Complexity Measures for System Architecture Models

by

Matti J Kinnunen

Submitted to the System Design and Management Program
in partial fulfillment of the requirements for the degree of

Master of Science in Engineering and Management

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2006

Author .....
System Design and Management Program
February, 2006

Certified by .....
Edward Crawley
Professor, Engineering Systems Division
Thesis Supervisor

Accepted by .....
Pat Hale
System Design and Management Program
Director

Complexity Measures for System Architecture Models

by

Matti J Kinnunen

Submitted to the System Design and Management Program
on February, 2006, in partial fulfillment of the
requirements for the degree of
Master of Science in Engineering and Management

Abstract

This thesis lays the necessary groundwork for measuring the complexity of systems architecture models. We propose a set of complexity measures, which are usable with models defined using the Object-Process Model (OPM). In order to do this, we introduce a new concept of *interface complexity multiplier* for compensating the hidden information at interfaces. We also define a set of complexity metrics for system architecture models.

We also develop models for three different systems for mobile entertainment. The purpose of these models is to show how OPM is suitable for modeling such systems and also to provide some comparative material for complexity measurements. We use the new metrics to determine the complexity of the models of mobile entertainment systems.

The thesis also contains a rigorous definition of complexity and a survey of existing complexity measurement methods.

Thesis Supervisor: Edward Crawley
Title: Professor, Engineering Systems Division

Acknowledgments

I would like to thank the following persons. Professor Edward Crawley for supervising this thesis and lecturing the course on systems architecting. Mr. Bill Simmons and Mr. Wilfried Hofstetter for completing the systems architecting course assignments with me. I have used some insights and pictures from the course assignments in this thesis. Dr. Ben Koo for helping me in putting together my thesis proposal and for discussions on systems architecting. Mr. Kannakumar Kittusamy for studying with me the whole year 2005 and for deep discussions on systems architecting. Finally, Ms. Sirkku Ikonen for philosophical advice on complexity, meaning and other concepts, and for her loving support over this and many previous years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The KISS-heuristic, which variably stands for "Keep it simple and short" or "Keep it simple, stupid" is arguably one of the most widely accepted heuristics in engineering. The basic idea is that the simpler the system is, the easier it is to design, implement and maintain it is. Or, in terms of money, the simpler the system is, the less it costs to design, build and maintain the system.

The main problem with the KISS-heuristic, as almost all heuristics in general, is that it relies on the intuition of the engineer. A comptetent and experienced engineer may quite well intuitively judge, which of two systems is simpler. But relying on the intuition of experienced engineers is problematic. For the first, several engineers may well rank two systems in different order of simplicity. For the second, in some cases, as with automatic generation of architectures, we have have hundreds or even thousands of alternative systems; ranking them using intuition would be too time consuming and inevitably error-prone. For the third, justifying one's intuition is remarkably hard - another heuristic suggests: "Regarding intuition, trust, but verify."

The main purpose of this thesis is to develop verification methods for engineers' intuitions on simplicity, or rather on the opposite of simplicity, complexity. If we are able to automatically determine the level of complexity of a system, we are able to proceed faster, and less error-pronely, in the design. Also, we would not need to rely on engineers' intuitions regarding complexity.

This thesis approaches its topic "Complexity measures for system architecture

models" in a way word by word. Only after defining all necessary concepts and doing thorough literature research are we able to really develop the new complexity measures and see how they work in some real-life example cases.

The section 2.1 discusses the concepts of system and system architecture. Defining these concepts precisely is important in order to know what is the thing we want to measure complexity of.

The section 2.3 defines what we mean by model. During desing phase of the product development process we do not have a system architecture per se, we have only different types of models of system architectures. We will define the requirements for any modelling methdology and resulting models: not any type of model supports complexity measures.

The section 2.4 discusses the reasons, why complexity of system architecture (models) matters. Basically, all reasons have to do with money: implementing/building a very complex system is going to be expensive, if at all possible. But there are other subtler reasons for which it is important to worry about complexity.

The chapter 3 defines the concept of complexity. We explain the statistical theory of information, the semantic theory of informmation, and the algorithmic complexity theory and then define complexity precisely. We also discuss the concept of equality of models.

The chapter 4 discusses several existing complexity measures after developing a set of requirements for sensible complexity measures. We also discuss the difference of complexity and complicatedness.

The chapter 5 then defines a system architecture modelling language, the Object Process Model-notation, which we use in our examples and in discussing the complexity measures. The chapter also explains the process of systems architecting, and in the following chapter 6 we develop example architectures models of three mobile entertainment systems.

In the chapter 7 we will develop complexity measures, which are meaningfull and usable with either OPM. We also apply the measures on the example models in order to check whether they are sensible.

Finally, we conclude the thesis with a short summary and a discussion of further research.

Chapter 2

Background

2.1 System Architecture

There are many different definitions of the concepts "system" and "system architecture". In this section, we will explore a few alternative definitions: by Crawley [6], by Rechtin [27], and by Muller [20] [19], and then adopt a set of definitions, which we will use in the rest of the thesis.

2.1.1 Crawley

In [6] Crawley defines system as

> *a set of interrelated elements which perform a function, whose functionality is greater than the sum of the parts*

and architecture as

> *the embodiment of concept, and the allocation of physical/informational function to elements of form, and definition of interfaces among the elements and with the surrounding context.*

and in other words

> *architecture is the details of the assignment of function to form, and the definition of interfaces.*

Form, function, and concept are system attributes, all created by the system architect. They are closely related as explained below.

Form is the set of elements, which together make up the structure of the system. Form can be decomposed into smaller and simpler forms, until we reach the atomic (not decomposable) parts.

The systems has a set of functions it performs while in use. The function of the whole system emerges as the form is assembled. Thus, the form is what is implemented and later operated. The externally, over the interfaces, delivered function defines, for the most part, the value of the system.

Concept is a vision, idea, or mental image of the system. It maps the form to the function. Concept is not a product attribute, it is a mapping. It involves a principle of operation and an abstaction of form and must allow for the execution of all functions, as specified by a set of system parameters. Formally, according to Crawley, the concept is the combination of the specific system operating processes, the specific system form object, and related through the generic form object.

In other words, Crawley states that

> *architecture is the details of the assignment of function to form, and the definition of interfaces.*

Related to the definition of interfaces is the definition of timing and operational sequence during run-time (use) of the system.

The architect starts architecting by identifying the needs of the beneficiary, who receives value by operating/owning the system, and converts the needs to system goals. The architect then desings the architecture, its form, function, and concepts, in such a way, that its meets the goals and thus also the needs of the benefiary. See section 5.2 for more discussion on the systems architecting process.

2.1.2 Dori

Dov Dori [8] has definitions, which are quite similar to Crawley's. Dori defines system as

system is an object that carries out or supports a significant function.

and further system architecture as

> *system architecture is the overall system's structure-behavior combination, which enables it to attain its functions while enbodying the architect's concept.*

Concept is, for Dori, a bit more concrete than to Crawley. Dori's defines concept as

> *concept is the system architect's strategy for a system's architecture.*

and function as

> *function is an attribute of object that describes the rationale behinds its existence, the intent for which it was build, the purpose for which it exists, the goal it serves, or the set of phenomena or behaviors it exhibits.*

According to Dori, some systems, the artificial ones, are products. He defines product as

> *product is an artificial system that is produced by an entity with the intent of selling it to another entity.*

2.1.3 Rechtin

Rechtin and Maier define system as [27]

> *system is a collection of different things, which together produce results unachievable by the elements alone.*

and further, that

> *these results derive almost solely from the inter-relationships among the elements*

which clearly emphasises the importance of interfaces, as with Crawlay above. We will later discuss the importance of interfaces in measuring complexityin detail in chapter 7.

Rechtin and Maier define system architecture as

> *architecture is the structure - in terms of components, connections, and constraints - of a product, process, or element*

and further systems architecting as

> *systems architecting is the art and science of creating and building complex systems. The part of systems development most concerned with scoping, structuring, and certification.*

Note, that [27] contains, in appendix C, a thorough discussion on definitions of architecture, systems architecting, and modeling.

2.1.4 Muller

In [20], Muller defines

> *System architecting is a means to create systems efficient and effective, by supplying overview, by guarding consistency and integriry, and by balancing*

Muller does not define, what he means by "system". He assumes that it is a well defined, or at least intuitively apparent concept. Muller has very elaborate method for systems architecting, but not very clear definitions for any concepts.

2.2 Definitions used in this thesis

In this thesis, we will adopt Crawley's definitions, because they allow as to make necessary distinctions later on, when we will develop systems architecture complexity measures. We will need definitions, and corresponding models, with which we

can clearly draw the system boundary, as well as distinguish forms, functions, and interfaces. We also need to be able to discuss and show the parameters of interfaces. For this reason, in chapter 7 we will extend the existing system architecture design notation.

2.3 Definition of a system architecture model

In this section we discuss the importance of modeling and models for systems architecting. We also establish some requirements for systems architecture models. Models, which meet the requirements, are going to be necessary, when we later develop complexity measures.

2.3.1 Importance of models

System architect does not build systems. Instead, he produces a set of models, which are textual and visual (pictures, graphs) abstract representations of the system [27]. Sometimes, a set of prototypes of increasing reality are built based on the models.

Models serve many purposes. The architect communicates with the client (beneficiary or his agent) using models. Before the system's detailed design and implementation (building) can start, the client and the architect must agree on the model of the system, i.e. on what is going to be designed and implemented. For this reason, at least some of the models must be such, that the client is able to understand them.

The architect also uses the models to communicate with the downstream designers. The designers must be able to understand, what the architect means with his models. The models must not contain (much) ambiguity, and they must contain enough details for the designer to do his work.

The system architect is, however, the most important user of his own models. The models help the architect to rigorously analyze architecture [6]. In this thesis, we are mostly interested in the models as they are used by the architect in his own work.

In short, Dov Dori's [8] definition of model sums up the reasons mentioned above for having models as

> *a model is an abstraction of a system, aimed at understanding, communicating, explaining, or designing aspects of interest of that system.*

2.3.2 Requirements for a modeling language

A model is an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system [27]. Any modeling system must have a set of words (a vocabulary), a grammar telling how the words may be combined, and a way of interpreting (semantics).

Since no single model can describe all aspects of a system, the architect needs a number of different models. The set of models should be as complete as possible so that no essential information or aspects of the system are left out of consideration. At the same time, the models should not be overlapping, there should not be duplication of information. Duplicated information is very hard to maintain and tends to lead to confusion and errors.

An ideal systems architecture modeling language should, according to [27], [6], and [29]

Be broadly applicable The language should be applicable to modeling physical, informational, organizational systems as well as any mixtures of them. The modeling language should not limit the size or complexity of the modeled system.

Represent Functions It is necessary to be able to represent functions independently of the forms, which perform the function. Without this possibility, it is impossible to do iterative systems architecting.

Represent Forms It is necessary to be able to represent forms independently of the functions, which they perform. Without this possibility, it is impossible to do iterative systems architecting.

Represent form and function in a common notation The notation must allow showing form and function in one picture (or textual representation). Without this possibility, we cannot show the links between form and functions, i.e. we cannot represent the concept.

Have precise semantics for all elements and their interconnections Since the models have many uses and users, they must be unambiguous and precise. Any inherent ambiguity will lead to confusion, errors, and thus to increased cost.

Allow symbolic manipulation Modeling language must allow symbolic manipulation. This is necessary for being able to automatically transform from one representation to another. For this thesis the most important transformation is from model to measures of complexity.

Have a precise semantics for denoting the system boundary We need to be able tell which forms and which functions are part of the system and which are part of the context of the system. For the first, and for this thesis, forms in the context increase complexity in a different manner that forms of the systems. For the second, deciding the system boundary is one of the most important architectural decisions.

Have a graphical notation The models serve as external memory structures [30], which enable the user of models to keep most of the information away from the short term memory. Humans are excellent in recognizing patterns and manipulating them. For this reason, having a graphical notation is essential.

Have a textual notation Sometimes we need to express the architecture using natural language, at least partly so. To do so, we need a way of representing the models in natural language. Also, we need to have a textual notation for the models in order to be store the models in computer and manipulate them.

Be able to represent topological connections We need to know the topological relations of forms. A form can surround another form, touch it, or be separate from it.

Be able to represent structural connections We need to know how the forms maintain their topological relations: how they are connected or kept apart. This information is very important when measuring complexity.

Be able to represent operational links During operation the connections between forms change, since they transmit information, energy, force, or stress. The modeling language must be able to represent this information.

Be applicable to multiple thinking styles Not all humans think alike. For some pictures are more important than words, for some other the other way round. The modeling language should not dictate the way of thinking.

Be a way of recording the work, and the progress of the work Even a precise way of modeling must allow incremental way of working, undoing decisions, and making changes while keeping track of them.

Allow incremental way of defining the system A modeling language, which only allows for complete descriptions of the system is useless as a design tool. The models must usable (for communication, for measuring complexity, etc) at any phase of design.

Declarative The language must be able to specify the vocabulary and grammar and to map the possible states of the system. The language must also be executable and allow computational simulations of models.

2.3.3 Summary

Models are the principal tools of the systems architect. He uses models for communicating with other member of the development team and with the clients. Moreover, he uses the models for enhancing his own thinking. For these uses the modeling language must meet the criteria presented in this chapter.

2.4 Reasons for measuring complexity

Measuring complexity for the sake of measurement would be a futile exercise, worthy sincere academic interest but of no value for practicing systems architects. Systems architects are interested in measuring or at least estimating complexity by their trade. Measuring and understanding complexity of proposed systems architecture (models) is, however, very important for the whole product development enterprise. As we will see, the more complex the system, the more expensive and risky is the design and implementation effort.

This chapter discusses reasons for worrying over and measuring complexity. Most reasons we are going to give are based on anecdotal evidence or intuitive reasoning, because there is no literature describing detailed, and controlled, experiments. There are quite natural reasons for this. First, there are no widely used systems architecture (model) complexity measures. Second, large systems development project are rare and not repeatable making empirical (comparative) studies hard to perform. Even software complexity measurement suffers from these two problems as we will discuss below and in chapter 4.

2.4.1 Complexity and cost

According to Rechtin [27]: *Generally speaking, the more complex a system, the more difficult it is to design, build and use*, and, intuitively, the more difficult a task, the more expensive it is, if not for any other reason than requiring rare experts or lot of time to complete.

Systems architecting phase of product development requires a very small amount of the total development budget, but deciding the architecture determines most of the total development cost. According to [29] and [35] systems architecting decisions are made after using often less than 1 percent of the total cost and they define up to 80 percent of the total cost. This is because changing architecture later during development is very expensive. Therefore, it is prudent to try to avoid mistakes in systems architecting. Measuring complexity, and trying to reduce complexity, is one

way of avoiding mistakes.

The same reasoning is found in [20], where Muller writes *an enabling factor for an optimal result is simplicity of all technical aspects. Any unnecessary complexity is a risk for the final result and lowers the overall efficiency.*

Boeing system architect Bob Liebeck claimed[1] that complexity depends on the part count, or the number of different parts. He was especially concerned about the long life of airliners and having to be able to provide spare parts for them. We can also assume, that the number of methods of inter-connecting the parts plays an important role in Boeing. Meyer [18] has a very similar argument:

> *Reducing complexity almost always reduces direct and indirect costs. Complexity fuels those costs, which grow geometrically if not exponentially. Every additional part requires that it be made or purchased, requiring time, people, and capital. Greater complexity means more purchase orders and more stockroom space.*

Furthermore, architectural complexity spills over to the organization design:

> *The complexity of [...] architecture will be mirrored in the firm's organizational complexity.*

Liebeck also emphasized the need to consider the whole product system, not only the product itself - the more complex the product, the more complex the supporting systems and the whole product system. Based on his experience, 85 percent of the cost will be determined after 5 percent of the work.

Software business has for a long time tried to measure complexity of software. Even though several measurement techniques have been developed, which we will discuss in chapter 4, a widely read SW project management book [17] uses a very simple measure: the size of the software to be developed estimated by experienced software engineer and type of the software (office software, system software, etc). The book claims, that such a simple measurement approach is enough to predict the cost of software development projects with sufficient accuracy.

[1] A lecture in MIT, October 2005

2.4.2 Complexity, predictability and managing product development

Lankford [15] provides an interesting reason for measuring complexity. If some subsystem is significantly more complex that the others, its is likely to become problematic in later phases of development and maintenance. This is intuitively true, especially if resources and attention are divided irrespectively of the distribution of complexity. Therefore, given a measure of complexity, systems architects and product development managers should strive for even distribution of complexity. If this is not possible, they should devote extra resources and attention to the more complex subsystems.

McGabe [16] states, that measuring complexity of a design is essential in being able to predict the cost and time needed to implement the design. He says, that before proceeding to implement the design, one must understand the complexity of it. Understanding the complexity of design also gives us a hint, whether the design as such is comprehensible for humans. Empirical studies referred to in [16] show, that there is strong positive correlation between measured complexity and number of errors found in the implemented system. McGabe's statistics are from software development, but his reasoning applies to other systems as well.

Not all problems in product development are due to complexity. Sometimes, they are due simple misunderstanding, lacking skills and knowledge, or sheer carelessness. Measuring complexity of problematic subsystems gives an idea whether the problems are inherent in design or somewhere else.

Complexity measurement is also important, when developing new versions of existing products. While trying to minimize complexity is important, addition of new functionality to meet the need of the beneficiaries increases complexity. Furthermore, complexity increases during product development after systems architecting has been completed. Measuring complexity of a ready product is important and can be done while updating design documentation at the end of product development projects. If complexity has increased significantly, trying to reduce it during development of the succeeding versions is very important.

2.4.3 Complexity, disruptive technologies, and creativity

Complexity plays an important role in the long-term development of industries. Hendesson [12], Utterback [33], and Christensen [5] have developed theory of disruptive technologies. In short, a simple version of this theory claims that each industry converges to a dominant design.

A dominant design is the one that wins the allegiance of the marketplace. To be specific, a dominant design is a new architecture which puts together individual innovations that were introduced independently in earlier products. A dominant design makes many performance requirements and product features implicit. All products in the market must adhere to the dominant design in order to meet the fundamental customer needs and expectations.

When a dominant design emerges, the focus in industry moves from product to process innovation. The number of firms drop to only a few, all of which have significant share of the market.

Every now and then, some new company invents or develops a new technology, which is able to disrupt the market, gain larger and larger share of it, and finally, in many case, even drive the incumbent firms out of the business. This happens even though the incumbents usually have even working prototypes of disruptive technologies ready. The incumbents somehow cannot see the disruption in time. One reason, according to Christensen, is that the disruptive technology is usually simpler and has worse (conventional) performance. Thus it is hard to sell to existing customers, and its commercialization is hard to justify.

According to the empirical evidence, many disruptive technologies are architectural innovations. A disruptive technology reinforces existing core concepts, but changes the linkages between concepts and components [12], or in our terminology: change in the concept, or mapping of function to form.

Since disruptive technologies are often simplifying architectural innovations, some disruptions could be foreseen by following the development of complexity of products. Since the incumbent companies often have both the dominant design and the new de-

sign at hand, they could easily see whether there are dramatic changes in complexity. If there are, the new design can be disruptive. In this way, complexity measurements could help in making strategic resource allocation and product roadmap decisions.

Furthermore, similar reasoning seems to apply for evaluation creativity of new designs. According to Horowitz [13], there is a class creative solutions, which satisfy two conditions. First, the solution does not add any new objects to the system or its immediate context. Second, a problem characteristic needs to change qualitatively from an increasing function to a either decreasing or an unchanging function. Intuitively, this would mean a less complex new and creative design. This claim is, however, just a guess. It would be interesting to study further the relationship between creativity and complexity. Such a study is beyond the scope of this thesis.

Chapter 3

What is complexity

Having an exact definition of complexity is a necessary condition for being able to discuss complexity and to measure complexity of a systems architecture models. Having defined complexity, we will, in chapters 4 and 7, explore different ways of measuring it.

In addition, we need to clarify the relationship between complexity of a model of a system and complexity of the system itself. Intuitively, it seems that the more complex the model, the more complex the system, but this may not always be the case.

In this chapter, we first discuss, based on [26], the concept of information and two theories of information: the syntactic one and the semantic, which are the basis of some accounts of complexity of systems. The theory behind most accounts of complexity of systems is the algorithmic complexity theory, which we introduce next. We then give our own formulation of the concept of complexity based on the algorithmic theory. Finally, we need to discuss equality of systems, because comparing models in terms of complexity makes sense only if the models are models of equal systems.

3.1 Preliminary notes

The basic idea behind all definitions of complexity is: the more information an expression, or model, contains, the more complex it:

$$I(M) > I(N) \Rightarrow K(M) > K(N) \tag{3.1.1}$$

where M and N are models, $I(M)$ is the amount of information in M, and $K(M)$ is complexity of M.

The definitions differ in the way they measure the amount of information. Before we discuss the definitions, we need to make some remarks, which will be necessary later.

Information is what is communicated from sender to the receiver, the contents of the message. Information is conveyed by the information bearer, which is a set of linguistic expressions or symbols. It follows, that a system (a satellite, a table, a submarine, etc) does not contain any information[1]. Instead, it is the model of the system contains information. The model is the bearer of information between sender and receiver, for example among beneficiaries and architects, among architects, and among architects and people working in the downstream processes.

We need to remember to keep *use* and *mention* of an expression distinct. This distinction is due to Frege [10] and to Quine [24]. Use is when use the expression to say something, mention is when we speak about the expression itself. In particular, when we use the model to describe the system, we use it as an expression. When we use the model for determining the complexity, we mention the model.

We also need to remember the distinctions between object language and metalanguage. We speak about object language using metalanguage. In some cases they may be the same, but usually it is easier to keep them distinct.

[1]Unless it is used as a symbol in some expression or model

3.2 The statistical theory of information

Statistical theory of information, as developed by Shannon [28], is an answer to the question: given a set of messages m_i each of which occurs with probability p_i, what is the amount of information they convey. The first step is the determine the amount of information provided by a single message m_i, which is

$$I(m_i) = -log_2\, p_i \tag{3.2.1}$$

Let then X be a set of discrete random variables with values $x_1, x_2, ..., x_n$ with x_i having probability $p_i (1 \leq i \leq n)$. Then we define the (Shannon) *entropy* of X, or $H(X)$ to be

$$H(X) = H(p_1, ..., p_n) = -\Sigma log_2\, p_i \tag{3.2.2}$$

This definition of entropy is purely syntactic. It does not say anything about the content, or meaning, of the messages. It is only concerned about the probability of occurrences of different messages. The higher the probabilities are, the higher the information content of the message.

In order to talk about the content of message, we need to turn to the semantic theory of information.

3.3 The semantic theory of information

In order to get from the content as entropy, as defined by the statistical theory of information, to the content as what is expressed by the messages, we distinguish the probability of string of symbols (a message) from the probability of the state of affairs the message expresses. This is done in the semantic theory of information developed by Carnap and Bar-Hillel [3].

Formally, let $W_1, ..., W_n$ be the set of mutually exclusive maximal consistent set of sentences in a language L. This means, that in L every sentence is compatible with some of W_i and excludes others. Assume further, that with probability p_i the state

of affairs $W_i (1 \leq i \leq n)$ holds, and that $p_1 + ... + p_n = 1$.

If $p(H)$ is the probability of state of affairs expressed by sentence H, we have two measures of information

$$cont(H) = 1 - p(H) \tag{3.3.1}$$

$$inf(H) = -log\, p(H) \tag{3.3.2}$$

$cont(H)$ can be said to be the substantive information of sentence H, whereas $inf(H)$ is a measure of the surprise value, unexpectedness, of H.

3.4 Algorithmic complexity theory

The basic idea behind algorithmic complexity theory is to measure the size of a program producing a given string by a given machine (usually a Turing machine). The theory was developed independently by Solomonoff [31], Kolmogorov [14], and Chaitin [4] in 1960s. To put it simply, the more difficult an object is to specify or describe, the more complex the object is.

Formally, given a Turing machine T, and a binary string s, we define the complexity of s as the size of minimal program p, which when given to T, prints s and halts.

Following [26], we have the following definitions of *algorithmic complexity* and *conditional or relative complexity.*

Definition 3.4.1. *The* algorithmic complexity *of a string s relative to Turing machine T, $K_T(s)$ is* $\min\{l(p) : T(p) = s\}$, *where $l(p)$ is the length of program p.*

Definition 3.4.2. *The complexity K_T of x* conditional *to y is defined by $K_T(x/y) = \min\{l(p) : T(p) = s\}$, and $K_T(x/y) = \infty$ if there is no such p.*

By using a coding, we can extend these definitions to any countable domain of object. The domain of man made systems is clearly such a domain.

3.5 Definition of complexity

In this thesis, we adopt the definitions of the algorithmic complexity theory. We define as follows.

Definition 3.5.1. *Given a system S and a coding $c : S \rightarrow L$ of system S in a language L, we call $c_L(S)$ the model of system S in L. Given a Turing machine T, the complexity of $c_L(S)$, $K_T(c_L(S))$, is defined by* $\min\{l(p) : T(p) = c_L(S)\}$.

Using this definition we can compare complexity of any number of models of any number of systems as long as we use the same coding language and same Turing machine.

The definition above does not say anything about the complexity of the system itself. To do so, we would have to prove, that the coding is a bijection, which we do not attempt to do in this thesis. We will, however, assume that complexity of the model correlates with cost and difficulty of implementing the system.

The definition is objective. It does not depend on the observer/measurer/designer. Of course, an observer/measurer/designer has some innate capability of managing complexity, which means that for each observer there is a maximum level of complexity, which he can tolerate. If the complexity increases over that level, the observer will not be able to understand the model and will make mistakes in design and implementation.

Furthermore, we will not attempt to find the minimal program p. Instead, we will use some characteristics of the modeling language and resulting models are proxies, which we take as reflecting the length of the minimal program p. We will justify our choices later in chapter 7.

3.6 On equality

When comparing systems and their models in terms of complexity, we need to define equality relations among systems and among models. In logic there are two ways to define equality: extensional and intensional. Two functions are extensionally equal,

if they both give same output when given same input. In other word, there is no way of telling the function apart by observing them from outside. Intensional equality of functions means, that the functions are similar also internally.

Consider then the main question we want to answer: "Which model of a system is the most (or least) complex?". Different models, when implemented, will yield different internal structures. Thus, if we would use intensional definition of equality, our question would become meaningless. Each model would denote a single system. Comparing models would not make any sense.

Instead, adopting extensional definition of equality makes our question sensible. We only require, that the systems, the complexity of models of which we want to compare, are not distinguishable from outside. This means, that they produce the same output with same input. We need, however, to interpret input and output in a wide sense: they cover also all external interfaces of the system.

In short, we consider two systems equal, if they share the same context, and if we can replace one with another in the context.

When comparing systems architecture models using our definition, in practice, we need to decide the level of abstraction (or detailedness) of the coding c_L. According to our definition, the more details (or the lower the level of abstraction), the higher the complexity of the model. If the models have the same parts and only differ in the way the part combine with each other, i.e. in interfaces, they clearly have same level of detail. But if the models have different parts, the situation in mode difficult. In this thesis, we do not try to develop a system of classifying abstraction levels of models. Instead, we rely on the intuition of the systems architect; he must consider the levels of abstraction of models carefully and make sure that they are equal.

Chapter 4

Existing complexity measures

Both software and mechanical engineering communities have developed several approaches for measuring complexity of their models. In this chapter, we will discuss a number of them. The purpose of the discussion is to gather ideas for the system architecture complexity metrics, which we will develop in chapter 7. For that purpose, we will also review what systems architecture literature [6], [27], and [8] has to say about complexity measurement.

We begin by presenting general requirements for any sensible complexity measurement. We use the requirements together with our definition of complexity to critique each proposed metrics.

4.1 Required properties of complexity metrics

Edmonds [9] requires that a complexity measure must not depend on the observer. McCabe [16] presents an informal set of desired properties of complexity metrics. In short, McCabe requires, that the metric related to the cost of the systems and is not counter-intuitive. In detail, a metric should be

1. objective and mathematically rigorous
2. related to the effort to integrate the design
3. of operational help

4. helpful in generating integration test plan

5. lending themselves to be automatized.

All these properties are clearly meaningful. Edmonds does, however, have still another requirement for a metric. A metric should be "intuitively correlating to with the difficulty of comprehending the design". It is not clear, whether every metric has to intuitively correlate with the difficulty of comprehending the design. For the first, the criteria depends on the observer. For the second, it is very difficult to say what "intuitively" means in this context.

AlSharif [1] develops, based on the work of Weyuker [34], a more formal set of required properties of complexity metrics. According to AlSharif, any sensible complexity metric must satisfy the following nine properties. Here M and N are systems and $K(M)$ denotes complexity of M[1].

1. *There exist M and N such that $K(M)$ differs from $K(N)$*, for the a metric, which gives the same value for all systems is useless.

2. *For any non-negative number p, there are only infinitely many systems of complexity c*, for the metric should have a range of values. Too narrow range of values is not practical.

3. *There are distinct systems M and N for which $K(M) = K(N)$*, for a metric which gives each system unique value is not useful - such a metric would be simple a bijective renaming of systems.

4. *There are functionally equivalent systems M and N for which $K(M) \neq K(N)$*, for the structure of the system determines (at least a partly) its complexity.

5. *For all M and for all N, $K(M)$ is smaller than $K(M \cup N)$, and $K(N)$ is smaller than $K(M \cup N)$*, for a system is more complex than its subsystems.

[1]For simplicity, we keep notation minimal here. $K(M)$ is an abbreviation of $K_T(c_L(S))$ in terms of our definition 3.5.1.

6. *There exists M, N, and O such that $K(M) = K(N)$ and $K(M \cup O) \neq K(N \cup O)$*, for O may interact with M in different manner than with N. Namely, the interfaces between M and O may be more complex than interfaces between N and O.

7. *There are systems M and N such that N is a permutation of components of M and $K(M) \neq K(N)$*, for changing the way connecting the components to each other, i.e. interfaces, may change the level of complexity.

8. *If M is a renaming of N, then $K(M) = K(N)$*, for complexity does not depend on the name of the system.

9. *There exist M and N such that $K(M) + K(N)$ is smaller than $K(M \cup N)$*, for putting systems together creates new interfaces.

Clearly, our definition of complexity (3.5.1) meets all these requirements.

4.2 Halstead difficulty

The Halstead difficulty is, according to [15], one of the earliest software complexity measures. It is one of Halstead's measures, which can be automatically derived from the source code of any program. If we denote the number of distinct operators by n_1, the number of distinct operands by n_2, the total number of operators by N_1, and the total number of operands by N_2, then *difficulty* of the program is

$$(n_1/2) * (N_2/n_2) \tag{4.2.1}$$

Furthermore, program *length* is defined as $L = N_1 + N_2$, *vocabulary* as $V = n_1/n_2$, *volume* as $L * \log_2(V)$.

This measure is said to reflect the computational complexity of the program. It is not intuitively clear why that would be the case. The measure has been criticized, mainly because it measures textual complexity of the program code, not the structure

of the program. For this reason, this measure does not meet requirement number 7 above; all permutations of program code will have same Halstead complexity.

4.3 Graph based: McCabe's cyclomatic complexity

Cyclomatic complexity measures the number of linearly independent control paths through a software program. McCabe [16] defines the measure as

$$v = e - n + p \tag{4.3.1}$$

where e is the number of edges, n is the number of number of vertices, and p is the number of distinct connected components in the control flow graph of the program, respectively. For a system to testable and understandable, v should be less than 10, according to McCabe.

Further, McCabe defines *module design complexity* S_0 as the cyclomatic complexity of the reduced graph of the design. The reduced graph is a minimal graph, which has the same interrelations than the original graph. The complexity of the whole design is sum of complexities of its modules. The *integration complexity* S_1 of a design with n modules is then

$$S_1 = S_0 - n + 1 \tag{4.3.2}$$

McGabe's complexity measurement takes into account both the program texture (the number of vertices) and its structure (the number of edges). It is thus more advanced and credible measurement than that of Halstead's. In addition, it fulfills all our requirements. McGabe's measure also agrees with our definition of complexity, as the higher the cyclomatic complexity, the longer program will be needed to print out the model.

4.4 Graph based: Fan-in and fan-out

Two common complexity measures for ordered graphs are *fan-in* and *fan-out.* Fan-in is the number of edges coming in a vertice, fan-out is the number of edges going out from a vertice. By calculating the maximum, minimum, and average fan-in and fan-out it is possible to get an understanding of complexity of a model.

Fan-in and fan-out meet our requirements for good measurements. They also satisfy our definition of complexity, because the higher the value of fan-in or fan-out, the longer program it takes to print out the model, even if the model would be highly regular as representing the regularity requires the longer program, the more comprehensive the regularity is.

4.5 Graph based: RSU and PSU by Dincel

Dincel et al [7] present a way service-definition based method of measuring complexity of product line architectures. The method is based on observation, that the more services components provide and require to and from each other, the more complex the overall architecture.

The basic concepts are *required service utilization* (RSU) and *provided service utilization* (PSU). For a component, RSU indicates how large a portion of the services the component requires are provided by other components. PSU indicates how large a portion of services provided by a component are used by other components. For any system, RSU should be 1: all components should get all services they need. There is no need for PSU to be 1, however, much lower PSU indicates unused services in the system, which causes unnecessary costs.

RSU and PSU do not satisfy the 5th requirement for sensible complexity metrics.

4.6 UML-based methods

Lankford [15] presents a way of adapting Halstead difficulty and McCabe cyclomatic complexity to software architecture descriptions, especially those defined in UML.

For measuring the Halstead of a software architecture, Lankford calculates the volume of architecture using the number of unique classes, the number of messages (for sequence diagrams) or the number of relationship (for class diagrams). For state diagrams, he calculates the volume using the number of states and number of state transitions.

For measuring the McCabe cyclomatic complexity, Lankford uses the number of messages, messaging participant and the number of connected graphs (from sequence diagrams), the number of relations, the number of classes and number of graph connected components (from class diagrams), and the number of transitions, states and connected state diagrams.

4.7 Regularity based method

Kazman [25] presents a method for measuring the complexity of an architecture based on the degree of regularity the architecture has. Intuitively, the more regular a description is, the simpler it is and the shorter the program required to print the model of the program.

The method is based on matching the architecture with a set of known architectural patterns. The smaller is the set of patterns required to cover the architecture, the simpler then architecture. In this way, Kazman's measurement meets our definition. I does not, necessarily, meet the requirements set above. For example, covering a combination of two systems may require less patterns than covering each of the combined system. For this reason, the set of patterns must be chose carefully or using regularity as a complexity metric is not sensible.

4.8 Meyer's part counting method

Meyer [18] presents a simple way of measuring complexity. First, calculate the number of parts, N_p, the number of types of parts, N_t, and the number of interfaces of each

of the parts, N_i. The complexity factor is then

$$(N_p * N_t * N_i)^{1/3} \tag{4.8.1}$$

This measure is very intuitive and fulfills all our requirements for a complexity method and is clearly usable when we have complete (down to level of individual parts) model of the system. When this is not the the case, the main problem with Meyer's measurement, as with other's presented in this chapter, the equal treatment of all part types and interfaces, appears. As we will see, this a deficiency which we need to correct.

4.9 Suh - information theory based method

Nan Suh [32] bases his theory of complexity on semantic theory of information. Suh defines

> *complexity as a measure of uncertainty in achieving the specified functional requirements.*

The idea is that the greater the information needed to achieve the functional requirement, the greater is the information content (of the functional requirements) and thus the greater the complexity.

Since our definition of complexity is not based on the semantic theory of information and thus Suh's definition of complexity does not correspond to our definition of complexity, we will not use it in this thesis. Suh definition does, however, meet our requirements for a complexity measure.

4.10 Dori's simple-complex continuum

Dov Dori [8] defines complexity as

> *Complexity is an attribute of a thing that determines whether it is a simple or non-simple.*

A simple thing has no parts, attributes, or specialization defined for it in the system. All other things are complex. Dori further distinguishes four types of complexity: aggregation, exhibition, generalization, and classification complexities. He does not give any specific measure for calculating any of these complexities. He just refers to the possibility of counting elements or interconnections, classified in the four complexity classes and compared to the number of process or objects. We will return to Dori's methods later in the chapter 5, when we discuss his Object-Process Methodology.

4.11 Crawley: actual and perceived complexity

Crawley [6] defines a complex systems as

> *A system having many interrelated, interconnected, or interwoven elements and interfaces.*

and a system which *requires a great deal of information to specify.* He emphasizes that complexity is an absolute and quantifiable system property, which is not dependent on the observer. The observer's limits of comprehension, or tolerance of complexity, defines the upper limit of acceptable complexity: if system is more complex that the observer (architect, designer, implementor) can comprehend, the system will be expensive and error-prone to architect, design, implement.

Crawley defines three complexity measures: the number of interconnections and their types, the sophistication of the interconnections, and sensitivity or robustness of the interconnections. All these are measures of the amount of information required to specify the system. Thus, Crawley agrees with definitions of Halstead, Meyer, and Dori above, but adds the idea that the quality or nature of interfaces is important. In other word, some interfaces are more complex than others. We will develop this observation further in chapter 7 based on Crawley's statement that suppressed operational or structural links are the interconnections of interest.

Crawley defines *part*, an element that cannot be taken apart, as the atomic unit of measurement of complexity. A *module* is a collection of elements. Parts are connected

with interconnections, which are one of the four types: logical relational, topological, implementation, or operational. We will discuss these, and interrelations in general, in detail in chapter 7.

Crawley distinguishes between essential, perceived, and actual complexity. Essential complexity is the minimum amount of complexity, which is necessary to deliver the required functionality. Perceived complexity is the complexity, which the observer perceives when looking at a model of certain abstraction level. Actual complexity is the amount of complexity that the system has. According to Crawley, it is necessary to keep perceived complexity below the limit of understanding (comprehension) and the actual complexity close to the essential complexity. The actual complexity is never smaller than the essential complexity. This is shown in figure 4-1, which also

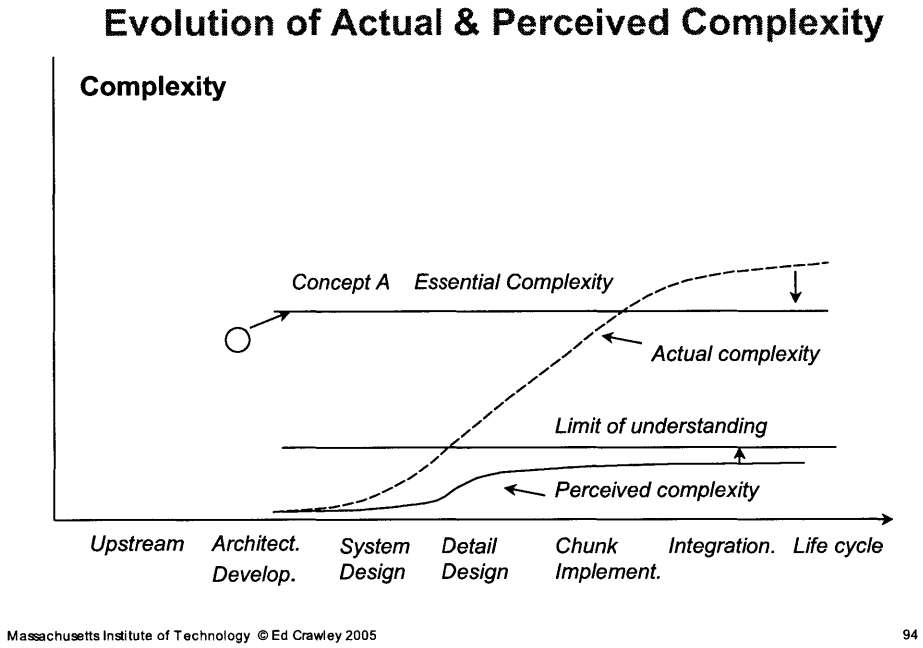

Figure 4-1: Evolution of complexity [6]

shows how the actual complexity increases as the abstraction level decreases during the product development process. We will discuss this later in chapter 7.

Chapter 5

OPM - a system architecture model

In chapter 2.3, we argued that for a systems architect a modeling language is his most important tool. We also listed a set of requirements for a ideal modeling language. In this chapter, we will introduce a modeling language: *Object-Process Methodology* (later, OPM) developed by Dov Dori [8]. OPM is a modeling language meant for modeling any kind of systems. Crawley has developed a method for using OPM in modeling systems architectures [6].

5.1 OPM

OPM consists of *Object-Process Diagrams* (later, OPD) and a natural language translation called *Object-Process Language* (later, OPL). They are equivalent in their expressive power and there is a simple translation between them. According to Dori, both are necessary for accommodating different types of thinking styles. Here, we will use only OPDs.

OPM has three basic entities, or things: objects, processes, and states. The objects exists, exactly as Crawley's forms in section 2.1.1, and processes transform objects either by generating, consuming, or affecting them. Object have states, which processes can change.

The graphical notation of OPD, shown in the figure 5-1, for objects, processes, and states are consists of rectangles, ellipses and ellipses, respectively.

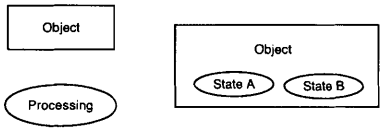

Figure 5-1: OPM basic things

The OPM hierarchy of entities, things, object, processes, and states is shown in the figure 5-2.

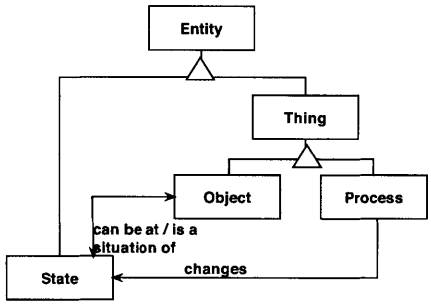

Figure 5-2: OPM entity hierachy

Both objects and processes have four structural relations. An object may be a part of another object (so called *aggregation-participate*-relation), an attribute of another object (so called *exhibition-characterization*-relation), of same type as another object (so called *generalization-specialization*-relation), or an object of objects of a certain class (so called *classification-instantiation*-relation). Similar relations are applicable to processes. A process can, in addition, be linked to an object with an exhibition link. OPM has the following (figure 5-3) graphical notation for these relations. Note, that an aggregation relation may have explicit constraints on the number of parts.

Structural links which may have explicit tags, exists between objects. Processes consume objects and result in other objects. They may also affect some objects. OPM has the following graphical notation (figure 5-3) for structural links.

Processes may have *enablers*, or *instruments*. An human *agent* is a special enabler, for which OPM has a specific notation. The notation for enabler is the following

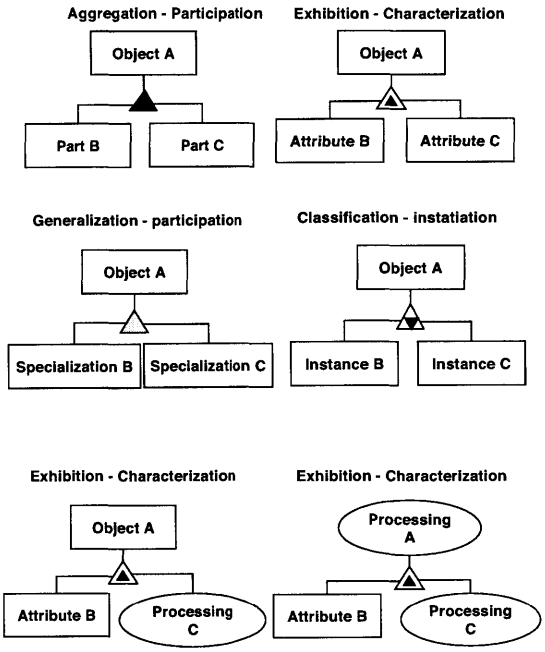

Figure 5-3: OPM basic relations

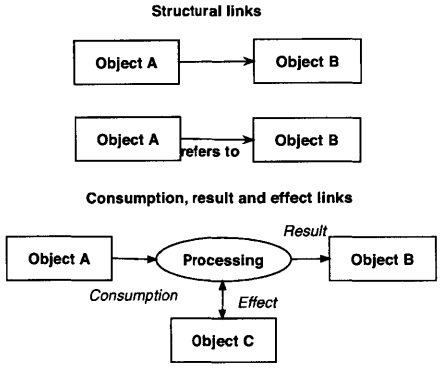

Figure 5-4: OPM structural relations

(figure 5-5).

OPM allows for *zooming in* processes. This is necessary for being able to work on different levels of abstraction: zooming in increases the level of detail. A zoomed-in process has objects and processes inside it. It also inherits the instruments and structural links from the outside (surrounding OPD).

A process must transform an object. Graphically this means that it is not syntactically legal to connect two process with a structural link. In some cases the

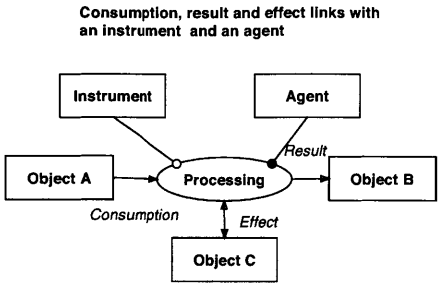

Figure 5-5: OPM instruments

inter-process object is not significant and can be omitted without causing confusion. In such cases, an *invocation link* is used as shown in the figure 5-6.

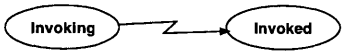

Figure 5-6: OPM invoking link

Equipped with this introduction to the notation of OPM we are able to apply OPM to an exemplary systems architecture case. The case will enable us to start to talk about complexity measures in details.

5.2 Applying OPM to systems architecture modeling

5.2.1 Level 0 : needs, concept, processes, objects

Systems architecting process begins, according to Crawley [6] by identifying the value provided by the system and getting from the value to concepts. Even though systems architecting process is highly iterative, we will present it here as a linear sequence of steps.

The following OPM-diagram (figure 5-7) identifies the necessary information in this phase, in the level-0 phase of architecture. The colors of objects are meant to emphasize the relative importance of certain object/forms[1].

[1]We will use object as equivalent to form and process as equivalent to function.

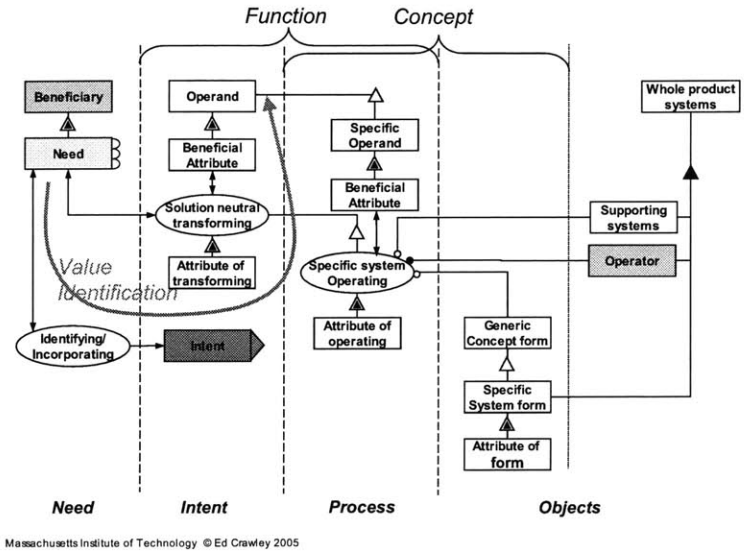

Figure 5-7: OPM: from value to concept [6]

The process starts by identifying the beneficiaries, or customers, of the systems. In addition to beneficiaries, each system has other stake-holders, such as investors, regulators, and employees. Each beneficiary has some needs, or overall desires or wants. Beneficiaries and their needs together make up the need definition for the system.

Next, the architect must identify the operand of the system, or the value related thing the system transforms, and the beneficial attribute of the operand. The transformation is done by the solution neutral process, which may have some attributes. The solution neutral transformation together with the needs of the beneficiaries makes up the intent of the system.

The next step is to determine the solution specific process. To do this the architect may have to specify the operand further in to specific operand and its attributes. This completes the process definition. In order to complete the concept, a mapping from function to form, definition, the architect next determines the generic systems form, specifies it to the specific form and its attributes.

In addition, the architect has to specify the supporting systems of the system, the

whole product systems, and the roles of operators.

5.2.2 Level 1: specializing intent, zooming processes, decomposing objects

Once the architect has captured the needs and developed the intent and concept, his next task is to expand the view of system to level 1. To get to the level 1, the architect flows down the process and operational intents and expands the concept by decomposing the objects and by zooming the processes. The figure 5-8 shows the overall process of moving from level 0 to level 1. In the figure, level 0 process

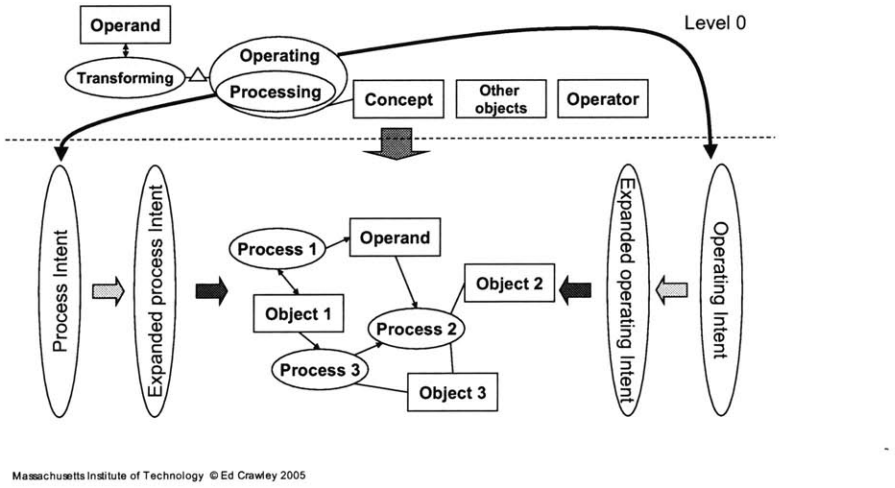

Figure 5-8: OPM: from value 0 to level 1 [6]

intent flows down to level 1 intend and gets extended. Similarly, the operations intent flows down and gets extended. The architect also has to zoom in the processes and decompose the form, i.e. to make the concept more detailed (less abstract). There resulting level 1 architecture of the refrigerator is shown in the figure 5-9

The figure 5-10 shows process and operational intents in more detail. The level 1 process intent consists of the primary intent (directly from level 0), other (non pri-

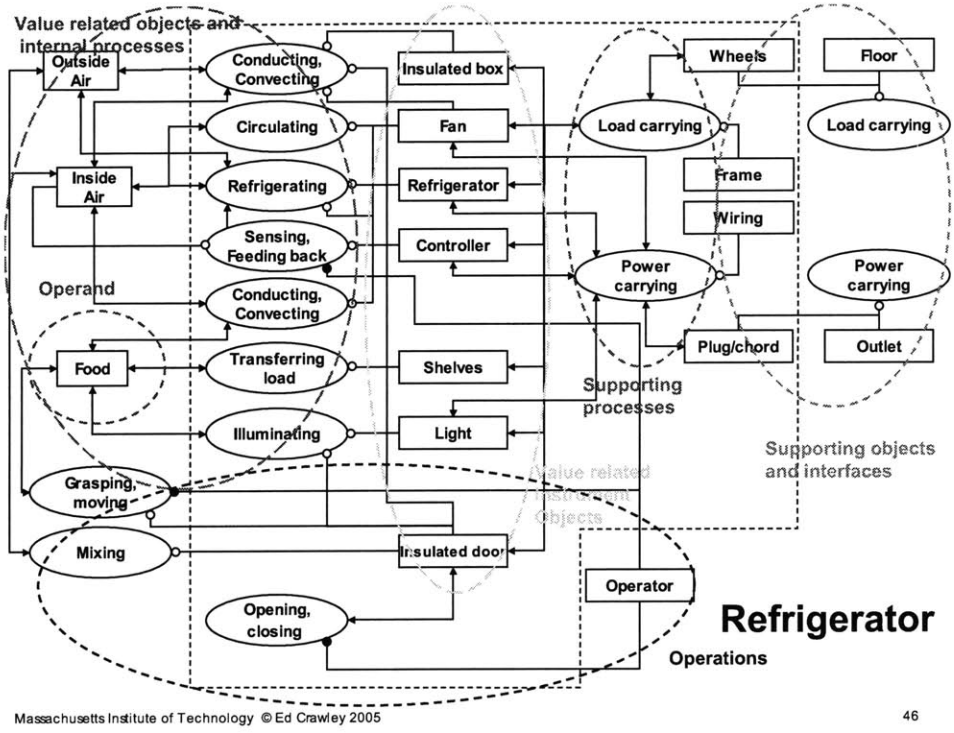

Figure 5-9: OPM: level 1 architecture of refrigerator [6]

mary) value creating processes, interfacing processes, and hidden processes. These are the intent sources. In detail, the interfacing processes consist of supporting (physical) processes, powering (heat exchanging) processes, and information (control, thought) exchanging processes. In addition, there may be some hidden intents, which are intents on hidden processes such as (from upstream): marketing, strategy, regulatory process ; (to downstream): developing, implementing, manufacturing, installing, etc.

The figure 5-11 shows the generic sequence of operational processes. Some of them, the most important for the system being architected, need to be shown in the level 1 decomposition and architecture model. First, the system is waiting in storage after which it is taken in use by executing the commissioning process, of which loading and preparing are parts. Next the system will be working in stand alone mode, namely without requiring interaction from most of the whole product system. This operational mode is used for trouble shooting and other similar tasks.

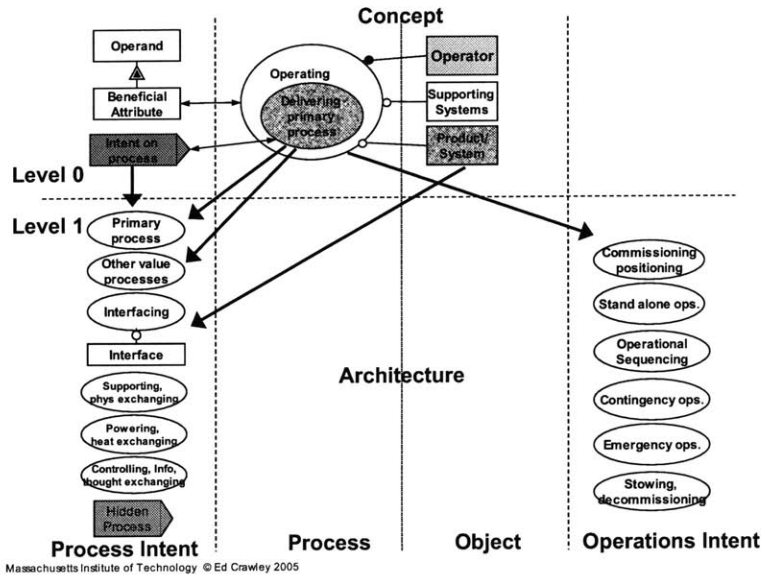

Figure 5-10: OPM: level 1 processes and operational intents [6]

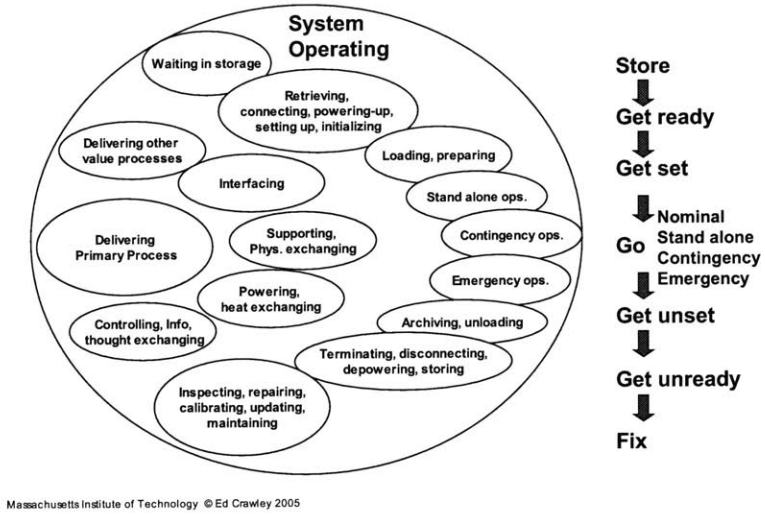

Figure 5-11: OPM: level 1 operational processes [6]

Contingency processes take care of graceful degrading of service (value) providing in case of errors, when there is no risk for losing property or life. Emergency operations

handle situations, where life or property are at risk, and may prevent all value delivery. The architect also has to plan processes for taking the system out of use. The processes from archiving via storing to disposal take care of these.

In addition to these special operational modes, there are the normal operational mode (the normal value delivery processes) and the maintenance/repair/update modes of operations. A complete description of the systems architecture must contain design for all these modes. It is not necessary to model all of them in level 1, but it is necessary to show their existence (or non-existence - not all systems have emergency modes, for example).

5.2.3 Level 2: Decomposing and zooming objects and processes

Getting from level 1 to level 2 requires recursive application of the methods for getting from level 0 to level 1. In detail, each level 1 process becomes a level 2 intent. The level 2 processes inherit level 1 interfaces and connections. Each level 1 objects/process-pair becomes a concept at level 2. The figure 5-12 illustrates this step.

The figure 5-13 shows, as an example, how the process of "refrigerating" zooms in and the object of "refrigerator" decomposes from level 1 to level 2. In this example, the object "refrigerator" decomposes to the objects "inner wall", "tubing", "refrigerant", "pump", and "radiator". The process "refrigerating" zooms in the processes "conducting, convecting", "transporting", and "decompressing, expanding". The process and object then have the relations shown in the figure 5-13.

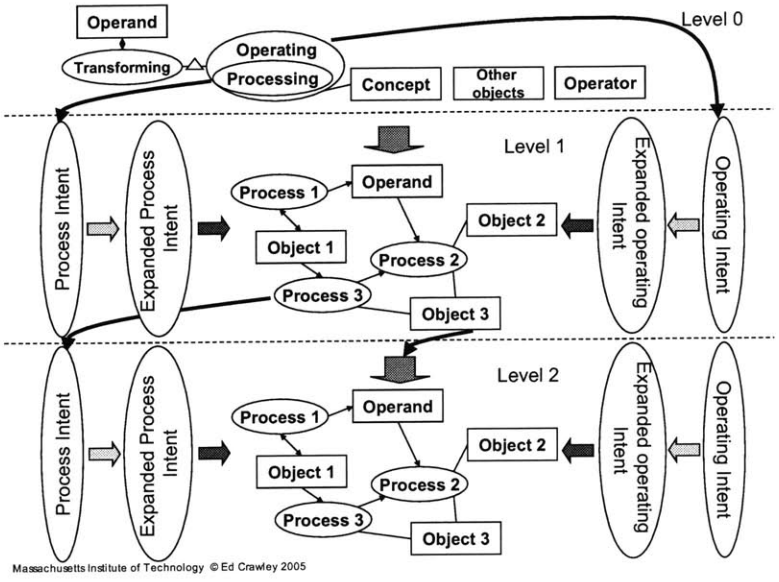

Figure 5-12: OPM: from value 1 to level 2 [6]

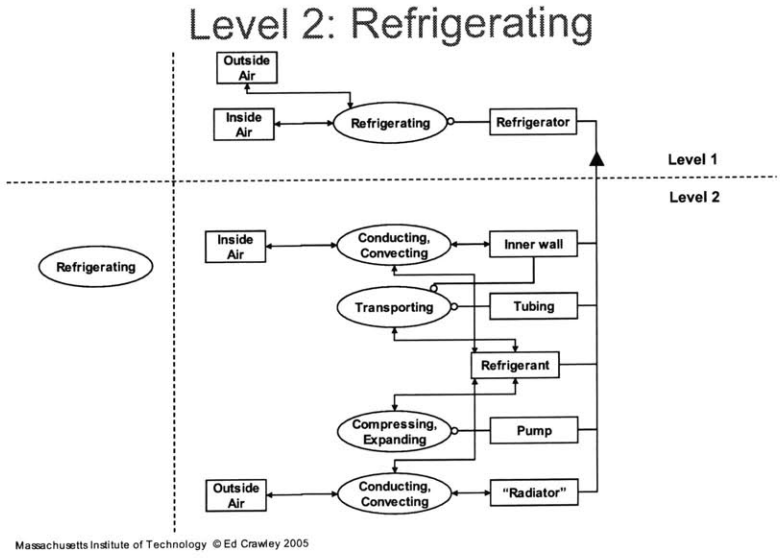

Figure 5-13: OPM: from value 1 to level 2 - a detail [6]

Chapter 6

An example: mobile entertainment systems

In this chapter, we develop an extended example of modeling a systems architecture using OPM and the methods of Crawley, which we introduced in the chapter 5. We use mobile entertainment systems as our example system. The example serves two purposes. First, we will see how usable and helpful the modeling methods are in modeling mobile entertainment systems. Second, we will later use the example when discussing complexity measurement approaches for OPM (and other similar) models.

6.1 Level 0 - system problem statement and concepts

The figure 6-1 shows the level-0 model of the mobile entertainment system. The beneficiary is the "person", who uses the mobile entertainment system. He has the need of being entertained. The operand is entertainment, which has three specializations: sounds, pictures, and games, which in turn have desired properties of interactivity and novelty. The system provides location independent availability of mobile entertaining using a delivery systems to mobile devices. The delivery happens faster than 128 kbit/s. The devices fit in a trouser pocket. The operator of the system is the

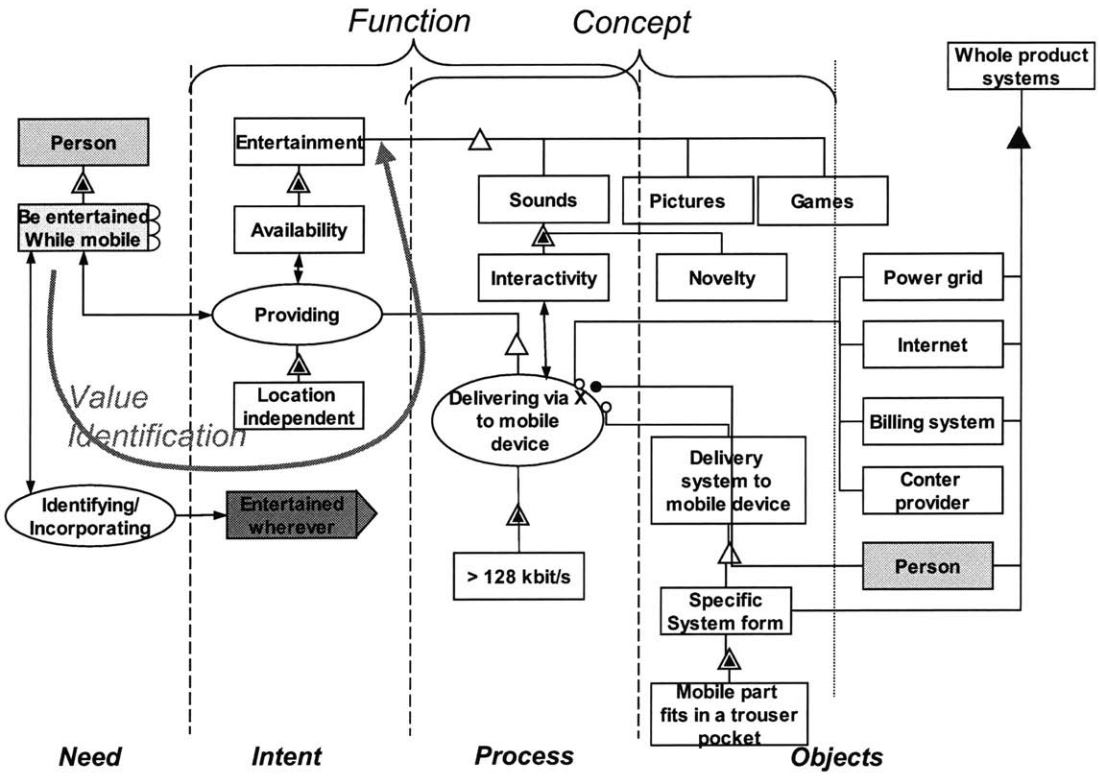

Figure 6-1: Mobile entertainment - level 0: system problem statement

person himself.

The figure shows the supporting systems (power grid, internet, billing system, and content (or entertainment) provider), which make up the whole product system.

Note that figure 6-1 does not elaborate on the specific system. Instead, it bears the name "Delivering via X to mobile device". Three concepts for mobile entertainment delivery systems are shown in figure 6-2. In the first concept "Delivering via Mobile network"-process uses a "Mobile network distributor"-object for providing availability of mobile entertainment. Examples of such object are the Preminet-system by Nokia [21] and the BREW-system by Qualcomm [23].

In the second concept "Delivering via FM-radio and mobile network"-process uses

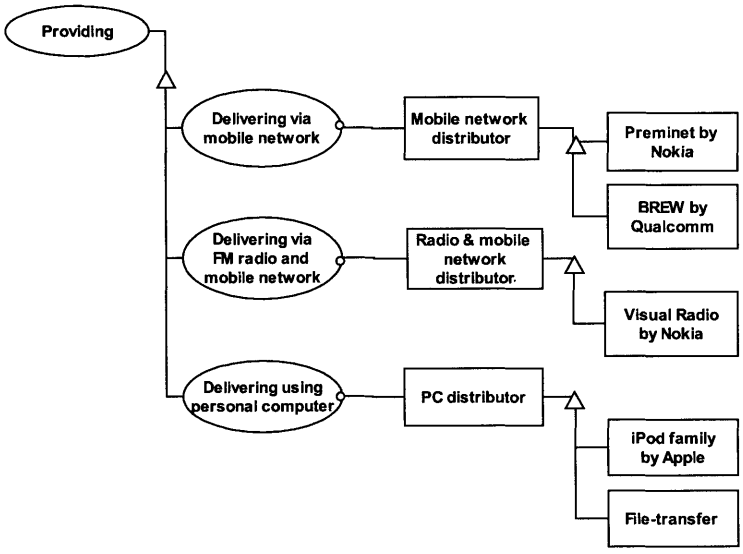

Figure 6-2: Mobile entertainment - concepts

a "Radio & mobile network distributor"-object for providing availability of mobile entertainment. An example of such object is the Visual Radio-system by Nokia [22].

In the third concept "Delivering using Personal computer"-process uses a "PC distributor"-object for providing availability of mobile entertainment. Examples of such systems are the iPod family by Apple [2] and any portable music and video player, here denoted by "File-transfer".

The figure 6-3 shows the specific systems forms and their attributes at level-0 abstraction. The concept "Delivering via Mobile network" requires a cellular network, a cellular phone, a delivery system, and a client in the cellular phone, with attributes shown the figure. The concept "Delivering via FM radio and Mobile network" requires in addition an FM radio and FM radio network, with attributes as shown. The client and delivery systems are different in the first two concepts. Finally, the concept "Delivering using Personal computer" requires a personal computer and handheld

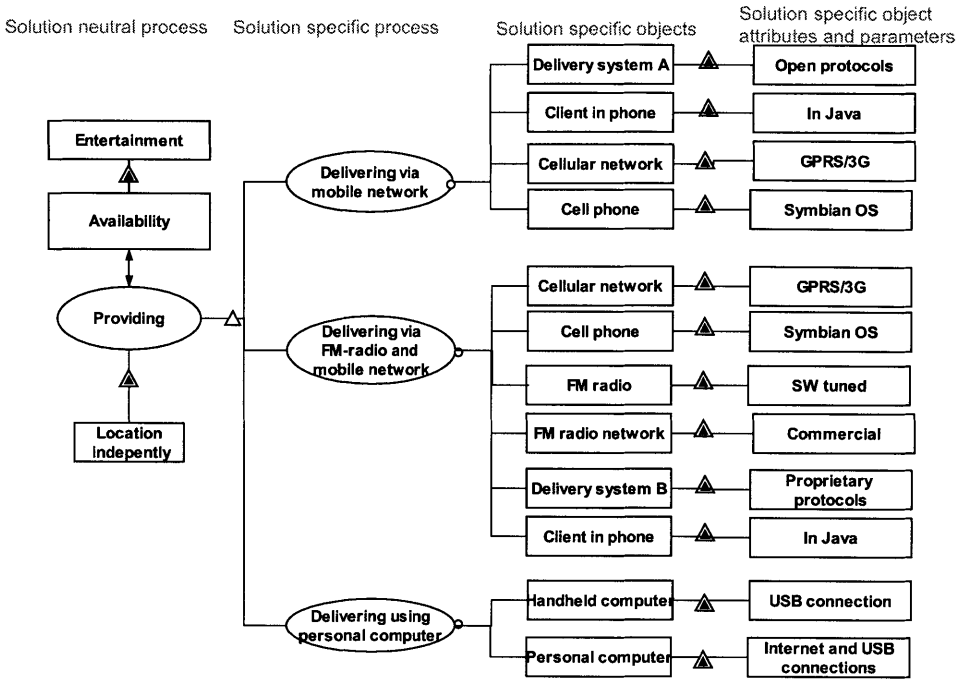

Figure 6-3: Mobile entertainment - level 0: specific system forms

computer. Note that all concepts require the objects of whole product systems shown in the figure 6-1.

6.2 Level 1 - operating sequences

The figures 6-4, 6-5, and 6-6 show the operating sequences of the concepts "Delivering via Mobile network" with Nokia Preminet-solution, "Delivering via FM radio and Mobile network" with Nokia VisualRadio-solution, and "Delivering using Personal computer" with Apple iPod, respectively. The operational sequences of the concepts are very similar. In each case, the system consists of a master entertainment database or databases, which have to be configured and connected to necessary networks. The mobile entertainment device must have client software installed. The normal operating sequence of primary processes is also very similar among the concepts. First

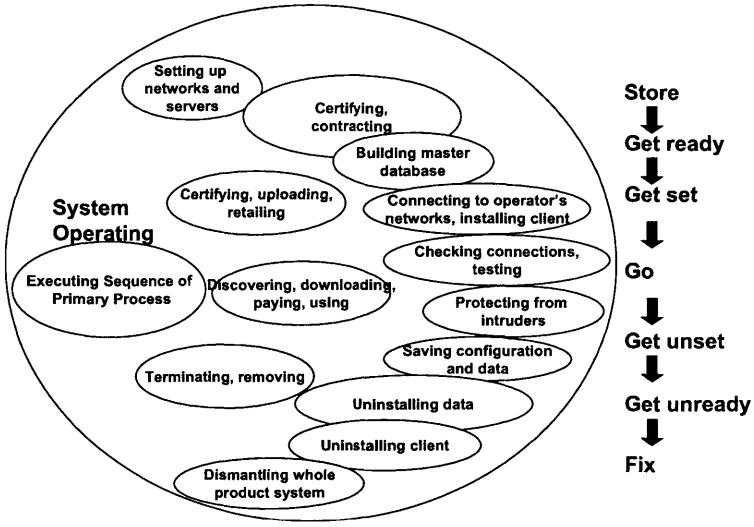

Figure 6-4: Mobile entertainment - level 1: operational sequence - Preminet

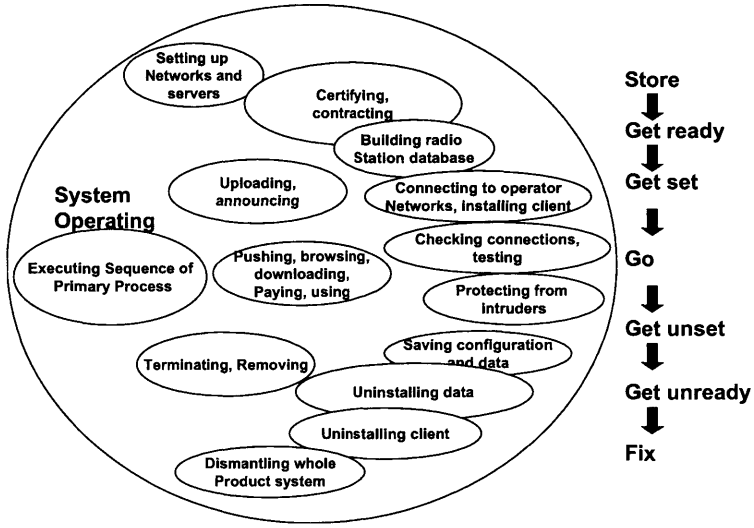

Figure 6-5: Mobile entertainment - level 1: operational sequence - Visual Radio

the user has to discover some entertainment of interest. Then, after some period of trying out the entertainment, the used pays and downloads the entertainment and

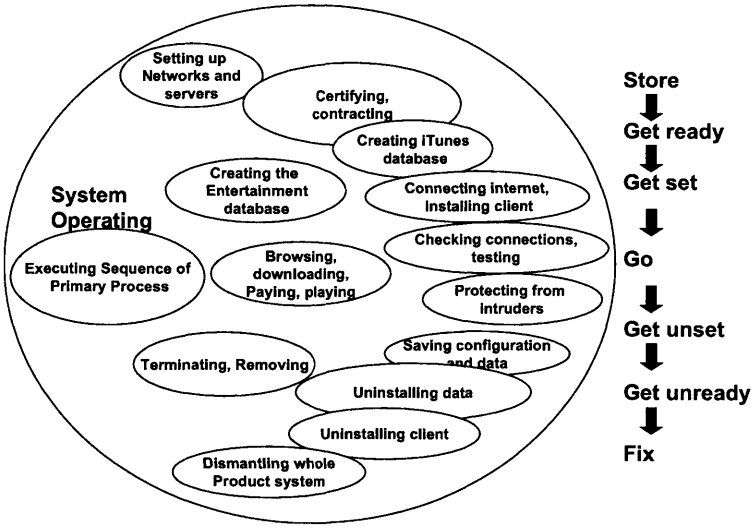

Figure 6-6: Mobile entertainment - level 1: operational sequence - iPod and iTunes

stores it in the device and possibly saves another copy of the data in some back-up device. If the entertainment is free, the user may directly download it. The user then plays the entertainment using his mobile entertainment device.

6.2.1 Level 1 architecture

As explained in 5.2.2, moving from the level 0 to the level 1 requires specializing intent, zooming processes, and decomposing objects. Since specializing the intent does not affect the complexity of the model, we will here constrain ourselves to just zooming in processes and decomposing objects.

In all cases, entertainment is the main operand and money is another value related object. The value related internal processes differ slightly as do the internal objects. In all architectures, there is an "information interfacing"-process, which transfers information among internal objects, external objects (from where the entertainment in each case ultimately comes), and necessary networks. Furthermore, all concepts have a billing system, which handles creation of bills and distributing money between

stake-holders, which in all cases are entertainment providers, networks operators, systems operators, and the main beneficiary:the mobile entertainment user.

The figure 6-7 shows the level 1 architecture of the Nokia Preminet solution. In the

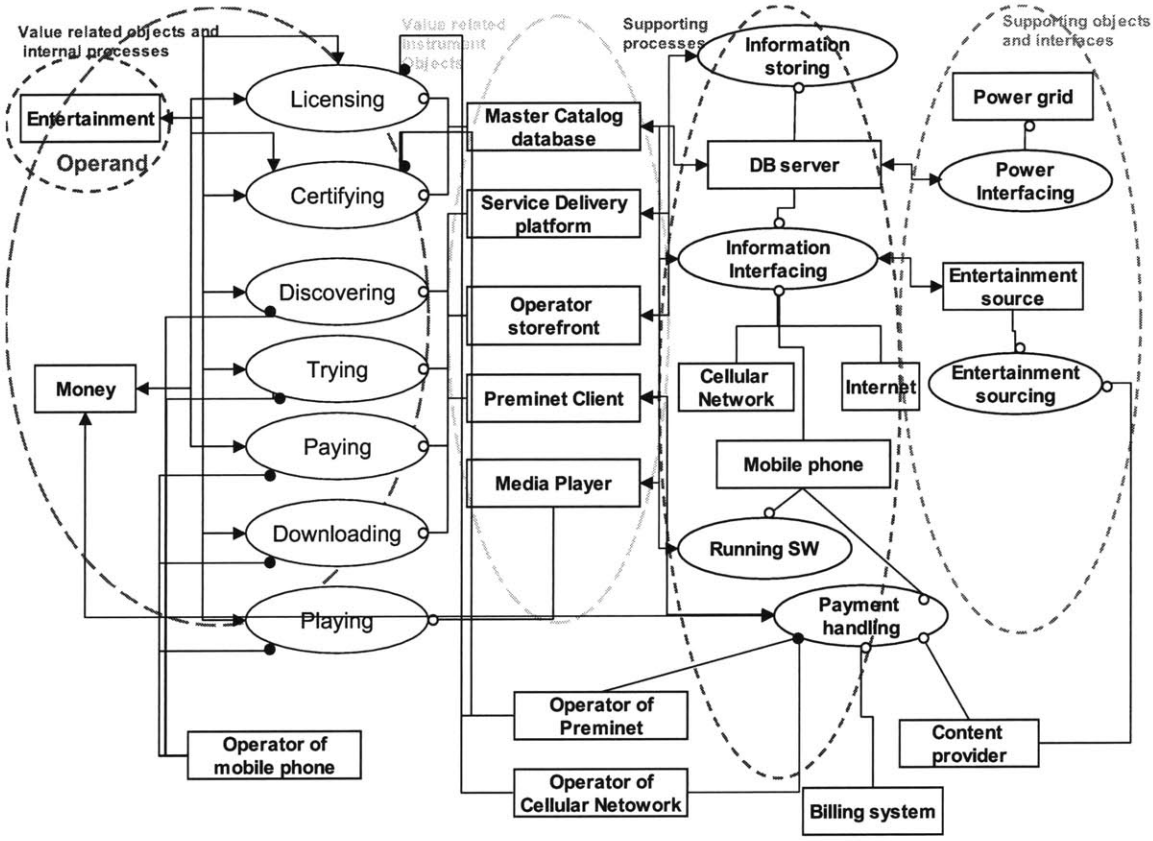

Figure 6-7: Mobile entertainment - level 1: Preminet

Preminet-concept by Nokia, the operator of Preminent, i.e. Nokia, makes contracts with content providers and licenses their entertainment (sounds, pictures, games). Nokia then certifies the entertainment, thus guaranteeing that the entertainment will not damage the ends users mobile entertainment devices (mobile phones). All entertainment is stored in the "master catalog database". The "service delivery platform" delivers the entertainment to "operator storefront", which is owned by the cellular network operator (e.g. Verizon). The "Preminet client"-software in the mobile phone

then contacts the "operator storefront" and gets a menu-structure. The mobile phone user uses the menu-structure in discovering new entertainment, which he then tries. If the entertainment pleases the user, he pays for the entertainment (if it carries a price), downloads the entertainment and plays it using the "media player"-software running in the "mobile phone".

The figure 6-8 shows the level 1 architecture of the Nokia Visual Radio solution. In the Visual Radio-concept by Nokia, there are two sources of entertainment: the

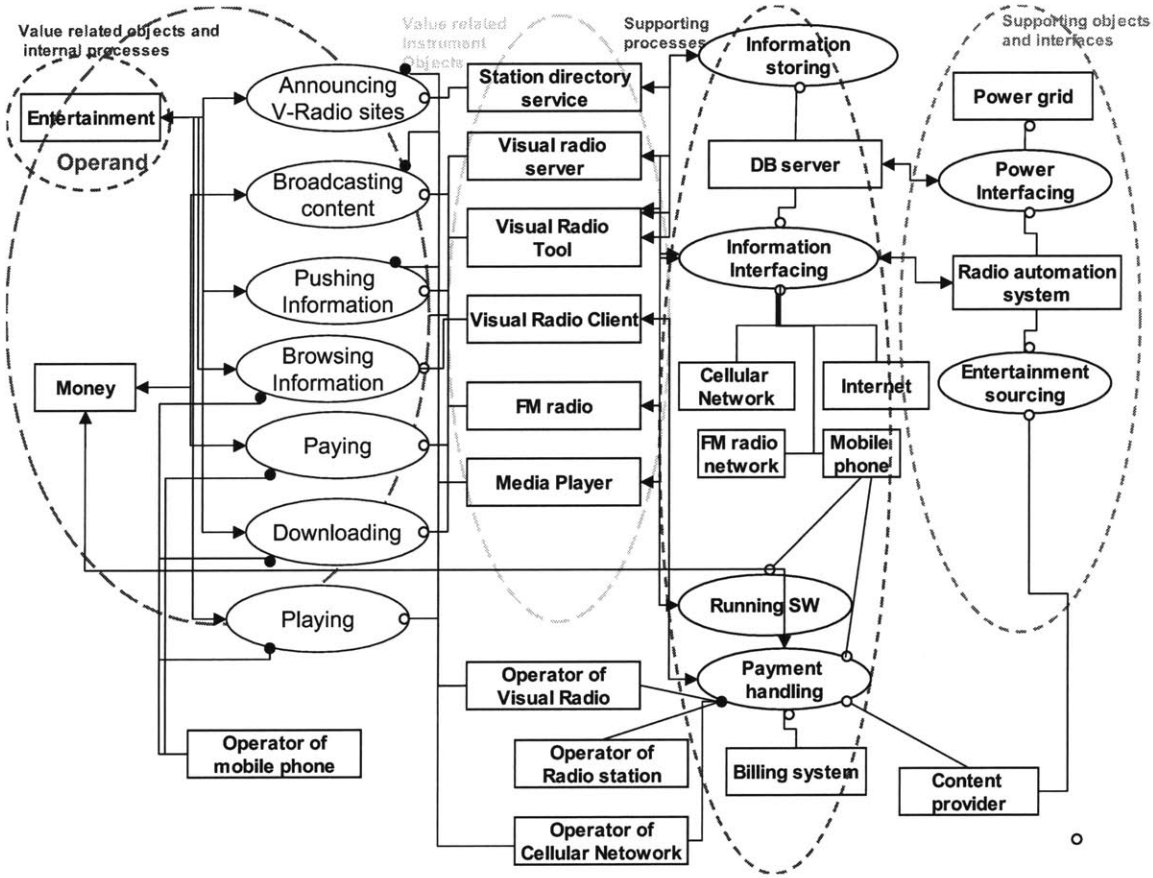

Figure 6-8: Mobile entertainment - level 1: Visual Radio

FM-radio network and the cellular network. The main idea of Visual Radio is to complement the FM-radio broadcast with extra information such as song lyrics, interactive voting, and to provide a possibility to buy and download music, pictures,

and games. In a way, Visual Radio is a more specific implementation of Preminet.

The Visual Radio-concept works as follows. The cellular network operator (e.g. Verizon) and the Visual Radio-system operator (Nokia) sign contracts with radio stations. The "station directory service" then announces the Visual Radio-service using the Visual Radio-client software in the mobile phone. The radio station operator uses the "visual radio tool" in creating visual content to he wants to add to the FM-broadcast. The "visual radio server" then sends the visual content using the cellular network to the mobile phones. The visual content contains timing data for synchronizing it to the FM-broadcast. The "visual radio client"-software then uses this timing sequence when it displays the visual content to the mobile phone user. In the model, "pushing information" and "browsing information" cover this phase of the operations. If the user likes the entertainment he gets, and wants for example to download the song he just heard, he pays for the entertainment, downloads it, and can then play it at will using the "media player". The billing system is more or less similar to that of Preminet's.

The figure 6-9 shows the level 1 architecture of Apple iPod-solution. The iPod-concept differs from both Preminet and Visual radio in that iPod does not have mobile connectivity. Instead, the user has to use his personal computer to run the iTunes-software. The iTunes-software connects to entertainment providers' (e.g. record companies') databases, and stores the entertainment in the "iTunes database". The user of iPod-mobile entertainment device then uses the iTunes-software to download entertainment to the iPod. The billing system transfers money from the user of iTunes/iPod to Apple and to the entertainment providers.

6.2.2 Level 1 complexity

The figure 6-10 shows some of the interface complexity multipliers of the Preminet-model at the level 1[1]. The interface complexity multipliers tell us that the total hidden complexity of interface information to and from the mobile phone is four times larger than that of interfacing information to and from the internet, and for other interfaces

[1]For discussion about complexity multipliers, see the chapter 7.

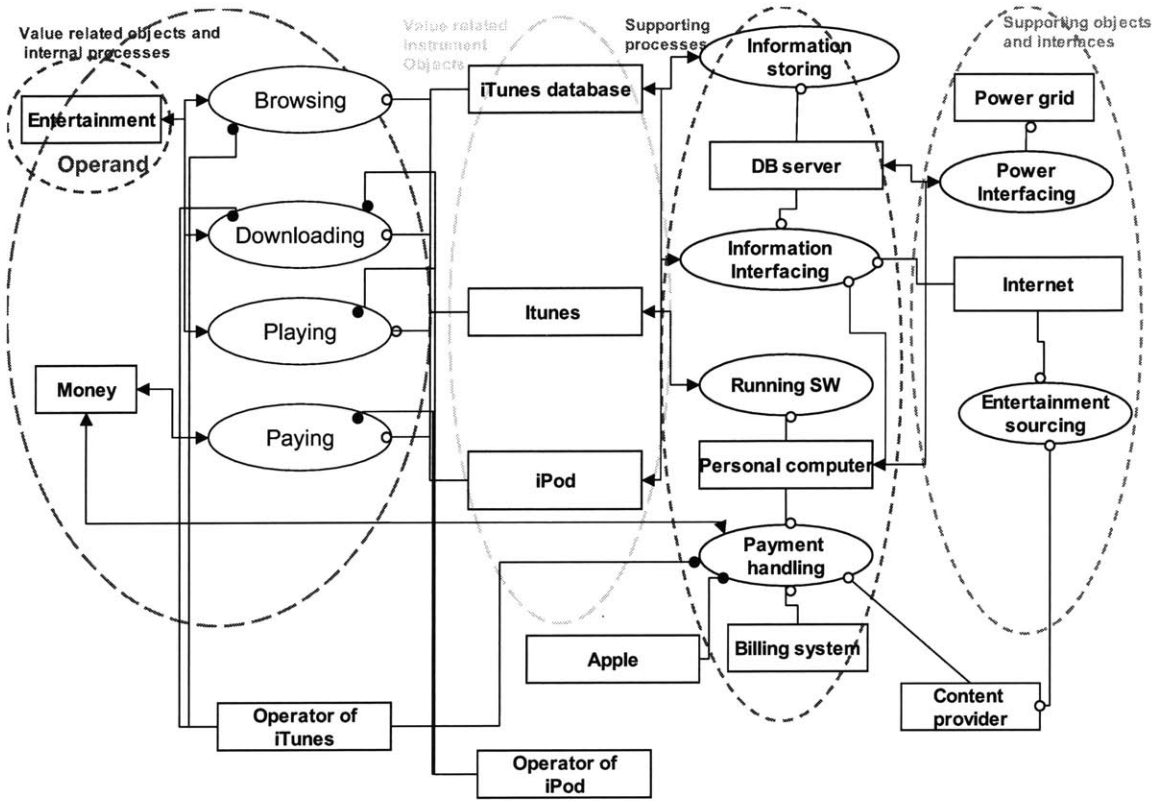

Figure 6-9: Mobile entertainment - level 1: iPod

respectively.

The figure 6-11 shows some of the interface complexity multipliers of the Visual Radio-model at the level 1. With Visual Radio, interfacing with the mobile phone is even more complex, as the mobile phone has to receive the information in time and with timing information necessary for synchronizing the FM-broadcast and the visual entertainment content. On the contrary, interfacing to and from the FM-radio network is considered to hide no extra information. This is because the mobile phone anyway has the FM-radio tuner.

The figure 6-12 shows some of the interface complexity multipliers of the iPod-model at the level 1. Information interfacing in the iPod-model with internet and the

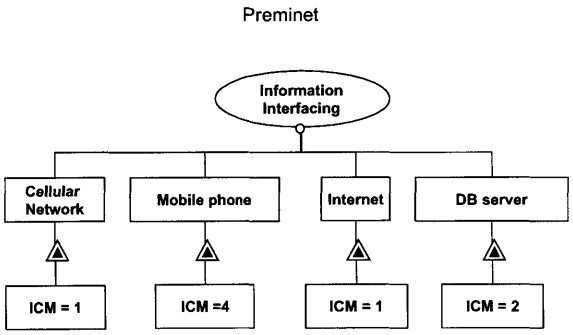

Figure 6-10: Mobile entertainment - level 1: Preminet - example of ICMs

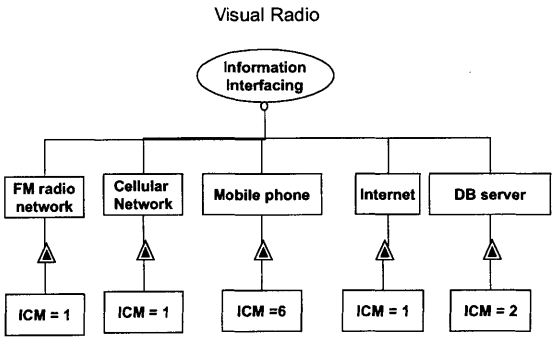

Figure 6-11: Mobile entertainment - level 1: VisualRadio - example of ICMs

personal computer does not hide any information. Thus the multipliers are 1. The multiplier for interfacing with the database is the same as that of Preminet's and Visual Radio's.

See also section 7.7 for discussion on the choice of ICMs in these three example cases.

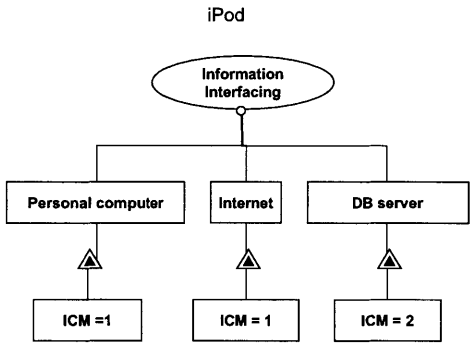

Figure 6-12: Mobile entertainment - level 1: iPod - example of ICMs

6.3 Level 2 - downloading and information interfacing

As explained in subsection 5.2.3, in order to get from the level 1 to the level 2 we need to apply the architecting process recursively. To do that completely for all three systems we consider in this thesis, would require too much space and would not help us in understanding the complexity measurement methods any better. For this reason, we will present the level 2 architecture only for the process of downloading entertainment and the client processes of the handheld mobile entertainment device. We select this process, because it is rather similar in all three systems.

The figure 6-13 shows the level 2 architecture for the process of downloading in the Preminet-system. In Preminet, the process of downloading zooms in processes of transporting, decoding, saving, and displaying progress. The Preminet-client decomposes in a http-client, a decoder, a file-manager, and a progress-meter. For the information interfacing process, the mobile phone decomposes in a cellular network receiver, a CPU, a memory, and a display. The CPU executes the software, the display displays the progress and the http-client receives information from the cellular

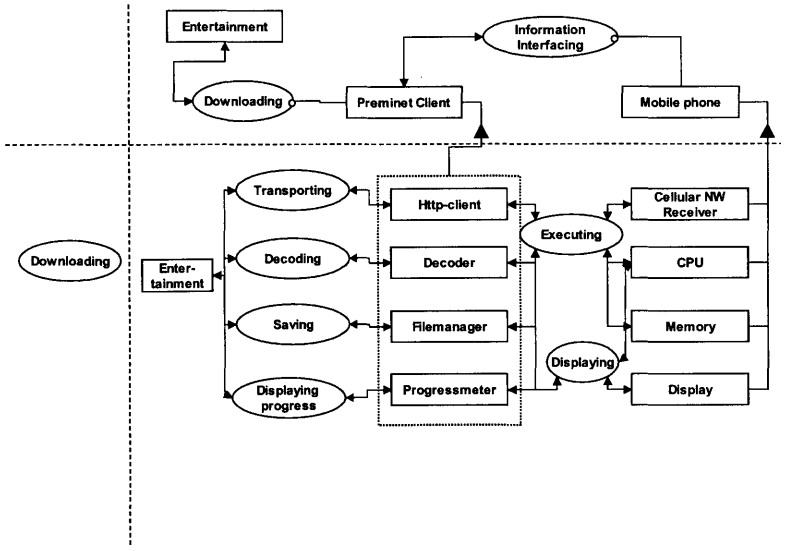

Figure 6-13: Mobile entertainment - level 2: Preminent / downloading

network receiver.

The figure 6-14 shows the level 2 architecture for the process of downloading in the Visual Radio-system. In Visual Radio, the process of downloading zooms in processes of transporting, synchronizing, decoding, saving, and displaying progress. The Visual Radio-client decomposes in a http-client, a synchronizer, a decoder, a file-manager, and an user interface. For the information interfacing process, the mobile phone decomposes in a FM-receiver, a cellular network receiver, a clock, a CPU, a memory, and a display. The CPU executes the software, the clock times synchronization, the display displays the user interface. The FM-receiver and the cellular network receiver transmit the information to the http-client and synchronizer.

The figure 6-15 shows the level 2 architecture for the process of downloading in the iPod-system. In iPod, the level 2 architecture is very close to that of Preminet. The only differences are replacing the http-client with and USB-interface driver, and cellular network with USB-interface.

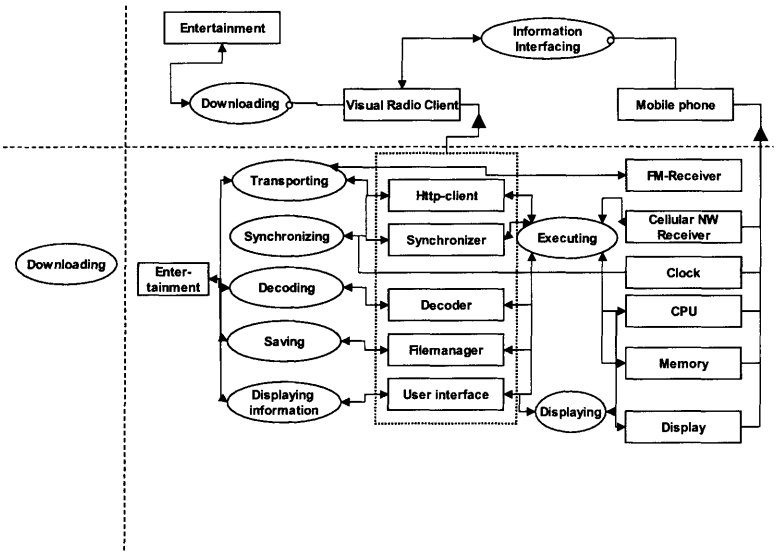

Figure 6-14: Mobile entertainment - level 2: VisualRadio / downloading

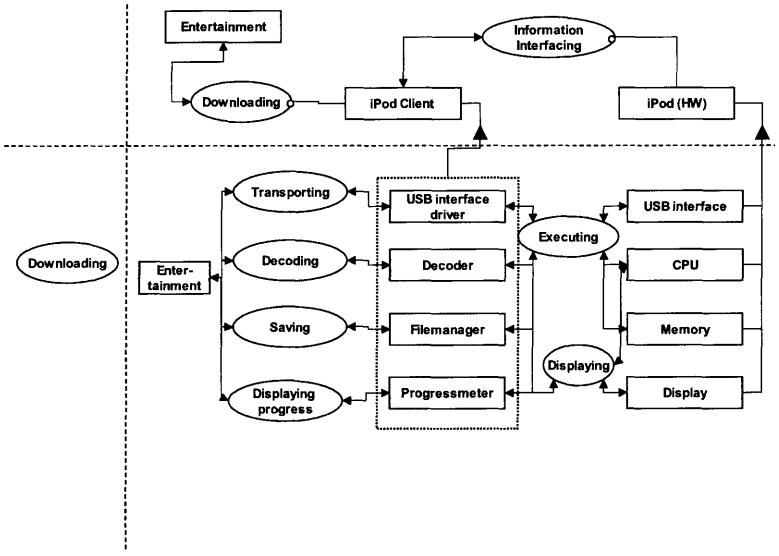

Figure 6-15: Mobile entertainment - level 2: iPod / downloading

Chapter 7

System architecture complexity measures

This chapter introduces systems architecture complexity measures. The measures are based both on the existing complexity measures, which we discussed in the chapter 4, and on the example system architecture models we developed in the chapter 6. After introducing the measures, we then apply them to the example systems. Finally, we discuss the usability and applicability of the measures and also ways of automatizing the measures.

7.1 Preliminary notes

Using of the definition of complexity 3.5.1 is not possible without proxy-methods. We cannot, in practice, design the Turing machines. Instead, we have to use some indirect means, which correlate with the length of the Turing-program, if such would be designed.

It is to be hard to say anything interesting about complexity at level 0. The level 0 is mainly the system problem statement. It also gives the whole product system elements, and thus also hints at the external interfaces, which will be explored in detail at level 1 and lower. However, since the level 0 contains information about specific processes, specific objects, and their parameters, simply counting the number

of processes and objects per different system concepts gives us a rough idea of the complexity of the model[1]. Note, however, that the level 0 model does not show the different concepts and so cannot be used in discussing the complexity of the concepts.

In level 1, we start to see the actual processes, objects, and interfaces between them, and also the interfaces between the system and the whole product system and the context of the system.

In level 2, we will zoom in processes and decompose objects of level 1. This will create more interfaces, which were hidden (abstracted away) in levels 0 and 1. We will also see more details of the external interfaces: zoomed in interface processes and decomposed interfacing objects, which preform the processes. The same methods of zooming in and decomposing will continue to reveal more details when moving from level 2 to level 3 and so on.

The above discussion indicates, that if we were able to, during the systems architecting process, zoom in and decompose until we reach the elementary parts of the system, we would be able to determine the complexity of the system very easily and without ambiguity. Any proxy-method would be usable, for example variants of measures introduced in the chapter 4. In practice, we are not able to do that. Instead, we are always working at some level of abstraction. The higher the level of abstraction, the higher the amount of information we are not aware of when modeling.

In a model, at a abstraction level, all processes and objects are of equally important. If they were not, the less important ones, or those which belong to lower level of abstraction, would not be included in the model. For these reason, just counting the number of processes and objects, or types of them, gives us an understanding of how complex the model is. But the really measure the complexity of a model, we have to take a look at the interfaces - the complexity of a model must somehow be a function of the interfaces.

For this reason, we must somehow estimate, at any level of abstraction, complexity of each interface. In other words, we must give each interface a parameter, which tells the *complexity multiplier of the interface* as shown in figure 7-1 The multiplier tells

[1]this assumes than all processes and objects contribute equal amount of complexity

how much information is hidden in the interface relative to other interfaces in the model. See the figures 6-10, 6-11, and 6-12 for examples of the complexity multipliers.

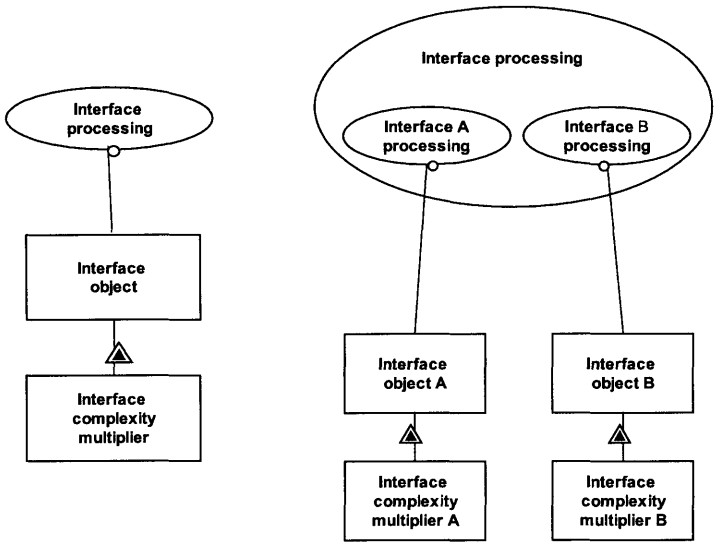

Figure 7-1: Complexity multiplier of the interface

The complexity multiplier of an interface reflects the amount of hidden, or not yet specified, information related to the interface. In principle, there is no single way of determining or deciding these multipliers. Fortunately, the exact numerical values of multipliers do not matter: as long as the choice of multipliers within models of a systems are consistent, the complexity measures will be consistent. It is the duty of the systems architect to see to that the multipliers are consistent. The meaningfulness of the results depend on his professional skills and experience in architecting similar systems.

7.2 The interface types

Before discussing guidelines of assigning interface complexity multipliers to interfaces, we must study the types and properties of interfaces. According to Crawley [6] there are four types of interfaces

Logical relational The relation between two objects may be any of the four relations expressed in OPM (see the chapter 5): specialization, decomposition, characterization, or instantiation. In addition one object can simple be a renaming or coding of another.

Topological Two objects can either be adjacent (touch each other), overlap each other, be separate, or one can surround the other (or contain the other).

Implementation or structural The structural interfaces tell how the systems is physically put together: how the systems maintains the topological interfaces, and what kind of exchanges there are between objects. We elaborate on this below.

Operation The operational interfaces tell, how the systems is operated, what the operator does in order to maintain value delivery of the system.

Of these, according to Crawley, *suppressed implementation and/or operational structural links are the interfaces of interest*, which is in line with our preliminary notes above. The reasons are easy to see.

First, the logical relational interfaces do not hide anything, on the contrary, they expose the architecture, and whatever their contribution to total complexity is, it is easy to calculate.

Second, topological relations do not by themselves tell anything how they are supposed to be maintained in the system. If two object are adjacent to each other, we need to know what is the implementation or structural interrelation holding them at the position.

Third, the higher the abstraction level, the less detailed information we have on implementation of interfaces and the more information we are suppressing. Thus,

we have to compensate for this hidden information with an interface complexity multiplier.

Fourth, similar reasoning as with implementation interrelationship applies to operation interrelationships: at high abstraction level we do not know the details of operations, the amount and type information, matter, or energy the operator has to apply to operate the system. Thus, we have to compensate with an interface complexity multiplier.

7.3 Types of structural interfaces

There are three classes of structural interfaces: matter, energy, and information [6]. Every structural interfaces falls into one of these classes.

The material interfaces are those, in which either matter itself or material related forces, such as momentum or pressure, is exchanged between two objects.

The energy interfaces are those, in which energy in some form is the dominating type of exchange between two objects. Energy can be in the form of heat, radiation, or (mechanical) work.

The information interfaces are those, in which signals or thought is exchanged between two objects. Signals always consist of data, which is exchanged using some coding. The coding is expressed in material (as in passing "kanban"-cards between work stations in lean manufacturing systems) or in energy (as in passing digital data between computers). In systems, such as organizations, with many human operators, stake-holders, and beneficiaries, the human objects also exchange ideas or thoughts. Also ideas and thoughts are always encoded in some material or energy form.

All structural interfaces involve an exchange of mass and thus energy. The idea of classifying them into matter, energy, and informations is to put emphasis on the purpose, or function, of the interchange. In models of high abstraction level, it is not important to know the actual implementation of the interchange, but it is very important to know the purpose, or function, of the interchange. For example, to continue using kanban-cards as an example, by knowing that the purpose of the

interface is to exchange a single bit of information ("produce the next item"), we know, that no matter how we choose to implement the interface, it will not require much information to model (and thus the complexity multiplier is small, most likely 1).

7.4 Determining the interface complexity multipliers

Since we are using the interface complexity multiplier as a proxy for estimating the complexity of a model, we must establish some reasoning for selecting the multipliers. Since our definition of complexity 3.5.1 is based on the length of Turing-programs producing the model, we will give reasons which will make the program for modeling an interface inevitably longer.

The following properties are important, when deciding the interface complexity multiplier for an interface

Distance The longer the distance the interface has to span, the more information we need to specify the interface. This is easy to see: to transport material, energy, or even information any longer distance requires pumps, repeaters, and similar. Specifying them requires information, which increases the length of the program.

Volume of interchange The larger the amount of volume the interface has to exchange, the more information we need to specify the interface. Similarly, if an interface has to exchange a very small amount of material, energy, or information, specifying it requires more information.

Quality requirements The higher the quality requirements for the interface, the more information is needed to specify the interfaces. For example, if the interface transfers information, the smaller the allowed error-ratio in transfer, the more information is needed to specify the interface.

Reversibility The more reversibility the interface has to allow, the more information is needed. For example, if the interface joins two physical object (e.g pieces of metal), specifying that they are welded together requires less information than they are connected with a set of screws.

Reliability requirements The more reliable the interface has to be, the more information is needed to specify it. For example, if an information interface has to tolerate a loss of a carrier, we will need to specify some kind of redundancy, which requires information.

Tolerances The more exact the interface has to be, the more information is needed to specify it. This is especially true with physical interfaces (e.g. positioning of objects with respect to each other), but also is of concern with information interfaces: information is usually coded in voltage levels and their changes.

Physical conditions The ambient physical conditions have an effect on the amount of information needed to specify an interface. For example, if the temperature at the interface is either very low or very high, specifying the physical implementation of the interface will require more information than if the temperature is closer to nominal (see below for discussion on the concept of nominal).

Level of reuse If the interface is novel, or similar type of interfaces has never before been designed, specification of the interface will require more information than if than interface is well-know and has been implemented several times. In essence, a completely reused interface (the process and the object) will be a part of the system.

Concurrency The more concurrent flows of material, energy, or information the interface has to facilitate, the higher amount of information needed to specify the interface.

Standardization If the interface is based on a standard, the amount of information needed to specify the interface is smaller than if there is no standard. In fact, a standard is a special case of reuse.

Vibration, noise, electric conductance These are commonly encountered special cases of physical conditions and quality requirements.

It is notable, that with many properties (e.g. distance, volume, noise) both requirements of high level/amount and low level/amount increase the information needed to specify the interface. This is because, there seems to be a "nominal" or customary level of requirements, which the designers are able to design. Radical deviance from the nominal value will increase the design effort, which will mean that a larger amount of information is needed, which increases the complexity of the model. Thus, in the higher abstraction levels of architecting models, the properties of designers and specific historical time (technological maturity of the field of design in question) do affect the amount information required and thus complexity. However, if we were able to specify the system down to its parts, the dependence on the designer and maturity would vanish.

As an example, consider the effect of spanning distance on the amount of information needed to specify the interface (or, the amount of information hidden by the interface in higher level of abstraction). In case of material interface, such as an oil pipe in a design of a oil transportation network, the longer the distance between the connected object, the more pumping stations will be needed, and the more varied the terrain to cross will be. All this is abstracted away by just specifying the interface as an oil pipe, and must be compensated by suitable interface complexity multiplier. Furthermore, the larger the amount of oil to be transferred, the more demanding the design task, which must be reflected in the multiplier.

As another example, consider system architecting an organization. Fundamental parts of an organization are humans, but at higher level of abstractions we will use "departments", "teams", or "product lines" as objects of interest. These objects will have some interfaces among them. For example, two teams may have an interface consisting of a "collaborating"-process and a "collaborator"-object. The final instantiation of this interface depends on the (physical) distance between the teams. The further away they are from each other, the more information we need for specifying the "collaborating"-interface. Note also, that quality and performance requirements

affect the amount of information needed for specifying. Specifying non-realtime, non-persistent collaboration requires less information than specifying real-time, persistent (in the sense of producing and storing a record) collaboration.

Or consider space systems. The external physical system boundary of a space system, for example a satellite or a manned space module, requires much more information to specify than similar terrestrial system would require. This is simply because in space we have to worry about and design for greater temperature differences, maintaining the internal pressure while being surrounded by vacuum, and radiation protection. Thus, if a system model contains both a terrestrial external system boundary and a space one, the space one will require a higher interface complexity multiplier. The exact choice of number is for the systems architect to decide.

The default interface complexity multiplier is 1. In practice, it is prudent to keep the maximum at 10. If, at a level of abstraction, an interface hides more than 10 times more information than another interface, it is necessary to specify the former interface in more detail, which means that it is necessary to zoom in the interfacing process and decompose the interface object.

7.4.1 Inheriting multipliers among levels

In systems architecture models, the more detailed levels of the model inherit the interfaces from the less detailed. This is shown in figure 5-13. For this reason, the architect has to decide how the interface complexity multipliers get inherited between levels as shown in figure 7-1.

There are several alternative ways to specify the inherited multipliers. Since the multipliers represent the amount of hidden, or abstracted information, the architect can simply use his judgement and assign new multipliers. Note that the multipliers do not have to be consistent, or comparable, between levels of abstraction. It is sufficient for them to be consistent and comparable within a level of abstraction.

Another, procedural method, is to distribute the original multiplier evenly among the new interfaces process-object-pairs. For example, if the original multiplier was 6, the new multipliers A and B, in figure 7-1, would become 3. The problem of this idea

is that the resulting multipliers quickly become 1, which is the default and minimum value.

7.5 The complexity measures

System architecture models contain a lot of information of different types: objects, processes, many different types of relationships between them, states, and object-process-pairs called interfaces. In addition, interfaces have their interface complexity multipliers.

Trying to reduce all this information in a single number, which would meaningfully represent the complexity of the model, is a highly reductionistic attempt. Several authors have tried to device ways of accomplishing similar goals (although their models have contained less information) and have come up with rather arbitrary formulas. For example, having two measures (e.g. number of parts and number of interfaces) an easy way is to take the square root of the sum of squares of measures. This combined measure indicates the distance from the origin (of simplicity, apparently). Another way is Meyer's (see section 4.8) of multiplying three measures and taking a cubic root of the result.

Whether or not such combined (or reduced) measures are sensible or not, we must first define the elementary measures. The next list contains the elementary measures, their justifications, and their usages:

number of distinct types of things The greater the number of types of things (objects, processes, and states) a model has, the longer program is needed to produce the model. For example, the level 1 model of iPod (figure 6-9) has 10 distinct types of processes and 14 distinct types of objects.

sum of number of things of each distinct type The more instances there are of a certain type of thing, the longer program is needed to produce the model. Note that the length of the program does not depend linearly on the number of instances. Instead, what matter is whether there is one or many instances.

Thus, if there is one instance, we count it as one, if there are several, we count them as two instances. In effect, we claim that the program to produce a model with many instances, is twice longer than than one producing a model with only one instance.

number of processes affecting an object The more processes affect a given object, the longer program is needed to produce the model. This is similar to fan-in and fan-out measures (see section 4.4). Calculating the minimum, maximum, average values of this measure is useful. If the minimum is zero, the model has a dangling object. If the maximum is very high (or much higher than the average), the model may have uneven distribution of complexity.

number of objects being affected by a process The more objects a processes affects, the longer program is needed to produce the model. This is very similar to the previous elementary measure.

number of operands per process The more operands a process has, the longer program is needed to produce the model. Again, calculating (minimum), maximum and average values is useful, since high average and high maximum values may indicate an unevenly or highly complex system.

number of interfaces weighted with their ICMs The more interfaces the model has, the longer program is needed to produce the model. As discussed above, interfaces tend to hide a lot of information, which we compensate by using the interface complexity multipliers (ICMs). For example, the "information interfacing" interface in the Preminet-model (see figure 6-10). The interface consists of one process and four objects. The cardinality of each object is one (1). Thus the weighted sum of interfaces is the sum of ICMs, which in this case is 8. Should there be two database servers, the sum would be 10. See section 7.7 for discussion on the choice of ICMs in this case.

This set of basic measures allows us to compare any number of models of equivalent systems in terms of complexity. With any real life systems and their models, the

results are very likely to be contradictory. A model is more complex than another according to one measure, but less complex according to another measure. For this reason, lumping measures together with any formula is not going to be meaningful in all situations. Thus, it is better to leave prioritization of measures to the architect of the systems. He is, using his experience and skill, best able to decide, which kind of complexity he and the rest of the product development organization should worry about.

7.6 Automatizing the measurements

Calculating the complexity measures defined above manually from the models (such as those in figures 6-10, 6-11, 6-10, and 6-11) is very error-prone and cumbersome. However, since the model are developed using computers, automatizing the calculation is easy. The only necessary thing is to know the structure of the model: it is a graph, with objects/processes/states as vertices and relationships as edges. All of this can easily be represented by a matrix, in which both rows and columns contain denote object, processes, states, and cells contain information about cardinalities, interface complexity multipliers, and other necessary things. It is not part of this thesis to develop such a matrix - it is a task more suitable for anyone, who is going to implement an OPM-based design tool.

7.7 Complexity measures of example systems

The figure 7-2 shows the level-1 complexity measures for the three mobile entertainment systems. All other measures but the last one *Sum ICMs of the "information interfacing interface"* are easy to determine, as they depend only on the topology of the model. For the last one, we need to determine the ICMs. See figures 6-10, 6-11, and 6-12 for the ICMs.

In all systems, the interfaces to FM radio, cellular network, internet, and the personal computer are already existing and even standardized. Thus, they are, in

essence, parts in the system. Thus, their ICMs are one (1). The interfaces to database servers are a bit more complicated. The interface is very much standardized, but there are some unknown details, which are due to the specific structure of the database. Thus, we estimate the ICM to be two (2).

For Preminet, the interface to the cell phone has the ICM of four (4). This is because the mobile phone platform (the Symbian operating system) is not very mature yet, which makes the required specification longer. Furthermore, the quality requirements of the interface are rather high due to business reasons. If the software does not work, the users will not use it and the whole Preminet-business fails.

For Visual Radio, the interface to the mobile phone has ICM of six (6). It requires all the same information as the mobile phone interface in Preminet, but in addition Visual Radio has to synchronize the FM radio with the internet in order to show the text at the right time. This requires specification of tolerances, thus giving an ICM of six.

The figure 7-3 shows the level-2 complexity measures for the three mobile entertainment systems, for the process of information interfacing. In both level-1 and level-2, iPod comes out as the simplest model for mobile entertainment, as would be expected since it does not provide for mobile downloading capability. Also, Visual Radio comes out as the most complex alternative, as would be expected since it requires two radio networks and receivers, and in addition an ability to synchronize downloading between the networks.

This simple exercise seems to prove, that the measurements are sensible. They produce results, which are not too surprising. The measurements themselves do satisfy the requirements for sensible measurements stated in chapter 4.

This reasoning shows clearly, that determining the values of ICMs is based on the judgement of the systems architect. He must use his experience and understanding of similar systems. Determining the ICMs by discussing with other architects and designers is a good practice.

Measure	Preminet	Visual Radio	iPod
Object types	18	22	14
Process types	13	13	10
State types	0	0	0
Sum of things	31	35	24
Processes affecting objects (min, max, avg)	0 6 1.2	0 5 1	0 2 1.3
Objects affected by a process (min, max, avg)	0 4 1.7	0 4 1.9	0 2 1.2
Operands per process (min, max, avg)	1 5 2.4	1 5 2.3	1 5 2.4
Sum ICMs of the "information interfacing" interface	8	11	4

Figure 7-2: Mobile entertainment - level 1: complexity measurements

Measure	Preminet	Visual Radio	iPod
Object types	9	12	9
Process types	6	7	6
State types	0	0	0
Sum of things	15	19	15
Processes affecting objects (min, max, avg)	1 4 2.2	1 4 2	1 4 2
Objects affected by a process (min, max, avg)	2 7 2.7	2 9 3.7	2 7 2.8
Operands per process (min, max, avg)	N/A	N/A	N/A
Sum of ifaces weight by ICMs (of "Information interfacing)	N/A	N/A	N/A

Figure 7-3: Mobile entertainment - level 2: complexity measurements

Chapter 8

Conclusions

According to Rechtin [27], the heuristic "Regarding intuition: trust but verify" is a good guideline for a systems architect. In this thesis, we have developed methods for verifying architect's intuition on complexity of systems architecture models. We introduced a new concept of interface complexity multiplier and defined a set of complexity metrics. We also showed how to use them in an example case study. So, what can we conclude?

In his book "Blink" [11], Malcolm Gladwell describes how humans are very capable pattern recognizers. For example, a trained sculpture expert can determine whether a sculpture is a fake or real just by glancing briefly at the sculpture. Similarly, experienced soldiers are able to make right decisions during a battle very fast and with rather limited amount of information. In all these cases, the key prerequisite for fast pattern recognition are training and experience. Similarly, experienced systems architects are able to grasp the complexity of an architecture model in a blink of an eye, very fast. They can even get a physical feeling of discomfort, when presented with a unnecessarily complex model.

The real users of complexity measures are then in-experienced architects, who can use the measures in training their intuition, and other people working on the design or having a stake in the product development. Such users include managers, customers (beneficiaries and their agent), quality assurance personnel, and system integrators and commissioners. They can use the measurement in order to check the

work amount estimates done by the architects and also use the measures as basis of their own planning - or even as basis for telling the architects that the models are too complex for being basis of their work.

The fact that architects themselves do not need complexity metrics may well be the main reason, why the literature on complexity measurements of system architecture models is so small. The systems architects are able to intuitively measure, or grasp, whether a model is excessively complex or not. This is true, if the number of models is small enough to be easily glanced at. If the number of models is very large, as is often the case with automatically generated models, the complexity measures are likely to be useful in pruning some of the models from more detailed consideration.

The use of interface complexity multipliers may seem to render the measurements subjective and thus violate the requirements we stated for valid measurements. If the architect decides the multipliers, does that not make the whole measurement effort subjective? The answer is no as long as the architect follows some pre-defined, and objectively justifiable, set of rules for deciding the multipliers.

Finally, this thesis shows, that measuring complexity of the system architecture models is quite tedious, and cannot be done using the existing, OPM, modeling language without extending the language and without doing extra work on the models. For this reason, complexity measurement should be allocated enough time and resources in the project plans.

8.1 Further research

In order to make the complexity measures really usable and practical some further research appears essential. The first and most important is to develop accurate guidelines for determining the interface complexity multipliers. Doing so will require cross-disciplinary research and cooperation, for one researcher cannot know all necessary details of all kinds of interfaces. Once the rules for determining the ICMs are known, the next thing to do is to implement the measurements in some architecture design frameworks. This may require some changes in the way the frameworks function. At

least, the ways of inheriting ICMs between abstraction levels need to be determined.

The most important thing is, however, to apply the measurements to several architecture models in several different fields of design. Such application, which is beyond the scope of this thesis, will both reveal weaknesses of this thesis and show how to proceed further in the field of complexity measurement.

Bibliography

[1] Mohsen N AlSharif. *Assessing the complexity of software architecture.* PhD thesis, Florida Institute of Technology, May 2005.

[2] Apple. Apple ifamily at http://www.apple.com/ipod/.

[3] Rudolf Carnap and Yehoshua Bar-Hillel. An outline of a theory of semantic information. Technical Report 247, MIT Reseach Laboratory of Electronics, 1952.

[4] Gregory K Chaitin. On the lenght of programs for computing binary sequences. *Journal of the ACM*, 13:547–569, 1966.

[5] Clayton Christensen. *The Innovator's Dilemma.* Harper Business Essential, 2003.

[6] Edward Crawley. *System architecture - course notes.* MIT, 2005.

[7] Ebru Dincel, Nenad Medvidovic, and Andr van der Hoek. Measuring product line architectures. *Lecture Notes in Computer Science*, 2290:346, 2002.

[8] Dov Dori. *Object-Process Methdology.* Springer, 2002.

[9] Bruce Edmonds. What is complexity? - the philosophy of complexity per se with application to some examples in evolution. *XXX*, 1999.

[10] Gottlob Frege. *Grundgesetze der Arithmetik.* Verlag von H. Pole, Jena, 1893.

[11] Malcolm Gladwell. *Blink : The Power of Thinking Without Thinking.* Little, Brown, January 2005.

[12] Rebecca Henderson. Architectural innovation: The reconfiguration of existing product technologies and the failure of existing firm. *Administrative Science Quartely*, 35:9–30, 1990.

[13] Roni Horowitz and Oded Maimon. Creative design methodology and the sit method. In *1997 ASME Design Engineering Technical Conference*, pages 1–10. American society of mechanical engineers, 1997.

[14] Andrei N. Kolmogorov. Three approaches for defining the concept of information quantity. *Problems of Information Transmission*, 1:1–7, 1965.

[15] J. Lankford. Measuring system and software architecture complexity. *Aerospace Conference, 2003. Proceedings*, 8(8-15, 2003):3849–3857, March 2003.

[16] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, 1989.

[17] Steve McConnell. *Rapid Development.* Microsoft Press, July 1996.

[18] Mark Meyer and Alvin Lehnerd. *The Power of Product Platforms.* Free Press, 1997.

[19] Gerrit Muller. *CAFCR: A Multi-view Metdod for Embedded Systems Architecting.* PhD thesis, Techische Universiteit Delft, July 2005.

[20] Gerrit Muller. *System Architecting.* ESI, 2005.

[21] Nokia. Nokia preminet at http://www.nokia.com/preminet.

[22] Nokia. Nokia visual radio at http://www.visualradio.com/.

[23] Qualcomm. Qualcomm brew at http://brew.qualcomm.com/brew/en/.

[24] W.V. O Quine. *Mathematical Logic.* Norton, New York, 1940.

[25] M. Burth R. Kazman. Assessing architectural complexity. *2nd Euromicro Conference on Software Maintenance and Reengineering*, 00:104, 1998.

[26] Panu Raatikainen. Complexity and information. *Reports from the Department of Philosophy*, 2, 1998.

[27] Eberhardt Rechtin and Mark W. Maier. *The art of systems architecting.* CRC, 2002.

[28] Claude E. Shannon. A mathematical theory of communication. *Bell System Techinical Journal*, 27:379–423, 623–656, 1948.

[29] Willard Simmons, Benjamin Koo, and Edward Crawley. Architecture generation for moon-mard exploraton using an executable meta-language. volume AIAA-2005-6726. American Institute of Aeronautics and Astronautics, American Institute of Aeronautics and Astronautics, 2005.

[30] Herbert A. Simon. *The Sciences of the Artificial.* MIT press, 1996.

[31] Ray J Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:1–22,224–254, 1964.

[32] Nam P. Suh. A theory of complexity, periodicity and the design axioms,. *Research in Engineering Design*, Volume 11(Issue 2,):Pages 116 – 132, Aug 1999.

[33] James Utterback. *Mastering the Dynamics of Innovation.* Harvard Business School Press, 1996.

[34] E Weyuker. The evaluation of software complexity measures. *IEEE transactions on Software Engineering*, 14:1357–1365, 1988.

[35] Jianjun Zhao. On assessing the complexity of software architectures. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 163–166, New York, NY, USA, 1998. ACM Press.