IT MANAGEMENT WITH ENTERPRISE ARCHITECTURE

PONTUS JOHNSON, ROBERT LAGERSTRÖM, MATHIAS EKSTEDT AND MAGNUS ÖSTERLIND

TABLE OF CONTENTS

Introduction

1

An introduction to the management of IT with Enterprise Architecture.

Chapter 1 Introduction

Today, there are information systems for most of the tasks performed in an enterprise. There are customer management systems, contract management systems, product design systems, production systems, financial systems, human resource systems, business intelligence systems, asset management systems, waste management systems, document management systems, workflow management systems, and hundreds of other systems. In recent years, these systems have been integrated with each other to such an extent that it is oftentimes necessary to view them, not as hundreds of different systems, but one single system of systems.

The resulting enterprise-wide information system is under constant change. Every year, new systems are developed and introduced, old systems are extended, modified, integrated with each other, and retired. In large enterprises, these changes are the result of many different stakeholders' requirements and many developers' actions. It has become increasingly evident that there is a need to plan and manage the evolution of this system in order to keep chaos at bay.

In this book, we describe an approach to enterprise information systems management that relies on models of the information systems and their environment. The main idea is very old. Instead of building the enterprise information system using trial and error, we propose a set of models to predict the behavior and effects of changes to the system. The enterprise architecture models allow reasoning about the consequences of various scenarios and thereby support decision-making. In order to predict whether scenario A or B is preferable, three things are needed. Firstly, models over the two scenarios need to be created. Secondly, it is necessary to define what is desirable; the goals. Do we want the systems to support business process efficiency or is organizational flexibility more important? Is high system availability more important than high information security or maintainability? Thirdly, we need to understand the causal chains from scenario selection to goals. Scenario A features hardware redundancy that positively affects the system reliability which in turn improves the service availability, leading to more efficient business processes. However, scenario B is built on a loosely coupled technology, which promotes the modifiability of the system. This, in turn, may be expected to have positive effects on the organizational flexibility.

In this first chapter we present a brief history of enterprise information systems, the most common types of systems found today and the general architectures for the interaction between these systems, a brief history of enterprise architecture, and finally some background information describing our view of enterprise architecture for IT management.

1.1 A brief history of enterprise information systems

In this section, we describe three epochs of enterprise information systems. We consider each era in terms of the technology, the users, and the maintainers and developers.

1.1.1 Mainframes and mini-computers

The first electronic computer was the ENIAC (Electronic Number Integrator And Calculator), which was developed in 1946 by the accounting industry and the emerging electronics industry in a joint effort.

Comprising of 17,468 vacuum tubes, the ENIAC filled a large room requiring air conditioning due to the intensive heat caused by the machine. The ENIAC and its successors were programmed one program at a time and were served data using punch cards - stiff pieces of papers with presence or absence of holes in predefined positions. The end-users were typically engineers, who communicated punch card decks to an intermediary, the mainframe operator, who in turn ran the programs. Executing a program could take hours, sometimes days, and when the program terminated, the user received his card deck back together with the output data.

In the 1950's the demand for computers slowly increased and mainframes were sold to large organizations including universities, corporations and civil and military government agencies. Universities bought computers to perform scientific calculations of engineering problems, whereas enterprises mainly bought computers for business-oriented issues, such as managing payrolls. The government bought computers for these purposes and also for supervision and real-time control over physical processes, such as air space surveillance. Computers were designed for specific purposes, so a computer designed for handling business purposes could do just that, and not be utilized for scientific calculations or real-time control. As the needs of organizations extended into several different areas, this machine specialization became a problem of both economics and convenience. Another problem was that differences between jobs in for example memory and storage usage often required tedious manual interventions when different programs within the same domain were executed. The operating system might have needed to be reprogrammed and sometimes the computer

Movie 2: *On the Importance of Information Systems*

needed to be rewired as well. As these problems grew, the need for an all-round computer emerged [1].

In the middle of the 1960's IBM released the S/360 mainframe, the first general-purpose computer. Besides enabling different types of programs to be executed on the same computer simultaneously, this was the first computer making a difference between architecture and implementation. This enabled IBM to release a suite of compatible designs at different price levels, promising customers that migration to more powerful versions would be possible as their needs grew. Some consider the introduction of the IBM S/360 on the market as the biggest advancement of IT in business and society as a whole [2]. In the same period as the IBM S/360 entered the market, the smaller minicomputers started to establish their own markets. The minicomputers possessed the same components as large mainframes but with reduced memory capacity and slower processing speeds. Before the advent of the minicomputer industry in the 1960's, companies wishing to automate their data processing were forced to use mainframes. Because of its performance limitations the minicomputer did not impose a threat to the mainframes. Instead, the minicomputer, with about the size of about a refrigerator, opened up new application areas and allowed managers to choose computers with substantially lower costs than mainframes. While, the mainframes were at that time handled by specific operators who managed the direct interaction with the machine, the smaller minicomputer enabled direct interaction between the multiple users and the computer. Their establishment in the market in late sixties has been seen as a cultural, technological and economic phenomenon [2]. One of the earlier successful models was the PDP-8 from Digital Equipment Corporation. The data centers managing the computers, were typically offshoots from different organizational departments, such as accounting. The specific-purpose characteristics among early computers often further increased the decentralization of system management and often resulted in a number of different data centers connected to each department, one for each specific purpose and computer. As computers broadened their application domain, these centers were centralized into larger ones, providing services to multiple organizational departments. The increased volumes of data gathered in databases and the criticality of the data to business called for methods to assure rigidity and handle the system configurations. With centralized systems, a centralized IT governance structure became possible.

1.1.2 Terminals, workstations and PCs

In the seventies, increased memory size led to a shift from batch-oriented operating systems to online processing systems, where data could be entered and response would be immediate. Until this point, mainframes had mainly supported back-office functions. In the seventies, local networks were set up, connecting the mainframes and minicomputers to terminals, allowing hundreds of users to simultaneously interact with the systems. At this time, a yet smaller computer entered the scene; the microcomputer, intended only for a single user. These machines and their operating systems, such as Microsofts DOS, were simple in comparison to minicomputers and mainframes.

As the costs of the microprocessor hardware became lower and the cost of deploying local area networks decreased in the eighties, the

microcomputers popularity increased in business applications. The microcomputers eventually evolved into personal computers, replacing the terminals, while the minicomputers migrated into servers for PC networks. The use of PCs instead of mainframes and terminals had an important impact on the software industry. Traditionally, software and hardware had been bought from one single vendor, and in the fifties and sixties, computers were often leased from developers for a time span of several years. As the PC revolution came, the software market changed and was filled with different vendors selling high volumes to push prices down.

The PC revolution in late eighties and early nineties moved computing power from the centrally located mainframes to distributed microcomputers. This made users less dependent on data centers and computer manufacturers, but with consequences for the IT department. Configuration management, backups and security became more difficult as PCs replaced terminals and mainframes. The shift towards online systems and databases containing business critical information introduced areas such as online transactions, database management and access security. This meant new responsibilities for the IT department.

Moreover, the users' increased independence from the IT department and the regional and international expansion of many enterprises drew the previously centralized IT organization closer to the business and IT management decentralized to become more responsive to the needs from business [1].

1.1.3 Extensive networks

As the business value of personal computers became clear to enterprises, the next logical step was to link them together. The US Defense Advanced Research Projects Agency's ARPANET initiative in the sixties and early seventies was the dawn of efficient and large-scale networking. In the early nineties it reached a global impact with the Internet. As the interconnected networks grew, the power and number of applications using them increased, and consequently their importance grew. With the evolution of extended computer networks, the computer itself lost its position as the main focus, which instead shifted to the availability and capacity of the networks.

The weak economy during the early 1990 forced companies to save costs. The decentralization of IT management during the eighties had made the IT departments more responsive to the needs from the business, but with duplicated efforts and lost economics of scale as a consequence. Outsourcing was seen as one way of controlling costs and when Kodak outsourced their data center operations to IBM on a ten-year deal in 1989, a new trend was born. Not all enterprises adopted the trend of outsourcing, but since they faced the same problem with high costs, many enterprises again started to centralize their IT management. The previous decentralized organization structure had besides inefficiency resulted in various, incompatible, technologies across enterprises that complicated this centralization process.

In addition to a technically heterogeneous information system portfolio, the need for enhanced IT management is also the result of the business's everincreasing dependence on the information systems as a means to perform work and communicate internally and externally. The share of IT in

business equipment investments in USA rose to above 50 percent in year 2000 [3]. This meant that the demand of IT knowledge among personnel in general and IT specialist in particular increased. The importance of information systems is reflected in the fact that the role of the Chief Information Officer (CIO), with an enterprise-wide responsibility for IT, is nowadays typically reporting directly to chief executives or the president [4].

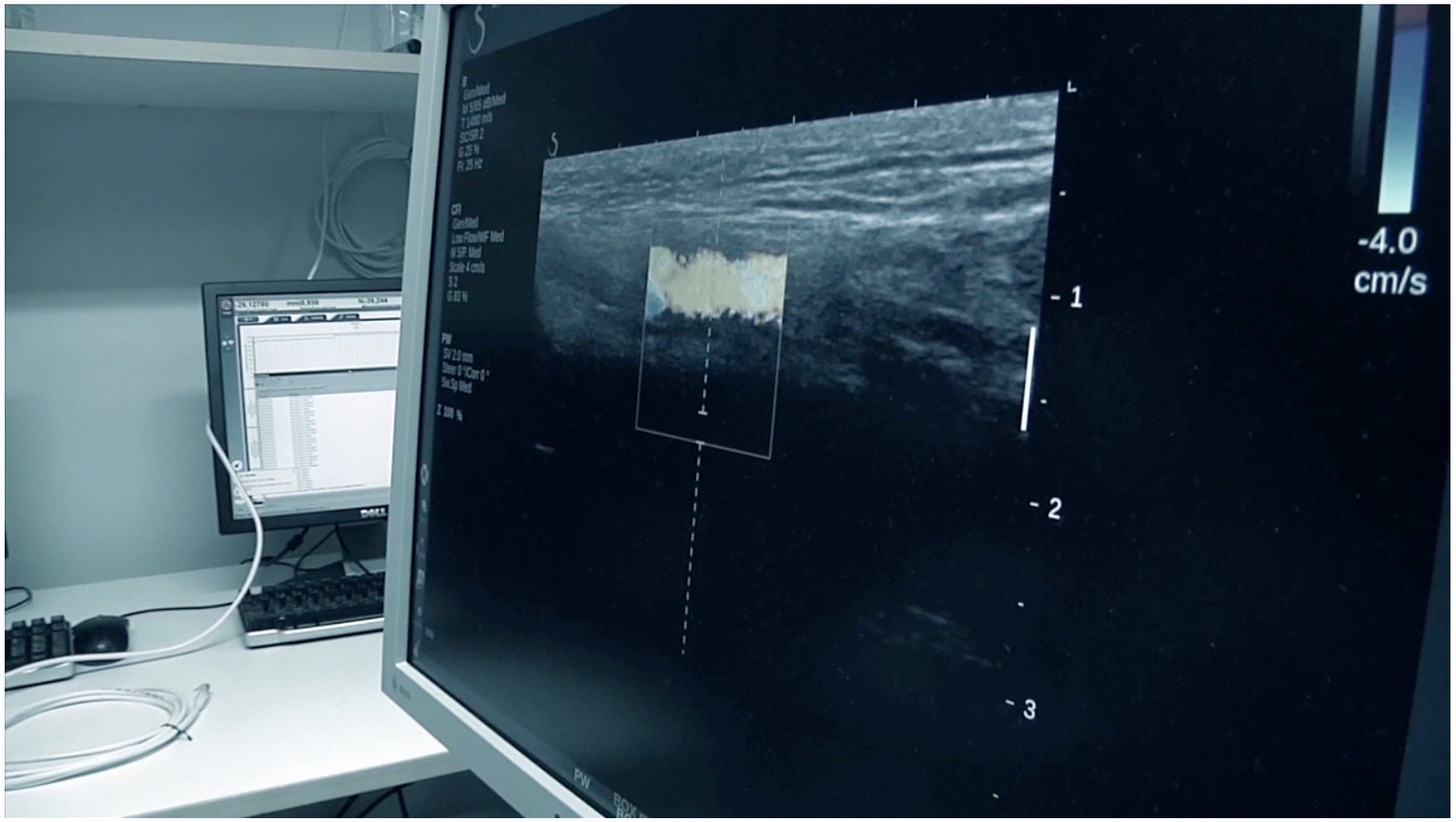
1.2 The information systems of today

Today, information systems are employed in virtually all parts of modern enterprises and the systems as such are extremely business critical being an integrated part of most departments. In this section, we review the most common services provided by the systems, and we also consider four very general architectures for the interaction between the systems.

1.2.1 Application services

We begin the information system walk-through with those systems that are closely related to the physical world and move toward the more abstract domains.

Industrial control systems provide measurement and control functionality for large-scale, possibly geographically distributed, physical processes, such as electricity generation, electric power transmission, electricity distribution, district heating, water distribution, pulp and paper production, manufacturing, defense, transport, chemical and telecom industry. In one end of the system, the sensors and actuators either measure or act on the process objects, and in the other end, human operators typically monitor the process. Functions provided include forecasting and planning,



Movie 3: *On the Difficulties of Information Systems Management*

remote measurement as well as control over the physical process, automatic control, and alarm and notification functionality.

Product management systems, or product life cycle management systems, handle most functions used during a products lifecycle, from conception and design to manufacturing and service. In the conception phase, these systems provide functionality for requirements management and design. The design phase is supported by computer aided design (CAD) functionality, supplying the developer with 2D or 3D drawing functions, as well as simulation, validation and optimization functions. Also the manufacturing phase is supported by for instance computer aided manufacturing (CAM) functionality. Other functions supported in this phase are process simulation and production planning. Further support may be provided for product testing. In the service phase, functionality provides customers and service engineers with information on the finished products, such as repair and maintenance information. Special kinds of product management systems are software development tools. These provide functionality for requirements management, design, production (compilation) and testing of software products.

Asset management systems provide functionality used for organizing and processing assets within an enterprise. The assets can be digital, for instance images, documents and presentations. They can also be of a physical nature, e.g. equipment and facilities. Asset management systems usually provide functions for collecting, managing, searching, retrieving, and archiving information about the assets. An important issue for these systems is the availability of the stored information. They may therefore offer mobility solutions, for instance allowing remote access via handheld terminals. Systems designed specifically for managing real estate are sometimes called property management systems.

Inventory management systems are used for monitoring quantity, location and status of the enterprises inventory, as well as supporting the shipping and receiving processes.

Geographical information systems may be used to maintain records of the whereabouts of the assets in enterprises with a geographically distributed infrastructure. These systems are based on databases of geographical maps over the region of interest. Satellite photographs, road maps, and asset information are then superimposed on these maps.

Human resource management systems are designed to support the responsibilities of the human resource department. These systems therefore provide functionality for the tracking of employee data like personal history, addresses and phone numbers, but also for tracking the employees skills and capabilities. The automatic gathering and calculation of information for the salary payment process like time, attendance, deductions and taxes are other functions of human resource management systems. Additionally, the systems may support administration and tracking of employee participation in benefit programs as well as the planning and tracking of learning activities.

To make sure that work is performed according to procedure, workflow management systems may be employed. These are systems that help organizations to specify, execute, monitor, and coordinate the flow of

work. The typical use is the flow of documents through an administrative process, e.g. the flow of tax declarations through the internal revenue service, where various persons and applications calculate, check, and approve of the various aspects of the document. Workflow management systems often feature modeling functionality, where the workflow is designed; simulation functionality, where the design is tested; and execution functionality, ensuring that the workflow is followed.

Service management systems are reminiscent of workflow management systems. An example of a service order in the power industry is the order to repair a power meter reader in a customers house. Such orders may be initiated by the customer call center, approved by the power distribution center, accepted and performed by the fieldwork subcontractor, and reported back to the distribution center and the customer call center. Service management systems support the flow of these activities. Related to workflow and service management systems are workforce management systems. These typically manage scheduling of the work force, time and attendance. They may also encompass functionality that is further described as human resource systems. Another type of workflow management systems used for planning and follow-up of work are project management systems. These typically provide functionality for scheduling projects, allocating resources, and follow-up of the planned projects.

Customer management systems, or customer relationship management systems, are designed to support sales, service and marketing in their contacts with the customer. The first function of these systems is the order management of customer information, so that for instance the service department is notified of what the sales department has sold to the customer. Customer relationship management systems also provide support for the management of potential customers, management of contracts with the customers, and analysis and forecasting of customer behavior. These systems are often integrated with workflow management systems.

An important part of the relationship to the customer is the billing. Billing systems provide support for creating and distributing invoices for performed services at suitable dates. For service-providing companies, billing systems can be very important to the business because it enables the company to differentiate the price of its services according to various circumstances. Telecom operators, for instance, typically charge their customers very differently for the same service depending on when and how much they use it.

Companies also need to manage the input side of the business, i.e. relations to suppliers. For these purposes, there are procurement management systems. There are also brokering systems to be used by both customers and suppliers. Prime examples of such systems are trading systems, providing a market place for the exchange of goods and services.

Management information systems are computer systems that present high-level summary information that assists management decision-making. One part of the management information system is typically the business intelligence system, which is a system employed to gather and aggregate information relevant for corporate management. An-

other part of management information systems is the decision support system, which may analyze the gathered information to aid the process of decision-making. The information gathered and analyzed typically concerns customers, competitors, business partners, the economic environment, or internal operations.

Financial systems are used for accounting and reporting. Business transactions are recorded and fixed assets and inventory are financially managed. Bank relations are supported, as well as tax accounting. Functionality for the reporting of financial statements is also provided.

1.2.2 Basic technical architecture

In the previous subsection, we presented different kinds of services provided by the most common information systems in modern enterprises. In this subsection, we briefly consider how these systems relate to each other.

Not so long ago, the various systems presented in the previous subsection were produced by different companies, procured independently of each other, and installed in different departments as what is often labeled as islands of automation, cf. Figure 1. Each system had its own hardware, gathered its own information from the environment and its users and performed its functions without regarding other systems in the vicinity. In those (normally rare) cases where one system required information available in another, information transfer was performed manually.

As the information systems grew in scope and number, the information they stored and the information they would benefit from having access

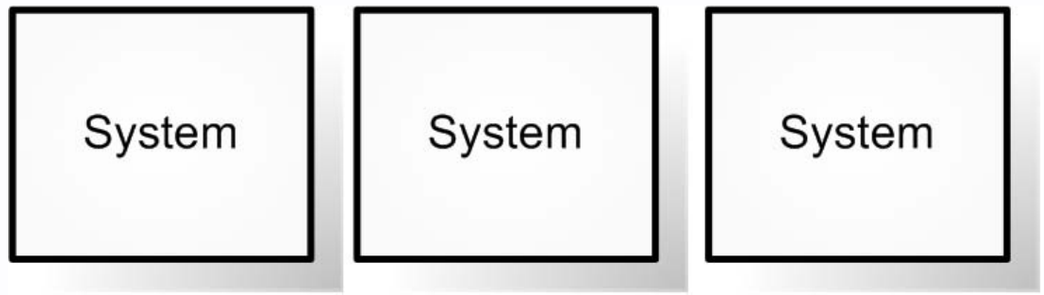


Figure 1: *Islands of automation*

to also increased. This situation led to demands for automated exchange of information between the systems. Technological development responded rapidly to these demands by devising engineering methods for connecting two systems to each other. Billing systems were thus connected to the financial systems in order transfer accounting information. Product management systems were connected to industrial control systems to allow the exchange of production-related information. Human resource systems were connected to work management systems in order to transfer information for schedules and salaries. And so on. Soon, however, the number of connections between systems became overwhelming. The common way of presenting this problem is by considering the introduction of a new system in an existing enterprise information system environment. Assuming that the new system needs to be connect to all other systems, this will be manageable when there are three, four and five systems, but when the number of systems reaches twenty, then there is also a need for twenty new separate

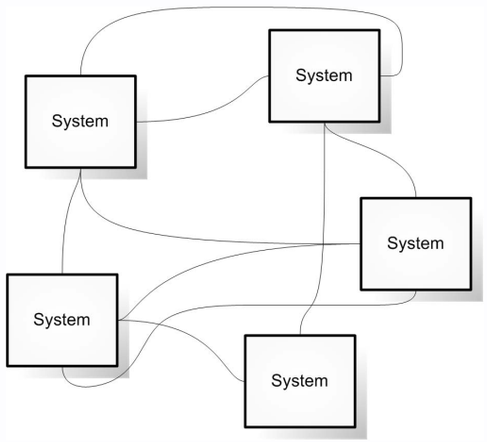


Figure 2: *Spaghetti architecture*

connections between the new system and the existing ones. If all systems connect to each other, the number of connections grows quadratically with the number of systems. The resulting mass of unmanaged connections is sometimes called "spaghetti architecture", cf. Figure 2.

In order to mitigate the problems of the spaghetti architecture, some information system vendors began offering more comprehensive solutions, providing many of the required functions. The attractive proposition with these systems is that the customer company only needs to buy one or a few systems, not hundreds. With such "suite architecture" the integration problems are thereby reduced dramatically, cf. Figure 3. These multi-functional systems generally have their origins in the administrative systems, such as the financial and human resource management systems, and they are commonly called enterprise resource planning systems. Today, these systems offer functionality covering virtually all application services described in the previous subsection. However, the enterprise resource planning systems' legacy from the administrative domain is still visible; many users complain that these systems cannot provide sufficiently high-qualitative functionality in other application domains, notably the technical domain of e.g. production management and industrial control systems. Furthermore, many customers are reluctant to make themselves too dependent on a single vendor, worried that this will reduce their bargaining position. Finally, many organizations have encountered difficulties when attempting to modify enterprise resource planning systems to their business operations; it is sometimes claimed that it is easier to make the organization fit the system than vice versa.

In an attempt to avoid the pitfalls of both spaghetti architecture and suite architecture, many companies have turned to a middleware-based architecture, or broker architecture (cf. Figure 4). Broker architectures try to avoid the negatives of the suite solution by returning to the multi-vendor scenario. This is sometimes called the best-of-breed approach. In order to not re-experience the problems of the spaghetti architecture, the point-to-point connections between systems are substituted by a centrally lo-

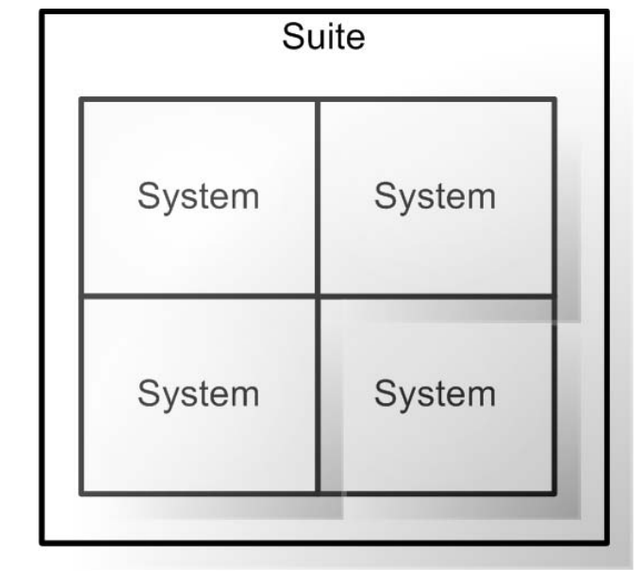


Figure 3: *Suite architecture*

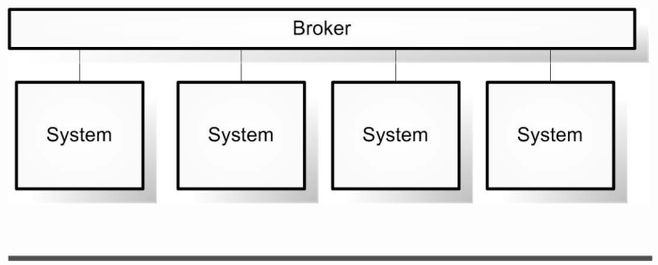


Figure 4: *Broker architecture*

cated hub, broker, or integration platform. Instead of connecting each system directly to all others, it is thus only connected to one other system, namely the broker. The broker provides basic message-passing services, but it is also specialized at translating data between various formats. Furthermore, the broker may implement intelligent routing, passing messages between various systems depending on various parameters, such as the contents of the message, the time of day, etc. There are many alternative broker-based solutions. The most influential trend in this domain is service-oriented architecture (SOA).

In reality, most organizations maintain a mix of the four approaches described above, cf. Figure 5. There are some old systems that are very sparsely interconnected to other systems. There are some systems that have direct connections between them. Certain, but not all, functional modules from an enterprise resource planning systems are typically implemented. Sometimes, there are two or more enterprise resource planning systems from different vendors. Finally, there is typically at least one, sometimes many, integration platforms, or brokers, to which some systems are connected.

1.3 A brief history of enterprise architecture

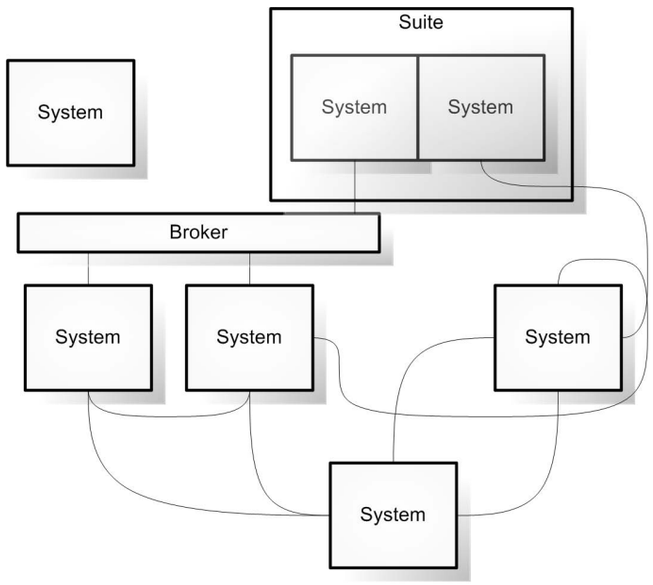


Figure 5: *Mixed architecture*

In the previous section, we considered the history and current state of information systems in general. Now, we focus on the concept of Enterprise Architecture (EA). Enterprise architecture, as we view it in this book, is an approach for managing the organization's information system portfolio and its relation and support to the business. At the base of the approach lies an architectural model incorporating concepts such as software components, connectors, functions, information, business processes, organizational units and actors. This section outlines the history of enterprise architecture by considering some of the most popular architecture frameworks in chronological order.

The Zachman framework

The history of enterprise architecture is generally considered to begin in 1987 with John Zachman's article *A Framework for Information Systems Architecture* [5]. Drawing analogies to the fields of classical architecture as well as to systems engineering, Zachman proposed a set of models for specifying information systems and their context.

Zachman thus claims that in order to manage a company's information systems, they need to be specified in the same way that e.g. an airplane or a building is. The current version (version 3.0) of The Zachman Framework for Enterprise Architecture (1) was released in 2011, cf. Figure 6.

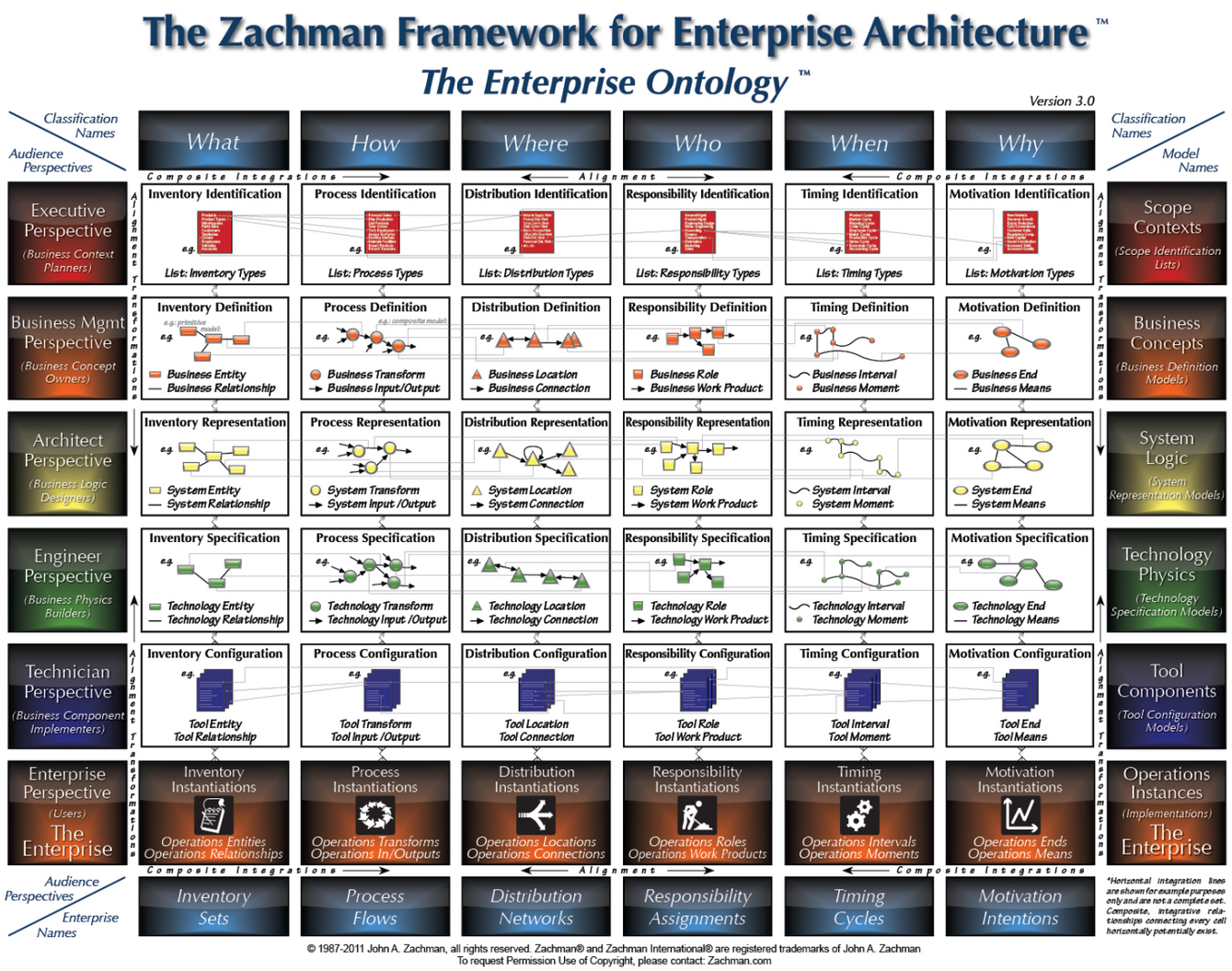


Figure 6: *The Zachman Framework 3.0*

The set of models proposed by Zachman are ordered on two axes. On the horizontal axis are sets of six aspects: inventory, process, distribution, responsibility, timing and motivation. These description types answer six fundamental questions: what, how, where, who, when and why. On the vertical axis are sets of perspectives, relating to the stakeholder posing the question. The first five rows include the executive, the business manager, the architect, the engineer, and the technician. The sixth row covers the final product of the enterprise system and is labeled the Enterprise. Altogether, the framework thus provides thirty-six cells from which an enterprise could be understood and described. The detailed syntax and semantics of the different models populating the cells are not given by Zachman, but are instead passed on to the user.

The Zachman framework does not provide concrete guidance for the process of enterprise architecting and it is not directed at any special kind of organization. The framework is still one of the most commonly referred to approaches to enterprise

architecture. The Zachman Institute is today active providing both courses and consulting services.

DoDAF

Within the United States Department of Defense, there is a fairly long tradition of enterprise architecture. The first version of an architectural framework was the Technical Architecture Framework for Information Management [6], published in the early 1990's. This initiative was followed by various successors, including the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) Architecture Framework [7], finally leading up to the currently supported version, namely the Department of Defense Architecture Framework (DoDAF) 2.0 [8].

The DoDAF describes fifty-two architecture products. Of these, two are of a summarizing nature. The remaining fifty products are divided into seven categories, or viewpoints: the capability viewpoint, the data and information viewpoint, the operational viewpoint, the project viewpoint, the service viewpoint, the standard viewpoint, and the system viewpoint. The DoDAF is arguably one of the most explicit frameworks with regards to different viewpoints and architecture products.

The main part of the DoDAF is focused on describing the architecture products. However, the DoDAF also contains some information regarding the use of these products by providing a fairly brief so called generic architecture description process consisting of six steps: 1) determine the intended use of the architecture; 2) determine the scope of the architecture; 3) determine the data required to support the architecture development; 4) collect, organize, correlate, and store the architecture data; 5) conduct analyses in support of the architecture objectives; and finally 6) present results in accordance with the decision-maker needs.

Because it is mandated by the Department of Defense, it is obvious that the US armed forces are the most diligent users of the DoDAF. However, the framework has been widely influential in the work of many other military forces, such as the NATO Architecture Framework [9] and Great Britains Ministry of Defense Architecture Framework [10].

TOGAF

The first version of The Open Group Architecture Framework (TOGAF) was presented in 1995. This framework was based on the TAFIM (also the origin of DoDAF, above), which was donated by the US government to The Open Group. The Open Group has since then published a number of improvements to the original framework; the current version is TOGAF 9.1 [11].

According to the TOGAF, enterprise architecture can be divided into four architecture domains or subsets, namely business architecture, data architecture, application architecture and technology architecture. TOGAF is designed to support all these subsets.

The core of TOGAF is the Architecture Development Method (ADM), cf. Figure 7. As the figure indicates, the process is constituted of nine steps. In the first step, organizational, administrative and scoping is-

sues are set up. In the second step, the purpose, or vision, of the architectural activities is articulated. In the third, fourth and fifth steps, the current and target business architecture, information systems architecture and technology architecture are modeled respectively. In the sixth and seventh step, a plan for migrating to the target architecture is created, and in the eighth step, the execution of this plan is supervised. In the final step, a process for managing changes to the architecture is set in place.

New in TOGAF 9 is the explicit presentation of a metamodel. This metamodel contains entities such as organization unit, actor, function, role, process, business service, data entity, application component, technology component and platform service.

Although the TOGAF has its origins in the Department of Defense, The Open Group has effectively eliminated all military specificities and when reading the documentation today the heritage is difficult to detect. The TOGAF is thus considered a general-domain framework, and is applied in all types of industries. The Open Group is a vendor- and technology-neutral consortium that continuously develops the framework.

1.4 Enterprise architecture as decision making support

Because enterprise architecture has a short history, there are many different views on what enterprise architecture really means, what it should mean, what problems it should address, and how. In this section, we argue for some propositions that this book is based upon.

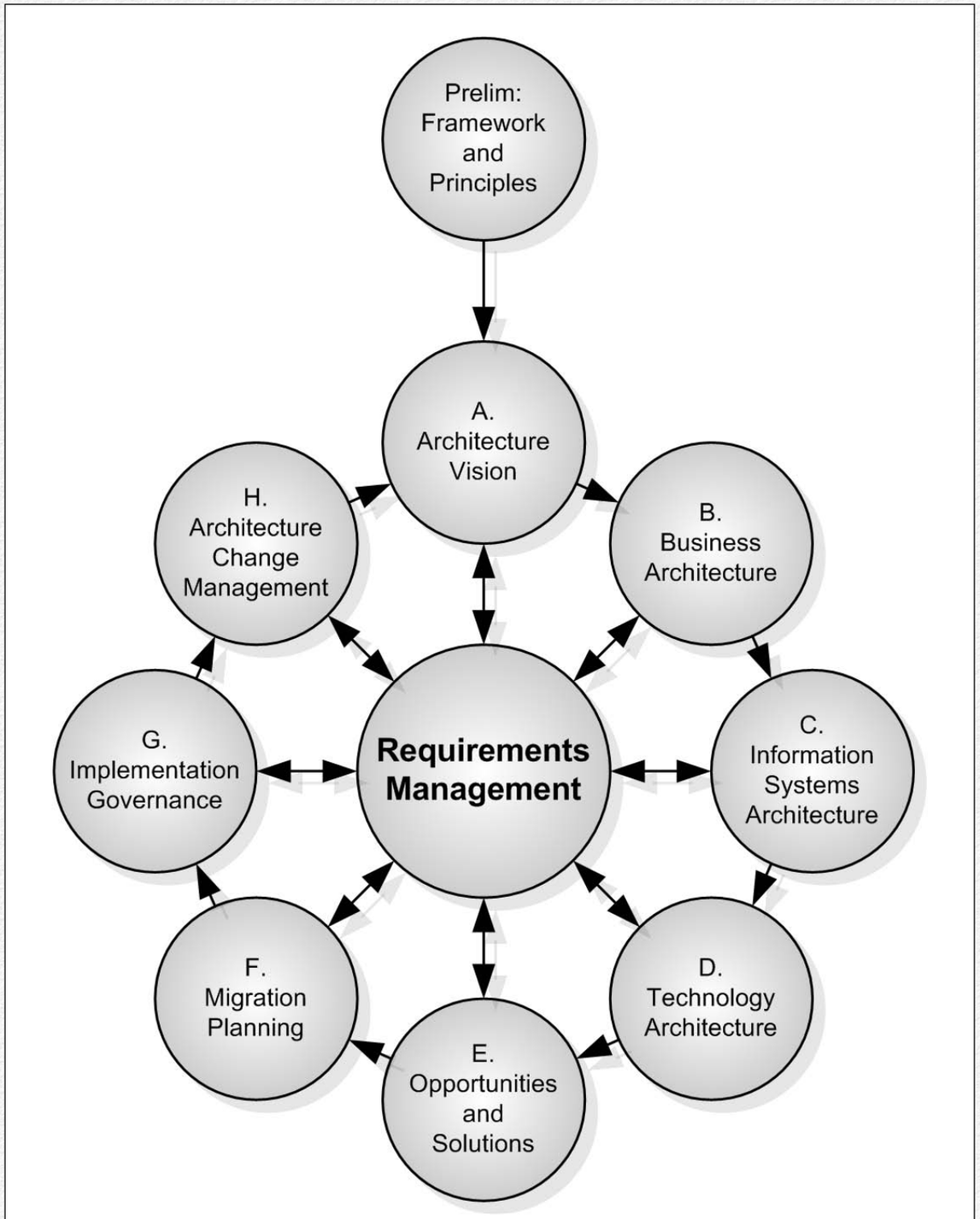


Figure 7: *The TOGAF Architecture Development Method*

1.4.1 Enterprise architecture analysis

In this book, enterprise architecture is mainly considered as a tool for making good decisions regarding the enterprise information systems. Decision-making can be viewed as a process of scenario selection. From the current state of the enterprise information system (to the left in Figure 8), various change decisions will result in new enterprise information systems. We call these potential new systems scenarios. Generally, IT decision makers have some ideas about how these scenarios could manifest themselves in the short and in the long term (to the right in Figure 8).

Enterprise architecture models can represent each future scenario, as well as the current state. Models over for instance applications, business processes, information, and technical infrastructure may all be employed for specifying these scenarios. The main problem in decision-making is to choose which one of the future scenarios to pursue; which one is the better one for a given purpose. This choice is aided by enterprise architecture analysis as illustrated in Figure 9.

In the ideal case, we would like to have a machine that takes the various potential scenarios as input and produces an output specifying which of the scenarios is best. We would of course select the best scenario, implementing the decision set required to reach that scenario. This machine is what we call enterprise architecture analysis. In enterprise architecture analysis research, the rules determining why one scenario is better than another are developed. An informal example of such a rule could be A system that is loosely coupled is better than one that is tightly coupled.

Movie 4: *On the Benefits of Enterprise Architecture*

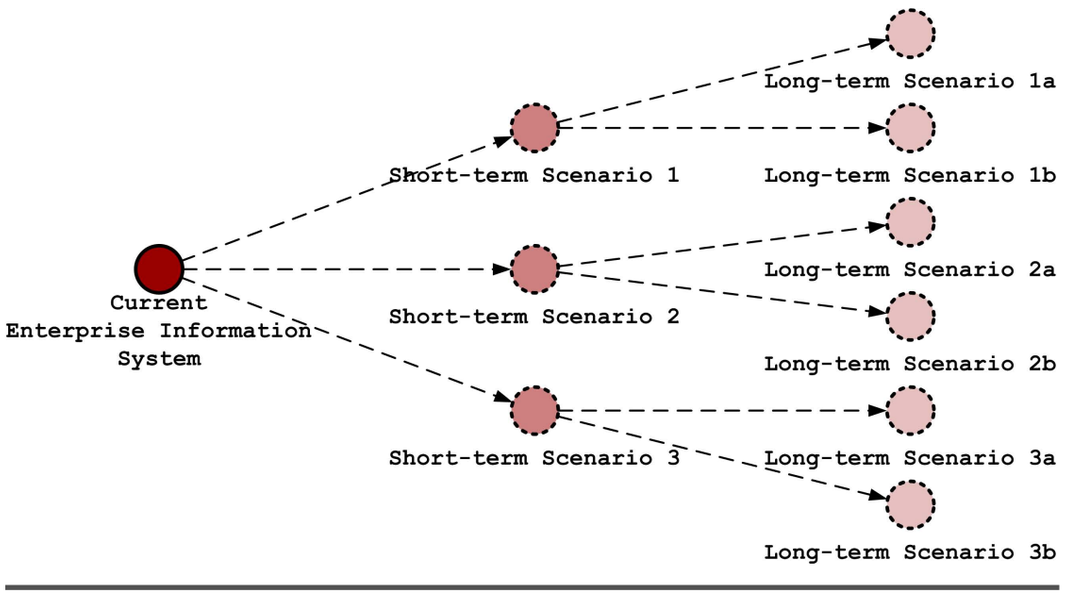


Figure 8: *Scenario decision making*

Model-based enterprise information systems management

The main difference between enterprise architecture and alternative approaches to enterprise information systems management is perhaps the focus on models of the systems and the context within which they reside. There are other approaches to enterprise information systems management that share many of the views of the enterprise architecture community, but no other approach places quite as large emphasis on modeling.

So, what is a model? Common models familiar to most people include geographical maps, architectural drawings, miniature buildings, airplanes, trains and cars. But there are also more abstract models, such as the Bohr model of the atom, where electrons orbit around the nucleus in the same way planets orbit the sun. Furthermore, there are

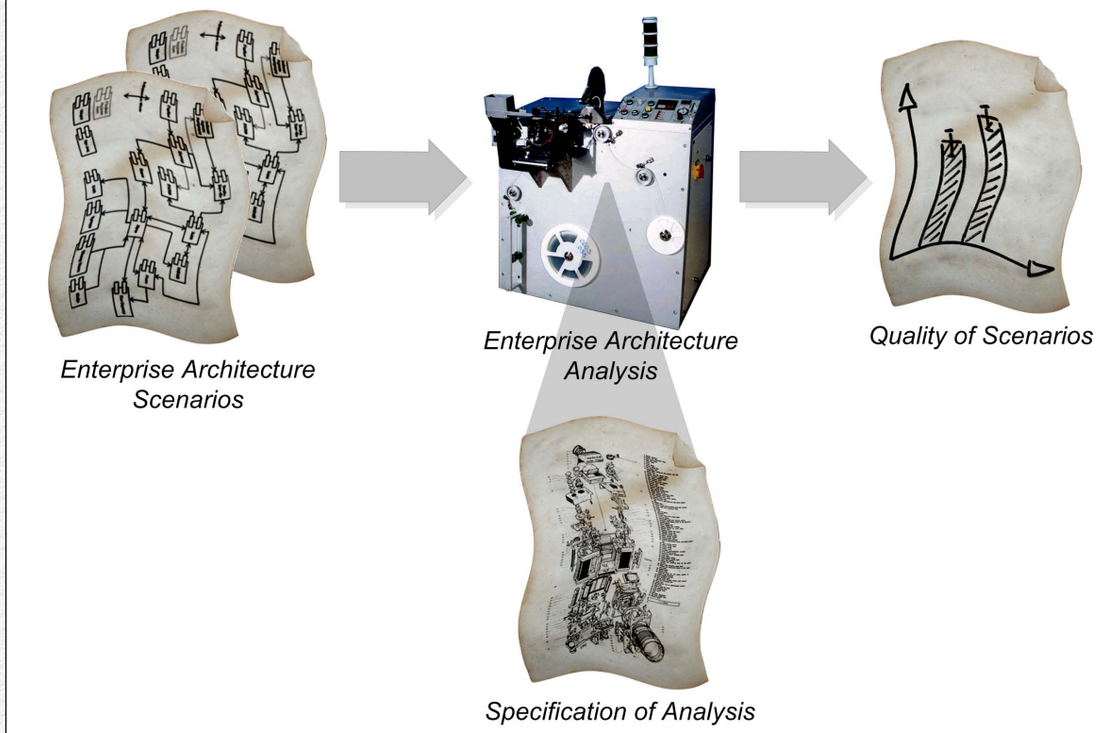


Figure 9: *The Enterprise architecture machine*

mental models, representing our cognitive appreciation of worldly phenomena. As indicated by the term enterprise architecture, we are mainly concerned with graphical models, i.e. drawings over how various things and phenomena are related. The analogy between enterprise architecture and the traditional architecture of buildings is in many ways appropriate.

Models are powerful tools for mainly two reasons: Firstly, they help us focus on the important issues when contemplating a certain problem. A common map depicting the different nation states of the world leads us to focus on questions like what the capital of this or that country is or

which country owns this or that island, while a road map leads us to questions about the driving distance between various locations. A weather map, of course, leads us to other questions. Secondly, models provide different people with a common view of an issue. Models both provide a common language that helps us communicate with each other and also guide us to focus on the same set of issues. Models are thus effective tools for planning, communicating, and of course, also for documenting (remembering). It is thus an important mission of enterprise architecture to provide useful models for the various decision-making activities of enterprise information systems planning.

The role of the CIO

There is a special relationship between the role of the Chief Information Officer (CIO) and the discipline of Enterprise Architecture. An individual or a small group of people close to the senior management of the company typically holds the CIO role. The CIO role definition differs with the company, but generally it is a position with an overarching responsibility for all of the enterprise's IT. Sometimes the CIO is the head of the IT department. Typically, the primary focus of the CIO is strategic information systems planning. Common work products consequently include vision documents for information systems, IT strategies, and IT plans. In brief, the CIO is typically responsible for overarching enterprise information systems management.

Because enterprise architecture is a tool for enterprise information systems management and the CIO is the role with the main responsibility for this activity, it is reasonable to view the CIO as the main stakeholder of enterprise architecture. Therefore, in this book, if no other stakeholder is indicated, the CIO will be assumed. Naturally, this does not mean that the CIO is the only role with an interest in enterprise architecture, on the contrary, all roles involved in enterprise information systems management should have some involvement in the enterprise architecture. But of these various roles, the CIO is in most companies the main stakeholder.

Goal-driven enterprise architecture

Enterprise architecture advocates the explicit modeling of the organization and its systems. Modeling, however, can be costly. The world is full of things that could be represented, and it would not be difficult to spend completely unreasonable efforts on modeling the details of existing and future systems. Such indiscriminate modeling would be of little value, not only due to the effort of producing the models, but also because the models would soon be as difficult to understand as the real world they represent.

In order to avoid indiscriminate modeling, we advocate for a goal-driven approach. In brief, only those phenomena that directly relate to our enterprise architecture goals are to be modeled. In other words, only the information required for answering our most pertinent questions will be gathered in the enterprise architecture models. This seems almost self-evident, but it is not easily accomplished. There are two main problems. Firstly, many organizations are not clear on what their goals are. For instance, is information security more important than usability in this information system? Or is increased business process efficiency the most important metric to strive for in this business domain?

Secondly, the relations between the enterprise information system management goals and the enterprise architecture models are unclear. If it is important to manage the availability of the information systems, what enterprise architecture models should we maintain? This book addresses both of these issues.

Decision-making

All attempts at rational decision-making need to contain a few activities [12]. Firstly, the decision-maker must settle on a goal, or a success criterion. What would characterize a good decision as opposed to a poor one? For instance, when buying a car, one goal might be to buy as safe a car as possible. Actually, as a rule there are multiple goals that need to be traded against one another.

Secondly, decision alternatives need to be identified. What options are available? Let us suppose that our hypothetical car buyer only has a choice between a 1963 Corvette Sting Ray and a 2005 Saab 9-5.

Thirdly, the effects of the decisions on the goals must be elicited. For instance, selecting a car that features seatbelts, airbags and good brakes, is more likely to lead to the goal than selecting one without these features. It is also preferable to select a car with good crumple zones, i.e. structural features designed to compress during an accident in order to absorb the impact energy.

Fourthly, based on the above analysis of the effects of the alternatives on the goal, the decision-maker needs to decide on what information to collect with respect to the different decision alternatives. In this case, the decision maker might write down a small checklist over questions to ask the dealer: Does this model have seat belts? etc.

Fifthly, the information needs to be collected. A problem often encountered here is that the gathered information is not completely certain. Perhaps the car dealer does not know the answer to some questions. It is then normally possible to find the answer elsewhere. For many questions, however, it is very expensive to get certain answers. To be really sure that the crumple zones are properly designed, for instance, it may be necessary to conduct crash tests. Such information gathering would be more expensive than the planned investment.

Sixthly, when the chosen information is gathered, it needs to be consolidated into an aggregated assessment; which decision alternative is to be preferred? For instance, the buyer might need to determine whether it is safer to drive a car with seat belts and airbags but poor brakes than a car with good brakes, seat belt but no airbag. Also, the decision maker needs to estimate whether the credibility of the assessment is acceptable. Is it okay to make the decision without knowing the answer to the question regarding crumple zones, for instance?

Finally, the decision needs to be made and acted upon. The buyer needs to go to the car dealer, pay the money and get the car. There might also be a need for monitoring that the decision is really implemented as decided. Perhaps the car needs some adjustments before delivery. The buyer would then be satisfied with the decision process only when the car was delivered.

The same decision making process is valid and inevitable for sound enterprise information systems management.

Setting the goals

In the context of enterprise information systems management, we consider information system goals (as opposed to the simple car buyer example above).

Because it often is hard to measure the effects of information system decisions on the enterprises ultimate goals (business goals such as maximizing the profit), it may be necessary to use intermediary goals associated with the information systems themselves. For instance, it may be sufficient to understand the difference between two systems with respect to their availability, performance and interoperability in order to choose between them. In this case, the effects of the systems on the organizations ultimate goals are even less understood in detail, but we are convinced that these information system properties do have a positive effect on these goals and might be content with that. The benefit of this approach is that we believe the assessments of availability, performance and interoperability to be much more certain than those of business profit. These information system goals are the focus of this book.

Specifying the decision alternatives

If one end of decision-making is the goal, then the other is the available set of decision alternatives.

Oftentimes, some decision alternatives are clear, while others are fuzzier. For instance, consider the case of the chief architect considering the technological direction of the information system architecture. An alternative that may be reasonably clear is the migration to a service-oriented architecture. Another alternative may be to aim for suite solutions, i.e. to try to reduce the number of vendors as much as possible, letting one or a few deliver all required systems. In addition to these two alternatives, there are surely others, but which are they? In short, it is important but oftentimes difficult for the decision maker to understand what the comprehensive set of options is.

Breaking down the goals

One important aspect of the above mentioned decisions is that they have causal effects on the goals. The chief architects decision to move the system architecture towards a service-oriented platform has for instance effects on the modifiability and performance of systems, which in turn affects the flexibility of the business processes supported by the systems, and in the end this (hopefully) has an effect on the ultimate organizational goals.

A second important aspect is that the causal relations from decision to goals are oftentimes complicated. It would have been a simpler world if it were possible to directly change the goals, so that the chief architect directly could make the decision to increase the flexibility of the business processes, instead of having to go via the information system architecture. Unfortunately, in this world, it is rare that the phenomena that we can manipulate are identical to the objectives we seek.

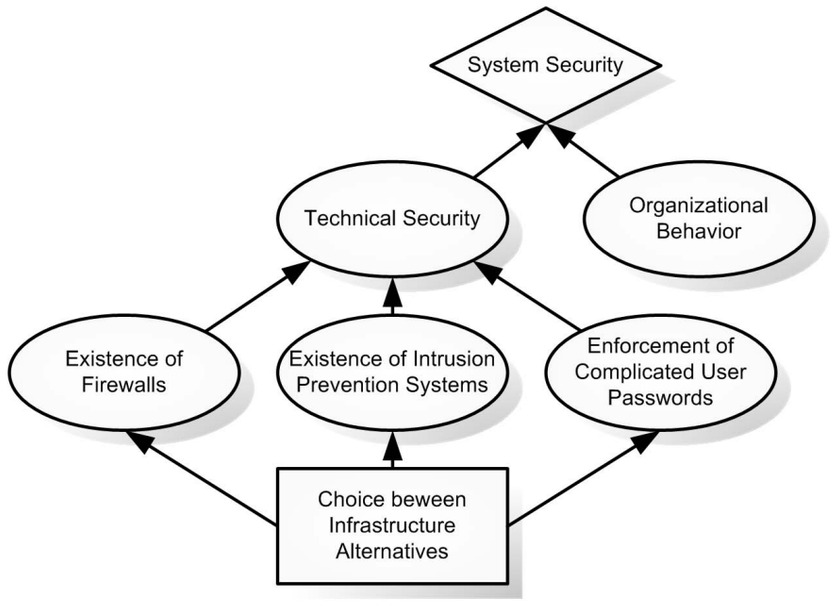


Figure 10: *A simple decision making theory on security*

In order to mitigate this problem, decision makers need to try to understand the chains of causality. This is no simple task, and we may safely claim that relatively few of the causal relations between IT decisions and IT goals are certain in any scientific sense of that word. However, incomplete knowledge is generally better than no knowledge. Figure 10 presents one hypothetic causal theory relating the decision between two infrastructure alternatives to the goal of information system security.

In this theory, it is believed that the most important determinants of information system security are the organizational behavior on the one hand and the technical security on the other. Since the decision only concerns technical issues, the theory needs not be explicit with regard to the organizational issues. Technical security, however, is believed to be dependent on the existence of firewalls, the existence of intrusion prevention systems and the technical enforcement of complicated user passwords. (We grant that this is a simplistic theory over security.) If the infrastructure alternatives under consideration differ with respect to these features, the decision maker may use the theory to predict the causal impact of the decision on the goal.

Eliciting information requirements

If the decision maker has settled on goals, is aware of at least a couple of decision alternatives, and has some understanding of the causal effects of the decision on the goals, it is time to start examining the alternatives in greater detail. Do all infrastructure alternatives include firewalls, or only some of them?

It is oftentimes convenient if the information required for the decision is collected and presented in a consistent manner to the decision maker. Since information gathering can be very time consuming, it would also be good if other people than the decision makers themselves could perform this activity. In order to allow such third-party data collection, one needs to clarify exactly what information is desired.

The importance of this activity is oftentimes overlooked. IT decision makers often commission various investigations aimed at providing decision-supporting information. Unfortunately, the information presented in the resulting reports is oftentimes not at all compatible with the goal breakdown of the decision-maker. The decisions taken are then based on unnecessarily incomplete information.

Collecting evidence

When it has been decided on what information to gather, the complicated task of data collection needs to be embarked upon. This task is very much like a scientific or a criminal investigation. The major problem is to collect evidence that is sufficiently credible for the given purposes. Many questions in enterprise information systems management are very difficult to find trustworthy answers to.

For instance, assume that we are interested in assessing the availability of some part of the enterprise information system in a large enterprise. One might then be interested in finding out whether two existing systems are exchanging information or not. For the investigator in this hypothetical example, the first step might be to refer to a previous survey over the company's information systems and their relations. Perhaps that study indeed does contain the required information, but it is five years old. It is then uncertain if this information is still correct. In order to improve the credibility of the information, the investigator might make a phone call to the person listed as system owner. Perhaps this person responds that yes, she believes that the systems are exchanging information, because the system administrator said so a while back. The investigator may then call the system administrator, but of course, that person may also be in error. In order to really get to the bottom of the question, our investigator then decides to read the system documentation. However, such information is often outdated. How then, should the investigator proceed to really ensure that the collected information is correct? Perhaps it would be possible to install an application to the local area network that attempts to identify communication between the two systems by checking the origin and target addresses of IP-packets on the network. This might fail if the communication frequency between the systems is lower than the test period. Another alternative would be to read through the source code of the systems to find any code that transmits messages between them. However, the source code might not be available. And so on.

The main point here is that data collection is a difficult endeavor, and it is rarely the case that the collected information is completely credible. On the contrary, it is often the case that the gathered information is associated with a degree of uncertainty.

Assessing goal fulfillment

When the required pieces of information have been gathered, they need to be consolidated into a comprehensive judgment. When we know that decision alternative A features solidly configured firewalls and intrusion prevention systems while decision alternative B incorporates encrypted communication and biometric authentication, we need to determine which is the better alternative. Of course, this goes back to the goal breakdown of Subsection 1.4.8. A good goal breakdown includes the relative importance of the different causal factors, so that it is possible to determine which of the alternatives is better. It is important here to note that all (non-random) decision makers have such weighted broken-down goal structures. It is just that these models are normally not made explicit. They remain hidden in the heads of the decision makers.

Furthermore, in the aggregation of the various gathered pieces of information into a single judgment, it is important to also attempt to get a feeling for the credibility of that judgment. If the information base is uncertain, the judgment will often also be dubious. If the decision maker feels that this uncertainty is excessive, she may choose to return to the information gathering activity, attempting to collect better information.

Decision-making, implementation and monitoring

A decision-maker supported by a credible analysis of the effects of decisions on goals is in a good position for making a decision. The only thing to do is to choose the option that has the best effect on the goals.

Of course, decision making in the real world is rarely quite so rational. It is oftentimes the case that there is more than one decision maker, so that activities such as lobbying and negotiation become a large part of the decision making process. However, such lobbying and negotiation may concern whether the correct goal has been selected, whether the identified decision alternatives are reasonable, whether the causal theory linking the decisions to the goals are really correct, and whether the collected information is credible or not.

In organizations, decision makers are oftentimes not the executors. There are various means by which decisions may be implemented. Policies and directives may be changed in order to affect the behavior of relevant organizational units. Specific projects may be initiated, aiming at coming about the required changes.

In whatever manner the decision maker attempts to realize the decision, there is the risk that the implementation does not succeed. Therefore, monitoring activities, measuring whether goals are really reached, are commonly employed instruments. Monitoring is very similar to the process described in this section. It is necessary to determine what (goal) to monitor, to break it down into measurable indicators, to collect that information, and to aggregate it in order to assess whether the goal was actually met.

1.4.2 EA modeling and analysis, the continuation of this book

This first chapter of the book presented the history of information systems and enterprise architecture, the enterprise systems in use today and their most common integration architectures, as well as the background of enterprise architecture as a decision making approach. The rest of this book focuses on presenting enterprise architecture models and tools supporting the decision making process.

Basic enterprise architecture

2

An introduction to Enterprise Architecture modeling.

This second chapter of the book focuses on presenting what enterprise architecture modeling is and how to do it. First, we introduce some basic modeling theory e.g. what is a metamodel and a model. Then, we provide a simple modeling tutorial.

2.1 Modeling theory

Enterprise architecture is a model-based approach to business and IT management. Just as any other models, enterprise architecture models are abstractions and simplifications of the real world. The use of models is pervasive; from for instance geographical maps and calendars that we use in our daily lives to advanced models of building constructions and the behavior of atoms and molecules that are used under very specific circumstances. The essence of modeling is to capture interesting phenomena in the real world, be it roads or electrons, and leave out everything else.

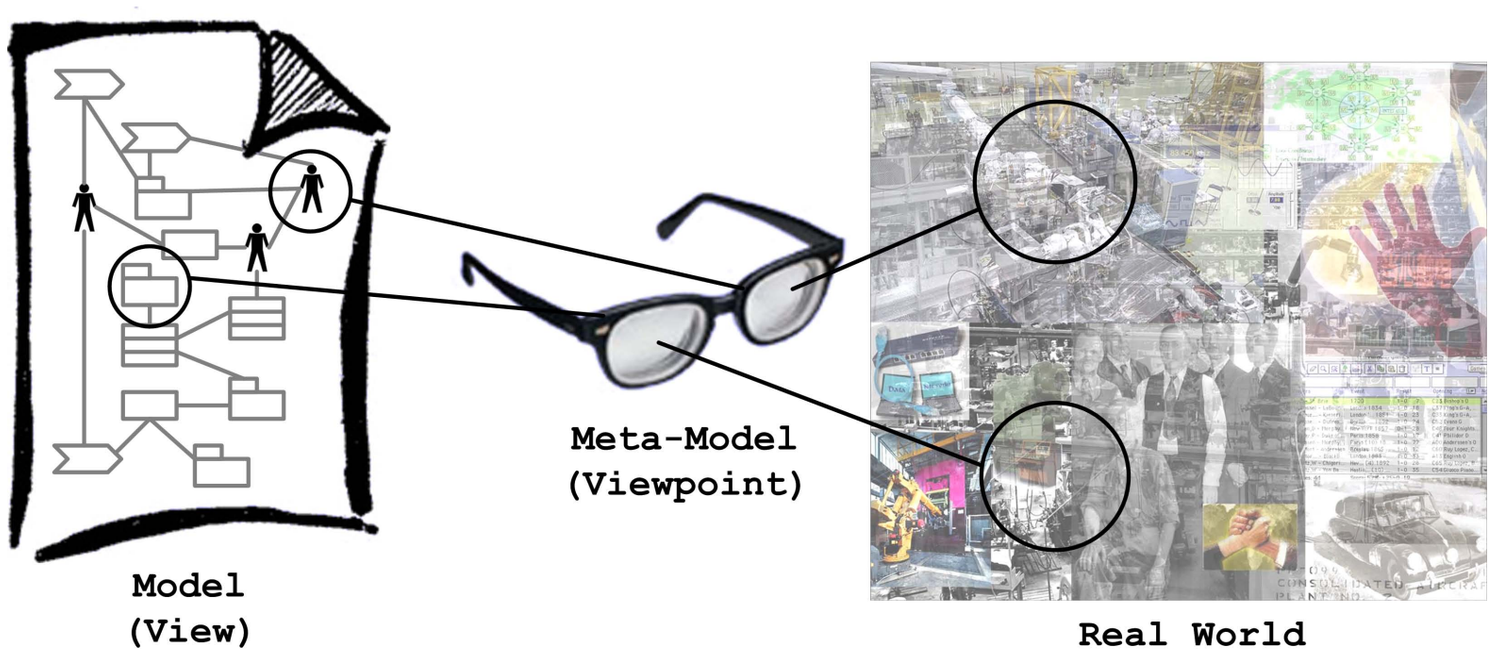


Figure 11: *Conceptual picture of modeling*

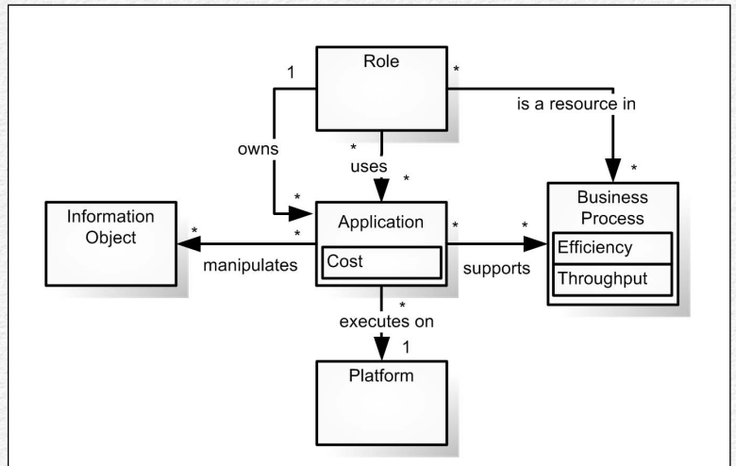


Figure 12: *A simplistic metamodel example.*

The choice of what to filter out from the real world into the model is determined by a metamodel, or a modeling language. In other words, the metamodel is the language in which we describe the phenomena of the real world, cf. Figure 11. In the discipline of enterprise architecture, we are interested not in roads or electrons, but in things like business processes, organizational roles, information systems, communication networks, how they behave, as well as how they relate to each other. A simple metamodel for enterprise architecture could look like the one depicted in Figure 12. The metamodel describes what classes should be modeled

and what relations between them that are of interest. The relations may also provide information about the multiplicity restricting the relation; a role may for instance own many applications, but the application may only have one owner. Moreover, a class may be described by a number of attributes; an application may have a certain cost and a business process a level of efficiency. The typical metamodel thus consist of classes, class relations and attributes.

There are many different metamodels proposed for enterprise architecture, including the already mentioned metamodel proposed by The Open Group in TOGAF [11]. Also, most EA tools have a built in metamodel. What metamodel to use is up to each company. We suggest that the choice of metamodel should be based on the goals the company has with its EA initiative.

2.1.1 A modeling example

This subsection contains a small modeling example, with focus on describing how a metamodel and a model are related. Thus, the presented metamodel and model are only examples. We have also chosen not to include any attributes here, only classes and relationships. The relation to certain information system goals, such as interoperability and modifiability is only touched upon and will be detailed later on.

Considering the IT-centered enterprise architecture models, perhaps the most common are the ones aiming at describing application (or system) cooperation. In its basic form such a model would describe what applications that are connected to each other, typically in terms of some form of information exchange through interfaces indicating what kind of services are provided by the applications. It could also contain information regarding what applications are grouped together in larger assemblies. A metamodel defining this is found in Figure 13.

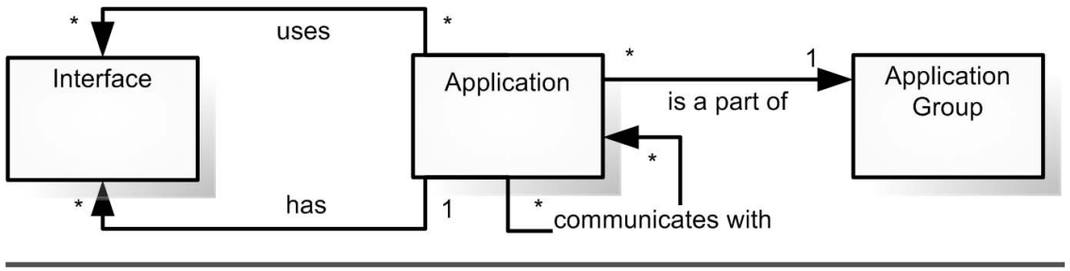


Figure 13: *Metamodel example focusing on application cooperation.*

An example of an instantiated application cooperation model is found in Figure 14. The model illustrates the relationship between the automatic meter reading, billing, and customer support applications.

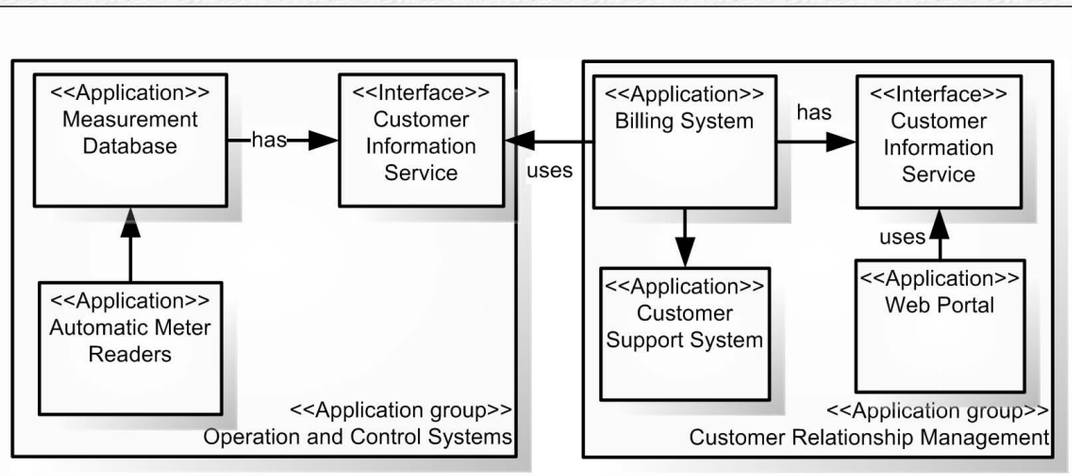


Figure 14: *Example application cooperation model.*

Notice the relation between the metamodel (Figure 13) and the model (Figure 14). The model only contains instantiated classes and relationships that are available in the metamodel. That is, the metamodel sets the constraints on what is possible to instantiate in a model. If an architect realizes that there is another type of class necessary in the model in order for it to provide decision support for the goals under consideration, then the metamodel must be modified first. For instance, the CIO might have requested that there is a need for information about application cooperation and application usage. The architect realizes that in order for the model to provide this information it needs to contain what processes there are and how they relate to the applications. Thus, the metamodel needs to be altered for this change, e.g. by adding a process class and process-application relationships.

Even this type of small metamodel can aid in decision-making if used in the right way. For instance, in relation to information system interoperability, the application cooperation model describes what applications that actually are connected, and through what interfaces. Related to information system modifiability, the application collaboration model can provide information about modifiability-affecting factors such as the external coupling of applications. Information about the source code, such as its size or complexity, could potentially also be included as application attributes in this type of model. More on this will be presented in chapter 3.

2.1.2 A commonly used metamodel

As explained earlier there are many initiatives presenting different types of metamodels, some more similar to each other than others. Our standpoint is that the choice of metamodel should be based on the goals one has. However, in order to show what a typical metamodel could contain when it comes to classes and relationships we here briefly introduce you to one of the more common ones.

Dr. Marc Lankhorst and his colleagues have proposed what is one of the most well-known and widespread metamodels, called ArchiMate [13]. ArchiMate is an open, independent, and general modeling language for enterprise architecture. The Open Group accepted the ArchiMate metamodel as a technical standard in 2009. The metamodel consists of three layers; the Business layer, the Application layer and the Technology layer, where the technology supports the applications, which in turn support the business. Each layer consists of a number of classes and defined class relationships. The classes in each layer are categorized into three aspects of enterprise architecture: 1) The passive structure - modeling informational objects. 2) The behavioral structure - modeling the dynamic events of an enterprise. 3) The active structure – modeling the components in the architecture that perform the behavioral aspects. Figure 15 presents the ArchiMate metamodel.

Below, the ArchiMate classes and relationships are described in detail in [13].

2.2 Modeling tutorial

This section will guide you through a modeling tutorial using the Enterprise Architecture Analysis Tool (EAAT). There are altogether two EAAT tools - an EAAT Class Modeler and an EAAT Object Modeler. See

chapter 11 for more information on class modeling and chapter 12 for modeling patterns and practices.

In this tutorial, we will introduce the classes and relationships of a metamodel capable of analyzing several different quality attributes, including service availability, modifiability, and cost. In this first subsection, the attributes are not explained, these will be presented later in the book (cf. chapter 3). The metamodel used here is partially based on ArchiMate and contains classes from the business layer, the application layer, and the infrastructure layer.

Application layer

An *application service* is defined as a unit of functionality that a system exposes to its environment and that displays automated behavior. An application service could be playing a game of chess with a user, or printing a shopping list or calculating a ballistic trajectory. The important thing is that the service provides something meaningful to the user.

An application service is exposed to the environment, but it is realized by an *application function*. For the chess game, an application function could be calculating the next move. To calculate the next move is not in itself useful to the chess player, but it is a function that will contribute to something useful, that is to an application service.

An *application component* is a modular, deployable, and replaceable part of a software system. It is the entity that performs application functions. It could for instance be the Chess software that plays the chess game with the user.

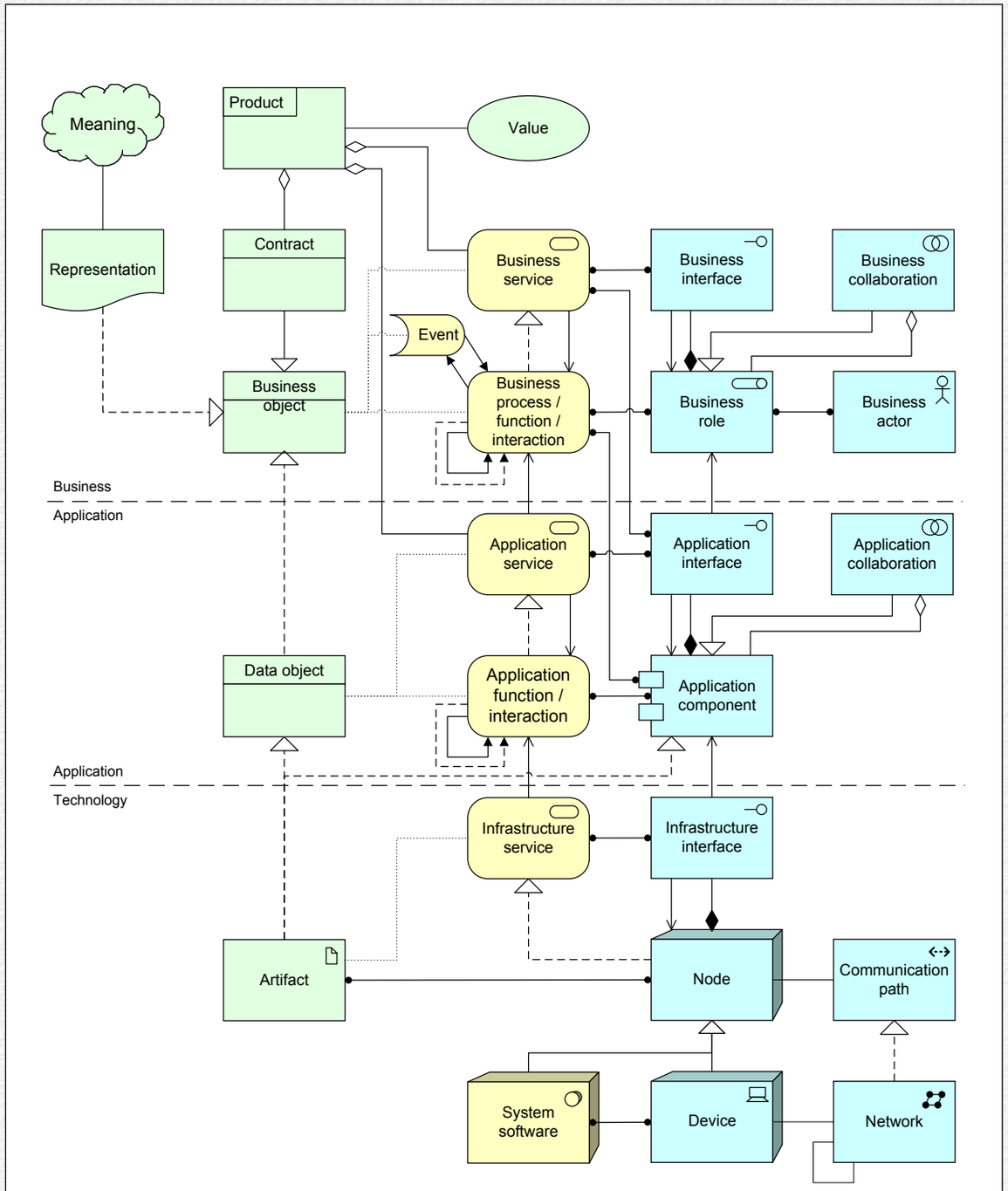


Figure 15: *The ArchiMate metamodel.*

An *application collaboration* is used in order to model what application components that are acting together to perform a collective behavior. It could for instance be the collaboration between the chess software and a web component, in order to find other players to compete with or to be able to post your game results online.

Business layer

Just as the chess player uses the chess application, a designer in the automotive industry may use a design application to sketch on a car.

The *business process* is the Designing of the vehicle, and processes may use application services, so here is an important link between the business and the IT. Other processes in a car company could be producing the vehicle and shipping the vehicle.

Business processes are performed by roles. In this case, the designer is a role. Other roles could be Chief Production Officer and Machine Operator.

The customer, however, sees nothing of the design, production and shipping of a car. The customers interact with the company through *business services*. An example of a business service is Car selling.

Infrastructure layer

Moving to the bottom part of the metamodel, the application layer is mirrored by an underlying infrastructure layer. In the same manner as business processes may use application services, application functions may use *infrastructure services*. Examples of infrastructure services are

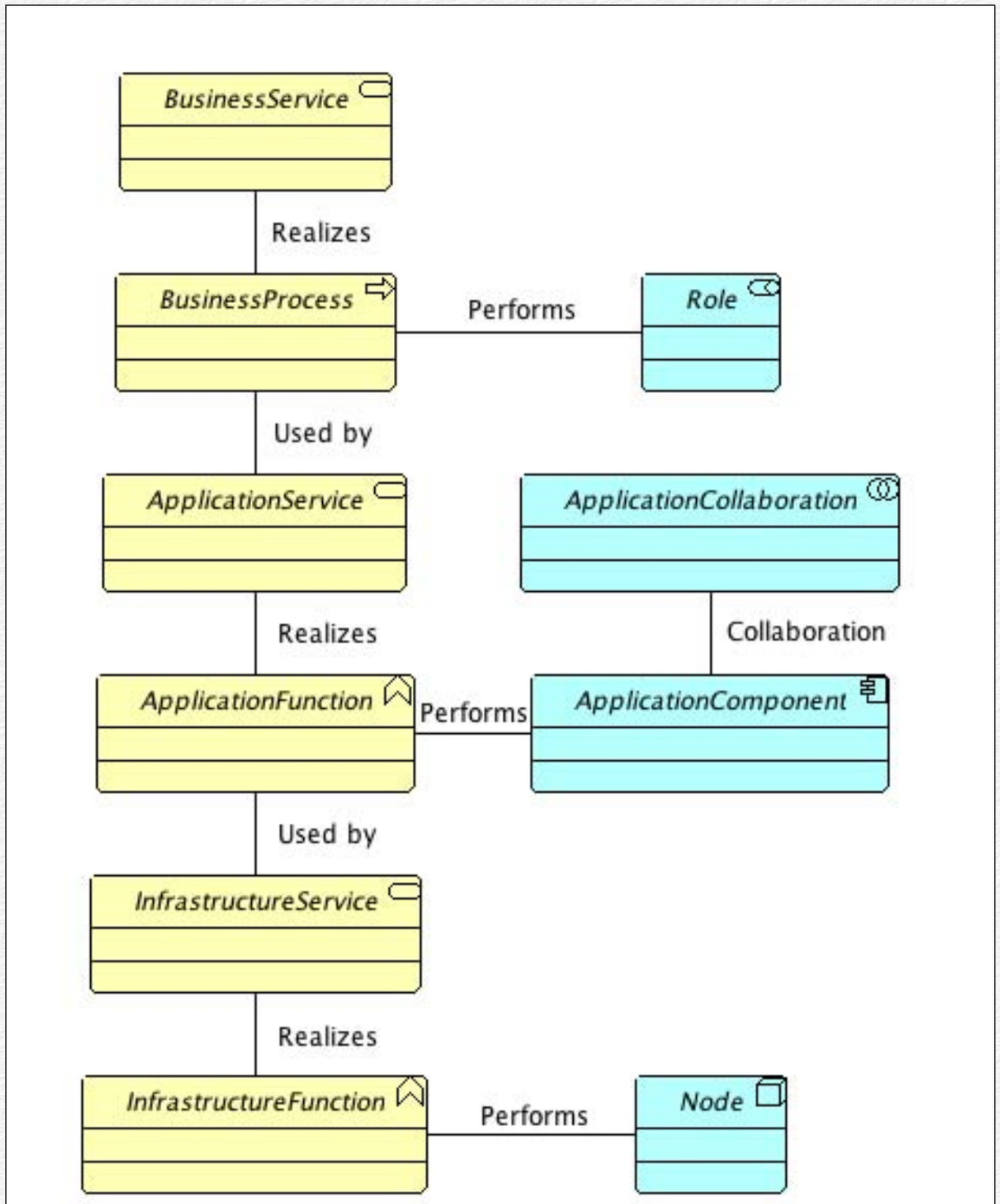


Figure 16: *Tutorial metamodel example.*

Messaging, Data management and Printing. Just as in the case of applications, infrastructure services are realized by infrastructure functions. These are in turn performed by nodes. A node could be an IBM Mainframe running z/OS, or a PC running Microsoft Windows.

In all, the metamodel is composed of ten classes, cf. Figure 16, and this metamodel will allow us to analyze system qualities such as service availability, modifiability, and cost.

The tutorial is composed of eight steps that will guide you through an enterprise architecture modeling and analysis example. The background is a case at a fictive energy company called ACME, which is about to start a project to improve the business process of analyzing automatic meter reading data. The architect at ACME has a proposal for the CIO that includes new applications and infrastructure supporting the process. The scenario also includes the hiring of a new person in charge of the meter data analysis. The CIO has expressed that the most important aspects to consider, besides getting the right functionality, is to have high availability and that the solution is easy to change in the future in case new requirements occur. Also the company has a tight budget for these kinds of investments, thus the total cost is also a major factor in this case.

Outline

1. Add a first infrastructure layer with three classes and evaluate the availability
2. Add a second infrastructure layer with three additional classes and evaluate its availability
3. Add an application function, connect it to the two infrastructure services, and evaluate its availability
4. Add an application component, connect it to the application function, and see how it affects the availability
5. Add an application collaboration with coupling evidence
6. Add an application service and evaluate the modifiability
7. Add a business layer and evaluate the business process availability
8. Add cost evidence to the model and evaluate the total business process cost

The following paragraphs will detail the modeling steps.

Step one: Add a first infrastructure layer

- Add an Infrastructure Function
- Name it Local meter reading
- Press the calculate button
- Consider the Availability results (the availability of the Local meter reading function should be a normal distribution centered around 0.95)
- Add a Node

- Name it Local meter readers
- Connect the two classes
- Add 0.992 (99,2 %) as evidence to the attribute Availability of the Local meter readers node
- Press the calculate button
- Consider the Availability results (the availability of the Local meter reading function should now be 0.992 instead)
- Add an Infrastructure Service and name it Aggregated meter readings
- Connect the Aggregated meter reading service and the Local meter reading function. The Aggregated meter readings service is realized by the Local meter reading function.
- Press calculate again and see how the availability of the node, that first had an effect on the function, now also has an effect on the service.
- Save the model regularly.

Step two: Add a second infrastructure layer

- Add a second Node and name it Database, add a second Infrastructure function and name it Data transmission, and add a second Infrastructure service and name it Data transfer.
- Connect the Node and the Function.
- Connect the function and the service with a RealizeAND association so the Data transfer service is Realized by the Data transmission function.
- Add 0.991 (99,1 %) as evidence to the attribute Availability of the Database node.
- Press the calculate button and consider the Availability results (the availability of the Data transfer service should be 0.991)
- Save the model regularly.

Step three: Calculate the availability of an application function

- Add an Application Function and name it Data collection

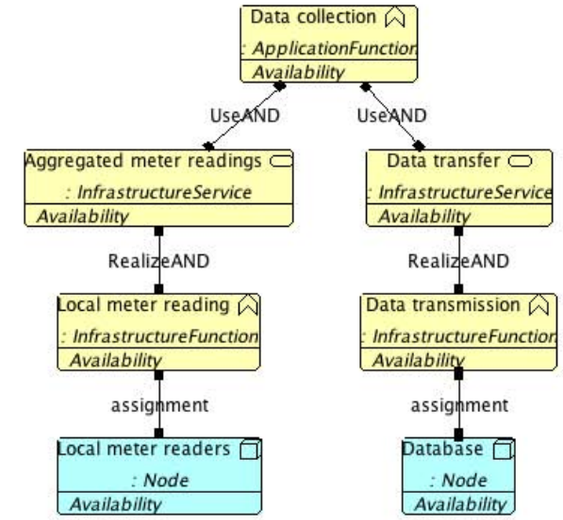


Figure 17: *The application function availability viewpoint of the tutorial model example.*

- Connect the Data collection function with the Aggregated meter readings and the Data transfer services, since the Data collection function uses both services the relationship should be of type AND. The relationships should be DataCollection.useAND.LocalMeterReading and DataCollection.useAND.DataTransmission.
- Press the calculate button and consider the Availability of the Data collection function, which should be 0.983 (98,3 %)
- See the application function availability viewpoint of the model in Figure 17.
- Save the model regularly.

Step four: Add an application component

- Add an Application Component and name it AMR master
- Connect the AMR master with the Data collection function
- Press calculate and consider the Availability result of the Data collection, the result should now read 94,9 %
- Add 0.995 as evidence to the attribute Availability of the AMR master component
- Press calculate and consider the Availability result of the Data collection function again, the result should now read 97,8 %
- Save the model regularly.

Step five: Add application collaboration

- Add a second Application component and name it Business intelligence.
- Add an Application collaboration class and name it Data exchange.
- Connect the two Application components through the Application collaboration class.
- Add a second Application function and name it Data compiler.
- Connect the AMR master and the Business intelligence components with the Data compiler function.
- Add Size 430000 and Gearing factor 53 to the AMR master, as well as Size 75000 and Gearing factor 55 to the Business intelligence component.
- Press calculate and consider the the External couplings attributes of the two components, all four should now be 1 (since we have one collaboration of unknown kind).
- Add 1 Content coupling, 2 Common couplings, and 11 Data couplings to the Data exchange collaboration between the two components.
- Press calculate again and consider the External coupling attributes of the two components, all four should new be 5,9 instead.
- Save the model regularly.

Step six: Calculate application modifiability

- Add an Application service and name it Compiled meter data.
- Connect the Compiled meter data service with the two Application functions, since the service is Realized by both functions the relationship should be of type AND.
- Add a baseline Gearing factor of 53 to the Compiled meter data service.
- Press calculate and consider the Modifiability of the service, which should be 3. This is a rather low value indicating that the service will be difficult/costly to change in the future, mainly due to large source code and the tight coupling between the components realizing the service.
- See the application service modi_ability viewpoint of the model in Figure 18.
- Save the model regularly.

Step seven: Add a business layer

- Add a Business process and name it Analyze meter data.
- The Analyze meter data process Uses the Compiled meter data service. Add this relationship (since there is only one user, either userAND or UserOR) can be selected.
- Add a Role and name it Meter data analyzer.
- Connect the Meter data analyzer role with the Analyze meter data process.
- Add 99 % Availability as evidence for the Role and 99,7 % Availability for the Business intelligence component.
- Press calculate and consider the Availability for the Business process Analyze meter data, which should be 97 %. This is a rather low value for a business process and it is mainly due to the high dependence of the manual work of the Meter data analyzer role.
- Save the model regularly.

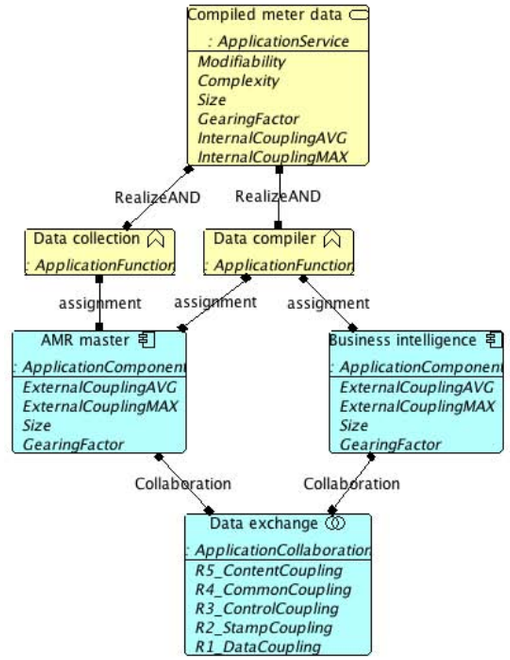


Figure 18: *The application service modifiability viewpoint of the tutorial model example.*

Step eight: Calculate the business process cost

- Add the Initial and Yearly costs of the Nodes, Components, and Role as described in Table 1.

Class	Type	Initial Cost	Yearly Cost
Local meter readers	Node	1000000	100000
Database	Node	50000	10000
AMR master	App.Comp.	300000	45000
Business intelligence	App.Comp.	450000	25000
Meter data analyzer	Role	500000	1000000

Table 1: *Initial and yearly cost data for modeling tutorial.*

- Press calculate and consider the cost of the business process Analyze meter data, which should be 3.480.000 SEK. This cost is based on the assumption that the role, components, and nodes are new. Once these have been in place for some time the initial costs will be written-off and the total cost of the service will only be based on the yearly costs (this has not been implemented in the metamodel yet). Also, in this small example architecture the business process does not share any components, nodes, or the role with others. If this would have been the case the costs would also have been shared, thus a smaller economic burden would have fallen on this particular service.
- Save the model regularly.
- The complete model can now be seen in Figure 19.

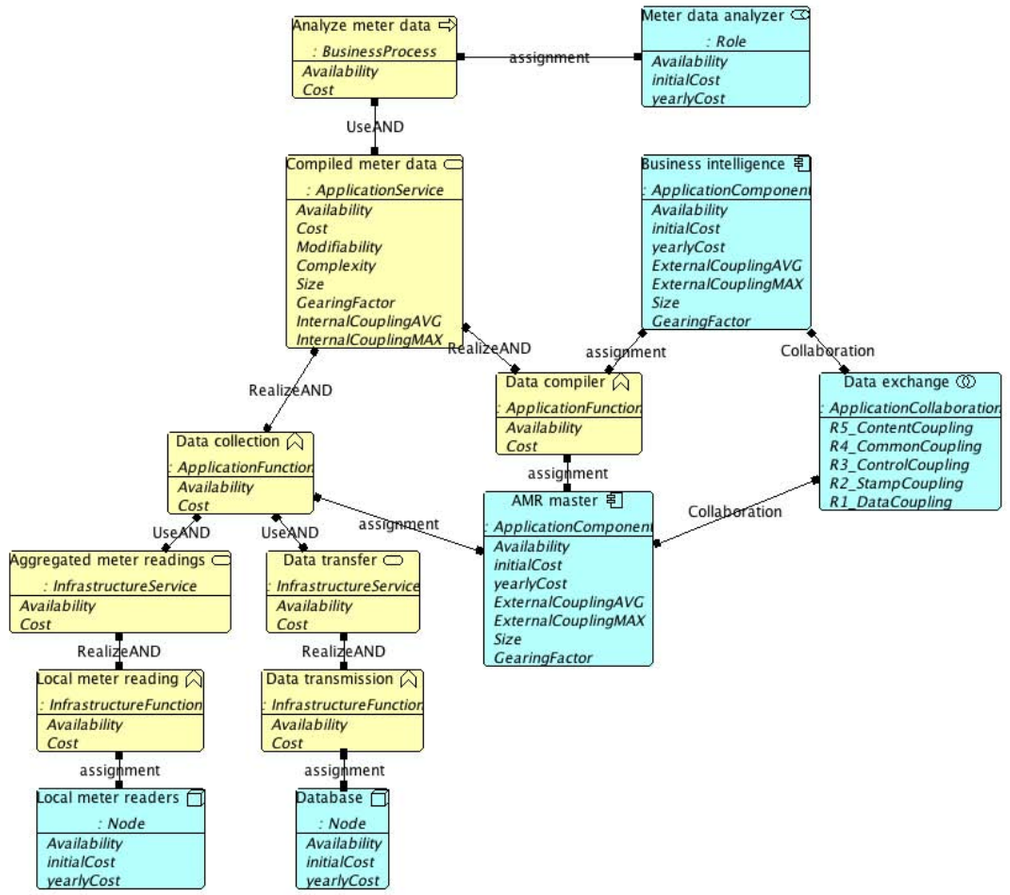


Figure 19: *Complete tutorial model example after the nine steps.*

The CIO is satisfied with the functionality of the solution and the cost, but there is a wish to find a scenario with higher modifiability and at least better availability in the infrastructure and application layers. Therefore, the architect needs to find a second scenario that the CIO can compare this solution with before a decision can be made.

This chapter of the book has focused on explaining what a metamodel is and how to model architectures. The next chapter will present a metamodel for enterprise architecture analysis of multiple attributes including modifiability, availability, cost, interoperability, application usage, and data accuracy.

The Multi-Attribute Prediction (MAP) class diagram

3

A brief description of the MAP class model.

The chapter briefly introduces the MAP class model. Subsequently, chapters 4 through 10 describe the individual components (viewpoints), which the MAP class model contains.

Chapter 3 The MAP class diagram

This chapter presents a multi-attribute analysis metamodel for application modifiability, data accuracy, application usage, service availability, interoperability, cost and utility. Previous work on combining different quality attributes in one metamodel has been presented in [14-16]. Each of the six attributes are then detailed as separate viewpoints in the coming chapters.

For the interested reader this chapter is based on two fundaments, namely the Enterprise Architecture Analysis Tool[3] (EA^{2}T) [17-20] and the Predictive, Probabilistic Architecture Modeling Framework (P^{2}AMF) [21], also referred to as Probabilistic Imperative Object Constraint Language (Pi-OCL) [22] or Probabilistic Object Constraint Language (P-OCL) [23, 24].

3.1 Metamodel elements

This section presents the metamodel, cf. Figure 20. The main classes in the metamodel are based on ArchiMate, thus the three layers; technology, application, and business are all used. The three categories; passive, behavior, and active structure are also employed in the metamodel. The metamodel has five viewpoints tailored for specific purposes, these are presented in the coming sections.

In the metamodel there are six classes (in grey) which are not used by the modeler namely; BehaviorStructure, Service, BehaviorElement, PassiveComponentSet, ActiveStructureElement and Requirement. These six classes all have two or more subclasses inheriting their properties. This means that the attributes, relationships, constraints, and behavior of the parent class is also existent in the child class, i.e. inheritance from its parent.

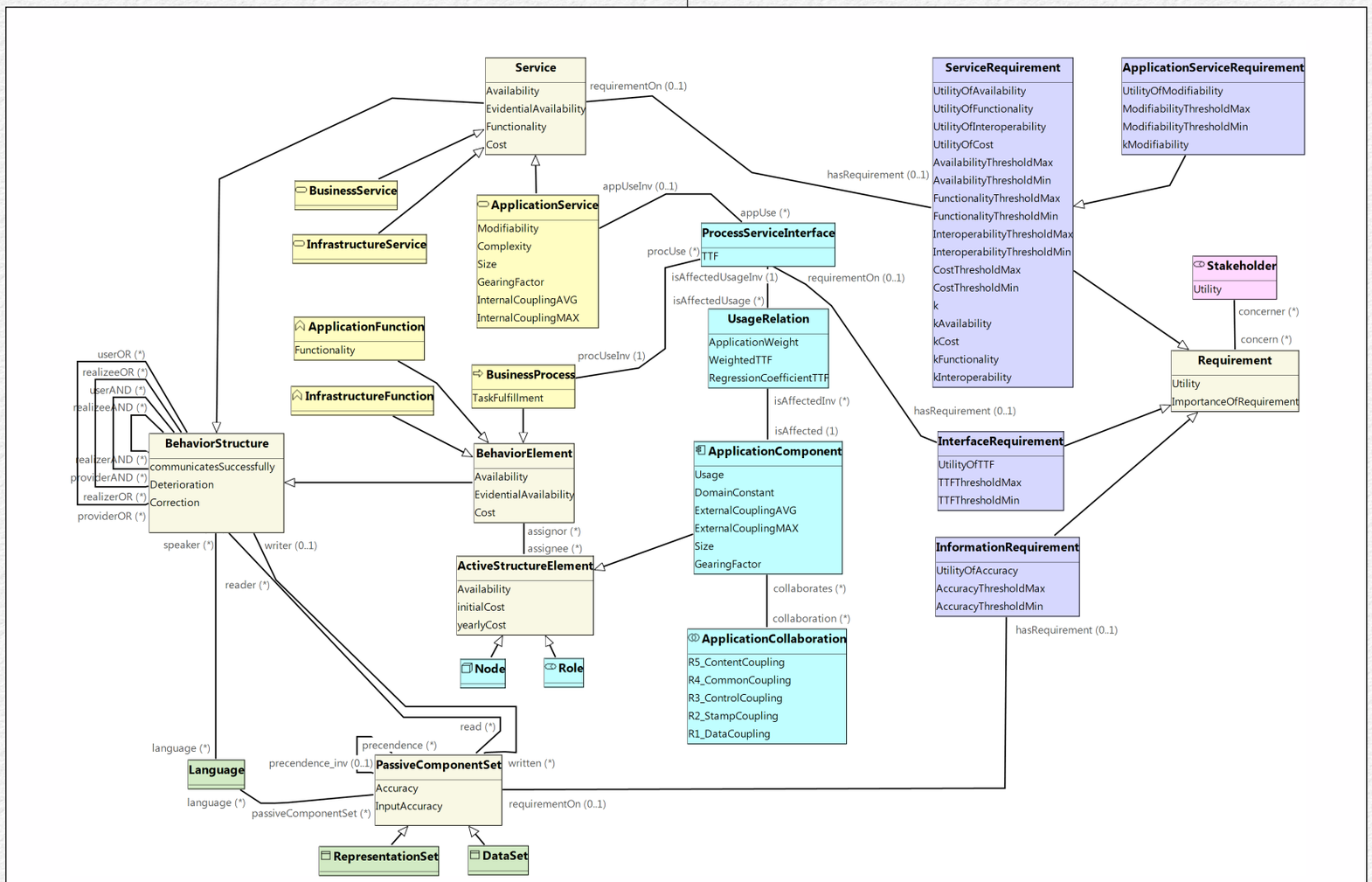


Figure 20: *The metamodel.*

Each child class can also have its own properties. A parent class is sometimes referred to as superclass and the child subclass.

The rest of the subsection will present the definitions of all the metamodel classes and class relationships. The attributes will be presented in the coming viewpoint sections.

3.1.1 Inheritance elements

Behavior element

`BehaviorStructure` is the central class in the metamodel. It is not used when modeling, but plays an important role since both `Services` and `BehaviorElements` are inheriting from this class. Furthermore the `BusinessService`, `ApplicationService`, and `InfrastructureService` all inherits from the `Service` class, while the `BusinessProcess`, `ApplicationFunction`, and `InfrastructureFunction` inherits from the `BehaviorElement` class. There are seven class relationships in the behavior element class.

Relationship: realize. In ArchiMate the realization relationship links a logical entity with a more concrete entity that realizes it. The realize relation `Service`.*realizer*.`BehaviorElement` or the other way around, `BehaviorElement`.*realizee*.`Service` is used when an behavior element is realizing a service. Meaning that the functionality of the service is provided by the behavior element.

Relationship: used by. In ArchiMate the used by relationship models the use of services by processes, functions, or interactions and the access to interfaces by roles, components, or collaborations. The `BehaviorElement`.*uses*.`Service` or the other way around `Service`.*usedBy*.`BehaviorElement` is used when a behavior element is using a service to perform its behavior.

The two relations, *realize* and *usedBy*, are available as both an AND and an OR option. The correct usage of these are of great importance when evaluating service availability (cf. section 7). In Figure 21 the `Service` is realized by four `BehaviorElements`, A, B, C, and D. Where, A and B realize the `Service` through the *realizeAND* relationship. While, C and D realize the `Service` through the *realizeOR* relationship. When evaluating for instance availability this example would mean that A, B, and one of C or D have to be available, $A \wedge B \wedge (C \vee D)$.

Relationship: speaks language. The speaks language relation, `BehaviorStructure`.*language*.`Language` or the other way around, `Language`.*speaker*.`BehaviorStructure` is used in order to model what languages a behavior element can understand or vice versa what behavior elements that are capable of understanding a certain language.

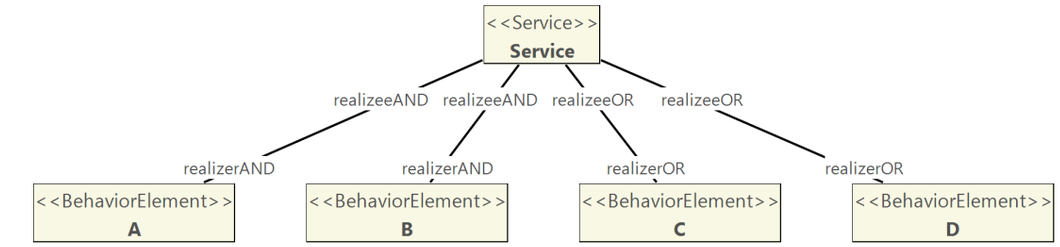


Figure 21: *Demonstration of the AND/OR relationships, $A \wedge B \wedge (C \vee D)$*

Relationship: read. The read relation, `BehaviorStructure.`*read*`.PassiveComponentSet` or the other way around `PassiveComponentSet.`*reader*`.BehaviorStructure` is used in order to model that a behavior element is reading a certain passive component set.

Relationship: write. The write relations `BehaviorStructure.`*written*`.PassiveComponentSet` or the other way around `PassiveComponentSet.`*writer*`.BehaviorStructure` is used in order to model that a behavior element can write to a certain passive component set.

Behavior element

`BehaviorElement` is a class not used when modeling, it is a subclass of `BehaviorStructure` and a superclass to `BusinessProcess`, `ApplicationFunction`, and `InfrastructureFunction`. `BehaviorElement` has one class relation added on top of the inherited relations from the `BehaviorStructure`.

Relation: assignment. In ArchiMate the assignment relationship links active elements (e.g., business roles or application components) with units of behavior that are performed by them, or business actors with business roles that are fulfilled by them. In the metamodel, `BehaviorElement.`*assignee*`.ActiveStructureElement` or the other way around `ActiveStructureElement.`*assignor*`.BehaviorElement` is used in the same way.

Service

The service class is not used for modeling, it is a subclass of `BehaviorStructure` and superclass of the three classes; `BusinessService`, `ApplicationService`, and `InfrastructureService`.

Active structure element

`ActiveStructureElement` is a class not used for modeling, it is a superclass to some of the classes in the metamodel which performs a behavior aspect. The `ActiveStructureElement` is a superclass to `Node`, `Role` and `ApplicationComponent`. The .`ActiveStructureElement` has one relationship.

Relation: Assignment. In ArchiMate the assignment relationship links active elements (e.g., business roles or application components) with units of behavior that are performed by them, or business actors with business roles that are fulfilled by them. In the metamodel, `BehaviorElement.`*assignee*`.ActiveStructureElement` or the other way around `ActiveStructureElement.`*assignor*`.BehaviorElement` is used in the same way.

Passive component set

`PassiveComponentSet` is a class not actively used for modeling, it is a superclass to two informational classes `RepresentationSet` and `DataSet.` The passive component set has four relationships.

Relation: Precedence. The precedence relationship is a self reference, it links the preceding passive component sets with the subsequent passive component set. The relationship is used to model which previous data that is composed to the new passive component set.

Relationship: read. The read relation, `BehaviorStructure.`*read*`.PassiveComponentSet` or the other way around `PassiveComponentSet.`*reader*`.BehaviorStructure` is used in order to model that a behavior element is reading a certain passive component set.

Relationship: write. The write relationships `BehaviorStructure.`*written*`.PassiveComponentSet` or the other way around `PassiveComponentSet.`*writer*`.BehaviorStructure` is used in order to model that a behavior element can write to a certain passive component set.

Relationship: language. The language relationship, `PassiveComponentSet.`*language*`.Language` is used to model which language the `PassiveComponentSet` is stored in.

Requirement

`Requirement` is a class not actively used for modeling, it is a superclass to the three requirement classes: `ServiceRequirement`, `InterfaceRequirement` and `ApplicationServiceRequirement`. In ArchiMate a requirement is defined as a statement of need that must be realized by a system. The Requirement has one relationship.

Relationship: concern. The concern relationship `Requirement.`*concern*`.Stakeholder` or the other way around `Stakeholder.`*concern*`.Requirement` is used to model which stakeholder that has a certain requirement.

3.1.2 Modeling elements

Business service

In ArchiMate a business service is defined as a service that fulfills a business need for a customer (internal or external to the organization). The `BusinessService` is a subclass of the `Service`. It is realized by business processes and can also be used by a business process. The business services is typically the core of an enterprise apart from physical products manufactured. Examples of a business service are accounting, marketing, selling, and intelligence gathering & analysis services.

Application service

In ArchiMate an application service is defined as a service that exposes automated behavior. The `ApplicationService` is a subclass of `Service`. It has one extra relationship, application use. The application service is realized by application functions, it can be used by both application functions and business processes. Examples of application services are sales order compiling, automated information tracing and intelligence collecting, and transaction processing.

Relationship: application use. The relationship, `ApplicationService.`*appUse*`.ProcessServiceInterface`, is used

to bind an application service the process service interface class when performing application usage evaluation (cf. section 6).

Infrastructure service

In ArchiMate an infrastructure service is defined as an externally visible unit of functionality, provided by one or more nodes, exposed through well-defined interfaces, and meaningful to the environment. The `InfrastructureService` is a subclass of `Service`. It is realized by infrastructure functions and can be used by application functions and infrastructure functions. A infrastructure function could for example be data storage, file naming and version control, or information passing.

Business process

In ArchiMate a business process is defined as a behavior element that groups behavior based on an ordering of activities. It is intended to produce a defined set of products or business services. In our metamodel the `BusinessProcess` is a subclass of the `BehaviorElement`. It also has its own reference, process use. The business process can realize `BusinessServices` through the realize relationship. It can use `ApplicationFunctions` modeled with the usedBy relationship and it can have `Roles` assigned to it with the assignment relationship. Examples of business processes are management processes such as governance and strategic management, operational processes such as manufacturing and development, or supporting processes such as recruitment and technical support.

Relationship: process use. The process use, `BusinessProcess`.*procUse*.`ProcessServiceInterface`, is used when evaluating application usage (cf. section 6).

Application function

In ArchiMate an application function is defined as a behavior element that groups automated behavior that can be performed by an application component. The `ApplicationFunction` realize `ApplicationServices` modeled with the realize relationship, it can make use of `InfrastructureServices` modeled with a use relationship, it can also read and write `PassiveComponentsSets` modeled with the *read* and *write* relationships. The application function defines the behavior of an application component, the application component of which the application function is performing its functionality is modeled with the assignment relationship. Essentially the application function describes the important behavior of an application component and how it acts with the environment. Examples of application functions are billing and work order administration.

Infrastructure function

In ArchiMate an infrastructure function is defined as a behavior element that groups infrastructural behavior that can be performed by a node. The `InfrastructureFunction` can realize `InfrastructureServices` through the *realize* relationship, it can also be assigned `Nodes` with the *assignment* relationship. Example of infrastructure functions are access control and data management and distribution.

Application component

In ArchiMate an application component is defined as a modular, deployable, and replaceable part of a software system that encapsulates its behavior and data, and exposes these through a set of interfaces. The `ApplicationComponent` is a subclass of `ActiveStructureElement` and has one of its own relationships. The behavior of the application component is modeled through the use of `ApplicationFunctions` related to the `ApplicationComponent` with the assignment relationship. Examples of applications functions are units of software such as web containers and data managers or a billing component.

Relation: collaboration. The collaboration relationship, `ApplicationComponent`.*collaboration*.`ApplicationCollaboration` is used to model what application components that are acting together to perform a collective behavior.

Node

In ArchiMate a node is defined as a computational resource upon which artifacts may be stored or deployed for execution. The Node is a subclass of the `ActiveStructureElement`. Examples of nodes are client workstations, web and database servers, and programmable logic controllers.

Role

In ArchiMate a business role is defined as the responsibility for performing specific behavior, to which an actor can be assigned. The `Role` is a subclass of the `ActiveStructureElement`.

Representation set

The `RepresentationSet` is a subclass of `PassiveStructureElement`. The representation set is used to model unstructured data stored either electronically or on paper. The representation set could for instance be used to model a collection of meeting protocols stored as PDF-files in a folder.

Data set

The `DataSet` class is a subclass of `PassiveStructureElement`. The Data set is used to model structured data, typically stored in databases. A data set is an aggregation of multiple data objects. As an example a data set could be a collection of customers in a customer database.

Application collaboration

In ArchiMate an application collaboration is defined as an aggregate of two or more application components that work together to perform collective behavior. Between every pair of `ApplicationComponents` that are collaborating there should be an `ApplicationCollaboration` connected to the two `ApplicationComponents` with the *collaboration* relation. An example of an application collaboration is when a product order component is collaborating with a billing component to create a bill in order to charge a customer.

Relation: collaboration. The collaboration relationship, `ApplicationComponent`.*collaboration*.`ApplicationCollaboration` is used to model what application components that are acting together to perform a collective behavior.

Process service interface

The `ProcessServiceInterface` is an intermediate class used for application usage evaluation (cf. section 6). It links a `BusinessProcess` with `ApplicationServices` and `ApplicationComponents` in order to do the evaluation. The class has three relationships.

Relationship: application usage. The relationship, `ApplicationService`.*applUse*.`ProcessServiceInterface`, is used in order to bind an application service to the usage evaluation.

Relationship: process usage. The relationship, `BusinessProcess`.*procUse*.`ProcessServiceInterface`, is used in order to bind a business process to the usage evaluation.

Relationship: is affected usage. The relationship, `ProcessServiceInterface`.*isAffectedUsage*.`UsageRelation`, is used in order to bind an application component to the usage evaluation through the intermediate class `UsageRelation`.

Usage relation

The `UsageRelation` class is an intermediate class used for application usage evaluation (cf. section 6). The class has two relationships.

Relationship: is affected usage. The relationship, `ProcessServiceInterface`.*isAffectedUsage*.`UsageRelation` connects the two intermediate classes ProcessServiceInterface and UsageRelation for application usage evaluation.

Relationship: is affected. The relationship, `UsageRelation`.*isAffected*.`ApplicationComponent`, connects the application component to the usage relation class for application usage evaluation.

Language

The `Language` class is used to model a language. It can be a natural language such as English as well as a technical language such as the programming language Java or a protocol following IEC 61850.

Relationship: speaks language. The speaks language relationship, `BehaviorStructure`.*language*.`Language` or the other way around, `Language`.*speaker*.`BehaviorStructure` is used in order to model what languages a behavior element can understand or vice versa what behavior elements that are capable of understanding a certain language.

Relationship: language. The language relationship, `PassiveComponentSet`.*language*.`Language` is used to model which language the `PassiveComponentSet` is stored in.

Service requirement

The `ServiceRequirement` class is a subclass of `Requirement` it is also the superclass of the `ApplicationServiceRequirement` class. The service requirement is used to model the requirements a stakeholder has on a certain service. The service requirement class has one relationship.

Relationship: requirement on. The requirement on relationship, `ServiceRequirement`.*requirementOn*.`Service` or the other way around, `Service`.*hasRequirement*.`ServiceRequirement` is used to model the requirements which a service should fulfill.

Application service requirement

The `ApplicationServiceRequirement` class is a subclass of `ServiceRequirement`. The application service requirement has extra properties specific to the application service domain.

Interface requirement

The `InterfaceRequirement` class is a subclass of `Requirement`. The interface requirement has extra properties specific to the application usage domain.

Relationship: requirement on. The requirement on relationship, `InterfaceRequirement`.*requirementOn*.`ProcessServiceInterface` or the other way around,

`ProcessServiceInterface`.*hasRequirement*.`InterfaceRequirement` is used to model the requirements which a process service interface should fulfill.

Application modifiability

4

A description of MAP's application modifiability viewpoint.

Chapter 4 Application modifiability

Business environments today progress and change rapidly to keep up with evolving markets. Most business processes are supported by information systems and as the business processes change, the systems need to be modified in order to continue supporting the processes. Modifications include extending, deleting, adapting, and restructuring the enterprise systems [25]. The modification effort ranges from adding a functional requirement in a single system to implementing a service-oriented architecture for the whole enterprise.

An essential issue with today's information systems is that many of them are interconnected, thus a modification to one system may cause a ripple effect among other systems. Also, numerous systems have been developed and modified over many years. Making further changes to these systems might require a lot of effort from the organization, for example due to a large number of previous modifications implemented ad hoc. Problems like these raise questions for IT decision makers such as: Is the source code easy to grasp? Which systems are interconnected and how? Are the systems too complex?

Several studies show that the modification work is the phase of a system's lifecycle that consumes the greatest portion of resources; [26] report that over 70 % of the software budget is spent on maintenance, [27] refers to studies stating that the maintenance cost, relative to the total life cycle cost of a software system, has been increasing from 40 % in the early 1970s up to 90 % in the early 1990s, and [28] states that "the cost of maintenance, rather than dropping, is on the increase".

The activities of modifying enterprise information systems are typically executed in projects, and information system decision

***Movie 5**: Modifiability*

makers often find it difficult to estimate and plan their change projects. Thus, a large proportion of the projects aiming to modify a system environment fail. That is, the projects tend to take longer time and cost more than expected. [29] state that 23 % of the software projects are cancelled before completion, whereas of those completed only 28 % were delivered on time, and the average software project overran the budget by 45 %. This can often occur due to lack of information about the systems being changed. According to [29], software engineers must be able to understand and predict the activities, as well as manage the risks, through estimation and measurement. Therefore, it would be useful for the decision makers to gather more information in a structured manner and use this information to analyze how much effort a certain modification to an enterprise information system would require.

This section of the book presents the modifiability viewpoint, which intends to provide such decision support. The original work that this section is based on can be found in [30-35].

4.1 How to measure application modifiability

The issue of dealing with modifiability is not an enterprise architecture specific problem. Managing and assessing information system change has been addressed in research for many years. Some of the more well-known assessment approaches include the COnstructive COst MOdel (COCOMO), and the Oman taxonomy.

COCOMO, COnstructive COst MOdel, was in its first version released in the early 1980's. It became one of the most frequently used and most appreciated IT cost estimation models of that time. Since then, development and modifications of COCOMO have been performed several times to keep the model up to date with the continuously evolving software development trends. Effort estimation with COCOMO is based on the size of the software, an approximate productivity constant A, an aggregation of five scale factors E (precentedness, development flexibility, architecture/risk resolution, team cohesion, and process maturity), and effort multipliers to 15 cost driving attributes [36].

The Definition and Taxonomy for Software Maintainability presented by Oman et al. in [37] provides a hierarchical definition of software maintainability in the form of a taxonomy. Oman et al. found three broad categories of factors influencing the maintainability of an information system namely; management, operational environment, and the target system. Each of these top-level categories is then further broken down into measurable attributes. According to Oman et al. the taxonomy can be useful for developers by defining characteristics affecting the software maintenance cost of the software they are developing.

Hence, the developers can write highly maintainable software from the beginning by studying the taxonomy. Maintenance personnel can use the taxonomy to evaluate the maintainability of the software they are working with in order to pin point risks et cetera. Project managers and architects can use the taxonomy in order to prioritize projects and locate areas in need of re-design.

COCOMO focuses on the cost of developing or changing information systems, where architecture modifiability is one part of it. Oman et al. do

not provide any support for analysis in their taxonomy. Thus, neither of these fit our purpose perfectly. However, both COCOMO and Oman et al. use cost driving/maintenance factors related to software complexity, size, and coupling. These three metrics are the most commonly used when estimating the modifiability of information systems. The following paragraphs of this section will present each one of these in detail.

Complexity

IEEE defines complexity as the degree to which a system or component has a design or implementation that is difficult to understand and verify [38]. Halstead's complexity metric was introduced in 1977 [39], it is based on the number of operators (e.g. and, or, while) and operands (e.g. variables and constants) in a software program. A drawback of Halstead's complexity metric is that it lacks predicting power for development effort since the value can be calculated first after the implementation is complete [29]. Information flow complexity, *IFC*, as presented in [40] is based on the idea that a large amount of information flows is caused by low cohesion, low cohesion is in turn causing a high complexity. One problem with the IFC metric is that it produces a lower complexity value for program code using global variables compared to a solution which uses function arguments when called, this is contradicting to common software design principles [41]. In this book McCabe's Cyclomatic Complexity (MCC) metric is employed [42]. [29] has identified that MCC is useful to, identify overly complex parts of code, identify non-complex part of code, and to estimate maintenance effort. MCC is based on the control structure of the software, the control structure can be expressed as a control graph. The cyclomatic complexity value of a system with the control graph G is calculated with the following equation: $v(G) = e - n + 2$ or equivalently $v(G) = DE + 1$ where e =number of edges in the control graph, n=number of nodes in the control graph, DE=number of predicates. Considering the example code presented in Figure 22 the control graph G_{sort} can be obtained.

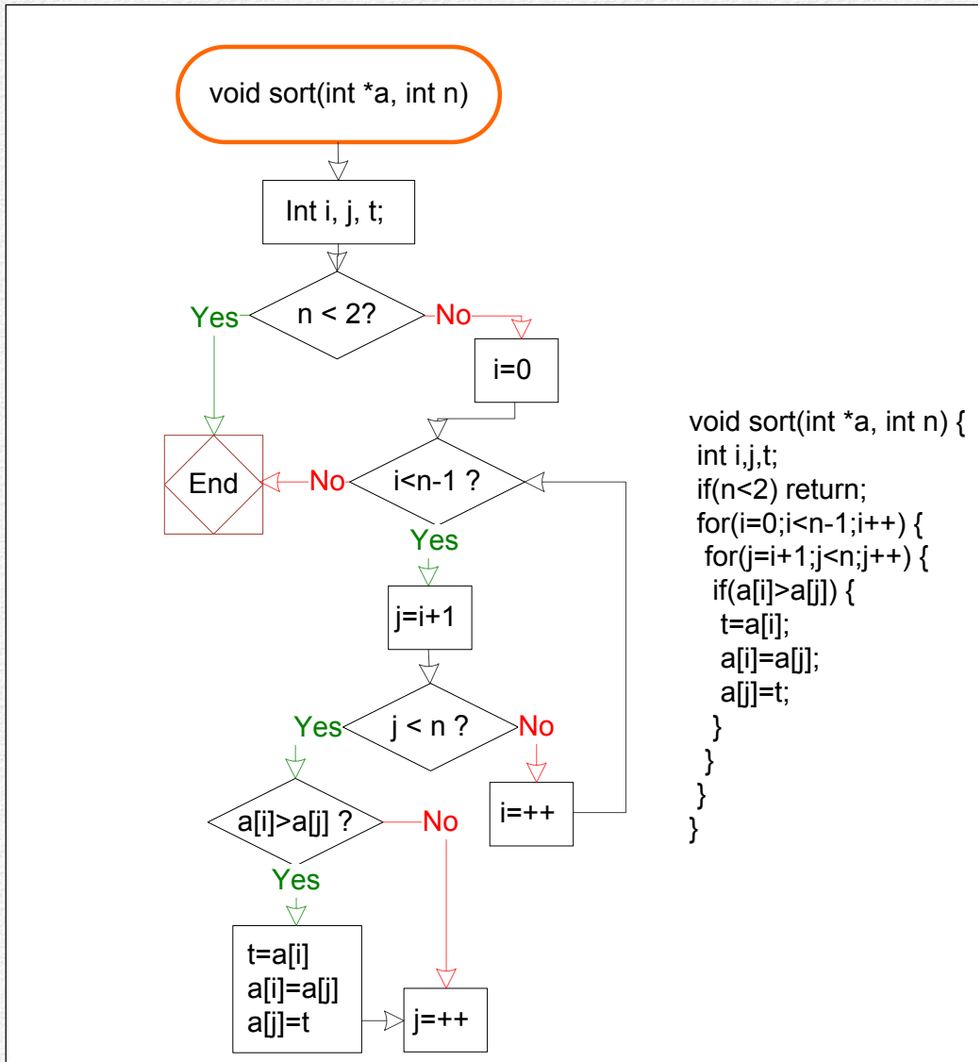


Figure 22: An example of a control graph G_{sort} with $v(G_{sort}) = 4$

The MCC value of G_{sort} (cf. Figure 22) is $v(G_{sort}) = 14 - 12 + 2 = 4$. McCabe has performed a study indicating that the cyclomatic complexity value of a component should be kept below 10 [42]. MCC has been used in other studies providing additional complexity levels and guidelines on how complex a piece of software code is [29]:

- 1-4, a simple procedure.
- 5-10, a well-structured and stable procedure.
- 11-20, a more complex procedure.
- 21-50, a complex procedure, worrisome.
- 50<, an error-prone, extremely troublesome, untestable procedure.

Size

Lines Of Code (LOC) and Function Points (FP) are two ways to measure the size of an information system. FP are based on the inputs, outputs, interfaces and databases in a system [29]. FP have the advantage of being technology independent, reasonably reliable and accurate, and they are effective from an early stage of the system life cycle [29]. The disadvantages of the FP size metric is that it requires significant effort to derive [29]. The LOC in a system provides the core functionality and can therefore be of importance when estimating how easy it would be to implement changes to the system. LOC in a system can be measured in different ways: Using source lines of code (SLOC) every line of code in the software implementation is counted. Non-commented lines of code (NLOC) is a subset of the previous option, where the blank lines and comments are excluded. Logical lines of code (LLOC) is another approach, where only the executable statements of the software are counted. The most popular option is NLOC, however the most important thing is to be consistent with the way you measure [29]. No matter which LOC measure that is used it needs to be well specified to provide a reliable measurement [29]. A framework on how to measure lines of code has been created by the Software Engineering Institute of Carnegie-Mello University [43], with the aid of this framework the LOC measure can be specified to provide a coherent way of how to measure LOC.

Aivosto suggests a classification of system size for systems coded with Visual Basic 2. The classification is based on long-time experience, but has not been validated making it less reliable. However, given the size of the systems studied in [44], the classification seems trustworthy. Related to system size, operating systems can be much larger with over 40 million LOC [45], however an operating system would not be modeled as an application that an enterprise wishes to modify. Rumor has it that SAP has over 250 million LOC in their product portfolio, but we believe that no enterprise would model SAP as one application. Thus, the classification of system size by Aivosto still seems appropriate for our purpose.

Since different programming languages are more or less expressive per line of code [29], a gearing factor can be used when comparing the lines of code of two systems if they are created in different programming languages. A high gearing factor value indicates poor expressiveness, hence a programming language with a low gearing factor require less lines

of code to implement a function; given that the language is appropriate to use. [46] has published gearing factors for some programming languages of which a subset is presented in Table 3.

Classification	LOC
Small	0-9.999
Medium	10.000-49.999
Semi-large	50.000-99-999
Large	100.000-499.999
Very Large	500.000≤

Table 2: *System size classification.*

Language	Gearing factor
Assembly-Basic	320
C#	59
C++	55
Java	53
Visual Basic	52
ASP	50

Table 3: *Gearing factors for some commonly used programming languages.*

Coupling

IEEE has defined coupling as the manner and degree of interdependence between software modules. Types include common-environment coupling, content coupling, control coupling, data coupling, hybrid coupling, and pathological coupling [38]. Fenton and Melton have developed a coupling metric based on Myers coupling levels [47], these levels are:

- Content coupling relation $R_5 : (x, y) \in R_5$ if x refers to the internals of y , i.e., it branches into, changes data, or alters a statement in y.
- Common coupling relation $R_4 : (x, y) \in R_4$ if x and y refer to the same global variable.
- Control coupling relation $R_3 : (x, y) \in R_3$ if x passes a parameter to y that controls its behavior.
- Stamp coupling relation $R_2 : (x, y) \in R_2$ if x passes a variable of a record type as a parameter to y , and y uses only a subset of that record.
- Data coupling relation $R_1 : (x, y) \in R_1$ if x and y communicate by parameters, each one being either a single data item or a homogeneous set of data items that does not incorporate any control element.
- No coupling relation $R_0 : (x, y) \in R_0$ if x and y have no communication, i.e., are totally independent.

The Fenton and Melton coupling measure is pairwise calculated between components, where n = number of interconnections between x and y . i = level of highest (worst) coupling type found between x and y.

$$C(x, y) = i + \frac{n}{n + 1}$$

Modifiability

To evaluate the modifiability, the complexity levels by [29], the coupling levels by [47] and the size levels by Aivosto are used in order to indicate how "good" a modeled architecture is. The modifiability level is then evaluated as the sum of the three individual metrics. The reason to summarize the values is to create a metric that can be used in order to indicate whether a system is likely to be easy to modify or not. According to the correlations levels in [41] the three metrics used are more or less equally important when estimating the level of modifiability in application services. The modifiability metric gives a rough estimation, which can be of value when making decisions regarding different architecture scenarios.

The change cost framework presented by Lagerström et al. [30-35] does not only contain the application service modifiability assessment, but it also takes change management processes, documents, and roles, as well as change project organizational attributes into consideration when estimating the cost of software change projects. We have however limit it to the architectural viewpoint.

The assessment method for application service modifiability presented in this book uses the common software evaluation metrics complexity, size, and coupling together with enterprise architecture modeling and analysis.

4.2 The application modifiability viewpoint

This section describes the application modifiability viewpoint, cf. Figure 23. The viewpoint has the following main concepts:

- Service
  - ApplicationService
- BehaviorElement
  - ApplicationFunction
- ActiveStructureElement
  - ApplicationComponent
- ApplicationCollaboration

Concerns

Using the modifiability viewpoint makes it possible to estimate the modifiability of different application services. This information can be of use when; choosing between different architectural solutions, assigning development efforts, looking for possible ripple effects before initiating a change project, and finding risks that are important to manage in order to prevent projects from exceeding the budget.

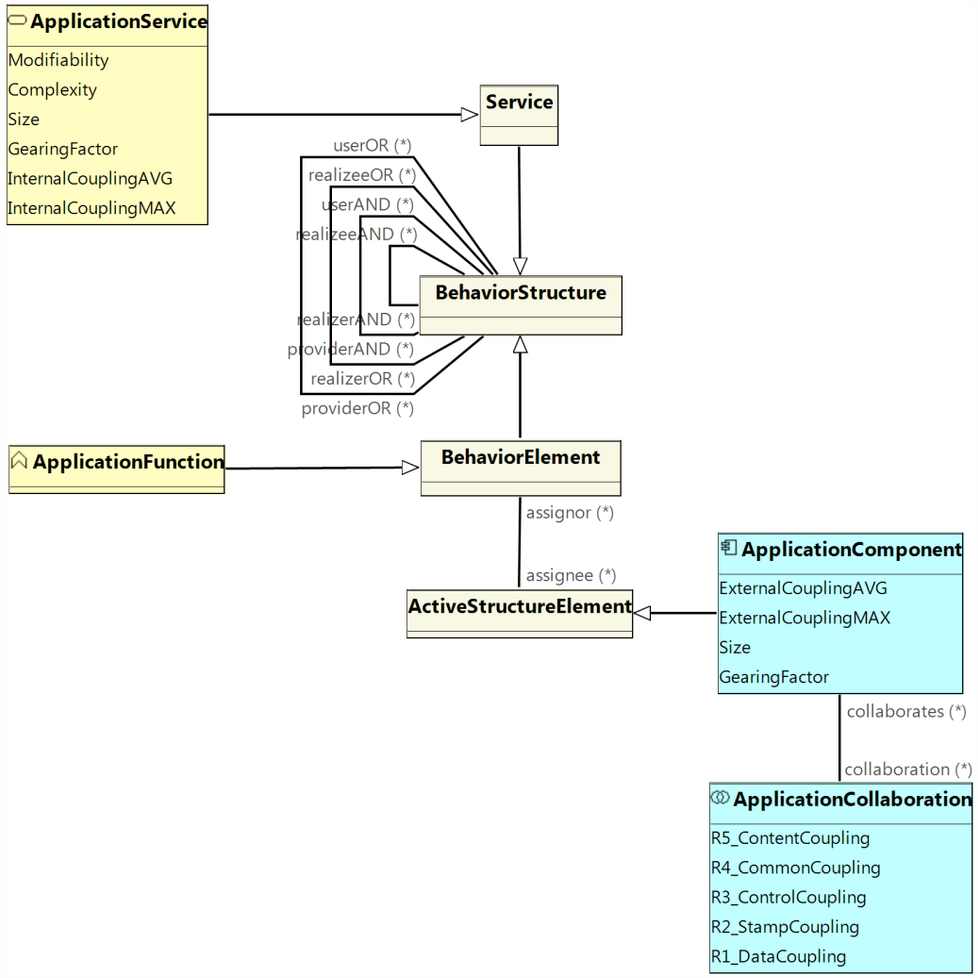


Figure 23: *The modifiability viewpoint.*

Stakeholders

The typical stakeholders for the modifiability viewpoint are; CIOs when handling the project portfolio, architects when choosing between different architecture solutions, project managers when planning the change projects, and developers when modifying applications.

Theory

The modifiability of an application service is in this book assessed based on three commonly used metrics namely; complexity, size, and coupling.

The `ApplicationService` class contains the attributes `Modifiability`, `Complexity`, `Size`, `Gearing Factor`, and `Coupling`.

`ApplicationService.Modifiability.` The modifiability metric is an aggregation of the attributes: `ApplicationService.Complexity` α, `ApplicationService.InternalCouplingMAX` β, and `ApplicationService.Size` γ.

The complexity levels from [29] are used to give complexity c a numerical value a, where $0 \leq \alpha \leq 5$.

- If `ApplicationService.Complexity` is $c = 0$, then $\alpha = 5$.
- If `ApplicationService.Complexity` is $1 \leq c \leq 4$, then $\alpha = 4$.
- If `ApplicationService.Complexity` is $5 \leq c \leq 10$, then $\alpha = 3$.
- If `ApplicationService.Complexity` is $11 \leq c \leq 20$, then $\alpha = 2$.
- If `ApplicationService.Complexity` is $21 \leq c \leq 50$, then $\alpha = 1$.
- If `ApplicationService.Complexity` is $50 < c$, then $\alpha = 0$.

The coupling levels from [47] are used to give internal coupling (max) icm a numerical value β, where $0 \leq \beta \leq 5$.

- If `ApplicationService.InternalCouplingMax` is ≤ 1, then $\beta = 5$.
- If `ApplicationService.InternalCouplingMax` is $1 \leq icm < 2$, then $\beta = 4$.
- If `ApplicationService.InternalCouplingMax` is $2 \leq icm < 3$, then $\beta = 3$.
- If `ApplicationService.InternalCouplingMax` is $3 \leq icm < 4$, then $\beta = 2$.
- If `ApplicationService.InternalCouplingMax` is $4 \leq icm < 5$, then $\beta = 1$.
- If `ApplicationService.InternalCouplingMax` is $5 \leq icm$, then $\beta = 0$.

The size levels from Aivosto are used to give size s a numerical value γ, where $0 \leq \gamma \leq 5$.

- If `ApplicationService.Size` is $s < 10.000$, then $\gamma = 4$.
- If `ApplicationService.Size` is $10.000 \leq s < 50.000$, then $\gamma = 3$.
- If `ApplicationService.Size` is $50.000 \leq s < 100.000$, then $\gamma = 2$.
- If `ApplicationService.Size` is $100.000 \leq s < 500.000$, then $\gamma = 1$.
- If `ApplicationService.Size` is $s \leq 500.000$, then $\gamma = 0$.

If S is an application service, then the modifiability value $= \alpha + \beta + \gamma$ with $\mathrm{V}(S.Modifiability) = \{x \in \mathrm{N} : 0 \leq x \leq 14\}$. A low modifiability value indicates that an application service is difficult to change just as a high modifiability value indicates the opposite.

`ApplicationService.Complexity`. The complexity attribute is calculated as the cyclomatic complexity by McCabe [42]. The application components are used as nodes and the values in the attributes of the application collaboration class are used as edges. This includes relations to an application component outside of the owning application service, in the case if an application collaboration exists between one application component realizing the service is collaboration with a component realizing a different application service. If $I = \{i_1, \ldots, i_n\}$ is a list of application collaborations, S is an application service, c is a application component where $c \subseteq S.realize$, then and $I \subseteq S.realize.collaboration$, then

$$f(S.Complexity) = \sum_{i=1}^{n} i_i.R5\_ContentCoupling + \sum_{i=1}^{n} i_i.R4\_CommonCoupling + \sum_{i=1}^{n} i_i.R3\_ControlCoupling + \sum_{i=1}^{n} i_i.R2\_StampCoupling + \sum_{i=1}^{n} i_i.R1\_DataCoupling - \sum_{i=1}^{n} c_i + 2.$$

$$\mathrm{V}(S.Complexity) = \mathrm{N}.$$

`ApplicationService.Size` Equivalent source lines of code (ENLOC) is a size measure which uses a gearing factor to get a size measure which allows size comparison between applications written in different

programming languages. If $F = \{f_1, \ldots, f_n\}$ is a list of application functions, S is an application service and $F \subseteq S.realizedBy$, then

$$f(S.ENLOC) = \sum_{i=1}^{n} \frac{S.GearingFactor}{f_i.assignee.GearingFactor} * f_i.assignee.NLOC.$$

$$\mathrm{V}(S.ENLOC) = \mathrm{R}.$$

`ApplicationService.GearingFactor`. If S is an application service, then $\mathrm{V}(S.GearingFactor) = \mathrm{N}$. The gearing factor is given as evidence in the model.

`ApplicationService.InternalCouplingAVG` The `InternalCouplingAVG` is the internal average coupling of the application service. It is calculated as the arithmetic mean of the Fenton and Melton Software Metric [47] for all pair wise coupling measures within the application service divided by the number of pairs. If $I = \{i_1, \ldots, i_n\}$ is a list of application collaborations, S is an ApplicationService, C is a `ApplicationComponent` where $C \subseteq S.realize$ and $I \subseteq S.realize.collaboration$, then

$$C.CouplingAVG = \frac{1}{n} \sum_{j=1}^{n} i_j.couplingInPair().$$

$$\mathrm{V}(C.CouplingAVG) = \{x \in \mathrm{R} : 0 \leq x < 6\}.$$

`ApplicationService.InternalCouplingMAX` The `InternalCouplingMAX` is the internal max coupling of the application service is. It gives the maximum value of all the connections pair to the application component within the application service. If $I = \{i_1, \ldots, i_n\}$ is a list of application collaborations, S is an ApplicationService, C is a `ApplicationComponent` where $C \subseteq S.realize$ and $I \subseteq S.realize.communication$, then $C.couplingMAX = couplingInPair(m)$ where $m \in P$ and $couplingInPair(i_j) \leq couplingInPair(m)$ for all elements in P.

$$\mathrm{V}(C.CouplingMAX) = \{x \in \mathrm{Q} : 0 \leq x < 6\}.$$

The `ApplicationComponent` class contains the attributes Coupling, Size, and Gearing Factor.

`ApplicationComponent.ExternalCouplingAVG` is calculated as the arithmetic mean of the Fenton and Melton Software Metric [47] for all pair wise coupling measures divided by the number of pairs. If $I = \{i_1, \ldots, i_n\}$ is a list of application collaborations, C is a `ApplicationComponent` and $I \subseteq C.collaboration$, then

$$C.CouplingAVG = \frac{1}{n} \sum_{j=1}^{n} i_j.couplingInPair().$$

$$\mathrm{V}(C.CouplingAVG) = \{x \in \mathrm{R} : 0 \leq x < 6\}.$$

`ApplicationComponent.ExternalCouplingMAX` gives the maximum value of all the connections pair to the application component. If $I = \{i_1, \ldots, i_n\}$ is a list of application collaborations, C is a `ApplicationComponent` and $I \subseteq C.communication$, then $C.couplingMAX = couplingInPair(m)$ where $m \in P$ and $couplingInPair(i_j) \leq couplingInPair(m)$ for all elements in P.

$$\mathrm{V}(C.CouplingMAX) = \{x \in \mathrm{Q} : 0 \leq x < 6\}.$$

`ApplicationComponent.Size` is measured as the number of non-commented lines of code (NLOC). If C is an `ApplicationComponent`, then $\mathrm{V}(C.NLOC) = \mathrm{N}$. The number of non-commented lines of code is given as evidence in the model.

`ApplicationComponent.GearingFactor`. If C is an `ApplicationComponent`, then

$$\mathrm{V}(C.GearingFactor) = \mathrm{N}$$

The gearing factor is given as evidence in the model.

The `ApplicationCollaboration` class contains the attribute Coupling (five different types).

If I is an application interaction, X and Y are both application components, and $\{X, Y\} \subseteq I.communicates$, then they have a:

`ApplicationCollaboration.R5_ContentCoupling` relation if X refers to the internals of Y , i.e., it branches into, changes data, or alters a statement in y.

$$\mathrm{V}(R5\_ContentCoupling) = \mathrm{N}.$$

`ApplicationCollaboration.R4_CommonCoupling` relation if X and Y refer to the same global variable.

$$\mathrm{V}(R4\_CommonCoupling) = \mathrm{N}.$$

`ApplicationCollaboration.R3_ControlCoupling` relation if X passes a parameter to Y that controls its behavior.

$$\mathrm{V}(R3\_ControlCoupling) = \mathrm{N}.$$

`ApplicationCollaboration.R2_StampCoupling` relation if X passes a variable of a record type as a parameter to Y , and Y uses only a subset of that record.

$$\mathrm{V}(R2\_StampCoupling) = \mathrm{N}.$$

`ApplicationCollaboration.R1_DataCoupling` relation if X and Y communicate by parameters, each one being either a single data item or a homogeneous set of data items that does not incorporate any control element.

$$\mathrm{V}(R1\_DataCoupling) = \mathrm{N}.$$

The number of content, common, control, stamp, and data couplings are given as evidence in the model.

Guidelines for use

To use the modifiability viewpoint follow this process: Firstly, model the application components, how these collaborate, and what services they provide. Suitable respondents are application or solution architects, system owners and developers. Secondly, elicit attribute data for coupling, size, and programming languages. This information is usually found by interviewing developers and/or by investigating the source code through their development tools. Thirdly, run the analysis.

A modifiability view example

At ACME Energy, fictive company, the application architect is about to choose between two application architecture solutions delivering the same application service for the company, namely Study Maintenance. The company is currently in a state where a lot of changes are being implement, thus having a flexible IT environment is important for the CIO. A key aspect to consider when choosing between the two architecture solutions is therefore the modifiability of the application services. The architect models the two scenarios and finds solution A (cf. Figure 24) to have higher modifiability than solution B. Regarding architecture flexibility the information the architect passes on to the CIO is that solution A is better.

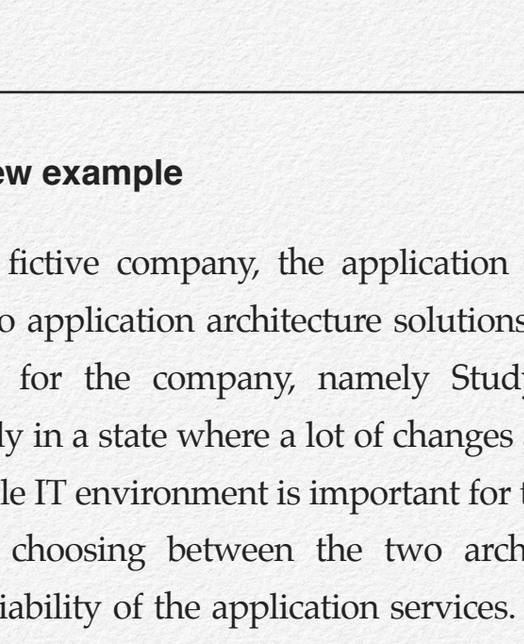


Figure 24: *A modifiability view example.*

In order to model architecture solution A (cf. Figure 24) of ACME Energy follow these steps:

1. Add four `ApplicationComponents` and name them:
   - CMMS (Computerized Maintenance Management System)
   - Asset Mgmt (Asset Management)
   - SCADA (Supervisory Control And Data Acquisition)
   - BI (Business Intelligence)
2. Add the following Size (NLOC) and `GearingFactor` (Programming language) information to the component attributes:
   - CMMS = 251.149 & 53 (Java)
   - Asset Mgmt. = 63.987 & 50 (ASP)
   - SCADA = 784.627 & 55 (C++)
   - BI = 52.345 & 53 (Java)
3. Add three `ApplicationCollaborations` with associated Coupling types, as in Table 4.
4. Press calculate and study the calculated coupling values for the four components. For the CMMS component the `ExternalCouplingAVG` should read 5.2 and `ExternalCouplingMAX` 5.9.

	No. of couplings between CMMS and		
Coupling type	**Asset Mgmt.**	**SCADA**	**BI**
Content coupling	5	0	0
Common coupling	1	1	2
Control coupling	1	5	2
Stamp coupling	0	1	0
Data coupling	10	2	2

Table 4: *Application collaboration information for modifiability view example.*

5. Add two Application Functions and name them:
   - Generate Failure Statistics
   - Compile Maintenance KPIs
6. Connect the two Application Functions with the CMMS component.
7. Add an Application Service and name it Study Maintenance. The service is Realized by both components, thus the relation should be of type *RealizeAND*. The relation should read Study Maintenance *realizerAND* Generate Failure Statistics.
8. Add Gearing Factor 53 (for programming language Java) as the baseline language for the gearing factor based `Size` calculation (ENLOC).
9. Press calculate and study the values for the Study Maintenance application service; the `Complexity` is calculated to be 6, Size 251.149, and `InternalCouplingMAX` 5.9. This leads to a `Modifiability` level of 4, which is generally considered to be a low score. However, obviously better then the assessed level of scenario B.

Data accuracy

Additional author: Per Närman

A description of MAP's data accuracy viewpoint.

5

Chapter 5 Data accuracy

Poor data quality in information systems can cause great economical impact resulting in costs of billions of dollars [48]. Analyzing the quality of data is therefore of great importance. Data is by IEEE defined as a representation of facts, concepts, or instructions in a manner suitable for communication, interpretation, or processing by humans or by automatic means [38]. The most common dimensions of data quality are completeness, consistency, currency, relevance and accuracy [49]. In this section data accuracy has the main focus. Accuracy is by IEEE defined as; (1) a qualitative assessment of correctness, or freedom from error, (2) a quantitative measure of the magnitude of error [38]. According to Redman [49] and Batini & Scannapieco [50] accuracy is defined as the closeness of a value V to V', where V' is an actual concept in the domain of reference and V is a datum that represents it. Depending on the accuracy requirements, the frame of reference V' can consist of an interval, e.g. 200 plus or minus one, such that a datum V with a value within the interval of 199-201 can be considered accurate whereas a datum V with the value 203 would be inaccurate.

Sound decisions are made based on accurate data. Unfortunately it does not matter if the data is used in a manual business process or within an automated application service. Neither is executed flawless at all time, resulting in steadily deteriorating data accuracy. The further away from the source the data gets, the poorer its accuracy becomes.

The original work that this section is based on can be found in [51, 52].

Movie 6: *Data Quality*

5.1 How to measure data accuracy

There are several modeling techniques proposed to capture quality characteristics of data. The traditional relational model has built in functionality to enforce some sort of data quality checks such as data integrity functions to minimize duplication of data or field data checks to enforce some sort of format on data. Other than these basic functions, data that is stored in systems based on the relational model are assumed to be accurate by design.

To remedy this limitation, the Quality Entity Relationship (QER) model extends the classical Entity Relationship (ER) model to accommodate several data quality dimensions on relations and fields [53]. The attributes in relations are associated with quality indicators (e.g. accuracy) and quality rankings (e.g. excellent). The QER model lacks the ability to incorporate information on the origin of the data. Tracing the origin of data is known as data provenance [50] and is important in most systems where data is collected from distributed sources with different data quality.

One of the earliest attempts in tracing data provenance is the Polygen model [53] which is more geared to analyze data quality in distributed heterogeneous data sources. The Polygen model is a relational model which defines a set of operators (e.g. union, Cartesian product etc) based on relational algebra that can semantically annotate the propagation of data. This is done by 'source tagging' data from multiple databases. The idea is essentially that the user or consumer of the data could judge the credibility of the data based on knowing from where it originated. The Polygen model was later extended to accommodate quality attribute associated with relations and fields, much like the QER model, but as an implementation on the relational algebra. According to [53] this extended the user's or consumer's ability to judge the credibility of the data by also allowing better interpretation and determination of the believability of the data.

While the QER and Polygen models are expressive and applicable to the database relational domain, they are less applicable to distributed enterprise information systems. In such systems the data does not only originate from database sources, but could also be collected in semi-structured format from business processes. Furthermore the structure of the data is not necessarily defined according to the relational model and could be more the result of an aggregation of data from one or several processes.

Information Product Maps (IP Maps) [54] accommodate such process based modeling of data. IP Maps are graphical models that treat data as a product from processes where the input is raw data, analogous to manufacturing processes where raw materials are inserted and a product exits the manufacturing line after being manipulated. Such manipulations can be represented in the IP Maps model with the use of specific constructs to represent the process and actions on the data, e.g. processing, quality check, or data receiver. Each construct has associated metadata which can be used in a model to specify the construct. The strength of the IP Maps is the ability to portray the data provenance as well as the elements in the process that manipulate the data.

The IP Maps model has been extended into an IP UML profile [55] to make use of UML's richer semantics. In the UML model the data units are associated through a quality association with a stereotyped quality dimension class e.g. timeliness. By using UML the IP UML provides the opportunity to model data units in interaction diagrams and to observe the data flow between object calls in activity diagrams. Neither IP Map nor IP UML can perform quantitative analysis of data accuracy in a process. Instead, their function is mostly to visualize data quality problems and aid system design.

A quantitative analysis method was proposed in [56]. The method known as the Data Flow/Process Method, is an analytical model which utilizes data flow diagrams to illustrate how the data quality of numerical data objects is affected by the data processing that takes place in applications. With this approach it is possible to trace data quality in applications. The limitation of this method is that it only considers numeric data and cannot accommodate alphanumeric data.

Another quantitative method from the accounting community was proposed by [57]. This method is not exclusively concerned with numeric data and can be used to show how by incorporating various improvement feedback cycles in processes, the overall quality of data could be improved. [57] also suggested an algorithm for assessing the general quality of data called 'reliability'. While overcoming the shortcoming of only being able to treat numerical data, and introducing the ability to reason about improvements of data the method is not intended to analyze the data quality across business processes.

The assessment method that is presented in this book is similar to the IP Maps and the IP UML model in the sense that it is aimed at the management of information systems by visualizing processes and data quality. Furthermore, our method uses the general approach of Data Flow/Process method to show how data quality, and specifically the accuracy dimension, in processes evolves. In following [57], the method does not confine itself to treating only numerical data.

5.2 The data accuracy viewpoint

This section describes the data accuracy viewpoint, cf. Figure 25. The viewpoint has the following main concepts:

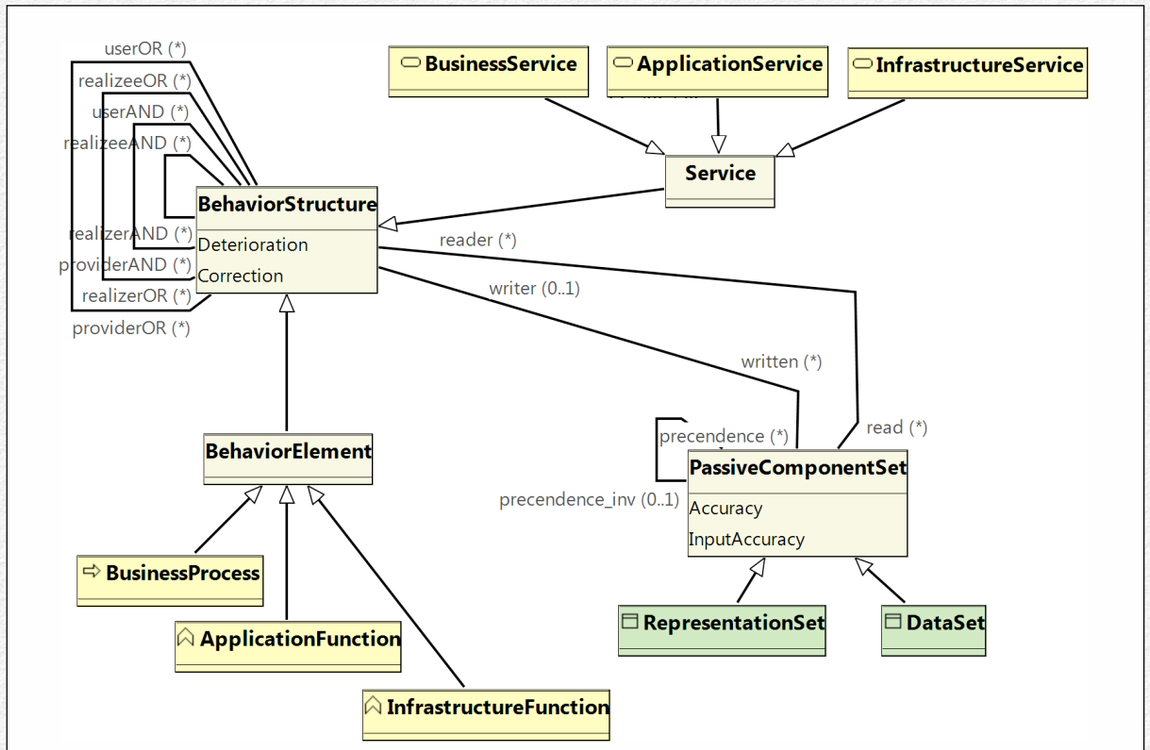


Figure 25: *The data accuracy viewpoint.*

- Service
  - BusinessService
  - ApplicationService
  - InfrastructureService
- PassiveComponentSet
  - DataSet
  - RepresentationSet
- BehaviorElement
  - BusinessProcess
  - ApplicationFunction
  - InfrastructureFunction

Concerns

Using the data accuracy viewpoint makes it possible to estimate the accuracy of data sets within the organization. It is also possible to determine which applications or business processes that introduce errors into the data sets.

Stakeholders

Obvious stakeholders are data custodians, i.e. those in charge of maintaining data quality, but also end users wishing to know the quality of the data which they use in their daily activities.

Theory

The data accuracy viewpoint employs process modeling in a manner similar to that of IP maps and that of [56]. Furthermore, following [57], the viewpoint also shows how data can improve when manipulated in business processes.

`The PassiveComponentSet` is used to describe sets of information objects whether stored in databases (then specialized into `DataSets`) or as more unstructured information (specialized into `RepresentationSet`). The attribute `PassiveComponentSet`.*Accuracy* is defined below.

Firstly, we denote the individual `Representations` and `DataObjects` `PassiveComponentObjects`. Next, we introduce the following:

N: Number of `PassiveComponentObjects` in the `PassiveComponentSet`.

N^{acc}: Number of accurate `PassiveComponentObjects` in the `PassiveComponentSet`.

N^{inacc}: Number of inaccurate `PassiveComponentObjects` in the `PassiveComponentSet`.

where "accurate" or "inaccurate" for the `PassiveComponentObjects` is defined as their value V being sufficiently close to the true value V' in line with [49, 50]. Since `PassiveComponentObjects` can be either accurate or inaccurate we have:

$$N^{acc} + N^{inacc} = N \quad (1)$$

The accuracy of the `PassiveComponentSet` can then be defined as

$$PassiveComponentSet.Accuracy = \frac{N^{acc}}{N} \quad (2)$$

The number of accurate `PassiveComponentObjects` in a `PassiveComponentSet` may change when processed by a `Function` or a `Service`. These may corrupt a `PassiveComponentObject`, which was accurate at process step $T = t$ into being inaccurate at time step $T = t + 1$. To be able to reason about this we introduce Ndet: the number of accurate PassiveComponentObjects at process step $T = t$, which were made inaccurate by a Function or Service at process step $T = t + 1$.

The frequency of this happening is

$$\alpha = \frac{N^{det}}{N_t^{acc}} \quad (3)$$

Similarly, a `Function` or a `Service` may correct inaccurate `PassiveComponentObjects`. We introduce N^{corr} : the number of `PassiveComponentObjects` that were inaccurate at process step $T = t$, but made accurate by a `Function` or a `Service` at time step $T = t + 1$.

$$\beta = \frac{N^{corr}}{N_t^{inacc}} \quad (4)$$

The number of accurate objects at process step $T = t + 1$ is given by:

$$N_{t+1}^{acc} = N_t^{acc} - N^{det} + N^{corr} \quad (5)$$

From the above an expression of the accuracy of a `PassiveComponentSet` at $T = t + 1$ can be derived:

$$PassiveComponentSet.Accuracy_{t+1} = \frac{N_{t+1}^{acc}}{N} =$$

$$= \frac{N_t^{acc}}{N} - \frac{N^{det}}{N} + \frac{N^{corr}}{N} =$$

$$= \frac{N_t^{acc}}{N} - \frac{\alpha * N_t^{acc}}{N} + \frac{\beta * N_t^{inacc}}{N} =$$

$$= \frac{N_t^{acc}}{N} * (1 - \alpha) + \frac{\beta(N - N_t^{acc})}{N} =$$

$$= \frac{N_t^{acc}}{N}(1 - \alpha) + \beta(1 - \frac{N_t^{acc}}{N}) =$$

$$= PassiveComponentSet.Accuracy_t * (1 - \alpha) +$$

$$\beta * (1 - PassiveComponentSet.Accuracy_t) \quad (6)$$

The data accuracy viewpoint can be found in Figure 25. The properties α and β are found as attributes Function.Correction, `Function.Deterioration`, `Service.Deterioration` and Service.Deterioration. Whenever a `PassiveComponentSet` is read or written by a Service or Function these attributes either improve or deteriorate the `PassiveComponentSet.Accuracy`. The attributes deterioration and correction are both given as evidence x in the model, where$\{x \in \mathrm{R} : 0 \leq x \leq 1\}$

`PassiveComponentSet.InputAccuracy` is an attribute used to specify the baseline accuracy of the first `PassiveComponentSet` in the process. The input accuracy attribute is given as evidence x in the model, where $\{x \in \mathrm{R} : 0 \leq x \leq 1\}$.

Guidelines for use

To use the data accuracy viewpoint follow this process: Firstly, model the data flow qualitatively. Suitable respondents are those performing the process who understands the process side of the flow and/or system architects who understand the application architecture. Secondly, elicit parameters input accuracy, deterioration, and correction from the same respondents. Thirdly, run the analysis.

A data accuracy view example

At ACME Energy, our fictive energy company, the analysts decide to investigate whether the reason Computerized Maintenance Management System (CMMS) users hold the application to a low esteem is due to poor data accuracy in the information provided by the application.

One important piece of information used when compiling the maintenance Key Performance Indicators (KPI's) is the “failure description" that the maintenance workers use in order to report what caused a failure in a piece of equipment. This is reported as a part of closing the work order, which was issued when the failure was first detected. Eliciting estimates of the correction and deterioration attributes of the processes and services involved in creating the maintenance KPIs was done through interviews, as well as eliciting the input accuracy of the initial work orders. Using these estimates in the modeled accuracy view, cf. Figure 26, it was estimated that the accuracy of the output Maintenance KPI's (with respect to failure statistics) was 87.8. This is a low number, and in order to improve the perceived usefulness of the application increasing this number might be a viable option.

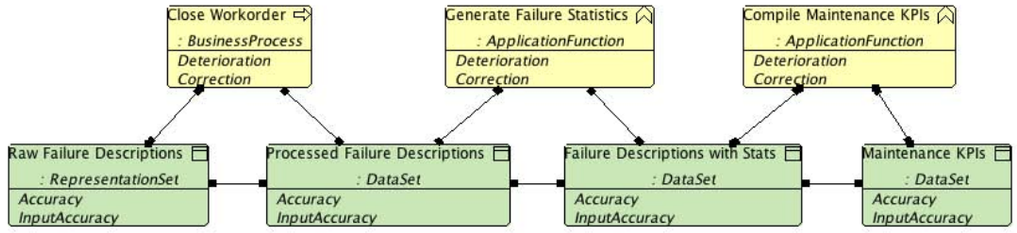


Figure 26: *A data accuracy view example.*

In order to model the data accuracy view example (cf. Figure 26) of ACME Energy follow these steps:

1. Add a Business Process class and name it Close Workorder. The Correction and Deterioration of this process are both 0.1, add this as attribute evidence.

2. The Close Workorder process Reads from a Representation Set called Raw Failure Descriptions. Add this class and relationship.

3. The Raw Failure Descriptions has an Input Accuracy of 0.98, add this as attribute evidence.

4. The CloseWorkorder processWrites to a Data Set called Processed Failure Descriptions. Add this class and relationship.

5. Press calculate, the Accuracy of the Processed Failure Descriptions is calculated to be 0.88. This means that the Close Workorder process has decreased the data accuracy of the failure descriptions.

6. Add two Application Functions and name them Generate Failure Statistics and Compile Maintenance KPIs (if you have modeled the modifiability example of the previous chapter re-use the application functions instead of creating new ones).

7. Both application functions have a Correction of 0.05 and Deterioration of 0.01. Add this as attribute evidence.

8. The Generate Failure Statistics function Reads from the Processed Failure Descriptions and Writes to a Data Set called Failure Descriptions with Stats. Add these two relationships and this class.

9. The Compile Maintenance KPIs function Reads from the Failure Descriptions with Stats and Writes to a Data Set called Maintenance KPIs. Add these two relationships and this class.

10. Press calculate and follow the Accuracy. See how it decreases from 98 % (Input Accuracy), to 88,4 % when processed, to 88,1 % after the statistics function, to finally 87,8 % when compiled to KPIs.

Application usage

Additional author: Per Närman

A description of MAP's application usage viewpoint.

6

Chapter 6 Application usage

Modern organizations have large application portfolios comprising hundreds if not thousands of applications. Despite the very large investments these portfolios represent, realizing their full value often proves to be elusive [58]. An important problem encountered by most organizations is the uncontrolled proliferation of applications leading to a heterogeneous application portfolio with redundant functions and data, high information system costs and poor business-IT alignment [59]. Thus, there is a case for structured application portfolio and landscape management [60] to support decision-making concerning changes to the application portfolio. Such rational decision-making calls for means to assess the value of the individual applications [61].

Delone and McLean introduced a six dimension model of information systems value [62, 63]. One of these dimensions is system usage. System usage has been found to explain business performance [64]. Similarly, [65] introduced system usage as one of five important parameters in assessing application portfolio health.

The original work that this section is based on can be found in [66].

6.1 How to measure application usage

For the past couple of decades, two theories have reigned supreme in explaining system usage; the Technology Acceptance Model (TAM) [67-75] and the Task-Technology Fit model (TTF) [76-83]. TAM is built around the two constructs Perceived Usefulness (PU) and Perceived Ease of Use (PEoU), and the TTF model on the idea that having a good match of functional capabilities and task requirements leads to higher usage.

The Technology Acceptance Model

The technology acceptance model (TAM) is arguably the most influential theory in information systems research. Originally proposed by [67] and drawing heavily on the psychological research field, the TAM suggests that the usage of information systems can be determined by two factors; the perceived usefulness and the perceived ease of use of the information system, Figure 27.

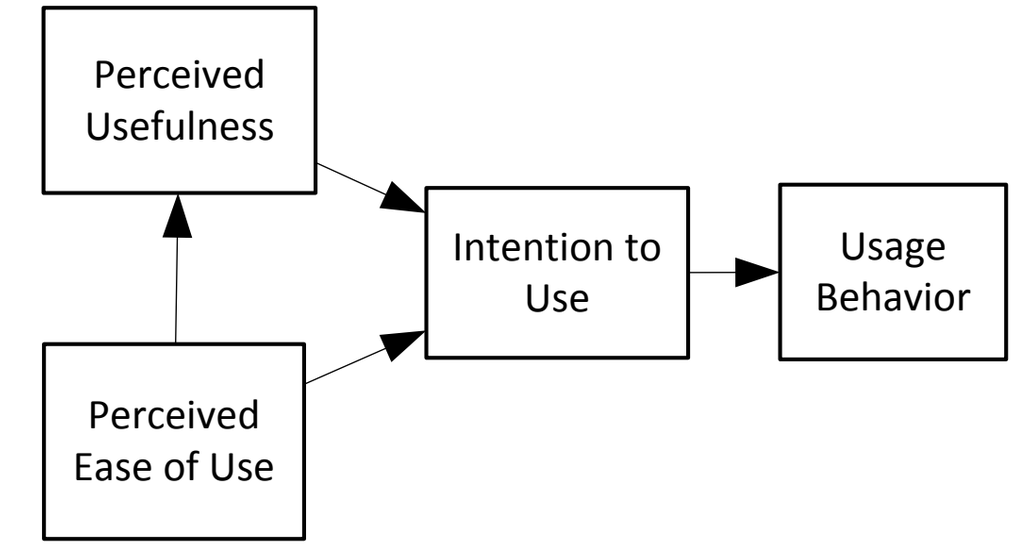


Figure 27: *The Technology Acceptance Model.*

The antecedents of both these constructs have been further deliberated by [73-75] and there are an overwhelming number of studies published where the TAM is used in different contexts and with slight variations of variables, see e.g. [68-72] and more. These studies have established conclusively that at least the perceived usefulness construct is an important variable in determining user acceptance and usage of a technology [84] with the

perceived ease of use variable having significant relations to the perceived usefulness as well as a smaller, yet important impact on system usage [84].

Task-Technology Fit

In this book we focus on the Task-Technology Fit model (TTF) for assessing the usage of applications. TTF is built on the idea that if the users perceive a technology to have characteristics which fit the user's work tasks, then i) the user is more likely to utilize the technology and ii) to perform the work task better. Figure 28 shows a basic TTF model.

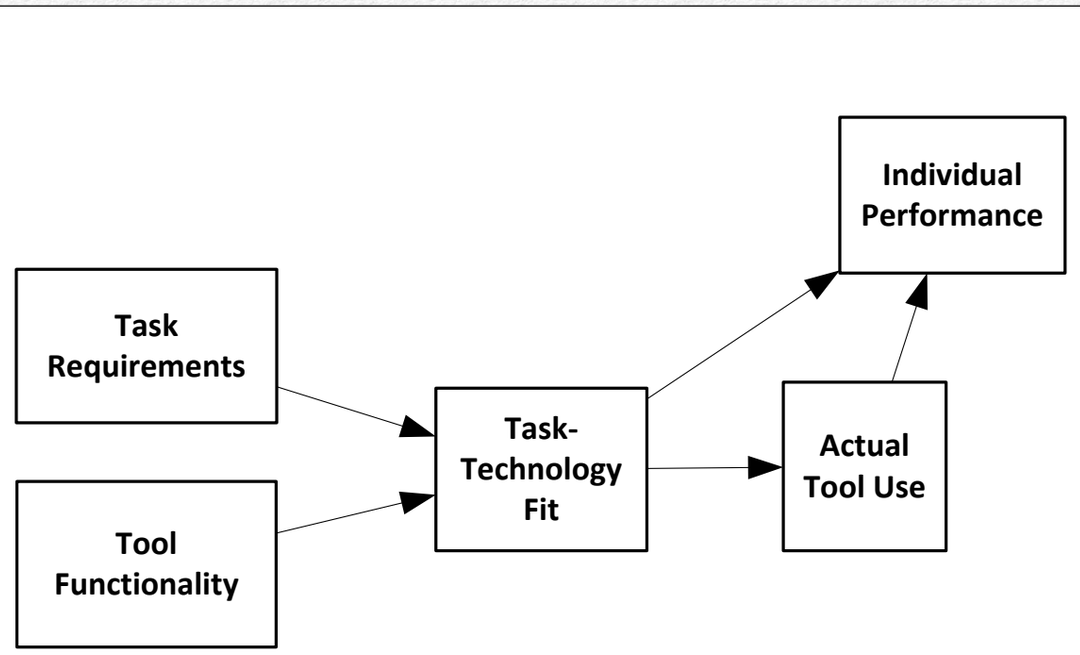


Figure 28: *A basic Task-Technology Fit (TTF) model.*

Several studies have used the TTF model to predict user performance and technology utilization see e.g. [77-83].

The original TTF model proposed by Goodhue [76] described the constructs task characteristics, tool characteristics and task-technology fit in general terms stating that task technology fit can be measured in terms of eight factors. The link between TTF and utilization was found to be rather weak using this setup.

Dishaw and Strong [79] used the concept of strategic fit as interaction [85] (meaning multiplying the values of task requirements with those of IT functional fulfillment) to operationalize TTF for a specific domain viz. computer software maintenance. Instead of defining a general notion of TTF they operationalized task and tool characteristics based on previously published reference models of computer maintenance tasks [86] and tool functionalities [87].

The assessment method for application usage presented in this book uses the Task-Technology Fit model together with enterprise architecture modeling and analysis.

6.2 The application usage viewpoint

This section describes the application usage viewpoint, cf. Figure 29. The viewpoint has the following main concepts:

- Service
  - ApplicationService
- BehaviorElement

  - BusinessProcess
  - ApplicationFunction
- ActiveStructureElement
  - ApplicationComponent
- UsageRelation
- ProcessServiceInterface

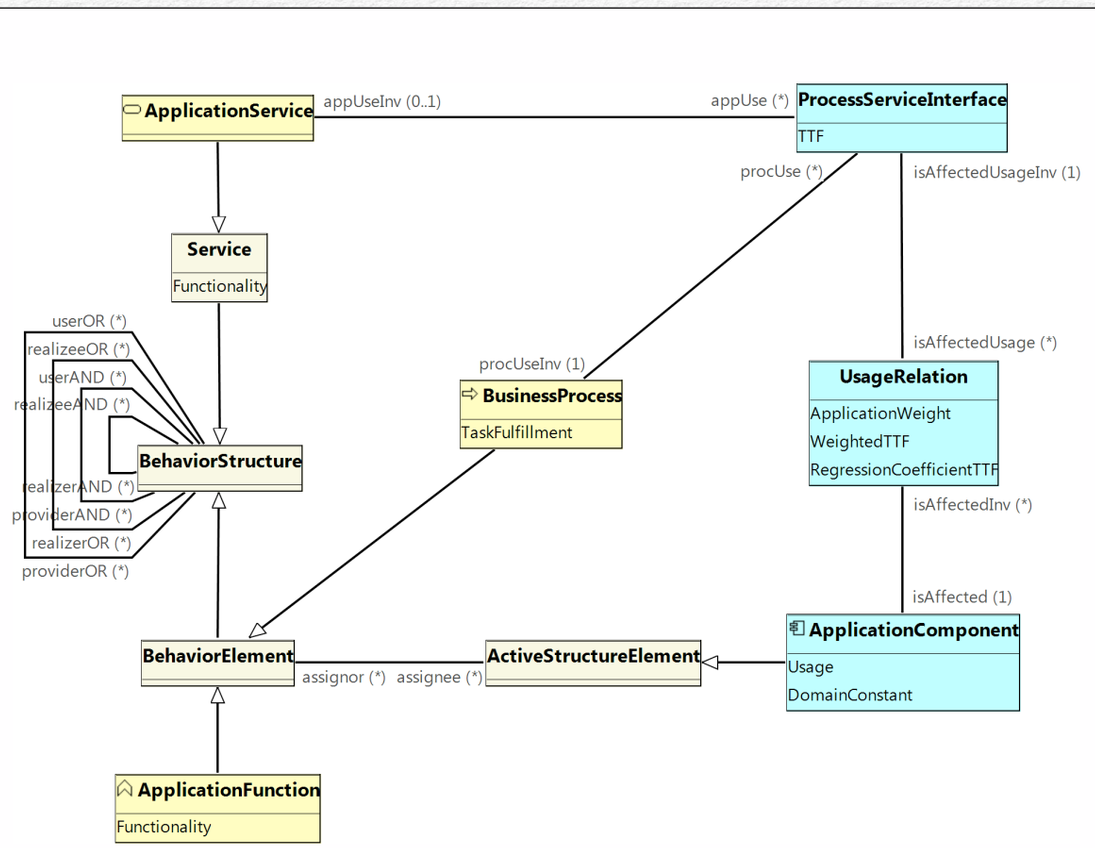


Figure 29: *The application usage viewpoint.*

Concerns

Why do users voluntarily embrace certain applications and object to using others? Voluntary application usage is a very important indicator of the quality of the application portfolio [65].

Stakeholder

The stakeholders are those interested taking a top-down perspective on the application portfolio. These may include enterprise architects, application architects, and ultimately the organization's CIO.

Theory

Application usage is in this book evaluated as task technology fit. The application usage viewpoint needs to be tailored to its application domain. The tailoring involves defining and operationalizing the domain's tasks, IT functionality, how the IT functions support the tasks (i.e. the TTF variables). The viewpoint is presented in Figure 29. The aim of employing the viewpoint is to obtain a value for the ApplicationComponent.Usage attribute.

`ApplicationService.Functionality` The functionality attribute is evaluated based on the functionality of the application functions which realize the service. If $AF = \{af_1, \ldots, af_n\}$ is a list of application functions, S is a `Service`, and $AF \subseteq S.realizedByAND \cup S.realizedByOR$ then

$$f(S.Functionality) = \sum_{i=1}^{n} af_i.Functionality.$$

$$\mathrm{V}(ApplicationService.Functionality) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`ApplicationFunction.Functionality` The functionality of the application function is set as an evidence in the model. The functionality of the application function is quantitatively assessed by using the mean of user assessments on a Likert scale.

$$\mathrm{V}(ApplicationFunction.Functionality) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`ApplicationComponent.DomainConstant` The domain constant of the application component is set as evidence in the model. The domain constant is derived by creating a linear regression model from the user elicited data out of which the adjusted coefficient of determination $\bar{R}^2$ is used as the evidence. Even though $\bar{R}^2$ can give a negative result only positive numbers are allowed in the model.

$$\mathrm{V}(ApplicationComponent.DomainConstant) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`ApplicationComponent.Usage` The usage of an application component is evaluated based on the domain constant and the weighted TTF values in the usage relationships which affect the application component. If $U = \{u_1, \ldots, u_n\}$ is a list of usage relations, C is an application component and $U \subseteq C.isAffectedInv$ then

$$f(C.Usage) = C.DomainConstant + \sum_{i=1}^{n} u_i.WeightedTTF.$$

$$\mathrm{V}(ApplicationComponent.Usage) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`BusinessProcess.TaskFulfillment` The task fulfillment of the business process is set as evidence in the model. The task fulfillment value is derived by taking the mean value of task fulfillment assessments by application users. Task fulfillment can be evaluated using a Likert scale.

$$\mathrm{V}(BusinessProcess.TaskFulfillment) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`ProcessServiceInterface.TTF` The task technology fit attribute in the process service interface class is evaluated based on the functionality of the connected application service and business process. If s is an application service where $ProcessServiceInterface.appUseInv = s$ and u is an usage relations where $ProcessServiceInterface.procUseInv = u$ then

$$f(ProcessServiceInterface.TTF) = s.Functionality * u.TaskFulFillment$$

$$\mathrm{V}(ProcessServiceInterface.TTF) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`UsageRelation.RegressionCoefficientTTF` The regression coefficient TTF attribute in the usage relation class is set as evidence in the model. The regression coefficient is set to show the domain specific strength of the `ProcessServiceInterface.TTF` on `Application.Component.Usage.`

$$\mathrm{V}(UsageRelation.RegressionCoefficientTTF) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`UsageRelation.ApplicationWeight` The application weight attribute in the usage relation is evaluated by dividing the application functions functionality with the application service functionality. If U is an usage relation, $F = \{f_1, \ldots, f_n\}$ is a list of application functions where $F \subseteq U.isAffected.assignor$ and s is an application service where $U.isAffectedUsageInv.appUseInv = s$ then

$$f(UsageRelation.ApplicationWeight) = \frac{\sum_{i=1}^{n} f_i.Functionality}{s.Functionality}.$$

$$\mathrm{V}(UsageRelation.ApplicationWeight) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`UsageRelation.WeightedTTF` The weighted TTF attribute is evaluated by multiplying the application weight with the regression coefficient TTF and TTF value from the process service interface. If u is the usage relation p is an process service interface where $u.isAffectedUsageInv = p$ then

$$f(UsageRelation.WeightedTTF) = p.TTF * u.ApplicationWeight * u.RegressionCoefficientTTF.$$

$$\mathrm{V}(UsageRelation.WeightedTTF) = \{x \in \mathrm{R} : 0 \leq x\}.$$

Guidelines for use

In the case the organization does not have reference models for tasks and functionality, these have to be developed in order to employ this viewpoint perhaps using the approach of [66]. Once the appropriate models are in place, however, the viewpoint may be employed as follows:

Firstly, compile list of all applications and processes relevant to the inquiry and generate the architecture view. These lists can be elicited by process managers or similar. Secondly, perform a survey with a sufficient subset of application users or process performers. For each function of the reference model, the respondents are asked to name the application that implements the function the most and to what degree. For all tasks of the task reference model, the respondents are asked to rate the degree to which they perform the activities.

An application usage view example

The ACME Energy CIO has ordered an exploratory study of the quality of ACME Energy's application portfolio. The application usage viewpoint was employed to determine which applications users liked and would use voluntarily. Here we model one of the applications, the Computerized Maintenance Management System (CMMS).

In the view of Figure 30 we see the single `ApplicationComponent` CMMS which offers two `ApplicationFunctions` *Generate failure statistics* and *Compile maintenance KPIs* which realize an `ApplicationService` *Study Maintenance* which in turn supports a `BusinessProcess` with the same name. The functionality and task fulfillment of the `ApplicationFunctions` and the `BusinessProcess`

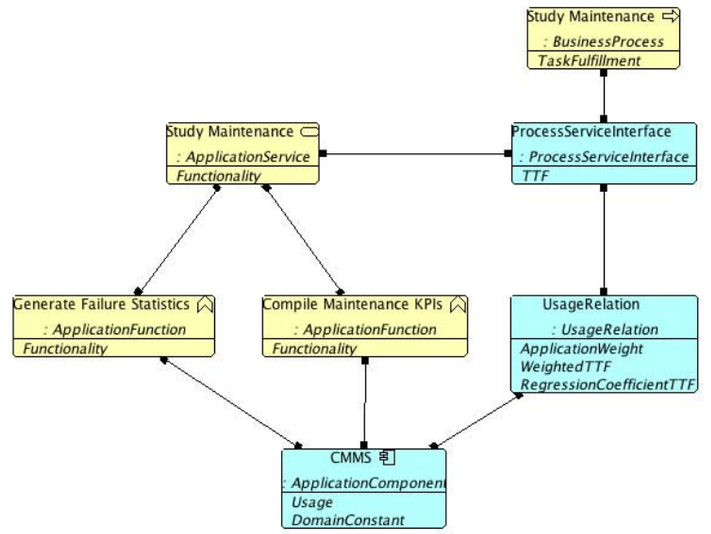


Figure 30: *An example application usage view.*

were found by taking the mean of user's assessments. It was discovered that the users considered the CMMS to be all right functionality-wise, which together with a high degree of `BusinessProcess.TaskFulfillment` yielded a high `ProcessService Interface.TTF` for the interface between the *Study* `ApplicationService` and `BusinessProcess`.

In order to model the application usage view example (cf. Figure 30) of ACME Energy follow these steps:

1. Add two Application Functions and name them Generate Failure Statistics and Compile Maintenance KPIs. Add one Application Service and name it Study Maintenance. Add one Application Component and name it CMMS (Computerized Maintenance Management System). All four classes have been used in previous view examples (Modifiability and/or Data accuracy), re-use these classes if possible.

2. The Study Maintenance service is Realized by both (AND) application functions, add these relationships. The two application functions are both Assignees of the CMMS component, add these relationships as well. These relationships are automatically generated if the classes are re-used from previous examples.

3. The CMMS component has a Domain Constant of 0.9, the Generate Failure Statistics has a Functionality of 4, and the Generate Failure Statistics 2.1. Add this as attribute evidence.

4. Add a Business Process and name it Study Maintenance.

5. The Study Maintenance process use the CMMS component via a Process Service Interface and a Usage Relation. Add these classes and relationships.

6. The Study Maintenance service and process are also related via the Process Service Interface. Add this relationship.

7. The Usage Relation has a Regression Coefficient TTF of 0.035 and an Application Weight of 1. Add this as attribute evidence.

8. Press calculate and study the Application Component Usage, it should read 2.18.

Service availability

Additional author: Per Närman

7

A description of MAP's service availability viewpoint.

Chapter 7 Service availability

Information systems availability is crucial in order to ensure continuous business operations [88] and as such rated very highly by IT system executives [89]. Not only are the direct costs of unavailable IT systems high [90], but IT incidents disrupting business operations also have an adverse impact on the market value of publicly traded companies [91]

The original work that this section is based on can be found in [92-94].

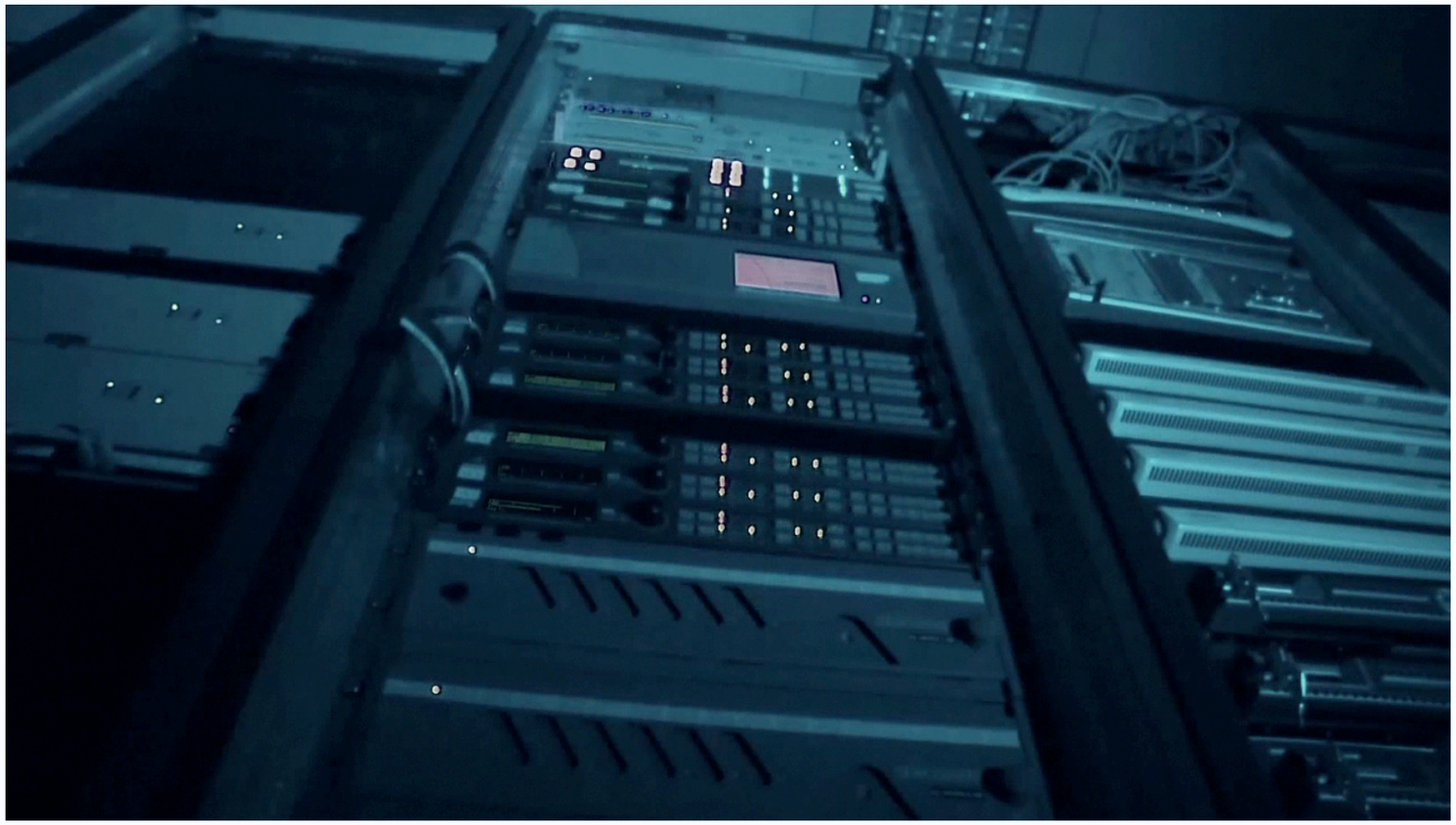
7.1 How to measure service availability

System availability analysis is a mature field and there is an abundance of availability analysis techniques including methods such as Reliability Block Diagrams, Fault Tree Analysis, Failure mode effects analysis, Markov processes and Bayesian analysis [95].

In the category of qualitative methods we find for example Failure Mode and Effect Analysis (FMEA) [96]. In FMEA, each system component's failure mode and its impact on the rest of the system is documented. The method is particularly useful for systems with single component failures. Thus the approach is not well suited for systems with a fair degree of redundancy [95].

An alternative approach is state-based analysis. State-based methods enumerate all possible system failure states and are not limited to stochastically independent failure of components. This expressiveness comes at a price: models for state-based analysis using Markov chains [97] grow exponentially with the number of system components [98, 99]. Hence, state-based analysis using Markov models contradicts the requirement of keeping the framework simple, both in terms of modeling size and with respect to the ease-of-use.

One of the most frequently adopted methods is Fault Tree Analysis (FTA), which translates the



Movie 7: *Availability*

failure behavior of a physical system into a visual diagram and a logical model [95]. The modeling structure of FTA allows the modeler to visualize the system architecture in terms of primary component's relational dependency on subcomponents [100]. Availability analysis using FTA is much similar to the approach based on reliability block diagrams (RBDs) [95, 101]. The concept behind RBD is to identify undirected relational paths between components within the architecture. First two nodes *s* and *t* are defined. Secondly, relational paths comprising system components between the nodes are identified. A system is said to be available if there exists at least one path comprising a chain of available components from *s* to *t*.

Both FTA and RBD are confined to analysis of static systems and do not take dynamic aspects like maintenance and repair into consideration [95]. However, both methods comply with the stated requirements with one exception; the analysis framework shall be aligned with general rules of modeling. RBDs allow multiple instantiations of single components to capture k-out-of-n structures, which would call for two separate modeling languages - one for architecture modeling and another for analysis using RBDs. FTA does not suffer from this problem and is the favored candidate method.

The assessment method presented in this book uses Fault Tree Analysis (FTA) together with enterprise architecture modeling and analysis.

Assumptions in Fault Tree Analysis

A first assumption in FTA is independence of failures among different component - implying that there are no common cause failures - which simplifies the modeling task [102]. However, if the same application is implemented on multiple nodes we model the application as one instance and thus a single failure affects each of them equally. This partly classifies as a common cause failure.

When dealing with redundancy of components we have the aspect of redundancy types. Depending on the situation and criticality of component's redundancy can be implemented with various properties such as active redundancy - components operating in parallel and sharing the load; and passive redundancy - the main component has all load and the reserve component is first activated once the main component fails. There are ways to deal with such property for example by using dynamic fault tree models [103]. We do, however, restrict the data gathering presented in this book to a single type of redundancy and leave that to future work.

When we consider components in the analysis another approximation made is that a repaired item is in an "as good as new" condition i.e. assuming perfect repair. If not, assuming a constant MTTF over infinite time will not be valid but instead the component would be in a different state after repair with a different probability of failure. In the ISO 9126-2 standard a similar assumption is stated implicitly [104].

Calculating average system availability using FTA

Availability is often defined as

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR}$$

where MTTF denotes "Mean Time To Failure" and MTTR "Mean Time To Repair", respectively. MTTF is the inverse of the failure rate of a component

$$MTTF = \frac{1}{\lambda}$$

and MTTR is the inverse of the repair rate of a component

$$MTTR = \frac{1}{\mu}$$

The average availability A_{avg} of a component can now be computed:

$$A_{avg} = \frac{\mu}{\mu + \lambda}$$

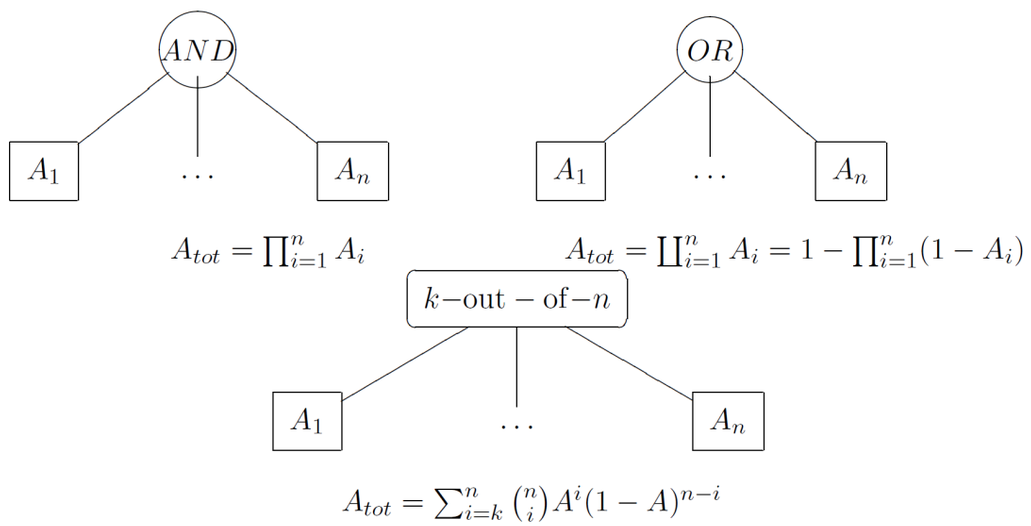


Figure 31: *The basic cases for parallel, serial and k-out-of-n systems, respectively.*

The quotient is easy to interpret as the time that a system is available as a fraction of all time.

Systems rarely consist of a single component. To model availability in complex systems, three basic cases are used as building blocks. These are used recursively to model more advanced situations. The basic cases are illustrated in Figure 31. All calculations assume independent component availabilities.

The AND case models systems where the failure of a single component is enough to bring the system down. The OR case models redundant systems where a single working component is enough to keep the system up. The k-out-of-n case models systems that are functioning if at least k components are functioning.

A simple example of how the building blocks and their mathematical equivalents are put together recursively is illustrated in Figure 32.

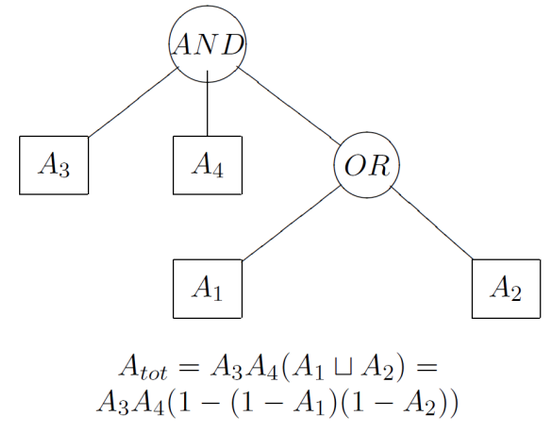


Figure 32: *A simple example of system availability calculations.*

7.2 The service availability viewpoint

This section describes the service availability viewpoint, cf. Figure 33. The viewpoint has the following main concepts:

- Service
  - BusinessService
  - ApplicationService
  - InfrastructureService
- BehaviorElement
  - BusinessProcess
  - ApplicationFunction
  - InfrastructureFunction
- ActiveStructureElement
  - ApplicationComponent
  - Node

Concerns

The service availability viewpoint addresses the concern of determining the availability of services in the present (as-is) and future (to-be) enterprise architecture.

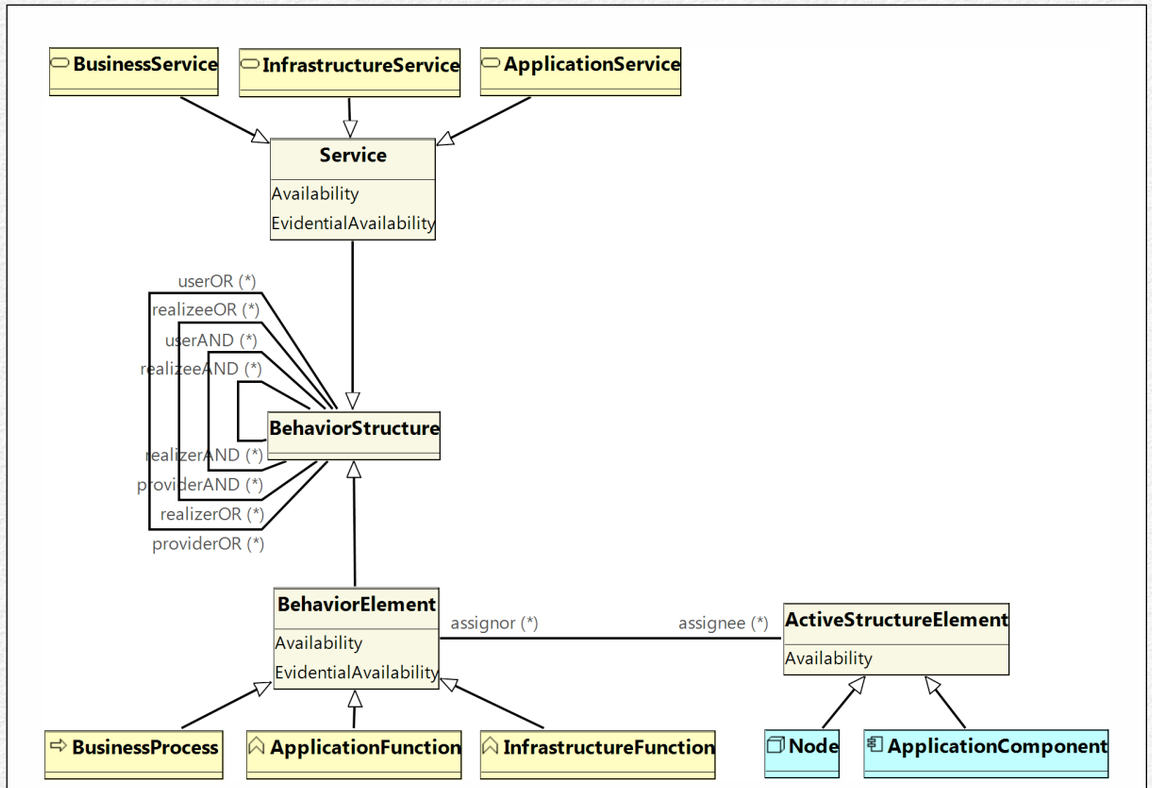


Figure 33: *The service availability viewpoint.*

Stakeholder

The stakeholders for the service availability viewpoint are service managers and end-users.

Theory

The service availability viewpoint utilizes Fault Tree Analysis (FTA) [100] for the availability analysis.

A first assumption in FTA is independence of failures among different components - implying that there are no common cause failures - which

simplifies the modeling task [102]. However, if the same application is implemented on multiple nodes we model the application as one instance and thus a single failure affects each of them equally. This partly classifies as a common cause failure. We make the assumption of passive redundancy, with perfect switching and no repairs, [95]. When we consider components in the analysis another approximation made is that a repaired item is in an "as good as new" condition i.e. assuming perfect repair. If not, assuming a constant MTTF over infinite time will not be valid but instead the component would be in a different state after repair with a different probability of failure. In the ISO 9126-2 standard a similar assumption is stated implicitly [104].

The service availability viewpoint can be found in Figure 33. The viewpoint incorporates FTA in line with the theory presented in the previous section. The behavior elements are represented by `Services` and BehaviorElements (or `Functions` for brief). Both of these have an availability which is represented in the `Service.Availability` and `Function.Availability` attributes respectively. The availability attribute gets a value x , where $\{x \in \mathrm{R} : 0 \leq x \leq 1\}$. A high value indicates that the service/function is more likely to be available.

`Services` are *realized* by `Functions` represented with the realize relationship available in both *AND* and *OR* options. Conversely, Functions use Services through the use relationship, also with both AND and OR options. The properties of the *AND* and *OR* options have been presented in section 3.1.1.

`Services` are merely externally visible containers of application behaviors and their availability is as such only dependent on the realizing `BehaviorElements`. `BehaviorElements.Availability` on the other hand depends also on the `ActiveResourceElement` to which it is assigned. When the `BehaviorElements` uses `Services`, `BehaviorElements.Availability` is the product of the `Services.Availability` and the `ActiveResourceElement.Availability`, since there is an implicit AND relationship between the underlying services and the application realizing the `BehaviorElements`. The `Availability` is calculated as shown in Figure 32 for the AND and OR options.

Sometimes, there is a need to set the availability directly on a `Function` or `Service`, and this can be done using the attribute `Function.EvidentialAvailability` or `Service.EvidentialAvailability` respectively. The attribute is given as evidence x in the model, where $\{x \in \mathrm{R} : 0 \leq x \leq 1\}$.

Guidelines for use

The following steps should be taken in order to use the service availability viewpoint.

Firstly, there is a need to identify and scope the service or services of interest, either from a service catalog or through interviews. Defining the service properly is essential to defining what the service being 'available' means. Secondly, use the viewpoint to qualitatively model the application and infrastructure architecture connected to the service. Thirdly, elicit quantitative measures of component availabilities. Usually, the easiest way

of obtaining the component availability is to ask the respondent (typically a system owner or similar) to estimate how often the component breaks down and estimate the repair time. Fourthly, run the analysis.

A service availability view example

To probe deeper into the rumors flourishing at ACME Energy regarding the incidents affecting the availability of the application service called study maintenance, the analysts performed an initial round of interviews with system administrators to obtain qualitative data concerning the architecture realizing the application service. This was modeled according to the service availability viewpoint described above. Quantitative data regarding component availabilities were collected during a second round of interviews.

In Figure 34 we see the result. The aggregated availability was found to be 98%, which is considered acceptable to most users. Thus, the analysts decide to scrutinize other aspects of the architecture.

In order to model the service availability view example (cf. Figure 34) of ACME Energy follow these steps:

1. Add the classes Study Maintenance (application service), Generate Failure Statistics (application function), Compile Maintenance KPIs (application function), and CMMS (application component). These should/could be re-used form previous modeling examples (modifiability, data accuracy, and application usage views).
2. The Study Maintenance service is Realized by both functions (AND), and both functions are Assignees of the CMMS component. Add these relationships. These relationships are automatically generated if you are re-using the classes from the previous examples.
3. The CMMS components has an Availability of 0.995. Add this as attribute evidence.
4. Add three Infrastructure Services and name them Generate GUI, Data Retrieval primary site, and Data Retrieval secondary site.

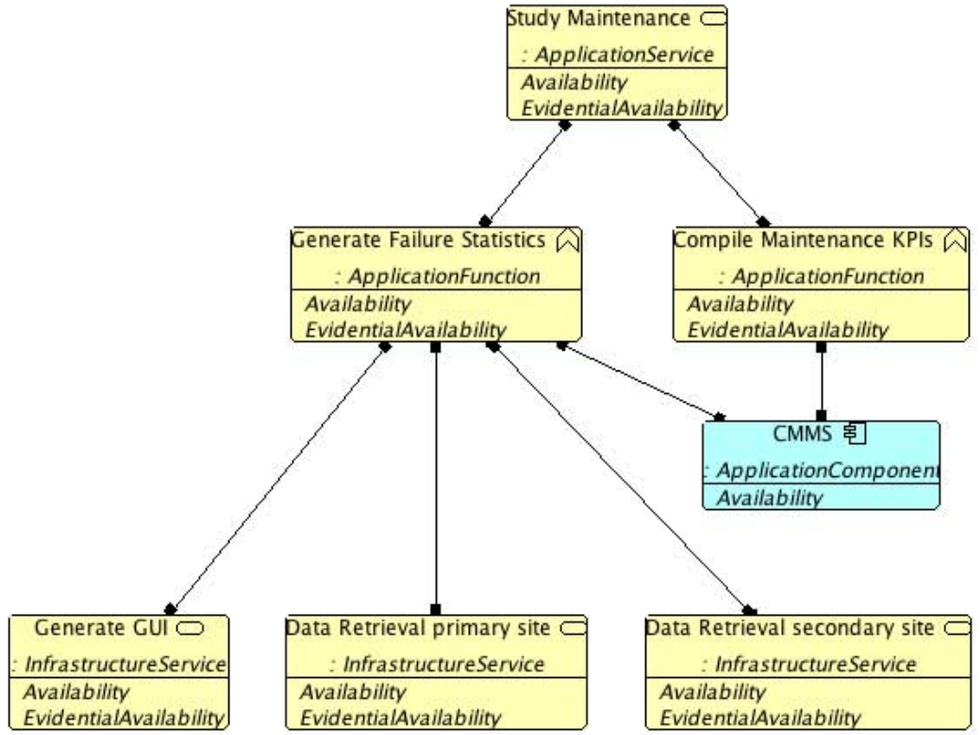


Figure 34: *A service availability view example.*

5. The Generate Failure Statistics function Use the Generate GUI and one of the Data Retrieval sites. That is there is an OR-relationship to the two Data Retrieval sites and an AND-relationship to the Generate GUI service. Add these relationships.

6. The Generate GUI has an Evidential Availability of 0.99, the primary site 0.992, and the secondary site 0.991. Add this as attribute evidence.

7. Press calculate, the analysis should declare that the Availability of the Study Maintenance application service is 98 %.

Interoperability

Additional author: Johan Ullberg

A description of MAP's interoperability viewpoint.

8

Chapter 8 Interoperability

Interoperability is a sought after quality for enterprises in today's competitive environment that has been approached from many different points of view and perspectives [105]. Several definitions of interoperability have been proposed, one of the most well known and the one employed in this book is that of IEEE [38], the ability of two or more systems or components to exchange information and to use the information that has been exchanged. Based on this definition interoperability can be seen from the perspective of a decision maker as the problem of ensuring the satisfaction of a set of communication needs throughout the organization.

The original work that this section is based on can be found in [23, 24, 106].

8.1 How to measure interoperability

Several initiatives on interoperability have proposed interoperability frameworks to structure issues and concerns in different ways. The European Interoperability Framework in the eGovernment domain [107] defines three aspects of interoperability: semantic, technical and organizational. A similar approach was also proposed in the e-Health interoperability framework [108] which identified three layers: organizational, informational and technical interoperability. The ATHENA Interoperability Framework (AIF) proposes to structure interoperability issues and solutions at the three levels: conceptual, technical and applicative [109]. The Framework for Enterprise Interoperability [110] is another interoperability framework that focuses on the problem dimension of interoperability. The objective is to tackle interoperability problems through the identification of barriers which prevent interoperability from occurring. This is done by defining a problem space as the

Movie 7: *Availability*

intersections of the two dimensions concerns and barriers. Together with a third dimension, approaches to mitigate the problems, the solution space is defined. Using the framework it is possible to classify interoperability knowledge. All these interoperability frameworks provide means to classify the interoperability problems and solutions. At the same time they lack the ability to describe the interoperability situations where the problems occur and are solved.

The Ontology of Interoperability (OoI) [111] is one approach towards a deeper understanding of interoperability than what is offered by the interoperability framework. OoI prescribes a set of metamodels to describe interoperability from various viewpoints, once again mainly aiming at classifying various problems and decision alternatives. OoI does however also provide a communication metamodel aimed at describing interoperability situations.

Several methods for assessing interoperability on a general scope have been suggested. The Levels of Information Systems Interoperability (LISI) [112], developed by MITRE and the C4ISR Integration task force, uses a maturity model for assessing interoperability ranging from the isolated level to the enterprise level. The assessment in LISI is based on an assessment process and utilizes a score card method and interoperability metrics. Employing LISI would require more domain knowledge in the field of interoperability than the assessment method presented in section 8.2. There are several similar approaches to LISI, such as Systems of Systems Interoperability (SoSI) [113] and Levels of Conceptual Interoperability Model (LCIM) [114] coupled with the same drawbacks as LISI.

The i-Score [115] is a methodology for quantitative interoperability assessment. The assessment is based on the concept of operational threads, a sequence of activities each supported by exactly one system. Such operational threads can be created from for instance an UML activity diagram [115]. For each pair of activities and their supporting systems in the thread an interoperability spin is calculated and then aggregated into an i-Score.

The interoperability analysis framework presented by Ullberg et al. [23, 24, 106] contains a number of classes and attributes for interoperability analysis e.g. Actor, Message-passing System, and Communication Need. The main goal is to assess is if the attribute Communication Need is satisfied or not (or to what degree).

The assessment method presented for interoperability in this book uses a simple check, namely whether a service, function, or data set share a common language for communication. This is evaluated in the architecture model by analyzing if two classes share a common language, we then assume that the communication need can be satisfied. Thus, the analysis used here is a simplified version of what Ullberg et al. presents.

8.2 The interoperability viewpoint

This section describes the interoperability viewpoint, cf. Figure 35. The viewpoint has the following main concepts:

- Service
  - BusinessService
  - ApplicationService
  - InfrastructureService
- PassiveComponentSet
  - DataSet
  - RepresentationSet
- BehaviorElement
  - BusinessFunction
  - ApplicationFunction
  - InfrastructureFunction
- Language

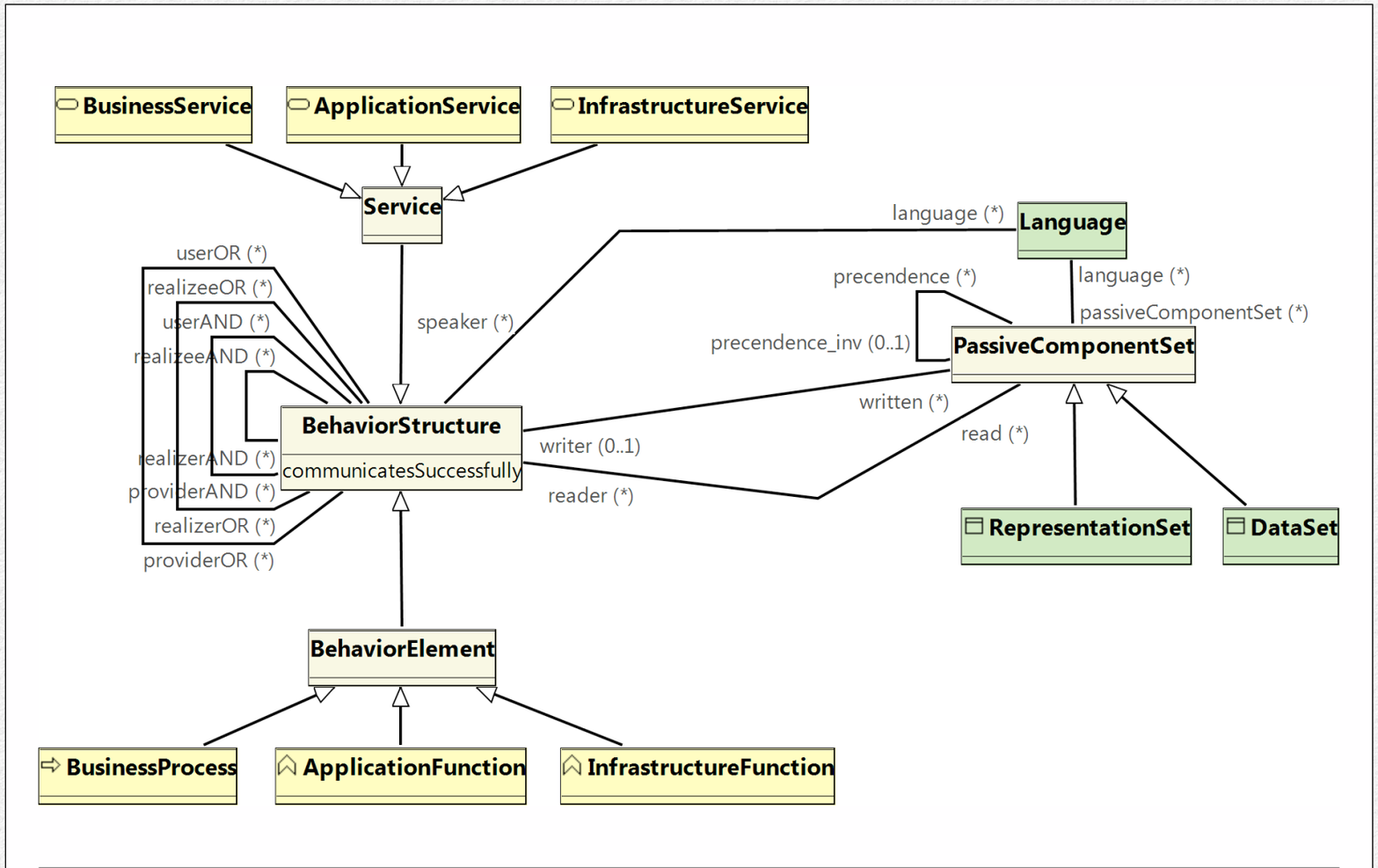


Figure 35: *The interoperability viewpoint.*

Concerns

Using the interoperability viewpoint makes it possible to determine whether data can be exchanged between services and functions without any loss due to deficiency in the communication. It also makes it possible to tell what services, functions, and data objects that speak a certain language.

Stakeholders

The typical stakeholder for the interoperability viewpoint is the integration and/or solution architect, but it could also come in use for developers implementing software that is integrated with other software.

Theory

Interoperability is assessed based on two assumptions: 1) Two entities are interoperable if the communication is successful between

these two. 2) The communication between two entities is successful if they share a common language.

If L is a Language, BE is an Behavior Element, PCS is a Passive Component Set, then

$$f(BE.CommunicatesSuccessfully) = \begin{cases} true & \text{if } PCS \subseteq BE.read \\ & \text{and } L \subseteq BE.language \cup \\ & PCS.writer.language \\ false & \text{otherwise} \end{cases}$$

Guidelines for use

To use the interoperability viewpoint follow this process: Firstly, model the services (business, application, and infrastructure), behavior elements (business processes, application functions, and infrastructure functions), and passive component sets (representation sets and data sets). These are all typical classes used in enterprise architecture thus finding this information can either be done by using existing models or by interviewing relevant stakeholders (e.g. enterprise architects, business architects, information architects, application architects, infrastructure/ technology architects). Secondly, find out what languages that are used related to all modeled classes and insert this information into the model. Thirdly, run the analysis.

An interoperability view example

The integration architect at ACME Energy is planning some major changes in the form of integrating some new functions. A first step for our architect is to model the As-Is state of the architecture, so that any future To-Be state can be analyzed with this as a baseline. The CIO has pointed out that one important part of their business is the compilation of maintenance KPIs based on the closed workorders. This includes

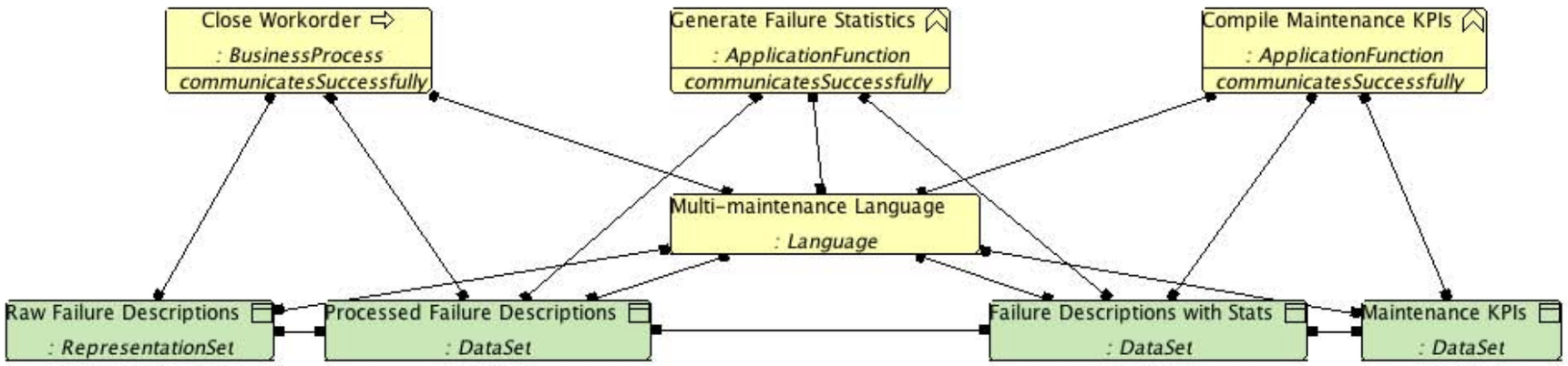


Figure 36: *An interoperability view example.*

several processes and functions that read and write to different data sets. In the current state everything is interoperable because they speak or are written in the same language. The as-is model verifies this, cf. Figure 36.

In order to model the interoperability view example (cf. Figure 36) of ACME Energy follow these steps:

1. Add a Close Workorder business process, a Generate Failure Statistics application function, and a Compile Maintenance KPIs application function. These have all been modeled in previous example, please re-use.

2. Add a Representation Set called Raw Failure Descriptions, a Data Set called Processed Failure Descriptions, a Data Set called Failure Descriptions with Stats, and a Data Set called Maintenance KPIs. These have all been modeled in previous examples, please re-use.

3. Have the Close Workorder Read from the Raw Failure Descriptions and Write to the Processed Failure Descriptions. Have the Generate Failure Statistics Read from the Processed Failure Descriptions and Write to the Failure Descriptions with Stats. Have the Compile Maintenance KPIs Read from the Failure Descriptions with Stats and Write to the Maintenance KPIs. These relationships are automatically generated if the classes are re-used.

4. Add a Language class and call it Multi-maintenance Language.

5. The Representation and Data Sets are all of this language type. Add three IsOfLanguage relationships.

6. The Business Process and the Application Functions can all speak the Multi-maintenance Language. Add three SpeaksLanguage relationships.

7. Press calculate and make sure that the communication is successful for all processes and functions.

Cost

Additional author: Per Närman

A description of MAP's cost viewpoint.

9

The use of information systems permeates all modern organizations; virtually every existing business process is supported by some sort of information system solution. As a consequence, system related operational and capital expenditures consume a substantial part of the overall budgets. This was illustrated in a survey of the IT spending of 2007 by Gartner where it was reported that information system costs consumed 4, 4 % of European firms revenue. Gartner (An information technology research and advisory company) further concluded that the information system costs were likely to grow, at least in absolute terms. Another study performed by the Bureau of Economic Analysis in the United stated concluded that the share of IT in business equipment investments in USA rose to above 50 percent in year 2000 [3]. The great importance of information systems for running a competitive business together with the significant costs associated with system investments has made it imperative to increase the quality of decisions concerning information systems management. Bad decisions not only jeopardize the smooth running of the business, they also cost a fortune. Information systems often stay with their organizations for a very long time, systems that were developed in the seventies still support core processes in some industries. The longevity of information systems means that cost assessment must not inly include the initial costs, but also the yearly costs associated with a system. This could for instance be maintenance, support, and license costs.

The original work that this section is based on can be found in [116].

9.1 How to measure cost

A frequently cited method for estimating IT development efforts is Barry Boehm et al.s COCOMO (COnstructive COst MOdel) method and its successor, aptly named COCOMO II, [36,117]. The method offer a

Movie 9: Cost of IT-systems.

framework and an algorithm to predict the number of man-months needed to complete software development projects. This effort estimation is based on the size of the software, an approximate productivity constant A, an aggregation of five scale factors E (precentedness, development flexibility, architecture/risk resolution, team cohesion, and process maturity), and effort multipliers to 15 cost driving attributes Although based on qualitative experiences and a substantial amount of quantitative data, the method is limited to software development projects, which excludes procurement and implementation of Commercial-Off-The-Shelf (COTS) products, and in particular does not address the entire life-cycle cost of IT investments. The COCOMO community has acknowledged the need to make estimations of COTS{related costs as well, which gave rise to COnstructive COTS (COCOTS) [118-120]. COCOTS explores technical factors related to integration of COTS products, but instead fails to address maintenance related costs or the costs related to the organizational change.

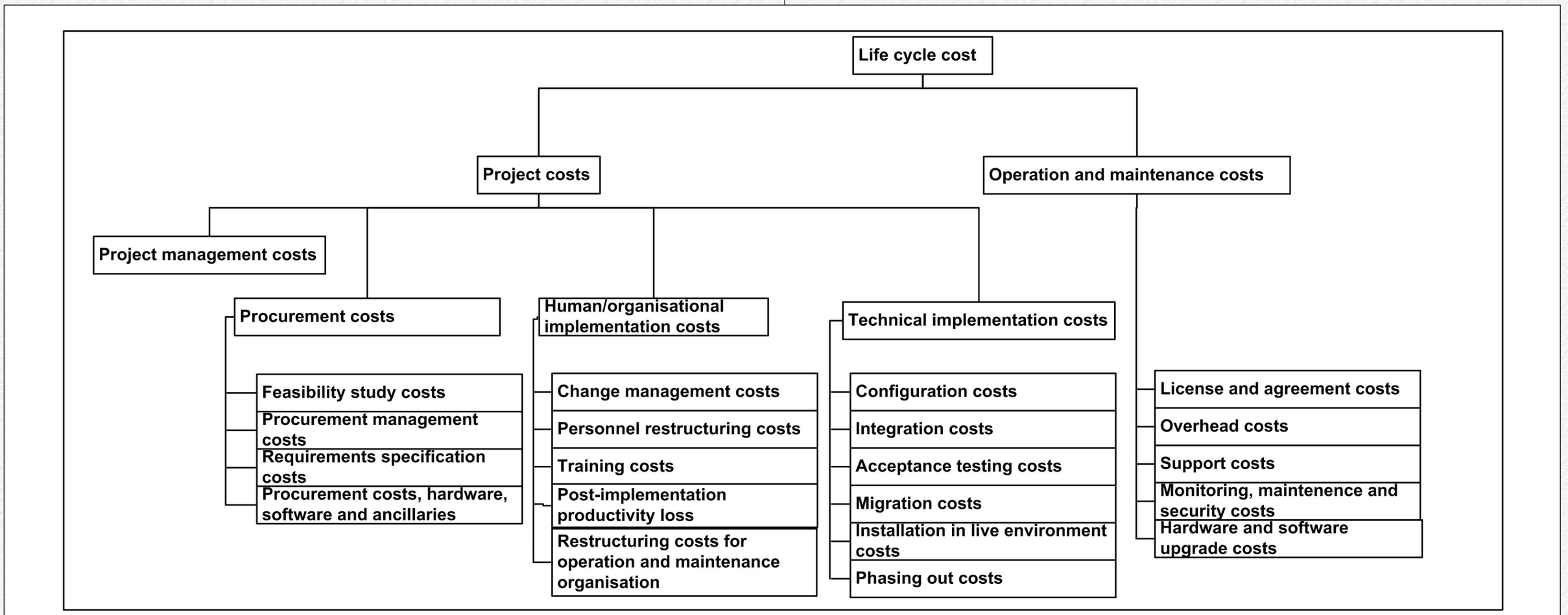


Figure 37: *The cost taxonomy by Närman et al [116].*

A more holistic approach taking softer cost drivers into account is presented in a number of IT cost taxonomies proposed by Irani et al. [121-123]. These taxonomies emphasize the importance intangible costs derived from for instance the introduction of new work practices that are associated with introducing new IT technology. Although complementing the more technical views expressed in COCOMO and COCOTS, the taxonomy says little on how to use cost factors to calculate IT investment costs in practice.

The assessment method for cost presented in this book focuses on the initial and yearly costs for different elements of EA. What these costs actually include depends on what costs a specific company using this viewpoint actually do have. The initial cost of an application could for instance include procurement, training, and configuration costs. While, the yearly cost could include maintenance, support, and license costs. The yearly cost for a certain role in an organization could include salary and educational costs. Närman et al. presents a cost taxonomy that could be used as a point of reference [116], cf. Figure 37.

9.2 The cost viewpoint

This section describes the cost viewpoint, cf. Figure 38. The viewpoint has the following main concepts:

- Service
  - BusinessService
  - ApplicationService
  - InfrastructureService
- BehaviorElement
  - BusinessFunction
  - ApplicationFunction
  - InfrastructureFunction
- ActiveStructureElement
  - ApplicationComponent
  - Role
  - Node

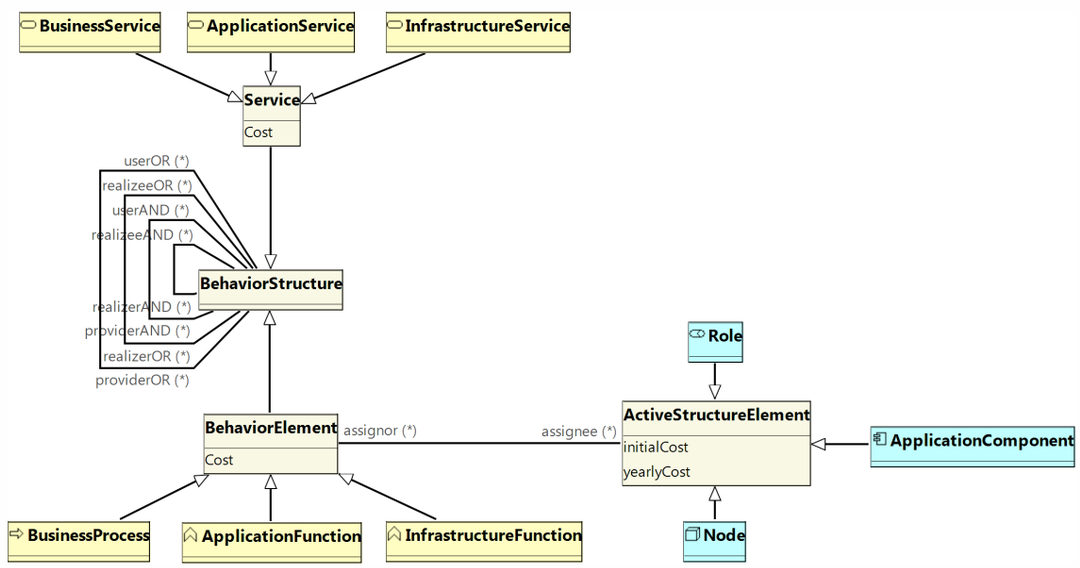


Figure 38: *The cost viewpoint.*

Concerns

The cost viewpoint aims at showing what direct costs that is associated with the elements of the enterprise architecture. E.g. it aims at answering what a certain service or process costs.

Stakeholders

The typical stakeholder for the cost viewpoint is the CIO or someone in similar position with a financial responsibility. Project managers or architects with cost requirements could also use this viewpoint when managing the budget plans.

Theory

The cost of an entity is assessed based on the initial and yearly costs of the related entities. This means, that if for instance two services share a function then the cost of this function is divided between the services. Or, e.g. if one service is realized by two functions then the cost of the service is a sum of the costs of the two functions.

`Service.Cost` (S.Cost) the cost of the `Service` is calculated by adding up the cost from all behavior elements that realize the `Service`. The cost from the `BehaviorElement` is equally distributed among the services. If $BE = \{be_1, \ldots, be_n\}$ is a list of `BehaviorElements`, S is a `Service`, and $BE \subseteq S.realizedByAND \cup S.realizedByOR$ then

$$f(S.Cost) = \sum_{i=1}^{n} \frac{be_i.Cost}{|be_i.realizesAND \cup be_i.realizesOR|}.$$

$$\mathrm{V}(S.Cost) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`BehaviorElement.Cost` (BE.Cost) the cost of the `BehaviorElement` consists of the costs from all services that the BE is using and the cost of the active structure elements that are assigned to the BE. The cost from the services and `ActiveStructureElements` is equally divided amongst all BEs to which it is used by/assigned. If $A = \{a_1, \ldots, a_n\}$ is a list of active structure elements, BE is a `BehaviorElement`, $S = \{s_1, \ldots, s_n\}$ is a list of services, $A \subseteq BE.assignee$ and $S \subseteq BE.usesAND \cup BE.usesOR$ then

$$f(BE.Cost) = \sum_{i=1}^{n} \frac{a_i.InitialCost}{|a_i.assignor|} +$$

$$\sum_{i=1}^{n} \frac{a_i.YearlyCost}{|a_i.assignor|} +$$

$$\sum_{i=1}^{n} \frac{s_i.Cost}{|s_i.userAND \cup s_i.userOR|}.$$

$$\mathrm{V}(BE.Cost) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`ActiveStructureElement.InitialCost` (ASE.InitialCost) the initial cost of an `ActiveStructureElement` is set as evidence in the model with the same currency used throughout the model.

$$\mathrm{V}(ASE.InitialCost) = \{x \in \mathrm{R} : 0 \leq x\}.$$

`ActiveStructureElement.YearlyCost` (ASE.YearlyCost) the yearly cost of an `ActiveStructureElement` is set as evidence in the model with the same currency used throughout the model.

$$\mathrm{V}(ASE.YearlyCost) = \{x \in \mathrm{R} : 0 \leq x\}.$$

Guidelines for use

To use the cost viewpoint follow this process: Firstly, model the services (business, application, and infrastructure), behavior elements (business processes, application and infrastructure functions), and active structure elements (roles, application components, and nodes). As stated in, for instance, the interoperability viewpoint these classes are typical elements of any enterprise architecture model. Thus, existing EA models could be re-used or the typical stakeholder of EA could be interviewed for information. Secondly, find out what the initial and yearly costs are for the active structure elements. Thirdly, run the analysis.

A cost view example

The CIO of ACME Energy has a tight budget for the coming year and since new technology have just been procured and some more is on the way there is a wish to manage these costs. The CIO asks the architects to calculate the expenses for the coming year based on the current architecture solution. The new elements of the architecture include a maintenance manager, three databases/servers, and the CMMS component. It is important for the CIO to divide the expenses between the business units of the company based on the amount that they actually do use. In this case the maintenance department, which owns the study maintenance business process, will be budgeted the expenses of the new maintenance manager, the three databases/servers, as well as the CMMS component (cf. Figure 39).In order to model the cost view example (cf. Figure 39) of ACME Energy follow these steps:

In order to model the cost view example (cf. Figure 39) of ACME Energy follow these steps:

1. Add the Study Maintenance business process, the Close Workorder business process, the Study Maintenance application service, the Generate Failure Statistics application function, the Compile Maintenance KPIs application function, the CMMS application component, the Generate GUI infrastructure service, the Data Retrieval primary site infrastructure service, and the Data Retrieval

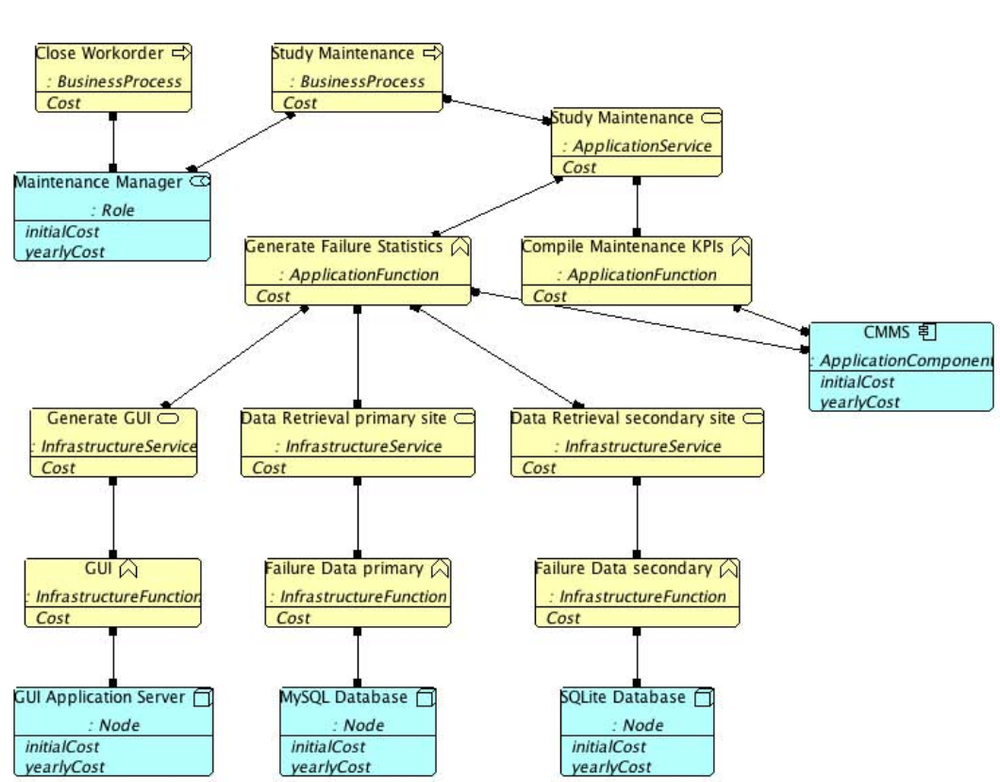


Figure 39: *A cost view example.*

secondary site infrastructure service. These have all been modeled in previous example views, re-use existing classes if possible.

2. The Study Maintenance process Uses the Study Maintenance service. Which in turn is Realized by both the Generate Failure Statistics and Compile Maintenance KPIs functions. Both these functions are Assignees of the CMMS component. The Generate Failure Statistics function Uses the Generate GUI service (AND) and one of the two Data Retrieval services (OR). These relations are all automatically generated based on the previously modeled views if the classes are re-used.

3. Add a new Role and name it Maintenance Manager. Assign it to the Study Maintenance and the Close Workorder processes.

4. Add three Infrastructure Functions and name them GUI, Failure Data primary and Failure Data secondary.

5. The Generate GUI service is Realized by the GUI function, the Data Retrieval primary site is Realized by the Failure Data primary function, and the Data Retrieval secondary site is Realized by the Failure Data secondary function. Add these three relationships.

6. Add three Nodes and name them GUI Application Server, MySQL Database, and SQLite Database.

7. The GUI Application Server node is Assigned to the GUI function, the MySQL Database is Assigned to the Failure Data primary function, and the SQLite Database is Assigned to the Failure Data secondary function. Add these three relationships.

8. Add the initial and yearly costs according to Table 5.

9. Press calculate and evaluate the cost of the Study Maintenance and Close Workorder business processes. These should read 1.795.000 and 550.000 SEK respectively.

Class	Initial Cost	Yearly Cost
Maintenance manager	100.000	1.000.000
CMMS	1.000.000	100.000
GUI Application Server	30.000	10.000
MySQL Database	45.000	10.000
SQLite Database	40.000	10.000

Table 5: *Initial and yearly cost data for cost view example.*

Utility

10

A description of MAP's utility viewpoint.

Chapter 10 Utility

"Nature has placed mankind under the governance of two sovereign masters, pain and pleasure. It is for them alone to point out what we ought to do, as well as to determine what we shall do. On the one hand the standard of right and wrong, on the other the chain of causes and effects, are fastened to their throne. They govern us in all we do, in all we say, in all we think: every effort we can make to throw off our subjection, will serve but to demonstrate and confirm it. In other words a man may pretend to adjure their empire: but in reality he will remain subject to it all the while. The principle of utility recognises this subjection, and assumes it for the foundation of that system, the object of which is to rear the fabric of felicity by the hands of reason and of law." – Jeremy Bentham [126].

As far back as 1780 Jeremy Bentham presented utility in the following way: "By utility is meant that property in any object, whereby it tends to produce benefit, advantage, pleasure, good, or happiness…" [126]. The idea was that utility over an act can be measured. This was followed by the felicific calculus which describes how such measurement can be constructed [127]; with the two units hedons, a unit of pleasure, and its antonym dolors, the unit of pain. When assessing the utility the following circumstances have to be considered: its intensity, its duration, its propinquity, its fecundity, its purity and its extent. Over the years, utility theory has evolved. In 1947 John von Neumann and Oskar Morgenstern presented the four axioms of rationality and a real-value utility function. This was later extended by, amongst others, Peter C. Fishburn, Ralph L. Keeney and Howard Raiffa, providing solid foundations to preference structures, independences and utility functions. This has been applied in game theory, healthcare and economics, where the satisfaction gained from goods and services can be measured in utility. The utility theory has been used to construct models for better decision making [128] and trade-off analysis to obtain the best outcome.

Utility is a representation of preferences that one has over something. Utility might be a bit abstract, but let's illustrate it in terms of money. A person would most probably prefer to have $100 on her bank account rather than zero; the same person would probably prefer to have even more. However, there comes a point where more money ceases to be as useful for this person as they were in the beginning. The effort in obtaining more, or trade-off to other things, might be too high. When having 10 or 10.1 billion dollars on her bank account the excess of $100 million might not bring that much extra utility to this person, her financial situation will be sound any way. The general idea is that when needs and requirements are not fulfilled the person is unhappy; utility is zero or very low. As the needs and requirements are meet the utility increases and when even wishes and desires are satisfied the person is completely happy and utility reaches its max.

In this book, utility is defined as the quality or condition of being useful. The method is based on individual preference structures (including independence assumptions) to perform the utility calculations [128]. This allows the decision to be presented in a quantifiable and comparable form. A significant change in utility between two alternatives implies that it would be a good idea to go for the higher scoring alternative. For example for an ordinary person, the increase in utility could be greater when wealth changes from $0 to $100 and worth working for. When wealth

changes from \$10.000 to \$10.100 the utility might not increase that much, the person might find it less rewarding to do the extra work to go from \$10.000 to \$10.100.

Utility theory has the potential to aid decision makers to evaluate enterprise architecture as-is and to-be scenarios and comparing them against each other. A business service which does not meet the requirements of the company is of no or little utility to the enterprise. At the same time a business service exceeding the requirements and wishes by far might not bring that much extra utility compared to if the service meet the requirements and just passes the extra wishes. The utility theory can aid the decision maker in making the trade-offs and decisions on what is important and which solution to implement.

An enterprise architecture can incorporate many quality attributes. The decision maker usually has to balance these attributes against each other to obtain the best possible architecture. This chapter presents a systematic and formal way of assessing the quality of an enterprise architecture based on a decision maker's preferences. The method presented aims to provide support to the task of making trade-off decisions and balancing the enterprise architecture.

10.1 How to measure utility

Utility is a subjective matter. It therefore needs to be evaluated based on a set of individual preferences. An absolute number specifying the utility of a certain state can be difficult to assess, but when given a choice of two or more it is somewhat easier to tell which is better or when the stakeholder is indifferent to them.

The following notation will be used throughout the utility chapter. $X = \{X_1, X_2, .., X_N\}$ is a set of N attributes or properties, X_1, which are included in the domain under which the decision maker will make her decision. An example of attributes in the EA domain is the non-functional attributes: availability, A, security, S, interoperability, I , and modifiability, M. Where $X = \{A, S, I, M\}$. The complement of an attribute, X_1, is denoted $\bar{X_1} = \{X_1, .., X_{I-1}, X_{I+1}, .., X_N\}$. In the previous EA example the complement of the security attribute is $\bar{S} = \{A, I, M\}$. Each attribute, X_1, can be assigned a state or value, x, which is within a state/value range containing two or more values, $x = \{x_1, x_2, .., x_n\}$, it can be both discrete or continuous. The non-functional attribute interoperability could for instance have the two states *true* or *false*. Meaning that it is either interoperable or not, $interoperability = \{true, false\}$. The availability can be expressed in a percentage value range going from completely unavailable, 0%, to always available, 100%, $availability = \{0\,\%, .., 100\%\}$. The decision maker has to be able to tell if one value is preferred over another, $x_i \succ x_j$, or if she is indifferent of the two. The preference relations are:

- Strict preference $\succ$, is a transitive relation. If $x_i \succ x_j$ and $x_j \succ x_k$ then $x_i \succ x_k$.

- Indifference $\sim$, is a transitive, reflective relation, $x_i \sim x_i$ for all x, and symmetric, $x_i \sim x_j \rightarrow x_j \sim x_i$.

- For each pair of , x_i and x_j, only one of $x_i \succ x_j$, $x_j \succ x_i$ or $x_i \sim x_i$can hold.

- Weak preference $\succeq$, is transitive and complete, for all pair of, x_i and x_j, either $x_i \succeq x_j$ or $x_j \succeq x_i$.

Preference relations are discussed in greater detail in [129, 132].

The utility of all attributes is denoted $U(X) = U(X_1, X_2, \ldots, X_N)$, and has a range $0 \leq U(X) \leq 1$. The utility for a specific attribute X_1 with the value x_i is denoted $u_i(x_i)$. The conditional utility function, which is the utility function over one attribute, has a range of $0 \leq u_i(x_i) \leq 1$, where the least preferred outcome of the attribute $u_i^0(x_i) = 0$ and the most preferred outcome $u_i^*(x_i) = 1$. If $x_i \succ x_j$ then $u_i(x_i) > u_j(x_j)$ (Note: preference relation, $\succ$. Inequality relation, $>$). If $x_i \sim x_j$ then $u_i(x_i) = u_j(x_j)$. Returning to the availability attribute, let's say that the stakeholder's least preferred outcome of availability on a system is 90%, $u_a^0(90\%) = 0$. The most preferred outcome is 100%, $u_a^*(100\%) = 1$. The stakeholder would typically prefer the higher availability and the conditional utility function would approach and reach 1 as the availability increases from 90% to 100%, given any two values in-between and the stakeholder can tell which one is the most preferred.

Scaling constants are important in addition to the conditional utility functions, $u_i(x_i)$, when assessing the total utility, $U(X)$. The scaling constants are used to scale the utility functions. This is done in order to keep them within the proper value range but also to reflect the decision maker's preferences on how important each attribute is. The scaling constants can be seen as a weight to each conditional utility function. The scaling constants to each conditional utility function, k_i, have a range of $0 < k_i < 1$. In some cases an additional scaling constant k is used. k is a non-zero scaling constant, $k > -1$. More about scaling constants and how to assign them is presented under each type of utility function.

Lotteries are an important concept to utility theory. The lotteries are used to elicit the stakeholder's preferences and to validate independence assumptions. [130] has provided a definition of a lottery. Given a set of prizes, Ω, "A lottery on Ω is a device for deciding which prize in Ω you will receive, on the basis of a single observation that records which one of a set of mutually exclusive and exhaustive uncertain events took place. It is possible that with each of these uncertain events there is associated a known chance; for example, this would be so if we were observing a single spin of a well-made roulette wheel."

Let's consider the lottery of flipping a perfectly balanced coin. Ω = {heads, tails}, where there is the known chance of 50% for either of the two. This lottery is graphically presented in Figure 40.

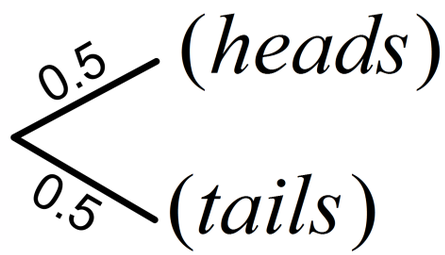


Figure 40: *A coin flip lottery.*

Independence

To simplify the tailoring of the utility function to the stakeholder there are three different kinds of independence assumptions which can be made, and validated with the stakeholder's preferences. The purpose is to reduce the workload when determining how the utility function behaves. The different kinds of independencies are explained below. Each kind of independence is accompanied by an example using the attributes availability A, interoperability I and cost C where $X = \{A, I, C\}$ is the set of all three attributes. For the sake of the example $u_A(80\%) = 0$, $u_A(100\%) = 1$, $u_1(false) = 0$, $u_1(true) = 1$, $u_C(\$5000) = 0$ and $u_C(\$0) = 1$.

Preferential independence

The preferential independence is important to determine to ensure if the stakeholder's preferences over one attribute remains the same regardless of the other attributes.

If attribute $X_I \in X$, then X_I is said to be preferentially independent of its complement $\bar{X}_I$ if the preference order, $x_i \succeq x_j$, for the outcome $(x_i, \bar{x}_i)$, where the values are fixed, does not depend on the fixed amount [128, 131].

- If $x_i, x_j \in X_I$, $x_i \succeq x_j$ and $x_i \in \bar{X}_I$, X_I is preference independent of $\bar{X}_I$ if $\forall \bar{x}_i((x_i, \bar{x}_i) \succeq (x_j, \bar{x}_i))$.

In other words, preferential independence means that a decision maker's preference over an attribute, X_I, is not dependent on any other attribute. [128, 131].

The two attributes X_I and X_J are mutually preference independent if X_I is preference independent of X_J and X_J is preference independent of X_I. On a similar note the attributes $X_1, X_2, \ldots, X_{N-1}$ and X_N are mutually preference independent if, for every subset $I \subseteq \{1,..,N\}$ the set X_I of these attributes is preference independent of $\bar{X}_I$. [132].

Example of preferential independence

Let's take the two attributes availability A, and cost C. If the stakeholder always prefers a low cost compared to a high, no matter in which state the availability is, then C is preference independent of A. So if C is preference independent of A, the stakeholder would e.g. prefer $(\$0; 85\%)$ over $(\$5000; 85\%)$ and $(\$450; 85\%)$ over $(\$451; 85\%)$. If there exists an availability state where the stakeholder would prefer a more expensive service compared to a cheaper service, then C is not preference independent of A. In the case of availability and cost, it might be obvious that one prefers to pay less regardless of the value of the availability attribute. Let us leave the systems and services to illustrate an example when you would not be preference independent. Given the domain of the human body we can look at the two attributes blood glucose concentration BGC, and insulin I. When there is an excess of glucose in the blood, insulin is produced in order to decrease the blood glucose concentration. If the blood glucose concentration is normal and insulin is produced it would cause a glucose deficiency. The body strives to keep a balanced blood glucose concentration. The two attributes are not preferentially independent of each other since $(I$ – not produced; BGC – normal) is preferred over

(I – produced; BGC – normal), but if we change the fixed amount in blood glucose concentration to be high (I – produced; BGC – high) is preferred over (I – not produced; BGC – high) .

Utility independence

If attribute $X_I \in X$, then X_I is said to be utility independent of its complement $\bar{X}_I$ if the preference order over lotteries on X_i, $(x_i, \bar{x}_i)$, with $\bar{X}_I$fixed, does not depend on the fixed amount of $\bar{x}_i$. [128, 131]. The attributes in X are mutually utility independent if every subset of $\{X_1, X_2, .., X_N\}$ is utility independent of its complement [128].

Example of utility independence

If the complement values to attribute are held fixed at the least desirable level , what is the decision maker's certainty equivalent for a 50/50 gamble yielding values 80% and 100% on the availability attribute? The certainty equivalent is the guaranteed amount from an outcome which to the decision maker is equally desirable as a gamble with known chance. Now let's say that the decision maker would be indifferent between the gamble and the certain outcome if the availability of the certain option is 90%. This scenario is illustrated in Figure 41.

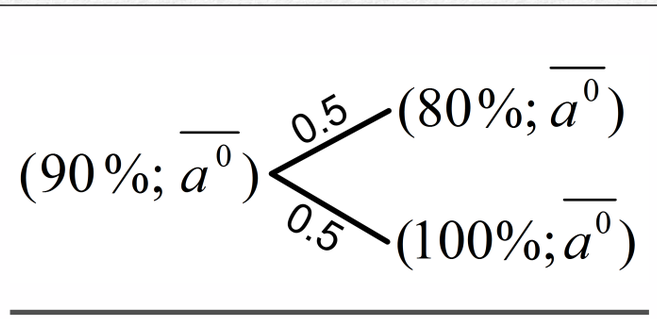


Figure 41: *Certainty equivalent assigned to 90% for a 50/50 gamble.*

If the complement values were held fixed at some other level, let's say the most desired outcome, $\bar{a}^*$, would the decision maker's certainty equivalent, 90%, be? If the decision maker sticks to the same certainty equivalent, the attribute is utility independent of its complement.

Utility independence can also be elicited using combinations of multiple lotteries. The decision maker is given a choice between the two lotteries like the one shown in Figure 42.

The decision maker may prefer lottery 1 over lottery 2 because there is less risk associated with regards to cost and availability in System 1. No matter which lottery the decision maker prefers, if the choice over the two lotteries remains the same when the interoperability is changed to , then cost and availability are utility independent of interoperability.

Additive independence

Two attributes, X_I and X_J are additive independent if the paired preference comparison of any two lotteries (like the two lotteries previously shown in Figure 42), defined by two joint probability distribution on $X_i \times X_J$, depends only on their marginal probability distribution (the probability of one variable taking a specific value irrespective of the values of the other) [128]. If the two lotteries in Figure 43 are equally desirable then X_I and X_J are additive independent.

Example of additive independence

Let's have a look at the two attributes availability and interoperability. If the decision maker thinks that the two lotteries

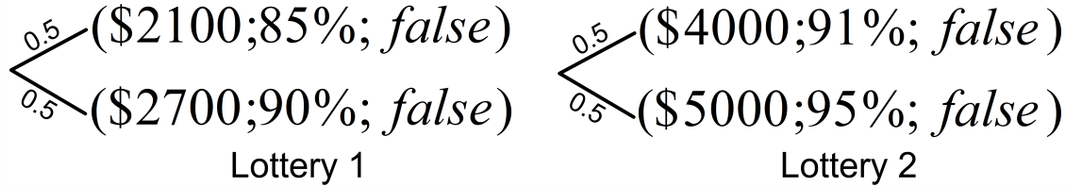


Figure 42: *Paired preference comparison to elicit if interoperability is utility independent of cost and availability.*

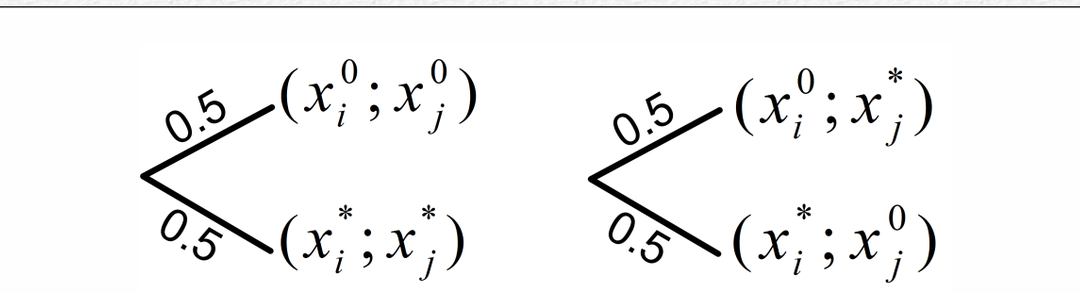


Figure 43: *Paired preference comparison lottery to elicit additive independence.*

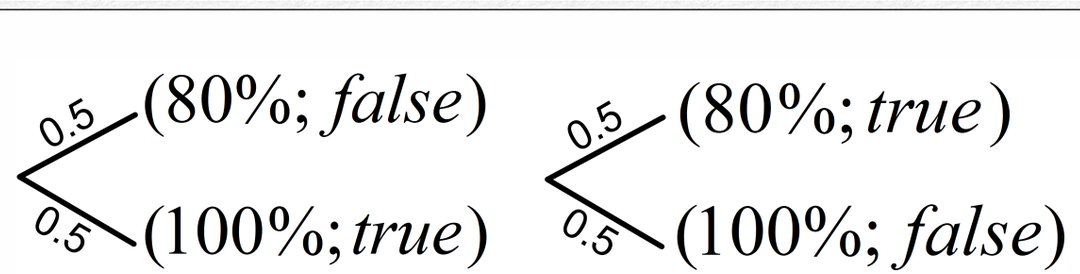


Figure 44: *Additive independence example.*

shown in Figure 44. Additive independence example are equally preferable, then the two attributes and are additive independent of one another.

In both lotteries there is a 50% chance of getting the most and the least preferred outcome on each attribute.

Utility functions

The general utility function over the outcome of the attributes in X is $u(x_1, x_2, .., x_n) = f(f_1(x_1), f_2(x_2), .., f_n(x_n))$, the downside with this utility function is that it can be resource consuming to define the behavior of function f. The conditional utility functions f_i always have to be determined and there is some assistance to get when defining the utility function f. This section presents three common utility functions. The key idea is to investigate the previous mentioned independence assumptions with the stakeholder to see if any of them apply. When the independence assumptions have been validated, the right utility function can be chosen to fit the stakeholder preferences. If none of the independence assumptions applies the reader is advised to read [128].

There are five stages to go through when determining the utility function for a decision maker [128]:

- Introducing the terminology and ideas.
- Identifying relevant independence assumptions.
- Assessing conditional utility functions.
- Assessing the scaling constants.
- Checking for consistency and reiterating.

Additive utility

If the attributes are all additive independent of each other the utility is calculated using the additive utility function, Equation 1. The additive utility function simply sums up the utility of each conditional utility function with a weight on how important each attribute is to the stakeholder. These weights are subsequently called scaling constants.

$$U(X) = \sum_{i=1}^{n} k_i u_i(x_i)$$

Equation 1.

All the scaling constants k_i, should be normalized so that $\sum_{i=1}^{n} k_i = 1$.

Assessing the scaling constants

When assessing the scaling constants, the stakeholder needs to rate the importance of each attribute. This can be done in a numerous of ways. In the end, the assessed weight for each attribute has to be normalized. One way of assessing the weights is to have the stakeholder assigning utility of $u(x_i^*; x_i^0) = p_i$. The weight can also be evaluated by proposing a bet, letting the stakeholder assign the probability p_i, illustrated in Figure 45.

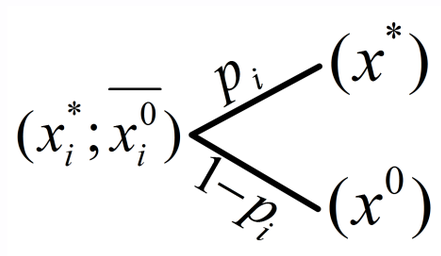


Figure 45: *Assessing the weight*

The scaling constants are then normalized: $k_i = \frac{p_i}{\sum_{i=1}^{n} p_i}$.

Multiplicative utility

If each attribute in is utility independent of its respective complement then the multiplicative utility function holds. The multiplicative utility function multiplies the conditional utility functions with each other using both individual scaling constants k_i and a final scaling constant k to ensure that $0 \leq U(X) \leq 1$.

$$1 + kU(X) = \prod_{i=1}^{n} (1 + kk_i u_i(x_i))$$

Equation 2.

To calculate the scaling constant k, all attribute utility functions are set to 1, $u_i(x_i) = 1$ and so $U(X) = 1$. This gives:

$$1 + kU(X) = \prod_{i=1}^{n} (1 + kk_i u_i(x_i))$$

Equation 3.

Out of which k can be calculated. Keep in mind that $k \neq 0$ and $k > -1$. If $k = 1$ the multiplicative utility function is reduced to the additive form.

Assessing the scaling constants

The same approach as presented in additive case can be used to assess the k_i values in the multiplicative utility function. The stakeholder assigns the utility of $u(x_i; \bar{x}_i^0) = p_i$. The weight can also be evaluated by proposing a bet, letting the stakeholder assign the probability p_i, illustrated in Figure 45, $k_i = p_i$.

Example of the multiplicative utility function

The example scenario is based on the four attributes in $X = \{X_1, X_2, X_3, X_4\}$. Lets assume that the weights have been set by a stakeholder: $k_1 = 0.95$, $k_2 = 0.07$, $k_3 = 0.4$ and $k_4 = 0.67$. k can then be calculated by using Equation 3.

$$1 + k = (1 + 0.95k)(1 + 0.07k)(1 + 0.4k)(1 + 0.67k)$$

$$0 = k^3 + 19.33k^2 + 80k + 61.16$$

$$0 = (k + 13.886)(k + 4.455)(k + 0.988)$$

Equation 4.

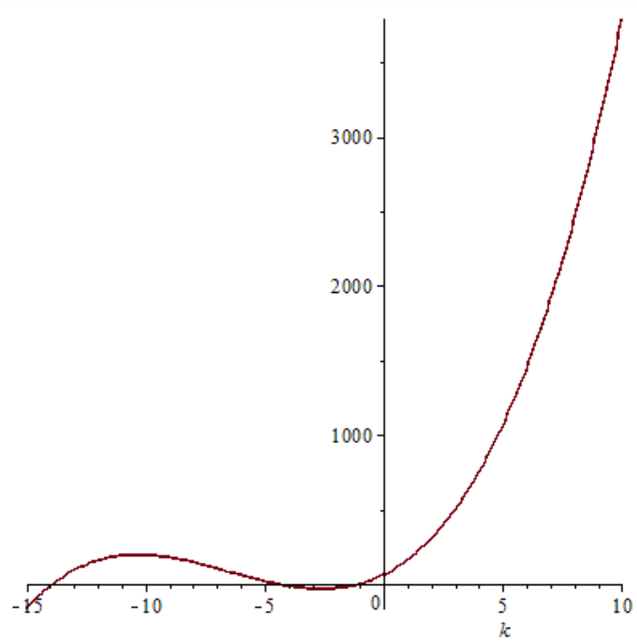


Figure 46: *The k polynomial*

Since $k > -1$ it follows that $k = -0.98$. Let's assume that in this example scenario the conditional utility functions have evaluated to $u_1(x_1) = 0.9$, $u_2(x_2) = 0.6$, $u_3(x_3) = 1$ and $u_4(x_4) = 0.1$. Now Equation 2. can be used to calculate the utility of the service.

$$1 - 0.988U(X) = 0.083$$

$$U(X) = 0.926$$

Multi linear utility

If the attributes in X are mutually utility independent, the multi linear utility function should be used.

The multi linear utility function is a generalization of the additive and multiplicative utility functions. To setup the multi linear utility function $2^n - 2$ scaling constants have to be assessed by the stakeholder. When there are four or more attributes, the overhead of assessing the 14 or more scaling constants can be too time and effort consuming and the result can often be approximated using the additive or multiplicative utility functions dependent on the independence characteristics of the attributes [133].

$$U(X) = \sum_{i=1}^{n} (k_i u_i(x_i)) + \sum_{i=1}^{n} \sum_{j>i} (k_{ij} u_i(x_i) u_j(x_j)) +$$

$$+ \sum_{i=1}^{n} \sum_{j>i} \sum_{l>j} (k_{ijl} u_i(x_i) u_j(x_j) u_l(x_l)) + \ldots + k_{123..n} u_i(x_i) u_j(x_j) \ldots u_n(x_n)$$

A two attribute example

This subsection presents a small example containing two attributes, availability and modifiability. Availability is measured in percentage ranging from 0 to 100. Modifiability is measured on a scale from 1 to 14. After introducing the utility concept to the decision maker the individual attribute utility functions are assessed.

Identifying relevant independence assumptions

First, the decision maker is questioned about potential preferential independence over a set of outcomes like the one in Table 6. Preferential independence question example where the decision maker's preferred outcome is marked with bold text.

Where:

$$*1: (x_\alpha^+, x_m^\beta) \text{ or } (x_\alpha^-, x_m^\beta)$$

$$*2: (x_m^+, x_\alpha^\beta) \text{ or } (x_m^-, x_\alpha^\beta)$$

The table shows that the two attributes are mutually preference independent, since the higher availability is always preferred over the lover when the modifiability is fixed. The same thing goes with the modifiability where the decision maker always prefers the higher modifiability independent of which fixed stat the availability is.

Thereafter, the decision maker is faced with setting the certainty equivalent for the 50/50 gamble vs. certain outcome for the two attributes. The purpose with this task is to validate the utility independence assumption. Let's say that the decision maker's certainty equivalent for the proposed 50/50 gamble yielding values 80 and 100 on the availability is 84 if the modifiability is 10. The same certainty equivalent was also given for another modifiability value of 13, illustrated in Figure 47.

A preference independent of M	M preference independent of A
(100%;1) or (0%;1)	**(14;50%)** or (1;50%)
(53%;10) or (49%;10)	(7;50%) or **(8;50%)**
(90%;14) or (75%;14)	**(9;23%)** or (5;23%)
(98%;11) or **(99%;11)**	**(14;100%)** or (13;100%)
*1	*2

Table 6: *Preferential independence question example.*

The procedure is repeated with different attribute values and for both attributes. The decision maker remains consistent when setting the certainty equivalents and the utility independence assumption is valid.

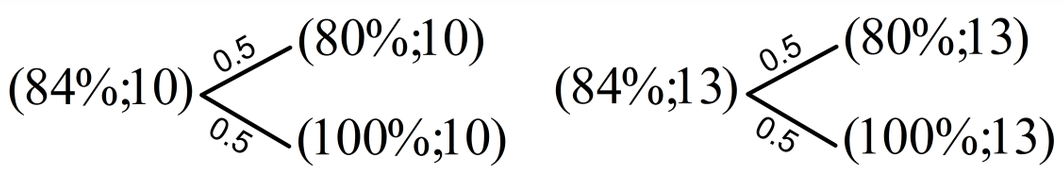


Figure 47: *Utility independence example*

The additive dependence is then explored as explained earlier. The decision maker is faced with the task of comparing the two lotteries in Figure 48. The attribute values are changed over and over to see if the

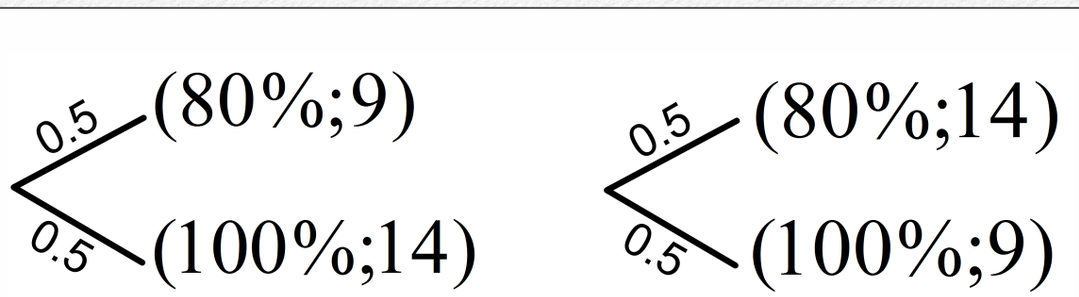


Figure 48: *Paired lotteries for the additive independence example*

decision maker's replies are consistent. In the example scenario the decision maker doesn't find the two lotteries to be equally desirable. The two attributes are therefore not additive independent.

Assessing the conditional utility functions

Starting with availability, the decision maker is asked to determine the endpoints of the conditional utility functions. The least desirable

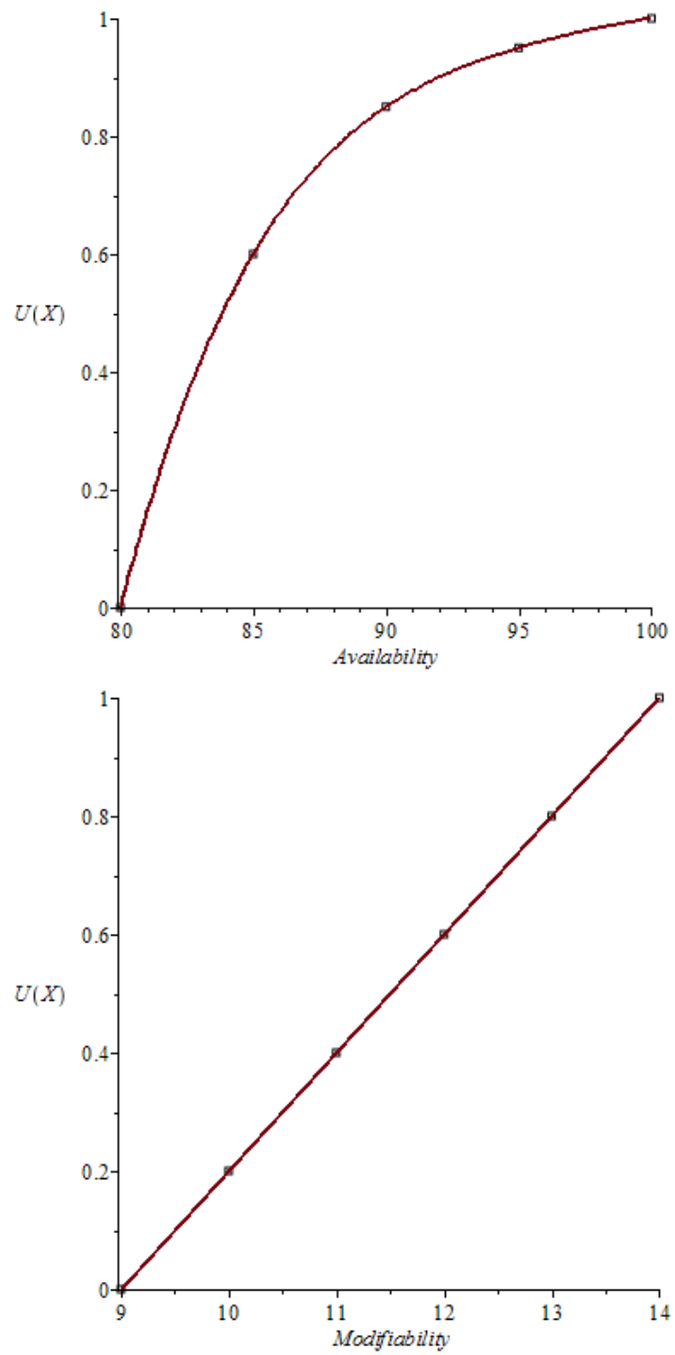


Figure 49: *Availability and modifiability utility functions.*

availability in the example is 80% and 100% is considered the most attractive, $u_a(80) = 0$ and $u_a(100) = 1$. Thereafter, the respondent is asked to assess a few points in between the threshold values. When those have been assessed, the attribute utility function can be determined by interpolation. The same procedure is carried out for the modifiability attribute. An example of this is shown in Figure 49. where the squares indicate the stakeholder's assessments.

Assessing the scaling constants

Starting with the availability attribute the stakeholder is requested to assess $u(100\,\%\,;9) = p_i$ or equally:

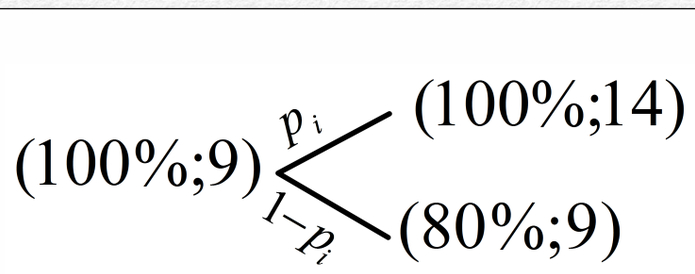


Figure 50

Let's say that the stakeholder sets $p_i = 0.95$. The same procedure is done for the modifiability attribute where the stakeholder assigns $u(14; 80\%) = 0.3$. The next step is to evaluate k.

The utility function

When the independence assumptions, conditional utility functions and scaling constants have been assessed, the multi attribute utility function can be derived.

$$1 - 0.87U(A, M) = (1 - 0.87 * 0.95 * u_a(x_a)) * (1 - 0.87 * 0.3 * u_m(x_m))$$

The utility function is visualized in Figure 51. The availability in this example scenario turned out to be 92% and the modifiability scored 12; $u_a(92\%) = 0.9$, $u_m(12) = 0.6$. The final utility score can then be calculated by inserting these values in the function:

$$1 - 0.87U(92\,\%,12) = (1 - 0.87 * 0.95 * 0.9) * (1 - 0.87 * 0.3 * 0.6)$$

$$U(92\,\%,12) = 0.9$$

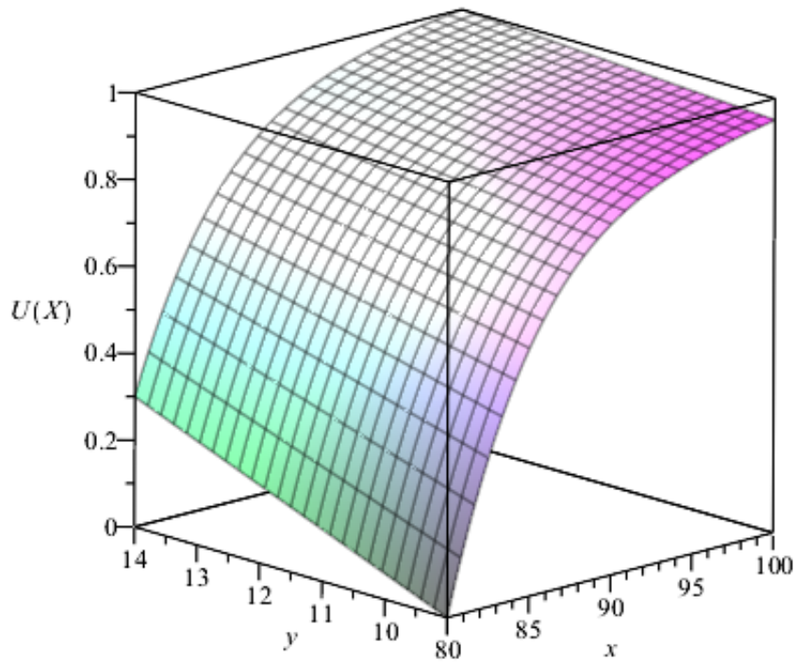


Figure 51: *A multiplicative utility function over the two attributes availability and modifiability*

10.2 The utility viewpoint

This section describes the utility viewpoint, cf. Figure 52. The viewpoint has the following main concepts:

- Stakeholder
- ProcessServiceInterface
- Requirement
  - ServiceRequirement
  - ApplicationServiceRequirement
  - InterfaceRequirement
  - InformationRequirement
- Service
  - BusinessService
  - ApplicationService
  - InfrastructureService
- PassiveComponentSet
  - DataSet
  - RepresentationSet

Concerns

Using the utility viewpoint makes it possible to estimate the utility of the system in modeled architectures. The use of the utility viewpoint provides a high abstraction level measurement that can be suitable for easy comparison between competing scenarios.

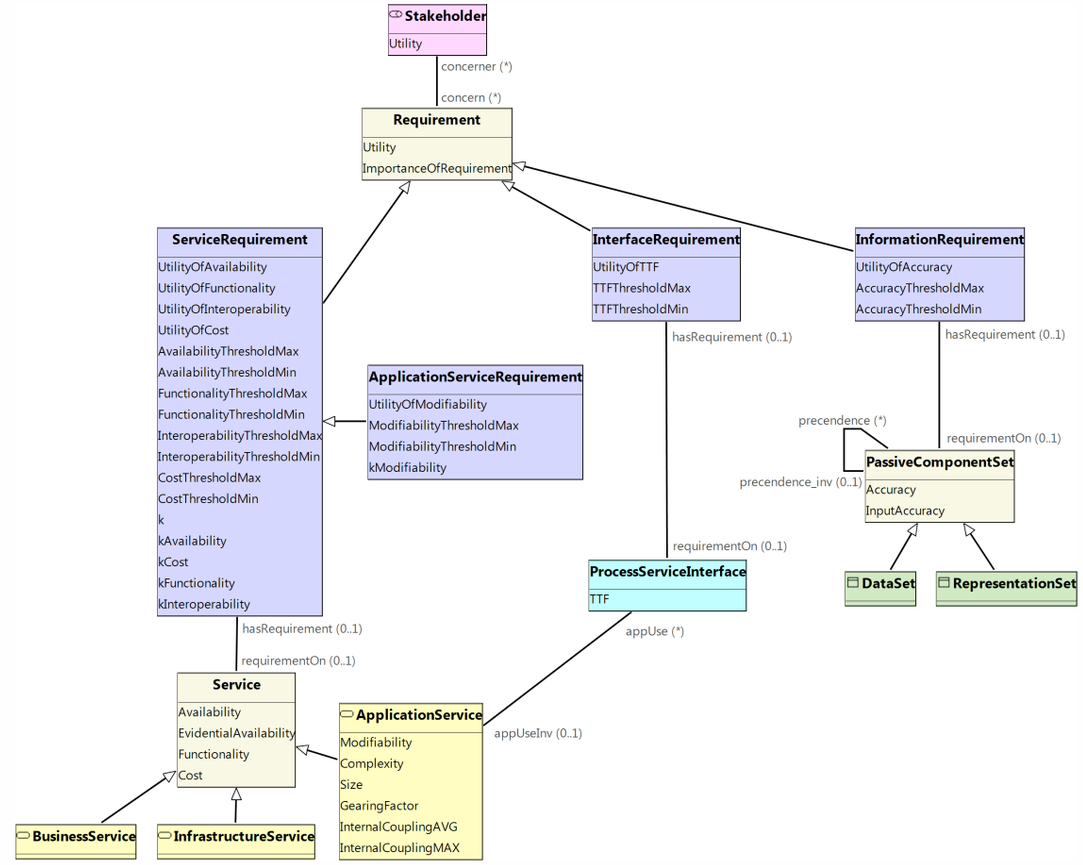


Figure 52: *The utility viewpoint.*

Stakeholders

The stakeholder for the utility viewpoint is the actor who has requirements on a service. The stakeholder is modeled in the viewpoint.

Theory

The utility in this book is evaluated as follows. In ArchiMate a stakeholder is de_ned as the role of an individual, team, or organization (or classes thereof) that represents their interests in, or concerns relative to, the outcome of the architecture. In the utility viewpoint the Stakeholder is defined in the same way. The `Stakeholder` contains one attribute `Utility`.

`Stakeholder.Utility` The utility in the Stakeholder class is based on the utility of each requirement the stakeholder has. The utility of each requirement evaluated based on the importance of the requirement and the overall utility score is a sum of this. If $R = \{r_1, \ldots, r_n\}$ is a list of `Requirements` and $R \cup Stakeholder.concern$ then

$$f(Stakeholder.Utility) = \sum_{i=1}^{n} r_i.Utility * r_i.ImportanceOfRequirement$$

$$\mathrm{V}(Stakeholder.Utility) = \{x \in \mathrm{R} : 0 \leq x \leq 1\}.$$

In ArchiMate a requirement is de_ned as a statement of need that must be realized by a system. In this viewpoint the statement of need is specified in terms of utility. If the requirement is not fulfilled there is no utility. On the other end, if the aspect on which the requirement is stated is above a certain value the utility is 1. The `Requirement` class has two attributes, `Utility` and `ImportanceOfRequirement`. The `Requirement` class is not intended for modeling but acts as a superclass to `ServiceRequirement`, `ApplicationRequirement`, `InterfaceRequirement` and `InformationRequirement`.

`Requirement.Utility` The utility of the requirement is evaluated as a product of all utility from each aspect the requirement is measuring. A `ServiceRequirement` for instance has four aspects, availability, functionality, interoperability and cost. If sr is a `ServiceRequirement` then

$$f(sr.Utility) = sr.UtilityOfAvailability * sr.UtilityOfFunctionality * sr.UtilityOfInteroperability * sr.UtilityOfCost$$

$$\mathrm{V}(Requirement.Utility) = \{x \in \mathrm{R} : 0 \leq x \leq 1\}.$$

For each aspect the requirement is measuring, a threshold maximum, and threshold minimum value, is defined by the stakeholder, and added as evidence in the two attributes associated with the aspect. If we follow the example of a `ServiceRequirement` one of the measured aspects is availability. The `ServiceRequirement` has the two attributes (amongst other) `AvailabilityThresholdMax` and `AvailabilityThresholdMin`. The threshold min value is added as evidence with the lowest availability the stakeholder accepts from the service. The availability threshold max value is added as evidence where the service is not considered to provide more utility if it is more available. For illustrative purposes let's say a stakeholder has the requirement on a service where the service is of no use if the availability is less than 80% and an availability higher than 98% is too resource consuming for the organization. The stakeholder sets the threshold minimum to be 80% and the maximum to 98%, the utility curve

of this is shown in Figure 53. If *sr* is a `ServiceRequirement` the utility of availability would then be

$$f(sr.UtiltyOfAvailability) = \frac{1}{sr.AvailabilityThresholdMax - sr.AvailabilityThresholdMin} * sr.requirementOn.Availability$$

$$V(Requirement.Utility) = \{x \in R : 0 \leq x \leq 1\}.$$

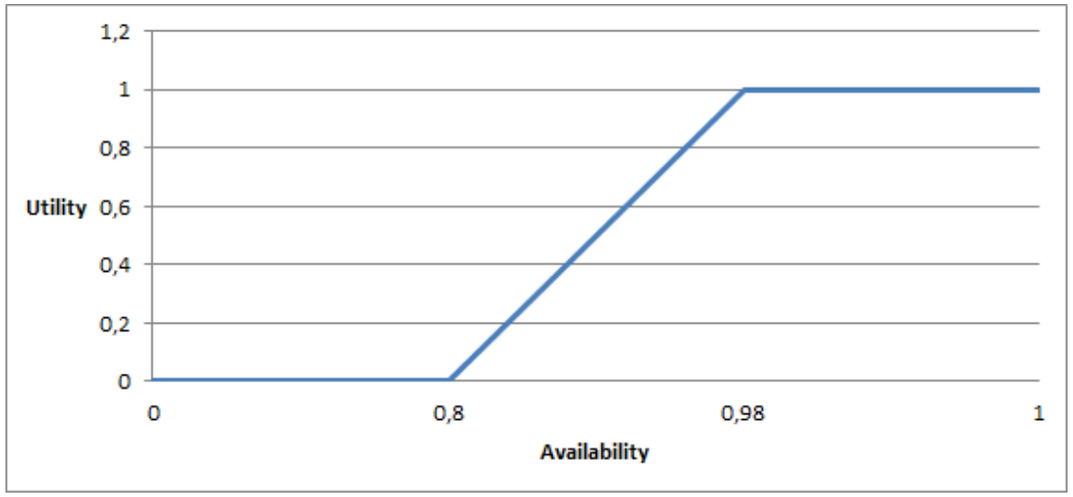


Figure 53: *An example of service availability utility.*

When dealing with cost, a low cost gives a high utility. For illustrative purposes a stakeholder thinks a service is really good if the cost is less than 8000. If the service cost more than 25000 the stakeholder thinks that it is too expensive and therefore of no utility. A cost utility curve example of this scenario is shown in Figure 54.

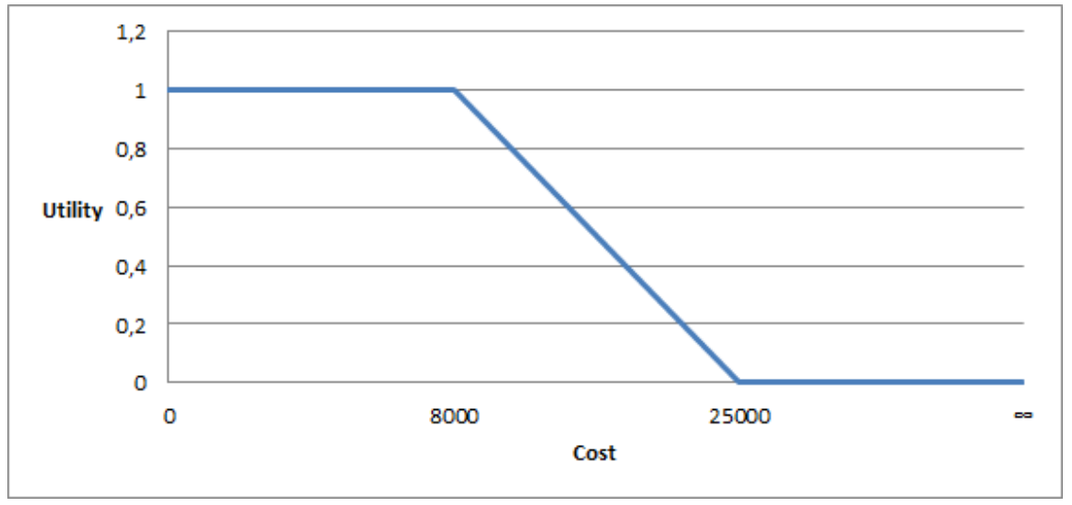


Figure 54: *An example of service cost utility.*

The threshold values do not necessarily need to reach zero. An illustration of this is when evaluating the utility of modifiability in an application service. Even though the modifiability is zero the stakeholder might still be provided with some utility from the service. For illustrative purposes, the stakeholder thinks that the utility of a certain application service is 0.4 when the modifiability is 0 and it is sufficient if the modifiability is 11. A utility curve example of this scenario is shown in Figure 55.

For attributes such as interoperability which has binary states (true/false) the threshold minimum value is the utility of the interoperability when it is evaluated to false. The threshold maximum value is the utility of the interoperability when it is evaluated to true. Let's assume a stakeholder considers the service to have a interoperability utility of 0.2 when the interoperability is evaluated to false and 1 if the service has full

interoperability (evaluated to true), the utility of this example is shown in Figure 56.

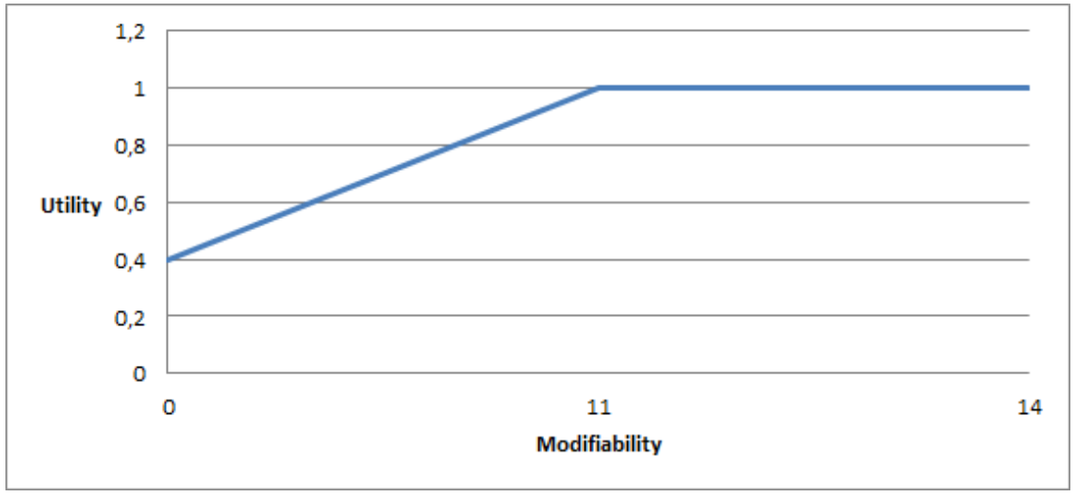


Figure 55: *An example of service modifiability utility.*

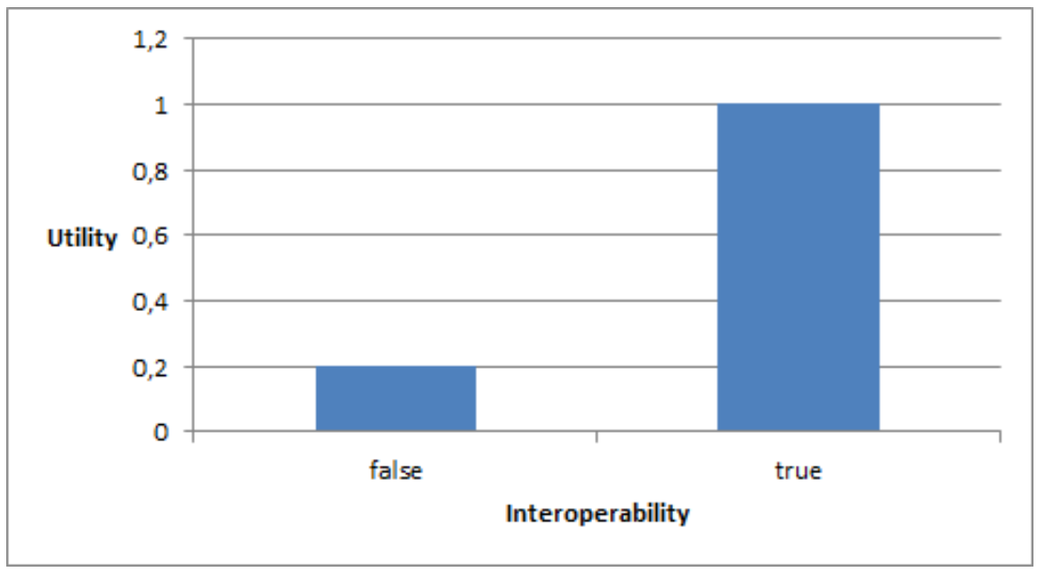


Figure 56: *An example of service interoperability utility.*

Guidelines for use

To use the utility viewpoint follow this process: Firstly, select the intended stakeholder and add the stakeholder in the model. Add the services, interfaces and passive component sets on which the stakeholder has requirements. For each of these add a requirement of right type in between the stakeholder and the services, interfaces and passive component sets. Secondly, have the stakeholder set the importance of the requirements and the threshold values and add these as evidences to the model. Thirdly, run the analysis.

A utility view example

At ACME Energy the CIO is concerned with the maintenance management process. An analysis to see if the application service for maintenance management fulfills the organizations requirements is conducted.

In order to model the utility view example (cf. Figure 57) of ACME Energy follow these steps:

- Add the Study Maintenance and the ProcessServiceInteface, they have been modeled in previous example views, re{use existing classes if possible.
- Add a new Stakeholder and name it CIO.
- Add a new ApplicationServiceRequirement and name it Study Maintenance Requirement.

- Add a new InterfaceRequirement and name it Maintenance Interface Requirement.
- Connect the two requirements to the stakeholder. Connect the Study Maintenance Requirement with Study Maintenance. Connect Maintenance Interface Requirement to ProcessServiceInteface.
- By now you are familiar with both ACME and the process of adding evidences. You may now act as the CIO and set your requirements.
- Press calculate. Does the application fulfill you requirements?

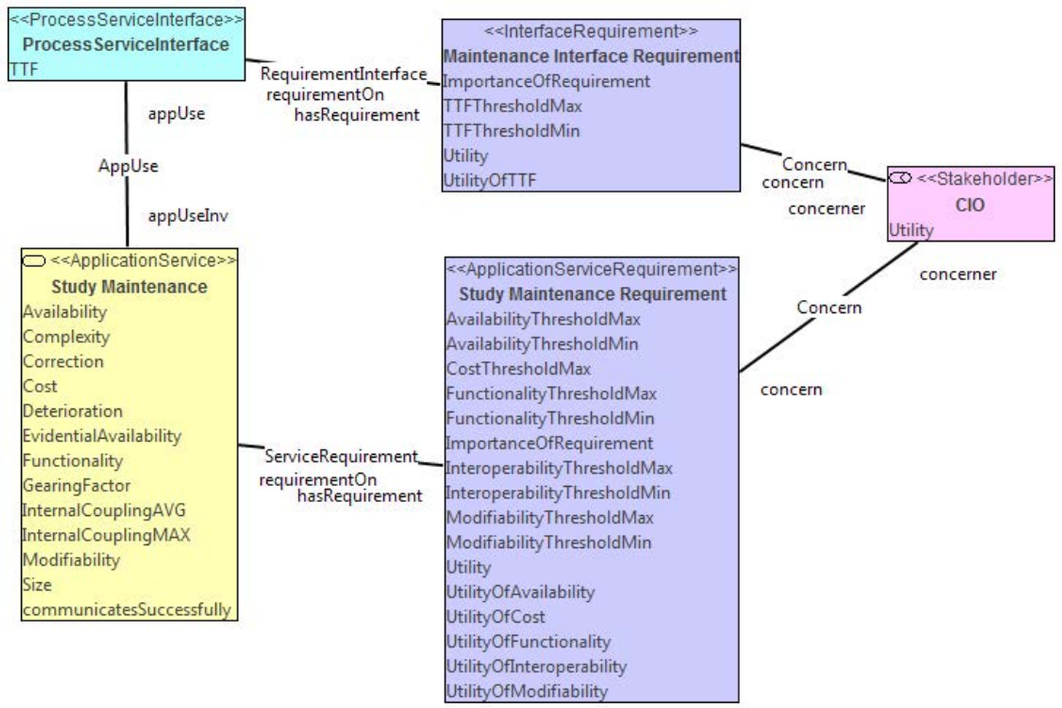


Figure 57: *A utility view example.*

Creating metamodels

Author:
Margus Välja

A guide through the process of creating class models.

This chapter introduces to creating class models using the EAAT Class Modeler.

Chapter 11 Creating metamodels

The EAAT tool and numerous metamodel viewpoints were introduced in earlier chapters. Now we come to the next question. What if we want to analyze attributes that are not supported by the metamodel that was described. The solution here is to create the metamodel by oneself with EAAT Class Modeler, which is another part of the EAAT family of modeling tools [134]. This chapter explains the relationship between the EAAT Class Modeler and the EAAT Object Modeler and contains a short metamodeling tutorial.

The EAAT family of modeling tools consists of two different applications. The first one you are already familiar with from previous chapters – the EAAT Object Modeler. The purpose of the EAAT Object Modeler is to allow modeling real life situations with the help of metamodels. In this book the MAP metamodel was introduced, which is divided into viewpoints for enterprise architecture analysis. These viewpoints are application modifiability, data accuracy, application usage, service availability, interoperability, cost, and utility. In each of the chapters that introduced those viewpoints, an example was included about how to model the viewpoint with the EAAT Object Modeler. We call this type of model object model. The second application of the EAAT family is the EAAT Class Modeler, and its purpose is to create metamodels like MAP, which can be later used with the EAAT Object Modeler. For the purpose of simplicity we will call the metamodels that can be created in the EAAT Class Modeler from now on class models. Anyone can create a class model, but for the class model and corresponding object model to be useful, they have to rely on scientific theories and common sense. Like the theories already covered in earlier chapters for MAP viewpoints. Therefore the tools are meant for different audiences.

11.1 Concerns

The EAAT Class Modeler is designed for creating class models for the EAAT Object Modeler. A class model is an implementation of an analysis theory in UML, OCL and P2AMF that is used for calculation purposes. The calculations based on class models can later be done in the EAAT Object Modeler tool using an object model structure, evidence, and chosen calculation method. While the class model contains theory, the object model reflects some real life situation. The calculation results show values about one or more enterprise attributes in a specific situation that the object model depicts. A summary of the tools and their purpose is presented in Table 7.

Note that while the design purpose of the EAAT software is to analyze enterprise architecture, it is in no way restricted to doing that.

11.2 Stakeholders

The audience for the EAAT Object Modeler tool is enterprise architects, or similar decision makers. The architect that wants an answer to a specific enterprise related question does not need to worry about the underlying theories. He or she can choose a class model or viewpoint and based on that start modeling with the EAAT Object Modeler. After the structure of the model is ready, evidence has been included, the results

Step	Step description	EAAT tool used
1	Define the problem	N/A
2	Choose or create a conceptual model	N/A
3	Turn the conceptual model into a UML, OCL and P2AMF based EAAT class model (metamodel)	Class Modeler
4	Create an executable model that addresses the problem	Object Modeler
5	Calculate results with the executable model	Object Modeler

Table 7: *EAAT modeling.*

can be calculated. These results tell what the underlying theory of class model shows about the enterprise's situation that was modeled.

The EAAT Class Modeler, however, is directed towards more scientific audience. This tool is for those who want to research and create class models of certain enterprise properties that are later used by a wider audience. These can be for example students or researchers. It is easy to create a class model in the EAAT Class Modeler tool, but the worth of the class model depends on the underlying theory.

11.3 The Theory behind the two tools

UML and OCL

The EAAT tools use UML notation for visual representation. The UML entities present are:

- classes that can be instantiated as objects in the EAAT Object Modeler,
- associations and inheritance that are two ways of connecting the classes.

Classes can contain invariants, attributes, operations, and operations can have parameters. Attributes can either be derived or non-derived. The way classes interact with each other and how calculations are done, is defined using Object Constraint Language (OCL). OCL is a declarative language for describing Meta-Object Facility (MOF) based models, and is maintained by Object Management Group (OMG) [135]. OCL is able to be used to describe restrictions for classes and provide object query expressions. In the EAAT Class Modeler, OCL can be used for defining invariants, operations, derived attributes and derived relationships.

P2AMF

Both tools of the EAAT family support uncertainty modeling with the Predictive, Probabilistic Architecture Modeling Framework (P2AMF). By using P2AMF syntax, the uncertainties of objects, relations and attributes can be expressed.

The probabilistic assessments based on P2AMF are possible in the EAAT Object Modeler. The P2AMF syntax is used for generating samples in a chosen calculation process in Monte-Carlo fashion. The calculations are done in the EAAT Object Modeler and the amount of samples generated can be defined in that tool together with the calculation method and numerous other properties.

An example of the P2AMF syntax is the expression of a normal distribution that can be used as evidence or part of operations.

$$myServer.availability = normal(1,0.1)$$

Here the object's attribute named availability is assigned a value from normal distribution, where the mean value is 1 and deviation is 0.1.

An example of the existence property is shown below. It depicts the case of object existence uncertainty and is mandatory for all classes.

$$P(myServer.E) = 0.8$$

Here the syntax tells us that there is an 80% probability that the object myServer exists. This affects the calculation process in a way that there are certain samples that show results without that object.

The syntax and available calculation methods are explained in detail in the EAAT manual.

Analysis in the EAAT Object Modeler

The calculations are possible only in the EAAT Object Modeler tool. There are three sampling algorithms that are supported to infer the values of the attributes that are part of the created model. The ones implemented are: forward sampling, rejection sampling and Metropolis-Hastings sampling, each having advantages and disadvantages. Forward sampling is available also in an extended version, allowing evidence injection.

For all sampling algorithms, the first step is to generate random samples from the existence attributes' probability distribution $P(X) : x_1, \ldots, x_M$. For each sample, x_i, and based on the P²AMF object model, a reduced object model, $N_i \in N$, containing only those objects and links whose existence attributes, X_j, were assigned the value true, is created. Some object models generated in this manner will not conform to the constraints of UML.

The details of the calculation methods are explained further in the latest version of the EAAT manual.

11.4 Guidelines for the EAAT Class Modeler

The graphical user interface (GUI)

The Class Modeler tool consists of the modeling canvas in the center and various other windows that allow to specify details of the model and to see messages generated by the tool. Figure 58 shows the graphical user interface of the tool.

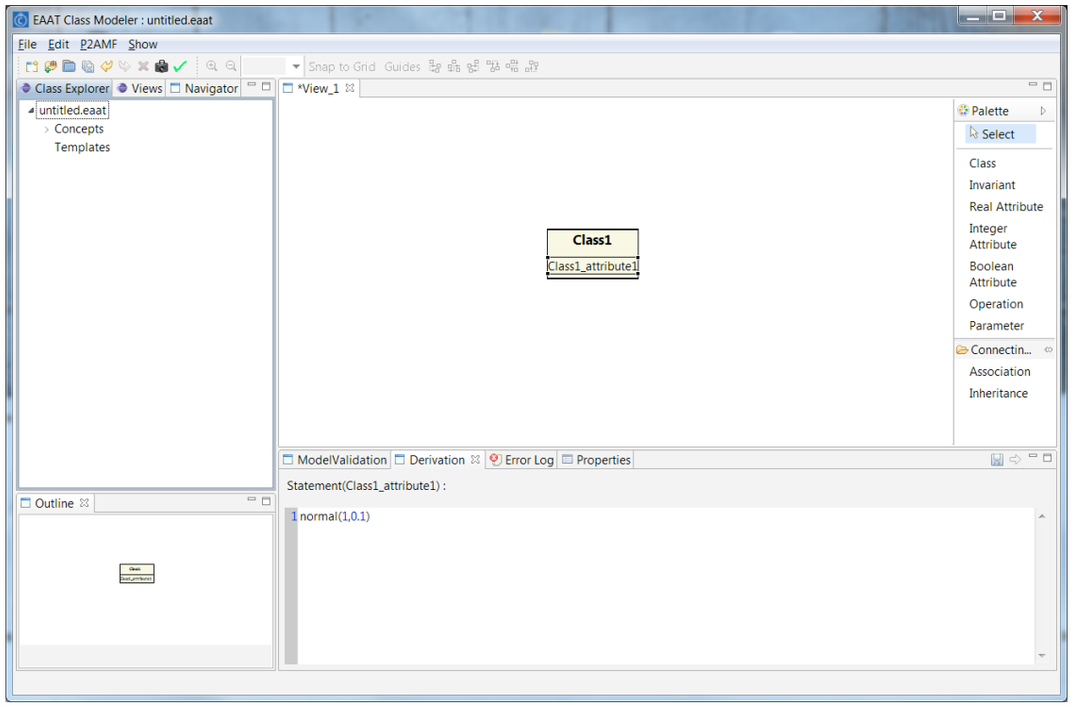


Figure 58: *EAAT Class Modeler GUI with a new class and a new attribute.*

The two important windows besides the main canvas are *Palette* and *Class Explorer*. *Palette* is located on the right side of canvas and by default is hidden. It can be invoked by clicking on the arrow in the right upper corner. *Palette* window lists all the objects and sub-objects that can be used for creating a class model. These are classes, invariants, attributes, operations and parameters for operations. The lower part of the palette shows the ways of connecting the elements – associations and inheritance relationship. *Class Explorer* lists the classes that have been created in the model and allows easy access to them. This is the place to access also created templates. Templates are a way of grouping classes together for easier modeling and comprehensibility in the EAAT Object Modeler tool. The templates can be created only in the EAAT Class Modeler tool.

The details of the class can be defined in the windows directly under the main canvas. After an attribute or an operation of a class has been selected, OCL and P2AMF syntax can be inputted to the *Derivation* window. This code can be validated in the EAAT Class Modeler tool, and the validation results are displayed in the *Model Validation* window. All the elements that can be used for modeling have properties and those can be changed from the *Properties* window, also available below the main canvas. The errors that are not related to model validation are displayed in the *Error Log*.

Complex class models can be divided into viewpoints, while retaining the functionality of the model. A viewpoint should represent an analytical capability of the class model, but doesn't have to. In the previous chapters the viewpoints of one class model that we called MAP, were explained.

Note that invariants and other elements can be hidden in a view. If you don't see an element you think you should, check the view properties by clicking in a random place in the main canvas, so that all elements would be deselected, and by opening the *Properties* window from down below. For example *Invariants* should be set to *true* for them to be visible in the main canvas.

The modeling process

A recommended process for creating a class model (metamodel) with the EAAT Class Modeler consists of 8 steps.

1. **Model classes.** Classes can be instantiated as objects later in the EAAT Object Modeler.

2. **Create relationships between the classes.** A relationship shows that the connected objects can navigate to other connected objects. How this navigation is possible, is defined with the type of connecting element and multiplicity.

3. **Define relevant attributes.** There are two kinds of attributes, derived and non-derived ones. Non-derived attributes are the data that the creator of the object model needs to input as evidence before calculation. Also, we need to differentiate between value types available like real, integers, and boolean.

4. **Define attribute derivation.** Derived attributes contribute to, or show, our analysis results. This is were the most critical part of the logic of the model together with operations is defined.

5. **Define operations.** Operations are used to derive values and aid with the calculation process. They can have arguments. There's an example with a recursive operation in the next chapter, which demonstrates the use of parameters.

6. **Define invariants.** Invariants are for enforcing business rules. They can also be used to exclude samples that don't correspond to some criteria to fine tune calculations.

7. **Create templates.** Templates can group various classes of same type or purpose together. They are needed to make the visual models more comprehensible and the creation and handling of models easier.

8. **Create viewpoints.** Viewpoint functionality is especially useful if a metamodel is large. Then it can be used to create smaller sub-metamodels by hiding certain model elements. Hiding elements does not change the analytical capabilities of the metamodel, but helps with visual comprehension.

The first 4 steps of the process are the most important ones, while the rest depend on the specifics of the theory chosen for the model, and the preferences of the modeler. An actual modeling process might happen in iterative manner, where all 8 steps are repeated in some point of time.

The next section follows the described modeling process. It is a walk-through tutorial showing how to create a class model.

An example class model

The scenario that we choose to model is the following.

In our scenario we have a medium sized company that offers financial services. The company has several web applications that are accessible

only to company's employees. The applications cater to different needs, mostly offering real time reports or data processing services. We have users with various data reading and writing needs that have been divided into role based groups and given access to the applications accordingly. The IT department is now planning to set up a new data warehouse to consolidate data from different applications and needs to know how much space is needed for the new data warehouse. They also want check whether the disk space currently available for different applications is enough according to the number of servers already running. The IT department doesn't know the exact number of employees, nor are certain if applications are used the way they are said to be used. So to incorporate uncertainty and stochastic values, the head of the department has decided to use EAAT for modeling purposes. He now turns to you as an employee of the department and asks first to create a framework of concepts and business logic. You rely on common sense to create the framework.

To create the class model we follow the steps shown earlier.

1. **Model classes.** First we identify what type of objects we need to create in the object model (with the EAAT Object Modeler). In our example, these are a user group, a web application, a data warehouse and a server.

   Create a class for each object in the EAAT Class modeler by dragging Class element from Palette to the main canvas and renaming them with type names. The suggested names are UserGroup, WebApplication, DataWarehouse and Server. If other names are chosen, then these names must be used throughout this tutorial.

2. **Create relationships between classes.** As a next step we need to define basic relationships between the created classes. In the EAAT Class modeler we have two types of connections available, and we choose association, which is the most basic one. Inheritance is explained in the next chapter.

   Create a connection between UserGroup and WebApplication, then WebApplication and DataWarehouse, and finally WebApplication and Server.

   Now set multiplicities. Because a user group has different needs for various applications, the multiplicity between the UserGroup and the WebApplication is 1 to many (*) , where * is on the user group side. We have dedicated servers for web applications, but a web application can be virtualized to run on several servers, so define the relationship between WebApplication and Server as 1 on the WebApplication side and * on server side. Now we come to the last relationship, and here we have several web applications and only one data warehouse. The assumption is that all web applications will be connected to a single data warehouse. Set the multiplicities correspondingly.

   Note that multiplicity determines what type of data will be available by traversing to the connection. It can be either a single value or several values in a set.

3. **Define relevant attributes.** We want to analyze space requirements, so we need attributes to do that. Here we have to divide the attributes into two groups – derived and non-derived ones. For now we just drag the attributes from the *Palette*. Note that there are different type of attributes - real, integer, boolean.

   Add the following integer attributes to the classes. To UserGroup memberAmount, to DataWarehouse differentDatabases. Add the following real attributes to the classes. To UserGroup memberSpaceNeeded, to WebApplication databaseSize and applicationSize, to Server diskSize, to DataWarehouse sizeRequired.

   Note that default values need to be assigned to the non-derived attributes. They must be placed where the statements for the derived attributes are normally put, and can use P2AMF syntax, like *normal(10, 0.1)*, if probabilistic distributions are desired.

4. **Define attribute derivation.** Attributes have properties that can be changed from the *Properties* window, after selecting the corresponding attribute in the main canvas. This way one can change an attribute from a non-derived state to derived one. However, an attribute will automatically change to derived state if OCL code is detected from its *Derivation* field. We chose the following attributes to be derived: databaseSize, diskSize, sizeRequired, differentDatabases, diskSize.

   Add the following code to the attributes, where "--" designates a comment:

   - WebApplication – databaseSize:

     ```
     -- We aggregate the output of an operation totalNeed
     -- for all connected user groups. The operation totalNeed
     -- shows the data needs for a single user group.
     self.userGroup.totalNeed()->sum()
     ```

   - Server – diskSize:

     ```
     -- We calculate the amount of disk size needed for 1 server
     -- as the division results of two operations,
     -- which get the amount of data needed by a web app and the
     -- amount of servers running it
     self.webApplication.totalSize()/self.webApplication.connectedServers()
     ```

   - DataWarehouse – sizeRequired:

     ```
     -- The operation totalSize is invoked and summed for all
     --connected web applications
     self.webApplication.totalSize()->sum()
     ```

   - DataWarehouse – differentDatabases:

     ```
     -- The amount of web applications is counted
     self.webApplication ->size()
     ```

   Note that although comments are not necessary for a class model to function, they are a good way to keep track of functionality and to communicate its purpose to others who want to use the model.

5. **Define operations.** Operations can be used within derived attributes. Operations help the programmer to structure the code. In our example we use operations to calculate the total data needs for user groups and web application and to find the amount of servers providing service to a web application. We already put the following operation names into our derived attributes, assuming they will return certain type of information: totalNeed(), connectedServers(), totalSize(). Create them now in your class model with the following content:

   - UserGroup – totalNeed()

```
-- Amount of space is calculated by multiplying a member space
-- need with amount of members
self.memberSpaceNeed * self.memberAmount
```

   - WebApplication – connectedServers()

```
-- Amount of connected servers is calculated.
self.server->size()
```

   - WebApplication – totalSize()

```
-- Data needs for a web application are calculated by
-- summing user needs with application size.
self.databaseSize + self.applicationSize
```

Note that operations have return values. Set the return value of the operation connectedServers() to integer, others to real.

6. **Define invariants.** Invariants are used to exclude unwanted samples and to make calculations more accurate. In our case, we use an invariant to specify that the amount of space needed for users cannot be negative. We use another one to specify that web application size cannot be negative.

   Add an invariant named spaceNeedNonNeg to UserGroup, and appSizeBiggerZero to WebApplication.

   Add the following content to the spaceNeedNonNeg:

```
self.memberSpaceNeed >= 0
```

   Add the following content to the appSizeBiggerZero:

```
self.applicationSize >= 0
```

   More information about template creation can be found from the next chapter. Viewpoints are covered in the latest version of the EAAT manual. The final class model is shown in Figure 59.

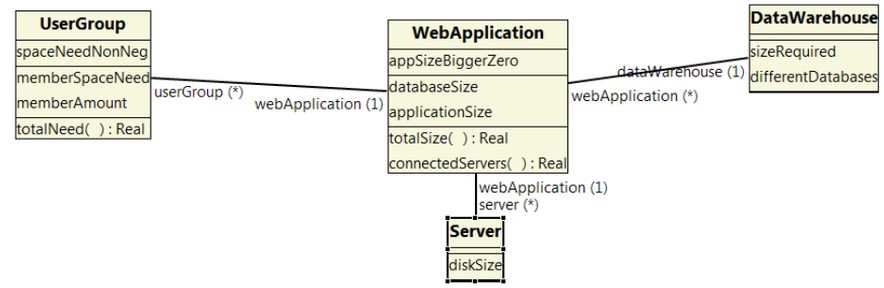


Figure 59: *The example class model*

Don't forget to validate code using the *Validate Model* functionality. If errors are shown, fix them before using the class model.

Congratulations, you have now created a framework for modeling a data warehouse space needs! The model can be used in an object model to obtain the following values:

- Database size needs for a web application
- Disk size requirement for servers, assuming that the space is divided equally between the connected servers.
- Total space required for the data warehouse and the amount of different databases that need to be merged.

Now it's time to try to use this model in the EAAT Object Modeler to model a real life situation. An example object model based on the created class model is shown in figure 60.

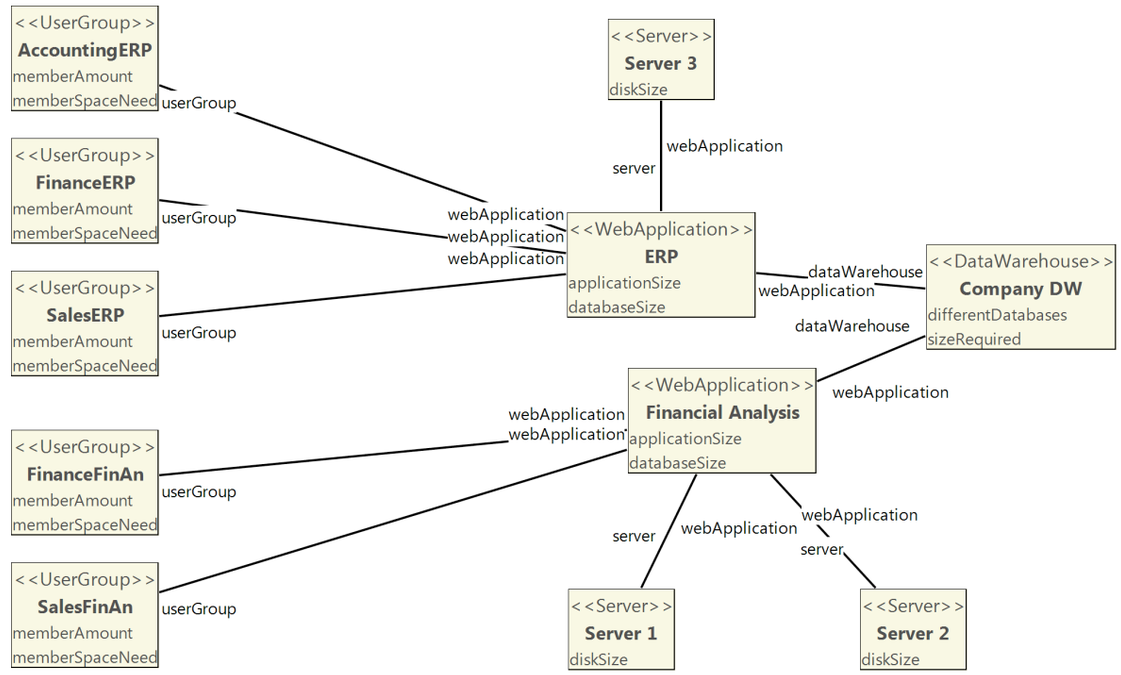


Figure 60: *An example object model.*

Modeling patterns and practices

Authors:
Matus Korman
Margus Välja

12

A collection of modeling patterns and practices that aid you in creating powerful and maintainable class models.

The chapter attempts to address common challenges with class modeling, present reusable solutions to them, and show the application of the solutions in concrete examples.

Chapter 12 Modeling patterns and practices

Creating a class model (an abstract model, a meta-model) of a piece of reality, which defines an automated evaluation and at the same time shall be general enough to allow for accurate modeling of a reasonable range of phenomena of interest within its target domain through object models, is not trivial. Having attempted to model a piece of reality this way, as the authors of the previous chapters have done, the reader surely recalls and understands the difficulty of the task. Moreover, that is just a part of the challenge picture. Another challenge turns up when creating models that are to be maintained over longer time. In other words, creating models and writing their code so that the code remains understandable and readable, requires some discipline and extra work, without which the value of the models is prone to deteriorate heavily in the long run.

This chapter presents a collection of reusable solutions to modeling challenges, also called modeling patterns here. Their purpose is to prepare the modeler for easily coping with a range modeling challenges – challenges that are likely to occur when trying to tailor a class model to capture some view of reality. This chapter also presents a collection of practices, which aim at helping the reader create class models that are easier to read, comprehend, and maintain.

This chapter focuses on tools that have served as the basis for creating the models presented earlier in this book. They are the *Enterprise Architecture Analysis Tool* (EAAT) [17-20], which implements the *Predictive, Probabilistic Architecture Modeling Framework* (P²AMF) [21]. For help on how to use EAAT, please refer to chapter 11 or consult the EAAT user manuals [139].

P²AMF is based on two well-known concepts - Unified Modeling Language (UML) [136] and Object Constraint Language (OCL) [137], which is a part of the UML standard. P²AMF [23, 24] makes use of UML and extends OCL by a few elements, among other the possibility to specify probability distributions such as the normal distribution, or the binomial distribution, including their respective parameters. On top of UML and OCL, P²AMF is a probabilistic framework for quantitative prediction and employs Monte Carlo simulation methods. EAAT is a graphical software tool, which implements P²AMF, and allows a computer user to create class models, object models, and run evaluations on the latter.

At a first glance, even creating class models in P²AMF (i.e., not only modeling concrete architectures according to some existing class model, but actually modifying or creating class models) may seem as a simple and straightforward task. There are, however, a few moments one is better off prepared for. Perhaps the major hill to overcome is to get accustomed with the way programming in OCL is done. OCL is a declarative programming language, and as such it is considerably different from imperative programming languages such as Java, C# or C/C++. Without previous experience with declarative programming, it might take a few tries to gain fluency in OCL. In such case, consulting an OCL guide (e.g., [138]) might be a good start. The OCL specification [137] could also help, and serve well as a reference. When modeling your own concepts, less or

more advanced, it is likely for you as a modeler to also face other types of challenges, however.

The rest of the chapter is structured into two major parts. The first part presents a collection of modeling patterns (i.e., reusable solutions to modeling challenges). The second part describes a collection of coding practices that relate to writing effective and clean OCL code.

Modeling patterns

In the context of this chapter, we define a modeling pattern as a reusable solution to a problem related to modeling or evaluation of object models through means defined in a class model.

We have identified the following candidates for modeling patterns:

- Stochastic values (value uncertainty)
- Stochastic existence of objects and relations (structural uncertainty)
- Class inheritance (child classes)
- Polymorphism
- Aggregation gates
- Self-associations
- Templates
- Derived connections

Following, we present the above mentioned patterns in more detail and with examples.

Stochastic values (value uncertainty)

Challenge/need. Often, our knowledge of a domain is not sufficient for us to create accurate deterministic models. On the other hand, we could have observed that a parameter of our interest varies in a specific range of values in a specific way, for example, reflecting a known probability distribution (e.g. a normal distribution with a known mean and variance).

Solution and benefit. Instead of having to generalize our parameter using its mean value according to our observation, we can specify it in a richer way through the probability distribution it seems to reflect. Let us consider trying to create a class model for traffic safety. Given certain specific circumstances, an accident tends to occur each two months. Given data from our hypothetical observations, there is a rather large variance, since there are periods of several months without an accident, as well as single months, in which several accidents occurred. Instead of trying to assign a static yearly frequency of accidents to six, which yields the same result each time (for each sample), we can specify that the yearly frequency of accidents corresponds to a normal distribution with a mean of six and a variance of nine and a half accident, according to our hypothetical observations. Although stochastic (non-deterministic), our new value definition is richer, because it reflects the observed reality in a more authentic way.

Example implementation. In P²AMF, such an expression reads e.g., "*Normal*(6, 9.5)", instead of simply "6". P²AMF uses Monte Carlo simulation and evaluates models through sampling. For each sample in the evaluation of an object model, the expression will obtain a numeric value that corresponds to the probability distribution. The larger amount of such samples, the more the mean of the sampled values closes in to the six, although the specific values are different from sample to sample.

Stochastic existence of objects and relations (structural uncertainty)

Challenge/need. Imagine that a set of objects are normally connected to another object (such as e.g., a set of transport means to a traveler), but it is uncertain whether they are available or reachable for the object. As a concrete example, imagine an employee of a company who wants to travel to work in the morning. The person has several alternatives to choose between, namely to travel by bicycle, car, public transportation and to walk. Each of the options have implications on the total time duration through the waiting time, travel time, and the time spent by moving between the endpoints of the travel by the transportation means. Also, the means might not always be available - they might be broken down, or not in service for some other reason. Given a set of probabilities for the different time durations and availabilities, let us formulate a question: *How long time is it probable for the person to spend traveling to work?* Let us not only obtain an average value, but an actual distribution of the probability over the different time durations.

Solution and benefit. One way of approaching the problem of uncertainty of object availability is algorithmically in the OCL code. Another and simpler way is through defining stochastic existence of objects on the object modeling level, by setting the object attribute called *Existence* to a real number between zero and one. This number serves as an input to a Bernoulli distribution, which for each sample simply yields whether the object exists (is available), or not. The same (stochastic existence) is applicable to relations, also through their *Existence* attribute, just as with objects.

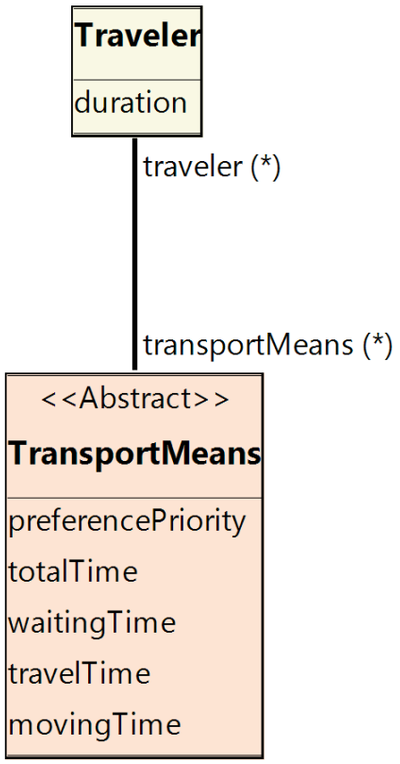


Figure 12.1. *An example class model for demonstration of structural uncertainty.*

Example implementation. For the example mentioned above, we employ both stochastic values and stochastic existence of objects, although we mainly focus on the latter. Let us consider a class model that defines a traveler

and a transportation means, each of them having their attributes defined. The class model is shown in figure 12.1.

The attributes *waitingTime*, *travelTime* and *movingTime* represent the time durations spent on waiting for the transport means to come and start traveling (e.g., a bus), the actual traveling time, and the time needed to move to or from them in order to start or finish the travel, respectively. The total time simply sums up the three previously mentioned attributes. The last attribute in the *TransportMeans* class is *preferencePriority*. It determines how much preferred is the transport means for the traveler. Finally, the traveler chooses the most preferred means out of the ones available in each sample in the calculation. The *Traveler* class has only one attribute, *duration*, which calculates the time spent traveling for each sample. It is defined by the following OCL code:

```
let highestPriority : Real = self.transportMeans.preferencePriority->max() in
let chosenMeans : TransportMeans = self.transportMeans->
                    select( t : TransportMeans | t.preferencePriority =
highestPriority )->
                    asSequence()->first() in

chosenMeans.totalTime
```

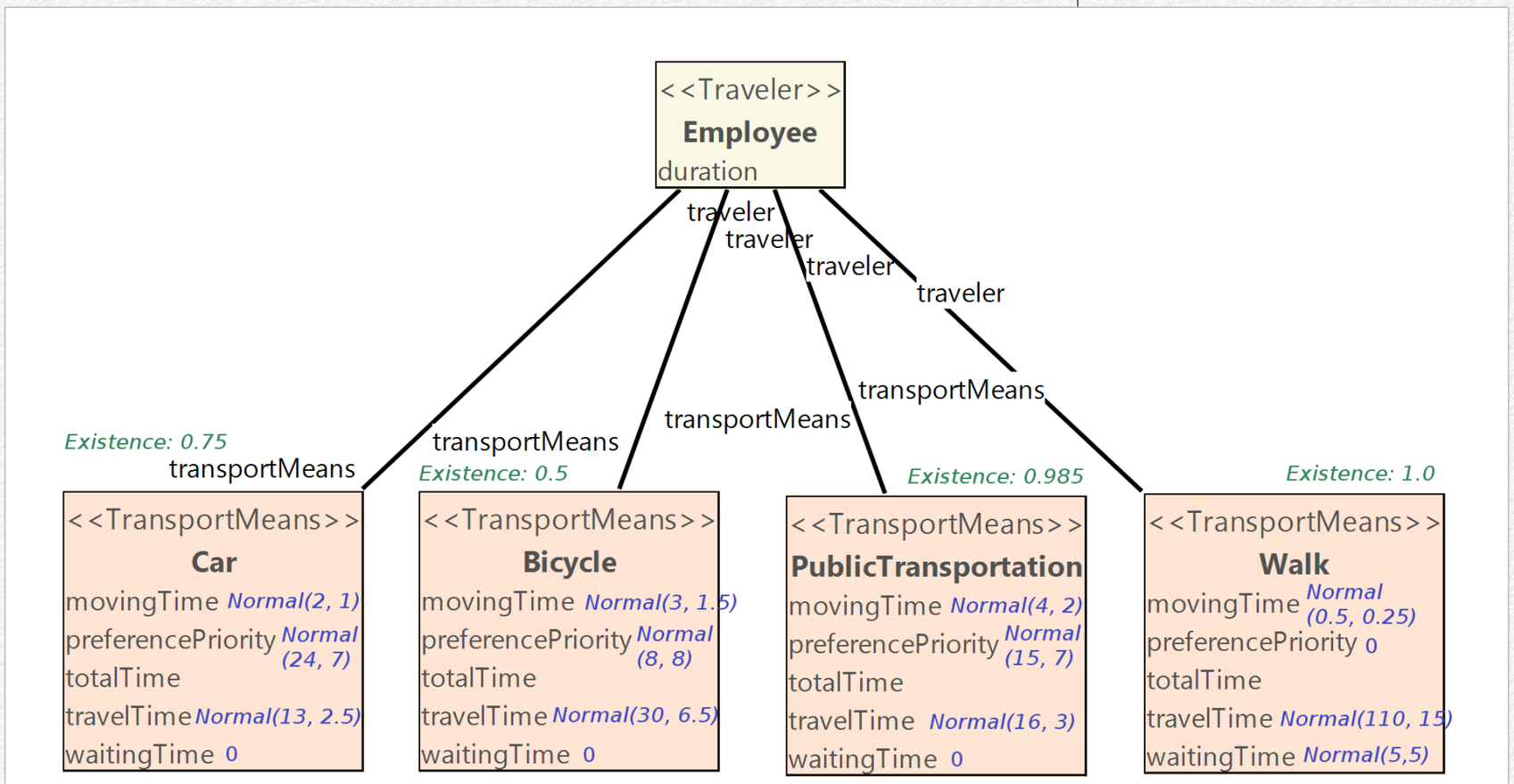


Figure 12.2. *An object model for demonstration of structural uncertainty, with attribute values (evidences) added.*

An object model corresponding to the class model and the example described above is shown in figure 12.2. The time durations are defined stochastically, since they in reality change from case to case, and it is arguably reasonable to assume that they follow a normal distribution. The availability of different transport means can depend on many factors. For example, employees of transport operators may occasionally strike, or something extraordinary may be happening, such as a heavy snowfall, which prevents the public transportation from operating. At the same time, a car may break down, but the traveler's wife or husband

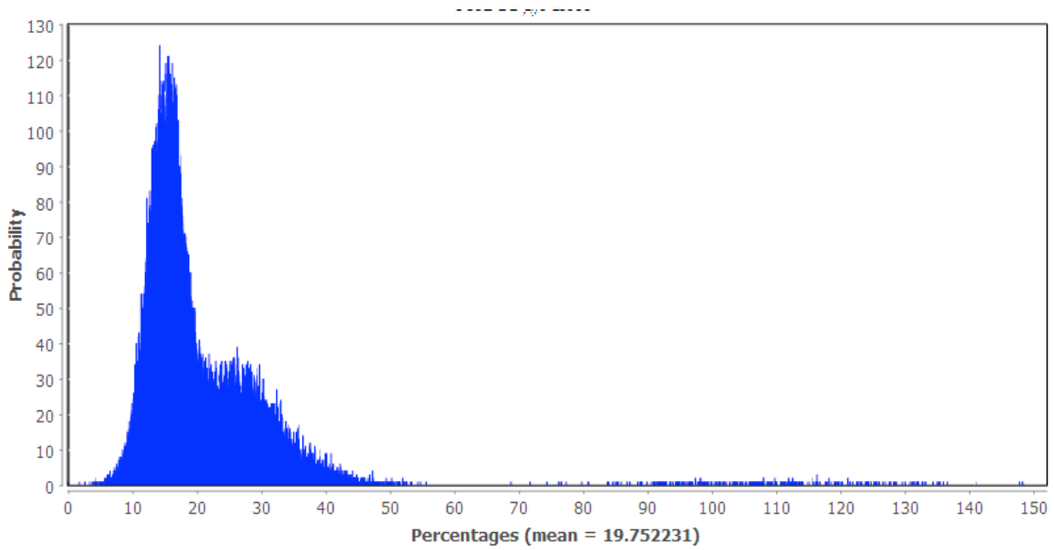


Figure 12.3. *The resulting probability distribution of travel durations from the example.*

may also be using it that day, which makes it unavailable to our traveler. Finally, the traveler may not consider bicycling as an alternative during a rainy day at all. These arbitrary assumptions were used to estimate the attribute values in the object model. The specific values seen in figure 12.2 (i.e., preference priorities and time distributions) were made up freely, not based on empirical data such as measurements or a specific person's answers.

Finally, the answer to our question, the probability distribution of time durations, is shown in figure 12.3.

As shown in the result, the travel would most often take around 16 minutes, sometimes up to around 30 (when taking bicycle). Although quite unlikely to happen, the least preferred case, walking, can also become the only available alternative, taking between roughly 90 to 130 minutes. According to our model and the chosen preference priorities, the traveler would only walk when all of the three more preferred transportation means were unavailable. To sum up, we used stochastic existence of objects to emulate the uncertain availability of the different transportation means.

Class inheritance (child classes)

Challenge/need. Let us suppose that we are creating a model about software deployment. The class model has already become quite large in size, and a few of the classes we use show notable likeness - *software installation*, *operating system*, *software service* and *application client*. Moreover, there are operations defined for each of the classes, which enable evaluation of software-architectural properties. The most straightforward way of implementing the class model is to implement the operations for each of these classes, although the operations are the same from one class to another. That solution, however, would not be very elegant, nor optimal - both with regards to the workload needed, and with regards to the decreased maintainability of the class model: Consider for instance that one finds a bug in one of the operations - one then needs to rewrite them all across the different entities.

Solution and benefit. A more elegant solution to the above mentioned challenge is to use class inheritance (i.e. is-a relationships between classes), so that we can set up inheritance relations between one class (called a base class, or a parent class) and others (the parent's child classes), and implement the otherwise redundantly defined operations

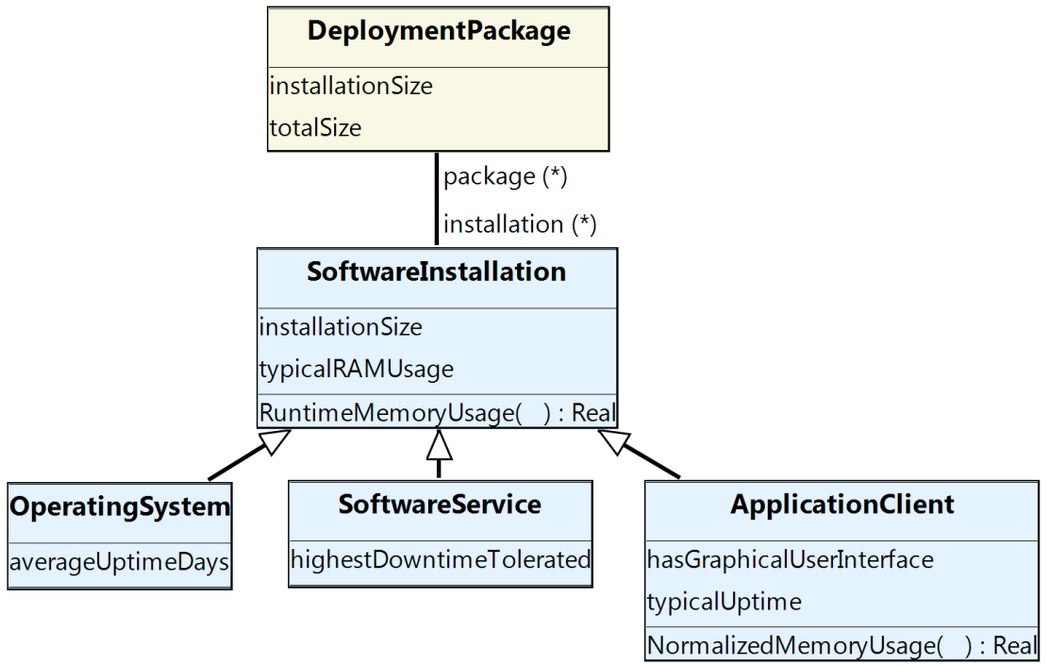


Figure 12.4. *Example of class inheritance (the lower part of the figure),*

and attributes within the base class. Consequently, both the operations and the attributes defined in the base class will be defined and accessible for the child classes, too. Thus, we minimize the amount of work to perform when modeling, and eliminate the unnecessary redundancy in the model, which keeps it more tidy and clean.

Example implementation. The implementation of inheritance can be shown using the example classes mentioned earlier. Let there be a base class (e.g., *software installation*) that defines and implements the operations common for all software installations. Let then the other classes (*operating system*, *software service* and *application client*) inherit from the *software installation* class. These child classes can define further operations upon need, while on any of them; any of the base classes' operations can be defined. The same applies to attributes, as well. This is depicted in figure 12.4. The inheritance relation is denoted by a non-filled arrow pointing at the parent class (following the UML notation [136]).

Polymorphism

Challenge/need. Let us suppose that we have a collection of classes that inherit from a single base class. Let us further suppose that the base class defines an operation, which is therefore also defined for all of its child classes, but the operation at child classes calculates the result slightly differently from one another, dependent on the specifics of the actual subclass. As a more concrete example, consider a base class called *Car* (an arbitrary car), which defines an operation *CalculateFuelConsumption* returning the fuel consumption in liters per hundred kilometers. The operation takes in three parameters as inputs - *rpm* (rounds per minute of the engine), *appliedTorque* (what torque the engine currently applies to the shaft) and *velocity* (how fast the car moves). There are a set of different cars with different engines our model needs to consider, but the *CalculateFuelConsumption* operation is equally needed for them, requires the same input parameters and produces the same type of output. On the other hand - in different cars the consumption is calculated using different constants and in some cases perhaps even different formulas. Now, we want to call the operation (*CalculateFuelConsumption*) for an arbitrary set of subclasses (specific cars) to the base class (*Car*). It is clear that the logic for the operation that calculates the consumption cannot be elegantly writ-

ten in the body of the base class, because each time a new car is added, removed or modified in the model, the base class itself would have to be modified, as well as the code of the operation implementation would be confusingly long, unnecessarily complicated and thus little comprehensible. Such implementation tends to be error-prone and difficult to maintain.

Solution and benefit. A solution to the above mentioned problem is to use polymorphism: We only access the objects as instances of the base class, and so access the value of the attribute for each object. This way is opposed to accessing the objects as instances of the multiple different subclasses, which requires more work, more code, and is more error-prone. Put even simpler, we can temporarily ignore the fact that the objects are instances of the different subclasses as long as the subclasses inherit from a common base class and we do not need the specifics defined by the subclasses. In our case, we only want to access an attribute that all of the classes inherit from the base class. Hence, we heavily simplify the way to access and read the values we need.

Let us come back to our concrete example from the world of cars. Our logic to calculate the fuel consumption for all the cars supported by our model can be implemented in the body of each of the classes corresponding to a specific car model, all of which inherit from our *Car* base class. The specific class' implementation of the operation simply overrides the implementation of the equally named base class operation (for all objects of the specific class, including other objects that eventually further inherit from the specific class). Hence, when we call the operation to obtain its result, we do not need to distinguish what specific car model (specific class) the object we are just dealing with corresponds to. We know that it inherits from the *Car* class, and therefore has the *CalculateFuelConsumption* method implemented - we access it only as an object of the Car base class. The OCL interpreter (or the EAAT tool, taken more generally) implicitly invokes the appropriate calculation with regards to the specific object being accessed (the calculation defined in its corresponding specific class). Hence, if we simply access a *Car* object that happens to be is a Volkswagen Golf GTI with 2.0 TSI engine and a manual transmission, we will get the consumption for this car model instead of some other model (say, Toyota Aygo+ 1.0), or a dummy calculation defined by the base class, on the level of which all details required to calculate an accurate consumption estimate are not known.

Example implementation. To implement a model for the above described sample scenario, we first create a *Car* class (in the class modeler). Within the *Car* class, we create an operation called *CalculateFuelConsumption*, which outputs a real number and takes three real inputs (*rpm*, *appliedTorque* and *velocity*)... or some other set of inputs as appropriate for the specific modeling purpose. Subsequently, we write some default (dummy) value this operation returns (e.g., zero), since at this point, a valid estimate of [an unknown car's] consumption cannot be made. Now, we can start to implement specific car models. For each such, we create a class, name it, and relate it with our *Car* class using an inheritance relation (i.e., our specific car inherits from the *Car* class). In our specific car class, we need to define an equally named operation *CalculateFuelConsumption*, which takes in the same set of inputs and produces the

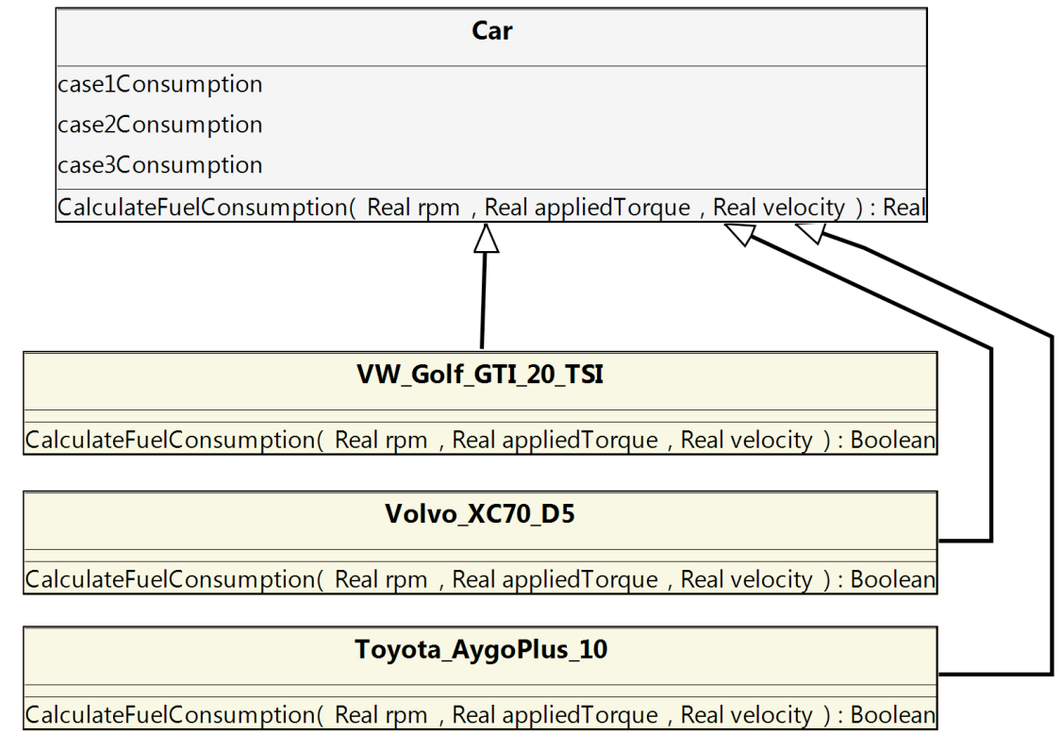


Figure 12.5. *Example implementation of polymorphism (in class modeler).*

same type of output as the *Car* class' operation. This operation, however, is implemented validly (fully), and actually calculates the estimate according to some formula and surely a set of constants, too (defined within the calculation). For the polymorphism itself to work in an object model created on this class model, this would suffice. In order to show (visualize) the results in a simple way however, we define a few attributes on the *Car* class, which simply call a current object of the *Car* class' *CalculateFuelConsumption* operation (e.g., **self**.CalculateFuelConsumption(1800, 30, 50)), each time with different

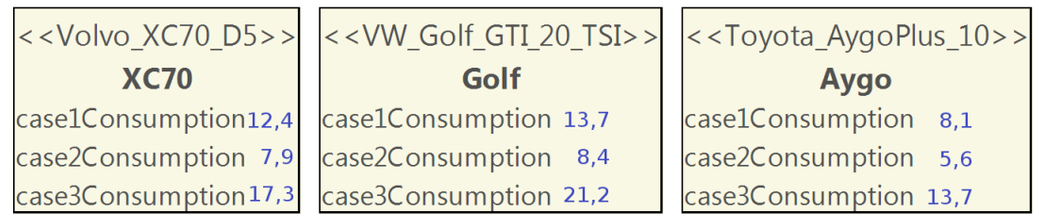


Figure 12.6*: Example instance model of fuel consumption of cars (in object modeler). Each attribute has different calculation results for each car, because the implementation is different for each specific car, although invoked by the same code and in the same way. The consumption values are purely hypothetical, not based on evidence.*

input parameters - just to see that our calculations actually work. The class model should then look similar to the one depicted in figure 12.5.

In order to see some calculations and how the polymorphism works, we need to run object modeler, load the class model, simply instantiate three specific cars and calculate. The model should look similar to the one depicted in figure 12.6.

Given that different non-equivalent formulas were used for the consumption estimation in the operation and that the attributes on the *Car* class (e.g., *case1Consumption*, …) call the operation with different parameters that should yield non-equal results, each attribute will have different calculation result for different cars, and even the different attributes calculated for the same car will differ. To recapitulate slightly, polymorphism is now being used in the definition of the attributes of the *Car* class. The attributes call the *CalculateFuelConsumption* operation of a car object, the

specific class of which is unknown in that context. Even though, the OCL runtime engine in EAAT chooses the appropriate calculation implicitly, returning the appropriate result.

On a final note, polymorphism works this seamlessly in EAAT only when overriding operations, not derived attributes. In case one would like to override the derivation of attributes, explicit code would be required at the base class, which would distinguish the specific type of the subclass, and call the operation from the object type-casted to the specific class. This would, again, not be particularly elegant. Overriding attributes in that way can be avoided through a design that overrides operations instead (and achieves the same computational result).

Aggregation gates

Challenge/need. Imagine that we want to write a class model for evaluation of reliability using a simplified variant of a fault tree analysis (FTA). Put simply, a node can connect to other nodes, on which it depends. The dependence can be of different kinds, such as when the upper-level node depends on all of the other nodes, or the function of one of the other nodes is enough to satisfy the function of the upper-node. The former is commonly denoted by an AND-dependence relationship and the latter by an OR-dependence one. Suppose that you want to model a dependency structure consisting of multiple such levels, each having an upper node and one or more lower nodes, on which the upper node depends. Imagine for instance a simple model of an airplane. For the airplane to be safely operable, a number of conditions have to be satisfied. The airframe needs to be firm enough to hold the tensions it is exposed to during a flight. At least one of two engines needs to be operational in order for the plane to stay airborne. At least one of three brakes needs to be operational in order for the plane to safely land and stop. On a level below this, air supply system has to be operational as well as there has to be some fuel in the plane for an engine to work. And so on.

Solution and benefit. A solution is to use AND- and OR-gates, which help us model the aggregations, so as to reflect and evaluate the simplified FTA dependency structure. Since the nodes can also model other than AND and OR relations, such as XOR, priority AND et cetera, let us simply call the gates aggregation gates. The benefit is that we can aggregate dependencies between objects in a tidy and hierarchical way.

Example implementation. An example implementation of the above mentioned solution is depicted in figure 12.7. There are three classes. First, a *node* represents a system, a function, a service, or simply something that can work, or can be broken. Second, an *AND gate* requires all nodes connected to it as *lowerNode*s (that "feed in" to the gate) to work, in order for the gate to work (i.e., be satisfied). Third, an *OR gate* requires at least one of the nodes connected to it as *lowerNode*s to work. Each gate needs to have at least one lower node, and exactly one upper node. An example of an object model (instance model) is shown in figure 12.8. For completeness, the attribute *Node.works* is derived using the following OCL code:

```
if (self.lowerAndGate->size() = 0 and self.lowerOrGate->size() = 0) then
    -- it is a leaf node and should not be derived; either it works or it does not
    let thisWorks : Boolean = bernoulli(0.5)
    if thisWorks = true then 1 else 0 endif
```

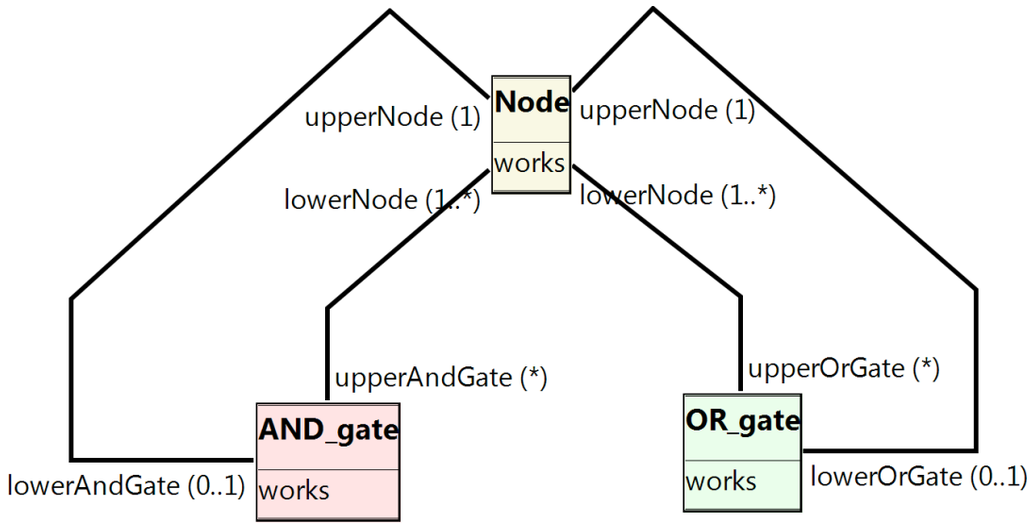


Figure 12.7. *Example implementation of AND- and OR-gates (in class modeler).*

```
else
        if (self.lowerAndGate->size() > 0) then
                -- there is an AND gate attached... is it satisfied?
                if (self.lowerAndGate.works > 0) then 1 else 0 endif
        else
                -- an OR gate is attached... is it satisfied?
                if (self.lowerOrGate.works > 0) then 1 else 0 endif
        endif
endif
```

Similarly, *AND_gate.works* is derived as **self**.lowerNode.works->min() and *OR_gate.works* as **self**.lowerNode.works->max().

In a concluding note, it is clear from figure 12.7 that a node is technically allowed to have a lower AND gate and a lower OR gate at the same time. Such state is, however, erroneous with regards to the FTA logic. Al-

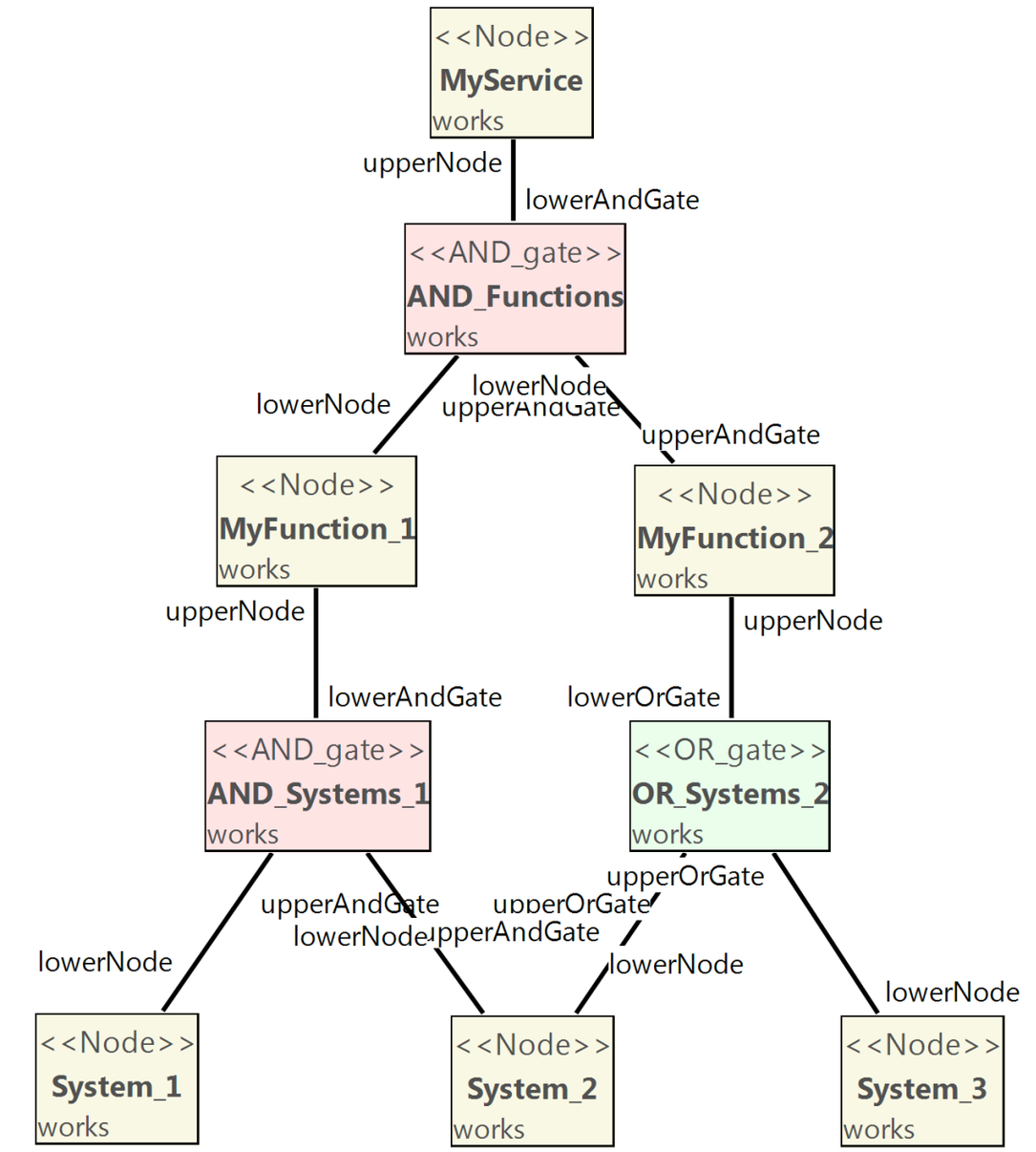


Figure 12.8. *Example object model of nodes and gates (in object modeler).*

though this case is not safeguarded in the example provided above, a

possible solution is to introduce a base class called *Gate*, from which both *AND_gate* and *OR_gate* would inherit. Then a relationship between an upper node and a gate could be set to (1)..(0..1), as well as the relationships between an upper node and a lower [AND/OR] gate removed. This would also be an example of using polymorphism as a necessity for such a solution. Yet another solution to the model in figure 12.7, perhaps even more elegant, could be to simply add an OCL constraint within the context of the *Node* class, stating that at most one gate can be connected as a lower gate at a time.

Self-associations

Challenge/need. Let us suppose that we need to create arbitrary hierarchies between objects of the same class. For a more concrete example, let us consider modeling the encapsulation of protocol data units (PDUs) in data communication in a computer network. A PDU typically contains of a header part (meta-information typically providing addressing information, data length, etc.) and a data part (the actual payload that is being transmitted by the PDU). Following the Open Systems Interconnection reference model[1] (RM OSI), different protocols which provide network communication functions on different abstraction levels, encapsulate the PDUs of each other. More precisely, lower-level protocols encapsulate the PDUs of higher-level protocols. Such encapsulation means that the whole higher-level PDU (both its header and its payload) becomes just the payload for the lower-level PDU, while a header for the lower-level pdu is created and added before the new payload. For instance, when an HTTP request (data) is produced by a web browser, it is first encapsulated into a TCP segment. The TCP segment is then encapsulated into an IP packet. Further, the IP packet is encapsulated into an Ethernet frame, which is then transmitted through cables in a binary fashion using physical signals. Each PDU except the HTTP one, which is at the top of the stack in this example, encapsulates one higher-level PDU in itself, only treating it as its data (payload). The question is how to model such PDUs and their encapsulation that both need to be defined at the level of instance modeling. Abstractly speaking, the problem is that we can't pre-create a definite structure in the class model, because it could, and in fact certainly would, be too inflexible to satisfy our object modeling needs. Thus, on the level of class modeling we only need to create a frame to enable structuring further at the level of object modeling.

Solution and benefit. There is a simple solution - associations of a class with the class itself. The benefit is that arbitrary hierarchies that are just "frame-wise" defined in the class model, can be defined in the object model. In other words, multiple objects of the same class can be hierarchically ordered or otherwise structured in the object model. As a concrete example, different PDUs (all objects of the *PDU* class) can be modeled as encapsulating each other so that the Ethernet frame encapsulates the IP packet, which encapsulates the TCP segment, which finally encapsulates the HTTP request.

Example implementation. Let us try to model the above mentioned PDU encapsulation. We create a *Protocol* class, which represents the protocol used (e.g., HTTP, TCP or IP). We also create a *LogicalNode* class, which represents an application or a device that is able to operate with

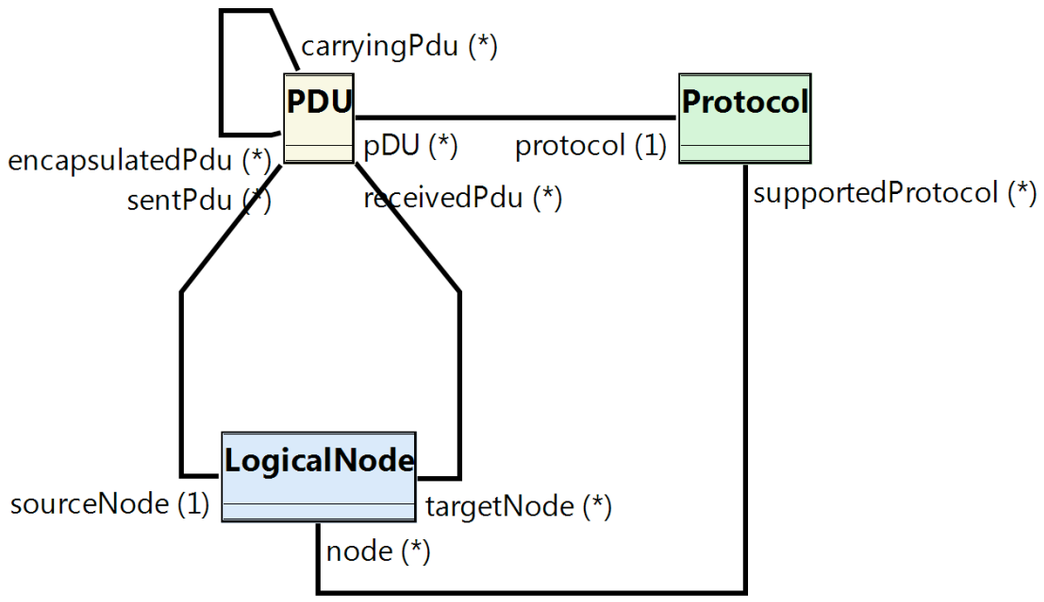


Figure 12.9. *Example implementation of a class model using a self-association (on the PDU class).*

PDUs. Finally, we create a *PDU* class, which represents an instance of a PDU. A logical node has to support a protocol in order to process PDUs corresponding to the protocol. Hence, we add such association between *LogicalNode* and *Protocol*. As we know, each PDU corresponds to a specific protocol, so we add a association between them, too. Since a PDU is created and sent, as well as intended to be received by some addressee, we create two associations between *PDU* and *LogicalNode*. The first association represents the source node for a PDU (the originator), and the other a set of targets (intended recipients). Finally, we add a self-association of *PDU* with itself, in order to be able to model that some concrete PDU object encapsulates other PDU objects (later in our object models). Having performed these steps, the class model should look similar to the one depicted in figure 12.9.

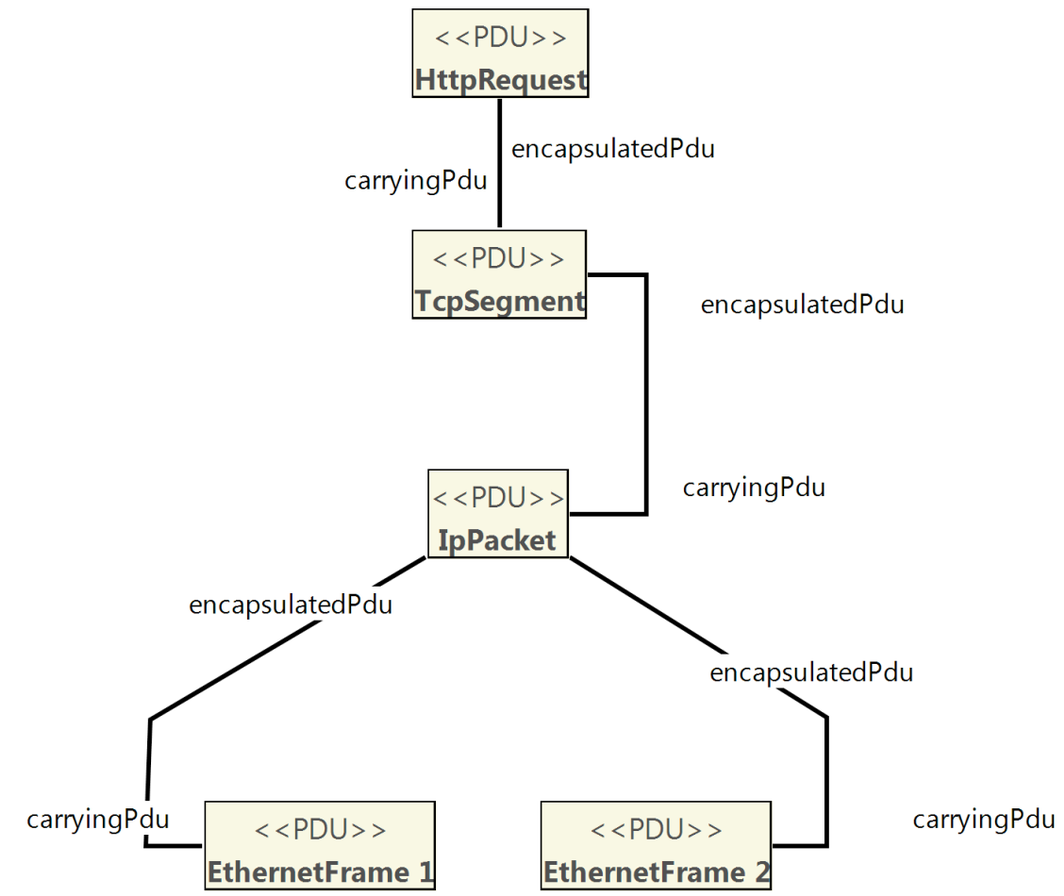


Figure 12.10. *A simplified object model describing structures between objects of the same class (PDU) defined on the level of object modeling. The diagram only contains objects of the PDU class, not their connections to objects of other associated classes.*

An object model corresponding to the described encapsulation case could look similar to the one depicted in figure 12.10 (simplified view) and 12.11 (more comprehensive view). The choice and interconnection of

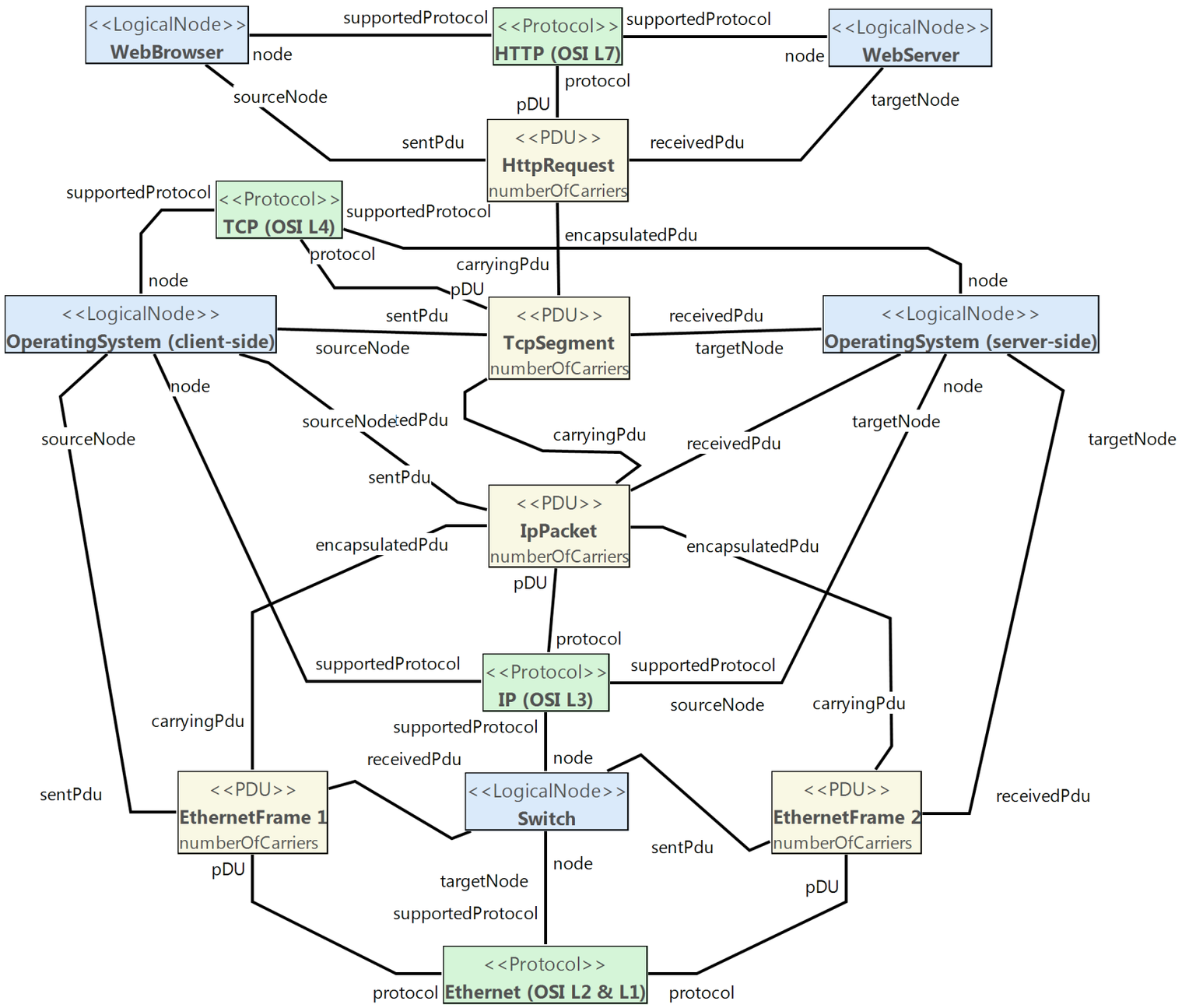


Figure 12.11. *A more comprehensive version of the same object model as in figure 10. Somewhat exotic style of the association lines in the diagram was chosen due to a minor technical issue that is not detailed here.*

objects reflects the description of how a HTTP request gets encapsulated down to an Ethernet frame (RM OSI layer 1 and 2). As a peculiarity, there are two ethernet frames encapsulating the same IP packet. This is the case because we consider that a single layer 3 switch is placed between the client computer on which the web browser runs and the computer on which the queried web server runs. Since each of the nodes (computers or the switch) have their own Ethernet addresses and the network is connected through the switch using a star topology (i.e., there is no direct Ethernet connection from the client computer directly to the server computer), first a separate Ethernet frame flows from the client computer to the switch, and then the switch forwards the IP packet under the frame further to the server computer, by creating a new Ethernet frame with the same data, but an Ethernet header that differs from the previous one in that the frame targets the server computer).

Templates

Challenge/need. Imagine a need to model a complex architecture with many repeating elements. Doing it in the same fashion as was used in exam-

ples described above, the modeling could easily become tremendously time-consuming, boring and error-prone. Moreover, it could make the model very difficult to modify. Put simply, it would be a poor and impractical approach to such modeling problems. Let us illustrate the idea on the following example. Consider modeling a company IT environment with the aim to evaluate its cyber-security disposition. Since the domain of cyber-security is very broad, a truly comprehensive model would be very extensive and detailed, even for small IT environments. For simplicity in this illustrative case, let us therefore limit ourselves to a few aspects only. Let us only consider applications, services, operating systems and network interconnections in a company. The company has a few services being run on servers located within the company, as well as a few offices, where each employee has a workstation or a laptop, a mobile phone and eventually also a tablet. All of these inter-networked IT devices run an operating system, have a variety of applications and services installed, and some of these have access to data or services that are considered sensitive for the company (e.g., e-mails or a central file sharing repository). If we are going to model each such device including some specifics of their operating systems, applications and services, there is going to be a great deal of repetition. For example, a typical workstation running Microsoft Windows 7, Apple Mac OS or some Linux distribution comes with a number of broadly used applications, as well as a number of services pre-installed and activated by default. Similarly, this applies to server systems. In any such case, we are talking about tens of elements per such system that do not change much from one workstation or server to another. Now consider having to model all these repetitively, say fifty times. That would be clearly annoying and wasteful, to say the least. We need some automation here. We need a feature that would compress the amount of this trivial work, or else we need to literally waste our time. The larger our models and the more repetition in them, the more trivial, boring and error-prone work we need to do, which in addition leads to heavily decreased maintainability of the models.

Solution and benefit. A solution is to define and use templates on the level of the class model, which are able to "box in" a set of entities and their associations, which the modeler can use as a single box instead of explicitly modeling instances of everything what is inside the box (the template). If the template changes, one only needs to apply the change to the template, not necessarily to its multiple instances in the object model. This is a huge benefit both as to simplicity and efficiency of modeling, as well as maintainability of such models over time.

Example implementation. An example of a simplified class model that would help us to model an IT environment can be constructed as follows. There is an entity *NetworkDevice* (any device capable of networking), which can have many *NetworkConnection*s (connections between network devices) and vice versa. This allows modeling one network device as connected with other network devices through network connections. *NetworkDevice* can be further specialized as *Computer* (an advanced, typically multi-purpose, computational device) or *NetworkConcentrator* (e.g., a network switch). A computer has an operating system, which can host three different types of software - applications, services and libraries. In

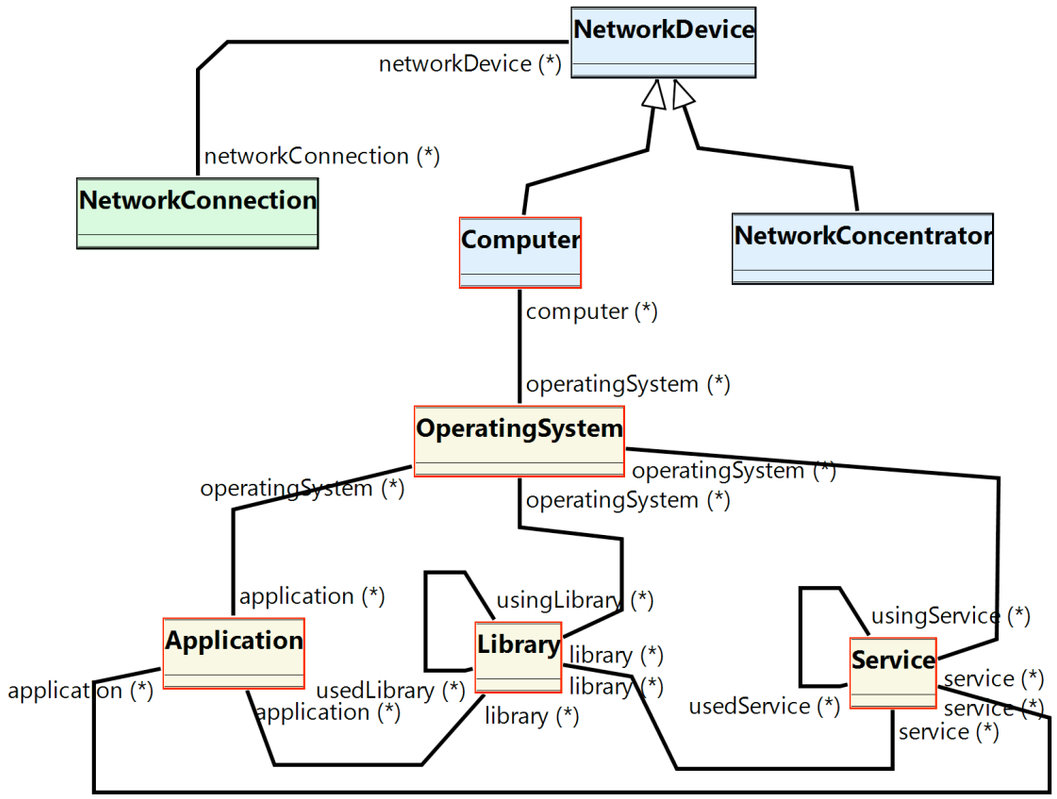


Figure 12.12. *An example class model for modeling an IT environment (computers, operating systems, software and network interconnections).*

our model, these can mutually use each other, too. The class model described above is depicted in figure 12.12.

As mentioned before, modeling applications, libraries and services of each operating system that is run by each computer we wish to model, would make up to an unnecessarily extensive model. What we can do instead, is to define templates in the class model, according to the probability (our certainty) that the templates will save a lot of unnecessary

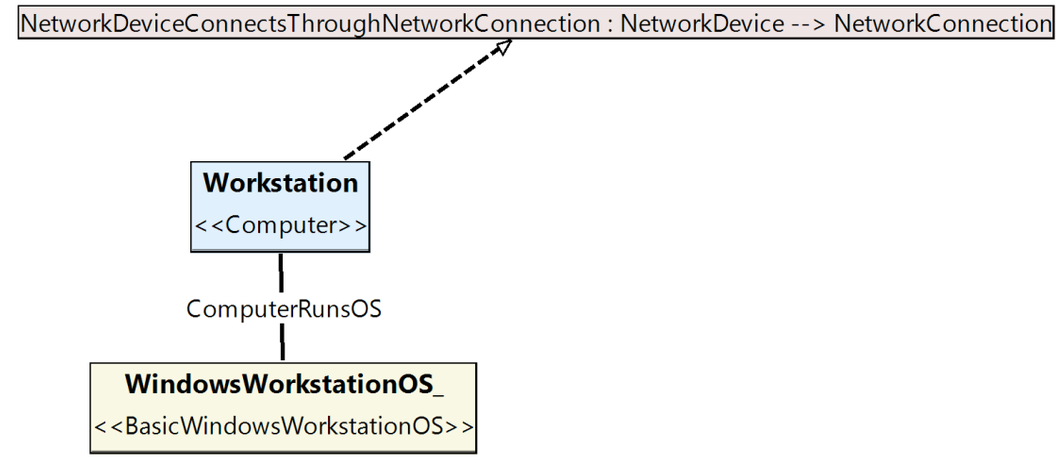


Figure 12.13. *Definition of a template called WindowsWorkstation, in class modeler. Note the gray rectangle with text, to which an arrowed dashed line points. It is a so called port; an external interface of the template that makes it connectable to in an object model.*

work, provide for more elegance and maintainability. Two example templates are depicted in figures 12.13 and 12.14. The former, called *WindowsWorkstation*, simply wraps a computer and its respective operating system (itself a template). The latter, called *BasicWindowsWorkstationOS* defines the applications, services, libraries, and their mutual usage, so that all that can be used as a single box anywhere we wish to use it - in the object model, or in other templates we choose to define. As is readily apparent, templates can use other templates that are defined. Cyclic use of templates, however, cannot work. Having a case in which template 1 uses template 2, an attempt to use template 1 from template 2 (creating a dependency cycle), would be erroneous.

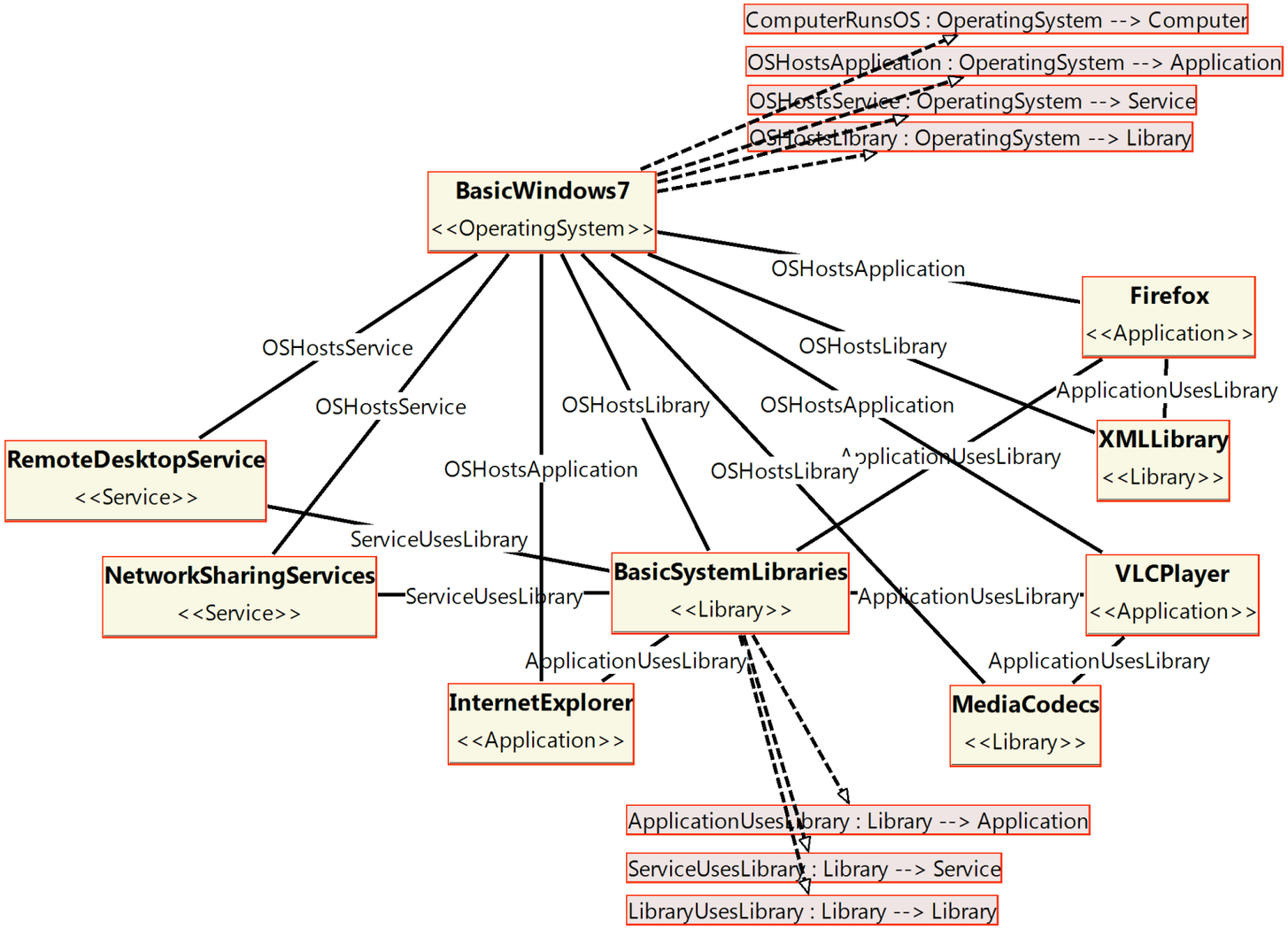


Figure 12.14. *Definition of a template called BasicWindowsWorkstationOS, in class modeler.*

Let us now model a simple IT environment consisting of three networks - one office LAN (local area network), one server network, and one DMZ (demilitarized zone). In the office LAN, there are a few workstations, some of them connected by network cables, the rest using a wireless connection. In the server network, there is a domain server and an internal file server. Finally, the DMZ hosts systems with increased exposure to the Internet, such as web server and e-mail server. This architecture modeled using a few templates, is shown in figure 12.15. Omitting templates and modeling all basic entities in every single instance of a server or workstation, the object model would consist of around sixty boxes instead of just nineteen[2]. That makes a difference. The difference becomes even clearer when modeling larger IT environments.

Derived connections

Challenge/need. Imagine that there are two classes, which are associated. Imagine further that you wish to associate objects of the one class with objects of the other class based on a condition, which depends on values of the concrete objects. It is clearly desirable to have some automation in the process of connecting the objects, so as to avoid tedious manual work. Although a collection of objects satisfying such a condition can be made accessible purely algorithmically through OCL, this comes at the expense of somewhat higher code complexity and lower comprehensibility. That might render the model slightly

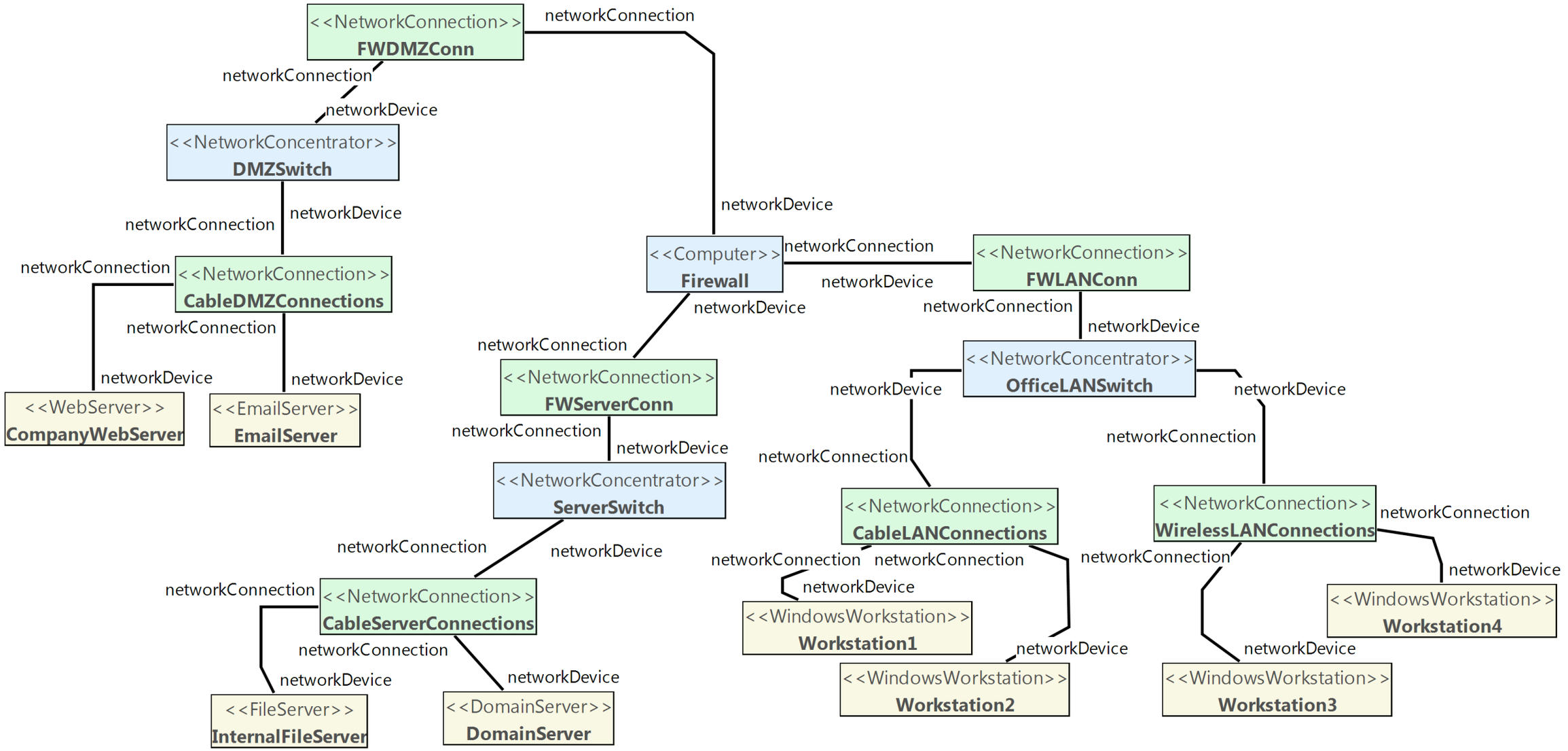


Figure 12.15. *An example object model using templates (at all leaves in the "graph" of entities above).*

more difficult and thus more error prone to develop and maintain.

Solution and benefit. Derived connections provide an elegant solution to such problems, which would otherwise have to be solved through some filtering or search implemented in an OCL operation. A derived connection is generated on the level of calculation (evalua-

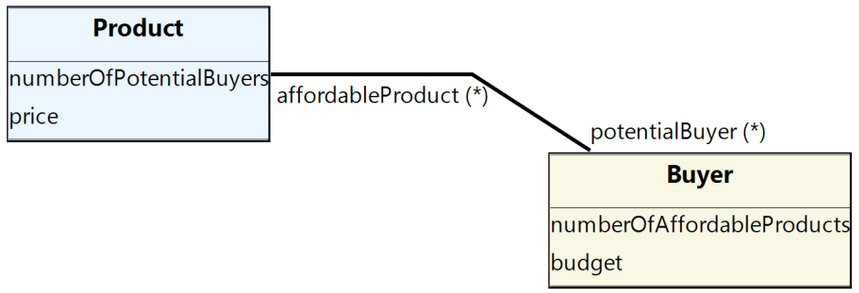


Figure 12.16. *A class model defining derived connections (in fact in form of associations between classes).*

tion) of an object model. On the level of class modeling, it has the form of an association with a derivation expression in OCL. Defining derived connections has some peculiarities, however. Such an association (that defines derived connections) is unidirectional. Hence, one has to choose one end of the association as a starting point for the derivation (or keep a default one), and write the OCL expression for the derivation accordingly. An OCL expression for the derivation having its starting point at one class, is supposed to return a set of objects of the other class. A concrete case is described below.

Example implementation. Consider a very simple scenario. Let us have a set of products and a set of buyers. Each product has a price and each buyer has a budget for shopping. Now, we wish to define an association between classes, which would act as a derived connection between objects, so that each buyer would only connect to the product he or she has the budget to buy. In a similar fashion, each product would connect to the buyers that have a real potential of buying the product, given their budget and the price of the product. The class model can look as follows in figure 12.16.

The derived connection is created as an association, the derivation of which is defined by an OCL-expression. In this example, it is enough to define one association to model both the product affordability and the buying potential. The OCL expression for the association (the derived connection) can be defined in two directions - as a collection of objects from the one class (*Product*), or from the other (*Buyer*). In this example the former is true, and the expression reads as follows:

```
-- from all products, reject those more expensive than the buyer's budget allows to buy
Product.allInstances()->reject( p : Product | p.price > self.budget )->asSet()
```

If it was defined in the other direction (from *Buyer*), it would read as follows:

```
-- from all buyers, reject those that have smaller budget than the product's price
Buyer.allInstances()->reject( b : Buyer | b.budget < self.price )->asSet()
```

In this case, both of the above mentioned alternative definitions yield equal sets of connections between objects of the classes.

Each of the two classes has two attributes. One is defined on the level of object modeling, and the other simply obtains the number of potential buyers for a product, or the number of affordable products for a buyer, respectively. The other attribute can be derived as follows (for buyer):

```
self.affordableProduct->size()
```

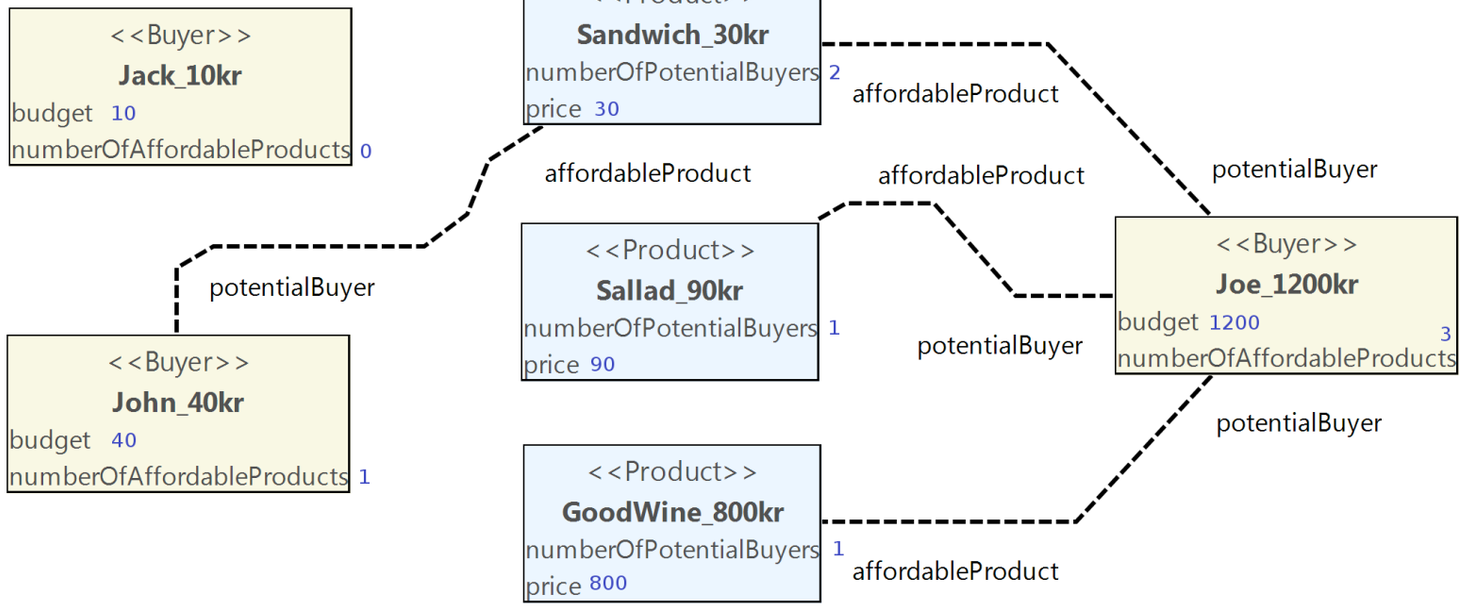


Figure 12.17. *An object model showing derived connections and the values of the visible objects' attributes.*

An object model corresponding to the above described class model is shown in figure 12.17.

OCL coding practices

In the context of this chapter, we define a coding practice as a specific way of writing program code (i.e., OCL code for P²AMF class models).

One may ask why or how something like a coding practice matters to those who model and those who have responsibility for the models. Although there could be a broad discussion on this topic, large body of experience from the practice of software engineering may shed sufficient light on it. Perhaps even your own personal experience. In any case, the authors see two major challenges that often make the use of coding practices reasonable. The first challenge comes in form of entropy (disorder) that keeps deteriorating the orderliness of our models over time as we keep changing them (e.g., due to maintenance), but also our knowledge about them (i.e., we are forgetting over time). It ultimately leads to that the models become too messy and difficult to maintain, or otherwise unusable, so that we throw them away and start from scratch, or try to address the needs they fulfilled in some other way. The second challenge is that we tend to come about great ideas rather seldom out of the blue. Our readiness to write a proper piece of software (or a model) at a reasonable cost (e.g., time) largely depends on how many relevant concepts we know, are able to choose from and use. Such concepts can be of the nature of modeling patterns discussed previously, or coding practices. Given conditions, using such concepts renders the implementation more effective than others. That said, one needs to know in what conditions it is useful to use a concept and when different ones fit better. The ambition of this text is to describe a set of coding

practices. Thorough discussion about the suitability and alternatives in different conditions is out of scope, however.

We have identified a collection of coding practices worth considering when developing P²AMF class models. These are grouped into two groups, according to their purpose. The first group aims primarily at improving the cleanness of the OCL code, and thus the maintainability of the class model, especially when considering its lifetime in the long run. The second group aims primarily at achieving more evaluative power/effectiveness. The coding practices are as follows (in their purpose groups):

Purpose: Achieving cleanness of code, thus maintainability of the class model

- Code comments
- Indentation
- Naming conventions
- Subdivision of code into multiple operations
- Structuring expressions into paragraphs

Purpose: Achieving more evaluative power/effectiveness

- Defining local variables through *let* expressions
- Recursive calls in operations
- Translating mathematical formulas

Cleanness and maintainability

Challenge/need. Knowing the structure of your code and where to find what you look for, is vital for being able to maintain the code, not to mention the importance of enjoying one's work. You might have experienced revisiting a code that you yourself wrote years ago, perhaps just months ago; or code you "inherited" from your colleague or a friend. In companies, turnover is a reality - new people are being employed while other retire, change position, responsibilities, team role, or workplace as such. At the same time, software that is being used, as well as models that are being used, must be maintained continuously. This means that new people often become responsible for maintaining code that they themselves haven't written. Often, it is not little code. Organizations therefore need to keep themselves ready for such events through having their workforce maintain their code clean, understandable and well maintainable for themselves as well as their colleagues and new unknown people that are once likely going to take over responsibilities for the piece of code one is writing right now. Although there are a number of well-established and validated techniques and principles available to achieve these goals and keep them achieved over time (e.g., refactoring, pair programming, pair review etc.), this text only provides a handful of simple ones. The reason is that the authors do not currently perceive demanding practices (those that demand considerable time, resources and systematic devotion) nearly as justified in the practice of class modeling (meta-modeling) as they are in the practice of complex software development. Moreover, the selection of the practices described below is based on a limited body of experience, and locally experienced benefits and needs we met when modeling.

Code comments

Proposed practice and expected benefit. In OCL, comments start with two dashes ("--") and end with the end of the line. The content put between these delimiters does not have any effects on the interpreted OCL code. Dependent on the complexity of the OCL code one is dealing with, code comments might have the potential to achieve greater comprehensibility. Again, one should always consider one's eventual successors in maintaining or otherwise having to deal with the code. One should, however avoid writing comments just to have comments written. Each and every comment should serve a rather clear purpose. OCL can be a counter-intuitive language to read and think in, especially when code statements get long and complicated. Comments can aid the modeler trying to read the code through describing the semantics of it in natural language, or so provide any other information that may be helpful for understanding the code. However, comments should not state what is plainly obvious from reading the OCL statements, such as the following:

```
self.someOperation() -- call someOperation on self .
```

Writing comments that state the obvious is just introducing unnecessary content and therefore making it all more messy. One needs to learn to see in what contexts a comment is appropriate and helpful, and when it is safe or even better to omit. This can require practice and experience.

Example. If we take the following example, we can see that parts of the code are just not readily obvious:

```
if (self.lowerAndGate->size() = 0 and self.lowerOrGate->size() = 0) then
        let thisWorks : Boolean = bernoulli(0.5)
        if thisWorks = true then 1 else 0 endif
else
        if (self.lowerAndGate->size() > 0) then
                if (self.lowerAndGate.works > 0) then 1 else 0 endif
        else
                if (self.lowerOrGate.works > 0) then 1 else 0 endif
        endif
endif
```

It might be obvious to the modeler right after having designed a solution and being mentally fully tuned to the context of it. However, the same person might have a difficult time recalling the context just a few months later, not to mention other people. Such an unnecessary difficulty can be remediated by appropriate commenting:

```
if (self.lowerAndGate->size() = 0 and self.lowerOrGate->size() = 0) then
        -- it is a leaf node and should not be derived; either it works or it does not
        let thisWorks : Boolean = bernoulli(0.5)
        if thisWorks = true then 1 else 0 endif
else
        if (self.lowerAndGate->size() > 0) then
                -- there is an AND gate attached... is it satisfied?
                if (self.lowerAndGate.works > 0) then 1 else 0 endif
        else
                -- an OR gate is attached... is it satisfied?
                if (self.lowerOrGate.works > 0) then 1 else 0 endif
        endif
endif
```

It might be beneficial to also comment at the beginning of operations, derived attributes and derived connections – simply any elements that employ some OCL code. Such comments should briefly and conceptually

explain what the operation/attribute/connection does, but without going into implementation specifics. An example follows:

```
-- This attribute derivation evaluates the functional availability of a node.
[OCL code···]
```

Indentation

Proposed practice and expected benefit. Indentation refers to placing the text further to the right to separate it from the rest of the text. In the context of OCL coding and programming in general, there can be arbitrarily many levels of indentation the modeler/programer uses. Using them we strive to separate different blocks of code, in order to achieve greater comprehensibility.

Example. Imagine the following code, which is not indented:

```
if (self.lowerAndGate->size() = 0 and self.lowerOrGate->size() = 0) then
let thisWorks : Boolean = bernoulli(0.5)
if thisWorks = true then 1 else 0 endif
else if (self.lowerAndGate->size() > 0) then
if (self.lowerAndGate.works > 0) then 1 else 0 endif
else if (self.lowerOrGate.works > 0) then 1 else 0 endif
endif
endif
```

Although the code at least spans over several lines (is not written in one continuous line), the comprehensibility is rather limited compared to the following (indented code):

```
if (self.lowerAndGate->size() = 0 and self.lowerOrGate->size() = 0) then
        let thisWorks : Boolean = bernoulli(0.5)
        if thisWorks = true then 1 else 0 endif
else
        if (self.lowerAndGate->size() > 0) then
                if (self.lowerAndGate.works > 0) then 1 else 0 endif
        else
                if (self.lowerOrGate.works > 0) then 1 else 0 endif
        endif
endif
```

Indenting code reduces the unnecessary workload required for one to identify its structure, so as to comprehend the content piece by piece in a divide-and-conquer manner.

Naming conventions

Proposed practice and expected benefit. Naming conventions have to do with how we name classes and objects, attributes, operations, local variables (defined by *let*-expressions), etc. Names affect the comprehensibility of our code to a large extent. They can make the code read more like a prose, a normal human-readable text. Of course, here one needs to balance how much information to put in names, too. If the names are too short and uninformative, although equally machine-friendly, they make little sense to humans, which need to orient themselves well in the code. If the names are too long, on the other hand, they are make writing code cumbersome. The art here is to choose shortest possible names that mediate information needed for the developer to comprehend what the code does. Things should not be underdone, nor overdone, but finding the right balance given conditions is not always easy, unfortunately. As other practices, becoming tuned to choosing effective names might require some time and experience.

Example. Consider our example again, with almost random-like, uninformative names:

```
if (self.g1->size() = 0 and self.g2->size() = 0) then
        let b : Boolean = bernoulli(0.5)
        if b = true then 1 else 0 endif
else
        if (self.g1->size() > 0) then
                if (self.g1.v > 0) then 1 else 0 endif
        else
                if (self.g2.v > 0) then 1 else 0 endif
        endif
endif
```

Here comes the original one (with comments left out):

```
if (self.lowerAndGate->size() = 0 and self.lowerOrGate->size() = 0) then
        let thisWorks : Boolean = bernoulli(0.5)
        if thisWorks = true then 1 else 0 endif
else
        if (self.lowerAndGate->size() > 0) then
                if (self.lowerAndGate.works > 0) then 1 else 0 endif
        else
                if (self.lowerOrGate.works > 0) then 1 else 0 endif
        endif
endif
```

Subdivision of code into multiple operations

Proposed practice and expected benefit. When creating a model, perhaps a simple, non-ambitious one, one might tend to put a lot of code into an attribute derivation, or an operation. The problem is that such code quickly becomes messy. A rule of thumb is that one operation should do one thing (i.e., fulfill one coherent function) - or at least as little more as possible. Every programmer probably knows the term called "*God methods*" - methods (operations) that are several screens long. Often even a few dozens of lines qualifies an operation as too long. Such long operations are no fun to read, because they are usually difficult to comprehend as to the functional whole they represent. Moreover, such programming typically reflects the lack of reuse in code at a micro level. To remediate such problems, one can subdivide the large operation by defining and calling within several more operations according to their logical purposes, which are then called from the original operation. Such practice both increases comprehensibility and maintainability, since the methods can be called from different places in the model, and if they become updated, the update applies to the whole model, not just a single instance in which the code is used.

Example is omitted for brevity.

Structuring expressions into paragraphs

Proposed practice and expected benefit. Dependent on what one models, it might be necessary to have attribute derivations or operations that span across several lines. In case the code consists of logically separable blocks, where one part of the code does something specific, some other does something else, it is a good idea to separate these parts by an empty line. Such separation increases comprehensibility of the code, similarly as indentation (described further above) does. It helps the one who reads the code to separate logical parts, which eliminates the need to additionally identify code structure needed to do such separation in the reader's mind anyway.

Example. Let us consider the following code:

```
let param1 : Real = 1.8375 in
let param2 : Real = 2.4398 in
let param3 : Int = 7 in
let param4 : Boolean = false in

[the calculation code here as an OCL expression]
```

Putting the space between the block of let expressions and the calculation expression is an example. Similarly, one can separate pieces of code into blocks within the expression that contains the calculation code.

Evaluative power/effectiveness

Challenge/need. While the coding practices that target cleanness and maintainability of code, discussed above, are rather supportive and defensive in the sense that they remedy sneaky and conditional problems that start to hurt with some delay (from days to years), practices that target increased evaluative power or effectiveness often have a more "offensive" character. They primarily aim to solve problems that are readily apparent and perceivable, sometimes even before one starts to write code; although they can also be used to increase clearness and maintainability, just as the previous ones.

Usually, there is more than one single way to implement a solution to a specific problem. Such implementation alternatives, however have different dispositions in different regards, e.g., as to how much code and/or complexity is needed for a functional implementation. In some cases it might be so that different alternative implementations have different limits in achieving results. The limits can be computational demands (throughput performance), precision, accuracy, or even correctness as such, etc. It is advisable to consider several alternative solutions (if several can be identified), and reason on which ones are going to satisfy relevant needs the best, or at least well enough. To find what are the relevant needs, however, is itself non-trivial. In real-world situations, one seldom obtains a complete and consistent set of requirements, so that one can jump straight into engineering a solution. One often needs to first ascertain that the right questions are being asked and that the solutions are going to be measured and matched against the right criteria. Thus, the question is broad and open; universal and unconditional answers should not be expected. After one makes the set of criteria for a solution explicit and accepts them, one can reasonably analyze, compare and discriminate between different implementation approaches. The coding practices mentioned below attempt to provide a few tips, which you can consider using when facing a specific modeling task or problem.

Defining local variables through let expressions

Proposed practice and expected benefit. In OCL, *let* expressions allow defining locally scoped variables within the code. Although there are many similar specifics in OCL and this text does not aim to introduce them all, we consider *let* a particularly helpful tool to remind about, or introduce to (if the reader is choosing a more hands-on approach than studying OCL first).

Among other, locally scoped variables enable us to declare constants in a separate block of code than that in which they are used. If the implemen-

tation of an operation or derivation of an attribute requires multiple constants or other values to be accessed by its name instead of a complicated sub-statement, it is reasonable to define them somewhere above the core calculation in the operation. If we only keep them spread across the code without assigning them a name, it might become non-apparent what they actually represent in the context in which they are used, as well as the overall structure and comprehensibility might suffer. Locally scoped variables declared through *let* expressions hence aid simplicity in programming as well as cleanness and maintainability of code.

Example. The structure of an expression statement is the following:

```
let [variable name] : [variable type] = [expression] in [expression]
```

The first expression provides a value for the local variable, and the other expression, the rest of the code, is the one, in and under which the local variable is defined. The "**in**" keyword seems to be optional, although it is advisable to use for more clarity. A *let* expression can be used as follows:

```
[some eventual code]
let localVariable : Real = self.neighbor.anotherNeighbor->someOperation() in
[rest of the code (in OCL always as a single expression)]
```

or equivalently, just without "**in**":

```
[some eventual code]
let localVariable : Real = self.neighbor.anotherNeighbor->someOperation()
[rest of the code (in OCL always as a single expression)]
```

For multiple such local variables, *let*-expressions can be stacked up as follows:

```
[some eventual code]

let localVariable : Real = self.neighbor.anotherNeighbor->someOperation() in
let localVariable2 : Real = self.neighbor->someOtherOperation() in
let localVariable3 : Real = localVariable->.yetAnotherNeighbor->someOperation() in

[rest of the code (in OCL always as a single expression)]
```

Let expressions can define any variables. They can be fetched from somewhere in the object model that is being evaluated.

Recursive calls in operations

Proposed practice and expected benefit. In procedural programming languages (also assuming Turing completeness[3]), looping algorithmic problems can be solved iteratively or recursively; while for each iterative solution has a recursive exists, and vice versa. Using declarative programming languages OCL, this might be the case for some problems, but for others, one might only be left to find a recursive solution to. In an iterative approach, there is a loop, which iterates through elements of a collection until a specific condition is reached. One can also use multiple loops under each other. In OCL, operations such as iterate or forAll enable for iterative solutions. In a recursive approach, one defines an operation so that it performs partial steps that lead to solving the problem and calls itself further (i.e., recursively calls itself), until a termination condition occurs. If a termination condition occurs, the operation just returns a value and does not recur further. Recursion thus creates a tree of calls of the same operation, each time with different input parameters. Given a limited amount of operating memory and stack size, recursive approaches to a same problem might pose tighter limits than iterative ones.

Sometimes however, recursion might be considerably more straight-forward and elegant to read in code, and thus more than welcome by a programmer who deals with some advanced computation. A potential challenge with recursion arises, however, when it comes to making sure that the terminating conditions are appropriately defined and implemented so that the recursion finishes after a finite amount steps given any possible combination of valid inputs. Also, correct implementation and eventual debugging of an advanced algorithmic design such as recursion is none of the simplest tasks, even more difficult given the limited feature set and rather low level of maturity of the tools that enable OCL development to date. Therefore, extra care is needed when using recursion. Not only it can be difficult to arrive at a correct implementation, but also make such an implementation operate efficiently. The benefit of using recursion may be decreased amount of code to write to implement some calculation. At other times, a recursive solution to a problem might be considerably more intuitive to comprehend than an iterative one. Yet at other times, it might not be practical to implement an iterative solution, which could justify for a recursive one in case it was more practical.

Example. For an example, let us revisit the PDU encapsulation model mentioned in relation to self-associations of a class. You might wish to refer back to the example to recall the context. Let us say that we want to calculate the number of PDUs that carry some specific PDU in a specific hierarchy. It is even difficult to think of an iterative solution, but let us look at a recursive solution to the problem. A slightly modified class model from figure 12.9 is depicted in figure 12.18. To the definition of the *PDU* class, two things were added - an integer attribute, called *numberOfCarriers*, and an operation called *getCarriers*. The operation takes in two parameters - a PDU object to begin the search from and a set of PDUs to exclude from all subsequent rounds in the search (as will read from the code given further below). The operation returns a set of PDUs - the actual carriers found. It not only returns direct carriers, but searches the hierarchy transitively.

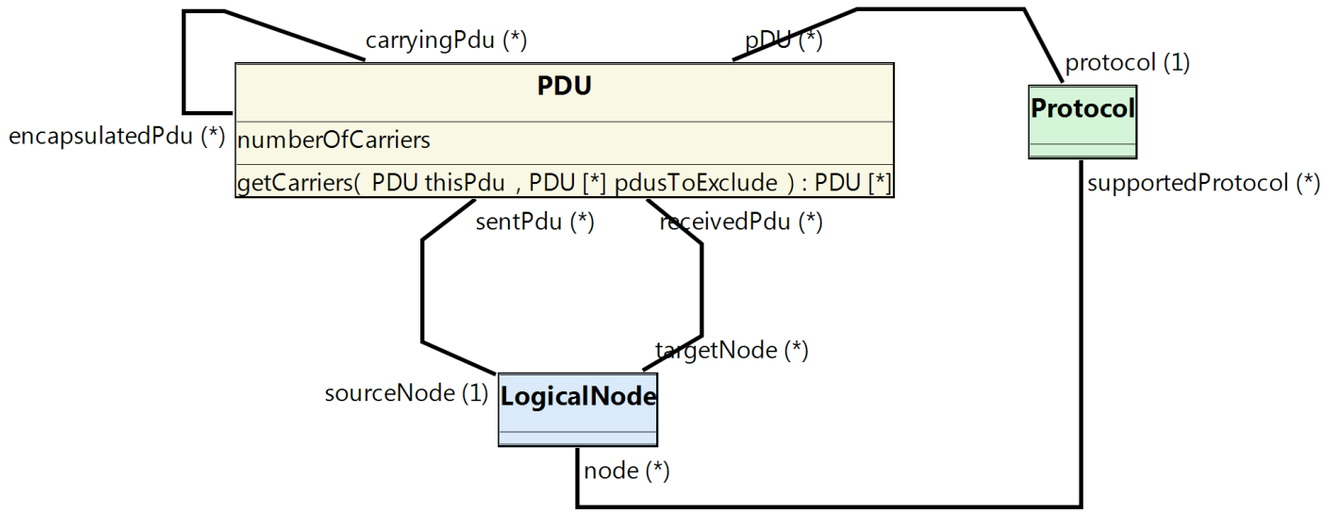


Figure 12.18. *Modification of the class model from the example given in the description of the self-association pattern (see the attribute numberOfCarriers and the operation getCarriers in the PDU class).*

The integer attribute *numberOfCarriers* is derived as follows:

```
self.getCarriers(self, Set{})->size()
```

The operation, *getCarriers*(*thisPdu* : PDU, *pdusToExclude* : Set(PDU)) : Set(PDU) is defined as follows:

```
let pdusToConsider : Set(PDU) =
        thisPdu.carryingPdu->asSet()->excluding(pdusToExclude) in

if ( pdusToConsider->size() = 0 ) then
        -- no carriers for this PDU
        pdusToConsider  -- this is an empty set
else
        -- return the carriers considered in this round and
        -- those that carry them (in a transitive fashion)
        let newPdusToExclude : Set(PDU) = pdusToExclude->union(pdusToConsider) in
        pdusToConsider->union(
                pdusToConsider->collect( p : PDU |
                        self.getCarriers(p, newPdusToExclude)  -- the recursive call
                )
        )
endif
```

A corresponding object model, which was only modified by the change in the class model and adding the numeric values besides the object boxes, looks as depicted in figure 12.19.

Translating mathematical formulas

Proposed practice and expected benefit. OCL has no built-in facilities for advanced mathematical calculations to date. In case such calculations are needed (e.g., exponential functions), the modeler needs to translate the mathematical formulas so that OCL is able to calculate the result. Dependent on how advanced the calculations are, it might be a matter of simply transforming the formula, or one might need to implement the operations in a satisfactory way through the facilities OCL readily provides.

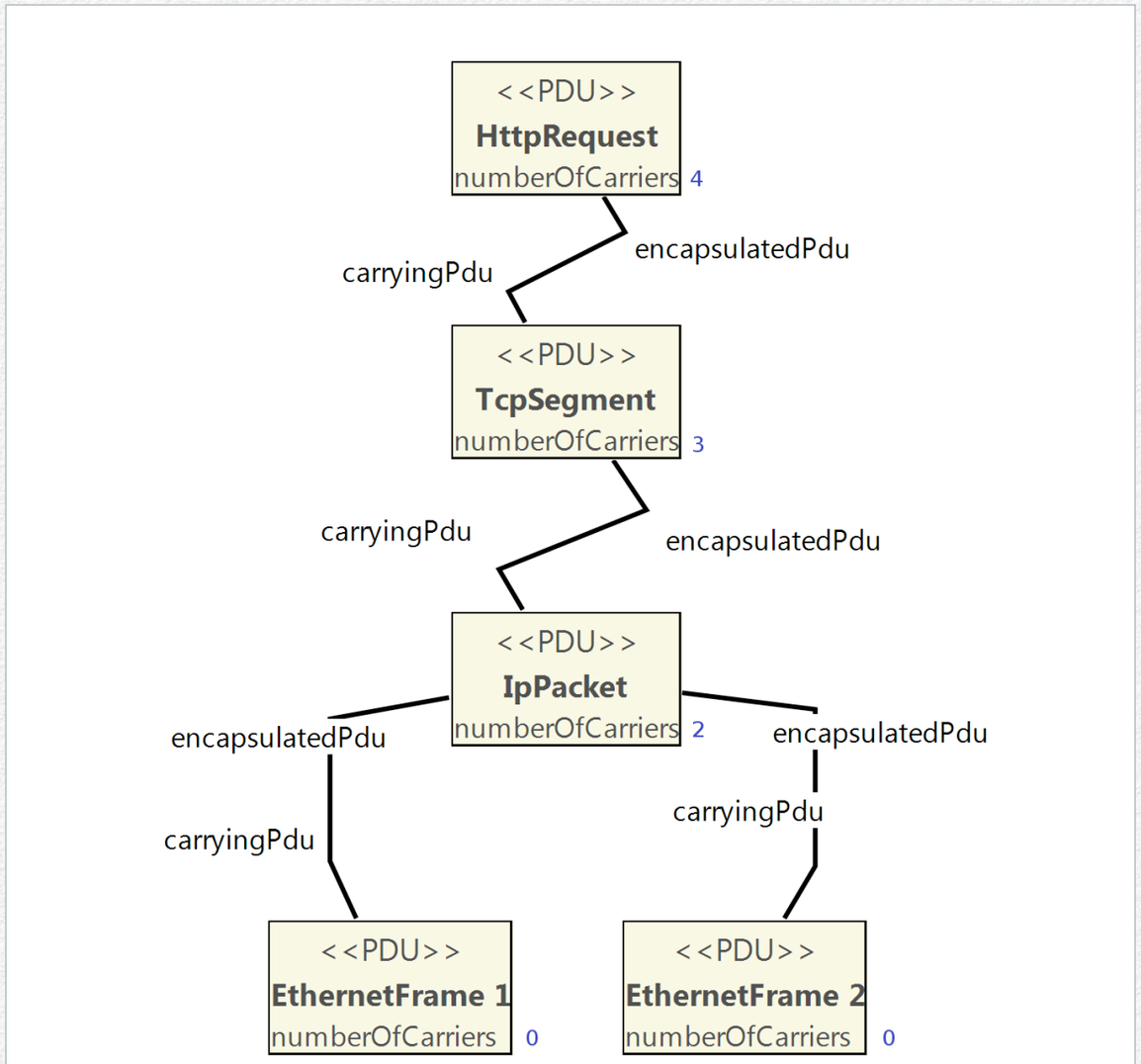


Figure 12.19. *An object model corresponding to the class model depicted in figure 18.*

Example. For example, formula for a power of three of a real value does not exist in the OCL standard, but can be simply translated as follows:

```
let value : Real = [expression for the value to be powered] in
value * value * value
```

Clearly, that example would not work for all formulas, simply because of its lacking generality. A more general power function can be implemented as a recursive operation. Let us consider operation *power*(*base* : Real, *exponent* : Int) : Real on a class defined as follows:

```
self._calculatePower(base, base, exponent)
```

And also the operation *_calculatePower*(*value* : Real, *base* : Real, *exponent* : Int) : Real, which we call from the *power* operation:

```
if ( exponent = 0 ) then
        -- a number powered by zero equals one
        1
else
        if ( exponent = 1 ) then
                -- a number powered by one equals the number itself
                value
        else
                -- otherwise we multiply the value by base and call the
                -- operation recursively with a decremented exponent
                _calculatePower( value * base, base, exponent - 1 )
        endif
endif
```

The calculation can then be called on the class through *power*([some base], [some exponent]) in an OCL expression. In the above provided example, however, the power operation only works with non-negative integer exponents. It is also possible to develop more general operations for arbitrary mathematical functions, which is out of scope for this text, though.

This is no longer a part of the above example. Here it is important to mention that although the OCL standard does not require the implementation of a power function, the EAAT tool now implements this. Hence, instead of needing to write the above code (which now only serves an illustration purpose), one can, and should, simply write the following to achieve the same functionality (equally applicable to integer as it is to real values):

```
let value : Real = [some value] in value->power( [some exponent] )
```

Another such mathematical functionality that might often come handy is now supported by EAAT – linear interpolation:

```
let x : Set(Real) = Set { [some comma separated values] } in  -- a set of x-values
let y : Set(Real) = Set { [some comma separated values] } in  -- a set of y-values (equally
many)

        x->linear( y, [some y-value] ) -- lin. interp. of an x-value that corresponds to the y-value given
```

Concluding remarks

This text has described several modeling patterns (cf. our definition of a modeling pattern) and several OCL coding practices. The authors hope that you as the reader will find the information found here useful or inspiring. All in all, the patterns and practices described are no magical or definitive solutions to concrete real problems, and a lot of mental work is required for a modeler to apply them to solve a real problem. On the one hand, the descriptions provided in the text are rather simple and far from being comprehensive. On the other hand, their prime purpose is to serve as a source of advice and inspiration for you as a modeler, who

might face more difficult and complex modeling problems, decompositions of which might allow you to constructively apply even as simple and general patterns as those, in order to solve your modeling challenges in ways that are satisfactory to you.

References

References

1. R. Schiesser, *IT systems management*. Prentice Hall, 2010.

2. P. Ceruzzi, *A history of modern computing*. The MIT press, 2003.

3. Bureau of Economic Analysis, “National Economic Accounts, Table: 5.5.6.Real Private Fixed Investment in Equipment and Software by Type," Feb. 2007. [Online]. Available: www.bea.gov

4. P. Gottschalk, “Strategic management of IS/IT functions: the role of the CIO in Norwegian organisations,” International Journal of Information Management, vol. 19, no. 5, pp. 389-399, 1999.

5. J. Zachman, “A framework for information systems architecture” IBM systems journal, vol. 26, no. 3, pp. 276-292, 1987.

6. TAFIM, “Technical Architectural Framework for Information Management, Version 2.0,” Department of Defense, Technical standard, 1994.

7. C4ISR Architecture Working Group, “C4ISR architecture framework version 2.0” US Department of Defense, Technical standard, 1997.

8. Department of Defense Architecture Framework Working Group, “DoD Architecture Framework, version 2.0” US Department of Defense, Technical standard, 2007.

9. NAF, ”NATO C3 Technical Architecture, Volume 1-5, Version 7.0” NATO, Technical standard, 2005.

10. MoDAF, “Ministry of Defense Architecture Framework Overview, version 1.0” Ministry of Defense, Technical standard, 2005.

11. The Open Group,, *The Open Group Architecture Framework (TOGAF) - version 9*. The Open Group, 2009.

12. H. Mintzberg, *The structure of organizations: A Synthesis of the Research*. Prentice-Hall, 1979.

13. M. Lankhorst, *Enterprise architecture at work: Modelling, communication and analysis*. Springer-Verlag New York Inc, 2009.

14. P. Närman, P. Johnson, and L. Nordström, “Enterprise architecture: A framework supporting system quality analysis” in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, 2007, pp. 130-130.

15. P. Närman, M. Schönherr, P. Johnson, M. Ekstedt, and M. Chenine, “Using enterprise architecture models for system quality analysis” in *Enterprise Distributed Object Computing Conference, 2008. EDOC’08. 12th International IEEE*. IEEE, 2008, pp. 14-23.

16. P. Närman, M. Buschle, and M. Ekstedt, “An enterprise architecture framework for multi-attribute information systems analysis” *Systems and Software Modeling*., 2012, accepted to be published 2012.

17. P. Johnson, E. Johansson, T. Sommestad, and J. Ullberg, “A tool for enterprise architecture analysis” in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007*. 11th IEEE International. IEEE, 2007, pp. 142-142.

References

18. M. Ekstedt, U. Franke, P. Johnson, R. Lagerström, T. Sommestad, J. Ullberg, and M. Buschle, "A tool for enterprise architecture analysis of maintainability" in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 327-328.

19. M. Buschle, J. Ullberg, U. Franke, R. Lagerström, and T. Sommestad, "A tool for enterprise architecture analysis using the prm formalism" *Information Systems Evolution*, pp. 108-121, 2011.

20. M. Buschle, P. Johnson et al., "A Tool For Enterprise Architecture Analysis," 2012, submitted.

21. P. Johnson, J. Ullberg, M. Buschle, K. Shahzad, and U. Franke, "P2AMF: Predictive, Probabilistic Architecture Modeling Framework" 2012, submitted.

22. J. Ullberg, U. Franke, M. Buschle, and P. Johnson, "A tool for interoperability analysis of enterprise architecture models using Pi-OCL" *Enterprise Interoperability IV*, pp. 81-90, 2010.

23. J. Ullberg, P. Johnson, and M. Buschle, "A modeling language for interoperability assessments" *Enterprise Interoperability*, pp. 61-74, 2011.

24. J. Ullberg and P. Johnson, "Predicting interoperability in an environmental assurance system" *Enterprise Interoperability V*, pp. 25-35, 2012.

25. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison Wesley Longman / Software Engineering Institute, 1998.

26. W. Harrison and C. Cook, "Insights on improving the maintenance process through software measurement" in *Proc. of the IEEE Software Maintenance Conf.*, Nov. 1990.

27. T. Pigoski, *Practical Software Maintenance*. John Wiley & Sons, 1997.

28. S. Jarzabek, *Effective Software Maintenance and Evolution: A Reuse-Based Approach*. Auerbach Publications, Taylor & Francis Group, 2007.

29. L. Laird and C. Brennan, *Software Measurement and Estimation: A Practical Approach*. IEEE Computer Society / John Wiley & Sons, 2006.

30. R. Lagerström, "Analyzing system maintainability using enterprise architecture models" *Journal of Enterprise Architecture*, vol. 3, no. 4, pp. 33-42, 2007.

31. R. Lagerström and P. Johnson, "Using architectural models to predict the maintainability of enterprise systems" in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 248-252.

32. R. Lagerström, P. Johnson, D. Höök, and J. König, "Software change project cost estimation-a bayesian network and a method for expert elicitation" in *Third International Workshop on Software Quality and Maintainability (SQM 2009)*, 2009.

33. R. Lagerström, P. Johnson, and M. Ekstedt, "Architecture analysis of enterprise systems modifiability - a metamodel for software change cost estimation" *Software Quality Journal*, vol. 18, pp. 437-468, 2010.

References

34. R. Lagerström, P. Johnson, and D. Höök, “Architecture analysis of enterprise systems modifiability-models, analysis, and validation” *Journal of Systems and Software*, vol. 83, no. 8, pp. 1387-1403, 2010.

35. R. Lagerström, “Enterprise systems modifiability analysis - an enterprise architecture modeling approach for decision making” Ph.D. dissertation, KTH - the Royal Institute of Technology, Apr. 2010.

36. B. Boehm, R. Madachy, B. Steece *et al., Software Cost Estimation with COCOMO II*. Prentice-Hall, 2000.

37. P. Oman, J. Hagemeister, and D. Ash, “A definition and taxonomy for software maintainability” Software Engineering Lab, Tech. Rep., 1992.

38. IEEE Standards Board, “IEEE standard glossary of software engineering technology” Published as IEEE Std 610-12-1990, The Institute of Electrical and Electronics Engineers, Tech. Rep., Sep. 1990.

39. M. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

40. S. Henry and D. Kafura, “Software structure metrics based on information flow” *Software Engineering, IEEE Transactions on*, vol. SE-7, no. 5, pp. 510-518, Sep. 1981.

41. M. Frappier, S. Matwin, and A. Mili, “Software metrics for predicting maintainability” *Software Metrics Study: Tech. Memo*, vol. 2, 1994.

42. T. McCabe, “A complexity measure” *IEEE Transactions on Software Engineering*, no. 4, pp. 308-320, 1976.

43. R. Park, “Software size measurement: A framework for counting source statements” DTIC Document, Tech. Rep., 1992.

44. B. Curtis, H. Krasner, and N. Iscoe, “A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268-1287, 1988.

45. V. Maraia, *The Build Master: Microsoft’s Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.

46. C. Jones, *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., 1991.

47. N. Fenton and A. Melton, “Deriving structurally based software measures” *Journal of Systems and Software*, vol. 12, no. 3, pp. 177-187, 1990.

48. D. Strong, Y. Lee, and R. Wang, “Data quality in context” *Communications of the ACM*, vol. 40, no. 5, pp. 103-110, 1997.

49. T. Redman, *Data quality for the information age*. Artech House, 1996.

50. C. Batini and M. Scannapieca, *Data quality: Concepts, methodologies and techniques*. Springer-Verlag New York Inc, 2006.

51. P. Närman, H. Holm, P. Johnson, J. König, M. Chenine, and M. Ekstedt ”Data accuracy assessment using enterprise architecture” *Enterprise Information Systems*, vol. 5, no. 1, pp. 37-58, 2011.

52. P. Närman, P. Johnson, M. Ekstedt, M. Chenine, and J. König, “Enterprise architecture analysis for data accuracy assessments” in

Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International, Sep. 2009, pp. 24-33.

53. R. Wang, M. Ziad, and Y. Lee, *Data quality*. Kluwer Academic Pub, 2001.

54. Y. W. Lee, L. L. Pipino, J. D. Funk, and R. Y. Wang, *Journey to data quality.* Cambridge, Mass.: MIT Press, 2006.

55. R. Y. Wang, *Information quality [Elektronisk resurs]*. Armonk, N. Y.: M. E. Sharpe, 2005.

56. D. P. Ballou and H. L. Pazer, "Modeling data and process quality in multiinput, multi-output information systems" *Management Science*, vol. 31, no. 2, pp. 150-162, 1985.

57. B. E. Cushing, "A mathematical approach to the analysis and design of internal control systems" *The Accounting Review*, vol. 49, no. 1, pp. 24-41, 1974.

58. E. Brynjolfsson, "The productivity paradox of information technology" *Communications of the ACM*, vol. 36, no. 12, p. 77, 1993.

59. J. Ross, P. Weill, and D. Robertson, *Enterprise architecture as strategy: Creating a foundation for business execution*. Harvard Business Press, 2006.

60. G. Riempp and S. Gieffers-Ankel, "Application portfolio management: a decision-oriented view of enterprise architecture" *Information Systems and E-Business Management*, vol. 5, no. 4, pp. 359-378, 2007.

61. D. Simon, K. Fischbach, and D. Schoder, "Application Portfolio Management - An Integrated Framework and a Software Tool Evaluation Approach" *Communications of the Association for Information Systems*, vol. 26, no. 1, p. 3, 2010.

62. W. DeLone and E. McLean, "Information systems success: the quest for the dependent variable" *Information systems research*, vol. 3, no. 1, pp. 60-95, 1992.

63. ….., "The DeLone and McLean model of information systems success: A ten-year update" *Journal of management information systems*, vol. 19, no. 4, pp. 9-30, 2003.

64. S. Devaraj and R. Kohli, "Performance impacts of information technology: Is actual usage the missing link?" *Management Science*, vol. 49, no. 3, pp. pp. 273-289, 2003. [Online]. Available: http://www.jstor.org/stable/4133926

65. P. Weill and M. Vitale, "Assessing the health of an information systems applications portfolio: An example from process manufacturing" *MIS quarterly*, vol. 23, no. 4, pp. 601-624, 1999.

66. P. Närman, H. Holm, D. Höök, N. Honeth, and P. Johnson, "Using enterprise architecture and technology adoption models to predict application usage" *Journal of Systems and Software*, 2012, online first.

67. F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology" *MIS Quarterly*, vol. 13, no. 3, pp. 319-340, 1989.

References

68. P. J. Hu, P. Y. K. Chau, O. R. L. Sheng, and K. Y. Tam, "Examining the technology acceptance model using physician acceptance of telemedicine technology" *J. Manage. Inf. Syst.*, vol. 16, pp. 91-112, September 1999.

69. D. Gefen and D. Straub, "Gender differences in the perception and use of e-mail: An extension to the technology acceptance model" *Mis Quarterly*, vol. 21, no. 4, pp. 389-400, 1997.

70. P. Y. K. Chau, "An empirical assessment of a modified technology acceptance model" *J. Manage. Inf. Syst.*, vol. 13, pp. 185-204, September 1996.

71. P. Pavlou, "Consumer acceptance of electronic commerce: Integrating trust and risk with the technology acceptance model" *International Journal of Electronic Commerce*, vol. 7, no. 3, pp. 101-134, 2003.

72. K. Mathieson, "Predicting user intentions: comparing the technology acceptance model with the theory of planned behavior" *Information systems research*, vol. 2, no. 3, pp. 173-191, 1991.

73. V. Venkatesh and F. Davis, "A theoretical extension of the technology acceptance model: Four longitudinal field studies" *Management science*, vol. 46, no. 2, pp. 186-204, 2000.

74. V. Venkatesh, "Determinants of perceived ease of use: Integrating control, intrinsic motivation, and emotion into the technology acceptance model" *Information systems research*, vol. 11, no. 4, pp. 342-365, 2000.

75. E. Karahanna, D. W. Straub, and N. L. Chervany, "Information technology adoption across time: A cross-sectional comparison of pre-adoption and post-adoption beliefs" *MIS Quarterly*, vol. 23, no. 2, pp. pp. 183-213, 1999.

76. D. Goodhue and R. Thompson, "Task-technology fit and individual performance" *Mis Quarterly*, vol. 19, no. 2, pp. 213-236, 1995.

77. I. Zigurs and B. Buckland, "A theory of task/technology fit and group support systems effectiveness" *MIS quarterly*, vol. 22, no. 3, pp. 313-334, 1998.

78. D. Goodhue, "Development and measurement validity of a task/technology fit instrument for user evaluations of information system" *Decision Sciences*, vol. 29, no. 1, pp. 105-138, 1998.

79. M. Dishaw and D. Strong, "Supporting software maintenance with software engineering tools: A computed task-technology fit analysis" *Journal of Systems and Software*, vol. 44, no. 2, pp. 107-120, 1998.

80. C. Lee, H. Cheng, and H. Cheng, "An empirical study of mobile commerce in insurance industry: Task-technology fit and individual differences" *Decision Support Systems*, vol. 43, no. 1, pp. 95-110, 2007.

81. J. Gebauer, M. Shaw, and M. Gribbins, "Task-technology fit for mobile information systems" Journal of Information Technology, 2010.

82. T. Ferratt and G. Vlahos, "An investigation of task-technology fit for managers in Greece and the US" *European Journal of Information Systems*, vol. 7, no. 2, pp. 123-136, 1998.

References

83. A. Majchrzak, A. Malhotra, and R. John, "Perceived Individual Collaboration Know-How Development Through Information Technology-Enabled Contextualization: Evidence from Distributed Teams" *Information systems research*, vol. 16, no. 1, pp. 9-27, 2005.

84. P. Legris, J. Ingham, and P. Collerette, "Why do people use information technology? A critical review of the technology acceptance model" *Information & management*, vol. 40, no. 3, pp. 191-204, 2003.

85. N. Venkatraman, "The concept of fit in strategy research: Toward verbal and statistical correspondence" *Academy of management review*, vol. 14, no. 3, pp. 423-444, 1989.

86. I. Vessey, "Expertise in debugging computer programs: An analysis of the content of verbal protocols" *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 16, no. 5, pp. 621-637, Sep. 1986.

87. J. Henderson and J. Cooprider, "Dimensions of I/S Planning and Design Aids: A Functional Model of CASE Technology" *Information Systems Research*, vol. 1, no. 3, p. 227, 1990.

88. D. Scott, "How to Assess Your IT Service Availability Levels" Apr. 2009.

89. U. Franke, M. Ekstedt, R. Lagerström, J. Saat, and R. Winter, "Trends in enterprise architecture practice - a survey" in *Trends in Enterprise Architecture Research*, ser. Lecture Notes in Business Information Processing, vol. 70. Springer Berlin Heidelberg, 2010, pp. 16-29.

90. IBM Global Services, "Improving systems availability" IBM Global Services, Tech. Rep., 1998.

91. A. Bharadwaj, M. Keil, and M. Mähring, "Effects of information technology failures on the market value of firms" *The Journal of Strategic Information Systems*, vol. 18, no. 2, pp. 66-79, 2009.

92. P. Närman, U. Franke, J. König, M. Buschle, and M. Ekstedt, "Enterprise architecture availability analysis using fault trees and stakeholder interviews" 2012, online first.

93. O. Holschke, P. Närman, W. Flores, E. Eriksson, and M. Schönherr, "Using enterprise architecture models and bayesian belief networks for failure impact analysis" in *Service-Oriented Computing-ICSOC 2008 Workshops*. Springer, 2009, pp. 339-350.

94. J. Raderius, P. Närman, and M. Ekstedt, "Assessing system availability using an enterprise architecture analysis approach" in *Service-Oriented Computing-ICSOC 2008 Workshops*. Springer, 2009, pp. 351-362.

95. A. Høland and M. Rausand, *System reliability theory: models and statistical methods*. Wiley New York, 1994.

96. D. Stamatis, *Failure mode and effect analysis: FMEA from theory to execution*. Asq Pr, 2003.

97. S. Meyn, R. Tweedie, and J. Hibey, *Markov chains and stochastic stability*. Springer London et al., 1993.

References

98. S. Mannan and F. Lees, *Lee's loss prevention in the process industries: hazard identification*, assessment, and control. Elsevier, 2005.

99. J. Andrews and C. Ericson, "Fault tree and Markov analysis applied to various design complexities" *Proceedings of the 18th International System Saftey* Conference, 2000.

100. M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J. Minarick, and J. Railsback, "Fault tree handbook with aerospace applications" 2002, http://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf.

101. B. Johnson, *Design and analysis of fault-tolerant digital systems*. Addison-Wesley Reading, MA, 1989.

102. V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of uml based software models," in *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*. New York, NY, USA: ACM, 2002, pp. 302-309.

103. J. Dugan, S. Bavuso, and M. Boyd, "Dynamic fault-tree models for faulttolerant computer systems" *Reliability, IEEE Transactions on*, vol. 41, no. 3, pp. 363-377, 2002.

104. International Organization for Standardization - ISO, "Iso-iec 9126-2:2003 software engineering - product quality - part 2: External metrics" JTC 1/SC 7 - Software and systems engineering, Tech. Rep., 2003.

105. J. Ullberg, D. Chen, and P. Johnson, "Barriers to enterprise interoperability" in *2nd IFIP WG5.8 Workshop on Enterprise Interoperability (IWEI 2009)*, Oct. 2009.

106. J. Ullberg, R. Lagerström, and P. Johnson, "Enterprise architecture: A service interoperability analysis framework" *Enterprise Interoperability III*, pp. 611-623, 2008.

107. E. Commission et al., "European interoperability framework for paneuropean egovernment services" *IDA working document, version*, vol. 2, 2004.

108. NEHTA, "Interoperability framework, version 2.0" National E-Health Transition Authority, Tech. Rep., 2007.

109. A. Berre, B. Elvesæter, N. Figay, C. Guglielmina, S. Johnsen, D. Karlsen, T. Knothe, and S. Lippe, "The athena interoperability framework" *Enterprise Interoperability II*, pp. 569-580, 2007.

110. D. Chen and N. Daclin, "Framework for enterprise interoperability" *Interoperability for Enterprise Software and Applications*, pp. 77-88, 2006.

111. T. Ruokolainen, Y. Naudet, and T. Latour, "An ontology of interoperability in inter-enterprise communities" *Enterprise Interoperability II*, pp. 159-170, 2007.

112. M. Kasunic and W. Anderson, "Measuring systems interoperability: Challenges and opportunities" 2004.

References

113. E. Morris, L. Levine, C. Meyers, D. Plakosh et al., "System of systems interoperability (sosi): Final report" DTIC Document, Tech. Rep., 2004.

114. A. Tolk and J. Muguira, "The levels of conceptual interoperability model" *System*, no. September, pp. 14-19, 2003.

115. T. Ford, J. Colombi, S. Graham, and D. Jacques, "The interoperability score" in *Proceedings of the Fifth Annual Conference on Systems Engineering Research*, 2007.

116. P. Närman, T. Sommestad, S. Sandgren, and M. Ekstedt, "A framework for assessing the cost of IT investments" in *Portland International Conference on Management of Engineering and Technology (PICMET)*, Aug. 2009.

117. B. Boehm, *Software engineering economics*. Prentice-Hall, 1981.

118. V. Basili and B. Boehm, "COTS-based systems top 10 list" *Computer*, vol. 34, no. 5, pp. 91-95, 2001.

119. C. Abts, B. Boehm, and E. Clark, "COCOTS: A COTS software integration lifecycle cost model - model overview and preliminary data collection findings" in *ESCOM-SCOPE Conference*. USC Center for Software Engineering, 2000.

120. B. Boehm and C. Abts, "COTS integration: Plug and Pray?" *Computer*, vol. 32, no. 1, pp. 135-138, 1999.

121. Z. Irani, J. Ezingeard, and R. Grieve, "Costing the true costs of IT/IS investments in manufacturing: a focus during management decision making" *Logistics Information Management*, vol. 11, no. 1, pp. 38-43, 1998.

122. P. Love, Z. Irani, A. Ghoneim, and M. Themistocleous, "An exploratory study of indirect ICT costs using the structured case method" *International Journal of Information Management*, vol. 26, no. 2, pp. 167-177, 2006.

123. Z. Irani, A. Ghoneim, and P. Love, "Evaluating cost taxonomies for information systems management" *European Journal of Operational Research*, vol. 173, no. 3, pp. 1103-1122, 2006.

124. P. Farquhar, "Utility assessment methods" Management science, pp. 1283-1300, 1984.

125. W. Edwards and F. Barron, "Smarts and smarter: Improved simple methods for multiattribute utility measurement" *Organizational Behavior and Human Decision Processes*, vol. 60, no. 3, pp. 306-325, 1994.

126. J. Bentham, An introduction to the principles of morals and legislation. Clarendon Press, 1879.

127. W. C. Mitchell, "Benthamís felicific calculus," *Political Science Quarterly*, vol. 33, no. 2, pp. 161ñ183, 1918.

128. R. L. Keeney, Decisions with multiple objectives: preferences and value trade-offs. Cambridge University Press, 1993.

129. P. C. Fishburn, "Utility theory for decision making," Publications in operations research/Operations Research Society of America (ISSN 0079-7723, no. 18, 1970.

References

130. F. J. Anscombe and R. J. Aumann, "A definition of subjective probability," *The annals of mathematical statistics*, vol. 34, no. 1, pp. 199ñ205, 1963.

131. R. L. Keeney, "Multiplicative utility functions," *Operations Research*, vol. 22, no. 1, pp. 22ñ34, 1974.

132. J. S. Dyer, "Maut-Multiattribute Utility Theory," Multiple Criteria Decision Analysis: State of the Art Surveys, vol. 78, p. 265, 2005.

133. D. Von Winterfeldt and G. W. Fischer, Multi-attribute utility theory: Models and assessment procedures. *Springer*, 1975.

134. KTH ICS, "Enterprise Architecture Analysis Tool," 2013. [Online]. Available: http://www.ics.kth.se/eaat. [Accessed: 31-Jan-2013].

135. OMG, "Object Management Group," 2013. [Online]. Available: http://www.omg.org/. [Accessed: 31-Jan-2013].

136. OMG, “Unified Modeling Language”, 2013. [Online]. Available http://www.omg.org/spec/UML/. [Accessed: 10-June-2013].

137. OMG, “Object Constraint Language”, 2013. [Online]. Available http://www.omg.org/spec/OCL/. [Accessed: 10-June-2013].

138. J. Cabot, “Object Constraint Language: A definitive guide”, 2012. [Online]. Available http://www.slideshare.net/jcabot/ocl-tutorial. [Accessed: 10-June-2013].

139. KTH, “Manuals”, 2013. [Online]. Available http://www.kth.se/ees/omskolan/organisation/avdelningar/ics/research/sa/p/eaat/manuals-1.387301. [Accessed: 10-June-2013].