The impact of architectural decisions on quality attributes of enterprise information systems: a survey of the design space

Wiebe Hordijk
wiebe.hordijk@ordina.nl

Dennis Krukkert
d.krukkert@ewi.utwente.nl

Roel Wieringa
R.J.Wieringa@ewi.utwente.nl

November 26, 2004

1 Introduction

1.1 Problem

Design of enterprise information systems is a problem-solving activity. A system architect, designer and programmer make numerous decisions about the structure and behaviour of the system on various levels. These decisions define the quality of the system under design (SuD) in all its aspects. An example of an application-level decision is whether to structure the domain logic according to a *domain model*, a *table module* or a *transaction script*. We want to investigate the effects of such decisions on quality attributes of software. This will allow us to make better software and to predict the quality of software before it is built.

In this research, we try to empirically validate or reject hypotheses like: "In the majority of systems above 500 function points, systems with a *domain model* have better changeability than systems with a *table module*." If the validity of such hypotheses depend on the context of the system, we want to know in which cases the hypotheses hold and in which they do not.

To be able to do such empirical research, we first need to develop a theoretical framework that defines the research context. This framework defines concepts like *design problems*, *options* and *quality indicators*. The design problems and options define choices a systems designer makes when designing a system. The quality indicators define if an option is better than another option: the notion of "better" is operationalized by means of quality indicators. The three together form the design space. Other design space models are discussed in section 4. The goal of this paper is to present a design space as a framework for empirical research.

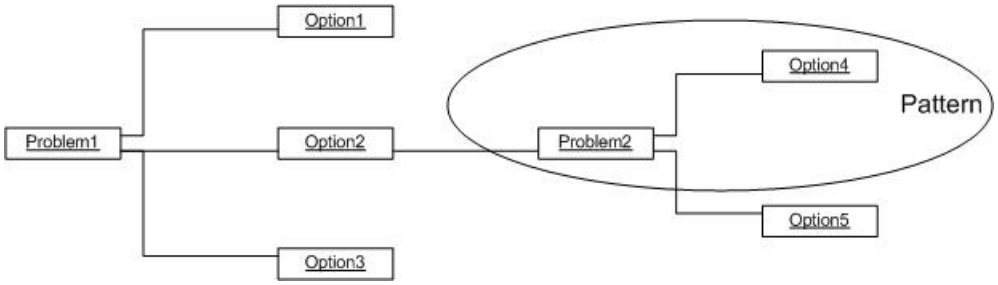

Figure 1: Abstracted design space

Our design space is an acyclic directed graph of problems and options, as shown in figure 1. In our design space, options for a problem are ranked with respect to quality indicators. A set of options may have different rankings on different quality indicators. On change effort, for example, option 4 might score better than option 5, while option 5 scores better on performance. These rankings are not shown in figure 1; they are introduced in section 3.

1.2 Related work

1.2.1 Architecture research

In Mary Shaw's categorization of research in software architecture [13], our research falls in the category "Empirical predictive model" (see figure 2). This is the only category for which Shaw gives no examples. We also strive to answer Shaw's call for better validation of research results.

1.2.2 Patterns

Another approach to describing design solutions are patterns. There is a lot of literature describing patterns; [8], [4] and [7] are among the most important. A design space differs from a collection of patterns in that it is more structured. A collection of patterns is mainly oriented towards solutions. Patterns are great for learning, because the solutions and how to implement them are described in great detail and nuances. For research, however, we need a more rigid structure. The problem that a pattern solves and the effects a pattern has on system quality, should be first-class citizens in our framework. We don't need elaborate implementation descriptions, but we do want to keep oversight over the set of options for a given problem. Therefore, we focus on design problems and quality indicators, and we describe options only as much as is needed for operationalization. The relation between our design space and patterns is shown in figure 1.

Research Product	Research Approach or Method	Examples
Qualitative or descriptive model	Organize & report interesting observations about the world. Create & defend generalizations from real examples. Structure a problem area; formulate the right questions. Do a careful analysis of a system or its development.	Early architectural models [21][51], architectural patterns[10],
Technique	Invent new ways to do some tasks, including procedures and implementation techniques. Develop a technique to choose among alternatives	Product line and domain-specific software architectures [14] [24], UML to support object-oriented design[8]
System	Embody result in a system, using the system development as both source of insight and carrier of results	Architecture description languages, especially ACME
Empirical predictive model	Develop predictive models from observed data	
Analytic model	Develop structural (quantitative or symbolic) models that permit formal analysis	HLA specification, COM inconsistency analysis

Figure 2: Categorization of research in software architecture, taken from [13].

1.2.3 Architecture documentation

Documenting architectures and architectural decisions is still an emerging discipline. Hofmeister et al. describe four views to an architecture [9]. The Archimate project [3] adds specific architectural constructs to UML. We choose a pragmatic approach to architecture documentation, using the views and models that make our ideas clear.

1.2.4 Software Engineering Institute

The Software Engineering Institute of Carnegie Mellon University has a research program called Software Architecture Technology (formerly Architecture Tradeoff Analysis), under which the field of Attribute Driven Design (ADD) is investigated. ADD has an underlying design space that is compared to ours in section 4. Our approach focuses on different quality attributes than ADD (for example, maintainability).

The SEI also investigates reasoning frameworks that enable reasoning about properties of entire systems, based on certified properties of the system's components. This approach is compared to our research design in paragraph 5.2.

The SEI has also developed evaluation methods for architectures. Among the SEI architecture evaluation methods, the ATAM is the best known. One of the steps in ATAM involves evaluating wether an architecture can meet some quality goals which are set in the previous phase in the form of scenarios. Our research results could be used in this step of the ATAM, because using our results in addition to the architects experience will yield a more robust evaluation.

1.3 Restrictions and assumptions

1.3.1 Types of systems

For practical reasons, only Enterprise Information Systems are subject of this research. These systems have the following properties:

- They serve administrative purposes
- At least some of the users are people
- They store and use data
- They are custom-built for the enterprise; no off-the-shelf software is considered.

1.3.2 Quality attributes

The main branches of quality attributes we are interested in, are *maintainability* and *reliability*, because they are thought to be heavily affected by architectural decisions and they are important cost drivers for systems. Also, our experience makes research into those attributes practical.

We do not expect to be able to add any relevant insights to the already large number of detailed models on the quality attributes *performance* and *resource efficiency*. We expect that most *usability* attributes are not affected by most architecture-level decisions. We do not take *security* attributes into consideration, because security is a too specialized topic for us to research.

1.4 Structure of this report

First we describe the design space, then we discuss how we will use the design space in our research.

In section 2, the core of the design space itself is described. In section 3, effects of decisions are added to the design space in the form of quality indicators. In section 4, our design space model is compared to some other models from the literature.

In section 5, we give a short description of how we plan to use our model for research purposes.

In appendix B, we describe the web site we are using to structure our hypotheses and results. This web site serves our research purposes, but also aims to aid system designers in reasoning about systems design.

2 The design space

A design space is a set of particular problems in the design of one system, their solution options (or *options* for short), and the relations between them. A design space is usually coherent, in the sense that the design problems apply to one system or part of a system.

We believe that a generalized design space for a class of systems can be given. In our case, this class of systems is the class of Enterprise Information Systems. A generalized design space consists of design problems and options that occur in most instances of a class of systems. Such a design space will never be complete. The number of design problems for a non-trivial class of systems is infinite. We aim to provide a part of the design space that is interesting enough to be relevant to most Enterprise Information Systems.

2.1 Structure

As explained above, a design space consists of design problems and their options. There is an obvious relation between them where one design problem is linked to multiple solution options. A less obvious relation is that a solution option for a given design problem can be the problem context for more detailed design problems. These more detailed design problems are called the *solution impact* of the solution option. For example, after choosing the option "relational database" for the design problem "where should the data be stored", up comes the design problem "what should the data model be?" This relation is called the *problem context* of a design problem. These relations are shown in figure 3.

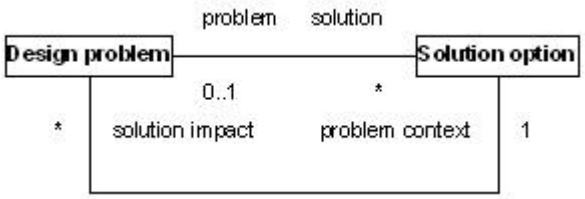

Figure 3: UML class diagram of design problems and solution options.

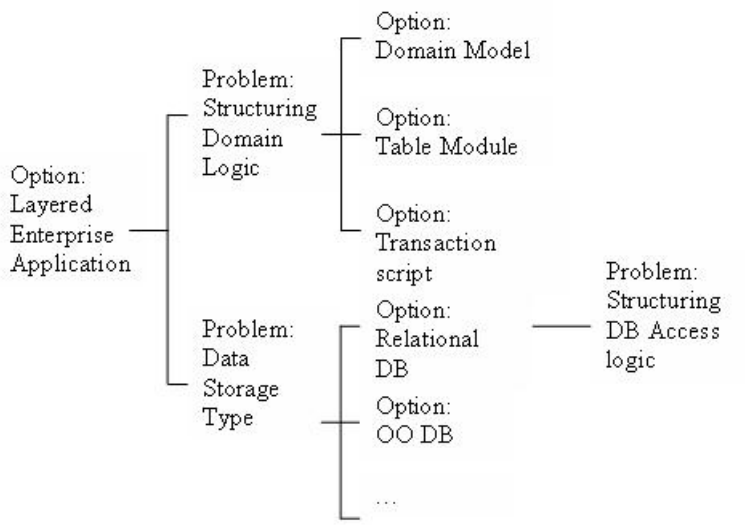

Figure 4: Example of an outline of a design space.

2.2 Form

Because design problems can only occur in one context and options can only solve one design problem, a design space forms a directed acyclic graph. A graph that represents part of our design space is shown in figure 4.

3 Adding impact relations to the design space

We want to research the effects of decisions on system quality. Therefore, *quality attributes* and *quality indicators* are added to the design problems and options to evaluate which option is best in a given situation.

3.1 Definitions

Quality indicator is a measurable property of a system that is of interest to at least one stakeholder. Examples include *modification effort per unit*

volume, *ratio of new faults at revision* and *mean time to failure*.

It's vital that a quality indicator be measurable, otherwise we couldn't use them in empirical research. This doesn't mean that the scale on which an indicator is measured should be a rational scale. We do require that the scales are at least ordinal, so that options can be ranked to each other based on indicators.

Quality attribute is an individually meaningful abstraction of a set of quality attribute indicators. Examples:

- *changeability* is an abstraction of *modification effort per unit volume*, *correction effort per defect* and *mean fault correction time*, among others.
- *stability* is an abstraction of *ratio of new faults at revision*.
- *maturity* is an abstraction of *mean time to failure* and *mean time between failures*, among others.

Quality indicators are usually defined as auxiliary to quality attributes. We turn this around. In these definitions, the indicator is chosen as leading over attributes, for two reasons:

- Indicators are measurable, attributes are not. Attributes can only be operationalized through their indicators, and most attributes have more than one indicator, so that attributes have different values for the same system depending on which indicator is chosen.
- We believe that indicators are more useful in requirements specifications, because what the customer really wants is a system with a certain value for a particular indicator, instead of an attribute being as high as possible.

Quality attributes and indicators for this study are mainly taken from the SERC Quint project [14], based on ISO-9126. The Quint categorization of Quality Attributes is shown in figure 5. When useful, we will use additional quality attributes and indicators, for example those from the SEI Architecture Tradeoff Analysis Initiative ([1], [2], [3]). Quality Indicators used in our study are listed in appendix A.

The model in figure 6 is the design space model from figure 3, with quality attributes and indicators added. The problem-solution relation from the former model is changed to a ternary relation with quality indicator as the third party.

Whenever a design problem is present, its different options are going to have different effects on the system's quality attributes. This is modelled as a ternary relation called *effect* between a design problem, solution option and quality indicator. The relation says that each solution to a problem has an effect on each quality indicator. For each existing combination of a design problem and a quality indicator, the solution options are ordered. This represents the hypothesis that one solution for a problem scores better with respect to a quality indicator than another solution. For this reason, the quality indicators are required to have at least ordinal scales. Effects can have a qualifier which limits

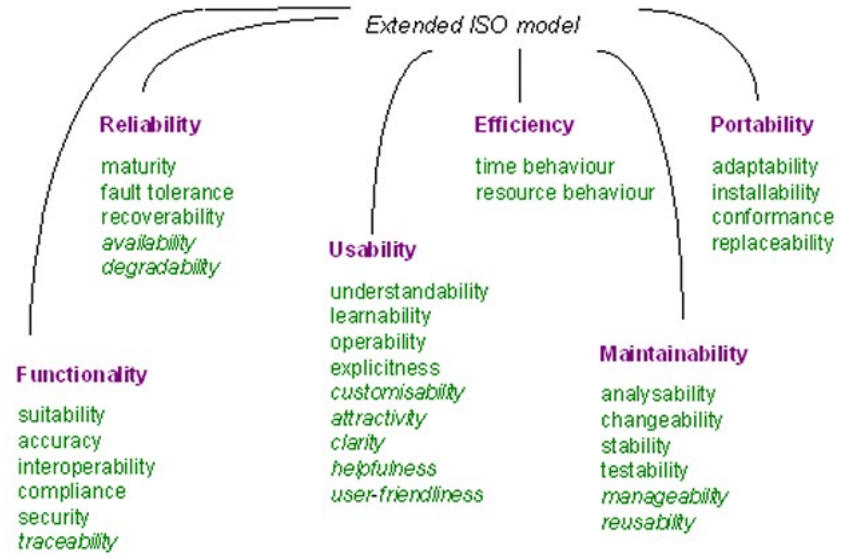

Figure 5: Hierarchical categorization of Quality Attributes, taken from [14].

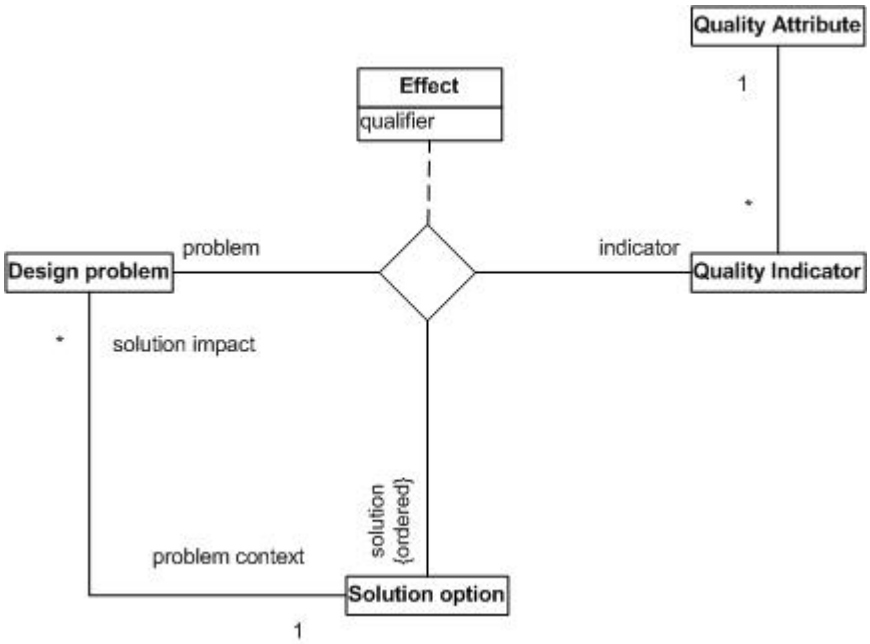

Figure 6: UML class diagram of the design space including effects.

the statement it makes to certain situations. Example: "*For large systems* (qualifier), *domain model* (option 1) gives lower *change effort* (quality indicator) than *transaction script* (option 2)."

The hypotheses identified in our study are listed in appendix C.

3.2 Example

The example in figure 7 shows the advantages and disadvantages of the different options for the design problem of where to check the business rules. When a user performs an action, for example, enters a new value for the age of a customer, then the system should check whether the action conforms to the business rules. The most logical place to check the business rules is in a domain component, but checking them in other layers has certain advantages. The example shows a set of hypotheses about the effects of these options on the system's quality.

The design problem here is "where to check the business rules". Each column of the table represents a solution option to this problem. On the left side of the table, the quality indicators are shown that are affected by the choice made for this design problem. For each quality indicator, a ranking is given. For the quality indicator "Change Effort", the option "Business rules in domain logic" is best. This means that in a system where the business rules are checked in the domain logic, we expect the average effort needed for changes will be lower than if business rules would be checked on the client.

For the indicator "Incorrect data ratio", a qualifier is given: "when the data storage is not only used through the SuD". We are not making any statement about the incorrect data ratio for systems that have exclusive access to their data storage, only for systems with shared data storages.

The orderings are partial. When two options have equal ranks for a quality indicator, such as the first three options for "Product fault density", we are not saying that those options are indifferent with respect to that indicator, but rather that we do not compare those options. The only statement we are making with respect to "Product fault density" is that the option "Business rules on client" is worse than the other options.

4 Other design space theories

4.1 DRL

Jintae Lee and Kum-Yew Lai in [10] present a graphical notation for design rationale, called Design Rationale Language (DRL).

Comparing DRL to our model, we see that DRL is a more complete language to capture design rationale, because they add an argument space in which the reasons why one option is better than another can be expressed. The argument space makes DRL more expressive in the context of actual design problems. Our model, however, includes a ranking of options for a combination of a design problem and a quality indicator. We hope that this aspect of our model will

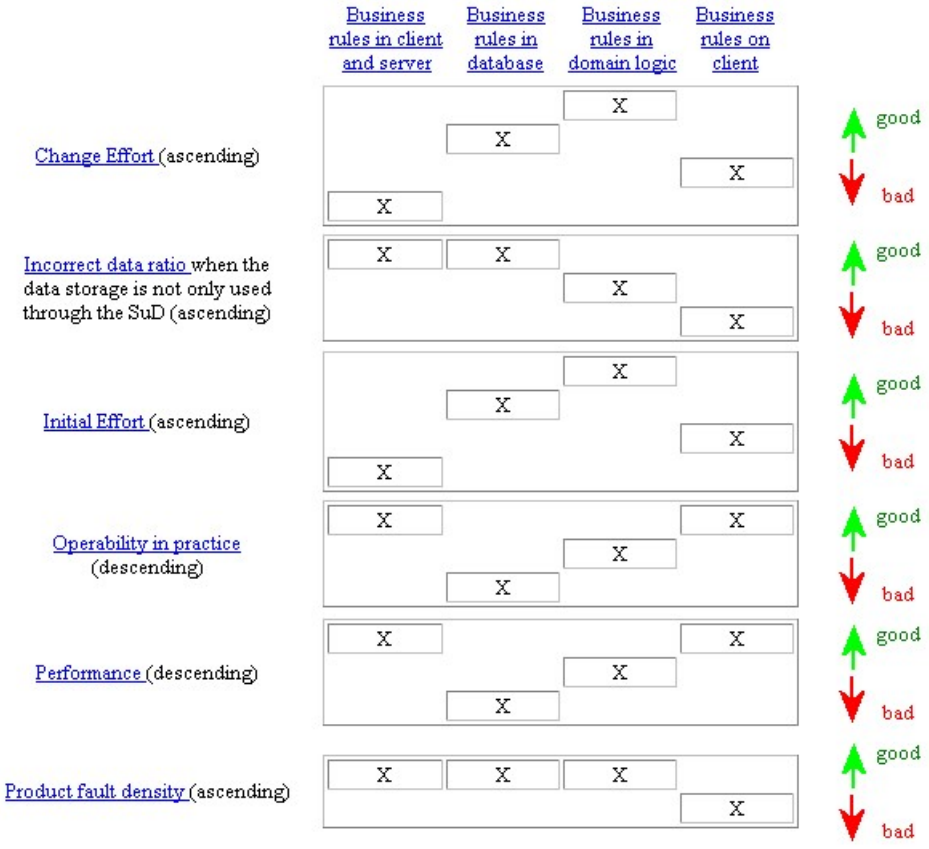

Figure 7: Example of the effects that the solution options of a design problem have on some quality indicators.

make it more useful for research purposes, in the process of generalizing from individual results to generic design problems.

4.2 QOC

In [11] a method (QOC) is presented for structured analysis of design spaces. Of all existing theories, our model matches QOC most closely. The Questions in QOC are the same as our design problems, their options correspond to our options and their criteria can be compared to our quality attributes. QOC has a relation between options and consequent questions, which is equivalent to our problem context relation.

The main difference between QOC and our model, is that in QOC, a specific design space is constructed for each problem, whereas we try to give a generalized design space for a class of systems.

4.3 IBIS

Another well-known decision making method is IBIS. This method contains the concepts of *issue*, *idea* and *argument*. These concepts on first sight seem to map on our concepts of *design problem*, *option* and *quality indicator*, respectively. However, an IBIS *issue* is a much broader concept than our *design problem*. An issue can be any question, whereas a design problem can only be a choice between some options. A question like "How can we improve this system's performance" can be an issue, but is not a design problem, because it isn't a choice. From the various solutions for improving performance, zero or more can be applied. That's not bad, but we expect that limiting ourselves to design choices makes it easier to generalize from our findings.

The difference between these concepts can be explained from the different goals. IBIS aims at *wicked problems*, where the problem is not in advance well defined and can change as more insight is gained. Therefore IBIS needs to be very flexible with respect to the problems that can be defined. Our framework is for well-defined problems, so we can use a more rigid structure.

4.4 AHP

The Analytical Hierarchical Process (AHP) as described in [12] and in [5] is a method for multi-attribute preference-based decision support.

The results of our method could be used as input to an AHP evaluation, when alternatives have to be generated and evaluated against objectives. Our model is different from the design space model of the AHP because the AHP design space model is meant for supporting a rational decision making process. Our design space model is meant for generalization of relations between decisions and their effects.

4.5 ADD

Attribute Driven Design [2] is a field of research at the SEI. Their unit of research is the *Design Primitive* (also known as *Architectural Mechanisms*). An example of a design primitive is *caching*. Design primitives can be compared to our Solution Options, except that a Design Primitive is not linked to a Design Problem. Design primitives are problem-oriented: "The performance is too low, so we use caching to fix that." Our design space is choice-oriented: "Should we use caching or not?"

Evaluation of Design Primitives is based on *General Scenarios*, which can be compared to the measurement protocols of our Quality Indicators, except that our Quality Indicators are measured on ordinal scales, and General Scenarios are pass/fail. A Design Primitive has one General Scenario as goal and can impact other General Scenarios as side effects.

Our approach differs from ADD in that we don't distinguish between goals and side effects. For what goal an option is chosen, in other words, which of the option's effects is most important for a system, is not relevant to our research. We treat all effects an option has on the system's quality indicators as equal. Another difference is our addition of design problems to structure the design space.

5 Discussion and conclusion

5.1 Empirical validation

5.1.1 Sub-hypotheses

Our goal is to empirically investigate (some of) the hypotheses we have collected. These hypotheses are about relations between design decisions and quality indicators.

We are looking for measurable correlations between design decisions and quality indicators. This correlation could be measured by taking a large enough sample of case studies. However, we fear that for most relations, it is not feasible to collect large enough samples, because we think the variance will be too large. That is because too many factors are working on the same quality indicators. For example, to directly measure a correlation between the choice of a domain logic structure and the changeability of the software, one needs to compensate for every other difference between the cases that affect changeability.

To avoid this problem, we will divide the relation between the design decision and the quality indicator into smaller steps, defined by intervening variables. The hypothesis then becomes that these intervening variables form a causal chain from design decision to quality indicator. These individual steps will be easier to investigate, because there will be less external variables. Some of these steps can even be validated analytically.

The model can also give the separate benefit of providing more insight. A fairly simple model of the relation between the choice of a domain logic structure

and the changeability of the software would be this set of sub-hypotheses:

- When using a *domain model*, less components need to be changed for an average change than with a *transaction script*.
- Change effort is proportional to the number of components changed.

These sub-hypotheses are individually easier to investigate. However, if they are still too difficult, they should be broken down into yet smaller sub-hypotheses. The claim that the sub-hypotheses together prove the entire hypotheses is supported by logic. For complicated structures of sub-hypotheses, a tree is drawn for the overview.

The given example is really too simple: if change effort would be proportional to the number of components changed, then lumping the entire system into one big component would improve changeability. That is obviously not the case, so before validating our hypotheses we should refine the model into more plausible sub-hypotheses.

5.1.2 Research designs

To substantiate the claim that change effort is proportional to the number of components changed, a quantitative research would be possible. We could take a large enough sample of cases in which a system was changed, count the number of components changed, and calculate the total associated change effort. Standard statistical techniques would then yield a correlation strength. If this correlation strength came above a certain treshold, we would have corroborated the sub-hypothesis.

Another way to substantiate the correlation between number of components changed and change effort, would be to do qualitative experiments in which the relation between the number of components and the change effort is isolated from all effects that other variables might have on change effort, such as team skill, component size, programming language, system functionality, to name just a few. It is extremely hard to isolate these effects. For example, to isolate team skill, one could use two teams are equally skilled in maintaining the system, which is difficult to determine. Alternatively, one could use the same team for both measurements, but then the team would be more skilled before the second measurement than before the first. Experiments are so hard in this area because they involve human and social factors, which in turn involve a host of extra variables. We think that experiments might be possible for sub-hypotheses in which a very direct cause-effect relation is mentioned.

5.1.3 Intermediate indicators

The general form of a model that breaks a hypothesis up into two sub-hypotheses, is "A influences B, and B influences C". In this form, B is an *intermediate indicator*. In the previous example, the number of components changed is an intermediate indicator.

Examples of well-documented intermediate indicators are:

- Cyclomatic complexity
- Number of lines of code

Intermediate indicators are important to this research, because they are the building blocks of our models. It is hard to find good intermediate indicators, because there are so many possible candidates. For example, instead of the number of components changed, we could choose the percentage of components changed, or the number of lines of code changed. Good intermediate indicators are very valuable, because when a relation between an intermediate indicator and a quality indicator has been proven, then effects of design decisions on the quality indicator can be investigated by determining their effect on the intermediate indicator, which may be much simpler.

A good intermediate indicator has a constant correlation with another indicator, be it a quality indicator or another intermediate indicator. This means that in a system, when you leave all other things the same and change only the first indicator, the other indicator changes in a predictable way. It is best if this change can be predicted on a rational scale, but an ordinal can be helpful too, as in the example: a larger number of components changed leads to a larger change effort, when all other circumstances remain the same.

Since we already said that this is probably not the case, we coin another indicator: the *duplication factor*. This is due to the common explanation why a *domain model* has better changeability than a *transaction script*: the latter leads to code duplication in larger systems. The exact definition of duplication factor will be part of this research.

A good intermediate indicator is easily measurable. The best are automatically measurable, for example, by running some tool over the source code. There are a number of tools for counting code lines, including or excluding comments, blanks, compiler directives etc. By decreasing the measurement effort, easily measurable intermediate indicators make larger samples of cases possible.

5.2 Other reasoning frameworks

The SEI has a research program on Predictable Assembly from Certifiable Components (PACC) [15]. In this approach, a Reasoning Framework is bound to an Abstract Component Technology, which describes the assembly structure of a system built of components. When a system's assembly structure can be described in terms of an Abstract Component Technology, the associated Reasoning Framework allows us to deduct quality attributes of the system from quality attributes of the components. Reasoning frameworks are targeted at specific quality attributes. The SEI describes example Reasoning Frameworks for attributes concerning performance, reliability and availability. Examples include Rate Monotonic Analysis and Temporal Logic. No Reasoning Frameworks for maintainability are given.

Our approach is very close to the SEI's Reasoning Frameworks. The main difference is that PACC sets out to predict system quality attributes based on component quality attributes, with the clear goal in mind of enabling a market

of certified components. We want to — eventually — predict system quality attributes based on decisions made in the system's design.

References

[1] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality attributes. Technical report, SEI, December 1995. CMU/SEI-95-TR-021.

[2] Len Bass, Mark Klein, and Felix Bachmann. Quality attribute design primitives. Technical report, SEI, December 2000. CMU/SEI-2000-TN-017.

[3] Len Bass, Mark Klein, and Gabriel Moreno. Applicability of general scenarios to the architecture tradeoff analysis method. Technical report, SEI, October 2001. CMU/SEI-2001-TR-014.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*, volume 1: A System of Patterns. John Wiley & Sons, 1st edition, August 1996.

[5] Ernest Forman and Mary Ann Selly. *Decision By Objectives (How to convince others that you are right)*. World Scientific, 2001.

[6] Martin Fowler. Patterns for things that change with time. Web Site. `http://www.martinfowler.com/ap2/timeNarrative.html`.

[7] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 1st edition, November 2002.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1st edition, January 1995.

[9] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Object Technology Series. Addison-Wesley Pub Co, 1st edition, November 1999.

[10] Jintae Lee and Kum-Yew Lai. What's in design rationale? In Thomas P. Moran and John M. Carroll, editors, *Design Rationale — Concepts, Techniques, and Use (Computers, Cognition, and Work)*, chapter 2, pages 21–52. Lawrence Erlbaum Associates, Mahwah, New Jersey, January 1996.

[11] Allan MacLean, Richard M. Young, Victoria M. E. Bellotti, and Thomas P. Moran. Questions, options and criteria: Elements of design space analysis. In Thomas P. Moran and John M. Carroll, editors, *Design Rationale — Concepts, Techniques, and Use (Computers, Cognition, and Work)*, chapter 3, pages 53–106. Lawrence Erlbaum Associates, Mahwah, New Jersey, January 1996.

[12] Thomas L. Saaty. *Mathematical models for decision support*, chapter What is the analytic hierarchy process?, pages 109–121. Springer-Verlag New York, Inc., 1988.

[13] Mary M. Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd international conference on Software engineering*, pages 656–664a. IEEE Computer Society, 2001.

[14] Bob van Zeist, Paul Hendriks, Robbert Paulussen, and Jos Trienekens. *Quality of software products — Experiences with a quality model*. Kluwer Bedrijfswetenschappen, 1996. Book in Dutch. Website contains all relevant information in English. See `http://www.serc.nl/quint-book/index.htm`.

[15] Kurt C. Wallnau. Volume iii: A technology for predictable assembly from certifiable components. Technical report, SEI, April 2003. CMU/SEI-2003-TR-009.

A Quality Indicators

This appendix lists the quality attributes and indicators used in this research. The indicators are listed in alphabetical order of attributes first and indicators second. For each indicator, the following properties are given:

Description: A short description of the Quality Indicator.

Unit: The unit the indicator is measured in, for example, *man hours* or *ratio*.

Source: The literature source the indicator was taken from, or "New" if the indicator was developed for this research.

Measurement protocols for indicators are not included in this report.

The choice of quality attributes and indicators in this report is based on the criteria that we need a certain volume of hypotheses to do interesting research. We have no intention of being complete in our description of software quality.

A.1 Accuracy

A.1.1 Failure ratio

Description: The ratio of incorrect processed transactions to the total of presented transactions.

Unit: Ratio.

Source: [14]

A.1.2 Incorrect data ratio

Description: The ratio of incorrect data items to the total number of data items.

Unit: Ratio.

Source: New.

A.2 Analysability

A.2.1 Analysis effort

Description: Mean time needed to analyze a failure, and to discover any faults arising from this failure, and separate the positions to be repaired by the maintainer who received the failure report.

Unit: Person-hours.

Source: [14]

A.2.2 Data analysis effort

Description: Mean time needed to analyze a record of data in the system and compare them to the state of the system's environment that the data represent.

The system's data can be data in the durable data storage, or in memory. The indicator can be used for all kinds of data.

When the data model is simple, this is straightforward. Many systems, however, have complex data models, including history, parameterized tables and columns, or abstraction relations, for example. With some complex data models, one needs special queries to find out what model the system has about its environment.

Unit: Person-hours per volume of data.

Source: New.

A.3 Changeability

A.3.1 Modification effort per unit volume

Description: Average amount of effort needed to modify the software product, per unit volume of the modification.

Unit: Person-hours per unit volume.

Source: [14]

A.4 Cost

A.4.1 Initial effort

Description: The effort needed to develop the system under development.

Unit: Person-hours.

Source: New.

A.5 Maturity

A.5.1 Product fault density

Description: The ratio of number of faults in a released product to the unit volume (e.g. pages, KLOC) of a released product (e.g. user documents, source code).

Unit: Number per unit volume.

Source: [14]

A.6 Resource Behavior

A.6.1 Memory usage

Description: Amount of internal memory needed to process a certain amount of data.

Unit: Bytes.

Source: [14]

A.7 Reusability

A.7.1 Ratio reusable parts

Description: Ratio of reusable parts of the software product to the total number of parts of the software product.

Unit: Ratio.

Source: [14]

A.8 Security

A.8.1 Resistance

Description: Estimate of the probability that with a certain amount of effort (time, money, equipment) the software security measurements will not be bypassed.

Unit: Probability.

Source: [14]

A.9 Testability

A.9.1 Test effort per unit volume

Description: Effort needed to test one unit volume of the software product with a certain testing coverage degree.

Unit: Person-hours per unit volume.

Source: [14]

A.10 Time behaviour

A.10.1 Average internal transaction time

Description: Average time a certain internal processing task occupies with a certain usage load.

Unit: Seconds.

Source: [14]

B Web site

The design space of this research has been made available as a web site, based on XML documents. The web site has two goals:

1. To help administrative information system architects and engineers to find and evaluate solutions to problems they face in designing a system.

2. To derive goals for research projects that evaluate the effects of solutions on system quality.

The current contents of the web site are outlined in figure 8. Currently, the web site is being extended with more design problems and solution options. A usability study is underway to check if professional systems designers are able to use it and benefit from it in their daily work. In the future, the web site structure will be extended so that our research results fit in.

Figure 8: Outline of the full current design space.

C List of Hypotheses

This appendix lists all hypotheses contained in the web site. The hypotheses are organized alphabetically, first by the design problem they belong to, second by quality indicator. For each hypothesis, the following properties are shown:

Text: A textual representation of the hypothesis (as opposed to the graphical representation in the web site).

Source: Literature source that has proposed the hypothesis, or "New" if no literature source has been found.

Support: Whatever support exists for the hypothesis to be valid.

The choice of hypotheses in this report is based on the criteria that we need a certain volume of hypotheses to do interesting research. We have no intention of being complete in our list of hypotheses. These hypotheses are not stated as truths, but rather as a starting point for research; we expect that a lot of these hypotheses will have to be altered after investigation. The current list of hypotheses satisfies our criteria.

As a rule, because all hypotheses compare two or more options against each other with respect to a quality indicator, the option with the most favourable score on that indicator is named first.

C.1 Structuring the application logic

C.1.1 Hypothesis H1

Text: For changes that affect application logic only, *Operation script* scores better on *Change Effort* than *Application logic in domain components*, and *Application logic in domain components* scores better on *Change Effort* than *Application logic in presentation components*.

Source: New.

Support: None.

C.1.2 Hypothesis H2

Text: When the application logic uses multiple transactional resources, *Operation script* scores better on *Initial cost* than *Application logic in presentation components*, and *Application logic in presentation components* scores better on *Initial cost* than *Application logic in domain components*.

Source: New.

Support: None.

C.1.3 Hypothesis H3

Text: When the application logic is used by multiple presentation logic components, *Operation script* scores better on *Initial cost* than *Application logic in domain components*, and *Application logic in domain components* scores better on *Initial cost* than *Application logic in presentation components*.

Source: New.

Support: None.

C.2 Structuring the data access logic

C.2.1 Hypothesis H4

Text: *Active record* scores better on *Analysis effort* than *Table data gateway*, and *Table data gateway* scores better on *Analysis effort* than *Row data gateway*, and *Row data gateway* scores better on *Analysis effort* than *Data mapper*.

Source: [7]

Support: Claimed by [7].

C.2.2 Hypothesis H5

Text: For changes in DB access logic or domain logic, *Data mapper* scores better on *Change Effort* than *Row data gateway*, and *Row data gateway* scores better on *Change Effort* than *Table data gateway*, and *Table data gateway* scores better on *Change Effort* than *Active record*.

Source: [7]

Support: Claimed by [7].

C.3 Data caching

C.3.1 Hypothesis H6

Text: *No data caching* scores better on *Failure ratio* than *Data caching on server*, and *Data caching on server* scores better on *Failure ratio* than *Data caching on client*.

Source: New.

Support: Author's experience.

C.3.2 Hypothesis H7

Text: *No data caching* scores better on *Initial Effort* than *Data caching on server*, and *Data caching on server* scores better on *Initial Effort* than *Data caching on client*.

Source: New.

Support: Author's experience.

C.3.3 Hypothesis H8

Text: *Data caching on client* scores better on *Performance* than *Data caching on server*, and *Data caching on server* scores better on *Performance* than *No data caching*.

Source: New.

Support: Author's experience.

C.4 Data storage type

C.4.1 Hypothesis H9

Text: Either *Object Database* or *Prevalent system* scores better on *Change Effort* than *File storage*, and *File storage* scores better on *Change Effort* than *Relational database*.

Source: New.

Support: None.

C.4.2 Hypothesis H10

Text: *Relational database* scores better on *Failure ratio* than *File storage*, and *File storage* scores better on *Failure ratio* than *Object Database*, and *Object Database* scores better on *Failure ratio* than *Prevalent system*.

Source: New.

Support: None.

C.4.3 Hypothesis H11

Text: Either *Object Database* or *Prevalent system* scores better on *Initial cost* than *File storage*, and *File storage* scores better on *Initial cost* than *Relational database*.

Source: New.

Support: None.

C.4.4 Hypothesis H12

Text: *Prevalent system* scores better on *Performance* than *Relational database*, and *Relational database* scores better on *Performance* than *Object Database*, and *Object Database* scores better on *Performance* than *File storage*.

Source: New.

Support: None.

C.4.5 Hypothesis H13

Text: *Relational database* scores better on *Product fault density* than *File storage*, and *File storage* scores better on *Product fault density* than *Object Database*, and *Object Database* scores better on *Product fault density* than *Prevalent system*.

Source: New.

Support: None.

C.5 Distribution

C.5.1 Hypothesis H14

Text: *Non-distributed system* scores better on *Initial cost* than *Distributed system*.

Source: New.

Support: Author's experience.

C.5.2 Hypothesis H15

Text: *Non-distributed system* scores better on *Test effort per unit volume* than *Distributed system*.

Source: New.

Support: None.

C.6 Distribution configuration

C.6.1 Hypothesis H16

Text: *Broker* (Node configuration can be changed at runtime, meaning that a node with certain services can be replaced by another node which provides the same services, without changing or even shutting down the other nodes) scores better on *Change Effort* than *Remote proxy* (for changes where nodes are transferred to different machines (but this is not possible at runtime, as with Broker)).

Source: [4].

Support: Claimed by [4].

C.6.2 Hypothesis H17

Text: *Remote proxy* scores better on *Initial cost* than *Broker*.

Source: New.

Support: None.

C.6.3 Hypothesis H18

Text: *Remote proxy* scores better on *Performance* than *Broker*.

Source: [4].

Support: Claimed by [4].

C.6.4 Hypothesis H19

Text: *Broker* (each single node is easier to test) scores better on *Test effort per unit volume* than *Remote proxy*.

Source: [4].

Support: Claimed by [4].

C.7 Structuring the domain logic

C.7.1 Hypothesis H20

Text: When system size is above a certain treshold, *Domain model* scores better on *Change Effort* than *Table Module*, and *Table Module* scores better on *Change Effort* than *Transaction Script*.

Source: [7]

Support: Claimed by [7].

C.7.2 Hypothesis H21

Text: When system size is below a certain treshold, *Transaction Script* scores better on *Change Effort* than *Table Module*, and *Table Module* scores better on *Change Effort* than *Domain model*.

Source: [7]

Support: Claimed by [7].

C.7.3 Hypothesis H22

Text: When system size is above a certain treshold and is not built in .NET, *Domain model* scores better on *Initial Effort* than *Table Module*, and *Table Module* scores better on *Initial Effort* than *Transaction Script*.

Source: [7]

Support: Claimed by [7].

C.7.4 Hypothesis H23

Text: When system size is below a certain treshold and is not built in .NET, *Transaction Script* scores better on *Initial Effort* than *Table Module*, and *Table Module* scores better on *Initial Effort* than *Domain model*.

Source: [7]

Support: Claimed by [7].

C.7.5 Hypothesis H24

Text: When system size is above a certain treshold and is built in .NET, *Table Module* scores better on *Initial Effort* than *Domain model*, and *Domain model* scores better on *Initial Effort* than *Transaction Script*.

Source: [7]

Support: Claimed by [7].

C.7.6 Hypothesis H25

Text: When system size is below a certain treshold and is built in .NET, either *Table Module* or *Transaction Script* scores better on *Initial Effort* than *Domain model*.

Source: [7]

Support: Claimed by [7].

C.7.7 Hypothesis H26

Text: *Transaction Script* scores better on *Performance* than *Table Module*, and *Table Module* scores better on *Performance* than *Domain model*.

Source: [7]

Support: Claimed by [7].

C.8 Layering

C.8.1 Hypothesis H27

Text: For changes that occur within one layer, *Layered Enterprise Application* scores better on *Change Effort* than *non-layered system*.

Source: [4].

Support: Claimed by [4].

C.8.2 Hypothesis H28

Text: For changes that occur in multiple layers, *non-layered system* scores better on *Change Effort* than *Layered Enterprise Application*.

Source: [4].

Support: Claimed by [4].

C.8.3 Hypothesis H29

Text: *Non-layered system* scores better on *Initial cost* than *Layered Enterprise Application*.

Source: [4].

Support: Claimed by [4].

C.8.4 Hypothesis H30

Text: *Non-layered system* scores better on *Performance* than *Layered Enterprise Application*.

Source: [4].

Support: Claimed by [4].

C.8.5 Hypothesis H31

Text: *Layered Enterprise Application* scores better on *Effort for portability* than *non-layered system*.

Source: [4].

Support: Claimed by [4].

C.8.6 Hypothesis H32

Text: *Layered Enterprise Application* scores better on *Product fault density* than *non-layered system*.

Source: [4].

Support: Claimed by [4].

C.8.7 Hypothesis H33

Text: *Layered Enterprise Application* scores better on *Ratio reusable parts* than *non-layered system*.

Source: [4].

Support: Claimed by [4].

C.8.8 Hypothesis H34

Text: *Layered Enterprise Application* scores better on *Test effort per unit volume* than *non-layered system*.

Source: [4].

Support: Claimed by [4].

C.9 Structuring the presentation logic

C.9.1 Hypothesis H35

Text: For changes in one part of the presentation together with its associated presentation and business logic, *Presentation-Abstraction-Control* scores better on *Change Effort* than *Model-view-controller*, and *Model-view-controller* scores better on *Change Effort* than *Model in presentation*.

Source: New.

Support: None.

C.9.2 Hypothesis H36

Text: For changes in either the presentation logic or the application and/or business logic, *Model-view-controller* scores better on *Change Effort* than *Presentation-Abstraction-Control*, and *Presentation-Abstraction-Control* scores better on *Change Effort* than *Model in presentation*.

Source: New.

Support: None.

C.9.3 Hypothesis H37

Text: *Model in presentation* scores better on *Initial cost* than *Model-view-controller*, and *Model-view-controller* scores better on *Initial cost* than *Presentation-Abstraction-Control*.

Source: New.

Support: None.

C.9.4 Hypothesis H38

Text: *Model-view-controller* scores better on *Product fault density* than *Model in presentation*, and *Model in presentation* scores better on *Product fault density* than *Presentation-Abstraction-Control* (Few people have experience with this pattern, which leads us to believe that many pitfalls are yet to be discovered).

Source: New.

Support: None.

C.10 Representing history

C.10.1 Hypothesis H39

Text: Either *Temporal object* or *Temporal property* or *Effectivity* or *Snapshot* or *Audit log* scores better on *Auditability* than *No history*.

Source: [6]

Support: Claimed by [6].

C.10.2 Hypothesis H40

Text: Either *No history* or *Audit log* scores better on *Data analysis effort* than *Snapshot*, and *Snapshot* scores better on *Data analysis effort* than either *Temporal object* or *Temporal property* or *Effectivity*.

Source: [6]

Support: Claimed by [6].

C.10.3 Hypothesis H41

Text: Either *No history* or *Audit log* scores better on *Initial cost* than *Snapshot*, and *Snapshot* scores better on *Initial cost* than either *Temporal object* or *Temporal property* or *Effectivity*.

Source: [6]

Support: Claimed by [6].

C.10.4 Hypothesis H42

Text: *No history* scores better on *Memory usage* than either *Audit log*, and either *Audit log* scores better on *Memory usage* than either *Temporal object* or *Temporal property* or *Effectivity*, and either *Temporal object* or *Temporal property* or *Effectivity* gives a better value on *Memory usage* than *Snapshot*.

Source: [6]

Support: Claimed by [6].

C.11 Server authorization

C.11.1 Hypothesis H43

Text: *No protection* scores better on *Initial cost* than *Protection on server*, and *Protection on server* scores better on *Initial cost* than *Protection proxy*, and *Protection proxy* scores better on *Initial cost* than *Protection on client* (Assuming multiple clients) .

Source: New.

Support: None.

C.11.2 Hypothesis H44

Text: *No protection* scores better on *Performance* than *Protection on client*, and *Protection on client* scores better on *Performance* than *Protection proxy*, and *Protection proxy* scores better on *Performance* than *Protection on server*.

Source: New.

Support: None.

C.11.3 Hypothesis H45

Text: *Protection on server* scores better on *resistance* than *Protection proxy*, and *Protection proxy* scores better on *resistance* than *Protection on client*, and *Protection on client* scores better on *resistance* than *No protection*.

Source: New.

Support: None.

C.12 Where to check the business rules

C.12.1 Hypothesis H46

Text: *Business rules in domain logic* gives a better value on *Change Effort* than *Business rules in database*, and *Business rules in database* gives a better value on *Change Effort* than *Business rules on client*, and *Business rules on client* gives a better value on *Change Effort* than *Business rules in client and server*.

Source: New.

Support: None.

C.12.2 Hypothesis H47

Text: When the data storage is not only used through the SuD, either *Business rules in database* or *Business rules in client and server* scores better on *Incorrect data ratio* than *Business rules in domain logic*, and *Business rules in domain logic* scores better on *Incorrect data ratio* than *Business rules on client*.

Source: New.

Support: None.

C.12.3 Hypothesis H48

Text: *Business rules in domain logic* gives a better value on *Initial Effort* than *Business rules in database*, and *Business rules in database* scores better on *Initial Effort* than *Business rules on client*, and *Business rules on client* scores better on *Initial Effort* than *Business rules in client and server*.

Source: New.

Support: None.

C.12.4 Hypothesis H49

Text: Either *Business rules on client* or *Business rules in client and server* scores better on *Operability in practice* than *Business rules in domain logic*, and *Business rules in domain logic* scores better on *Operability in practice* than *Business rules in database*.

Source: New.

Support: None.

C.12.5 Hypothesis H50

Text: Either *Business rules on client* or *Business rules in client and server* scores better on *Performance* than *Business rules in domain logic*, and *Business rules in domain logic* scores better on *Performance* than *Business rules in database*.

Source: New.

Support: None.

C.12.6 Hypothesis H51

Text: Either *Business rules in domain logic* or *Business rules in database* or *Business rules in client and server* scores better on *Product fault density* than *Business rules on client*.

Source: New.

Support: None.