evolution

Software evolution is often considered equivalent to software maintenance because of the lack of a standard definition. Software maintenance is defined in the ISO/IEC 14764 IEEEStd 14764-2006 standard as:

The totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage (Planning for post-delivery operations, supportablity, and logistics) as well as the post- delivery stage (software modification, training, and operating a help desk).

The majority of following evolution-related research themes are crucial activities in software maintenance: software comprehension, reverse engineering, testing, impact analysis, cost estimation, software quality, software measurement, process models, configuration management, and re-engineering [MWD+05].

However, these two terms should not be considered as synonyms. The maintenance itself indicates that the software should naturally weaken or get worse over time, which is not the case at all. In general, software maintenance is a

supporting process of a software product which keeps the operationality and usability of the product. Maintenance activities are a correcting process that improves the performance or other attributes, or to help the software adapting to a changing environment, such as an upgraded operation system. On the other hand, adding new functionality to a software is considered an evolutionary activity and not a maintenance activity. For instance, in software versioning (Apache, 2010), major version number reflects important changes and minor version number reflects improvements or corrections.

Evolutionary activities are performed at the architectural level involving substantial improvements, while maintenance activities involves minor changes in components.

Evolution as part of the Development Process

Since the term software evolution is perspectively broader, many authors preferred this term over maintenance when refering to a software life cycle phase which starts from the initial creation of the product and ends until its retirement. This results in the following development methods where evolution plays a crucual part: Evolutionary developement (, the Spiral model, the Staged model, and Agile Software Developement. The software evolution phase involve adapting the product to new requirements:

[The phase of] software evolution takes place only when the initial development was successful. The goal is to adapt the application to the ever- changing user requirements and operating environment. The evolution stage also corrects the faults in the application and responds to both developer and user learning, where more accurate requirements are based on the past experience with the application.

In conclusion, software evolution can be defined as activities and processes that improve, correct, reduce or extends its current functionality to fulfill new requirements based on experience from the past.

Architecture-Driven Software Development

Software development is an executable specification of its behavior, commonly in the form of source code derived from a large number of design decisions. For years, the pay off of considering software system's structure and organization alongside it's behavior for reasons of dependability, understandability, and maintainability has been concluded.

To ensure designs remain comprehensible in complex software systems, multiple levels of abstraction are mandatory which results in several types of de-

sign. It is challenging to define the notion of software architecture in a single definition. Many definitions has been collected by The Software Engineering Institute in 2006. A model of software architecture provided by Perry and Wolf [PW92] consists of elements which are processing, data, and connecting elements, form which includes the relationships between architecture elements, and rationale which provides the motivation for the decisions that produces a particular set of elements and form.

The definition found in IEEE-1471 not only includes elements and their relations, but also principles such as the use of a particular architectural style or particular conventions during design and maintenance of a software system. A more frequently used definition given by Bass et al. [BCK03] states that there is no such thing as the structure of a software system and to describe the architecture of a single system, different types of structures can be used. Kruchten [Kru04] states that software architecture encompasses a set of significant decisions regarding system organization, selection of elements, their composition, and selection of an architectural style to guide these decisions.This definition was later developed by Jansen [JB05].

To summarize, software architecture can be defined in two ways: 1) as a set of (architectural) design decisions, or 2) as the structure that is the result of those decisions. Definitions thus far gave rise to unclarity of whether a design decision is architectural or not. To clarify, Eden et al. [EHK06] provided a criterion that can be applied to design statements. This criterion states that a design statement is local if the system cannot be violated by mere expansion. Architecture statements are in the class of non-local statements. For instance, expanding one of the layers with a component that interacts with components in nonadjacent layers can violate the layered architectural style of a software system. Conversely, a design pattern cannot be violated by only expanding a system. Thus, decisions regarding design patterns are not architectural. In industry architecture is defined differently and different sets of decisions are considered to be architectural.

5.4.2 Model-Driven Software Evolution

Architectural documentation and models must be distinguished in order to properly use the automated software engineering tasks of UML for documentation.

An Architecture must be defined that contains the high-priority decisions regarding the software design in order to develop a software design.

Architectural Descriptors containing models and architectural views are part of the Model-Driving Architecture Framework.

Models, that can be used for tasks like code generation and analysis, often

follows MetaModels, which along withviewpoints address a specific group of concerns.

In Model-Driven architecture, models and views re usually connected using UML diagram. Views are created from the same models as the source code using the connection between the view and models. In this way, views become model driven.

A metamodel for a viewpoint can be defined based on an ADL that is about a related concern. A viewpoint description can be used and the metamodel relations and elements are generated.

Metamodels are normally created very easily due to the simplicity of the ADL syntaxes.

The mapping between the metamodel and UML is specified and implemented using model transformation languages. In order to transform an architectural model to a UML model, existing architectural model mappings to UML can be specified as model transformations.

The UML diagram doesn't represent the architectural model precisely, but it is suitable for communication purpose. This is why UML is so commonly used for architectural documentation in industry.

5.4.3 Dynamic Software Evolution

The diversity and complexity of the existing software systems have made software architecture more important.

Dynamic Software Architectures are the architecture descriptions that include the descriptions of the both fixed and changing parts. This means that the dynamic software architectures can react to specific requirements or events by run-time reconfiguration of its components and connections.

If the structure of the system is changed, the dynamism is of the dynamic reconfiguration type, and if the types that compose the structure are changed, it is of the dynamic evolution architectural types.

Dynamic reconfiguration is the first evolution type that is about run-time changes in the software structure. According to Endler and Wei [EW92], dynamic reconfiguration is the changing part of an application while it operates.

Magee and Kramer [MK96] define dynamic reconfiguration as a feature of ADLS used to describe and model software systems with a dynamic structure.

Dynamic Evolution of architectural types, which is the second type of dynamisism discussed in this report, can be used to change the type of an element

of architecture and its instances at runtime. This type is essentially required to create open systems or to update highly-available systems.

Dynamic Evolution of architectural types can introduce architectural types and links, can remove already existing architectural types, and can modify such architectural types at run-time. Instances of the above-mentioned modification include updating a component type or a connector type and the migration of the state of all the instantiations of an architectural type.

It is important to note that the dynamic reconfiguration acts at the configuration level, which the dynamic evolution of architectural types acts at the type level. In specific cases such as changing the topology of a software, both kinds of dynamism are required together.

5.5 Conclusion

The goal of the research presented in this study is to investigate and understand software evolution architecture to reduce the risks and costs involved in the evolution of software architectures.Building the evolvability models are are the first steps towards understanding and studying evolvability.Software architecture evolution is an inevitably fact due to many changes based on technological and business subjects over the life time of softwares.

Bibliography

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[BDW99] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91–121, 1999.

[BM99] E. Burd and M. Munro. An initial approach towards measuring and characterising software evolution. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 168–174. IEEE, 1999.

[BWL99] L.C. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 475–482. IEEE, 1999.

[CM03] D. Cubranic and G.C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408–418. IEEE, 2003.

[CS11] C. Costa Soria. Dynamic evolution and reconfiguration of software architectures through aspects. 2011.

[EHK06] A.H. Eden, Y. Hirshfeld, and R. Kazman. Abstraction classes in software design. *IEE Proceedings Software*, 153(4):163, 2006.

[EW92] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Configurable Distributed Systems, 1992., International Workshop on*, pages 68–79. IET, 1992.

[FB99] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[FHK+97] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Muller, J. Mylopoulos, S.G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[FO00] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *Software Engineering, IEEE Transactions on*, 26(8):797–814, 2000.

[FPG03] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 90. IEEE Computer Society, 2003.

[GHJ98] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 190–198. IEEE, 1998.

[GJK03] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 13–23. IEEE, 2003.

[GL00] M.W. Godfrey and E.H.S. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools*, 2000.

[GM03] N. Gold and A. Mohan. A framework for understanding conceptual changes in evolving source code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 431–439. IEEE, 2003.

[HNS00] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley Professional, 2000.

[JB05] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 109–120. IEEE, 2005.

[KC99] R. Kazman and S.J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[Kru04] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[KS03] R. Koschke and D. Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 36. IEEE Computer Society, 2003.

[Lan01] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th international workshop on principles of software evolution*, pages 37–42. ACM, 2001.

[LB85] M.M. Lehman and L.A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 3–14. ACM, 1996.

[MNS95] G.C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 18–28. ACM, 1995.

[MWD$^+$05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE, 2005.

[PGJ05] M. Pinzger, H. Gall, and M. Jazayeri. Archview–analyzing evolutionary aspects of complex software systems. *Vienna University of Technology*, 2005.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Riv04] C. Riva. *View-based software architecture reconstruction*. 2004.

[RW02] J.E.B.K.V. Riediger and A. Winter. Gupro–generic understanding of programs. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[SG96] M. Shaw and D. Garlan. Software architecture: perspectives on an emerging discipline. 1996.

[TMD09] R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[VHJG95] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 1995.

[YMNCC04] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30(9):574–586, 2004.

[ZWDZ04] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 563–572. IEEE, 2004.

Chapter 6

Software Architecture Reconstruction

Zheng Zhou

Contents

6.1 Introduction . . . 102
6.1.1 Architecture Description . . . 102
6.1.2 Basic Notions . . . 103
6.2 Software Architecture Reconstruction Goals . . . 104
6.3 Architecture Reconstruction Process . . . 105
6.4 SAR Inputs . . . 109
6.4.1 Nonarchitectureal Inputs . . . 109
6.4.2 Architectural Inputs . . . 110
6.5 Reconstruction Techniques . . . 111
6.5.1 Quasi-Manual Techniques . . . 111
6.5.2 Semi-Automatic Techniques . . . 111
6.5.3 Quasi-Automatic Techniques . . . 113
6.6 SAR Outputs . . . 113
6.6.1 Visual Software Views . . . 114
6.6.2 As-Is Architecture Description . . . 114
6.7 Model Driven Architecture Reconstruction . . . 114
6.8 Summary and Future Research Opportunities . . . 117
Bibliography . . . 117

Abstract: This seminar paper summarizes the current software architecture reconstruction techniques and methods. The state of the art techniques and methods relates to the different viewpoints that have been proposed in architecture design. Using model driven architecture helps software architecture reconstruction as whole process. Software architecture reconstruction (SAR) is the form of reverse engineering. SAR gathers information from the domain knowledge, source code, source code documentation, system's execution, stakeholder interviews.

6.1 Introduction

Software architecture reconstruction is the process of obtaining a documented architecture for an existing software system. Software architecture reconstruction is a reverse engineering approach, and its purpose is to rebuild a viable software application architecture view. The documented architecture includes domain knowledge, available documentation, stakeholder interviews. Ducasse et al. present deep and large survey of work using process oriented taxonomy [DP09] and my study of this seminar paper is based on it.

The structure of this paper is as follows. In this paper, a conceptual review for architecture reconstruction is introduced. Section 2 describes the goals of software architecture reconstruction. Section 3-6 cover the inputs, process, techniques and output of architecture reconstruction. Before conclusion, Section 7 surveys model driven architecture reconstruction as well.

6.1.1 Architecture Description

Architecture descriptions document the intended architecture. The definition for software architecture is given by the standard IEEE Recommended Practice for Architectural Descriptions [MEH01]: "Software architecture is the fundamental organization of system embodied in its components, the relationship of those components to each other and to the environment, and the principles guiding its design and evolution." Thus, software architecture and its components and the way they interact each other are important for comprehending the software system. Many approaches have been developed from different aspects to help comprehend software systems. To understand a software system as a whole, a summary of different views and domain knowledge of it can be helpful.

" The software architecture is the structure or structures of the system that is consisted of software elements and external visible properties of these elements and the relationships among them" [BCK03]. The definition implies that there are always several views of architectural structures. Each structure is modelled by means of software elements (e.g. components) and their

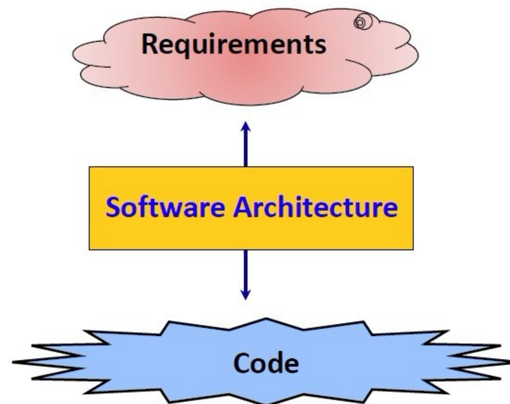

Figure 6.1: Development with Architecture.

relationships. All external properties of all software elements needed for interaction of the components have to be defined. Figure 6.1 shows that the architecture is very important in the development process and has the decisive role in software qualities. Therefore, changes in software systems need a global understanding of the system to be changed. Such changes influence software qualities. The first law of Lehman's software evolution [LL96] present about continuing change that a problem is used and as an implementation of its specification reflects some reality, undergoes continual change or becomes progressively less useful. Hence, the system will change over its life time. Consequently, a software architecture must cope with changes.

Instead of defining a merger set of described architecture structures, the IEEE Standard P1471 [MEH01] recommend to describe a software system by several views. A view of an architecture describes it from a particular stakeholder's perspective. For an explicit interpretation, the content of views must be defined clearly and exactly.

6.1.2 Basic Notions

- Views and Viewpoints

  $View$ View is a representation of the whole system from the perspective of a related set of concerns. To correlate domain information and existing software systems, different viewpoints are considered and modelled.

  $Viewpoint$ Viewpoint is a form of abstraction achieved using a selected set of architectural constructs and structuring rules, in order to focus on particular concerns within a system. Viewpoints guide the extraction of architectural views, later the different system facets would be representing. Architectural domain knowledge expresses as viewpoints in order

to guide the extraction process [BH92]. It provides multiple architectural views according to multiple given viewpoints. Kruchten [Kru95] proposed the very good examples of viewpoint are the logical, physical, process and development viewpoints.

- Conceptual Architecture

  Conceptual Architecture is the architecture that is concepted in human mind or in the software documentation. In the literature, concepted architecture can be also named as logical, as-designed, intended or idealized architecture [TH99].

- Concrete Architecture

  Concrete Architecture is the architecture that stems from the source code. It can also be named as the physical, realized, as-built or as-implemented architecture [KC99] [TH99].

- Software Architecture Reconstruction (SAR) As a reverse engineering method, *Software Architecture Reconstruction* targets at the reconstruction of viable architectural views of software system or software application. It is also known as the discovery, recovery, mining, architecture extraction or reverse architect [ME03]. The term discovery refers to a top-down process and the term recovery refers to a bottom-up process. SAR usually involves three steps: extracting raw data on the system, applying the appropriate abstraction technique, and visualizing or presenting the obtained information.

6.2 Software Architecture Reconstruction Goals

Redocumentation and Understanding

One of the main goal of software architecture is to provide architectural views, which describes an abstraction of software system and provides understanding of its overall design [vDHK$^+$04] [SG01].

Reuse Investigation and Product Line Migration

Architectural descriptions facilitate reuse at multiple levels. Besides, the reuse of large or small components and frameworks, into which components can be integrated, can be facilitated by architecture design. In software, product line makes it possible to share commonalities among products, as well as to customize products. Architectural views can help identifying commonalities and variabilities among products in a line [EOGB98] [SO01].

Conformance and Coevolution

$Software\ Architecture\ Reconstruction$ helps in the evolution of a software application by means of conformance checking. It can examine the conformance between the concepted and the actual architectures [MNS95] [Mur96]. DiscoTect [YGS+04], DAMRAM [ME03], Focus [DM01] [MJ06], SARTool [FKVO98] and Symphony [vDHK+04] are good example to check conformance within $Software\ Architecture\ Reconstruction$.

Architectural views shows clearly the actual constraints and indicates the better expected trend/direction of evolution for the software application. As two levels of abstraction, architecture and implementation evolve with different paces. In a ideal situation, these two levels of abstraction should be kept synchronise, so that architectural drift can be avoided [DP09] [Gar00].

Analysis

The application of new analyses is based on the high abstraction level of architectural view. Examples of these new useful analysis include quality attribute analysis, dependence analysis or style conformance analysis. An analysis framework can possibly control a $Software\ Architecture\ Reconstruction$ framework, as it offers architectural views to compute architectural quality analysis. As a result, such analysis frameworks help stakeholders in the process of decision making [SOV03] [SROV06] [DP09] [Gar00].

Evolution and Maintenance

The success of the development task depends on which extent is the view of the software system clear to the developers. Software architecture reconstruction provides the first step to software evolution and maintenance [Gar00].

6.3 Architecture Reconstruction Process

This section describes the architecture reconstruction process which is based on views and viewpoints. Software architecture reconstruction usually involves three steps: extracting raw data on the system, applying the appropriate abstraction technique, and visualizing or present the obtained information. The process can be performed in bottom up, top down, or hybrid. A hybrid process is conducted in two major phases: reconstruction design and reconstruction execution. These two phases are highly incremental and iterative.

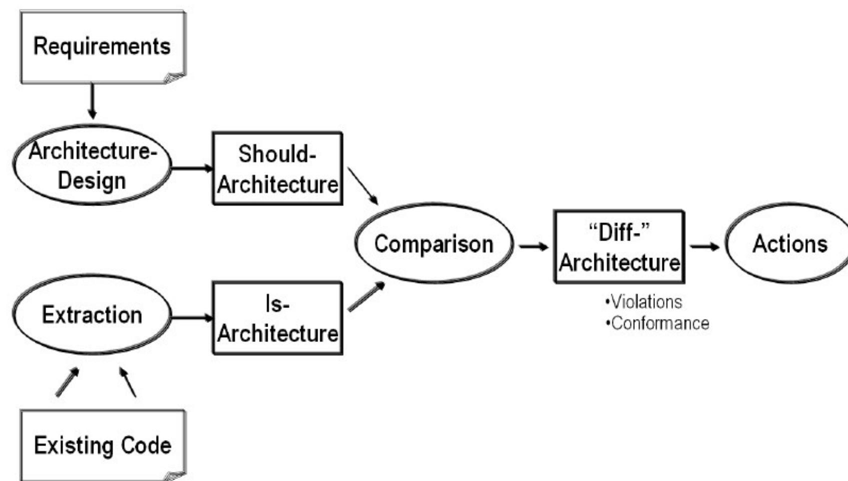

Figure 6.2: Reflexion model: is- and should-architecture.

- Bottom Up Process

  Bottom up process in figure 6.3 starts from low level knowledge to recover architecture. The level of abstraction from the source code model raise gradually until it reaches a high level understanding of the application. The analysis of the source code to fill in a repository, which is queried to abstract system representations, then presented in an appropriate interactive form to reverse engineers.

- Top Down Process

  The top down process in figure 6.4 begins with high level knowledge such as requirements or architectural styles. The goal is to find the architecture through formulating conceptual hypotheses and matching them to the source code.

- Hybrid Process

  Hybrid process in figure 6.5 is the combinations of bottom up with top down processes. The hybrid process conducts in two major phases: reconstruction design and reconstruction execution. These two phase are highly incremental and iterative [Kos08] .

- Reconstruction Design

  Reconstruction design from Symphony by Koschke [Kos08] in figure 6.6 determines the problem, to plan the reconstruction by defining source, to set the context and target views using viewpoint with the mapping rules.

- Reconstruction Execution

  The reconstruction execution phase from Symphony by Koschke [Kos08] in figure 6.7 executes the plan, analyses the system with reverse en-

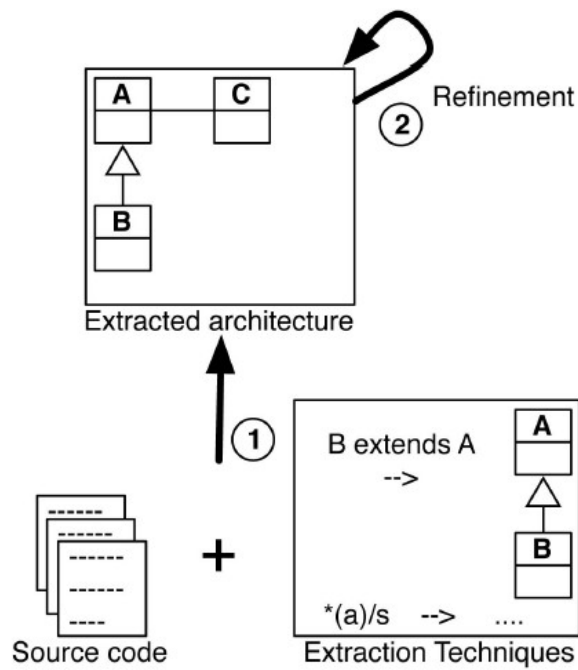

Figure 6.3: A bottom up process: From the source code, (1)views are extracted and (2) refined [DP09].

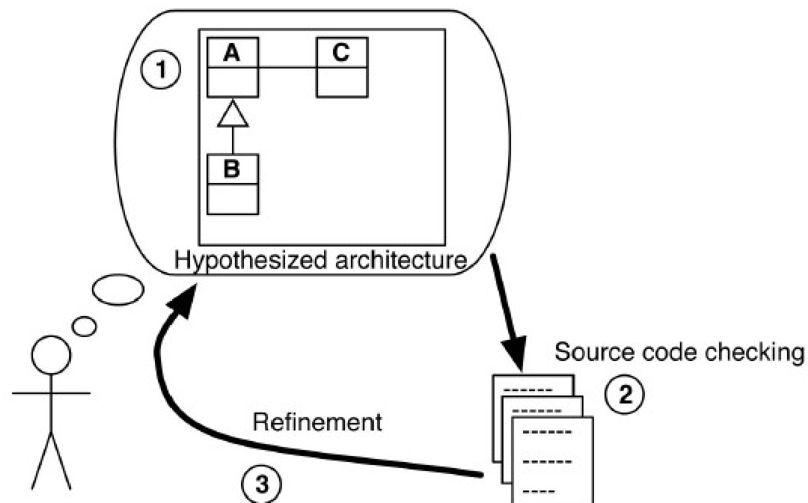

Figure 6.4: A top down process. (1) A hypothesized architecture is defined. (2) The architecture is checked against the souce code. (3) The architecture is refined [DP09].

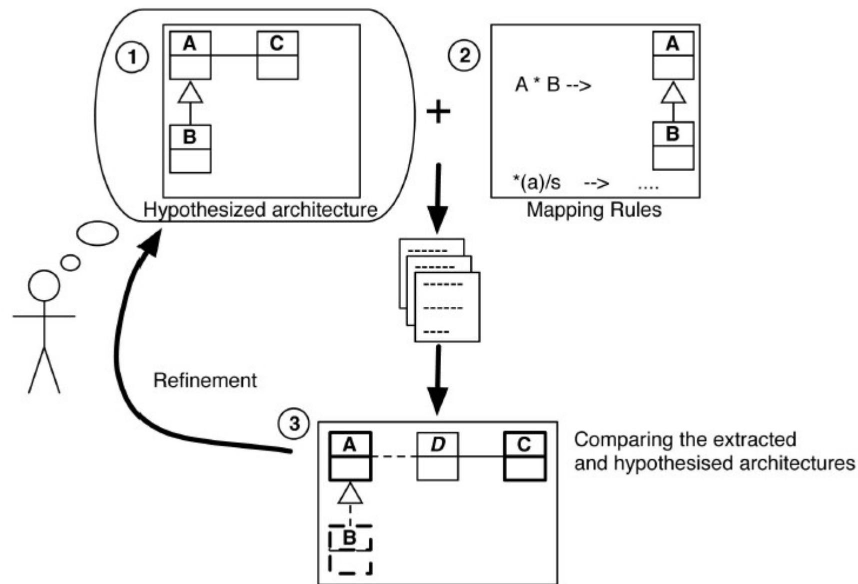

Figure 6.5: The Reflexion Model, a top down process. (1) A hypothesized architecture is defined. (2) Rules map source entities to architectural elements. (3) RM tool compares the extracted and hypothesized architectures and the process iterates [DP09].

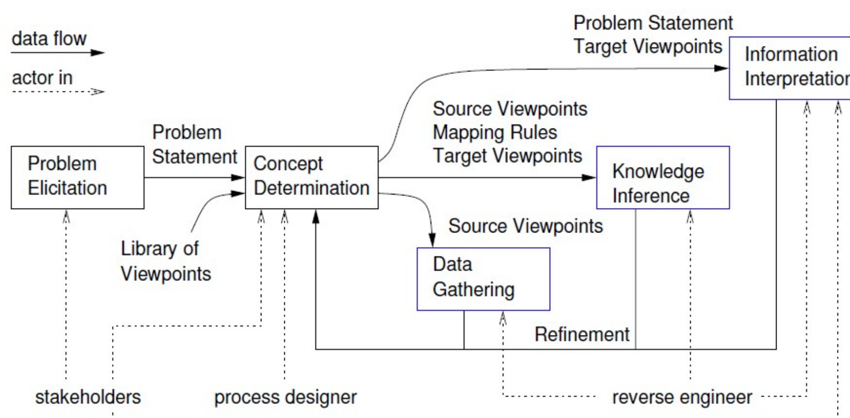

Figure 6.6: Activities in Reconstruction Design [Kos09].

gineering, extracts the source views and applies the mapping rules to propagate the target views.

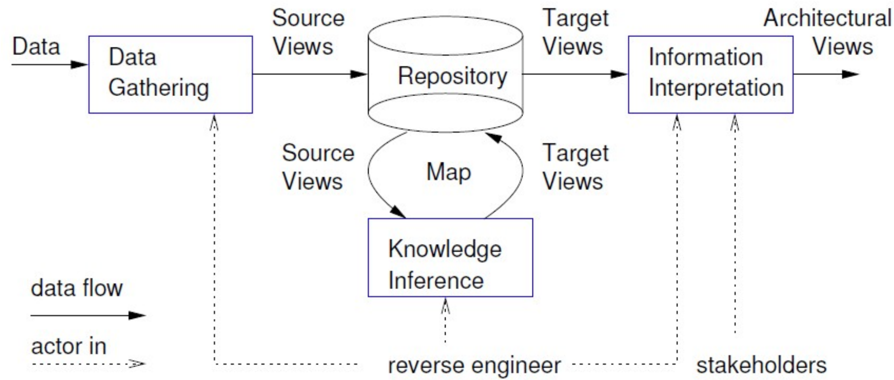

Figure 6.7: Activities in Reconstruction Execution [Kos09].

6.4 SAR Inputs

6.4.1 Nonarchitectureal Inputs

- Source Code Constructs

  Most applications consider the source code as a ubiquitous trustworthy source of information. Some of the approaches also directly integrate the source code within regular expressions like Revealer[PG02] or RMTool[FKVO98].

- Symbolic Textual Information

  The symbolic information available in the comments[PFGJ02] [PG02] or in the method names[MSRM04] [KDG07] are utilized in some approaches.

- Dynamic Information

  Static information supplies with only a limited vision into the runtime intent of the analysed software. Therefore, it is insufficient for SAR. As more relevant dynamic information is needed [LN95], dynamic information is even exclusively alone utilized in some $Software\ Architecture\ Reconstruction$ approaches. [YGS+04] Other approaches mix static knowledge and dynamic information [RR02] [RD99] [QHS+05] [HMY06]. Dynamic information is illustrated by Walker et al. [WMSR00] to architectural views. In a lot of approaches, dynamic information is used to extract design views [Sys99] [RD99] [KJG+05] [HLL04] [HLBAL05].

- Physical Organization

  Architectural information can often be sketched out by observing the physical organisation of application in terms of files. For example, mapping classes or packages to components and using hierarchical physical organization in some approaches as architectural input [LSP05] [PFG04] [Wuy98].

- Human Organization

  It is written in Conway's thesis that the communication structure of an organization will be reflected in the system design by this organisation. Thus the human organization can have an influence on the extracted architectures or views [Con68] [BH98].

- Human Expertise

  Although human knowledge is not totally trustworthy, it can be helpful when it is available. $Software\ Architecture\ Reconstruction$ is iterative at high abstraction levels and thus human knowledge is needed to guide it and to validate its results [MNS95] [MJ06] [HH04].

6.4.2 Architectural Inputs

As architectural styles and viewpoints are the primitives of software architecture, they are necessary to be analyse, if $Software\ Architecture\ Reconstruction$ approaches take them for inputs to control the extraction process.

- Styles

  Like design patterns, the architecture styles represent recurrent architectural situations and they are very common. Examples are like pipes and filters, data flow and layered system. In a top down process, requirements can be helpful as high level knowledge to discover the conceptual architecture. In a bottom up process, system implementation can be used to recover the concrete architecture [ME03].

- Viewpoints

  The system architecture works as a mental model that is shared among stakeholders [Hol02] . The diverse interests of the stakeholders decide the importance that $Software\ Architecture\ Reconstruction$ needs to consider the different viewpoint of stakeholders [IEE00] [SHI$^+$01].

- Mixing Inputs

  Although multiple inputs are needed for the generation of different architectural views, most approaches need to work under conditions of a limited source of information [KC99] [LN95]. Knodel et al. [KJG$^+$05] describes the mixing inputs such like source code, documents and historical data.

6.5 Reconstruction Techniques

The comprehensive summary of existing architecture reconstruction techniques will be listed. There are a lot of technologies and approaches supporting architecture reconstruction. A variety of formalisms that are used to represent, query, and exchange into three automation levels will be introduced [DP09].

6.5.1 Quasi-Manual Techniques

$Quasi - manual$. The reverse engineers uses tools manually to identify architectural elements and can assist them to understand their findings [DP09].

Construction-Based Techniques

These techniques manually abstract low level knowledge and thereby reconstruct the software architecture, by means of interactive and expressive visualization tools like GraphViz [GN00], CodeCrawler [LD03], Verso [LSP05], Rigi [MTW93] [KM10], or 3D [MSRM04].

Exploration-Based Techniques

These techniques guides reverse engineers with the implementation tools like Softwarenaut [LLG06] and in this way provides them an architectural views of the system. The architectural view is thus closely associated with the developer's view.

6.5.2 Semi-Automatic Techniques

$Semi - automatic$. The reverse engineers manually use tool to automatically discover refinement or recover abstractions. Semi-automatic techniques automatize the repetition perspectives in $Software\ Architecture\ Reconstruction$. The reverse engineer controls the iterative refinement or abstraction and eventually identifies architectural elements.

Abstraction-Based Techniques

The goal of these techniques is to mapping low-level concepts with high level concepts. Reverse engineers identify specific reusable abstraction rules and

apply them automatically. The following process will be listed:

Relational queries. Out of entity-relation databases, data abstraction can often be done by relational algebra engines. Example like ARMIN [OSL05] and Dali to define grouping rules with SQL queries [KC99].

Logic queries. These queries are powerful, as their underlying unification mechanism makes it possible to write close multivalued queries. Wuyts [Wuy98], Kramer and Prechelt [KP96] identify design patterns with Prolog queries.

Programs. Analyse are built in some approaches as in plain object oriented programs. For example, Nerstrasz et.al [NDG05] expressed groupings made in Moose as object oriented problem to manipulate model represents different input.

Lexical and structural queries. The lexical and structural information in the source code provides directly foundation for some approaches [Mur96] [MNS95] [PFGJ02] [PG02] .

Investigation-Based Techniques

These techniques mapping high level concepts with low level concepts. The high level concepts mentioned here include a wide range from architectural descriptions and styles to features and design patterns. The following approaches will be listed:

Recognizers. Tools such as ART [FATM], Cliche[FTAM96], ARM [GAK99] and ManSART [HRY95] [YHC97] rely on recognizers for architectural styles or pattern as basis, which are written in query language. These tools inform the source code elements in a accordance with the recognized structures.

Graph pattern matching. It is possible to transform pattern definitions into graph pattern in ARM [GAK99] to match a graph-based source code representation. It is similar to Alborz's method [Sar03].

State engine. State engine are used to examine architectural styles conformance in DiscoTect [YGS^{+}04]. The state machine description satisfies the system execution at run time and outputs architectural events.

Maps. On basis of the Reflexion Model [Mur96] [MNS95] use rules, Software Architecture Reconstruction approaches mapping hypothesized high level entities with source code entities [CTH95].

6.5.3 Quasi-Automatic Techniques

$Quasi - automatic.$ The tool takes the control and the reverse engineer drives the iterative recovery process and most automated approaches. Concept, dominance and clustering algorithms are often combined in approaches in this area.

Concepts

Formal concept analysis can be defined as a branch of lattice theory to recognize modules [SR99] , features [GD05] [EKS03] or design patterns [ABN04].

Cluster analysis techniques

Clustering Algorithms recognize groups of objects with similar member in some way. They are applied to produce software views of applications [AL99] [AFL99] [Wig97].

Dominance

Dominance analysis recognizes the connected parts in an application in software maintenance. Within the framework of software architecture extraction, in accordance with Koschke's paper, Trifu combines cluster and dominance analysis techniques, in order that architectural components can be retrieved in object oriented legacy systems [Tri01] [CV95] [GK97].

Layers and Metrics

Applications are constructed with layers in conception: the lower layers should not exchange information with the upper ones. Dependency Structure Matrix (DSM) identifies cycles and layers in large applications. The dependency structure matrix is adapted from process domain management to analyse architectural dependencies in software [SJSJ05] [SGCH01].

6.6 SAR Outputs

Most approaches mentioned above concentrate on the recognition and presentation of software architectures. There are some others that provides valuable

additional information, for example, conformance of architecture and implementation. The SAP Goals and outputs are obviously related.

6.6.1 Visual Software Views

Supporting Visualization Tools

Architecture as boxes. Source code entities are presented and grouped as boxes in source visualization approaches with the tool like ArchVis, GraphViz or CodeCrawler[Sar03] [PG02] [KC99] [FHK^{+}97].

Source entity visualization. Adjacent to true architectural entities, some tools concentrate on source code visualization or abstractions. For example, condensed views of software source code entities are presented in CodeCrawler [LD03], Distribution Map [DGK06], and Package Blueprints [ADPA10] . Likewise Verso uses 3D to integrate more information per entity .

Architectural views. Enhanced views with architectural information [LLG06] are provided by some tools. In this case, some approaches use 2D/3D [FDJ98] [MFM03] for the improvement of their visualizations. Erben and Loehr specify dedicated tool support to illustrate architectural elements and layers [EL05].

6.6.2 As-Is Architecture Description

Good SAR approaches provide better understanding and better architectural views to stakeholders. With the evolution of codes, some approaches concentrates on the coevolution of the reconstructed architecture. With its implementation the architecture are intensively synchronized and the differences due to evolution are highlighted [MKPW06]. Some approaches lay their emphasis on identifying conformance of an application to a certain architecture. There are two kinds of architecture conformance: vertical conformance at different abstraction levels and horizontal conformance between similar abstractions [Men00].

6.7 Model Driven Architecture Reconstruction

The reconstruction process and techniques are presented in the first half of this paper. Here we present a abstract perspective on software architecture reconstruction with model driven software engineering.

Because model driven software engineering has the potential to cover the

whole spectrum of software engineering process, the integrating model driven engineering and software architecture reconstruction is much more promising. Software architecture reconstruction can just be seen as a particular model driven engineering process.

Favre [Fav04] present a generic metamodel driven process for software architecture reconstruction called Cacophony. In Cacophony, the metamodels are keys for representing viewpoint. The differences between Model and Metamodel and View and Viewpoints are key to understand how model driven engineering and software architecture can be integrated.

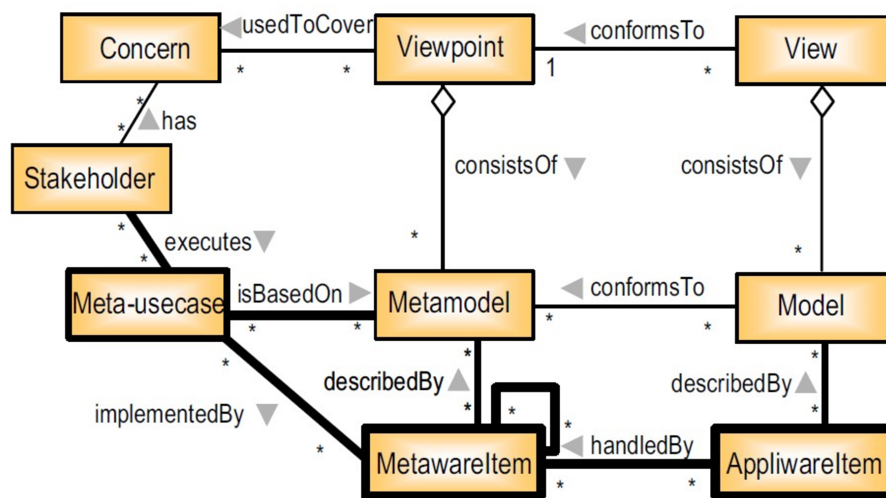

Figure 6.8: A megmodel for architecture recovery [Fav04]

The overall structure of the Cacophony process in figure 6.9 can be described in six steps, as a whole, they are by nature iterative and incremental.

Buchgeher et al. [BW09] present an Architecture Description Languages (ADLs) approach called LISA (Language for Integrated Software Architecture). Based on LISA architecture models, components and system can be modelled on the system layer with high-level abstraction, low-level architectural abstraction (class and interfaces in object-oriented software) is connected to the high-level abstraction during system modelling.

Callo Arias et al. [CAAA+11] present an approach based on metamodel to construct execution views of large software intensive system. The execution views show the software system at runtime with dynamic analysis technique. The source of information in the metamodel (set of concepts and relationships between them) can be used to describe the runtime of system.

	Step	Sub activities
I	**Metaware domain and asset analysis** (section 5.1)	Metaware inventory Metamodels recovery Metamodels integration Metamodel clustering Metamodel packaging
II	**Metaware requirement analysis** (section 5.2)	Meta-level actors identification Meta-level use cases identification Metaware assessment Metaware improvement analysis Meta-level use cases description
III	**Metaware specification** (section 5.3)	Meta-model filtering and extension Presentation specification Metaware specification packaging
IV	**Metaware implementation**	Extractors development and reuse Viewers development and reuse Extractors and Viewers integration
V	**Metaware execution**	Deployment Execution Monitoring
VI	**Metaware evolution**	Evaluation Feedback Change analysis

Figure 6.9: The Cacophony metamodel-driven process [Fav04]

6.8 Summary and Future Research Opportunities

This paper presents a state of the art in software architecture reconstruction approaches. Software architecture are mostly not documented sufficiently. When we change the system, the architecture description must be reconstructed.

For maintenance and understanding of large applications, it is important to know their architecture. The first problem is that unlike classes and packages, architecture is not explicitly represented in the code. The second problem is that successful applications evolve over time, so their architecture inevitably drifts. Reconstructing the architecture and checking whether it is still valid is therefore an important aid. Future research should better address the discrepancies.

Bibliography

[ABN04] Gabriela Arevalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 122–131, Washington, DC, USA, 2004. IEEE Computer Society.

[ADPA10] Hani Abdeen, Stéphane Ducasse, Damien Pollet, and Ilham Alloui. Package fingerprints: A visual summary of package interface usage. *Inf. Softw. Technol.*, 52(12):1312–1330, December 2010.

[AFL99] Nicolas Anquetil, Cédric Fourrier, and Timothy C. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 235–, Washington, DC, USA, 1999. IEEE Computer Society.

[AL99] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, May 1999.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[BH92] Erich Buss and John Henshaw. Experiences in program understanding. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research - Volume 1*, CASCON '92, pages 157–189. IBM Press, 1992.

[BH98] Ivan T. Bowman and Richard C. Holt. Software architecture recovery using conway's law. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '98, pages 6–. IBM Press, 1998.

[BW09] Georg Buchgeher and Rainer Weinreich. Connecting architecture and implementation. In Robert Meersman, Pilar Herrero, and Tharam S. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Confederated International Workshops and Posters, ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009, Vilamoura, Portugal,*, volume 5872 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2009.

[CAAA+11] Trosky B. Callo Arias, Paris Avgeriou, Pierre America, Krelis Blom, and Sergiy Bachynskyy. A top-down strategy to reverse architecting execution views for a large and complex software-intensive system: An experience report. *Sci. Comput. Program.*, 76(12):1098–1112, December 2011.

[Con68] M.E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[CTH95] I. Carmichael, V. Tzerpos, and R. C. Holt. Design maintenance: unexpected architectural interactions (experience report). In *Proceedings of the International Conference on Software Maintenance*, ICSM '95, pages 134–, Washington, DC, USA, 1995. IEEE Computer Society.

[CV95] Aniello Cimitile and Giuseppe Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28(2):117–127, 1995.

[DGK06] Stephane Ducasse, Tudor Girba, and Adrian Kuhn. Distribution map. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 203–212. IEEE Computer Society, 2006.

[DM01] Lei Ding and Nenad Medvidovic. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 191–, Washington, DC, USA, 2001. IEEE Computer Society.

[DP09] Stephane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Softw. Eng.*, 35(4):573–591, July 2009.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, March 2003.

[EL05] N. Erben and K. Lohr. Sab - the software architecture browser. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '05, pages 7–, Washington, DC, USA, 2005. IEEE Computer Society.

[EOGB98] Wolfgang Eixelsberger, Michaela Ogris, Harald Gall, and Berndt Bellay. Software architecture recovery of a program family. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 508–511, Washington, DC, USA, 1998. IEEE Computer Society.

[FATM] R. Fiutem, G. Antoniol, P. Tonella, and E. Merlo. Art: An architectural reverse engineering environment.

[Fav04] Jean-Marie Favre. Cacophony: Metamodel-driven architecture recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 204–213, Washington, DC, USA, 2004. IEEE Computer Society.

[FDJ98] Loe Feijs and Roel De Jong. 3d visualization of software architectures. *Commun. ACM*, 41(12):73–78, December 1998.

[FHK+97] P. Finnigan, Ric C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[FKVO98] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Softw. Pract. Exper.*, 28(4):371–400, April 1998.

[FTAM96] R. Fiutem, P. Tonella, G. Anteniol, and E. Merlo. A cliche-based environment to support architectural reverse engineering. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, WCRE '96, pages 277–, Washington, DC, USA, 1996. IEEE Computer Society.

[GAK99] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A software architecture reconstruction method. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, WICSA1, pages 15–34, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[Gar00] David Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM.

[GD05] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, CSMR '05, pages 314–323, Washington, DC, USA, 2005. IEEE Computer Society.

[GK97] Jean-Francois Girard and Rainer Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 58–65, Washington, DC, USA, 1997. IEEE Computer Society.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, September 2000.

[HH04] Ahmed E. Hassan and Richard C. Holt. Using development history sticky notes to understand software architecture, 2004.

[HLBAL05] Abdelwahab Hamou-Lhadj, Edna Braun, Daniel Amyot, and Timothy Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, CSMR '05, pages 112–121, Washington, DC, USA, 2005. IEEE Computer Society.

[HLL04] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '04, pages 42–55. IBM Press, 2004.

[HMY06] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engg.*, 13(2):257–281, April 2006.

[Hol02] R. Holt. Software architecture as a shared mental model. *Proceedings of the ASERC Workhop on Software Architecture, University of Alberta*, 2002.

[HRY95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 186–195, New York, NY, USA, 1995. ACM.

[IEE00] IEEE. Working group. ieee recommended practice for architectural description of software-intensive systems. *IEEE Architecture*, pages 1471–2000, 2000.

[KC99] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engg.*, 6(2):107–138, April 1999.

[KDG07] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, March 2007.

[KJG+05] Jens Knodel, Isabel John, Dharmalingam Ganesan, Martin Pinzger, Fernando Usero, Jose L. Arciniegas, and Claudio Riva. Asset recovery and their incorporation into product lines. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 120–129, Washington, DC, USA, 2005. IEEE Computer Society.

[KM10] Holger M. Kienle and Hausi A. Müller. Rigi-an environment for software reverse engineering, exploration, visualization, and redocumentation. *Sci. Comput. Program.*, 75(4):247–263, April 2010.

[Kos08] Rainer Koschke. Architecture reconstruction. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, volume 5413 of *Lecture Notes in Computer Science*, pages 140–173. Springer, 2008.

[Kos09] Rainer Koschke. *Architecture reconstruction. Tutorial on reverse engineering to the architectural level.*, pages 140–173. Berlin: Springer, 2009.

[KP96] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, WCRE '96, pages 208–, Washington, DC, USA, 1996. IEEE Computer Society.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, November 1995.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, September 2003.

[LL96] M M Lehman and Mm Lehman. Laws of software evolution revisited. In *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[LLG06] Mircea Lungu, Michele Lanza, and Tudor Girba. Package patterns for visual architecture recovery. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[LN95] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Pro-*

ceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '95, pages 342–357, New York, NY, USA, 1995. ACM.

[LSP05] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 214–223, New York, NY, USA, 2005. ACM.

[ME03] Nenad Medvidovic and Alexander Egyed. Stemming architectural erosion by coupling architectural discovery and recovery, 2003.

[MEH01] Mark W. Maier, David E. Emery, and Rich Hilliard. Software architecture: Introducing ieee standard 1471. *IEEE Computer*, 34(4):107–109, 2001.

[Men00] Kim Mens. *Automating Architectural Conformance Checking by Means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 2000.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 27–ff, New York, NY, USA, 2003. ACM.

[MJ06] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engg.*, 13(2):225–256, April 2006.

[MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views. *Comput. Lang. Syst. Struct.*, 32(2-3):140–156, July 2006.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[MSRM04] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[MTW93] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference*

of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1, CASCON '93, pages 217–226. IBM Press, 1993.

[Mur96] Gail C. Murphy. Lightweight structural summarization as an aid to software evolution, 1996.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 1–10, New York, NY, USA, 2005. ACM.

[OSL05] Liam O'Brien, Dennis Smith, and Grace Lewis. Supporting migration to services using software architecture reconstruction. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, STEP '05, pages 81–91, Washington, DC, USA, 2005. IEEE Computer Society.

[PFG04] Martin Pinzger, Michael Fischer, and Harald Gall. Towards an integrated view on architecture and its evolution. In *In Proceedings of the Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra¡¯04). Elsevier Science Publishers*, page 2005, 2004.

[PFGJ02] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 170–, Washington, DC, USA, 2002. IEEE Computer Society.

[PG02] Martin Pinzger and Harald Gall. Pattern-supported architecture recovery. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 53–, Washington, DC, USA, 2002. IEEE Computer Society.

[QHS+05] Li Qingshan, Chu Hua, Hu Shengming, Chen Ping, and Zhao Yun. Architecture recovery and abstraction from the perspective of processes. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 57–66, Washington, DC, USA, 2005. IEEE Computer Society.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 13–, Washington, DC, USA, 1999. IEEE Computer Society.

[RR02] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, CSMR '02, pages 47–, Washington, DC, USA, 2002. IEEE Computer Society.

[Sar03] Kamran Sartipi. Software architecture recovery based on pattern matching. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 293–, Washington, DC, USA, 2003. IEEE Computer Society.

[SG01] Davor Svetinovic and Michael Godfrey. A lightweight architecture recovery process, 2001.

[SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. *SIGSOFT Softw. Eng. Notes*, 26(5):99–108, September 2001.

[SHI+01] Kari Smolander, Kimmo Hoikka, Jari Isokallio, Mika Kataikko, Teemu Maekelae, and Heikki Kaelviaeinen. Required and optional viewpoints – what is included in software architecture?, 2001.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *SIGPLAN Not.*, 40(10):167–176, October 2005.

[SO01] Christoph Stoermer and Liam O'Brien. Map - mining architectures for product line evaluations. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 35–, Washington, DC, USA, 2001. IEEE Computer Society.

[SOV03] Christoph Stoermer, Liam O'Brien, and Chris Verhoef. Moving towards quality attribute driven software architecture reconstruction. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 46–, Washington, DC, USA, 2003. IEEE Computer Society.

[SR99] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *IEEE Trans. Softw. Eng.*, 25(6):749–768, November 1999.

[SROV06] Christoph Stoermer, Anthony Rowe, Liam O'Brien, and Chris Verhoef. Model-centric software architecture reconstruction. *Softw. Pract. Exper.*, 36(4):333–363, April 2006.

[Sys99] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 304–, Washington, DC, USA, 1999. IEEE Computer Society.

[TH99] John Tran and Richard C. Holt. Forward and reverse repair of software architecture. In *Proceedings of the IBM CAS Conference*, 1999.

[Tri01] A Trifu. Using cluster analysis in the architecture recovery of object-oriented systems. 2001.

[vDHK+04] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*, pages 122–134. IEEE Computer Society, 2004.

[Wig97] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, WCRE '97, pages 33–, Washington, DC, USA, 1997. IEEE Computer Society.

[WMSR00] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '00, pages 12–. IBM Press, 2000.

[Wuy98] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, pages 112–, Washington, DC, USA, 1998. IEEE Computer Society.

[YGS+04] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *In Proc. 26th International Conference on Software Engineering*, pages 470–479, 2004.

[YHC97] Alexander S. Yeh, David R. Harris, and Melissa P. Chase. Manipulating recovered software architecture views. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 184–194, New York, NY, USA, 1997. ACM.

Chapter 7

Extraction of the static view of software architectures

Endri Gjino

Contents

7.1 Introduction 128
7.2 SAR Approaches and Frameworks 129
7.2.1 Practice Scenarios 129
7.2.2 The Reflexion Model 131
7.2.3 Symphony and CaCOphoNy 132
7.2.4 X-Ray Approach for Distributed Systems 135
7.3 Recovery Algorithms 135
7.3.1 Cluster-based 135
7.3.2 Pattern-based 138
7.4 Tools Overview 139
7.4.1 Classification 139
7.4.2 Tools 141
7.5 Visualization Techniques 145
7.5.1 Visualization Overview 145
7.5.2 Innovative visualization techniques 146
7.6 Conclusion 147
Bibliography 148

Abstract: Code comprehension for new and old developers, keeping the software in synch with the initial design, as new user requirements come in, checking for architectural violations, and retrieving architecture views targeted to specific stakeholders are crucial in the software development lifecycle. Architecture reconstruction has the potential to greatly help with the above problems. This paper is a survey of architecture reconstruction, based on static analysis. The paper explores the different approaches (eg. The Reflexion Model), algorithms (eg. Revealer) and tools (eg. the SAVE tool), from the early 2000 to today, on retrieving architectural views from source models. This paper is therefore ideal for scholars, new to the topic, that want a quick tour into the most influential proposals on extracting the static views, of the architecture of a software system.

7.1 Introduction

Software architecture is an abstraction of the software system. It comprises a set of views that target specific concerns of specific stakeholders. Managers are, for example, interested in the module view, describing the layering and subsystem organization, of the software architecture so that they can appropriately plan for resource distribution. A developer would be interested in the model view to get a highly abstract idea of the system or on the logic view, describing functional requirements in class diagrams and class templates, if he wants to create a clear picture of the classes of a specific subsystem. But software is in continuous evolution driven by changing requirements therefore it becomes paramount to be able to synchronize the software architecture description, if it was ever formally defined, with the current running implementation of the system. The benefits are clear. As development teams change continuously, new developers will need to get acquainted with the current architecture, new managers need to be able to estimate the effort needed for a new customer requirement and a VIP manager would be interested if he/she can come up with product lines that would eventually drive costs and development time down for the company. Software architecture reconstruction becomes thus essential and makes it possible that stakeholders can:

- Evaluate the conformance of the as-built architecture to the as-documented architecture and stop the architecture erosion [Mer10a] .
- Reconstruct the architecture descriptions that are poorly documented or for which documentation is not available.
- Analyze and understand the architecture of existing systems, to enable modification of the architecture to satisfy new requirements.
- Developers can continuously check during development that changes they make do not adversely effect the architecture, by for example breaking the layers or making components dependent on each other's internal working. [BW08]

In section 2 of this paper I will survey some approaches that have been proposed by the academic community on Software Architecture Reconstruction. I will describe from the very specific "The Practice Scenarios" proposed by [OSV02] to the more general approaches like the "Reflexion Model", 'CaCOphoNy", "Synphony" [Fav] [Kru95] and the "X-Ray Approach for Distributed Systems" [Men]. In section 3 I will describe shortly the algorithms that can be used to assist the user in recovering architecture views: the cluster-based approach by [CKS] and the patten-based approach like the one described by [Sar06] or the hybrid approach in "Revealer" [PFGJ02] that combines syntactic analysis with lexical analysis. Section 4 goes over some tools that are out there. Section 5 brings some visualization techniques that have been employed in some of the tools presented in the previous section and some new methods that have the potential to increase the information throughput without adding visual cluttering.

7.2 SAR Approaches and Frameworks

7.2.1 Practice Scenarios

The attempt to reconstruct software architecture starting from source code is not an easy job. The nature (eg. distributed) of the system, the use of different languages, the use of binary components from third party vendors that are not interested in revealing their code or the absence of solid documentation for such components are some of the problems faced by analysts trying to reconstruct the software architecture. Some other issues arise when trying to match the output terminology, concepts and layout of the reconstruction process with the stakeholders needs and background. [OSV02] tried to address these issues and others by compiling a set of practice scenarios that the software architect can apply according to the striking similarities with his task at hand.

The first, described by [OSV02], is the **View-Set Practice Scenario**. It tries to address the problem of identifying the appropriate architectural view that sufficiently describes the different stakeholders' needs. The output of this approach is a collection of views and the appropriate terminology that can act as a contract between the stakeholder and the analyst responsible for performing the reconstruction.

Another scenario that [OSV02] named **The Enforced-Architecture Scenario** addresses the problem of matching the as-build with the as-desired architecture. The output of this scenario concerns the identification of patterns in the design. The problematic addressed here is the absence of constructs in the programming language that directly implement architecture concepts such as layers for example. During the implementation phase of the development the

programmer sometimes applies his/her interpretation of such concepts in the respecting programming language driving focus out of the design, towards easiness of implementation. The writer stresses the importance of methods and tools used in the implementation that put emphasis toward implementation guidelines that would strengthen the bond between design and implementation decisions.

Another scenario, **The Quality-Attribute-Changes Scenario**, underlines the importance of identifying the contribution of certain architecture patters in satisfying the quality attributes of the system, which are sometimes competing with each other and driving design from one architecture patter to another. There is always a need for a tradeoff, driven by business goals the system should fulfill. This scenario stresses both the importance of singling out the architecture patterns that contribute to a certain quality attribute and the design tradeoffs. The solution to this need, according to [OSV02], should comprise a set of methods and tools that identify how a system quality attribute is fulfilled by architecture patterns.

The Common and Variable Artifacts Scenario addresses the need of identifying the common and variable parts in the software portfolio of an organization, as the first step in defining product lines, or the consolidation into one system of many software products used by a company. The solution according to [OSV02] should comprise methods and tools for identifying common and variable parts across products. The benefits are immediate since this analysis can be used as an argument to the company's management for product lines or for consolidating software products within the company into one.

The Binary Components Scenario, in [OSV02], deals with third party binary components which are becoming more and more used in the development lifecycle. Because the vendors ship usually in binary format to hide implementation details, very important here becomes the third party artifact description and interface. It addresses the problem of assessing the feasibility of using such components and assessing the conformance to the interface description. A common solution to this scenario, used in industry, is by building small proof-of-concept "toys" of the component to mitigate the risk when using them in the real product. Software architecture reconstruction of these components, starting from component description and not source code, would help the architects assess whether the binary component fulfills requirements and how it interacts with the other components.

The last scenario that [OSV02] looks into is **The Mixed-Language Scenario**. This scenario focuses of the needs for techniques to reconstruct the architecture views of a system implemented with heterogeneous languages and language types. The writer suggests that the reconstruction in such cases can be driven by elements other than language such as configuration files and directory structure. But more approaches are needed to combine the abstractions from the competing languages in common architecture views.

7.2.2 The Reflexion Model

The Reflexion, introduced by [MNS95] summarizes the concrete view, derived from the static analysis of source code, in terms of the hypothetical view, an abstraction of the system from the architect point of view. The concrete view contains concrete entities which are extracted from the tool ([MNS95] used awk) together with source dependencies derived from a class hierarchy, include headers ,method calls or even the source code directory structure. On the other hand the hypothetical view, also containing entities and relations between these entities, is manually constructed by the analyst reflecting the perceived system architecture and containing a lot of architecture patterns such as layers, subsystems, components etc. The next step in deriving the reflexion model is a manual or tool-assisted, user-driven mapping of concrete entities into hypothetical entities.

The result of the process is a graph with 3 disjoint types of edges connecting the hypothetical entities:

- **Convergence Edges:** when a connection between 2 hypothetical entities is also present in the mapped-to concrete entities.
- **Divergence Edges:** when a connection between 2 concrete entities is not present in the mapped-to hypothetical entities.
- **Absence Edges:** when a connection between 2 hypothetical entities is not present in the mapped-to concrete entities.

[MNS95] showed that the reflexion model can be computed relatively fast allowing for a user driven, iterative approach to building the entire overview of the system. The model was used to reconstruct the module view of NetBSD (a unix flavor). [MNS95] brings 3 examples that show the benefits of using the reflexion model in re-engineering, assessing design conformance and system understanding. A software engineer at Microsoft was able to quickly understand the architecture of Excel and plan accordingly re-factoring activities. The authors of [MNS95] were able to check whether layer constrains in the hypothetical view where also preserved in the concrete view (extracted from source) of Grisworld's reconstruction tool. They were also able to clarify, in another situation, why students at university of Washington had major difficulties in extending a compiler, which turned out, shown by a high number of divergence edges in the reflexion model, to suffer from a high degree of coupling. [CKS] extend the Reflexion Model with automatic clustering to ease the demanding activity of mapping concrete entities with hypothetical entities. Details about their approach will be given in the next section.

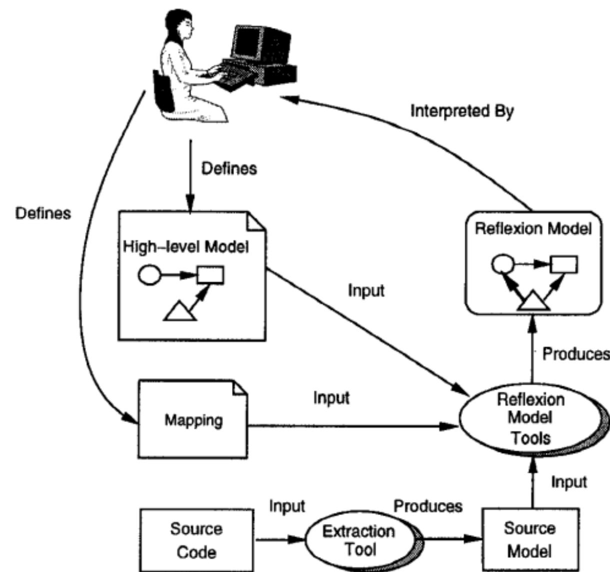

Figure 7.1: Reflexion Model [MNS95].

7.2.3 Symphony and CaCOphoNy

Symphony, introduced by [vDHK$^+$04], is a general method aiming at guiding the reconstruction process. It includes 3 main steps and several sub-steps that try to reconcile the stakeholder needs with the information presented in the views recovered by the reconstruction process.

Symphony, as described by [vDHK$^+$04], can be broken down into the following steps:

1. **Problem Elicitation**. In this step the different perspective of each interested stakeholder are consolidated in a big picture through workshops, checklists, role-playing and scenario analysis, into a two page memorandum describing the problem at hand.
2. **Concept Determination**. In this step the problems and motivations, identified earlier, are translated into views understood by the stakeholders.
   (a) First, potential useful viewpoints are identified, with the help of stakeholders and are later refined into target viewpoints and a prioritized list of views that the reconstruction process should generate, where candidate views, relationships between these views and their respective priorities are identified, and assigned in an iterative manner.
   (b) Second, source views containing only the information needed by the stakeholders are defined together with their mappings to the

target view. The mapping produced here contains both formal and informal parts.

(c) And last, the hypothetical views and their role in answering stakeholder needs are determined. The authors distinguish two main roles for hypothetical views: as *a guide* to the reconstruction process and as *a baseline* in interpreting the conformance of the as-build architecture with the desired architecture. Hypothetical views are usually embedded in the target views and their viewpoint are defined as part of the target viewpoint.

3. **Reconstruction Execution** is realized by an iterative extract-abstract-present approach. It is further subdivided into 3 sub-steps:

   (a) **Data-Gathring**, where results from a static analysis of the source code artifacts and dynamic analysis of the system execution are gathered and analyzed by the architect with the goal of producing the views that conform to the viewpoints defined in the previous step. This process can be done by manual inspection, using lexical analysis tools such as grep, parser based (where a parser builds a parse tree representation of source code language constructs), through fuzzy parsing (which discards tokens that are considered unimportant to the reconstruction), or through semantic analysis (which looks at things like naming conventions or type resolution).

   (b) **Knowledge Inference**, whose goal is to derive the target views out of the source views, produced in *Data-Gathring*. This can be done according to [vDHK+04] using a manual, semi-automatic or fully automatic approach. All approaches use the formal and non-formal mapping rules defined earlier.

   (c) **Information Interpretation** is the final sub-step. Target views are selected, inspected and interpreted to solve the problems listed in the memorandum. It is worth mentioning here that because of the high user involvement in this step visualization takes special importance.

CaCOphoNy, introduced in [Fav], can be viewed as a complement to Symphony in that it also focuses in identifying the stakeholders needs and formalizing them in a set of viewpoints. The authors designed the method by focusing on the idea that the definition of software architecture in a big company is imbedded in the company's know how and on the company's culture. For them there is not a single definition of software architecture but software architecture is rather a group consensus among stakeholders and materialized by a large set of tools and repositories used in the company to develop the software products. [Fav] introduces the term Appliware to refer to the set of software developed by the company and Metaware to refer to the set of software the company uses in the development life-cycle. Metaware and Apliware together are important elements in defining the companies' know-how in tangible terms.

[Fav] propose to use Metamodels, to represent viewpoints, used to cover the architecture views, and argue that Metamodels represent the neat conceptual part of Metaware. On the other part models are the neat conceptual part of Appliware. The big picture comes together in [Figure. 7.2] with the views described by the viewpoints addressing the stakeholder concerns.

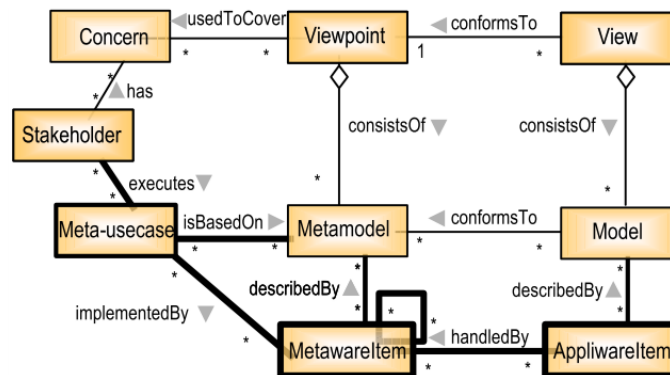

Figure 7.2: Merging Model Driven Engineering and Software Architecture into one model [Hol].

A summary of CacOphony process is given below:

- Metaware domain and asset analysis. Since tools and repositories codify the company's know how, their analysis is very important. A metaware inventory is build by the analyst and the metamodels that describe them are recovered by incrementally going through metaware-items. Metamodels are then integrated to produce a global metamodel which is at the end broken down into clusters to manage the complexity.

- **Metaware requirement analysis**. As with any analysis phase, actors (stakeholders) and meta-level use-cases are identified and the metamodels are improved through meetings aiming at identifying inconsistencies and missing elements.

- **Metaware specification**. In this step the identified use-cases are run in order to identify with precision the subset of the global metamodel in use by the company.

- **Metaware implementation, metaware evaluation and metaware evolution**. These steps are the "productive" steps in that their goal is to deliver the concrete metaware from the specifications gathered in the previous steps. These steps finally enable the analyst to describe what software architecture means in terms of views targeting specific stakeholders. It is worth noting that the metaware continuously evolves through the evaluation and feedback gathered from the stakeholders.

On a personal note, I consider this model to be too general. It is also only focused on big companies, which does not always apply since a considerable number of start-ups are of relatively small size and do not contain a significant number of metaware-items.

7.2.4 X-Ray Approach for Distributed Systems

Traditional programming languages do not provide explicit constructs for distributed high level design abstractions such as client, server, filters, pipes. A significant challenge when analyzing such systems is their distributed nature at runtime. The traditional approach for describing distributed systems has been from a multiple view perspective. **X-ray**, introduced by [Men], is an integrated, static analysis technique for distributed systems. X-ray encompasses 3 static analysis techniques:

1. **The component module classification** technique distinguishes source code models and maps them to processes.
2. **The syntactic pattern matching** technique discovers specific code fragments that implement component interaction features.
3. **Structural reachability analysis** maps code fragments, discovered by syntactic pattern matching, to executable units

These techniques exploit and enrich the mappings of artifacts in the 4 views: the logical view, the process view, the development view and the physical view described by [Kru95]. The module classification technique maps elements of the development view to those in the process view. The pattern matching and reachability analysis techniques enrich mappings of elements in the logical view to elements in the physical view.

7.3 Recovery Algorithms

7.3.1 Cluster-based

[CKS] introduced an extension to the Reflexion Model in order to help the user map concrete (source) entities into hypothetical entities. Their main assumption is that modules of existing software are build respecting the principles of low coupling and high cohesion. They use clustering techniques to group source entities together into clusters, to create additional candidate mappings given a number of already mapped source entities. Candidates of unmapped source entities are ranked and presented to the user. These candidates fall into

the same cluster with already mapped source entities maximizing cohesion and minimizing coupling with other source entities mapped into other clusters. A relationship between concrete entities is established when the one concrete entity calls/is-called-by (eg. a function calls another in C) or it accesses variables and fields (eg. calls or accesses to member functions/fields in Java) of the second entity. Similarities between identifiers, naming conventions are also used.

Before going into the details of the similarity functions [CKS] introduces to drive the clustering algorithm, lets first formalize the concepts introduced by the Reflexion Model in the previous section with the same notation that [CKS] uses.

The **Hypothesized View** represented as a graph by $HV = (N_H, E_H)$:
$N_H = \{h_1, h_2, \ldots, h_n\}, n \in N$: as the set of hypothesized entities,
$E_H \subseteq (N_H \times N_H)$ as the set of hypothesized dependencies
$partof : N_C \rightarrow N_C$ function representing hierarchies.

The **Concrete View** represented as a graph $CV = (N_C, E_C)$:
$N_C = \{c_1, c_2, \ldots, c_n\}, n \in N$ set of concrete entities,
$E_C \subseteq (N_C \times N_C)$ set of concrete entity dependencies.

The $maps - to : N_C \rightarrow N_H$ function maps concrete entities into hypothesized entities. This mapping is either user-defined (represented by $dmaps - to$) or inherited from the parent concrete entity. Otherwise the concrete entity is classified as unmapped.

$$maps{-}to(c) = \begin{cases} dmaps - to(c) & \text{if } c \in dom : dmaps - to \\ maps - to(c') & \text{if } c \notin dom : dmaps - to \wedge part - of(c) = c' \\ undefined & \text{otherwise} \end{cases}$$

Additionally an edge in either E_C or E_H can be assigned a type and one edge type may be a subtype of another edge type, forming an is-a type hierarchy. The function $type$ takes a concrete entity and returns its type while the hierarchy is given by the function $is - a$ which returns true if the type is a subtype of another type or false otherwise. The motivation resides in representing class hierarchies. In order to be consistent with the graph representation, the output of the reflexion model is represented by [CKS] as another graph summarizing convergences, divergences and absences between hypothesized entities $\mathfrak{R} = (N_H, E_{\mathfrak{R}})$:

$convergence(h_i, h_j) \in E_{\mathfrak{R}}$[1] $:\Leftrightarrow$
$\exists [c_k, c_r] \in E_C, [h_i, h_j] \in E_H : maps - to(c_k) = h_i \wedge maps - to(c_r) = h_j \wedge type([c_k, c_r])\, is - a\, type([h_i, h_j])$

[1]When an expected relation between 2 hypothesized entities can also be found between the concrete entities that map into them [CKS][MNS95].

$absence(h_i, h_j) \in E_{\mathfrak{R}}$[2] $:\Leftrightarrow$
$\nexists[c_k, c_r] \in E_C, \exists[h_i, h_j] \in E_H : maps-to(c_k) = h_i \wedge maps-to(c_r) = h_j \wedge type([c_k, c_r])\ is-a\ type([h_i, h_j])$

$divergence(h_i, h_j) \in E_{\mathfrak{R}}$[3] $:\Leftrightarrow$
$\exists[c_k, c_r] \in E_C, \nexists[h_i, h_j] \in E_H : maps-to(c_k) = h_i \wedge maps-to(c_r) = h_j \wedge type([c_k, c_r])\ is-a\ type([h_i, h_j])$

Now the **Algorithm** is straight-forward. The Goal is to map automatically concrete entities for which the decision is unambiguous and to help the user with a more restricted set of entities to choose from otherwise.

1. Filter unmapped concrete entities. The algorithm relies on source relationship between unmapped and mapped concrete entities to find missing mappings. Therefore it is problematic when an unmapped entity contains many source relationship with other unmapped entities as opposed to mapped entities. A filtering function δ is applied and concrete entities that do not pass the threshold w are not considered. $\delta(S \subset N_C) = \{c_i \in S | \frac{|\{(c_j,c_i) \in E_c | c_j \in dom(maps-to) \wedge c_i \neq cj\}|}{|\{(c_j,c_i) \in E_c | c_i \neq cj\}|} \geq w$

2. Calculate attraction matrix for concrete unmapped to hypothesized entities. [Fav] suggests 2 functions to calculate the attraction $CountAttract$ and $MQAttract$ with the first giving better experimental results. Moreover the values returned by $CountAttract$ are more spread out than the ones returned by $MQAttract$ thus easier to read. For this reason I am only going to present the calculations for $CountAttract$.

$$CountAttract(c_i, h_k) = overrall(c_i) - toOthers(c_i, h_k)$$

$$overrall(c_i) = \sum_{\forall (c_i,c_j) \in E_C : c_j \in dom(maps-to)} \lambda((c_i, c_j))$$

$$toOthers(c_i, h_k) = \sum_{\forall (c_i,c_j) \in E_C : c_j \in dom(maps-to) \wedge maps-to(c_j) \neq h_k} \gamma((c_i, c_j))$$

$$\gamma((c_i, c_j)) = \begin{cases} \lambda(type((c_i, c_j))) \times \phi & \text{if } \exists maps-to(c_i), maps-to(c_j) \in E_H \\ \lambda(type((c_i, c_j))) & \text{otherwise} \end{cases}$$

λ function applies a weight on source dependencies to reflect their semantic importance. In case of unwanted coupling the function γ returns

[2] When an expected relation between 2 hypothesized entities can not be found between the concrete entities that map into them [CKS][MNS95].

[3] When an unexpected relation shows up between 2 hypothesized entities and is backed up by an existing relation between the concrete entities that map into them [CKS][MNS95].

its weight unchanged while for allowed coupling it multiplies the weight by a factor $\phi \in [0,1]$. $\phi = 1$ has the effect of ignoring conceptual information. $CountAttract$ subtracts from the overall attraction value the attraction value of the concrete entity to all the other hypothetical entities except the one under consideration.

3. Find potential mappings based on the attraction matrix.

4. Automatically map concrete entities to a hypothetical entity for which only a single candidate could be detected.

5. Otherwise present to the user, ranked by their attraction, all candidates for a manual selection.

The main deficiency of this algorithms is that it assumes that the developers follow the principle of high cohesion and low coupling.

7.3.2 Pattern-based

The Pattern-based approach relies on first extracting a set of architectural patterns, expressing them in some format, saving them into a repository and then querying the source model for those patterns, recovering thus the system architecture piece by piece. The problem with this approach has always been in expressing architectural constructs into a language that is flexible enough and that allows defining queries to express architecture patterns. For this purpose Architecture Description Language (ADL) and Architecture Query Language (AQL) [Sar06] have been developed to express the architecture and architecture patterns respectively for a system. They have not found their way into the industry though mainly because of their lack of integration and synchronization capabilities with the source. The synchronization problem has been addressed by variants of ADL such as xADL and Fractal ADL but both of them provide only 1 way mapping [BW09].

The approach presented in [Sar06] consists of an offline and an online phase. During the offline phase the source code is parsed and represented as an attribute relation graph. In the online phase, first architecture patterns (eg. pipes, filters, client/server) are retrieved through expert analysis, documentation analysis and tool provided analysis and then they are expressed into AQL. The second step of the online phase is the graph pattern matching where the patterns defined in AQL are tested against the attribute graph generated in the offline phase. In the last step the user investigates the results and if it is satisfactory stops the recovery process. Otherwise he/she augments the pattern repository and runs the graph pattern matching step ones more.

[PFGJ02] comes with an innovative approach in Revealer. Revealer combines lexical and syntatic analysis and overcomes the problems associated

with parsing. Parsing is a heavy-weight process, needs all the information to be ready available (included files) and considers everything not just the interesting parts (where the elements of a design patter are being declared or used). On the other hand a pure lexical approach with regular expressions has problems considering white spaces. Revealer comes with a language to define patterns and combine them into pattern graphs expressing architectural hotspots. The language is furthermore written in XML, which makes it easy to read. At the base we have primitive patterns: RegExp (to handle regular expressions), StringExp (to handle simple strings) and Block (to handle blocks enclosed by start and end delimiters). To match continuous text blocks, patterns are linked into pattern sequences with only white-spaces allowed between matches. This is realized by the *rel* tag which contains the start and end attribute containing the pattern-ids and the type attribute which can be *next* for combining sequences of patterns or *contain* for unordered containment test. Other constructs such as ORPattern or ANDPattern can be used to combine patterns into more complex ones. Revealer also provides a mechanism for nesting patterns (a rel tag of type=*constraint*) by having the outer pattern match only if all the inner patterns match. With these mechanisms the user can easily define patterns that describe islands of interest in the code and associate them with a design pattern without focusing to much on the details in the code.

A drawback of this approach, and of all lexical approaches in general, is that the lack of control and data flow information does not allow to follow certain paths through the source code to find all relationships of code pieces that are concerned with a particular architectural aspect [PFGJ02]. The main advantage of Revealer is that it allows the analyst to define design patterns easily and quickly for any text based language and filter the source code with focus on only architectural hot-spots interesting to him/her.

7.4 Tools Overview

7.4.1 Classification

There are many aspects to Software Architecture Reconstruction, therefore no single classification exists. Tools have been classified by [Pol09] as being quasi-manual, semi-automatic or quasi-automatic depending on their role and degree of automation in the reconstruction activity.

In the **Quasi-manual** technique the tool assists the analyst in understanding his findings. It is further subdivided into construction-based and exploration-based techniques with the first assisting the analyst in manually constructing the views and the second guiding the analyst to the highest abstraction levels through input exploration, zoom-in an abstraction level, zoom-out capabilities

Listing 7.1: The following pattern is a class that uses a SocketServer [PFGJ02]

```
<RevealerPattern>
  <pe id="SS" type="Pattern"/>
  <pe id="new" type="StringExp">
    <attr name="expr" value="new SocketServer"/>
  </pe>
  <pe id="param" type="Block">
    <attr name="startDel" value="("/>
    <attr name="endDel" value=")"/>
  </pe>
  <pe id="CL" type="Pattern"/>
  <pe id="class" type="StringExp">
    <attr name="expr" value="class"/>
  </pe>
  <pe id="clId" type="Var"/>
  <pe id="block" type="Block">
    <attr name="startDel" value="{"/>
    <attr name="endDel" value="}"/>
  </pe>
  <rel from="new" to="param" type="next"/>
  <rel from="SS" to="new" type="contain"/>
  <rel from="class" to="clID" type="next"/>
  <rel from="clID" to="block" type="next"/>
  <rel from="CL" to="class" type="contain"/>
  <rel from="block" to="SS" type="constraint"/>
</RevealerPattern>
```

[Pol09].

For the **Semi-automatic** approach, according to [Pol09], the analyst instructs the tool on how to refine the abstraction that has been recovered. It is further subdivided into abstraction-based techniques and into the concrete approaches such as *Graph queries*, *Relational queries*, *Logic queries*, *Lexical and Structural queries*. The second subdivision consists of investigation-based techniques, with the main approach being *Recognizers*, *Graph-Pattern Matching*, *State Engine* and *Maps*. Abstraction-based techniques aim at mapping low-level concepts into high level ones such as styles and design patterns. On the other hand investigation-based techniques aim at mapping high level concepts into low level ones.

For the **Quasi-automatic** technique, according to [Pol09], the analyst only steers the iterative recovery and the tool does the rest. Pure automation does not exist. The two main approaches are clustering and concept dominion. Clustering techniques try to subdivide the system into loosely coupled clusters exhibiting high cohesion (see section 3.1). Concept dominion relates to graph theory and identifies parts of the source that belong together by singling out dominating nodes and their subordinates in the source dependency graph.

On the other hand [Pol09] classifies the whole approach as being top-down, bottom-up or hybrid. In the top-down approach the analysts starts with a high-level abstraction of the system derived from architectural styles, documentation, interviews, the company's organization and experts opinion. Iteratively the analyst formulates hypothesis of the system architecture and tries to match them with the implementation. In the bottom-up approach on the other hand the analysts starts with a model derived from the implementation details such as source code, configuration files or directory structure and then tries to abstract this constructs in a higher level of abstraction. [Pol09] refers to it as a **extract-abstract-present** approach. The hybrid is just a mixture of the top-down and the bottom-up approach. At the same time low-level knowledge is abstracted using various techniques and high-level knowledge is refined and confronted against the previously extracted views.

7.4.2 Tools

The Dali Toolkit is bundled together with other utilities such as Rigi, used for visualization, and a PostgreSQL providing for information storage and querying capabilities. After the source information is exacted, it is loaded into the PostgreSQL database. Queries, representing design patterns, defined in SQL and Perl are then run against the information stored in the database. Views are visually materialized and manipulated in Rigi [OSV02].

Lisa Toolkit, described in [BW09], is an Eclipse plug-in and provides for visualization, validation and integration. It comes together with an architecture

description language, Lisa-ADL, that supports 2 way integration of both the architecture and implementation. Traditional architecture description languages such as ADL, xADL, Fractal ADL and ArchJava provide at most 1 way synchronization. Another advantage of Lisa is that it is possible to detach architecture and implementation and fully simulate refactoring to close architecture violations such as layer violations and interface violations. Lisa-ADL is expressed in XML thus allowing for extensions and is able to express relationships between heterogeneous language components. The mapping of implementation and architectural concepts is supported through technology bindings (eg. java technologies), but is not limited to only Object-Oriented languages. Consistency checks are also defined and observed into the Model Validation perspective. In Figure. 7.3 shows the violation detection functionality of Lisa at work.

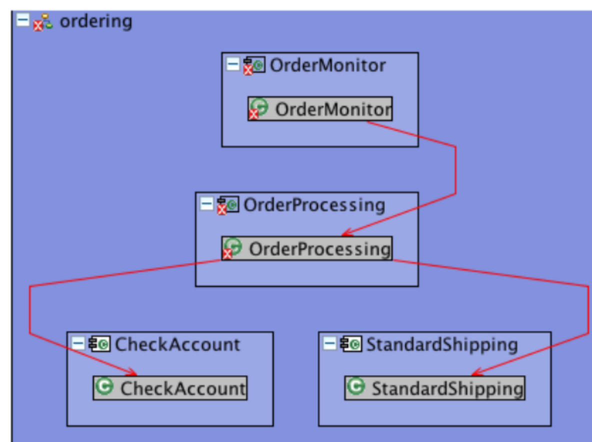

Figure 7.3: Violation: static dependencies instead of access through a defined interface [BW09].

Softwarenaut, as described in [LL06], is another tool designed in the form of a framework and supports visualization and exploration of different abstraction levels of the system at hand. It also comes with multiple perspectives such as: *Exploration Perspective*, *Detailed Perspective* and *Map Perspective*. The Exploration Perspective presents a graph of the visible modules. The Detailed Perspective visualizes only the currently selected entity and the Map Perspective shows the position in the hierarchy of the entity in order to give the analyst some sense of context and orientation. Softwarenaut also provides primitive operations such as expand, filter and collapse. It features 3 analysis types: *Package Dependency* that uses conventions to group subpackages into packages, *Directory Include Relationships* that considers include relations aggregated at the directory level, and *Semantic Cluster Interaction* that uses Latent Semantic Indexing to detect concepts.

SotoArc and **SonarJ** supports various languages such as Java and C++. Source code information is extracted from a parser and inserted into a mysql

database, similarly to Dali toolkit. The main functionality of SotoArch is to explore architecture violations and also simulate refactoring. It visualizes these violations in an intuitive manner using red for violation dependencies and green for allowed dependencies [Figure. 7.4]. It also comes with algorithms for detecting and breaking cyclic dependencies. SonarJ uses the same database and provides for searching for design patterns and is also capable of displaying extensive trend analysis metrics. [hel12b] [hel12a] [BW09] [Mer10b].

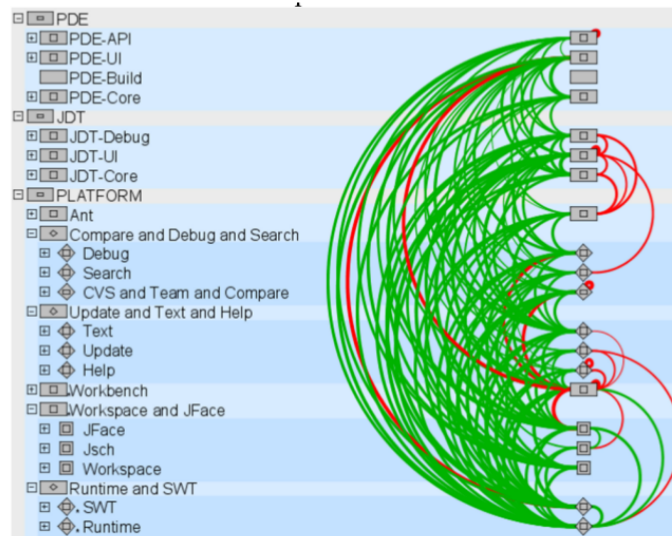

Figure 7.4: Visualizing of access violations in SotoArch [Mer10b].

Structure Analysis for Java (Stan4J) is another eclipse plug-in that supports structural analysis and include code exploration, dependency analysis and visualization of the system components. It uses build library includes and code dependencies to generate and aggregate dependencies between java packages. It includes operations such as filter through drag and drop, expand and collapse. It also shows dependency violations using red arrows. [Sof12] notes that the tool has been design to be used by developers to measure quality and report design flow during and after implementation. It also uses some of the novel visualization techniques like color cushions which are described in the next section.

Bauhaus from Axivion, described in [Mer10a], supports multiple languages like C/C++, Java but also COBOL and VB. It stores its information about the system into an intermediate language and resource flow graph files. Influenced by the Reflexion Model, it features two perspectives, laid out side by side, one showing the is-architecture, extracted from the static analysis of the implementation, and the other the should architecture defined by the architect. Divergence edges are displayed in red in the is-architecture. It also supports metrics, trend analysis and user defined queries describing a design pattern to be found in the is-architecture. Different from other tools described so far, Bauhaus is also capable of detecting cloned code [Axi12] [Figure. 7.5].

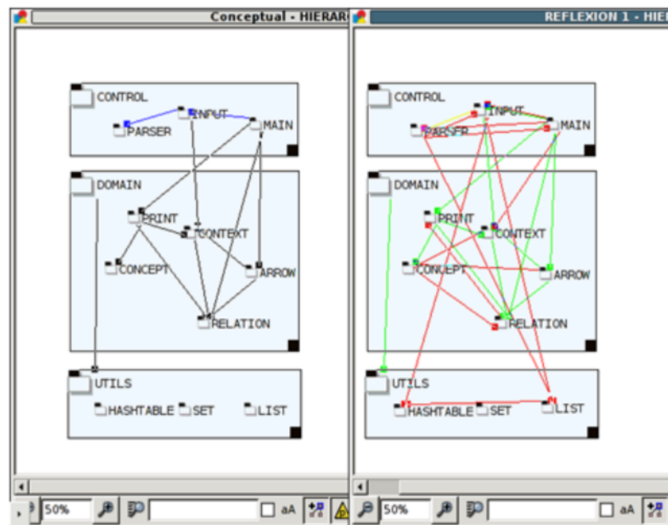

Figure 7.5: Visualizing of access violations in Bauhaus [Mer10b].

Structure 101 supports multiple languages such as Java, C/C++, .Net, PHP etc. The user can also define parsers for new languages and integrate them into Structure101. The architect defines a reference architecture diagram, capable of expressing containment, visibility and layering of the system, and then maps physical code to blocks in the diagram through regex patterns. Stricture101 additionally from displaying current violations [Figure.7.6], discovered after the mappings have been defined, also warns developers at compile time when code changes introduce new violations. Structure101 comes with a free plugin (Sonar) that tracks quality metrics and how architecture violations in the implementation evolve over time [str12].

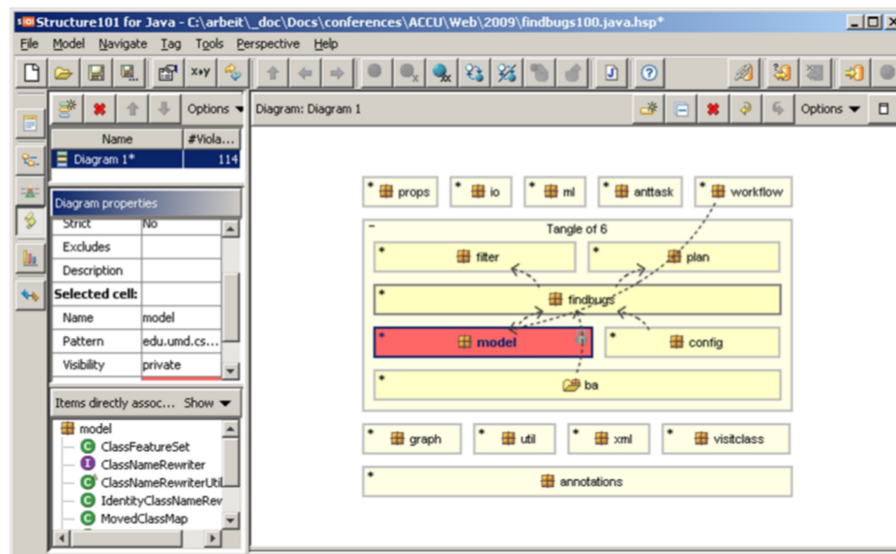

Figure 7.6: Visualizing of access violations in structure101 [Mer10b].

SAVE is the last tool I will explore. It has been developed by the "Fraunhofer Center for Experimental Software Engineering" under the vision of having an environment that connects architectural rules and constrains, and implementation of a system throughout the entire software life-cycle [LM08]. SAVE automatically extracts and visualizes the architecture module views. It has the capability of detecting strong dependencies and extensive coupling. It helps the developer understand the architecture and identify commonalities and deviations between different software product. The later functionality comes handy for companies interested in identifying the potential for product lines. SAVE's extraction tool is able to analyze the source code in C/C++, Java, Ada, Delphi, Fortran and Simulink. SAVE follows the process defined by the Reflexion Model. In the first two steps the planed and as-build architecture are defined, the first one manually and the second extracted from the tool. The third step corresponds to manually mapping the 2 views. In the forth, fifth and last step the deviations are identified, some are manually marked critical and a plan is build for removing them. Research is under way in adding forward engineering capabilities to SAVE by having it identify violations before they are committed. Also researchers are looking into adding requirement traceability to SAVE or integrating it with behavior modeling tools such as Simuling and Reactis which have shown to be quite useful in the embedded software industry.

7.5 Visualization Techniques

7.5.1 Visualization Overview

Typical visualization techniques for the tools I have surveyed so far consist primary of text and simple geometric shapes to highlight aspects of the software systems. Colors have found limited use for displaying violations and allowed dependencies. Violations are usually featured with red arrows in SotoArch [Figure. 7.4] and Bauhaus [Figure. 7.5]. A red background has been used in Structure101 [Figure. 7.6] to emphasis layer violations. In SotoArch [Figure. 7.4] layer violations in the should architecture are shown by counterclockwise red arrows while clockwise red arrows show interface violations, where components bypass the public interface and use instead a concrete implementation. Sometimes the graphical dependency, which gives only a quick overview, is supplemented by a separate numerical representation for highlighting measures of for example coupling between packages [Mer10b]. Because of the overwhelming volume of information, most of the time, these traditional visualization techniques are not able to keep visual cluttering at bay. Filtering or zoom-in and zoom-out features have been employed to ease the problem but on the other hand filter out some of the surrounding context the user needs to better interpret the view at hand. Visual cues like shading (lighting effects) and texture are not used, although the human visual system is very well capable of processing them [HVW05].

7.5.2 Innovative visualization techniques

[Hol] [HVW05] suggests and gives simple examples that visualization techniques such as treemaps, cushions, color, texture and bump mapping can help the tools display high volumes of information without introducing additional visual cluttering.

Treemaps and Cushions can be used to visualize hierarchies. [HVW05] uses rectangle to represent a node. The size of the rectangle is used to give the user a sense of the size of a component in terms of LOC. Internal rectangles are used to represent sub-components and subdivide the enclosing rectangle into parts proportional to the size of the sub-component. In order to make the division clear the direction is alternated between two adjacent levels. [HVW05] also suggests that Cushions can be used together with Treemaps to make the distinction between components clearer to the human eye and at the same time convey more information. The depth between valleys created between 2 adjacent rectangles is proportional to the distance between nodes in the tree.

Color and Texture can also be used to visualize scalar and ordinary types of arbitrary software metrics. [HVW05] uses color to visualize a single software metric at the leaf level and texture to visualize another type of metrics. Color ranges are defined by interpolating between a start and end color. A model calculates the texture look and feel by direct manipulation of a texture's spatial frequency, regularity and contrast characteristics. [HVW05] suggests that **Bump mapping** can be added to yield images with realistic looking surface wrinkles without the need to model each wrinkle as a separate surface element.

By putting everything together, [HVW05] shows an example of using the aforementioned techniques to visualize fan-in and code smell in [Figure. 7.7].

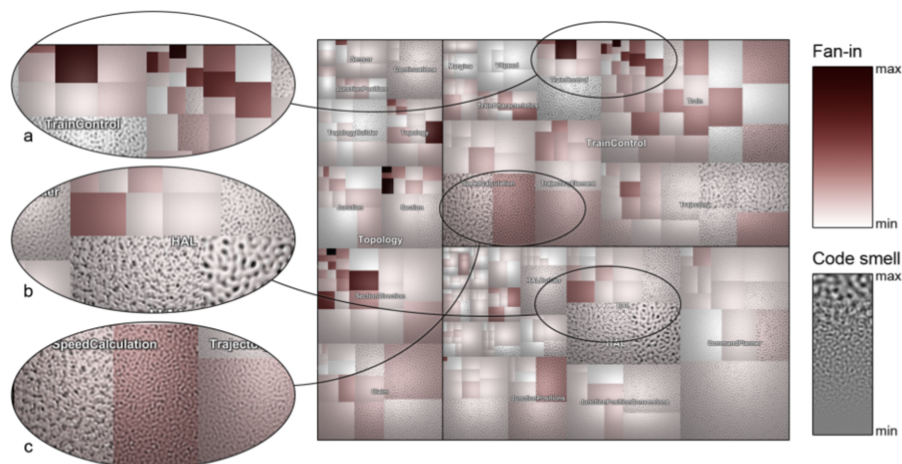

Figure 7.7: Visualizing software metrics using cushions, treemaps, texture, color [Hol].

In [Hol], the authors demonstrate another technique developed by them to display a hierarchical graph [Figure. 7.8]. Their approach consists of bundling adjacent edges and bending them towards the center as a B-spline curve. The hierarchical bundling reduces visual clutter. The relationship between 2 parent nodes is realized visually by the edges bundled together in the respective child nodes. In [Figure. 7.8] the hierarchy is shown using standard techniques.

Figure 7.8: Visualizing hierarchies [Hol].

7.6 Conclusion

In this paper, I demonstrated the importance of Architecture Recovery in answering the needs of different stakeholders. First I went through some of the approaches and frameworks that can be used by the analyst to reach the goals set for the Architecture Recovery. I first went through the Practice Scenarios of [OSV02], with each exploring a specific set of stakeholder concerns. When and if the analyst finds similarities between one of the practice scenarios and his task at hand, he can use the scenario to guide him in the reconstruction process. Then, I made a resume of the influential Reflexion Model. I also presented the generic approaches of Symphony and its complement CaCOphoNy. Both are quite general and can be adapted to a specific situation. The generality of these approaches is both an advantage and disadvantage. It is an advantage in that they can be adapted to a big range of possible architecture recovery goals, and a disadvantage in that their generality makes them difficult to adapt. The last approach, X-Ray provides for a guide in recovering architectural views of distributed systems. The techniques listed in X-Ray exploit and enrich the mappings of artifacts in the 4 views described by [Kru95].

In section 3, I presented the reader with an algorithm that adds automatic clustering to the Reflexion Model, with the intention of helping the analyst in the demanding task of mapping concrete entities to hypothetical entities. The algorithm defines a similarity function between unmapped concrete entities and hypothetical entities and uses it to limit the set of unmapped concrete entities presented to the user at each mapping decision. Then in Revealer, I presented an approach that combines syntatic analysis with lexical analysis. I also gave, among other things, a short example of the XML based pattern language definition introduced in Reveler and its advantages over parsing techniques.

In section 4, I made a summary of some of the tools that use static analysis. Most of these tools try to help the user identify violations in the architecture patterns exhibited by the software under consideration. Moreover tools like, for example, Lisa can simulate the removal of such violations while tools like SAVE can be used throughout the entire life-cycle of software development. Section 5, on the other hand, presented some traditional and some innovative techniques that can be used in displaying the recovered views. Since there does not exist any fully automatic recovery tool, visualization is really important since the user has to rely on an iterative extract-abstract-present approach. Techniques like: treemaps, cushions, colors and texture come with the promise of adding extra information to the different views presented to the user without introducing additional visual cluttering.

Bibliography

[Axi12] Axivion bauhaus. `http://www.axivion.com/products.html`, 2012.

[BW08] Georg Buchgeher and Rainer Weinreich. Integrated Software Architecture Management and Validation. *2008 The Third International Conference on Software Engineering Advances*, pages 427–436, October 2008.

[BW09] Georg Buchgeher and Rainer Weinreich. Connecting Architecture and Implementation. pages 316–326, 2009.

[CKS] A. Christl, R. Koschke, and M.-A. Storey. *Equipping the Reflexion Method with Automated Clustering*, pages 89–98. IEEE.

[Fav] J.-M. Favre. *CaCOphoNy: metamodel-driven software architecture reconstruction*, pages 204–213. IEEE Comput. Soc.

[hel12a] hello2morrow. Sonargraph. `http://www.hello2morrow.com/products/sonargraph`, 2012.

[hel12b] hello2morrow. Sotoarc. `http://www.hello2morrow.com/products/sotoarc`, 2012.

[Hol] Danny Holten. Interactive Software Visualization within the RECONSTRUCTOR Project. pages 4–5.

[HVW05] D. Holten, R. Vliegen, and J.J. van Wijk. *Visual Realism for the Visualization of Software Metrics*, pages 1–6. IEEE, 2005.

[Kru95] P Kruntchen. Architectural Blueprints. The 4 + 1 View Model of Software Architecture. *IEEE Software*, 12(November):42–50, 1995.

[LL06] M. Lungu and M. Lanza. Softwarenaut: exploring hierarchical system decompositions. *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 2 pp.–354, 2006.

[LM08] Mikael Lindvall and Dirk Muthig. (SAVE) Bridging the software architecture GAP. *Computer*, (June):93–96, 2008.

[Men] Nabor C Mendonça. Architecture Recovery for Distributed Systems. *Automated Software Engineering*.

[Mer10a] Bernhard Merkle. Stop the Software Architecture Erosion : Building better software systems. pages 129–137, 2010.

[Mer10b] Bernhard Merkle. Stop the Software Architecture Erosion : Tutorial SplashCon 2010. pages 295–297, 2010.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[OSV02] L. O'Brien, C. Stoermer, and C. Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, DTIC Document, 2002.

[PFGJ02] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer : A Lexical Pattern Matcher for Architecture Recovery. *Main*, 2002.

[Pol09] Damien Pollet. Software Architecture Reconstruction : A Process-Oriented Taxonomy. *Architecture*, 35(4):573–591, 2009.

[Sar06] Kamran Sartipi. Software Architecture Recovery based on Pattern Matching. *Computational Complexity*, 2006.

[Sof12] Odysseus Software. Stan: Structure analysis for java. `http://stan4j.com/`, 2012.

[str12] Structure101. `http://structure101.com/products/`, 2012.

[vDHK+04] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: view-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 122 – 132, june 2004.

Chapter 8

Extraction of the Dynamic View of Software Architectures

Alexey Ledovskiy

Contents

8.1 Introduction 152
8.2 General Information 152
8.3 Dynamic View Extraction Techniques 153
8.3.1 Aspect-Oriented Programming 153
8.3.2 Dynamic Program Slicing 155
8.4 Dynamic View Extraction in Practice 156
8.4.1 DiscoTect 156
8.4.2 Alborz toolkit 157
8.5 Case Study 158
8.5.1 Description of Kieker Framework 158
8.5.2 Example of using Kieker Framework 160
8.6 Summary 163
Bibliography 164

Abstract: This work gives a general overview of existing methods, approaches and techniques for extraction of the dynamic view of software architectures. Tools that are widely used in the industry are presented here. Moreover, the use case of the Kieker framework is shown in this work. Results from the use case and conclusions given in the last part of the work.

8.1 Introduction

There are a lot of large software systems, sue to a fast growing industry. These systems are extremely hard to maintain and understand. Thus, we need specific tools that help to understand the workflows of these systems. This work is aimed to describe approaches and systems for extraction of the dynamic view of software architecture. These techniques are very helpful for getting different kinds of information about the system at runtime.

The first part is going to describe a general and background information. It will include motivation of using dynamic analysis and some general terms.

The second part gives a description and overview of different techniques for extracting the dynamic view of software architecture. This includes following techniques: Aspect-oriented programming and dynamic program slicing.

The third part includes an overview and some specific descriptions of implemented tools, frameworks and plugins, which are based on the techniques mentioned previously. These are the Kieker framework, DiscoTect and its visual representation in AcmeStudio(plugin for Eclipse) and the Alborz toolkit.

A case study is presented in part four. It contains two parts: the first part is a detailed description of one of the tools; the second one is an example of using this tool on a simple program.

Results and conclusions are presented in the summary part.

8.2 General Information

There are many software systems that are complex and take a lot of time to understand how they work. One of the approaches could be researching the source code, but it is time consuming and some times, in large systems, an impossible task. Another method could be to extract a static architecture view from the system. Extracting the static view is a better approach, because it has more human readable appearance, but this will only offer a static representation of the system[4]. Usually, in a static view there is not enough information about how does the system's components interact with each other.

However, it is necessary to have a clear and complete understanding of the software system to be able to do tasks such as maintain, extend, modify, reuse the system or track errors in the system. Many approaches exist to extract different kinds of information about systems. The ”4+1” view model of software architecture one of the complex approach to describe the architecture of the software-intensive systems. It is using multiple different views to get a complete description of the program. It includes logical view, development view, physical view, process view, scenario. The logical view is about the logic of the system and what services it should provide to a user. The process view shows a performance requirements. The software development modules represented in the development view. The physical view shows physical requirements of the system. To synchronize all components, the model uses the scenario, which provides use cases for interaction between components.

In order to get more clear view of the software system, it is good to know how different components interact with each other during runtime. This part of analyzing is more difficult than static, because user analyzes the system at runtime, and applies different techniques and use special tools.

8.3 Dynamic View Extraction Techniques

Extraction of dynamic view of software architecture could be performed by certain techniques. This part has an overview of some common techniques. This includes Aspect-oriented programming(AOP), Dynamic Program Slicing technique.

8.3.1 Aspect-Oriented Programming

Overview

Aspect-oriented programming(AOP) is a programming paradigm[2], which allows to modify the program, and inject pieces of functionality into existing code. It uses aspects for these purposes. An aspect contains pointcuts. Pointcut is a term that let the program know where to apply a cross-cutting concern. Cross-cutting concern (for Java it usually calls joinpoint) consists synchronization, logging, memory allocation and etc. Aspect-oriented approach can be used, for example, in metrics injecting into your code. The Advice is an extension to the definition of joinpoints, represented as an additional code to an existing source code. It could be logging for example, to log each method call.

Existing Frameworks

Frameworks that use an aspect-oriented approach became commonly used. Therefore, exists many tools and frameworks which use an AOP approach. Overview and basic description about few of them is presented in this part.

AspectJ There is one popular framework: AspectJ[3]. This is an aspect-oriented extension for Java. It is in an Eclipse foundation open-source projects. It became hugely popular because of it simplicity. It also has IDE integrations for displaying cross-cutting structure. One of the aspects of AspectJ needs to be considered, that it injects functionality during compiling time. AspectJ has two mechanisms to implement joinpoints: 1) dynamic crosscutting; 2) static crosscutting. AspectJ consists of joinpoints, pointcuts, advices and aspects.

AspectWerkz Another framework for extraction information from a running software system made for Java is an aspect-oriented programming framework AspectWerkz[9]. It has a lot of dynamic AOP operations: 1) add or remove an advice at a specific pointcut; 2) reorder advices at specific pointcut; 3) swap the implementation of a mixin. For dynamic constructs, AspectWerkz uses approach based on a static weaver. Static weaver is a component of AOP for integrating aspects in to program at specified location of the program. Using bytecode modification AspectWerkz allows to modify your classes at different moments: runtime, build-time, class load time. AspectWerkz widely used now because it is fairly lightweight and dynamic framework.

Conclusion

Description of these instruments shows us that there are not complicated methods and tools to extract information about software systems at runtime. All this approaches were presented for Java, although there are many tools with the same approach for different languages, for example a framework AspectC++ is made for C++. These tools successfully using aspect-oriented programming technique, that helps to find performance problems.

AOP approach has some advantages for analyzing systems:

1. This instrumentation can be applied during deployment process without any source code modification

2. Inserting probes into specific method. It helps to do not overload the analysis of the software.

3. There are many open source frameworks that are use AOP. That makes development and profiling of software systems cheaper and easier.

8.3.2 Dynamic Program Slicing

In order to get an information of a software system there is another techniques called dynamic program slicing. This part contains description of that approach. There are different parts that need to be reviewed to get an idea about the dynamic program slicing. Overview is about a general idea and more specific parts of the technique[7]. On the next section is description of the tools and frameworks. Conclusion summarizes the topic about dynamic program slicing.

Overview

Dynamic program slicing is aimed to find all statements in a program that somehow affect the value of a variable occurrence. The definition of term program slice — is a set of statements that might change value of a variable, and it is totally independent of the input values to the program.There are several approaches for dynamic slicing. First two approaches extract a static view of the program. Third uses Dynamic Dependence Graph for dynamic view.

Existing Frameworks

Few tools and frameworks exist which use dynamic program slicing. In following part, two tools are going to be described. It will give brief overview of existing tools in industry.

JSlice First tool calls JSlice[10]. It is a tool that makes debugging and errors finding of the program easier. JSlice is a high performance tool, because compresses execution trace and then analyzes it. That approach helps to JSlice make faster automatic search of an error. It supports execution trace compression at runtime. That is one innovation that makes analyzes of large systems faster. Another special thing about JSlice is that it can analyze a non-human readable code (machine code). That particular ability allows JSlice to collect information about external libraries. Therefore, all these features of JSlice became widely applied in industry today.

The Wisconsin Program - Slicing Tool The Wisconsin Program is a second tool[8]. The Wisconsin Program is aimed to support programs written on C lan-

guage. It also uses different types of slicing programs: forward and backward slicing, and also chopping. However, the Wisconsin Program has different approach that JSlice. Nevertheless, it is also very efficient. It creates a program dependences and a control-flow graph. One of the main package of the Wisconsin Program made for ”manipulating program dependence graph”[8]. This package extends this graph to a system dependence graph. Slicing tool gives to users many capability, including different types of program slicing. Chopping of a program aimed to part of program or specific target, and showing component dependencies inside a program. The Wisconsin Program - Slicing Tool was tested and made for SunOS also known as Solaris 2.5.1.

Conclusion

Many tools use different techniques of dynamic program slicing. Moreover, we can say that it is widely used in industry now. Because, some of them build for specific systems. As we can see, slicing tools developed not only for regular-users systems that are most popular (such as Windows, Mac OS, Linux), but as we see for the Wisconsin Program Slicing Tool is made for Solaris system. Dynamic program slicing, in general is a good approach for extracting runtime information. It also can show a dynamic dependence graph, which is good because user can see how different components of the program interact with each other.

8.4 Dynamic View Extraction in Practice

There are many tools, frameworks for monitoring programs and extracting different information at runtime. Here, I am going to present few of them: Kieker framework, DiscoTect and its visual representation in AcmeStudio(plugin for Eclipse), Alborz toolkit.

8.4.1 DiscoTect

Overview

DiscoTect is a toolkit for deriving software architecture from the system at runtime, and also for monitoring and analyzing the system during execution. The system uses probes that are inserted in methods and functions calls, and tracks value changing, creation of an object, etc. Then DiscoTect analysis runtime events and map it to architectural events. Architectural events then interpreted by architectural tool and presents the runtime architecture.

Technical aspects

One of the technical aspects is a DiscoSTEP Programming approach. It contains three components: 1) Events; 2) Rules; 3) Compositions. Events are basically the XML scheme that defines how to receive events from the system. Next, Rules contain information how to get an architectural view from events. It has four specific terms: Input events, Output events, Trigger (checks input, if we should apply the rule or not), Actions (actions for producing output). Composition allows to create complicated sets of rules.

The DiscoTect Formal model is fairly not complicated. The best representation of this mode is the Colored Petri Nets (CPN). The DiscoTect Formal model has the same approach. Each rule has an equivalent of CPN transition. Each event is a place in CPN representation. The concept is that we have an input value for certain rule, then it check if it suited by the color of the event, then it maps place into transition and complete petri net, by moving token from the transition to the final place.

Conclusion

DiscoTect toolkit now widely used by many software developers. It has been used for deriving architecture in software systems, such as mobile systems, Enterprise JavaBeans (EJB) systems. The DiscoTect tool in many use-cases shows good results. The DiscoTect technical aspects allow to extract an architectural dynamic program view significantly good. As The DiscoTect developers says[6] that results of using program are different to origin. Some of the dynamically extracted architectures has differences comparing to the original architecture. The DiscoTect tool is easy to use, but I personally had problems with finding new builds of the tool. All versions that I found was for old versions of a software (Eclipse).

8.4.2 Alborz toolkit

Overview

Alborz toolkit[5] analyzes a program from two perspectives. Therefore, it can provide an architectural information form both static and dynamic views. It is developed as an Eclipse plugin. Frequent pattern of execution traces techniques in software system allows Alborz dynamically analyze the program.

Technical aspects

The Alborz toolkit consist of following components: User interface component, Datastore component, System analysis component, I/O component. The User interface component interacts with user, and helps to a user to create a specific request. It has wizard options for analyzes, which makes it easy to use. Datastore component is used for storing all analyzed information. Datastore also helps the system components communicate with each other. System analysis component is a main component of the Alborz tool. It contains all mechanism and functions for online analyzes. That also includes algorithms for processing the data, for example search algorithms. I/O component is also very important component and holds a lot of functionality. That component received the data from different sources, create an output data using evaluation component. Creates a wrapped source code for visualization tools.

Conclusion

Summarize all aspects of the Alborz toolkit there is some lack of the system at this moment. One of them is that it analyzes the program by extracting the execution trace of a certain order of tasks. Therefore, the user has to know the domain of the software system. Also, with reference to developers of the Alborz toolkit[5], the tool has memory size constraints, so it cannot process very large systems. In summary, I think that system need to be improved before it can be used on full strength in software developing companies.

8.5 Case Study

This part shows how extraction of dynamic view of a software architecture could be done in practice. Therefore, I am going to present a use-case of the Kieker framework.

8.5.1 Description of Kieker Framework

Overview

One of the frameworks for dynamic view extraction is the Kieker framework[7]. Kieker framework is the framework that monitors the program at runtime. Moreover, it has measurement probes for analyzing performance of the software. It

is a useful tool for analyzing runtime behavior and representing visually architecture model with quantitative observation.

Technical aspects

Here are presented Kieker framework components. Which contains: monitoring probes, monitoring records, monitoring writers, monitoring readers, monitoring log and stream, analysis and visualization plugins. These components of the framework allows to analyze software systems. Components overview is presented on Figure 8.1. Monitoring writers and corresponding readers components are just logging and monitoring writers and readers for different parts of the system such as a file system, a database, a virtual memory and etc. Monitoring records component allows to save monitoring data of operations that collected during the execution. Monitoring probes include methods and mechanisms that allow to monitor method calls and also timing information for a particular method. From the analysis and visualization plugins Kieker can visualize the architecture of the monitored system by representing it as call tree, dependency graph or sequins diagram.

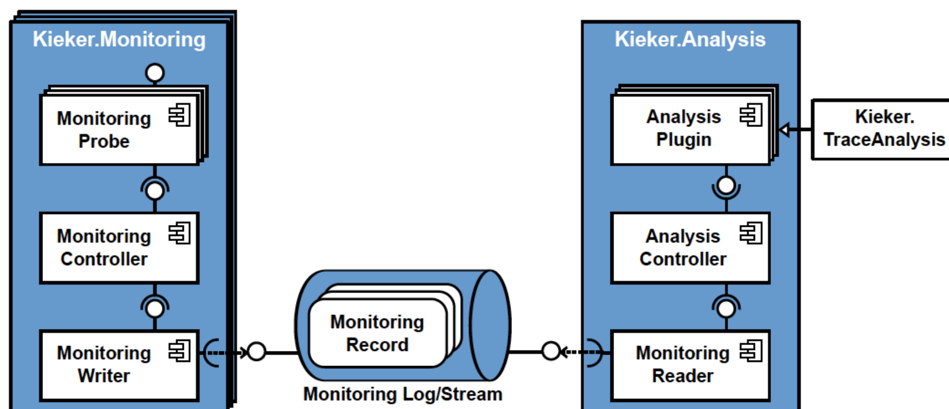

Figure 8.1: Shows components of the Kieker framework[7]

Conclusion

In summary, all components are properly divided by certain functionality. That really helps to understand the framework itself. Therefore, it leads us to better usage of this tool. In general, Kieker framework is a growing framework, according to the official web page[1], besides Java language, currently under developing the versions for Visual Basic 6, .NET, COBOL programming languages.

8.5.2 Example of using Kieker Framework

In this part I will try to make a simple practical example that will help to understand workflow of analyzing software systems.

Overview of an example program

The test application is represented as a Bookstore[7] that has a Catalog of books in the store. It also has Customer Relationship Manager(CRM). Thru Customer Relationship Manager Bookstore can search books. Catalog is meant to be a list of books that the bookstore has. Customer can get an item thru the bookstore. For clear understanding I present a class diagram (Figure 8.2) of this program.

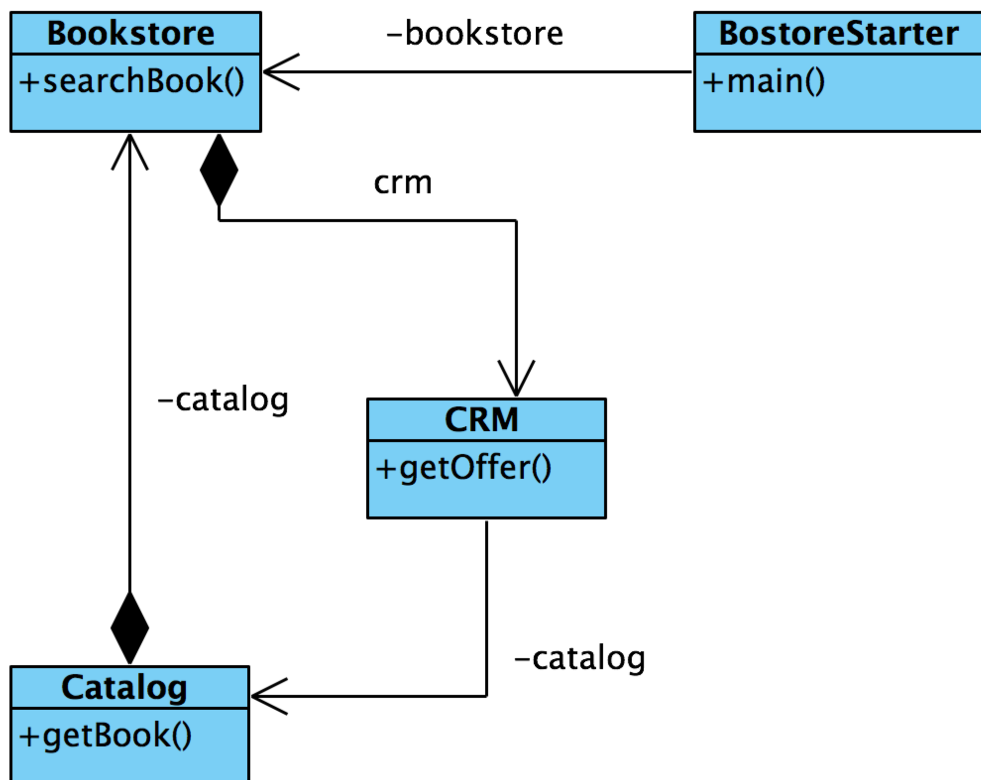

Figure 8.2: Class diagram of test application

Important parts of the source code

In this example I will try to extract a sequence diagram of the application, an operation dependency graph and a call tree. In a source code there is not much to describe. The code is pretty much straight forward. Here are a couple important things and program work-flow. The program starts with BookstoreStatrer class. It starts sending requests to Bookstore by triggering the searchBook() method. Then the Bookstore calls the getBook() method from the Catalog class and sends a request to the CRM by calling the getOffers() function. The CRM.getOffers also calls the getBook method from the Catalog. In the Catalog class the getBook function checks an input value for true or false, and just pause the program.

Integrating Kieker framework into the code

For Integrating Kieker framework into code Kieker framework has probes of AspectJ. For declaration these probes in code we have to use Kieker annotation: @OperationExecutionMonitoringProbe. The following part shows, where in the program we add probes to monitor methods.

In class Catalog we add one probe:

```
public class Catalog
{

      @OperationExecutionMonitoringProbe
      public void getBook(final boolean complexQuery)
      {
      ...
      }

}
```

```
The same approach we use in CRM class:

... @OperationExecutionMonitoringProbe

public void getOffers() { ... }
```

Bookstore class:

```
... @OperationExecutionMonitoringProbe

public void searchBook() { ... }
```

For using these probes we need to use kieker — aspectj library. That library comes along Kieker framework.

Execution application

In this part I will go step by step to build an application and show the main flow of the program execution and extraction data from the program:

1. First of all, we need to build the project. For that class path needs to be specified as following:

   ```
   -classpath lib/kieker-1.6_aspectj.jar
   ```

2. Now when we have builded classes, we execute the program and extract data from the program executions. In order to do that we put aop.xml and property files to META-INF folder. And then run:

   ```
   java -javaagent: lib/kieker-1.6_aspectj.jar

   -classpath build/kieker.examples.userguide.ch5bookstore.BookstoreStarter
   ```

3. As a result after the previous step we have two files kieker — ... — .dat and kieker.map. They contain extracted data about the execution trace of the program.

4. Then we need to analyze this data. Keiker.TraceAnalysis made for extracting useful data from these files. Procedure is following: on these files we need to apply shell script named trace-analysis.sh.

5. After fourth step we have gotten the system-entities.html file which contains information about system components. Also deploymentSequence-Diagram — {threadID}.pic, deploymentOperationDependencyGraph.dot and callTree — {threadID}.dot files, which contain a sequence diagram, an operation dependency graph and a call tree respectively.

6. Finally, after getting these files, we can visualize these properties of the system by converting them in to image files.The results are going to be presented in the following part.

Results

Diagram of the system's workflow shown on following figures. On Figure 8.3 shown the call tree. We can see that it is work as it should work accordingly to a description of the program. The operation dependencies graph presented on Figure 8.4. It shows how components of the program interact with each other. Finally the sequence diagram is on Figure 8.5. The sequence diagram shows the object interactions during a period of time.

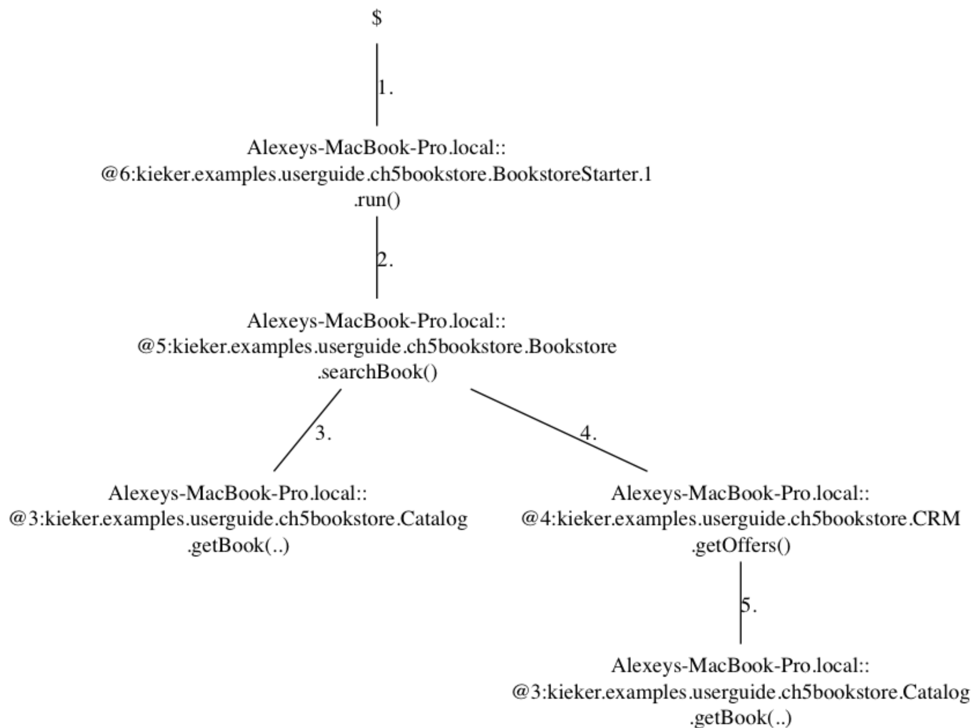

Figure 8.3: The call tree

Conclusion of use-case

Kiker Framework provides good instruments for extracting the information about a program at runtime. It is highly flexible and easy to apply. From the one raw source file we can extract a lot of information about the program at runtime. One of negative sides of this framework I would name, that data from the analysis is hard to visualize. For that you need to use other tools or scripts which does not have any opportunity to customize an output.

8.6 Summary

In summary, there are many approaches for extracting dynamic view of a software system at runtime. They are all have their pros and cons, and may be applied to specific problems. Therefore, one approach can be better in one area of the industry and another approach in a different direction. In my opinion, this is extremely important part of software developing process. Because, that helps to find errors faster, which makes development of software cheaper and more efficient. It helps to understand and modify old programs, which have been working and not modified for a long time because of its complexity.

Using dynamic analysis helps to understand work-flow of a system, and how components communicate with each other inside the system. Building the simple program in this work, I easily extracted a lot of information which is enough to understand a workflow of the program. In this work also was considered different frameworks and techniques that are commonly used in industry. Many of them are growing really fast and promise to be good soon. In general analysis of the programs need to be used, it makes development more efficient, productive, easier and cheaper, and all these aspects are hugely valuable now days in the software systems development industry.

Bibliography

[1] Kieker Framework. Web-page of project [online]. 2012. URL: `http://kieker-monitoring.net/`.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, and J. Irwin J. Loingtier. Aspect-oriented programming. *the European Conference on Object-Oriented Programming*, 1997.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[4] Welf Lowe, Andreas Ludwig, and Andreas Schwind. Understanding software - static and dynamic aspects. In *17th International Conference on Advanced Science and Technology*, 2001.

[5] Kamran Sartipi, Lingdong Ye, and Hossein Safyallah. Alborz: An interactive toolkit to extract static and dynamic views of a software system. *IEEE International Conference on Program Comprehension*, 2006.

[6] Bradley Schmerl, David Garlan, and Hong Yan. Dynamically discovering architectures with discotect. *SIGSOFT Softw. Eng. Notes*, 30(5):103–106, September 2005.

[7] Kiel University Software Engineering Group. Kieker 1.6 user guide, 2012. URL: `http://kieker-monitoring.net/documentation/`.

[8] The Wisconsin Program-Slicing Too. Web-page of project [online]. URL: `http://research.cs.wisc.edu/wpis/slicing_tool/`.

[9] Alexandre Vasseur. Dynamic AOP and runtime weaving for Java—How does AspectWerkz address it? In Robert Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors, *DAW: Dynamic Aspects Workshop*, pages 135–145, March 2004. URL: `http://aosd.net/2004/workshops/daw/Proc-2004-Dynamic-Aspects.pdf`.

[10] T. Wang and A. Roychoudhury. Jslice: A dynamic slicing tool for Java programs.

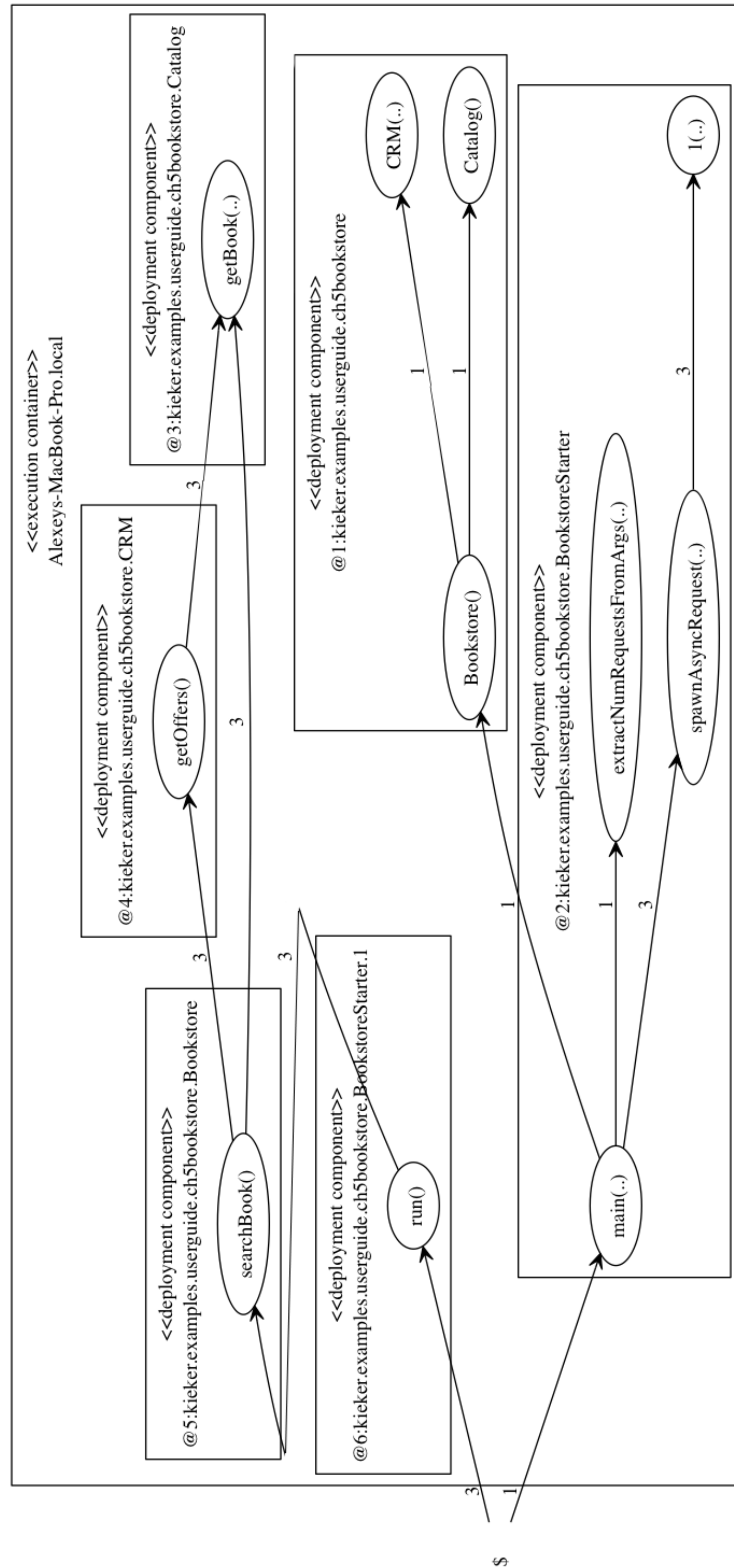

Figure 8.4: The operation dependencies graph

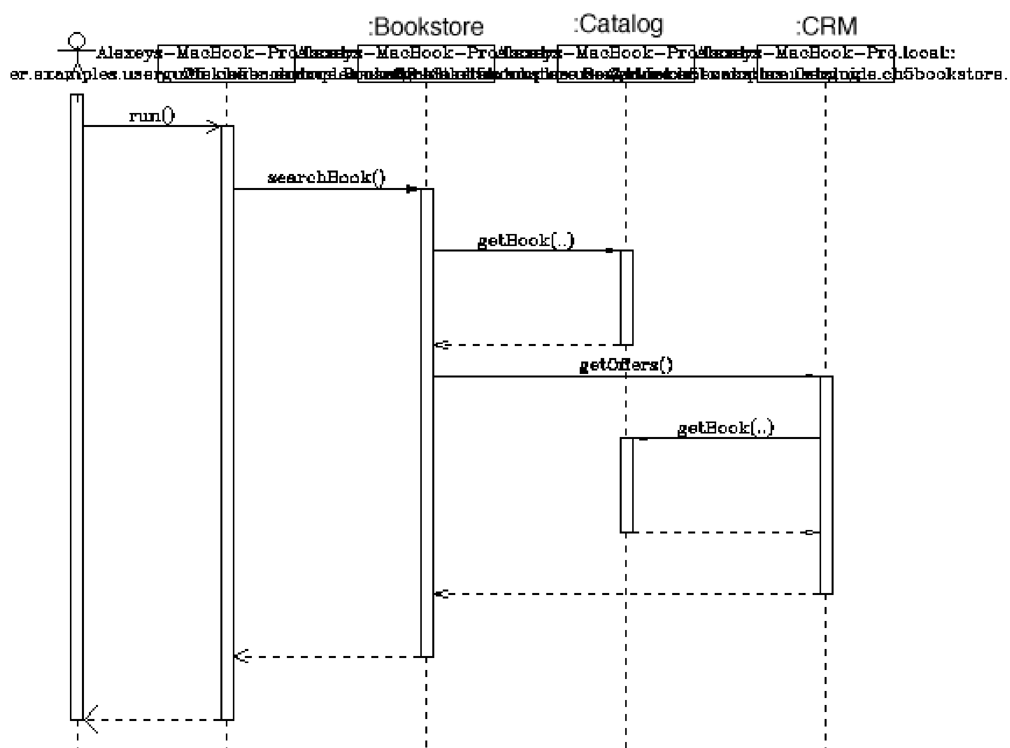

Figure 8.5: The sequence diagram

Chapter 9

Architecture evaluation

Hafiz Hasanov

Contents

Abstract: Software Architecture(SA) is one of the most important artifacts during software development process, because the rest of implementation is build upon it. Therefore, it is also crucial to ensure the correctness of SA, because it is time and resource consuming to apply possibly required changes in later stages of development life-cycle. The evaluation of the SA is required to achieve this. In this paper we will discuss costs, benefits and some major challenges of conducting SA evaluation. Also, we classify and compare several evaluation methods which are widely used in industry.

9.1 Introduction

Software architecture (SA) evaluation has become one of the most important phases of a software development life-cycle, because it reveals potential risks and possible architectural discrepancies early in the process, thereby avoiding costs and effort which would have been spent later on. However, software evaluation requires some resources and effort as well. Therefore, it is crucial to ensure that conducting a SA evaluation is more beneficial than it is costly. Sometimes, for this purpose, it is required to conduct an additional pre-evaluation cost-benefit analysis. Such an analysis can also be required to persuade the owner of a project, who is financially responsible for it, to pay these costs.

In the next section we will introduce some common terms to build our discussion upon. Then in section 1.3 we will continue on discussing about costs, benefits and some purposes of, and major challenges on SA evaluation. Afterwards, in section 1.4 we will classify and compare some of the well known evaluations methods. Finally, in section 1.5 we will sum everything up by a conclusion.

9.2 Background

In this section we will introduce some terms which will be used throughout the rest of this paper. Basically this section contains some introduction on SA, when it is designed, how it is commonly represented and what information this representation contains. Moreover, we will talk about architecture quality, quality attributes and SA metrics.

9.2.1 Software Architecture

Software architecture (SA) is the main output of the design phase of a software system. It is defined as follows: *"The software architecture of a program or computing system is the structure or structures of the system, which comprise*

software elements, the externally visible properties of those elements, and the relationships among them" in [BCK03]. In other words, it is a blueprint, that the system would be build upon and/or the foundation of the future software system. It contains all details on functionality, requirements, as well as design decisions that are crucial for the lifespan of the system it is designed for. Moreover, SA is an abstraction of the structure of a system, which is much easier to understand and explain. Therefore, SA is also a common ground for communication for each of the stakeholders like customer, developer, architect, project manager and so on [ISO11, Sha90].

SA representation

A software system can have stakeholders of different specialization background (e.g customer, user, developer, etc.). Each of these stakeholders might be interested in some characteristics of the system and don't event want to understand the specification of other aspects. Therefore, to represent a SA, different architectural views are needed, for people of various backgrounds.

In [Kru95], a representation of SA in 5 views is suggested, which is widely used in industry. Each of these views describe some part of SA, from a certain perspective as shown in Figure 9.1. Each view would then be described differently, by means of diagrams and notation, which are understandable to the audience the view is referring to.

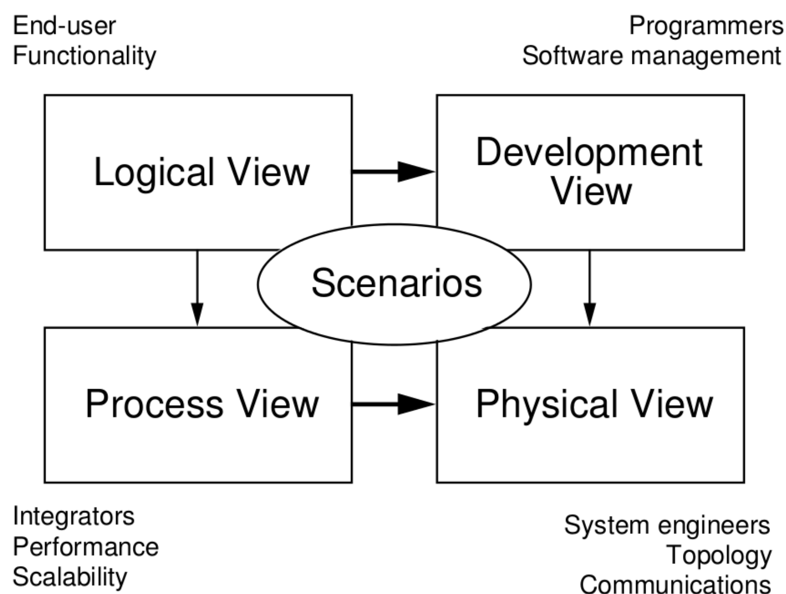

Figure 9.1: The "4+1" view model. [Kru95]

According to another source, SA is described by static, map, dynamic and

recourse views [LBK+97].

So, although there is no single idea of which views are more useful, they all basically serve one purpose: *"Separating different aspects into separate views help people manage complexity"* [DN02], divide and conquer principle.

Moreover, in [ISO11] a thorough diagram is given, which clearly depicts architecture description.

9.2.2 Quality of SA

The quality of a software system is measured by its relevance to project requirements. In other words, if a system meets all its requirements, it can be considered as a high quality system.

Functional and non-functional requirements. A required functionality of a system is categorized as a set of functional requirements of that system. These define, what and when the system should do certain things as well as how it should react to some actions and/or inputs. Non-functional requirements, on the other hand, define the level of quality, at which the system should perform the required functionality. Therefore, non-functional requirements are often referred as quality attributes.

Quality attributes

As we already mentioned above, quality attributes of a software system are it's non-functional requirements. A quality attribute is defined as *"A characteristic of software, or a generic term applying to quality factors, quality sub-factors, or metric values"* in [IEE98].

Depending on the purposes of the SA evaluation, various combinations of quality attributes can be defined. Although the standard [IEE98] does not mandate any specific attributes to choose and leaves this decision to the organization that does the evaluation, it states that the required attributes should be precisely defined. As a result, the chosen combination of quality attributes basically define the SA quality.

Although the paper's topic is related only to architectural level, it is very important to state that, the quality attributes are not dependent only on this level, but also have to be monitored during the implementation and deployment phases. There are some commonly referred quality attributes that we will be discussing in more detail:

- **Performance:** We concentrate on user-centric software systems and

not on automated systems or others. Performance of a software system comprises three sub-elements: the execution time of a single user operation, the resource access and the amount of memory used. Execution time is the sum of time of an operation to go through the whole stack of software layers(e.g client-side, back-end, business logic, data sources and etc.). Resource access refers to the efficiency by which the system accesses the resources. The latter sub-factor addresses the efficiency of memory usage. Sometimes, only one or two of these sub-factors are considered to be enough to represent the performance, but this doesn't have to always be the case. So, performance partially depends on communication among different components, which is defined in architecture, but it also depends on the algorithms that are implemented for the required functionality.

- **Modifiability:** For a system to be modifiable, it should be readable and easy to understand. These are determined by implemented coding styles and how the required functionality is divided among components respectively [BCK03]. The first one is handled implementation phase and the latter needs to be designed in architectural level. In our case, we are interested only in architectural level procedures.

- **Usability:** How easy the system is for the user to use defined by its usability level. Whether user has the functionality to be able to provoke some action once she clicks cancel, import or save button solely depends on whether this functionality was included in architectural design. However, the alignment and sizes of these buttons in user interface also effect the usability of the system.

- **Reliability:** It is defined by probability that the system is working and is available for use. It can be broken down into sub-attributes like fault tolerance and recoverability, which means that the system is tolerant to failures and can recover in those cases.

- **Flexibility:** *"The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed"* [IEE90]. We can see from the definition, that flexibility is closely related to modifiability and also maintainability, which we discuss next. So, this attribute defines in what level the software can be modified(if needed) and how much effort it will require for this.

- **Maintainability:** This quality attribute is also very tightly related with flexibility and modifiability. The standard [IEE90] defines it as *"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"*.

- **Portability:** It defines only one sub-element of maintainability or modifiability, namely the ease of software to adapt to some changes in the environment(e.g platform change).

- **Security:** In general, it is the ability of a system to deny system usage and services to unauthorized users, while providing its services to authorized users without negatively affecting the quality of service. However, in more specific systems(e.g banking, military, etc.) the purpose of this attribute may differ [BKLW95]. For example, while in a banking system, the unauthorized modification of user information is the central risk, in military or government related systems the secrecy of information is the main concern.

An important message here on quality attributes is that *"Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architectural level"* [BCK03].

Another important aspect is the relationship between quality attributes. As some of the attributes' improvement can lead to other attributes' deterioration. For example, the increasing number of servers in a web-application system would positively affect the performance. However, the security would suffer, because there would be more attack points. Thus, sometimes there should be trade-off between different quality attributes while evaluating them to achieve some level of quality. In such cases a trade-off might be needed.

Other quality attributes

Apart from commonly referred attributes we introduced earlier, there are also several other attributes that also used quite often and need a special attention:

- **Sustainability:** According to [Koz11], a sustainable software system is long-living and can be cost-effectively modified according to changing requirements or environment. Sustainability is different from other relative quality attributes like evolvability, maintainability, etc., because they consider notions of longevity and/or cost-effectiveness only to a limited extent [BE02].
- **Adaptability:** It can be defined as sub-characteristic of portability as the ability of a software system to adapt its functionality to environment changes. However, in [Tar07], the run time changes(which is not the case with portability) and behavioral and structural changes in requirements are also considered.
- **Evolvability:** It defines how easy it would be to adapt the system to the changes in the requirements with the least possible costs.

As it is seen from all examples, quality attributes are in a tight relationship with each other, where some overlap while others negatively effect each other. Therefore, it is crucial to consider these factors while choosing the attributes for quality definition of a SA.

9.2.3 Architecture quality metrics

A software quality metric is basically a function, as defined in [IEE98], which takes in software data and outputs some numerical value interpreted as the level of quality for the desired quality attribute. A quality metrics framework shown in Figure 9.2, depicts that quality is defined as a collection of it's sub-factors, where each quality factor has its direct metric. Direct metric is just a quality metric that only depends on its direct quality factor and not on any other factor. For example, depending on aims of evaluation, performance quality

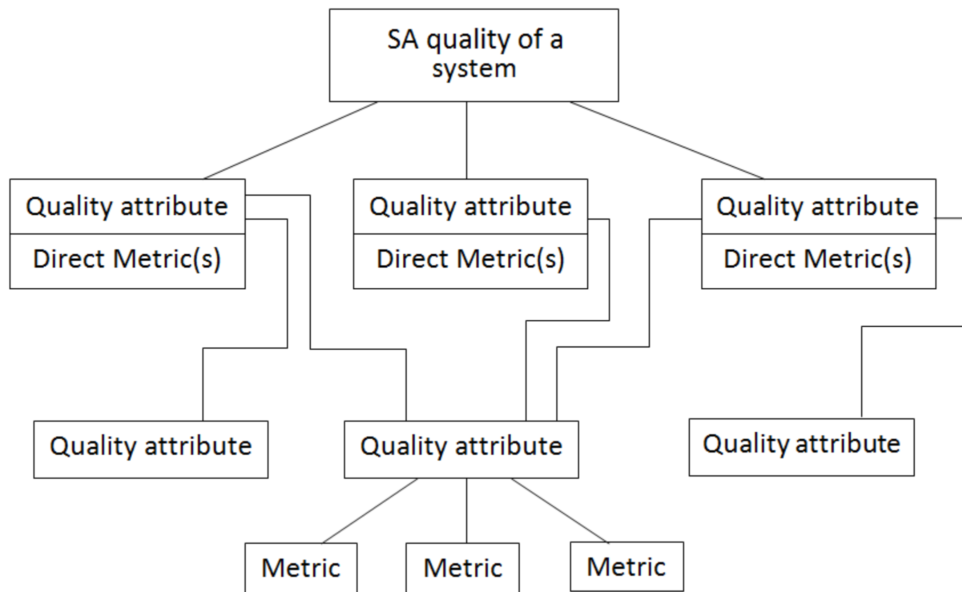

Figure 9.2: Quality metrics framework [IEE98].

attribute can have several quality metrics as throughput, latency or memory access. One may need only one of these metrics, or in other case all three of them, depending on the purpose. Coupling is anther example for SA metrics. It is the number of relations between different components. Coupling can be used to measure the maintainability or flexibility of a SA.

9.3 Benefits and costs of evaluation

As it is always important to conduct a benefits versus costs analysis before starting any SA evaluation activity, it is intuitive to look into possible costs as well as benefits of evaluation activities in general. Additionally, various purposes for conducting SA evaluations and major challenges are discussed in detail in this section.

9.3.1 Benefits

First of all, SA evaluation saves time and effort for changes related to poor design decisions, which without evaluation would probably have been detected later on software development process, and would have costed a lot more. So, there are apparently financial benefits, excluding the case that the evaluation would cost more than its benefits. Therefore, Clements et al. introduced rules on when to conduct an evaluation in [CKK01] : i) hold the evaluation when the development team start to make decisions that depend on the architecture; and ii) when the cost of undoing those decisions would outweigh the cost of holding the evaluation. For the second rule, a team basically should conduct an additional analysis as discussed earlier.

Additionally, SA evaluation has some extra benefits like positively affecting the communication between stakeholders. Also, conducting evaluation filters out some possible defects and/or unclear specifications from SA documentation. Thus, this basically results in avoidance of unexpected modifications later on. Moreover, conducting evaluations reveals and highlights some functionality or requirements that could have been forgotten and not been discussed thoroughly otherwise. This may naturally result in misunderstandings, in later stages of development, and some additional inevitable costs later on. These matters are discussed in more detail in [RG08].

9.3.2 Costs

Conducting most of the SA evaluation activities requires the highest possible level of expertise in the field. Therefore, evaluation teams are made up of experts of various backgrounds. This naturally reduces development productivity in some other ongoing projects, or even might be affecting the current project. Although it most probably introduces some other costs, the cost of reduced productivity can be mitigated by constantly refreshing the evaluation team, which is of course possible mostly in large organizations. The problem with this is that, new team members would need some time for training sessions, until they learn the current state, etc., which in turn introduces some additional time and resource costs. Nevertheless, such costs don't mean much for large organizations and are part of their daily routine.

Because of lack of experience and maturity, for start up companies, on the other hand, these costs can mean a lot, especially when they are beginning to apply architecture evaluation to their development life-cycle. These matters are discussed in more detail in [GLP+97]. Some SA evaluation methods require performing simulations and/or creating prototypes during some of the phases, which is very costly most of the time. However, these procedures are part of the standard software development process anyway.

9.3.3 Purposes

The most concrete aim of architecture evaluation is to identify potential risks and verify quality requirements. For this, evaluation of the quality of different aspects of architecture is needed. We define these aspects as quality attributes, which we introduced in previous section. The nature of the aspect or the combination of these aspects(or quality sub-factors), which are being evaluated, determines the actual purpose of the evaluation. Apart from major goals, there are some other purposes for SA evaluation, which can be listed as follows:

To choose among several different SA design alternatives. Many times, there are several alternatives for specific parts of SA. In this case, evaluation is conducted to somehow score all alternatives and be able to compare their results.

Product line potential. This is analysis of several, independently designed architectures, to evaluate how well they fit together in a product line. For this purpose, several quality attributes for each SA may be evaluated to get the final results.

Reuse potential. To decide if reuse candidate fits the architecture, evaluation if conducted to find out the degree of dependencies, need for adaptation, cost of performing the operations and etc.

Completeness. When part of documentation such as some of the SA views are missing, static SA evaluation methods can be used to recover the whole SA from the implementation, in the later stages of software development process. This can be used to ensure the completeness of documentation.

These and some more are given in: [KLMN06].

9.3.4 Major challenges

Although it is very beneficial to conduct SA evaluation, it is a challenging process overall and not every organization has the capacity to conduct evaluations. Some major challenges described in [GLP$^+$97] are as follows:

Evaluation teams should be gathered from experienced people of different organizational levels of a company, so that each team member would have a different technical as well as non-technical background. However, this could be costly for reasons discussed on costs of SA evaluation.

SA and requirements are very hard to specify precisely. This is the main reason why most of the evaluation methods we would look at in the next section require involvement of stakeholders at least in initial steps of evaluation. The problem with this, is that it is difficult to gather all stakeholders together.

9.4 Methods of SA evaluation

In this section, we first introduce some categories of SA evaluation methods from different perspectives. Then, several methods are discussed and compared in more detail.

9.4.1 Classification of SA evaluation methods

SA evaluation methods are often classified according to different perspectives. As SA evaluations can and should be conducted(according to [GLP+97]) several times during any phase of software development process, they are commonly classified by the time, they are applied to the SA. Thus, early evaluation is conducted before implementation of a system, where late evaluation is conducted after start of implementation [RG08].

Another perspective that is commonly described [GLP+97, RG08, DN02] is whether evaluation method is based on qualitative or quantitative matters. Often, they are also referred as questioning techniques and measuring techniques respectively. Questioning techniques basically used to evoke discussions, by generating some questions on specific or general requirements and design decisions. There are three questioning techniques:

- *Scenarios:* defined as *"Description of single interaction of user with system"* in [RG08]. They are a good way to define a general understanding of quality attributes, because once generated, they represent a system's functionality, by describing the set of actions which lead to the execution of a certain function.
- *Questionnaires:* is a list of questions related to quality requirements.
- *Checklists*

Measuring techniques, on the other hand, come up with solid, quantitative results that refer to the questions that may have been raised by the team members. These results are based on the measurements calculated on some metrics suggested and/or defined by the team. Measuring techniques include:

- Metrics
- Simulations, prototypes and experiments

Of course, in practice it is possible to mix the questioning and measuring techniques, which is actually done in some case studies [KDG+12]. The whole

classification is given in Figure 9.3. Moreover, regarding generality of evaluation, there can be also general, domain-specific or system-specific evaluation methods as given in [DN02]. The scenario-based approaches are mostly

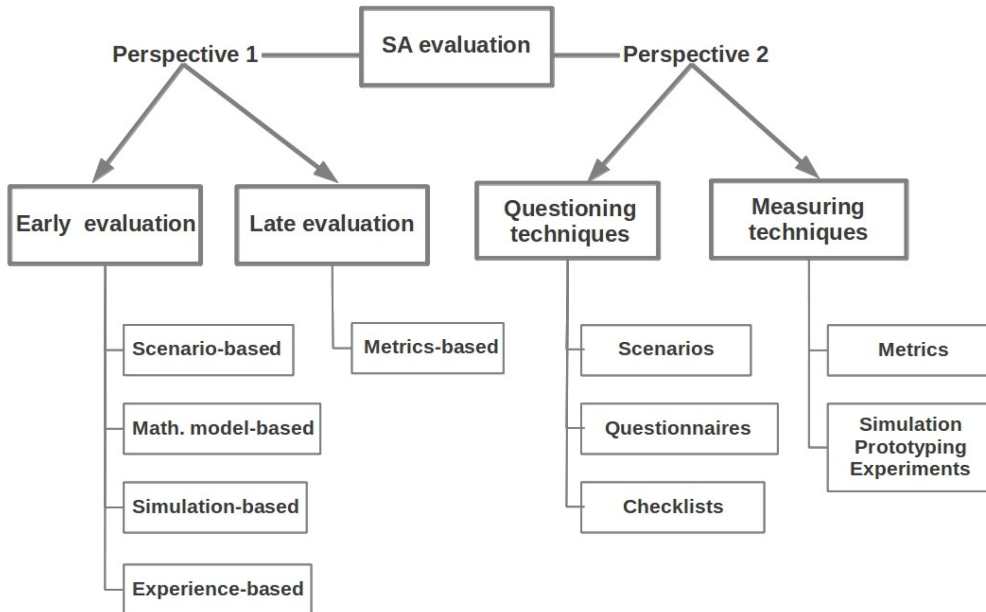

Figure 9.3: Classification of evaluation methods from different perspectives. **Perspective 1** here is, phase of development life-cycle, when the method is applied, **Perspective 2** is whether method is qualitative or quantitative

used to evaluate non-operational quality attributes such as maintainability and reusability, whereas simulation and mathematical model-based methods are more suitable for estimation of operational attributes. The Experience-based method results, on the other hand, solely depend on intuition and experience of the team.

In this paper we concentrate on the scenario-based SA evaluation during early evaluation and the metrics-based as late evaluation methods because, these methods are the most commonly applied in industry [RG08, LRVA, TLC02]. Although mathematical model-based early evaluation methods are also common, especially on estimating reliability and performance attributes, we decided to leave them out of the paper's scope.

9.4.2 SA evaluation methods comparison framework

To compare all the methods, we will be using simple framework, which is a tear-down version from [DN02]. All evaluation methods are compared and described against each item of the framework. Table 9.1 shows the criteria which we will use later on.

Item	Description
Specific goals	Which specific attributes does the method aim?
Inputs	What are the inputs of this method?
Outputs	What are the results
Experts involved	Which experts should be involved in evaluation?
Stakeholders involved	Is stakeholders attendance needed, in what level?
Activities	What are the activities of the method?

Table 9.1: SA methods characterization and comparison framework items.

9.4.3 Early SA evaluation methods

Early evaluations are helpful because most of the times it is extremely costly to make changes or fix errors and incorrect design decisions on later stages of the project. Therefore, it is useful to start early evaluation iteration even when SA itself hasn't been completely described. In that case, evaluation is conducted on specific parts of SA, iteratively going through other parts and/or enlarging the evaluation scope.

Scenario-based SA evaluation

Scenario-based methods are applied to SA with the set of scenarios of interest. The scenarios of interest are dependent on the specific goal of an evaluation. For example, if evaluation aims to assess the evolvability of a system, then these scenarios should encounter some potential future changes, referred as change scenarios [LRVA]. Afterwards, the scenarios which are not supported, or in other words, cannot be executed on the system, reveal possible errors or potential improvements. Also, scenario-based methods list the changes needed for the SA to comply to those scenarios and estimate the costs. These methods require the presence of stakeholders mostly during the generation of the scenarios, which as we stated before, is challenging and can be considered as a drawback. Another disadvantage is the fact that there is no particular number of scenarios that is needed to be generated, to assure the correctness of the SA. This is one of the reasons for an evaluation team to be consisted of experienced professionals, which reduces the negative effect. Also, the outcome is based on intuition, rather than solid quantitative results, in which case, we do not gain any specific statistical values. For example, these methods cannot estimate concrete numeric value on how costly would a change be, but only gives abstract results.

Scenario-based SA Analysis Method(SAAM)

The main goal of SAAM is to verify if the given SA representation complies to the basic principles of architecture design like architectural styles or design principles, on a desired level. During this process, it requires the evaluation team to focus on potential trouble areas which helps the team to asses the risks and detect requirement conflicts. Moreover, SAAM can be used to compare several architecture alternatives.

Specific goals. SAAM does not refer to one or several specific quality requirements. The attributes that it assesses are defined by the scenarios that are evoked by the team.

Inputs. The main inputs are problem statement, SA description and quality requirements.

Outputs. The outputs are the scenarios which refer to quality attributes of interest and also, required modifications and estimated effort, required to realize the scenarios.

Experts involved. Because this method does not address specific quality attributes, it does not require any specific experts. So the people involved in the evaluation can be any professional. However, as for any evaluation method, team members are selected among experienced professionals, because of required estimations and other activities which require special expertise.

Stakeholders involvement. Stakeholders are required to attend in requirement specification, SA description and scenarios elicitation steps.

Activities. There are six steps of conducting SAAM according to [RG08], which are as follows:

1. **Specifying requirements and design constraints.**
2. **Description of SA.** Here, candidate architectures are described by means of views [Kru95], with addition of several other views like relationship view between components(represented by UML communication diagram) and dynamic view(UML sequence diagram). The requirement of any of these views depends on the goal of evaluation. If the goal is to reason about performance, for example, then process view is required.
3. **Elicit scenarios.** In this step, presence of stakeholders is required for brainstorming sessions. The desired set of scenarios are generated according to the goals of the evaluation. This is of course, as depicted in Figure 9.4, together with prioritization of the scenarios step are done iteratively.
4. **Prioritize scenarios.** In this step, scenarios are prioritized and weighted

according to importance. The level of importance is defined by stakeholders' thoughts and opinions.

5. **Evaluate SA with respect to chosen scenarios.** The scenarios are applied by investigating the impact of the scenario on the SA. According to this impact, the scenarios are categorized to indirect(modifications needed on SA) and direct(requires no modification to SA). In case of an indirect scenario, changes needed to be made are explored, and effort is estimated by listing all components and relations which are affected by this modification. In case, where the goal of evaluation is to choose the best SA candidate, the results of all alternatives are compared in the end. Naturally, the weight of each scenario is taken into account in this process.

6. **Interpretation of results.** Finally, the results are analyzed in this step. For example, the high number of relations among components of unrelated scenarios may indicate high coupling. However, as stated earlier, SAAM does not give any precise metrics of fitness, for example, but rather just some estimates for comparison on scenario basis.

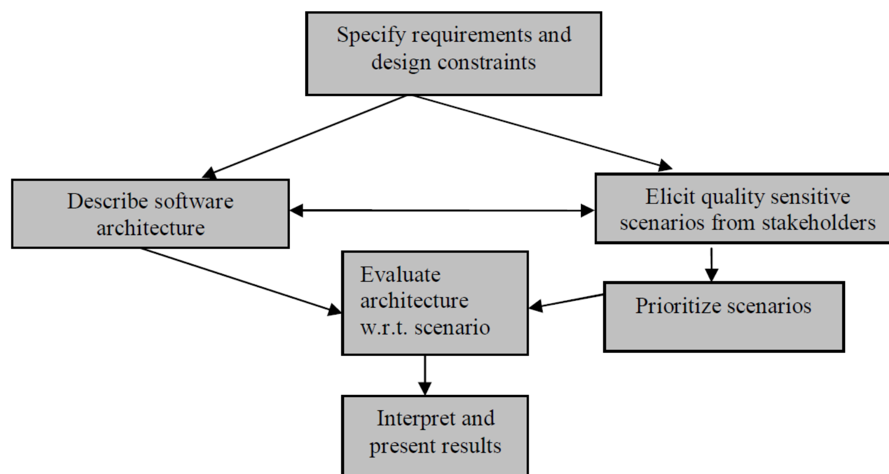

Figure 9.4: Activities of SAAM [RG08].

The Architecture Trade-Off Analysis Method(ATAM)

Although SAAM is very useful in early evaluation of SA, this method does not take into account the relationship among quality attributes. It does not analyze the effects that one attribute may have on other attributes. ATAM on the other hand, shows how quality attributes interact with each other. This is crucial mainly because, the trade-off decision needs to be made.

Specific goals. Same as SAAM, ATAM does not refer to specific attributes and covers multiple attributes. The range of attributes varies by the the generated scenarios.

Inputs. Inputs of ATAM are architecture, business goals and perspectives. It does not mean that these are final versions, and in fact, are redefined when the evaluation reveals some problems.

Outputs. The outputs are sensitivity points, trade-off points and potential risks.

Experts involved. ATAM also does not concentrate on specific quality attribute, allowing evaluation of various attributes. Therefore, no specific expert involvement is required.

Stakeholders involved. All stakeholders involvement is required by ATAM.

Activities. ATAM can be used at various stages of software development process. However, it is crucial to find the trade-off between the quality attributes early in the SA design process. The steps of conducting ATAM might seem similar to SAAM, however, there are some differences and additional steps. In brief, the steps are as follows:

Presentation

1. **Present the ATAM.** The method and it's activities are explained to stakeholders.
2. **Present business drivers.** The business goals and motivation that drive the project are described by project manager.
3. **Present architecture.** With the emphasis on how it refers the business goals, the candidate architecture is described in this step.

   Investigation and analysis
4. **Identify architectural approaches.** Architectural approaches are identified.
5. **Generate quality attribute utility tree.** The high priority scenarios are elicited with respect to quality attributes using utility trees. It is done by specialization, starting from an attribute towards the scenarios that affect that attribute. Three type of scenarios are important for ATAM, which are
   - use case scenarios: depict typical uses of the system.
   - growth scenarios: represent potential changes and modifications to the system in the future.
   - exploratory scenarios: which represent extreme cases, which help to explore the system's peak performance(referred to any quality attribute).

6. **Analyze architectural approaches.** During this step the architectural approaches that refer to the scenarios are identified and analyzed. The sensitivity points and trade-offs between quality attributes are also identified during this step. Sensitivity points are key architectural decisions that have a huge effect on the system.

   Testing

7. **Brainstorm and prioritize.** The scenarios are further filtered by voting, which involves all stakeholders.

8. **Analyze architectural approaches.** This step is a reiteration over step 6, but this time with a focus on scenarios with the highest priority. These scenarios are chosen to be test cases and therefore may reveal further risks, sensitive points, etc.

   Reporting

9. **Present results.** Previously collected information on scenarios, architectural approaches, utility tree, sensitivity points, etc. is presented in a systematic reports and presentation to the whole group of stakeholders. These reports also may contain mitigation strategies and so on.

Important point here is that the order of these steps is not in a straightforward waterfall process, but rather can be executed in parallel, previous steps might be restarted or internal iterations can be performed. This can depend on various aspects such as availability of involved personnel or artifacts and so on. ATAM is discussed in more detail in [KKC00].

Scenario-Based Architecture Re-engineering(SBAR)

SAAM does not provide a specific way for evaluating operational quality attributes, which is understandable, because it is hard to estimate run time specific attributes only by scenarios. Unlike SAAM, SBAR provides four techniques for quality attribute estimation, to assess each attribute in a more suitable way. These techniques are scenario-based, simulation-based, mathematical model-based and experience-based, as we discussed earlier. For example, for operational attributes, such as stability and performance.

In general, SBAR aims to estimate how well the SA reaches it's quality requirements. This is achieved by iterative transformation of SA, until it meets all requirements. First, the initial version of SA is defined. Then estimation is done for each quality attribute by the chosen approach. The results are interpreted and compared. If all quality goals are met, the re-engineering process is finished. Otherwise, SA is transformed and taken into next iteration.

In case of scenario-based technique, suitable scenarios are elicited for each attribute and manually executed. The ratio of scenarios that can be handled

by SA and the ones that cannot be handled defined the level of quality for each attribute. For scenario elicitation, the team have two options, *complete* and *representative*. In complete approach, all possible scenarios should be generated, where as in representative option, only extreme cases are chosen, which represent all different scenarios. Both approaches have disadvantages as for first approach, it is merely possible to find all possible cases, especially for a large architecture and for latter approach, it is unclear how the representative scenarios are chosen.

Additionally, there are five transformation techniques, provided by SBAR. These are:

- Impose architectural style
- Impose architectural pattern
- Apply design pattern
- Convert non-functional requirements to functionality
- Distribute requirements

SBAR does not require many stakeholders to be involved in evaluation. The evaluator is the designer of SA. SBAR is described in more detail in [BB98].

Architecture-Level Modifiability Analysis(ALMA)

The scenario-based methods discussed so far, address multiple quality attributes. ALMA on the other hand, is designed to evaluate only modifiability. This method is specialized on modifiability, therefore it analyses this attribute in more detail, hence resulting in more clear estimations. However, it needs to be accompanied by other evaluation methods, because, as we already, it is crucial to analyze several quality attributes and their relationships.

Specific goals. ALMA specifically assess modifiability of SA.

Inputs. Architecture description.

Outputs. Depending on the goal which is set in the first step of evaluation, output may slightly differ. It can be identification high-cost changes, estimation of possible maintenance costs or comparison of alternative architectures with respect to maintenance costs.

Experts involved. The SA designer and several evaluators are main experts involved in the process.

Stakeholders involved. Stakeholders' involvement is required almost in all steps of the evaluation.

Activities.

- **Goal setting.** There are three goals that ALMA can be used to achieve. In this step, it is crucial for the stakeholders to decide on the purpose of the evaluation. This is important mainly because, the scenarios that will be generated slightly differ according to the goal of evaluation. These goals are:
  - Maintenance cost prediction
  - Risk assessment
  - Alternative SA-s comparison and selection
- **Architecture description.** In this step, the representation of the part(or view) of architecture is collected, which is related to modifiability. This representation needs to contain information on components of the system, the interaction between these components and their relationship with the system's environment.
- **Change scenario elicitation.** The scenarios that are generated in this step of evaluation are dependent on the aim of evaluation chosen in first step. If the goal is to estimate maintenance costs and effort then change scenarios with high probability are chosen. On the other hand, if the goal is to assess risks, the scenarios that underline those risks are generated. Finally, for comparison of alternative architectures, the team needs to elicit the scenarios which accentuate those differences.

  An example change scenario elicitation is given in [LRVA], for the aim of identifying risks. The scenarios that are generated are somewhat *complex* changes that likely to occur. Also, the complex changes are categorized into changes that involve different system owners, that strongly affect the architecture and those that introduce version conflicts, etc.
- **Change scenario evaluation** In this step, the effects of the change scenarios on architecture is estimated. This consists of three main steps:
  - Determine the directly affected components.
  - Identify the nature of effects on these components.
  - Determine ripple effects.

  The most difficult step here is the last one, because the ripple effect can propagate into more than several layers. It may even take a cyclic form and hence, make it very hard to detect the level of propagation. Moreover, there is not much information on ripple effects on architectural level, which makes it even harder to estimate. In such case, the experience of team members helps to do an average estimation. Also, there some quantitative approaches on calculating the approximate impact of changes on SA.
- **Interpreting results** This step is straightforward, the collected and estimated data is processed and visualized.

Adaptability Evaluation Method(AEM)

AEM is designed to evaluate adaptability of a system. It defined how adaptability can be represented in SA and how this SA can be assessed and validated whether quality goals are met. AEM consists of four phases which are as follows:

- **Defining adaptability goals.** In this phase the adaptability requirements are collected with the involvement of stakeholders. These requirements then refined and mapped to specific functionality of a system. Also, trade-off analysis is conducted to assess the risks of conflicts with other quality attributes.
- **Representing adaptability in SA.** This phase guides the team on mapping the selected scenarios to elements of SA so that it can be evaluated directly in architecture.
- **Adaptability evaluation.** AEM provides both qualitative as well as quantitative techniques for evaluation of SA for adaptability requirements. These techniques are rather complementing each other to gain more thorough results. The goal of qualitative evaluation is to calculate the impact of scenarios and find out the adaptability level under these changes. Qualitative evaluation aims to reason about the problems that may occur if the adaptability requirements are not met by the candidate SA.

AEM is thoroughly discussed in [Tar07].

9.4.4 Late SA evaluation methods

Late evaluation methods are helpful for static evaluation, in other words, mapping the implemented architecture to earlier intended SA to check whether it complies to addressed quality requirements. There can be various reasons for deviation; the developers may not always follow the the best way to implement modifications, many different developers may work on same implementation and change according to their understanding and etc. Therefore, the process of evaluation should be iterative, so that evaluation would be applied on each implementation and etc.

Metrics-based SA evaluation

Metrics-based approaches are often applied late in the development process, because it is easier to define metrics after the implementation is started.

Tvedt et al.'s method

A metric-based approach suggested in [TLC02] helps the analysis team to prevent the architecture from deviation by regular and systematic application of this method. Here are the steps of this method in brief:

- First step is to choose a perspective, to be evaluated. On the other hand, choose a specific goal, a one or set of quality attributes that we want to prevent from degeneration.
- Then, the team needs to define the design guidelines and respective metrics with respect to selected perspective. For example, if flexibility or maintainability are under assessment, the guideline could be that components should be loosely coupled and metric can be derived from it, to be some threshold value for the number of relations between certain components.
- Afterwards, the planned SA is defined. This is basically the input of the evaluation, the SA which is under assessment.
- Next, the actual implemented architecture is derived from the source code. This is the abstracted version of implementation. There are some tools available for reconstruction of static views of SA. Different tools are available for various programming languages.
- The architectural deviations are identified in the next step. These are the differences between actual architecture and the planned one. The differences can be, for example, the values of metrics which exceeded the predefined threshold.
- Finally, some high-level recommendations towards changes, to recover the SA to planned version are formulated. Moreover, after the changes are applied, the whole process is repeated again, for verification of correctness of changes.

This approach, together with a case study is presented in more detail in [TLC02].

9.5 Conclusion

In this article, we first started with defining main terms, needed for discussion on SA, SA quality and it's evaluation. Next section, contains information on benefits and costs, purposes and major challenges of conducting SA quality evaluations. This basically is to answer why and when evaluations need to be conducted and when not. Afterwards, we discussed the classification of SA

quality evaluation methods and presented several important evaluation methods. To conclude, it can be seen that scenario-based methods appear to be used more than other evaluation methods.

Bibliography

[BB98] P.O. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. In *Proceedings of International Conference of Software Reuse 5 (ICSR5)*, June 1998.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, volume 2nd of *SEI series in software engineering*. Addison-Wesley Professional, 2003.

[BE02] Rami Bahsoon and Wolfgang Emmerich. Evaluating software architectures : Development , stability , and evolution. *Foundations*, 2002.

[BKLW95] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality attributes. Technical report CMU/SEI-95-TR-021, Software Engineering Institute, 1995.

[CKK01] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.

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

[GLP+97] Gregory Abowd, Len Bass, Paul C. Clements, Rick Kazman, Linda M. Northrop, and Amy Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Technical report, 1997.

[IEE90] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 1, 1990.

[IEE98] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1998*, 1998.

[ISO11] ISO/IEC/IEEE. Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1 –46, 1 2011.

[KDG+12] Heiko Koziolek, Dominik Domis, Thomas Goldschmidt, Philipp Vorst, and Roland J Weiss. MORPHOSIS : A Lightweight Method Facilitating Sustainable Software Architectures. 2012.

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon Uiversity, Software Engineering Institute, 2000.

[KLMN06] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. Static evaluation of software architectures. *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–294, 2006.

[Koz11] Heiko Koziolek. Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 3–12, New York, NY, USA, 2011. ACM.

[Kru95] P Kruntchen. Architectural Blueprints - The "4+1" View Model of Software Architecture. *IEEE Software*, 12(November):42–50, 1995.

[LBK+97] Chung-horng Lung, Sonia Bot, Kalai Kalaichelvan, Rick Kazman, Canada Ky, and Canada Nl. An Approach to Software Architecture Analysis for Evolution and Reusability Department of Computer Science Nortel University of Waterloo. pages 144–154, 1997.

[LRVA] Nico Lassing, Daan Rijsenbrij, Hans Van Vliet, and About Alma. Experience with ALMA. (c).

[RG08] Banani Roy and TCN Graham. Methods for evaluating software architecture: A survey. *School of Computing TR*, 2008.

[Sha90] Mary Shaw. Toward higher-level abstractions for software systems. *Data Knowl. Eng.*, 5(2):119–128, July 1990.

[Tar07] Pentti Tarvainen. *Adaptability Evaluation at Software Architecture Level*, volume 02. IEEE Computer Society, Washington, DC, USA, compsac '0 edition, 2007.

[TLC02] Roseanne Tesoriero Tvedt, Mikael Lindvall, and Patricia Costa. A process for software architecture evaluation using metrics. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 191–, Washington, DC, USA, 2002. IEEE Computer Society.