Adaptivity
in
Software Architecture

A. F. M. van der Sijpt

Supervisors: dr. J. J. Lukkien (TU/e – W&I-SAN)
            ir. M. F. Offermans (Luminis)

Arnhem, June 2007
Summary

Driven by various technological advancements, the world in which software systems live is getting ever more dynamic. With more and more electronic devices being driven by software and getting networking capabilities, the landscape of services and users is calling for software systems that can adapt themselves to these changes.

In an attempt to understand this adaptivity, this project starts out with working toward a taxonomy of software architecture styles, driven by the secondary research statement,

“What are quality factors of architectural styles?”

in which quality factors are defined as the characteristics that divide or unite software architecture styles.

With the need for explaining adaptivity in mind, a view on software architecture has been developed that focuses on the notion of concepts. Using this, a method for classifying and comparing both software architectures and architecture styles in the form of a typing report has been developed. Although this report does not provide direct evidence for the link between quality factors and concepts, two case studies show the soundness of the method.

Adaptivity itself is handled by the primary research statement,

“How to integrate adaptivity into software architecture?”

We provide a definition, and create a categorization of the kinds of adaptivity available.

Six intuitive concepts get linked to adaptivity. From the 11 software architecture styles collected for the taxonomy, three are selected for their ability to support adaptive behavior, based on the concepts they incorporate.

We conclude that adaptivity is a quality factor. In the category of automated adaptive systems, the gradation of adaptivity is determined by the amount of environment modeled in the system.
Preface

This Master’s Thesis is the result of project *Adaptivity in Software Architecture*, conducted at Luminis, Arnhem, and supported by the System Architecture and Networking group of Eindhoven University of Technology, from October 2006 to June 2007.

Having started my life as a student in 1998, and having earned a Bachelor’s degree in Computer Science from Fontys Hogeschool Eindhoven in 2003, I knew I wanted something new to end my Master’s program at Eindhoven Technical University. I have completed my share of engineering assignments for both internships and my graduation project, and decided to do something more fundamental and abstract.

With this assignment, Luminis has provided me with exactly that: a fundamental assignment with a chance to find my own direction in the course of the project. Both software architecture and adaptivity are elusive subjects, requiring a tremendous amount of thinking time and headaches, making this into one of the most intense project I ever did.

All throughout the project, I have received much support from various directions. First of all, I want to thank Luminis for having me. In particular, I want to thank Marcel Offermans, who was my supervisor and has found a great balance between steering me in the right direction and letting me flop about, and Hans Bossenbroek and Jeroen Bouwrie for spending so much time discussing the subjects I was working on, even though their time was limited.

I would like to thank Johan Lukkien, who helped me develop the shape of the assignment, and has provided feedback while writing this thesis.

During the project, I had a great number of contacts outside of Luminis and Eindhoven Technical University. On March 9, 2007, some 15 architects of Luminis’ customers gathered at Panalytical, Almelo, to let me validate my ideas, and discuss the subjects of this thesis. Finally, I would like to thank my contacts in the Open Source Software community for sharing their views on the impact of software architecture on their projects.

Angelo van der Sijpt,
June 2007
# Contents

1 Introduction .................................. 1
   1.1 Architecture and style ...................... 1
       1.1.1 Traditional views: Parnas, Zachman, and Perry & Wolf .... 2
       1.1.2 Alternative views: Fielding .......................... 3
       1.1.3 Working views ................................. 3
       1.1.4 The components .............................. 5
   1.2 Adaptivity .................................... 6
       1.2.1 Types of adaptivity ......................... 6
       1.2.2 Conclusion .................................. 7
   1.3 Goals ......................................... 7
   1.4 Approach ..................................... 8

2 Toward a taxonomy of styles .................. 9
   2.1 Literature .................................... 9
       2.1.1 A field guide to boxology ...................... 9
       2.1.2 Classifying elements .......................... 10
       2.1.3 Classifying styles ............................ 11
       2.1.4 Extending styles ............................. 12
       2.1.5 Conclusions ................................ 13
   2.2 Collecting architecture styles ............. 13
       2.2.1 Description method ........................... 13
   2.3 Discussion sessions ......................... 14
       2.3.1 Session 1 .................................. 15
       2.3.2 Session 2 .................................. 16
       2.3.3 Conclusion .................................. 17
   2.4 Conclusion .................................... 17

3 A model of style ................................ 19
   3.1 The model ..................................... 19
   3.2 Typing report .................................. 20
       3.2.1 Quality factors ............................... 20
       3.2.2 Typing report ................................ 23
   3.3 Taxonomy ...................................... 23
       3.3.1 In style descriptions .......................... 23
       3.3.2 In literature: on styles ....................... 23
       3.3.3 In literature: on projects ................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.11</td>
<td>REST</td>
<td>73</td>
</tr>
<tr>
<td>B</td>
<td>Preparation discussion 1</td>
<td>78</td>
</tr>
<tr>
<td>C</td>
<td>Results discussion 1</td>
<td>88</td>
</tr>
<tr>
<td>D</td>
<td>Results discussion 2</td>
<td>93</td>
</tr>
<tr>
<td>E</td>
<td>An engineer is not a carpenter</td>
<td>98</td>
</tr>
<tr>
<td>F</td>
<td>Software architecture as a wicked problem</td>
<td>109</td>
</tr>
<tr>
<td>G</td>
<td>Typing report</td>
<td>115</td>
</tr>
<tr>
<td>G.1</td>
<td>Quality factors</td>
<td>115</td>
</tr>
<tr>
<td>G.1.1</td>
<td>Physical properties</td>
<td>115</td>
</tr>
<tr>
<td>G.1.2</td>
<td>External factors</td>
<td>116</td>
</tr>
<tr>
<td>G.1.3</td>
<td>Internal factors</td>
<td>117</td>
</tr>
<tr>
<td>G.2</td>
<td>Usage</td>
<td>117</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This document is the result of Master’s project ‘Adaptivity in Software Architecture’. It presents a method of comparing and classifying software architecture styles, and focuses on the subject of ‘adaptivity’ using this method.

The project’s planning and the original assignment can be found in [vdS06].

The remainder of this chapter shows a rough sketch of the landscape this research lives in, shows the codified goals of the project, and details its approach.

1.1 Architecture and style

The term architecture has been used in construction for millennia, referring to the science and art of designing structures for human use. Over the ages, architecture evolved from a method of joining needs and means (e.g., need for housing and the availability of sticks and leaves) to the current profession of designing structures in relation to their environment.

In a more general setting, architecture refers to translating a vision, based on needs and constraints, into artifacts which can be used as a basis for further development; this makes architecting an inherently subjective activity.

Ever since programming became a discipline of increasing scope and complexity, terms as ‘structure’, ‘design’ and ‘architecture’ have been used to describe a high-level view of the system at hand.

Software architecture is hardly the only –architecture. All fields dealing with creating complex structures rely on a form of architecture. Information systems, enterprises, buildings, tools, etc. can all have a high-level view which allows relating the artifact to its environment.

It has long been known that similar systems tend to lead to architectures which have overlapping properties; this notion has been labeled architecture style\(^2\), after its namesake in construction. It is hard to find a widely accepted definition of style; Merriam-Webster uses the following definition.

“A distinctive quality, form, or type of something.”

\(^1\)The ‘creation’ is a key element: many other fields of science deal with complex structures, but feel no need of describing an architecture. Examples include astronomy and medicine.

\(^2\)In this document, the term ‘software architecture style’ is preferred over ‘software architectural style’, since the first links software architecture to style, instead of software to architectural style.
which is not really specific to architecture, although the ‘distinctive’ element is useful. Dutch dictionary Wolters uses the definition

“De gezamenlijke kenmerken van een zekere richting of periode in de kunst/bouwkunst.”

which translates as

“The set of common characteristics of a given school or period in art or construction.”

This definition is more like it: a style is a set of common characteristics. In construction, this set of characteristics is usually dictated by the available techniques and materials, the zeitgeist, and certain beliefs of a group of architects; Art Deco was inspired by the technological advances shortly after WW I, embodied in a.o. facets and stepped motifs\(^3\).

Architectural styles in software are dictated by problem domains, available technology and the ‘hype of the moment’. Although the basics remain the same, construction architecture is much more tangible; a building in a given style can immediately be recognized as such. For example, the palace on Dam Square in Amsterdam is a classicist building, mixing the grandeur of Roman architecture with the most modern techniques of the 18th century. For software systems, the architecture is far less visible; two systems that appear to be identical to the end-user may have been built using very different architectural styles.

1.1.1 Traditional views: Parnas, Zachman, and Perry & Wolf

One of the earliest writings on the necessity of structured programming was [Dij69], but one of the best-known is On the criteria to be used in decomposing systems into modules by David Parnas [Par72]. Although it does not use the term architecture explicitly (the first articulate link between construction and software was made by John Zachman, [Zac87]), it is easy to recognize the elements that today are taken as important for software architecture: modularization, definition of interfaces and separation of concerns. The main goal of this approach is dividing a problem into smaller, manageable pieces with a clear focus.

In the early 90’s, scientific interest in software architecture started picking up, first with Foundations for the study of software architecture by Dewayne Perry and Alexander Wolf [PW92], shortly followed by An introduction to software architecture by David Garlan and Mary Shaw [GS94]. Perry and Wolf present an intuition of software architecture and style, most prominently stating that

- an architecture is an underspecification of a system,
- architecture has multiple views for different use(r)s,
- “The architect is the mediator between the user and the builders of the system”,
- a style is just a less specific architecture.

Garlan and Shaw build upon this notion, which is also acknowledged by Perry and Wolf. In addition, [GS94] also sketches a path towards a professional software architecture discipline, expecting advances in sharing knowledge about software architecture by, a.o.,

\(^3\)A notable example of this is the Chrysler Building.
• “Better taxonomies of architectures and architectural styles.”
• “Notations for describing architectural designs.”
• “Techniques for extracting architectural information from existing code.”
• “Better understanding of the role of architectures in the life-cycle process.”

Summarizing, Parnas’ view of modularization and separation of concerns has remained intact for several decades.

1.1.2 Alternative views: Fielding

There are, however, people who protest the traditional view. In his doctoral dissertation, *Architectural Styles and the Design of Network-based Software Architectures*, Roy Fielding [Fie00] presents an idealized view modern web applications, today known as REST [FT02]. To do so, however, he needs a view on software architecture that allows him to portray his ideas on these applications. Being dissatisfied with the contemporary views, the first chapter of his dissertation is dedicated to developing his own vision on software architecture.

Fielding’s views differ from the traditional views on some crucial points.

• A **software architecture** is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.”
  This deviates from the rather static view of modules with responsibilities; the responsibilities and activities of components may shift during operation. Furthermore, this definition does away with the notion of an architecture as a description of a system, ready to be started: an abstraction of a running system may not necessarily be equal to the still state of its components.

• On a technical note, Fielding does not support the distinction between control flow and data. Traditional views rely on control signals that tell components what to do, and data flows to do it on; in Fielding’s view, all activity is driven by data.

1.1.3 Working views

The view below has been developed during the course of the project. It is presented here to create a basis for understanding the rest of the document.

Appendix E shows some more background on this view, its reasons for my acceptance, and argues an altered understanding of software architecture; it really is different than the disciplines it borrows it connotation from. This view is not intended as a ‘new theory of architecture’, but is rather a collection of elements from various theories that are suitable as a means of handling adaptivity.

**Introduction**

Software architecture uses a borrowed terminology, placing the architect in a role where he is the one that envisions the system, and seems to be mainly occupied with creating its high-level structure. Like this is not entirely true in construction, I believe it is not in software either.
Architecture is about making a system fit its context. This applies to the system as a whole; I do not believe architecture is primarily about structure and ‘chopping up’ a problem into manageable pieces [Par72]. Rather, it is about cohesion and alignment.

In construction, the blueprints and maquettes are manifestations of the architecture. The actual choices the architect makes are about the style of a building; decisions like ‘I don’t want any corridors ending in three doors’. These choices embody unshakeable constraints on the architecture, stating what is required to make the product fit its context. Note that this can include, but is not limited to, structure.

While the architectural choices above are very articulate, it is often possible that they are more ‘hidden’, they will manifest themselves in the resulting artifacts, but cannot be articulated as such. Still, they are part of the architecture, and can find their ways into elements like rationale.

A model

An architecture is a set of concepts, guidelines for creating the resulting artifacts. This idea puts styles on the same level of abstraction as architectures, being a collection of concepts, brought together with a convenient name.

Both styles and architectures can support certain quality factors, given that the implementation does not break them. A system can only exhibit desirable non-functional properties when both the architectural support and engineered implementation are in place. The artifacts that embody the architecture give rise to the system. This system will exhibit measurable, non-functional properties. This cohesion is shown in Figure 1.1.

In this figure, dashed lines represent the named relations of the previous paragraph. Cardinalities are apparent, and have been left out. Two relations are marked with a question mark. First of all, the link between concepts and quality factors is what the model from Chapter 3 attempts to explain. The second question marked connection is between non-functional properties and quality factors. This project has assumed them to be the same. I believe they actually are, or at least one-on-one mappings can be formulated.
1.1.4 The components

Quality factor The nature of quality factors will be discussed in Chapter 3.

“Quality factors are the (non-)functional properties that an architecture can support, given an upholding of the architecture’s concepts in all relevant artifacts.”

Architecture “An architecture is a set of concepts, brought together to make the product fit its context.”

Style Extending the ideas in Section 1.1 and following Perry & Wolf’s view, a style is a less specific architecture.

“A style is a collection of concepts to be called by a convenient name, possibly accompanied by a rationale for this combination of concepts.”

Concept The most intangible element of this model is the concept. These embody the choices made by the architect, the leading principles of an architecture. The introduction has shown that concepts in construction are decisions that could be applied to all buildings, but have been chosen to be included in just this situation.

“A concept is a fundamental choice—articulated in a pattern, guideline or even some examples—making the product fit its context, and should be recognizable throughout the product.”

This principle means that the key concepts making up the architecture should fit on no more than a few pages. However, rationale and elaboration of the concepts can take up quite some more paper, but are not considered part of the architecture.

Artifact Artifacts are the embodiment of the architecture, tangible ‘stuff’ that shows the concepts the architecture is made of. This can include documents, but also code. In all of these, the influence of the architecture has to be held up.

The element that is most commonly seen as ‘the’ architecture, the system’s structure with modules and interfaces, requires some extra attention. When the context requires a fixed structure, this can very well be a concept; however, when other concepts dictate some structure, it will only be part of the artifacts. Furthermore, in a system with a dynamic structure, it will not even be part of any document, but will be dictated at runtime by governing concepts.

System The system is the physical manifestation described in the artifacts, i.e., the system behavior.

Non-functional The non-functional properties of a system indicate ‘how well’ a system does its job.
1.2 Adaptivity

Merriam-Webster’s dictionary states that adaptivity comes from the adjective adaptive, meaning “showing or having a capacity for or tendency to adaptation”. Adaptation leads to adapt, a verb coming from Latin adaptare, which means ‘(to make) fit’, ‘make suitable’. So, we propose a definition of adaptivity as

“having the capacity or tendency to make suitable.”

That leaves us with the question ‘what makes what suitable to what?’. In the remainder of this section, the what’s shall, respectively, be known as object, subject and environment.

To get some feeling for the matters at hand, let’s consider the nice, technical example of IP packet routing. Even in its most basic form it shows good adaptation to changing environments, such as failing lines (the example will be revisited in Section 4.2). So, what is adapting what to what?

- [Object] The best way is to see the network as the intelligent element. After all, the components that make up the network (systems, routers, switches) together exhibit this behavior, while it cannot be seen in a single component.

- [Subject] The network adapts itself; the changes happen in a number of the networks components.

- [Environment] The components that make up the network adapt for failing of their colleagues, like malfunctioning equipment or broken lines.

So, does this mean that ‘the network makes the network suitable to (changing make-up of) the network’. Uninspiring as it may seem, this is a correct conclusion.

1.2.1 Types of adaptivity

With the definition above, we can define six different types of adaptivity when filling in different ‘things’ for the object, subject and environment; they are summed up in Table 1.1. The letters do not actually denote anything, they merely indicate ‘the same’ or ‘different’. This categorization is a corollary of the definition above, which in turn stems from the dictionary definition of ‘adapt’.

The types of adaptivity we are striving for are categories 2 and 3; anything below is not ‘automatic’ enough, and 4 is (currently) unattainable. A potential seventh type, A-B-A, is possible, but just not useful. This could be ranked between 0+ and 1, and could be something like ‘customization’; when an environment seems to adapt a system to its needs, the boundaries of the system are likely ill-defined.

To get some feeling for the types of systems the categories define, we will consider some examples for each category.

Category 0 systems are systems with no potential for adaptation whatsoever; these are very rare, only the proverbial brick would classify. Anything that even has a slight possibility of being altered will fall into category 0+.

Classic control systems fall in category 0+ or 1. Open loop controllers are members of category 0+: they may be adapted by an outside source, but have no inkling of the thing they are controlling. Closed loop controllers fall in category 1. For instance, a cruise control
system adapts an outgoing parameter (throttle control) to an incoming parameter (vehicle speed). Even when extending the boundaries of the system to include its terminators, it still is a category 1 system. Also included are systems with special provisions to be adapted, such as systems that require certain adapters to be created before they can operate, ERP and CRM systems.

Systems today classified as ‘robust’ are found in category 2. These systems are able to cope with changes in their own parameters, with IP routing (see Section 1.2) as the classic example. PID controllers are able to adapt their behavior to the reactions of the environment, and therefore are category 2.

Systems beyond the notion of ‘robust’ display properties of homeostasis; with a changing environment, they will try to function as well as possible by scaling back some qualities, instead of falling over and dying when their requirements are not fully met. [Sha00] provides the example of news gathering websites (nowadays embodied by Google News): they will provide best service when all their sources are available, but will continue to function, albeit with a lower quality of service, when one of them becomes unavailable. These systems make up category 3.

Category 4 systems display ultimate adaptivity. These are able to adapt to any new circumstance, without making provisions of the parameters that change. The human mind might qualify, but even we might be rule-based agents, unable to understand our own complexity.

1.2.2 Conclusion

With a definition of adaptivity in place, we now have an intuition of the kind of adaptivity we are looking for to handle: systems that can adapt themselves to changing parameters in their environment, with given parameter ranges and given autonomy. The categories of interest are 2 and 3.

1.3 Goals

The main research goal of this project is

“How to integrate adaptivity into software architecture?”
with a secondary research question,

“What are quality factors of architectural styles?”

or, more concretely, how do we define the ‘things a style can do’, and how can we use this notion to compare and classify software architecture styles?

In their overview of the current state of the field, *The Golden Age of Software Architecture*, Mary Shaw and Paul Clements [SC06] show the areas of software architecture where advances can be expected; it is nice to note that these two research questions are part of those areas.

- “Organizing architectural knowledge to create reference materials. Mature engineering disciplines are characterized by handbooks and other reference materials that provide engineers with access to the systematic knowledge of the field. Cataloging architectural patterns is a first step in this direction. But in addition, we need reference materials for analysis of realized architectures for evaluation of designs to predict properties of their implementation.”

- “Developing architectural support for systems that dynamically adapt to changes in resources and each user’s expectations and preferences. As computing becomes ubiquitous and integrated in everyday devices, both base resources such as bandwidth and information resources such as location-specific data change dynamically. Moreover, each individual user has different needs that change with time. Developing architectures that can dynamically anticipate and react to these changes would help to maximize the benefit each user can obtain. Achieving this will require not only adaptive architectures but also component specifications that reflect variability in user needs as well as intrinsic properties of the component.”

### 1.4 Approach

To reach the main goal, we will first need some understanding of what software architecture is made of. This immediately points to the secondary goal; we have chosen the approach of quality factors to find this understanding. As an exercise in using them, we will attempt the creation of a taxonomy of architecture styles.

The various approaches for creating this taxonomy will be shown in Chapter 2. They include the creation of a collection of styles to be classified (see Appendix A), a study of available literature, and two discussion sessions with software architects from Luminis and its customers.

The knowledge collected gives rise to a framework for linking styles to qualities in Chapter 3. We will show that filling in this model runs into problems, but also present recommendations for using it, the potential applications of the resulting catalog, and formulate some hypotheses about the link between concepts and quality factors.

Chapter 4 handles adaptivity. We revisit the different kinds of adaptivity shown above, and try to link them to relevant concepts in architecture. Finally, we map these concepts onto the styles we have collected in Appendix A, to see which styles seem to be best suited for given types of adaptivity. This will be accompanied by some general advice for creating software with adaptive properties.

The main text of this thesis is wrapped up in Chapter 5, evaluating the course of the project and its contributions.
Chapter 2

Toward a taxonomy of styles

This chapter explores the existing possibilities for creating a taxonomy of software architecture styles, and searches for leads into creating this. As a focus for this taxonomy, we want the ‘things a style can do’, defined as quality factors in Section 1.1.

A study of available literature will show that although guides for classifying styles or elements of software architecture do exist, they have not yet been adopted en masse. Combining this with a focus on patterns and ‘engineerable’ properties leads to a landscape of potentially usable material that is not expanded, and a general lack of focus on software architecture as defined in Section 1.1.

A collection of architecture styles does not give any direct leads in the direction of a taxonomy, but at least provides us with potential ‘inhabitants’, and presents a good exercise in describing styles with very different properties and levels of abstraction.

Two discussion sessions with architects from Luminis and its customers led to an understanding of what it is we were not looking for, and on why this subject seems so elusive.

2.1 Literature

Various founding endeavors to compare and classify elements in software architecture have been undertaken in the past. We will show a number of these, and interpret their usability using the Publish/Subscribe architectural style, as detailed in Section A.10.

2.1.1 A field guide to boxology

In *A Field Guide to Bozology: Preliminary Classification of Architectural Styles for Software Systems*, Mary Shaw and Paul Clements [SC97] describe the status of the emerging field, and outline some of the steps necessary to make it a mature engineering discipline. They give a description of five styles, followed by guidelines for selecting one of these based on problem properties.

Figure 2.1 is based on this guideline. Styles Repository, DBMS, Blackboard and Abstract data are handled in [SC97], while Publish/Subscribe has been added to give some feel for the method of classification.

However simple the diagram in Figure 2.1 may be, it gives a number of interesting clues about the domain.
Figure 2.1: Mapping problems to styles in [SC97]. Ovals indicate statements from problem space, boxes indicate styles, lines indicate connections and arrows indicate sub-style relations.

- Styles look rather disjoint in this figure. This suggests a strict distinction between the styles, making them unsuitable for problems with different characteristics. However, this figure does not show some of the more abstract properties that are likely to connect seemingly separate styles.

- A hierarchy of styles is indicated by this figure. Some ‘basic’ styles can be used for problems with given properties, with specializations for additional problem properties. Note that the manner of drawing hints another hierarchical composition, being a hierarchy of problem properties.

2.1.2 Classifying elements

One of the main drives for software architecture research is the promise of design reuse; the creation of systems composed from off-the-shelf components and patterns with well-understood properties, decreasing time and effort required for system development.

To facilitate reuse of architectural elements (being components, connectors or patterns), [MG96] presents the ‘Software Shelf’. This tool’s matching method is based on the style that an element belongs to, and the role it plays: a Publish/Subscribe system (pattern) can be instantiated with a sensor and a monitor (components), connected to the same event service (connector).

For well understood styles and elements, this tool can be very helpful. However, it is exactly this assumption that limits its use. The tool provides no mechanisms for classifying elements or styles, but rather relies on a detailed understanding of the landscape of architecture styles.

In another effort to allow the sound combination of off-the-shelf elements in an architecture, [KCBA97] provides a method for signature- and semantic matching. This method uses the notion of feature to describe and match architectural elements, i.e., it can decide what elements can ‘work together’. Tables 2.1 and 2.2 show the division in temporal and static features, combined with the feature description of Publish/Subscribe’s Process element.

The Process’ temporal features are largely dictated by the asynchronous nature of its style; getting and giving control can happen at any moment, data transmission and receipt are of no interest at the moment of creation and destruction of the component, but can happen at any other moment, a process cannot spawn children, and a process will retain its own state. On the static side, a Process shares its data with others using the event service, but also
Table 2.1: Temporal features of the Publish/Subscribe Process, method from [KCBA97].

Table 2.2: Static features of the Publish/Subscribe Process, method from [KCBA97].

allows matching elements with similar (or at least compatible) features. An added benefit of this method is what the authors call “naturally-occurring clusters”, or seemingly unrelated elements that turn out to share many features, and may well be compatible.

2.1.3 Classifying styles

Having roughly the same intention as [KCBA97], [SC97, sec. 3] provides a conceptual framework for classifying architecture styles based on five features, together making up a multidimensional space. Styles are represented by ‘points’ in this space, while families of styles are subspaces. A hierarchical view is discouraged, since this loses information about relations between, and overlap of, style families.

Table 2.3 shows the features as distinguished by this method, exemplified with the Publish/Subscribe style.

This table shows four dimensions; a fifth dimension, not pictured here, deals with the type of reasoning that can be applied when analyzing properties of the system.
Control issues

<table>
<thead>
<tr>
<th>Style</th>
<th>Component</th>
<th>Connectors</th>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publish/Subscribe</td>
<td>Process</td>
<td>Event service</td>
<td>star</td>
<td>asynch</td>
<td>invocation, runtime</td>
</tr>
</tbody>
</table>

Data issues

<table>
<thead>
<tr>
<th>Style</th>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
<th>Isomorphic shapes</th>
<th>Flow directions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publish/Subscribe</td>
<td>star</td>
<td>continuous or sporadic, low and high volume</td>
<td>multicast</td>
<td>runtime</td>
<td>yes</td>
<td>same</td>
</tr>
</tbody>
</table>

Table 2.3: Features for Publish/Subscribe, method from [SC97].

Note that the method does not prescribe the ‘possible answers’ like Section 2.1.2 does. While suggesting some possible answers based on current styles, it leaves much freedom for expanding the universe of classification. This allows the classification of new styles, and the definition of currently non-existent styles by identifying a ‘void’ in the space.

In What is Style?, David Garlan [Gar95] shows requirements for a theory of software architecture style. While not giving a definite best model, he gives some requirements that each model should handle and, hence, each description of a style should incorporate.

- [Vocabulary] A set of definitions of the ‘parts’ that make up the style. These are likely names for specific component and connector types, such as pipes and filters or clients and servers.
- [Configuration rules] Also known as topological constraints, these rules define how elements may be composed into a system. This ranges from rules that state the number of connections that can be handled by a single connector, to rules on the topology of the system, such as prescribing a hierarchical makeup.
- [Semantic interpretation] A method of interpreting the (topologically correct) connected elements, such that the meaning (as opposed to the syntax) of the system emerges.
- [Analyses] Based on a correct composition of elements, an architecture can be analyzed to have various properties. A style should define ways of analyzing, and properties to be analyzed, or at least have a framework that allows the creation of analyses.

2.1.4 Extending styles

Selection of architecture styles is a complex matter, since there is no unambiguous description method for styles. In Attribute-Based Architecture Styles, Mark Klein et al. [KKB+99] propose a ‘wrapper’ for architectural styles, containing the following elements.

1. [Problem description] For what kind of problem is this style usable?
2. [Quality attribute measures] What attributes are measurable?
3. [Architectural style] The architectural style, presented in any manner that seems fit.
4. **[Quality attribute parameters]** How to measure the attributes described in part 2?

5. **[Analysis]** A formal relation between the style and the measurable attributes’ values.

This wrapper is known as an Attribute-Based Architecture Style, or ABAS. The method speaks of “. . .enhanced architectural styles, ( . . . ) and we view them as the next generation in the development of architectural styles.”

The main drawback of this approach is its dependency on an understanding of the architecture style under investigation, like the Software Shelf of Section 2.1.2.

### 2.1.5 Conclusions

One eye catching feature of the methods in the preceding sections is their focus on technical properties, such as the intended environment (Section 2.1.1), whether a component will fork (Section 2.1.2) and the names of components (Section 2.1.3). The only exception to this is Section 2.1.4, but this provides few leads for starting a classification.

The ABAS presented in Section 2.1.4 lives on the wrong level to be usable in a taxonomy as intended in this study; it assumes knowledge of the basic principles that make up styles. As soon as these are known (so, as soon as a taxonomy is known to do its job correctly), ABASs may provide a useful addition for documenting collected knowledge. The same applies to the Software Shelf of Section 2.1.2.

In the end, none of the investigated methods provide guidance for selecting, measuring and using quality factors. The basic description method of Section 2.1.3 is the most useful framework for now.

### 2.2 Collecting architecture styles

Appendix A shows 11 architecture style descriptions. These have been created for a number of reasons,

- Architecture styles, especially in software, are complex material. Like recognizing their use in specific architectures, recognizing ‘what is’ an architecture style based on a description is best done by seeing a lot of them; given enough examples, they will be recognized on sight.

- Even as recent as [SC06, Sec. 5], it is claimed that reference material on software architecture is necessary. The collection presented in aforementioned appendix is likely a step in that direction.

- One of the approaches mentioned in [vdS06] to reach a classification of styles is the identification of patterns in a homogeneous collection of styles.

#### 2.2.1 Description method

The styles in Appendix A represent very different domains and levels of abstraction. The method of describing, therefore, is rather loose. To get a somewhat homogeneous form, making comparison possible, the following form is maintained.
• Each description starts out with a narrative description of the style, detailing the important issues, considerations and motivation. The styles’ variedness prevents a ‘standard’ shape of this narrative.

• Following this is a prime example, in which the style is illustrated, showing the motivations described above. Again, the variedness of the styles calls for a specific example for each style; a general running example would wrong the descriptive capabilities each style has for its own domain.

• A more codified element in the description is based on Section 2.1.3, showing the necessary four elements of a style description.

• The wrapup summarizes the main elements of the style, and provides a partial typing report (see Chapter 3). As will become apparent in the following chapter, it is not possible to provide a founded full typing for all styles; instead, we will handle the physical properties in Appendix A, and focus on the quality of adaptivity in Chapter 4.

Note that this setup is somewhat similar to the ABAS from Section 2.1.4. Although I do not believe that ABASs are a next step in the development of architectural styles, I do believe that bringing together styles with some of their typical properties and considerations is beneficial.

The material presented in Appendix A is not necessarily new. Most of this has been written about before, and will be accompanied by the necessary literature references. Of the four style elements from Section 2.1.3, semantic interpretation and analyses are usually not provided in literature; they have been reconstructed as well as possible with the available information.

Furthermore, a collection of styles in this way is not available in literature. Garlan and Shaw’s An introduction to software architecture [GS94] shows a basic set of styles, but lack the accompanying elements, and is from 1994. Gamma et al’s Design patterns: elements of reusable object-oriented software [GHJV95] handles a collection of design patterns with all necessary elements, but does not focus on architecture styles, which, as shown in Section 1.1, are not necessarily interchangeable.

2.3 Discussion sessions

The process planning arranged for at least one, and possibly two discussion sessions among architects. There are three reasons for this approach,

• Undocumented information is required.

• Having extensive experience in working with the material, architects are likely to have a ‘feeling’ for the qualities of given architecture styles.

• I lack knowledge and experience to keep a one-on-one interview going. Therefore, although harder to arrange, a discussion among architects, in which I mainly facilitate, is a better option.

Two sessions have been held, each with a specific focus. The following sections describe the ideas shaping the session, the material used, the course of the session, and the main results.
It should be noted that the material presented below is linked directly to the sessions, and not all of it can be found in this document. More material on the discussion sessions can be found in Appendices B through D, with Appendix E containing material on the subject of ‘what is software architecture’ omitted in the main text.

2.3.1 Session 1

The first discussion was held December 1, 2006, with four Luminis architects present, together with my TU graduation supervisor.

Preparation

The discussion tries to find a definition of ‘quality factor’, together with initial factors for use in the model described below. A secondary goal is assessing preliminary conclusions of the project.

Hypotheses

At roughly one-quarter into the project, the model was a multi-dimensional space with axes representing quality factors, and styles being ‘regions’ in this space, with smaller regions denoting more specific styles. This means an inherent coupling between styles and quality factors like ‘communication style’.

Preliminary conclusions at the time included that

- there is no consensus as to the nature of architecture,
- there is no lack of interest in the field, but its terminology is often misused,
- architectural choices are rarely made in a ‘pure’ way, taking only technical issues into account,
- software architectures strive for a ‘quality without a name’, and
- this quality is not engineerable, since software architecture can be classified as a wicked problem.

Material

Two weeks before the session, a preparatory document was sent to the participants (Appendix B), detailing the project’s status, as above, and the session’s goals. The document intends to create a common image and terminology, making sure that we ‘talk about the same thing’.

Three types of material were used during the session,

- A set of propositions based on preliminary conclusions, intended to check their correctness with practitioners.
- Comparisons of architecture styles, to identify elements that are ‘present’ in one style, and not in another: these are indicators of quality factors.
- Practical examples, intended to observe the process of finding a style for a problem.
Session  Discussion on propositions went well: the resulting discussion is interesting for the participants, and shows what is deemed important.

In comparing styles, it became apparent ‘the wrong stuff’ was looked for: while asking for quality factors, the session was steered in the direction of physical properties. These can help in classifying styles, but do not record what a style can do. This created a not-so-smooth discussion.

The practical examples were, in retrospect, not suitable for discussing in less than an hour. Contrary to what was expected, two activities unfolded:

- Even with seemingly clear problem statements, there is a tendency to request additional information: much more context is required than was anticipated.

- Even for underspecified problems, it seems to be possible to identify ‘load-bearing’: issues that must be addressed to solve the problem. These are mostly not technical ones, but about the system’s intended use and external influences.

Results

A prominent movement noticed in literature is the tendency to isolate software architecture from its context to study it. The approaches above, unfortunately, seem to have done the exact same thing: isolating styles and high-level decisions from the problems’s context, hoping that without external influences, ‘pure’ style decisions can be made. It turns out these external influences and non-functional properties are in fact key elements in the process.

In the end, the ideas of quality factors and their model turn out to be wrong. Instead of creating a new system, it is better to ‘scale back’ and create a model which uses well-known software quality factors.

A full evaluation of this discussion session is shown in Appendix C (in Dutch, since it was intended for internal circulation).

2.3.2 Session 2

The second discussion was held March 9, 2007, with some 15 architects from various clients of Luminis.

Preparation

This session was intended to fill in the model in Chapter 3. Shortly before the session, the insight arose that software architecture is harder than can be captured in the model, for reasons that will not be elaborated here. Since exactly that is a useful conclusion, it was decided to test that instead.

Hypotheses  Little over halfway into the project, quite a feel for the essence of software architecture arose: there is something that causes the approaches to ‘break’ at a certain point (more on this in Appendix E).

Also, from correspondence with administrators of open source projects (see Section 3.3), each of them men with a lot of experience, it became apparent that there are certain ‘feelings’ and ‘traits’ of architects that are present, but hard to articulate; it simply cannot be named.
Material Instead of using a preparatory document, it was decided to ‘tap in’ to the experience of the participants by giving a quick overview of (1) the goals of the project and (2) what has been done, followed by propositions similar to those used in the first session.

Session With the limited background given in the introduction, it turns out to be difficult to convey ‘what it is I am looking for’, especially since this is hard to formulate. A short introduction was chosen, which was too brief to get this ‘feeling’ across: instead of jumping directly from the project goals to its current state, it would have been better to show more of the ‘journey’ that led to those insights.

However, this understanding soon arose during the discussion, partially caused by the presented propositions. Unlike in the first session, I now had the necessary knowledge, built by studying the subject, to join in the discussion and clarify the propositions.

Results

Although there are some different opinions on details, there is a general consensus that supports my hypotheses. For a more detailed discussion of the session’s results, see Appendix D.

2.3.3 Conclusion

The original goal of finding a system of quality factors to characterize architectural styles, based on knowledge that can be ‘retrieved’ from architects, has not been reached. Indeed, there is a system now, as shown in Chapter 3, but this has not been influenced by the discussion sessions in the anticipated way.

The first session showed the ‘wrong things’ were being pursued. After the necessary changes to the approach, an image developed that ‘pointed to’ why software architecture is a ‘tough subject’. Although there is no direct evidence of this in literature, the second session supported this view.

The original assumption that I do not have the knowledge necessary to participate in a discussion is no longer true, as shown during the second session. Still, with this newly acquired knowledge, one-on-one interviews would not provide the results that have been reached now: a one-sided view may emerge, for which I still lack the necessary skills to question it.

In short, the discussion sessions, although hard to arrange, provide some of the greatest and most useful influences on the project. They provide more steering than can be expected from studying literature and doing thought experiments, and result in a more balanced picture than one-on-one interviews.

2.4 Conclusion

Studying the subject has left us with a load of inspiration for classifying and comparing styles, 11 styles to be classified, an intuition on what the taxonomy should be able to do, and an idea of the trickyness of software architecture.

It turns out some rather workable ideas for creating classifications of software architecture (and styles) have been created, though these rarely have any followup. Methods focussing on styles usually focus on ‘engineerable’ properties, or the elements that an engineer can ‘put into’ the style; this is not aligned with the ideas of software architecture in Section 1.1.
Still, the material and knowledge shown above have given rise to a model of style, as shown in the next chapter. Elements from literature and research that have been adopted into the model will be introduced there.
Chapter 3

A model of style

Based on the research in Chapter 2, this chapter will collect the components which are suited for our needs. These will be fit together into a report that allows typing of both systems and software architecture styles.

Section 3.3 shows attempts for using this report to find the concept-quality factor link, followed by some practical experiences in using the report in Section 3.4.

3.1 The model

The secondary research goal (see Section 1.3) reads “What are quality factors of architectural styles?”. While we have not yet answered that question, we know what we need to reach this goal: some way of linking concepts to quality factors (see Section 1.1).

The concepts of an architecture are the first element to be included in our model. For this purpose, we adopted the style typing method from [SC97], as shown in Section 2.1.3. This method focuses solely on physical properties; still, this is the only standardized method for now, and can be explained and used quite easily. For future expansions of this, refer to Section 3.5.

The quality factors from Section 1.1 apply to a given system; they represent two layers of properties for the model, since we can make a distinction between internal and external properties. The external properties include everything that is ‘measurable’ when observing a running system. Internal properties are identified when the system is being maintained. As inhabitants for these layers, we chose a subset of ISO 9126 [ISO]. While these may not be identified as quality factors, Section 1.1 has noted that quality factors and non-functional properties are closely related.

Drawing some inspiration from ATAM [BCK98, ch. 11], it was decided domain specific factors should be allowed, differentiating similar styles intended for use in different domains. Furthermore, the properties that have been identified for a given category can ‘leak’ to another: for example, for most systems, maintainability is an internal property, but it can well be an external property (e.g. support for on-line updates).

Next to supporting non-functional properties that will exhibit themselves in the resulting system, there can be other considerations in choosing a style. These may include the architect’s or team’s experience, the style’s ease of use, or endorsement by some governing body. These properties are quality factors of the style; even though we defined a style as a ‘less-specific’ architecture, it can display quality factors of its own. This creates a fourth layer
of properties, completing the model. We have not selected a standard for this layer, but will allow any kind of input. Once enough style typing reports have been collected, some pattern will emerge of properties that keep showing up, and can be accepted as standard contents.

Figure 3.1 shows the resulting model, relating the elements described above.

### 3.2 Typing report

With the conceptual shape of the model in Figure 3.1 in place, we need contents for this. The previous section has already hinted the inhabitants of the layers; these will be detailed below. This section ends with a typing report, combining the shape and its inhabitants into a single form.

#### 3.2.1 Quality factors

Section 3.1 has given two sources for properties to fill in the typing report [SC97, ISO]. While they are not necessarily the best properties, they can at least be assumed to be well understood.

**Physical properties**

These properties characterize the physical properties of the conceptual system. Its items are likely to have pointable counterparts in code, but it is the concept that counts: e.g., using packet-based communication, a conceptual stream (continuous) can be created, and a streaming channel can be used to deliver bursts of information (sporadic).

- **[Constituents]**
  - **[Components]** By what ‘name’ do we call the active components of this style? Examples can be clients, filters, etc.
  - **[Connectors]** By what mechanisms do the components communicate? Examples can be shared data, remote method invocation, data streams, etc.

- **[Control issues]** The subject of control flow handles the way components work together, i.e., request others’ assistance, delegate work units, etc.
– [Topology] Which elements invoke functionality in others, and what ‘form’ does it take when this is drawn? Examples are sequential programs with subroutines (creating a hierarchical topology) or client-server systems (having a star-shaped topology).

– [Synchronicity] In what way do elements synchronize their actions? [SC97] proposes the following types,
  * **sequential lockstep**, every other components’ state can be derived from a single component’s state,
  * **parallel lockstep**, like synchronous lockstep but with more potential threads of control; still, all elements have a rough idea what others are doing,
  * **synchronous**, synchronizations between elements happen regularly, but not necessarily always with the same partner,
  * **asynchronous**, synchronization based on message passing,
  * **opportunistic**, communications only when necessary, not to reach some synchronization between components; typical for agent-based systems.

– [Binding time] At what time is the ‘other side’ of a cooperation known?

• [Data issues] How does data move around the system?
  
  – [Topology] The shape of data flow, similar to the flow of control above.
  
  – [Continuity] Does data flow through the system continually, or in bursts?
  
  – [Mode] What mechanism is used to allow data flow through the system? [SC97] proposes the following types,
    * **passed**, message passing,
    * **shared**, using a shared store,
    * **copy-out-copy-in**, shared, but with discrete modifications,
    * **broadcast**,
    * **multicast**.
  
  – [Binding time] When is the partner found for a data transfer?

• [Control/data interaction] How do data and control flow influence each other?
  
  – [Shape] Are the shapes of the data and the control flow similar?
  
  – [Directionality] Does data flow in the same direction as control does?

External factors

External factors are measurable in a running system. ISO 9126 provides characteristics that can well be reused here. Note that the distinction in internal and external factors is not as rigid as it seems here; see Section 3.1 for more information on this.

• [Reliability] “A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.”
  
  – [Maturity] “The capability of the software product to avoid failures as a result in the software.”
The capability of the software product to maintain a specified level of performance in cases of software faults or infringement of its specified interface.

- [Recoverability] “The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure.”

- [Efficiency] “A set of attributes that bear on the relationship between the software’s performance and the amount of resources used under stated conditions.”

  - [Time behavior] “The capability of the software product to provide appropriate response and processing time and throughput rates when performing its function.”
  
  - [Resource behavior] “The attributes of software related with measuring the amount of resources required to perform its function.”

**Internal factors**

Internal factors can be recognized in the design of a system. Again, we stick to ISO 9126 for this.

- [Maintainability] “A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adaptations) of software to environmental changes and changes in the requirements and functional specifications.”

  - [Analyzability] “The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or the parts to be modified to be identified.”

  - [Changeability] “The capability of the software product to enable a specified modification to be implemented.”

  - [Stability] “The capability of the software product to avoid unexpected effects from modifications of the software.”

  - [Testability] “The capability of the software product to enable modified software to be validated.”

- [Portability] “A set of attributes that bear on the ability of software to be transferred from one environment to another (this includes the organizational, hardware or software environment).”

  - [Adaptability] “The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.”

  - [Installability] “The capability of the software product to be installed in specified environment.”

  - [Replaceability] “The capability of the software product to be used in place of another specified product for the same purpose in the same environment.”
3.2.2 Typing report

To make use of the model described in the previous sections, we need to ‘give it a face’. Marrying the conceptual form of Section 3.1 to the concrete elements in Section 3.2.1 gives rise to a typing report, as shown in Figure 3.2. Appendix G will show a separate description of the report, and some guides for using it, based on Section 3.4.

In the typing report, the original shape of the model with four layers is still recognizable. For each of the questions, it proposes some answers, but also leaves room for custom ones. As stated in Section 3.1, no contents have been decided upon for the upper layer; hence, it is empty.

3.3 Taxonomy

With the typing report in Section 3.2.2 completed, we can apply this to various knowledge sources, as detailed below. We hope to come up with some undeniable links between the concepts on layer 1, and the properties on layers 2 and 3.

3.3.1 In style descriptions

With nearly a dozen style descriptions in Appendix A, each of them accompanied by at least an evaluation of its physical properties, it should be possible to recognize some ‘patterns’.

As reasonable as this may sound, it turns out to be based on a false assumption. Like in the first discussion session (see Section 2.3), a system of physical factors was aimed for, while that was not what is interesting. Furthermore, even now this rather elaborate body of knowledge is available, when keeping the factors that are actually intended in mind, no creeping suspicions or grand themes seem to emerge. This could be due to the broadness of the field and the relatively small number of styles described.

3.3.2 In literature: on styles

Explicit literature on style evaluation is rare; most notable (once again) is [SC97]. Other references to qualities of architectures are made, such as [Che03, LOP02]. Evaluating architecture styles, however, is not the purpose of these documents, meaning that knowledge of architectural styles is taken as a given fact. Unfortunately, this leads to an over-simplification of the issues at hand, resulting in claims being made about styles without proper justification or references.

3.3.3 In literature: on projects

Books like [BCK98] present cases in which one of the authors has been working as a consultant for some firm with an interesting problem. These cases clearly present the information that is most relevant to an academic view of the problem; for a nice example, see [BCK98, Sec. 11.4]. Given this ‘openness’ that organizations apparently have agreed upon, it seemed reasonable to expect the existence of more literature on this subject.

Alas, this assumption turned out to be wrong. Case studies of very specific elements of software projects are to be found, but a full analysis, starting at software architecture, and ending up with an evaluation of planned and actual quality factors, is not to be found. This, in fact, is not that surprising: with software becoming an ever more important asset of
<table>
<thead>
<tr>
<th>Style</th>
<th>Architectural factors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Desirable
- Undesirable

<table>
<thead>
<tr>
<th>Internal factors</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzability</td>
<td>Desirable</td>
</tr>
<tr>
<td>Changeability</td>
<td>Desirable</td>
</tr>
<tr>
<td>Testability</td>
<td>Desirable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External factors</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maturity</td>
<td>Desirable</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>Desirable</td>
</tr>
<tr>
<td>Recoverability</td>
<td>Desirable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time behavior</td>
<td>Desirable</td>
</tr>
<tr>
<td>Resource behaviour</td>
<td>Desirable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical factors</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Control issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>Asynchron.</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>Opportunis.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
</tr>
<tr>
<td>Linear</td>
</tr>
<tr>
<td>Acyclic</td>
</tr>
<tr>
<td>Hierarchic</td>
</tr>
<tr>
<td>Star</td>
</tr>
<tr>
<td>Arbitrary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td>--- And ---</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td>High-volume</td>
<td>Broadcast</td>
<td>Run-time</td>
</tr>
<tr>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control/data interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same shape?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
</tbody>
</table>

Figure 3.2: An empty style typing report.
products and services, organizations are not that eager to expose this part of their business. Furthermore, the most interesting evaluations would be those of ‘when things break down’, but no organization feels like connecting their names to recent, failed projects, especially in a way that points exactly where things went wrong, and the impact this had on the quality of the resulting product.

### 3.3.4 In open source projects

Sourceforge and ObjectWeb provide access to some of the most prominent open source projects currently available. Although open source projects are usually a combined effort, in which architecture is a secondary concern, there are some notable exceptions. These projects provide freely available documents on their architecture, philosophy, mechanisms, etc., for example in developer guides. These documents reflect some of the key concerns of the project, allowing some sort of interpretation towards the most prominent quality factors.

So, it is possible to find out the most prominent concerns, but it is hard to find out the ‘values’ for the quality factors. Main sources for these evaluations can be either evaluations of the resulting project, bugtrackers or mailing lists. The main objection against all of these sources is the amount of fragmentation: reviews generally focus on a ‘feeling’, usually caused more by UI design than by anything else, and bugtrackers and mailing lists rarely discuss more than ‘matters at hand’.

### 3.3.5 Contacting open source administrators

In a last attempt to collect information that can help tune the model, administrators of prominent open source projects were contacted with rather direct questions about the product’s architecture and its qualities.

Although the response rate was rather high, some 20%, the responses only scratched the surface, stating general good advice about the ‘right’ way to create software and the importance of modularization. Still, these are smart, busy people, who for some reason took the time to put this into writing. This most definitely means something.

As shown in Appendix E, these answers have had an impact on the views on software architecture in this project. After understanding that I may not be able to come up with a ‘smooth’ model of software architecture and its qualities, I asked the open source contacts for their views on this problem. Their responses confirmed my views, as did the second discussion session (see Section 2.3.2).

### 3.3.6 Conclusions

As the approaches before have shown, it is currently not possible to create a defendable link between concepts and quality factors. However, this does not mean it is not possible to find this; it is just not possible to use ‘shortcuts’.

Section 3.5 will show recommendations for using this model to end up with a theory that shows the link. Without going into too much detail, it can be stated this requires too much effort for this graduation project.
3.4 Experience

The previous section has shown that shortcutting the model in Section 3.1 cannot be done. Still, I believe the model is a good framework for typing and comparing styles, and it would be nice to give at least an informal proof of its correctness.

This can be established by showing the basic ideas of the model conform to practice. To do so, I asked architects within Luminis to come up with examples of projects from their past which show this link.

Note that this does not actually prove anything; only a large-scale investigation can do that. The cases are based on projects done by Luminis architects, and have therefore been anonimized. Where possible, a typing report (see Section 3.2.2) has been included.

3.4.1 Case 1: distributed subscription

The stage

The system under consideration consists of numerous (like, several hundreds) of devices detecting events from the domain, and a single central authority. All elements are connected using a network. The central authority (1) needs to know about all the events and (2) maintains a database, of which each device should have its relevant portion.

The network is dynamic: devices may come and go regularly (though not constantly). Devices can act on events that happen to their colleagues.

The choices

Given the stage above, two major architectural choices have been made.

- To abstract away from the network’s peculiarities, a distributed service oriented architecture (SOA) has been chosen.
- Events propagate throughout the system in a subscription-like way; hence, inversion of control.

The results

The SOA technology takes away ‘worries’ about the network, making the system rather robust (external property). Furthermore, the inversion of control has led to a situation in which new devices can be added easily (internal property), as well as adding new behavior to devices by simply subscribing to the interesting events of others (internal property). Figure 3.3 shows the partial typing report for this case, based on the information in this section.

3.4.2 Case 2: Paints, scales and automobiles

The stage

This case considers a large manufacturer of paint for automobile body shops. As a service to their customers, and for reasons of marketing, they want to provide their customers with a computerized system that assists in mixing a matching paint for a given vehicle.

Paint mixing is a process consisting of various steps, summed up in formulas. Formulas can be developed by the manufacturer, based on research in matching the paint of new vehicles,
<table>
<thead>
<tr>
<th>Internal factors</th>
<th>Maintainability</th>
<th>Portability</th>
<th>External factors</th>
<th>Reliability</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzability</td>
<td>Desirable x</td>
<td>Undesirable</td>
<td>Installability</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
<tr>
<td>Changeability</td>
<td>Desirable</td>
<td>Undesirable</td>
<td>Replaceability</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
<tr>
<td>Testability</td>
<td>Desirable x</td>
<td>Undesirable</td>
<td>Maturity</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fault tolerance</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recoverability</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time behavior</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Resource behaviour</td>
<td>Desirable x</td>
<td>Undesirable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical factors</th>
<th>Components</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Devices</td>
<td>Service communication</td>
</tr>
<tr>
<td></td>
<td>Central authority</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control issues</th>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Hierarchic</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td></td>
<td>Star</td>
<td>Asynchron.</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>Arbitrary</td>
<td>Opportunis.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data issues</th>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear</td>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td>Acyclic</td>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Hierarchic</td>
<td>--- And ---</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td></td>
<td>Star</td>
<td>High-volume</td>
<td>Broadcast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arbitrary</td>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control/data interaction</th>
<th>Same shape?</th>
<th>Same direction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>x No</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Partial typing report for case 1.
or be created by shop employees in search of a matching paint for a vehicle in the shop, with a possibly discolored finish.

The mixing process is based around a scale, which is used to weigh the amount of a specific compound added to the mix. Since regular computer systems are not allowed in the mixing area (due to risks of explosion), only certified scales are allowed, while recipes are usually more complicated than can reasonably be remembered. Virtually all scales have some form of interface, be it a numerical display, or a more sophisticated graphical display with soft buttons; this allows guiding of the mixing process, even adjusting the process to deviations in the weighing process. Since the scales do not have processing power of their own, all ‘intelligence’ should be in a computer system. They usually have an RS-232 interface to control the display and get values. In larger shops with multiple scales, a single computer can control more than one scale.

With new paint colors being ‘discovered’ regularly, it is important to keep systems in the field up to date about this. Furthermore, as new types of scales enter the market, the software should be updated to accommodate for this. For marketing purposes, branding and licensing of functionality should be possible, as is gathering management data (e.g., typical use of the systems, types of compound being used more or less, etc).

In the end, the process the system should support is (1) the distribution of new formulas by the manufacturer, (2) selection of a formula using a computer-based application, (3) guiding the execution of the formula using the scale’s limited interface, and (4) gather management information for the manufacturer about the use of the system.

The choices

Three main qualities have been defined,

- *Device independence* With the variety of different scales with different input/output methods, it should be possible to make use of each one’s specific capabilities. Still, it should be possible to easily add new types, so the mixing process should be independent from the actual device.

- *(Remote) manageability* Since the system has a number of more-or-less variable components (e.g., drivers for scales, new formulas, usage information), support should be included for updating and extracting this information without requiring structural changes.

- *Persistency* With many steps in the chain (the producer, the shop computer, and the scale), data loss is to be avoided.

Given the ‘stage’ above, a services paradigm seems a logical choice. However, team expertise required a component-based approach, doing away with the unpredictability services live in.

Within this component paradigm, two important concepts were used.

- Device independence dictated a decoupling of not only the domain knowledge and applications, but also of applications and the user interface. The application is an ‘intermediate layer’, describing the mixing process, tailored to the capabilities of the scale, not necessarily the type of interface. (Capabilities include e.g. the possibility for user interaction.)
Note that the computer’s UI is also decoupled from the application: this satisfies the branding requirement.

- Management information can come from anywhere in the system, but should not interfere with the structure of the system. Therefore, it has been decided to use aspect orientation to develop code that extracts management information.

This leads to two discernable structures in the system.

- When regarding the manufacturer’s back-end system, the shop computers and the scale, we end up with a star-shaped configuration, communicating using a networked infrastructure (see Figure 3.4).

- The software running on the shop computer is a layered system, decoupling domain knowledge, applications, and UI.

The results

The decoupling of domain knowledge and representation had several nice consequences, being a desirable installability and replaceability; installability is also influenced by the device independence, which is also responsible for the good resource behavior (i.e., available resources are used to their fullest extent, but the system can ‘scale back’ when required).

The three-tiered makeup of the system creates some basic fault tolerance. Choosing persistency as a key quality replaces the fault tolerance with a good recoverability.

Due to the provisions for remote management in aspects, analyzability of the system is good. Although these same mechanisms can be used for testing, no explicit support for testing has been included.

3.4.3 Conclusion

Above, two cases of different levels of detail have been presented. Judging from these, the model is fairly usable for typing practical systems, and the generalization of this information to styles seems promising.

In creating the description of the systems, I have simply noted down what their architects have told me. In typing the systems, I have provided some help in clarifying the ISO factors used for this.

Figures 3.3 and 3.4 show some ‘gaps’ in typing: some factors are simply of no real concern to a system.

These cases have the link ‘near the surface’. This could be a coincidence, because I asked for it; in other projects this link may be more ‘buried’, making typing harder. On the other hand, it could also be a sign of a good architecture that this link is readily ‘pointable’; I have no means for testing that now, but it is a nice thing for future research.

3.5 Conclusions

This chapter has presented the typing report in Figure 3.2, together with a discussion of its contents and experiences in using it.
### Internal factors

<table>
<thead>
<tr>
<th></th>
<th>Maintainability</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzability</td>
<td>Desirable</td>
<td>Undesirable</td>
</tr>
<tr>
<td>Changeability</td>
<td>Desirable</td>
<td>Undesirable</td>
</tr>
<tr>
<td>Testability</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
<tr>
<td>Portability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptableity</td>
<td>Desirable</td>
<td>Undesirable</td>
</tr>
<tr>
<td>Installability</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
<tr>
<td>Replaceability</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
</tbody>
</table>

### External factors

<table>
<thead>
<tr>
<th></th>
<th>Reliability</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Maturity</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
<tr>
<td>Recoverability</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time behavior</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
<tr>
<td>Resource behavior</td>
<td>Desirable</td>
<td>Unsatisfactory</td>
</tr>
</tbody>
</table>

### Physical factors

<table>
<thead>
<tr>
<th>Components</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>o Back-end</td>
<td>o Network communication</td>
</tr>
<tr>
<td>o Shop computer</td>
<td>o Serial port communication</td>
</tr>
<tr>
<td>o Scale</td>
<td>o</td>
</tr>
</tbody>
</table>

### Control issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>Asynchron.</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitary</td>
<td>Opportunis.</td>
<td></td>
</tr>
</tbody>
</table>

### Data issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>And ---</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>High-volume</td>
<td>Broadcast</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitary</td>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

### Control/data interaction

<table>
<thead>
<tr>
<th>Same shape?</th>
<th>Same direction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 3.4: Partial typing report for case 2.
To complete this process of tuning, a great amount of data is needed. From current observations, the best method seems to be interviewing practitioners: in a single interview of one hour, I get more information then I can ever gather from documentation. Furthermore, documentation can be deliberately sketchy on important issues. One result can be a more usable set of ‘inhabitants’ of the model.

Gathering this data, picking the most prominent architecture style for each, results in a catalog of architecture styles with non-functional properties. Once this catalog is available, it is likely that general rules can be distilled, showing the link between physical properties and internal-, external- and architectural factors; these represent the link this research was looking for.
Chapter 4

An adaptive way of architecting

4.1 Introduction

Section 1.2 introduced six categories of adaptivity. It also gives an idea of the kind of adaptivity we are looking for,

“(…) systems that can adapt themselves to changing parameters in their environment, with given parameter ranges and given autonomy.”

Categories 0, 0+ and 1 require human intervention, and are therefore ignored. Category 4 has been deemed unattainable, and will not be considered either.

This leaves us with the two interesting categories, 2 and 3. The difference between these is the environment from the definition; 2 adapts to itself, whereas 3 adapts to the outside world. This also implies a different view of the system: category 2 systems are regarded as a collection of cooperating components, while category 3 is a ‘black box’ in a changing environment.

Section 4.2 will show an introductory example of an adaptive system with computer network routing, and distills some concepts which seem to enable adaptivity. Section 4.3 will show which of the properties in the typing report (see Section 3.2.2) have a link with adaptive behavior, after which Section 4.4 applies this to the styles described in Appendix A. This chapter is concluded with an interpretation of important adaptivity-inducing concepts, styles best suited for adaptivity, and a positioning of ‘adaptivity’ in the wealth of terms related to software architecture in Section 4.5.

4.2 Some intuition

Given the definition of Section 1.2 and its resulting categorization, we can develop some intuition about the important aspects for adaptivity. We will expand this intuition using an example, which will be used to find important concepts.

4.2.1 Example: IP routing

IP networks are made up of multiple networks, connected to each other by routers; each system is connected to exactly one router, or can be a router itself. Local packets, intended for systems connected to the same router, can be delivered directly. To send a packet to a
system connected to another router, i.e., in another network, the packet is sent to the router for handling.

Each router is connected to one or more other routers. To illustrate the adaptive properties of this network, we will use the RIP (Routing Information Protocol), the initial routing protocol used in ARPA-NET. Each router maintains a table of all other routers, notes how many steps it takes to reach it, and which is the first step toward the destination. Every 30 seconds, each router advertises its host table to its direct neighbors; upon receiving this advertisement, a router knows that each host in that table can be reached in one more step than advertised, and maintains shortest routes in its own table.

As an example of this process, consider the network shown in Figure 4.1.

Each of the nodes represents an IP router, possibly with a network behind it; this is of no interest for now. RIP begins by identifying its neighbors, and advertising who it can reach in a single step. For example, A advertises ‘I can reach B, C and D in one step’. C receives this, and in the next round advertises ‘I can reach A and E in one step, and B in two’. After 3 round, this converges to the following set of routing tables.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dst</strong></td>
<td><strong>stp</strong></td>
<td><strong>via</strong></td>
<td><strong>dst</strong></td>
<td><strong>stp</strong></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
<td><strong>A</strong></td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
<td><strong>C</strong></td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>D</td>
<td><strong>D</strong></td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>E</td>
<td><strong>E</strong></td>
<td>3</td>
</tr>
</tbody>
</table>

Line 4 for node A reads ‘Node E can be reached in 2 steps by C’. Now, suppose the connection between C and E is broken. E will start advertising C as reachable in 16 steps (one over the maximum count for RIP), and vice versa. A receives this update from C, and considers its options: D claims to reach E in one step, so A can now reach E in two steps. C receives the update from A, and knows it can now reach E in three steps. The routing table stabilizes as follows, in which the changed elements have been underlined.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dst</strong></td>
<td><strong>stp</strong></td>
<td><strong>via</strong></td>
<td><strong>dst</strong></td>
<td><strong>stp</strong></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
<td><strong>A</strong></td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
<td><strong>C</strong></td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>D</td>
<td><strong>D</strong></td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>D</td>
<td><strong>E</strong></td>
<td>3</td>
</tr>
</tbody>
</table>

Note that for B nothing has changed; the failure has been solved locally.
This example hints a number of important properties, for a system that displays category 2 adaptivity.

- **Emergence**/ None of the routing elements knows all routes, but together, the network has all necessary knowledge. From the rather simple rules the components follow, a more complex behavior emerges; this process is known as emergence.

- **Redundancy**/ Knowledge about reachability is spread throughout the network. If a single router goes down, its knowledge can be reconstructed from that of its neighbors. Furthermore, the nature of the RIP protocol makes that a router can be plugged in place of another one without any configuration; its neighbors will tell it everything it needs to know, simply by doing their jobs.

- **Decoupling**/ Notice how the influence of the failure stops at A: only storing necessary local information prevents the propagation of events through the entire system.

It should be noted that some concepts that seem useful for adaptive behavior have not been shown in this example.

- **Diagnosis**/ The adaptive properties and ‘routing around’ problems are side effects of the core activities of the protocols. No doubt, they have been engineered in exactly this way, but there seem to be no specific rules that handle failures. In the network above, the network is not able to diagnose itself for problems (modern extension can do this, however).

### 4.2.2 Resources

IP networks adapt to changes in ‘reachability’; this can be seen as a resource. When we look at other resources in the realm of computing, some other concepts emerge.

- **Networking**/ Ever more equipment is becoming networked. The mobile realm is an especially interesting part in this change towards networked equipment. With cheaper communication hardware coming available (a consumer-grade Bluetooth module, when bought in bulk, runs less than $3, with low-power ZigBee promising to break the $1 barrier), more equipment is becoming networked. The grade in which the devices make use of these capabilities varies, but nevertheless shows a trend toward the ‘fully networked’ state [LMPF01].

This means that networked services are becoming a resource. The fridge might stock up ice cubes when it knows a warm day is ahead, and the airconditioning might cease operation earlier if it knows outside temperature will drop shortly. The weather is not the only application: my harddisk recorder may start recording my favorite show when it notices my instant messaging application is still set to ‘@work’.

Network connectivity is not something that can always be relied on. The local network connection may be down, or the remote service provider may malfunction. In these cases, the appliance should continue its job normally, and try to get the information it needs at another moment. An all-encompassing term for handling this could be service awareness.
• [Processing and multi-core] Processes have long since shared processing power on a single machine. Traditional processes will try to claim the amount of power they need; if more power is claimed than the system has available, everything will just go slower. Two things can be done to handle this. First, add more processing power. As we are reaching the boundaries of the power of a single processing core, this could mean that more processors are added. For processes to take advantages of this so-called multi-core setup, they should be parallelizable. This can be extended to a networked environment, where parts of the process might spread out over multiple machines.

Another approach is to make processes scale their service level according to the amount of processing power available; for instance, a web server can, when exceeding its amount of processing power or network bandwidth, switch to a text-only mode. This has been given the term sufficient correctness [Sha00]: do the best you can in the given circumstances, instead of striving for perfect operation and breaking on the first sign of trouble.

4.2.3 Conclusion

We end up with a number of concepts that seem beneficial for adaptive processes of both categories 2 and 3.

• [Emergence] With relatively simple components working together, we can end up with complex behavior. This seems to be suited for category 2 adaptivity.

• [Redundancy] Prepare for components to break, but make it in such a way, that other components can take over the broken one’s tasks. Furthermore, these provisions make replacing components easy, making it suited for category 2 adaptivity.

• [Decoupling] Making a system able to dampen the effects of changes by localizing them, is beneficial for both category 2 and 3.

• [Self-diagnosis] Provisions that allow the system to assess its own operation and handle accordingly seems beneficial for category 2.

• [Service awareness] Creating components in such a way that they can handle the failure of other services gracefully, while taking advantage of them when available is useful for both categories 2 and 3.

• [Parallizability] Processes that can be broken up into discrete parts allow the use of multi-core systems, or even the infrastructure available at the edges of a network; this is typically useful for category 2 adaptivity.

• [Scalability] Taking the availability of resources as an input value for processes makes them respond to changes in that; both categories 2 and 3 benefit from that.

4.3 Adaptive concepts

Section 3.2.2 shows the typing report for software architecture styles. In this section, we will discuss the applicability of each of the properties in relation to adaptivity. This discussion is combined into a ‘grading system’ for measuring the adaptive properties of styles.
4.3.1 From Physical properties

Among the physical properties, there should be some indications about which are better suited for some kinds of adaptivity than others. We will stick to the same order as Section 3.2.1.

- [Components and connectors] Although component and connector names can have some connotation of adaptivity (e.g., ReconfigurationManager sounds more adaptive than MonolithBlock), they do not seem linkable to types of adaptivity.

- [Control topology] Prescribing a flow of control seems to tie down the possibilities for adaptivity: this implies an order in which components get activated. This is especially true for Linear and Acyclic topologies. However, their simplicity does make them suitable for category 1 adaptivity.

Hierarchic may be able to have some sort of resilience, isolating the effect of changes. A Star topology implies some form of replication, with many components located around a core. While the system may be able to recover from breaking components, it does have a single point of failure, as does the hierarchic approach. The isolation of failure implies a category 1 adaptivity, while supporting category 2 adaptivity when including special provisions.

The most free control topology is Arbitrary, allowing any shape. Although this does not say anything about the structure, leaving all possibilities open does have a ring of autonomy.

- [Synchronicity] Both lockstep variants do not come very far; category 2 adaptivity seems to be all but ruled out.

Synchronous has a sound of control to it, but also causes visions of deadlock: waiting for colleagues that may have died or just moved.

Both Asynchronous and Opportunistic seem to have the amount of detachment that allows the handling of changing circumstances, supporting category 3.

- [Control binding time] It is obvious that later binding can handle changing makeup of the system itself, so both categories 2 and 3 adaptivity benefit from late binding.

- [Data topology] Although the shapes are equal to those of Control topology, their influence on adaptive properties seem to be very different. A strict data topology seems beneficial: a component knows exactly where to expect data to come from, and can act accordingly when something seems to be going wrong, whereas a strict control topology is more likely to cause component starvation.

In the end, data topology seems to be a non-issue in adaptivity.

- [Continuity] Continuous communication seems to be bad for adaptive behavior, since dependencies on other components are high.

The decoupling reached by Sporadic communication leaves more autonomy to components, creating an environment in which components have to take care of themselves. This could support category 3 adaptivity.
The volume of information transferred seems to have smaller influence. A large amount of data to be transferred will take longer, causing a bigger window of opportunity for errors in the transmission. Therefore, \textit{low-volume} seems to be better suited to adaptive system.

- \textit{[Mode]} Whether communicating with a single entity or a group, the mode of communication does not seem to have much influence on the adaptive properties of the system.

- \textit{[Data binding time]} Like in control binding time, later binding seems to be beneficial. For category 2 adaptivity, delaying the decision of who to communicate with is a good thing. For category 3 adaptivity, it seems to be a little more relaxed, since the internals of the system are not under consideration.

- \textit{[Control/data interaction]} It seems arguable that simplicity, equal shape and direction, is beneficial, but this influence is handled by the earlier factors.

### 4.3.2 From External factors

Using the intuition of objects that adapt subjects to an environment, there should be some currently defined quality factors that are linked to adaptivity.

- \textit{[Maturity]} Perhaps evolving systems could benefit from maturity, but a system that is intended to handle its own problems does not seem to be benefited by it.

- \textit{[Fault tolerance]} ISO defines this as resilience to bad use of the interface. On both the component and the system level, this seems beneficial. It helps in handling gone-bad colleagues (category 2), and faulty requests from the environment (category 3).

- \textit{[Recoverability]} Recovering from errors sounds good. In fact, ISO uses the term ‘a specified level of performance’, creating opportunities for homeostatic properties, implying category 3 adaptivity.

- \textit{[Time/resource behavior]} Efficiency and adaptivity seem to have little in common. One of the enablers is redundancy, which likely comes with a performance penalty.

### 4.3.3 From Internal factors

- \textit{[Analyzeability]} This seems to be mainly related to category 0+ and 1 adaptivity, being tailored towards human use. However, a mechanism that monitors the health of components can be used by by the system itself too; this is limited to category 2 adaptivity.

- \textit{[Changeability]} Provisions that allow a human engineer to change a system are beneficial for at least category 0+.

- \textit{[Stability]} ISO defines this as preventing unexpected effects \textit{from modifications}; modification by an engineer are not under consideration.

- \textit{[Testability]} Although provisions intended for testability may help the system determine its state, this seems to be a non-issue.
Table 4.1: Physical properties and adaptivity categories. Number of pluses indicate level of support.

<table>
<thead>
<tr>
<th>Property</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear/acyclic control topology</td>
<td>++</td>
</tr>
<tr>
<td>Hierarchical/star control topology</td>
<td>++</td>
</tr>
<tr>
<td>Arbitrary control topology</td>
<td>+</td>
</tr>
<tr>
<td>Lockstep synchronicity</td>
<td>++</td>
</tr>
<tr>
<td>Synchronous control</td>
<td>+</td>
</tr>
<tr>
<td>Asynchronous control</td>
<td>++</td>
</tr>
<tr>
<td>Invocation control binding</td>
<td>+</td>
</tr>
<tr>
<td>Run-time control binding</td>
<td>++</td>
</tr>
<tr>
<td>Data topology</td>
<td></td>
</tr>
<tr>
<td>Continuous communication</td>
<td>++</td>
</tr>
<tr>
<td>Sporadic communication</td>
<td>++</td>
</tr>
<tr>
<td>High-volume communication</td>
<td>++</td>
</tr>
<tr>
<td>Low-volume communication</td>
<td>++</td>
</tr>
<tr>
<td>Communication mode</td>
<td></td>
</tr>
<tr>
<td>Invocation data binding</td>
<td>+</td>
</tr>
<tr>
<td>Run-time data binding</td>
<td>+</td>
</tr>
</tbody>
</table>

- **[Adaptability]** Category 1 systems are bound to have a high adaptability; that is what they are for. Having provisions to be adaptable (adapters, for instance) can also be used by the system itself to adapt to the environment, implying category 3 adaptivity; there does not seem to be much benefit for category 2.

- **[Installability]** This is a kind of adaptivity to the environment, but in a static way. Category 1 systems benefit from this.

- **[Replaceability]** ISO means by this the replacing of another system by this one; this means the system will have to (externally) behave like the other one. Although as such this does not imply adaptivity, it can very well benefit from the same provisions that make a system adaptive.

4.3.4 Conclusion

Tables 4.1 and 4.2 show the information from Sections 4.3.1 through 4.3.3. The number of pluses indicate the ‘level’ of support, with ++ being the maximum attainable.

These tables show that one of the assumption from the beginning of this section, that categories 2 and 3 are very different because of their focus, is not entirely true. The differences are discussed below.

- **[Hierarchical/star topology]** A hierarchic topology allows for good localization of events, but is not very flexible.

- **[Sporadic communication]** Only communicating when it is really necessary is a good thing for category 2 systems, so limiting the moments that ‘stuff can break’. For category
<table>
<thead>
<tr>
<th>Factor</th>
<th>Category</th>
<th>0+</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maturity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault tolerance</td>
<td></td>
<td>++</td>
<td>++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recoverability</td>
<td></td>
<td>++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time/resource behavior</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyzability</td>
<td></td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Changeability</td>
<td></td>
<td>++</td>
<td>++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptability</td>
<td></td>
<td>++</td>
<td>+</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>Installability</td>
<td></td>
<td>++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replaceability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Quality factors and adaptivity categories. Number of pluses indicate level of support.

3 systems, this is not of much concern, since internal communication is not considered.

- [Late data binding] Selecting communication partners as late as possible is beneficial for both categories, but more so for category 3: it implies autonomy.
- [Recoverability] The ability to restore service and data after a failure is well-suited for systems that have to adapt to their environment; in the component-level category 2 systems, this is not as important.

### 4.4 Adaptive styles

In Appendix A, 11 software architecture styles are shown. Using the ‘grading’ from Section 4.3, we quantify the possibilities for each to support adaptive behavior of a given category by counting the ‘pluses’ they earn. We will handle the notable elements from the resulting grading, and compare this with the intuition of adaptivity we have with some of these styles.

#### 4.4.1 Styles

For each style from Appendix A, we will handle the adaptive properties below. The ‘scores’ are a direct result from combining Table 4.1 with the style’s typing report, and will therefore not be elaborated here. If a style supports multiple answers, we will use the ‘best attainable’. The results are shown in Tables 4.3 and 4.4, for category 2 and 3 adaptivity, respectively.

Some peculiarities catch the eye here, apart from the ‘good and bad’ styles, which will be handled in the next section.

- Like in Section 4.3.4, there does not seem to be much difference between category 2 and category 3 adaptivity; the maximum difference is 1.
- Contrary to intuition, it seems that synchronicity and continuity are not of much influence. This could be due to the generality of the styles describes, which ‘support’ many different modes of communication.
<table>
<thead>
<tr>
<th>Style</th>
<th>Control</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Topology</td>
<td>Synchronicity</td>
</tr>
<tr>
<td>Abstract data</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Event based</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Layered</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Blackboard</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Interacting processes</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Pipe and filter</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Peer-to-peer</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>SOA</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>C2</td>
<td>++&quot;a&quot;</td>
<td>++</td>
</tr>
<tr>
<td>Publish/Subscribe</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>REST</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

*C2 uses the ‘layer-ish’ topology, which is not part of Table 4.1. Still, it gets ‘++’ for its flexibility, see Section A.9.*

Table 4.3: Category 2 adaptivity, per style.

<table>
<thead>
<tr>
<th>Style</th>
<th>Control</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Topology</td>
<td>Synchronicity</td>
</tr>
<tr>
<td>Abstract data</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Event based</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Layered</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Blackboard</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Interacting processes</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Pipe and filter</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Peer-to-peer</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>SOA</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>C2</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Publish/Subscribe</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>REST</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

Table 4.4: Category 3 adaptivity, per style.
• The most influential elements seem to be topology and binding time.

4.4.2 Conclusion

For both categories, three styles reach the maximum score of 11, being Event based, Peer-to-peer and SOA. Neither of these are a big surprise,

• [Event based] With its inversion of control, a system built in an event based style is highly resilient to breaking elements, and new elements being added.

• [Peer-to-peer] Praised in Section A.7 for its advantages of “easy configuration, scalability and robustness,” peer-to-peer is one of the most adaptive, and certainly most scalable, styles described. It shows most of the intuitive concepts from Section 4.3, emergence, redundancy, decoupling, parallelizability and scalability.

• [SOA] With its inherent notion of ‘use services when available, cope without them when not’, SOA combines resilience with rich features.

A notable runner up is Publish/Subscribe, being a distributed and more resilient variant of Event based; its fixed topology, though with measures for handling damage, costs it one +. Two styles seem to be least suited for adaptivity.

• [Abstract data] Although it is the basis of the majority of modern software, object orientation as such lacks adaptive concepts. However, the possibilities for sound code organization allow the creation of highly adaptive software. Abstract data is probably just too basic.

• [Pipe and filter] When basic pipes are agreed upon, pipe and filter is a style that allows easy reorganization of a system. However, its strict structure and strong coupling between elements when the system is operating rules out adaptive behavior.

4.5 Conclusion

Adaptivity keeps showing up throughout various styles, and seems to be influenced most by topology and binding time. The best position of adaptivity is a quality factor, influenced by the same concepts that other quality factors are. Adaptivity is not a concept as such, but is brought along by cleverly combining others, as shown below. Neither does it seem to be a real style: adaptivity shows up in many different styles, in different shapes.

4.5.1 The concepts

The intuitive concepts from Section 4.2 have, all but one, shown their applicability in this chapter.

Intuitively, self-diagnosis seems to be an important factor in adaptivity. In the examples we have seen in this chapter, it has not shown itself: it rather seems that well-designed systems do not need to diagnose themselves, but handle adapting to changing circumstances as part of their core business.

We have not had opportunities to handle the quality factors from Sections 4.3.2 and 4.3.3; this is due to the fact that we do not have complete information about these factors for
each style. Testing the proposed links between adaptivity and other quality factors can be interesting future research, however.

4.5.2 The styles

From the styles in Section 4.4, three seem to be very promising for creating adaptive systems, being Peer-to-peer, SOA and Publish/Subscribe (it seems that event based, although scoring well, is too general). Next to scoring well in Tables 4.3 and 4.4, they seem to incorporate the majority of the concepts from the previous section.

More basic or more strict styles, like Abstract data and Pipe and filter seem to be less suited to handling adaptivity.

It seems that the styles we see as beneficial have found a good balance between the concreteness required for displaying measurable properties, and the flexibility required for showing adaptive behavior.

4.5.3 General

Concluding, the ‘rules to follow’ in creating a system with adaptive properties are best summed up as the intuitive concepts from Section 4.2, emergence, redundancy, decoupling, service awareness, parallelizability and scalability.

Depending on the peculiarities of the system, a style of choice can be Peer-to-peer, SOA or Publish/Subscribe.

Adaptive categories 2 and 3 seem to blur in this chapter. The difference in focus described in Section 4.1 may not be so clear after all: category 2 considers the internals of the system, while 3 looks to the outside, but 2 allows relevant elements of the environment to be modeled and included into the system, becoming a core element of the system. That way, instead of having to ‘hope’ for the best, at least something can be said about the environment. This does, however, mean that boundaries of the system are blurring. The amount of adaptivity can then be defined as the amount of environment that is considered in the system’s design.

The complexity in software creation induced by adaptivity calls for it to be a major concern in software engineering and architecture. A system has to be designed with adaptivity in mind. To conclude this chapter, I would like to revisit a quote by Andrew S. Tanenbaum,

“Security, like correctness, is not an add-on feature.”

which I believe to be ever so true about adaptivity.
Chapter 5

Conclusions

Based on the research in this document, this chapter will present an overview of the most prominent results and contributions of this research, and sketch how this research can be used for future advancements. Finally, the research goals are evaluated.

5.1 On architecture and styles

We have presented a view of software architecture in Chapter 1, suitable for reasoning about quality factors in general, and adaptivity in particular. This view has been created during the course of this project.

This view on software architecture has been used to develop a typing report, intended to allow linking quality factors to architectures. Attempts to ‘shortcut’ this model to find these links have failed, but two practical cases have shown the applicability for comparing properties of systems. Extension to styles seems promising.

Experiences in using the report have shown the current properties to be sufficient, but additional concepts and quality factors need to be developed.

5.2 On adaptivity

Adaptivity is a quality exhibited by a system, and is not an architecture style or a concept, according to the definitions used in this project. We have defined adaptivity as the capability to make oneself suitable to changing circumstances.

We have identified six important concepts that can help bring adaptivity in a system about, being Emergence, Redundancy, Decoupling, Service awareness, Parallelizability and Scalability. By judging the adaptive capabilities of the elements of the typing report, we have identified three architecture styles with good potential for adaptivity, being Peer-to-peer, SOA and Publish/Subscribe.

The initial categorization of interesting adaptive systems as categories 2 and 3 has blurred. Adaptivity has become a sliding scale, in which the amount of adaptivity a system exhibits is determined by the amount of environment that is internalized in the system.

Systems intended to show adaptive properties should be engineered as such from the very beginning, adaptivity cannot be added later. Systems can only adapt to predetermined changes in availability of resources; these resources should be modeled in some way in the
system, so adapting to change is a core activity of the system, not an additional set of rules for special cases.

5.3 Future research

Based on the research presented in this document, two intertwined directions of future research are available.

A catalog of styles Using the typing report, systems can be typed in a comparable and, not unimportant, anonymized way.

Collecting typing reports from many projects can give rise to

- A catalog of software architecture styles, categorized by the quality factors they exhibit. This catalog is useful for selecting the architecture style of future projects, in which given quality factors have been identified as being important. Although this can never replace the architect’s intuition, it can help in filtering the wealth of available styles for wanted properties.

- By finding patterns in the collection of typing reports, links can be identified between concepts and quality factors. These links can have the form “Styles with concept \( x \) have a more-than-average chance of having a desirable quality factor \( y \)” or “When quality factor \( x \) is present in a system, it is likely that quality factor \( y \) is undesirable.”

Quality factors The treatise of adaptivity in Chapter 4 shows a way of investigating the properties of a desired factor. We have created an intuition of the link between concepts and adaptivity, which can be proven or invalidated by means of the typing process described above.

This same approach can be applied to other desired properties as well. When a property has been selected,

1. determine a definition of the property,
2. build an intuition of important concepts for this property,
3. grade the concepts in the typing reports for expected support of the property,
4. select architecture styles based on this grading, and check the conformance of this selection with intuition.

Creating treatises of desirable properties in this way results in new quality factors for use in the typing report, making it ever more complete; the new factors can now be used to select the styles that support them.

The subject of adaptivity has been handled extensively in this document using the tools above. Therefore, we will not provide recommendation for future research in adaptivity.
5.4 Research goals

The project started out with the main research question,

“How to integrate adaptivity into software architecture?”

This question has been answered. We have shown that adaptivity is a quality factor, established by a set of six concepts and the guideline ‘model the environment’.

The secondary research question

“What are quality factors of architectural styles?”

has been answered by the view on software architecture used throughout the project, which has been validated simply by using it to answer the main question. Quality factors are properties, exhibited by a system, and are not directly engineerable, but rather the result of cleverly combining concepts in an architecture. Hence, the original assumption that quality factors and non-functional properties are very closely related holds true.

Concluding, adaptivity can be very well be incorporated on an architectural level, but requires a focus on it from the very start.
Bibliography


[ISO] ISO/IEC 9126. Restricted standard; definitions reprinted in [KG04].


Appendix A

Detailed style descriptions

A.1 Abstract Data

Also known as object oriented and modular, the abstract data style describes a way to hide representation from the outside world, i.e., to abstract away from it. Starting with modularization in [Dij68], this has been followed by criteria for modularizing large problems in [Par72], with one of the first modern representations presented in [Boo82] (although the Ada-heritage of this method has been described earlier, this paper gives the nicest overview).

Although object orientation and modular or component-based design are modern, better known terms, I prefer the term ‘Abstract data’; it covers the subject nicely, and does not have the connotations the other terms have.

The aim of the method is abstraction of data, by encapsulating the data in an object, only allowing interaction by invoking methods. For this purpose, [Boo82] uses a representation that looks suspiciously like modern UML packages, see Figure A.1.

There are many well-known examples of designs including data abstraction. Figure A.2 shows the less well-known example used in [Boo82]: the goal of this system is, on input of a binary tree, to return the number of leaves. This is done by breaking the tree at its root, putting the ‘parts’ on a pile, and recursively picking up a tree, breaking it up, and puttings its parts back. A leaf is a subtree too, so when it is picked up, the counter can be incremented, and that subtree discarded; this procedure is repeated until the pile is empty.

The description from [Boo82] is tailored to Ada’s needs. Figure A.3 shows a more general representation of the abstract data style.

It should be noted that Figure A.3 does not portray a class diagram, but is about objects. As such, advanced object-oriented concepts as inheritance and polymorphism are not part of this style.

Vocabulary

The abstract data representation consists of

- [Object] A logical unit of data and/or computation.

---

1 It is nice to note that Booch regards programs as sequential entities, whereas tasks can be performed in parallel: hence the parallelogram.
2 Another peculiarity of this figure is the notion of interface; an interface is not the ‘full’ set of functionality, but an object has multiple interfaces, each represented by the black lines.
Figure A.1: Representation of data abstraction in [Boo82].

Figure A.2: Leaf counting example from [Boo82]. The arrows indicate the visibility of a package within another.
Figure A.3: Abstract data generalized.

- *[Interface]* The externally visible part of an object, intended for interaction with the object.

- *[Method invocation]* Manner of communication between objects.

**Configuration rules**

Each object has zero or more interfaces. A method invocation connects the body of an object to an interface. One interface may be used by multiple method invocations, one object may have multiple method invocations.

**Semantic interpretation**

A method invocation that connects an object to an interface means that the object can use the services provided by the interface, which in turn will be fulfilled by the object that owns the interface. Depending on the chosen concurrency-model, this may or may not mean a transfer of control.

**Analyses**

As one of the basics of modern software construction, this style benefits from quite some support for analyzing designs, such as metrics on usage and code generation. Tools have been developed to support these analyses.

**Wrapup**

Abstract data is a style that can live on various levels in a system. When used as an overall style in a project, it will show up as the basis for the component interaction. With most modern programming languages being object oriented, data abstraction and information hiding has also become a tool for shaping just about any programming solution.

This versatility has both advantages and disadvantages for its use as a style.

- The paradigm of objects, communicating through method invocation, is well-understood. This means that architectures built using abstracted data are easily understood, well-suited for analysis and typically easily translatable into code.
• As a very basic style, it cannot guide the architect in making choices for modularization and setup of the system. Many styles can be used ‘on top’ of an object oriented system, making its status as a ‘style’ debatable.

### A.2 Event based

Also known as implicit invocation and a wealth of other terms like ‘inversion of control’ and ‘the Hollywood principle’, event based operation is the counterpart of method invocation as used in abstract data. In abstract data, components communicate by invocation of each other’s methods. In implicit invocation, events are announced, allowing other components to react (or ignore them).

In its most basic form, implicit invocation is depicted in Figure A.5. Components can register themselves as being interested in an event, and when the event occurs, all interested components get a notification.
This style is very basic, so many different interpretations exist. To name a few,

- Java uses a mechanism in which the dispatcher is integrated in the generating component; interested parties (objects) implement an interface prescribed by the generator, register themselves, and get a callback when the event occurs.

- A similar, though simpler, callback mechanism is used by operating systems; processes can specify an entry point that gets invoked when a certain condition, detected by the OS, holds. This principle is usually referred to as interrupt handling.

- The Windows message system has a form of implicit invocation: by sending messages either to a specific process or a group of processes, an event is announced, on which the receivers can react.

Whatever the method chosen, a distinctive feature is that components do not need to know each other’s identity at compile time. At run time, new components may be introduced to a system by ‘hooking up’ the correct notifications.

Vocabulary

The vocabulary used for implicit invocation is very diverse, but all incarnations have a form of each of the elements of Figure A.5.

- [Component] An entity of computation, capable of generating and/or responding to events.

- [Event] The announcement of ‘something’.

- [Registration] Indication from a component to the dispatcher that it wants to be notified of a given (set of) event(s). This is also known as an ‘event-method binding’.

- [Dispatcher] Service that handles the notification of interested parties when an event occurs.

Configuration rules

A system has one or more dispatchers, either specific entities for this purpose, or embedded in components. Components can be connected to dispatchers.

Semantic interpretation

Events are generated by components, and get distributed by dispatchers. Only a component which has registered its interest in a given (set of) event(s) will receive notifications when an event enters a dispatcher. Note that this implies a temporal coupling; only events dispatched after the registration of interest can be received by a component.
Physical factors

<table>
<thead>
<tr>
<th>Components</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>o Components</td>
<td>o Events</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>

Control issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>Asynchron.</td>
<td>Run-time</td>
</tr>
</tbody>
</table>

Arbitrary | Opportunis.    |

Data issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>And ---</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>High-volume</td>
<td>Broadcast</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

Control/data interaction

<table>
<thead>
<tr>
<th>Same shape?</th>
<th>Same direction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure A.6: Typing report for event based.

Analyses

In general, event based systems are considered to be hard to reason about or analyse. Formal approaches are all but ruled out because of the combinatorial explosion of possible states; one approach is presented in [GK00], which uses model checking. With appropriate abstractions, a finite state machine can be created from an implicit invocation system. This allows the checking of LTL\(^3\) properties, such analysis of deadlock and progress, and the verification of invariants.

Wrapup

As stated before, event based styles are very basic, providing merely a communication mechanism. This makes the style open for different interpretations and implementations, allowing use in various situations.

Event based systems are flexible, allowing the addition or replacement of components with no impact on the rest of the system. Scalability is limited, however: there is no logical upper limit to the number of components that can register for a given event, causing potential performance problems.

Concluding, implicit invocation is primarily a communication paradigm, making its status as an actual ‘style’ debatable.

---

\(^3\)LTL, or linear temporal logic, is a notation method for describing the state of variables in relation to time. LTL formulas can describe that a certain property will always hold, that one state is always followed by another, etc. LTL formulas are used mainly for processes without a clearly defined end state. See also [http://en.wikipedia.org/wiki/Linear_temporal_logic](http://en.wikipedia.org/wiki/Linear_temporal_logic).
A.3 Layered

The layered approach to software structuring is one of the most intuitive, given the structure of computer systems. Building layers of abstraction on top of more concrete, low-level layers has proved useful since the earliest of operating systems. Still, operating systems are one of the major application areas for layered systems; for an example, see Figure A.7 for the layered structure of Linux, in which the layers are connected by method calls.

Communication systems are another typical area of use for layering, dealing with the hardware responsible for transmission, up to the application which provides the user with services, regardless of underlying implementation. A part of a system, described in a layered fashion, which is concerned with one type of communication is known as a ‘protocol stack’.

Vocabulary

Layered systems have one type of component, and one type of connector.

- [Layer] A layer is a logical unit of operation, defined by the level of abstraction it works on.

- [Border] Layers communicate through their top- and bottom boundaries.

Configuration rules

A system is composed of layer components, such that components share boundaries.

Semantic interpretation

Layers can only communicate with other layers when they are directly adjacent, and only through top-bottom boundaries, not sideways. Services are provided to layers which are above this one, and used from below. This implies that a layer defines the interface on its top side, and uses the interface of its underlying companion.

Figure A.7 hints the possibility of ‘skipping’ intermediate layers. As such, this is not a problem for both the configuration rules (which only require sides to touch) and the interpretation (which allows communication along horizontal boundaries).
Figure A.8: Typing report for layered.

Analyses

The main thing that can be measured in layered systems is, well, the layering itself. The success of the system in achieving the properties implied by using this style, such as good locality of changes, is determined by the strictness of the layering.

Not only the strictness of the layering is a measure for the ‘correctness’ of the division. The number of interconnections between layers that should communicate is another useful indicator. A very high number of interconnections might hint a boundary that has been placed in a wrong spot.

Wrapup

The layered approach is a natural choice for systems in which different levels of abstraction can be distinguished. By limiting communication only to directly adjacent components, changes and replacements have only a local impact. The main disadvantage is the necessity for strict layering to gain these benefits.

Being a little more concrete than abstract data and event based styles, layered systems impose clear rules with clear benefits, providing a guideline for the development of large systems.

A.4 Blackboard

One of the earliest systems that explicitly mentions the term ‘blackboard’ in a shape we would now recognize it was Hearsay-II [EHRLR80], a system intended for speech understanding. Next to the speech understanding system, it “represents (...) a general framework for coordinating independent processes to achieve cooperative problem-solving behavior”, which
is exactly what a blackboard system does. Figure A.9 shows the architecture as used in the project.

The system’s blackboard is a layered one, with the most interpreted information in the uppermost layers. Recognition starts by breaking down a ‘piece’ of speech into its audible components, and subsequently assembling parts into syllables, words and sentences. Since speech can be ambiguous, many hypotheses are created ‘what it could mean’. Useful combinations of information get interpreted, and posted to the blackboard. From this blackboard, other components can formulate new hypotheses, and so on.

In Figure A.9, KS represent Knowledge Sources, units of computation that have the ability to create new hypotheses based on hypotheses from the blackboard. This architecture contains all the classic components that modern blackboard based architectures have. Without calling it an architectural style explicitly, [Nii86] formalizes the blackboard model, stating it is “(. . .) a highly structured, special case of opportunistic problem solving.”.

Vocabulary

The blackboard style consists, as shown in Figure A.9, of two main components.

- [Knowledge sources] These components have the (static) knowledge necessary to solve parts of the problem.
- [Blackboard data structure] This component stores the “problem solving state data”, the pieces of information that initially make up the problem, and get transformed towards a solution.

Configuration rules

One or more knowledge sources get connected to a single blackboard. No direct communication between knowledge sources is allowed. A representation of this is given in Figure A.9.
A.10.

Whereas Figure A.9 shows the controlling components an separate entities, [Nii86] defers the control issues to the implementation.

Semantic interpretation

Knowledge sources can only read and write data on the blackboard they are connected to. The information stored on the blackboard can be retrieved by knowledge sources based on the type of information they want to have.

Analyses

For blackboard systems, no ‘biblical document’ exists that describes the complete style with all the properties that Garlan wishes in a style. Therefore, no analysis techniques have been defined that correspond directly to the style.
Wrapup

Blackboard systems are generally found in the field of artificial intelligence, with a very nice example in [Kei96]. In systems with long-lived data, which is operated upon by specialized and logically disconnected entities, in an order that cannot be determined beforehand, blackboard seems a good choice.

While the blackboard style is extensible due to the makeup of the system (extra functionality can be added by including more knowledge sources), the use of a blackboard introduces a single-point-of-failure. Together with the control problem put forward by [Nii86], this gave rise to a wealth of alternative blackboard approaches, as summed up in [CL92].

A.5 Interacting processes

In the early days of computing, programs were sequential, handling a given task to completion. With the advent of interactive machinery, it became apparent that each part of the system needed a specific piece of software to handle it (drivers); all these pieces would act concurrently within an operating system, and have to communicate to achieve some useful task together. This way, process interaction was born.

It was soon understood that having processes run in parallel has many more advantages. Not only could systems become more responsive, by combining the computing power of multiple units (being processes, processors or systems), they could become more powerful.

Also known as ‘distributed processes’, ‘distributed computing’, ‘multicomputing’ and many other terms with slightly different connotations, all systems built from interacting processing are characterized by independent processes, running in (virtual or actual) parallel, communicating by message passing.

The most well-known descendant of this style is pipe and filter, which is often used as the exemplary style for explaining the style of interacting processes.

Vocabulary

In its basic form, a system of interacting processes consists of two types of elements,

- [Channel] A means for passing messages between processes.

Configuration rules

Processes are connected to one or more channels. Channels can be connected to exactly two, or more than one process (depending on the messaging paradigm).

Semantic interpretation

Processes can interact when there is a channel that connects them.

Analyses

Two basic analysis types can be defined for interacting processes; derived styles may have more.
• \textit{Port compatibility} Systems can have many different types of processes and channels. For well-defined elements, it can be determined whether all processes have the correct role for the channel they are connected to.

• \textit{Deadlock freedom} For elements with clear specification in terms of input and output, a given composition can be analyzed to find potential deadlock situations, i.e., components indefinitely ‘waiting’ for each other.

Wrapup

As another very basic style, interacting processes provides a solid basis for very specific styles, as shown below. Since the 1960’s, work has been done to gain a deep understanding of this style, resulting in numerous papers such as [And91]. This scientific interest, combined with the promise of high-performance and scalable systems, makes this one of the most used style-basics for computer software.

A.6 Pipe and filter

Being based on the style of interacting processes, pipe and filter is the style of choice when explaining architectural styles, since it is easy to visualize (both graphically and mentally). That said, it is very hard to find a good description of this concept; the description below is based mainly on [GS94].

Being one of the first styles identified, it shares most resemblance to classic business processes. The best analogy is clerks handling documents, and passing them to other clerks for another step of transformation. When formalized, this becomes a system based on a number of data transforming entities (the filters), connected to each other by (directed)
Figure A.13: Pipe and filter architecture of an MPEG-2 encoder.

pipes. Basic pipe and filter allows loops and multiple inputs and outputs per filter. A sub-style only allowing sequential composition is known as a pipeline, and is exemplified by the Unix pipeline mechanism.

One of the most natural uses for pipe and filter is digital signal processing. Figure A.13 shows an MPEG-2 encoder, built as a pipe and filter system.

This figure has all necessary elements for a pipe and filter system, being filters, connected by pipes, and in this case including a loop. Intuitively, this paradigm can be extended to include new filters, or the entire system depicted in Figure A.13 can be used as a single filter in a larger system, allowing hierarchical composition.

Vocabulary

The style has two constituents,

- [Pipe] Represented by an arrow, the sole function of a pipe is transport of data. This is usually, though not necessarily, a stream of data.

- [Filter] Filters accept data from pipes, transform it, and output the data to another pipe.

Configuration rules

A filter can have multiple incoming pipes, and multiple outgoing pipes. At least one side of a pipe is connected to a filter.

Semantic interpretation

All data is transported by pipes; data cannot flow from one filter to another without a pipe transporting is. A pipe which has a ‘dangling’ input is known as a source, with a dangling output it becomes a sink. Note that filters do not know who is listening on the other side of the pipe, or even if the data they produce is being used. This implies there is no flow of control along the pipes, each filter is an autonomous entity.

Analyses

Being a highly regular structure, pipe and filter is well suited for various types of analysis, such as deadlock detection and throughput analysis.
### Physical factors

<table>
<thead>
<tr>
<th>Components</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>o Pipe</td>
<td>o Filter</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>

### Control issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>Asynchron.</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>Opportunis.</td>
<td></td>
</tr>
</tbody>
</table>

### Data issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>--- And ---</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>High-volume</td>
<td>Broadcast</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control/data interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same shape?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
</tbody>
</table>

Figure A.14: Typing report for pipe and filter.

## Wrapup

All things considered, pipe and filter is one of the simplest and best understood architectural styles. It allows easy linear and hierarchical composition, as well as basic analyses. When implemented correctly, systems can take advantage of the inherent concurrency implied by the separate filters\(^4\).

## A.7 Peer-to-peer

When abstracting away from the applications that internet-connected systems deliver, all those systems are equal: they all can send and receive datagrams. Since the earliest forms of the internet, its structure has been inherently peer-to-peer: a system of interconnected nodes, each with their own administrative structure, and possibly another network behind it. The distinction in separate roles (such as clients and servers, see also Section A.11) is an artifact of the applications provided, like HTTP, or the notion of web services.

In the last decade, a movement has arisen that wishes to use the resources that live on the ‘edges of the internet’, that is, resources that are available to users wishing to share them. Typical resources are CPU power, storage space, content and network bandwidth. This user involvement cannot work with the typical client/server distinction that is the basis of web applications. As an example of a resource-sharing system (more specifically, file-sharing), consider the Gnutella protocol.

Instead of using the topology of the internet, the Gnutella specification\(^5\) requires each node

\(^4\)A system in which pipes and filters are used, but each filter first consumes all data, processes it, and outputs it as a whole, does not have the concurrency advantage. These systems are known as batch sequential.

\(^5\)The Gnutella specification: [http://www.limewire.com/developer/gnutella_protocol_0.4.pdf](http://www.limewire.com/developer/gnutella_protocol_0.4.pdf)
(known as a servent, a combination of server and client) to maintain an open connection to at least one other servent. This way, a virtual network on top of TCP is created, in which all servents are initially equal. The Gnutella protocol is used for communication and routing in this virtual network.

Servents which are ‘better suited’ than others (for example, those on a high-bandwidth connection) can become superpeers, working as hubs in the network. All other nodes, on the ‘edges’ of the network, are known as leaves. In the virtual network, search requests are propagated by a flooding mechanism. Actual file exchange happens by regular direct HTTP connections.

This description gives an indication of some of the inherent properties of the peer-to-peer style: the existence of equal nodes, without a central authority, that work in a not fully connected network to provide an application to the user.

It should be noted that the description above applies to pure peer-to-peer networks, such as Gnutella and Freenet. Many hybrid forms exist, which employ some centralized control to manage the network, such as Napster.

**Vocabulary**

Although internet-based peer-to-peer systems are typically message-based, there is no reason they cannot use another interaction paradigm. Therefore, in its most basic form, a peer-to-peer system has only one constituent.

- *[Peer]* An entity that speaks the protocols required of a peer.

**Configuration rules**

In its most basic form, peer-to-peer does not imply any configuration rules. It should be noted that many implementations use rules like “maintain connections with at least two and at most eight peers”, which, given a large-enough number of peers, causes the emergence of a regular structure, such as a small-world network.

**Semantic interpretation**

Due to the emerging regularity, all nodes stay equal, and have an equal ‘view’ of the network. Since no communication paradigm is provided, a clear semantic interpretation cannot be given.

**Analyses**

Although peer-to-peer networks are inherently chaotic, their size can lead to regular structures, which can be analyzed efficiently. Typical analyses include throughput and utilization analyses for given numbers of nodes and analysis of the maximum capacity of the network.
Physical factors

<table>
<thead>
<tr>
<th>Components</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>o Peer</td>
<td>o (Dependent on technology)</td>
</tr>
</tbody>
</table>

Control issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Synchronicity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>Asynchron.</td>
<td></td>
</tr>
<tr>
<td>Arbitrary</td>
<td>Opportunis.</td>
<td>Run-time</td>
</tr>
<tr>
<td>Regular netw.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data issues

<table>
<thead>
<tr>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td>Hierarchic</td>
<td>And ---</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td>Star</td>
<td>High-volume</td>
<td>Broadcast</td>
<td>Run-time</td>
</tr>
<tr>
<td>Arbitrary</td>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

Control/data interaction

<table>
<thead>
<tr>
<th>Same shape?</th>
<th>Same direction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure A.15: Typing report for peer-to-peer.

Wrapup

Although peer-to-peer is a style which makes an architect ‘lose control’, its use has some interesting advantages, all given a large-enough network to obtain some regular structure.

- **[Easy configuration]** Even with huge numbers of system, a well-design peer-to-peer system will be easy to configure: as soon as a new peer contacts just one other, it will soon be an integrated part of the network.

- **[Scalability]** The ‘rules-of-behavior’ for all peers are equal. When chosen carefully, this will create a regular network with good scaling capabilities.

- **[Robustness]** Since there is no single point of failure, the system as a whole will continue to function even when large portions stop functioning.

- **[Cost]** Popular services can collapse under their own popularity, e.g., from being ‘slash-dotted’⁷. To prevent this, starting services could be required to invest in hardware that will only be used to its full potential once. The scaling capabilities of a peer-to-peer architecture can handle this, by allowing peers to ‘donate’ their bandwidth, e.g., distributing Linux Distribution ISO-files using BitTorrent⁸.

It should, however, be noted that the lack of central authority has some disadvantages.

---

⁶In a small-world network [http://en.wikipedia.org/wiki/Small_world_network](http://en.wikipedia.org/wiki/Small_world_network), a limited number of connections between nodes causes a power-law relation between the number of nodes in a network and its diameter.


• *[No guarantees]* With many peers, all being administered by another authority, it can be safely assumed that an average number of peers will remain active, ensuring availability of the service as a whole. However, the availability of specific entities in the network cannot be guaranteed.

In the end, peer-to-peer is a very high-level style, coming down to ‘all peers are equal’. For this style to be useful, the ‘voids’ must be filled in with various paradigms and technologies.

### A.8 Service Oriented Architecture (SOA)

A continuing trend in software engineering, aiming to cope with the increasing complexity of software, is the movement towards more abstract notions; for example, object orientation (see Section A.1). A similar movement is abstracting away from functionality [ML07].

The movement towards abstraction started with the decomposition of machine-code applications into separate jobs. Composing statements leads to subroutines, and the advent of procedural languages. In the eighties, similar subroutines were combined with the data they are intended for, hence object orientation. Objects working together bring about components.

A binding theme in all of these abstraction steps is the attempts to model business relevant objects such as ‘customer’ and ‘order’ in software. Service oriented architectures are another step of abstraction, moving away from the implementation entities (components) to ‘the thing they do’, the service they provide.

It should also be noted that this implies a distinction between *servers* and *services*; whereas servers have been the home to components, services can move around servers, allowing more flexibility.

Service orientation allows the decoupling of services from components, and therefore decoupling from physical computing hardware. This has a number of nice consequences,

• *[Maintainability]* Since the implementation and the identity of the component to be used are of no interest, it is possible to replace service-providing components with, new, improved ones.

• *[Highly dynamic]* Since services are invoked through interfaces, it is possible to move service providing components to different hardware, or to add multiple components that provide the same service to the system, for increased stability or performance.

• *[Reuse]* For components to be able to use a service provided by some other component, this service must be described and modularized thoroughly. This well-defined and published interface allows direct use from different applications, by using a service, instead of the component that represents it. In fact, the entire service can be reused, including the hardware it runs on, given enough capacity.

The consequences above are very desirable in a business-setting; organizations must try to adapt to changing market situations at an ever increasing pace. At the time of writing, SOA is one of the technologies that has a high buzzword-value, exemplified in Microsoft .NET (“Driving Business Value with the Microsoft Platform”), OSGi (“Universal Middleware”), IBM SOMA (“On Demand Business”), SAP NetWeaver (“Helps put you ahead of the curve”), and various consultancy firms promising assistance in achieving business value with SOA. The modularization does, however, make defining this easier.
Although inspired by business processes, the approach is useful for modeling other ‘real world’ entities, leading to a better understanding of their software counterparts. This could cause a number of other sought-after features, such as graceful degradation: the ability to continue operation, albeit in a less-efficient or less-precise way, when one of the services used by a component quits operating. In a service oriented system, the component can try to find another component providing the same service. A disadvantage of the service oriented approach is that it is still required to define the service in such a way that it handles the dynamic appearance and disappearance of services.

**Vocabulary**

Since SOA is an abstract style, decoupled from issues like deployment and interaction style, it is hard to give a definition of its constituents. Central in all forms of SOA, however, is the service.

A service is a self-contained, logical entity of computation or storage. [ML07] provides a hierarchy of integration-, business- and infrastructure services, which differ in their use. From a conceptual point of view, they have roughly the same setup, providing services through a published interface.

Many descriptions of services require some infrastructure (e.g., a container) or technological constraints (e.g., services being network-addressable). Though these considerations are important for a resulting architecture, they are not part of the basic style of service orientation.

**Configuration rules**

The configuration of services into an application is subject of a coordination method. The loose coupling of services is a good thing, but services need to be able to work together. Two main types of coordination can be defined, orchestration and choreography.

Orchestration requires a ‘director’, a central control entity that acts as a ‘hub’ for the other services. The director is responsible for passing data to the correct service, in conformance with a process description, usually some business workflow. An example of this, is BPEL. This central authority implies that all services should be controlled by a single entity.

In choreography, there is no central control. Services know what they should do, and know what other services to invoke for getting their job done. An example of this is WS-CDL, a language which allows all services to have an overview of what happen, and devise their own role in the process. In essence, a choreography is a formalized description of protocols businesses use to communicate, such as a protocol describing an ordering process. The lack of central authority makes it possible for a choreography to span enterprise boundaries.

A somewhat similar form of choreography shows up in the context of embedded systems, where choreography is (a) too complex to be encoded in all service-providing components, and (b) too complex to be stored in some general configuration file. Here, a dependency manager is used to resolve choreography issues, by ‘hooking the services up’.

---

9In which ‘published’ means that the interface has been made available to entities outside trust boundaries; it is not possible to know all the entities that use this interface. This implies that changes to the published interface cannot easily be made.


11Web Services Choreography Description Language, [http://www.w3.org/TR/ws-cdl-10/](http://www.w3.org/TR/ws-cdl-10/)
Semantic interpretation

A set of cooperating services, either by choreography or by orchestration, together represent an application.

Analyses

Service orientation abstracts away from components. There is no direct link between components and services, as components can provide zero or more services, and services are not used by other services, but by components. As such, the analyses cannot be derived from component-based styles.

It is more interesting to note that service oriented systems are an (almost) direct representation of business processes. By first checking the conformance of the system to the business processes and protocols, it is possible to use methods from the science of business administration.

Wrapup

Service orientation is a logical step in the movement towards larger-grained and more abstract ways of ‘looking at IT’. By decoupling services from the components and hardware that provide them, this movement allows a better match between software elements and business processes.

While SOA has the potential for delivering highly interesting properties, it should be noted that SOA requires much effort to ‘get it right’. Most effort is required for defining concise services, and making the services handle their dynamic environment in a graceful way. The abstraction of features into concise, self-contained and reusable services requires a thorough
analysis of the nature of the software system and the business. However, this initial effort can repay itself by enabling better software reuse.

SOA is often mixed up with ‘web services’; web services are an instantiation of the paradigm of service orientation.

A.9 C2

C2, short for Chimera-2 [TMA+95], formalizes the key features of Chiron-1, a system for GUI-based computer software. This leads to a style intended for the creation of software with an interactive interface.

A nice example from [TMA+95] considers the creation of a drawing tool, with given constraints. The tool allows drawing boxes and lines, with the constraint that lines can only be attached to boxes; when boxes are moved, the lines should be updated accordingly. Figure A.17 shows the architecture of this system.

The Image data component stores the boxes and lines. The Layout engine is responsible for the actual shape and location of the boxes and lines. The layout is sent downward; the Constraint manager receives these same notifications, and checks the sketched layout against the constraints (i.e., whether all lines are actually connected to boxes). If a constraint is violated, the Constraint manager sends requests up, requesting an update of the layout.

Note the modularity of this design: even if the Constraint manager was not planned beforehand, it can easily be introduced by making it commission requests to ascertain that the restrictions are kept, in response to notification of moving boxes.

Vocabulary

This style has two main parts,

- **Components** Components are entities which encapsulate a given logical object, such as a data store or a window manager. Communication with the outside world is done solely through a top-port and a bottom-port, using the predefined message types for the given direction (notifications down, requests up). Typing of components is allowed,
with a parameterized form of inheritance (interface, behavior and/or implementation). This subtyping mechanism is also used to support evolution of components; it also supports virtual objects (that is, objects to be instantiated later, e.g. for of-the-shelf components).

- **Connectors**  As shown in Figure A.18, connectors are intended to relay messages in a given direction. Like components, connectors can be subtyped, and have constraints defined over them. Connectors can ‘choose’ which messages to pass on to which components.

**Configuration rules**

A component’s port is connected to at most one connector. Both the top and bottom port of a component can be in use at the same time. The top- and bottom connector cannot be connected to the same connector. A connector can have an unlimited number of connections; even direct connections to other connectors are allowed; see Figure A.18.

**Semantic interpretation**

Two types of information move around the system: requests go up, and notifications go down. Information can only flow between two components which are connected to the same connector. However, a component is not aware of the components ‘below’ it. Connectors can filter messages, based on interests of the components they serve or precedence rules.

**Analyses**

C2 has some nice properties that allow efficient analyses of architectures created in this style. First, C2 has a sort-of layered structure. Analyses from this style allow checking the correctness of the encapsulations, by estimating the number of events fired by one component and reacted upon by another. If two components have an excessive amount of communication among them, this could be an omen that some modularization has been done incorrectly.
C2 also shares characteristics with event-based architectures, using a sort of semi-directed implicit invocation; notifications are definitely intended to go down, but component can never know the actions that get taken in response to them. Although general implicit invocation systems are hard to reason about (except when using the LTL approach, see Section A.5), the constraints imposed by C2 allow analysis of e.g. the number of events likely to be caused by an outside event, the performance implications of adding a new component, or the location of likely bottlenecks.

Wrapup

Chimera-2, or C2 for short, is one of the first styles in literature that has been specifically engineered to support GUI applications. It describes a sort-of layered structure, and fixes the interaction style to be implicit invocation.

The system of connectors and components with just two ports allows for good flexibility: new components can be added without considerable reconfiguration, and components can be replaced without much trouble. Like in implicit invocation, adding new components can lead to performance issues.

It is nice to note that, along with the style, a specific ADL and tooling has been developed for C2. The tool has now been included in xADL\textsuperscript{12}.

A.10 Publish/subscribe

In 1993, [Boa93] appeared, describing Thales’ SPLICE (Subscription Paradigm for the Logical Interconnection of Concurrent Engines). Thales (former Hollandse Signaal) had developed

\textsuperscript{12}http://www.isr.uci.edu/projects/xarchuci/
SPLICE as a Naval Combat Management System, intended for integrating all of a ship’s systems, such as sensors, weapons and navigation systems. Living in a demanding and hostile environment, SPLICE was built to be distributed, data driven and battle-damage resistant. Currently, the SPLICE-based TACTICOS Combat Management System is used in various navies around the world.

The main motto for SPLICE is “the right information at the right time at the right place”. In order to achieve this, SPLICE relies on time- and space-decoupling. The main structure of SPLICE is shown in Figure A.20.

In this figure, Processes make up parts of an application. With their Agents (which are all identical) they register what information they produce or consume. The Agents then take care of distributing the correct information to the correct Processes.

Time-decoupling means that information never gets outdated; all data that has ever been produced can become of use in the future. To store all information, the Agents manage a distributed database of events. The distributed nature of the agents in the system ensure robustness; with this setup, it is easy to incorporate redundant systems, in case one fails. The network is now the only single-point-of-failure; in most practical systems in use, however, the network is also replicated.

The term publish/subscribe appeared only several years later, around 1996/1997. [EFGK03] gives an overview of the various publish/subscribe systems, abstracting the architecture to the one shown in Figure A.21.

In this abstracted version, the use of agents is left implicit. Instead, all processes connect to a single event service, which takes care of the tasks which were performed by the Agents.
in SPLICE. All underlying details of the event service are deferred to the implementer. This leaves a number of key characteristics of the Publish/subscribe style.

- All processes can publish data.
- All processes can register to receive data. The ‘filtering’ of this data can happen in a number ways (by type, a USENET-like topic structure, …), but that is beyond the scope of this document.
- The event service causes decoupling in time. Information never gets lost. It may get outdated by new information, but it is always possible that ‘old’ data is useful for some purpose. The decoupling means that a publisher can publish data, which is later consumed by a subscribed customer, without the necessity of both processes being connected at the same time.

**Vocabulary**

As shown in Figure A.21, there are two main components.

- *[Processes]* The components that together make up an application.
- *[Event service]* A component which takes care of getting “the right information at the right time at the right place”. Note that, as in Figure A.20, this service can be distributed, and made of multiple agents.

**Configuration rules**

All processes get connected to the same (distributed) event service.

**Semantic interpretation**

Processes connected to the same event service can publish data and subscribe to data, produced elsewhere in the system. Published data will always be received by all processes that are subscribed to the kind of data this publication belongs to.

**Analyses**

None of the documents gives a set of possible analyses. However, given the right information, it should be possible to establish at least the following metrics based on the architecture,

- Average time needed to respond to a given event.
- Maximum number of copies of a given process on a system.
- Maximum sustainable input and output rate for given events and responses.

One formalized approach can be found in [GKK03], modeling the publish-subscribe system as a system of message-passing finite state machines; this allows for LTL model checking to validate safety and liveness properties.
Wrapup

Publish/subscribe is a paradigm which requires some investment; it needs specialized infrastructure, places tough constraints on the (network) environment, and needs specialized applications to provide useful services. When running, however, publish/subscribe based systems are remarkably powerful and reliable, using the reliability and redundancy of underlying layers to its advantage. These characteristics make it very well suited for mission-critical systems, without the possibility for including absolute time constraints.

A.11 REST

In preparation for the definition of a new HTTP specification, the need was felt for an idealized model of ‘well-designed’ web-applications. In [Fie00], Representational State Transfer (REST) is given as the ‘ideal’ model of a web application. As an example, consider the typical example of a RESTful application, the Atom Publishing Protocol.

The Atom Publishing Protocol, or APP, is a conceptual model of web resources, a format for discovering and retrieving these resources, possibly grouped in collections, and a protocol for publishing and editing. The main use of Atom is the maintenance and syndication of weblogs. Figure A.23 shows an intuitive view of publishing content with APP.

Both the publisher and the reader are located on the client side of the system, for they are the system’s users. Resources are created by the publisher using the HTTP based publication protocol, using HTTP-verbs POST and PUT. By using the syndication format (as standardized in [NS05]), the reader can GET a representation of the resource. (e.g., for a weblog, this could be ‘the most recent post’, ‘all posts of today’ or ‘post nr. 238’. Note that some of these designations have a temporal element, meaning the same designation can point to a different piece of data at different moments.)
The distinction between resources and their representation is a key characteristic of REST. Figure A.24 shows the connection between identifiers, resources and representations. The notion of ‘resource’ allows the linking of identifiers (e.g. URI’s) and representations (e.g. HTML pages).

This data model shows REST’s focus on data. The philosophy behind this is ‘there are more nouns than verbs’; REST promotes the use of built-in verbs in the communication technology used, such as the GET, PUT, POST and DELETE vocabulary of HTTP.

Structurally, REST is a formalization of the ‘shape of the web’, comprising user agents like browsers, content servers like Apache and intermediary devices such as proxies. Figure A.25 shows a reprint from [Fie00], with some exemplary connections of the various components. The center of the figure is a user agent having three interactions. This agent has its own cache ($\$\$), but this does not have the requested information. The three scenarios shown are

(a) The user agent contacts a proxy, which cannot generate a reply based on its own cache, and therefore contacts a gateway. This gateway, too, cannot satisfy the request; it is therefore forwarded to an origin server.

(b) The user agent directly contacts the origin server, which replies using its own cached data.

(c) Using a proxy, the user agent contacts a WAIS database.

The descriptions above give some hints to the concept of an ‘ideal’ web application. [Fie00] uses the following characteristics. Note that some of these may seem trivial now; they are not a set of definitive requirements, but rather a reconstruction of the requirements of the original infrastructure of the web.

- [Low entry barrier] The success of the internet is based on the possibility for all users to create content; therefore it should be ‘easy’ to create new content and applications.
• **[Extensibility]** The web is susceptible to changes, based on its changing technological or social context.

• **[Hypermedia support]** Web resources are rarely self-contained. They contain functionality or images located on other systems. This calls for an architecture that allows efficient transport of information from many resources.

• **[Anarchic scalability]** One of the main characteristics of the web is that it is not ‘owned’ by a single entity; this means no assumptions can be made about the existence of resources or the amount of ‘traffic’ a given resource will receive.

**Vocabulary**

REST uses an explicit distinction between components and connectors. The connectors used are stateless, abstract interfaces that provide a streaming mechanism for parameter- and result-passing.

• **[Client]** A client connector initiates requests.

• **[Server]** A server connector listens for incoming requests.

• **[Cache]** A cache connector saves responses to requests. A cache connector is part of either a client connector (to reduce network load) or to a server connector (to prevent the re-generation of responses). It should be noted that caching rules apply: only idempotent requests can be cached (i.e., requests that always yield the same response, and do not change the state of the origin server), and cached responses ‘expire’.

• **[Resolver]** A resolver is responsible for locating a resource, as described by a resource identifier, on the ‘physical’ network. The prime example of this is DNS.

• **[Tunnel]** Components, connected by client- and server connections may make up a ‘chain’ of connecting elements. For certain purposes, the original client connectors wants to ‘think’ it is communicating directly with the end-station. In this case, the intermediary components merge to form a tunnel.

Four types of components are used in REST,
• **[User agent]** A user agent acts on behalf of a user, utilizes a client connector to perform its task. A typical example is a web browser, but automated spiders are also user agents.

• **[Origin server]** Origin servers are the habitat of resources. They respond with representations of their resource on request, and handle requests which alter the resource.

• **[Proxy]** A user agent can use a proxy as an intermediary for connecting to an origin server. Reasons for this could be the reduction of network load (a caching proxy) or enforcement of security rules (firewall).

• **[Gateway]** An origin server can use a gateway to reduce its load, allowing the gateway to distribute requests over multiple servers, or reply from cache.

It should be noted that [Fie00] provides a paragraph on data elements; since these are not a part of Garlan’s vocabulary, I will stick with the distinction shown in Figure A.24.

**Configuration rules**

Components can have both client and server connectors, where user agents only employ client connectors, and origin servers only employ server connectors. Connections are made between one client connector, and one server connector.

**Semantic interpretation**

Information can only be transported between elements that are connected (using one client- and one server connector).

**Analyses**

Due to the 'anarchic scaling' of the Internet, it is hard make any predictions about REST-based architectures. However, when factoring out the unpredictable elements like network latency, some useful metrics can be constructed based on characteristics of the components, for example

• Maintainable and maximum amounts of service requests per second.

• Maximum number of users serviceable at one moment.

**Wrapup**

REST is a typical example of a style which formalizes well-known principles. In its core, it is an abstraction of the HTTP protocol and the web infrastructure supporting that. It should be noted this does not mean HTTP is the only transport mechanism usable for REST.

The main characteristic of REST is the principle of ‘many nouns, few verbs’, allowing loose coupling. It supports the use of existing technology (using HTTP’s GET and PUT), instead of creating an abstraction layer above this (like SOAP’s XML-based procedure invocation); the main rationale for this is the fact that the web is built to handle HTTP requests using its limited set of verbs.

Like many terms that have to do with the web, REST is often used inappropriately. A quick review using a search engine will find REST showing up as a technology, a toolkit, a framework, and many more things which it is not.
<table>
<thead>
<tr>
<th>Physical factors</th>
<th>Components</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>o User agent</td>
<td>o Client/server connection</td>
<td></td>
</tr>
<tr>
<td>o Origin server</td>
<td>o Cache</td>
<td></td>
</tr>
<tr>
<td>o Proxy</td>
<td>o Resolver</td>
<td></td>
</tr>
<tr>
<td>o Gateway</td>
<td>o Tunnel</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control issues</th>
<th>Topology</th>
<th>Synchronity</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear</td>
<td>Seq. Lockstep</td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td>Acyclic</td>
<td>Par. Lockstep</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Hierarchic</td>
<td>Synchron.</td>
<td>Invocation</td>
</tr>
<tr>
<td></td>
<td>Star</td>
<td>Asynchron.</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>Arbitrary</td>
<td>Opportunis.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data issues</th>
<th>Topology</th>
<th>Continuity</th>
<th>Mode</th>
<th>Binding time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear</td>
<td>Continuous</td>
<td>Passed</td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td>Acyclic</td>
<td>Sporadic</td>
<td>Shared</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Hierarchic</td>
<td>And ----</td>
<td>Cp-in-cp-out</td>
<td>Invocation</td>
</tr>
<tr>
<td></td>
<td>Star</td>
<td>High-volume</td>
<td>Broadcast</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>Arbitrary</td>
<td>Low-volume</td>
<td>Multicast</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control/data interaction</th>
<th>Same shape?</th>
<th>Same direction?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure A.26: Typing report for REST.
Appendix B

Preparation discussion 1
Abstract

Eén van de onderzoeksmethoden gebruikt in de domeinstudie voor het project Adaptive Architectures is een discussiemiddag met architecten. Deze discussiemiddag bestaat uit een kort overzicht van de voorlopige conclusies van het onderzoek, het vergelijken van een aantal architectuurstijlen, en een aantal discussies over de beste stijl in een gegeven situatie.

In voorbereiding hierop geeft dit document een overzicht van (1) de invalshoek die is gekozen in het onderzoek, (2) de voorlopige bevindingen, en (3) de wijze waarop ik tegen de materie aan kijk. De doelen van dit document zijn op te sommen als (1) het verkrijgen van een gezamenlijke basis voor het begin van de discussie en (2) het verschaffen van “stoof tot nadenken”.

1 Inleiding

1.1 Het project

In dit onderzoek is gekozen voor de invalshoek ‘kwaliteitsfactoren’, dat wil zeggen, kernwoorden die onlosmakelijk met een bepaalde stijl verbonden zijn. Dit zijn kenmerken die architectuurstijlen verbinden of juist onderscheiden. Door deze factoren in kaart te brengen, te classificeren in een taxonomie, en de kwaliteiten te ‘mappen’ op probleemfactoren, moet het mogelijk worden een gefundeerde en verdedigbare keuze voor een architectuurstijl te maken.

Om het systeem van kwaliteitsfactoren te toetsen, wordt een stijl ontwikkeld waarin een arbitrair gekozen kwaliteitsfactor een belangrijke rol speelt. Deze stijl wordt door middel van case studies geëvalueerd.

1.1.1 Huidige status

Naast de obligate activiteiten in verband met mijn master project, zijn de volgende taken inmiddels uitgevoerd.

- Er is een inventarisatie gemaakt van methoden om een taxonomie op te bouwen. Hierniet is, voorlopig, een methode gekozen waarin kwaliteitsfactoren gegroepeerd worden in dimensies, waardoor een multidimensionale ruimte ontstaat.

- Om een basis voor de kwaliteitsfactoren te krijgen, is gezocht naar richtlijnen die architectuurstijlen aan één of meer probleem eigenschappen koppelen. Dit bracht helaas de nodige problemen met zich mee; meer hierover in paragraaf 1.4.2.

- Om ‘bewoners’ voor de taxonomie te verkrijgen, is gezocht naar bestaande architectuurstijlen, en gepoogd deze op een eenduidige manier te beschrijven. Daarnaast is het nuttig om een verzameling van stijlen te hebben; deze verzameling wordt gebruikt om verbindende en onderscheidende factoren te bepalen. Om een gevoel te krijgen voor de methode van beschrijven, is een voorbeeld hiervan opgenomen in paragraaf 2.
1.2 Doelen van de sessie

In paragraaf 1.4 wordt uiteengezet waarom het lastig is om een formeel model van software architectuur te maken; dit geldt eveneens voor een model van architectuurstijlen. Hoewel het niet mogelijk is een formele aanpak te gebruiken, en de kunst van het verkrijgen van de ‘quality without a name’ (paragraaf 1.4.3) lastig over te dragen is, probeert dit project meer grip te krijgen op de kern van software architectuur. Het doel van de sessie is het bundelen van de kennis van diverse architecten, om op deze manier richtlijnen voor toekomstige projecten te creëren.

1.3 Termen en definities

De hieronder gebruikte termen en definities zijn deels overgenomen uit de literatuur, en deels voor dit project gecreëerd (zie ook paragraaf 1.4.1). Deze lijst claimt niet compleet of correct te zijn; de lijst moet eerder worden gezien als een gezamenlijke basis, zodat ‘we het over hetzelfde hebben’. Omdat de definities uit hoofdstuk 1 van de domeinstudie komen, zijn deze in het Engels opgesteld.

**Architectural style**

An architectural style defines the elements that make up an architecture and the way they should interact. Examples include pipe-and-filter, blackboard and service-oriented. The term *architectural pattern* is also used. For the purpose of this project, I propose the following definition

“The set of common, distinctive characteristics of architectures intended for solving similar problems.”

**Reference Architecture**

Whereas an architectural style can be applied in a wide range of applications, a reference architecture is intended for a given domain or applications with a common goal. It describes basic building blocks and construction, usually based on an architectural style. Although not identical, a reference architecture is related to a Domain Specific Architecture, which combines a style with knowledge from a given domain.

**Architecture**

An architecture defines a high-level view of an actual system, dictating which components have what purpose, and how they should interact. Architectures are *intensional* and work on a *non-local* level [EK03]. Therefore, choices made on the architectural level have implication for the entire system, including parts that are not yet designed. IEEE1471 uses the definition “The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.”

**Design**

A design gives a component-level view of an actual system. Designs are said to be *intensional* and work on a *local* level, meaning that a design can incorporate choices, but only within a specified scope.

**Implementation**

From a software engineers point of view, this is the lowest possible view at a system, leaving no room for interpretation; an implementation is *extensional* and works on a *local* level.

1.4 Voorlopige bevindingen

Ondanks de korte looptijd (nog geen kwart van de tijd is nu besteed), kunnen een aantal voorlopige conclusies getrokken worden. Dit zijn wellicht zaken waarover een ervaren architect zich niet zou verbazen, maar aangezien ik enkel kan putten uit ‘wat geschreven is’ en mijn eigen beperkte ervaring, zijn ze hier het vermelden waard. Uiteraard sta ik open voor debat over mijn bevindingen.
1.4.1 Definities

Er is na jaren van gebruik nog altijd geen overeenstemming over wat ‘architectuur’ precies is. Hetzelfde geldt voor de term ‘stijl’.

De voornaamste reden hiervoor is de ‘ongrijpbaarheid’ van deze concepten, waardoor er verschillende aanpakken ontstaan. Om architectuurstijl als voorbeeld te nemen, hebben Perry en Wolf de stelling dat een stijl niets anders is dan een ‘minder specifieke’ architectuur; deze stelling is later overgenomen door Fielding. Eden en Kazman gaan er juist vanuit dat een stijl een abstractie van een specifieke architectuur is, bestaand uit regels die in het hele systeem horen te gelden, waardoor stijl op een heel ander niveau leeft dan architectuur. Shaw en Clements houden het erop dat een stijl weinig meer is dan een set toegelaten componenten (constituents), en hun interacties; dit is een meer structurele aanpak, die ook door Garlan wordt aangehangen.

Voor dit onderzoek heb ik een positie ingenomen die wat in de richting van Fielding gaat. Omwille van de omvang van dit document, zal ik daar hier niet verder op ingaan.

1.4.2 Publicaties

Het leek redelijk om te verwachten dat de software architectuur een groot aantal wetenschappelijke publicaties zou opleveren, waar ik mijn voordeel mee kan doen. Helaas zijn over bepaalde onderwerpen, waaronder richtlijnen en classificaties, maar weinig documenten te vinden. De analyse hieronder is gebaseerd op hoofdstuk 3 van mijn domeinstudie.

Op zich heeft het vakgebied ‘software architectuur’ zeker niet aan een gebrek aan aandacht te lijden. Na een langzame start in de jaren 80, is het aantal publicaties in de jaren negentig bijzonder enthousiast gegroeid, waarna aan het einde van de jaren negentig een stabilisatie in het aantal publicaties per jaar optrad. Ook zijn er in het verleden een aantal bijzonder steekhoudende aanzetten geweest om te komen tot een ‘bundeling’ van de kennis van software architecten.

Helaas is het vaak bij die aanzetten gebleven: zelden zijn follow-up documenten gecreëerd die de algemene strekking van de aanzetten uitwerken. Toch is er, zoals hierboven vermeld, een stabiel aantal publicaties per jaar. Waar gaan deze documenten dan over?

- De term ‘software architectuur’ heeft een professionele connotatie. Hierdoor wordt de term vaak te pas en te onpas gebruikt: een ontwerp wordt vaak aangezien voor een architectuur, en patronen en reference architecturen gaan door het leven als architectuurstijlen.
- De academische wereld heeft de neiging om te formaliseren. Hierdoor zijn tientallen methoden verschenen om een architectuur op een formele en ‘bewijsbare’ manier te beschrijven. Helaas leeft software architectuur in een wereld die té divers is (zie ook hieronder), waardoor formalisatie vaak veel meer energie kost dan het potentieel op kan leveren. Een voorbeeld hiervan is de π-method, een methode ontwikkeld met steun van de Europese Unie, waarin software-ontwikkeling van architectuurstijl tot implementatie bewijsbaar correct kan worden uitgevoerd.

Daarnaast kan formalisatie leiden tot het ‘ontkennen’ van de rijkheid van het probleemgebied. Zie hiervoor ook paragraaf 1.4.4.

De moeilijkheden van software architectuur zijn in de academische wereld een bekend onderwerp. Er zijn talloze pogingen ondernomen om methoden te ontwikkelen die software architectuur ‘eenvoudiger’, of toch in elk geval tastbaarder, pogen te maken. Dit gebeurt vaak door middel van formalisatie.

Aan de andere kant kijkt een deel van de academische wereld op een fundamenteler manier naar de zaken. Helaas lijd na deze aanpak wel onder een gebrek aan interesse: het afstand doen van formalisatie leidt tot een bepaalde blik op software architectuur, die leeft op het raakvlak van een groot aantal vakgebieden, waaronder design-theory, systeentheorie, economie, sociale factoren, industrie, . . . Hierdoor is het lastig om een duidelijk omlijnd onderzoeksgebied aan te wijzen.
Jarenlang wordt er werk verricht aan de ontwikkeling van grote software systemen, waarbij software architectuur enthousiast wordt toegepast. Het lijkt niet meer dan logisch dat de collectieve kennis van deze architecten op zekere moment gebundeld is.

Tot op zekere hoogte is dit gebeurd. Helaas gaat men hierbij zelden verder dan de ‘triviale’ en goed begrepen voorbeelden en stijlen. Op het gebied van ‘stijlkeuze’, ofwel het bepalen van een architectuurstijl die dominant wordt voor (een deel van) een architectuur, is er een voornam factor die het gebrek aan optekening verklaart: stijlkeuze is zelden ‘puur’. In het ideale geval is de keuze voor een stijl gebaseerd op factoren die direct met het probleem te maken hebben. Vaker spelen echter zaken mee die met het originele probleem weinig van doen hebben, zoals

- persoonlijke voorkeur van de architect (bekend als de ‘null-stijl’),
- bedrijfs-politieke overwegingen (“We gebruiken deze stijl al jaren, en hij levert voorspelbare resultaten”),
- het team waarmee ontwikkeld wordt, en dan met name de grootte en de beschikbare kennis en ervaring,
- de ‘hype of the moment’, de stijl die op dat moment populair is, en ‘toevallig’ ondersteund wordt door één of meer grote vendors. Voorbeelden hiervan zijn zijn layerd systemen, later gevolgd door multi-tier oplossingen, en op het moment service oriented architecturen.

Als we kijken naar algemener zaken, wordt dit vaak door bedrijven geheim gehouden. Documenten als ‘best practices’ worden beschouwd als de kern waarmee een producent of dienstverlener zijn voorsprong op anderen haalt; hierdoor is men huiverig deze kennis ‘weg te geven’.

Kortom, er zijn genoeg publicaties, maar slechts weinigen lijken echt de kern van de zaak te raken. Dit is zowel begrijpelijk als jammer: het is begrijpelijk, omdat het een vrij abstract gebied betreft, waarin enkele inherente problemen de kop op steken van het leven in een sociale context. Het is jammer, omdat een goed begrip van deze kern een ‘kapstok’ geeft waarmee de rest van het vakgebied begrepen kan worden.

1.4.3 De ‘quality without a name’

Christopher Alexander beschreef in ‘The Timeless Way of Building’ de ‘quality without a name’; een eigenschap van een gebouw of ruimte waardoor de gebruiker zich prettig voelt. Een zelfde eigenschap kan bestaan voor software architecturen, namelijk het vermogen om in een toestand te blijven van ‘minste conflict’ tussen de vele belangen die invloed hebben op een software systeem. Het essentiële probleem hiervan is door Alexander beschreven als

“The quality [without a name] is objective and precise, but cannot be named.”

ofwel, de kwaliteit is wel degelijk aanwezig, maar niet aan te wijzen, en dus ook niet te engineeren,

“This quality in buildings and in towns cannot be made, but only generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from a seed.”

Deze wijze van ‘genereren’ is lastig te formaliseren; veel architectuur is daarom gebaseerd op intuïtie. Hiervoor is een vrij duidelijke oorzaak aan te wijzen, zoals beschreven in de volgende alinea.

1.4.4 Wicked problem

De term ‘wicked problem’ is door Rittel en Webber geïntroduceerd in de context van social planning [RW73], waaronder bijvoorbeeld het bepalen van regels voor uitkeringen, oplossingen voor criminaliteit en stedenbouwkundige planning. Een verbindende karakteristiek van deze problemen
is dat ze ‘niet oplosbaar’ zijn; daarnaast brengen wicked problems een aantal andere eigenschappen met zich mee.

Rittel en Webber noemen tien eigenschappen die er voor zorgen dat een probleem als een wicked problem te classificeren is, waarbij ze opmerken dat een probleem niet alle eigenschappen hoeft te hebben om toch een wicked problem te zijn. Een volledig bewijs van de wicked-ness van software architectuur is onderwerp van een ander document; hier zal ik volstaan met de opmerking dat (1) problemen in software architectuur vaak dusdanig ‘vaag’ zijn dat het probleem vaak pas begrepen wordt wanneer de architect een mentaal model van de oplossing heeft, (2) software architectuur door tijdgebrek vaak een ‘one-shot business’ is, en (3) er geen objectieve methoden bestaan om te bepalen of een architectuur de beste oplossing is.

1.4.5 Conclusies

Opsommend kunnen we vaststellen dat

• er geen overeenstemming is over wat architectuur precies is,
• er zeker geen gebrek is aan interesse, maar de termen vaak misbruikt worden,
• stijl- en andere architectuurkeuzen zelden op een ‘pure’ manier worden gemaakt, en daarom lastig als voorbeeldmateriaal te gebruiken zijn,
• software architecturen een ‘quality without a name’ nastreven, en
• deze kwaliteit niet te engineering is, omdat software architectuur als een wicked problem te classificeren valt.

Hiermee is de cirkel rond. We kunnen concluderen dat het doel van dit onderzoek, het verkrijgen van inzicht in de basis van software architectuur, om goede redenen nog niet vaak beschreven is. De noodzaak is eveneens duidelijk. De gekozen insteek, het classificeren van architectuurstijlen, heeft te lijden onder dezelfde ‘vaagheid’, waardoor formalisatie niet mogelijk is zonder het ontkennen van de rijkheid van het onderwerp. Wel is het mogelijk om een stap te wagen richting het ‘optekenen van collectieve kennis’, in de hoop daarmee handreikingen voor toekomstige projecten te creëren.


2 Stijlvoorbeeld

Hieronder worden een voorbeeld gegeven van een stijlbeschrijving. Om voorlopig een enigszins uniforme vorm te hebben, heb ik gekozen voor het beschrijven van de elementen zoals Garlan die graag wil zien, als in [Gar95]. Deze zijn,

- **[Vocabulary]** Waaruit kan een architectuur in deze stijl bestaan? Te denken valt aan filters, pipes, clients, etc.
- **[Configuration rules]** Topologische beperkingen; op welke manier mogen de elementen ‘aan elkaar geknoopt’ worden?
- **[Semantic interpretation]** Wat betekent een correct opgebouwde architectuur precies?
- **[Analyses]** Architecturen die op dezelfde stijl zijn gebaseerd, zullen vergelijkbare eigenschappen hebben. Welke van deze eigenschappen zijn meetbaar?

De beschreven stijl is wederom afkomstig uit de domeinstudie.

2.1 Publish/subscribe

In 1993, [Boa93] appeared, describing Thales’ SPLICE (Subscription Paradigm for the Logical Interconnection of Concurrent Engines). Thales (former Hollandse Signaal) had developed SPLICE as a Naval Combat Management System, intended for integrating all of a ship’s systems, such as sensors, weapons and navigation systems. Living in a demanding and hostile environment, SPLICE was built to be distributed, data driven and battle-damage resistant. Currently, the SPLICE-based TACTICOS Combat Management System is used in various navies around the world.

The main adagium for SPLICE is “the right information at the right time at the right place”. In order to achieve this, SPLICE relies on time- and space-decoupling. The main structure of SPLICE is shown in Figure 1.

![Figure 1: Structure of SPLICE.](image)

In this figure, Processes make up parts of an application. With their Agents (which are all identical) they register what information they produce or consume. The Agents then take care of distributing the correct information to the correct Processes.

Time-decoupling means that information never gets outdated; all data that has ever been produced can become of use in the future. To store all information, the Agents manage a distributed database of events. The distributed nature of the agents in the system ensure robustness; with this setup, it is easy to incorporate redundant systems, in case one fails. The network is now the only single-point-of-failure; in most practical systems in use, however, the network is also replicated.

The term publish/subscribe appeared only several years later, around 1996/1997. [EFGK03] gives an overview of the various publish/subscribe systems, abstracting the architecture to the one shown in Figure 2.

In this abstracted version, the use of agents is left implicit. Instead, all processes connect to a single event service, which takes care of the tasks which were performed by the Agents in SPLICE.
All underlying details of the event service are deferred to the implementer. This leaves a number of key characteristics of the Publish/subscribe style.

- All processes can publish data.
- All processes can register to receive data. The ‘filtering’ of this data can happen in a number of ways (by type, a USENET-like topic structure, ...), but that is beyond the scope of this document.
- The event service causes decoupling in time. Information never gets lost. It may get outdated by new information, but it is always possible that ‘old’ data is useful for some purpose. The decoupling means that a publisher can publish data, which is later consumed by a subscribed customer, without the necessity of both processes being connected at the same time.

2.1.1 Vocabulary
As shown in Figure 2, there are two main components.

- \([\text{Processes}]\) The components that together make up an application.
- \([\text{Event service}]\) A component which takes care of getting “the right information at the right time at the right place”.

2.1.2 Configuration rules
All processes get connected to the same event service.

2.1.3 Semantic interpretation
Processes connected to the same event service can publish data and subscribe to data, produced elsewhere in the system. Published data will always be received by all processes that are subscribe to the kind of data this publication belongs to.

2.1.4 Analyses
None of the documents gives a set of possible analyses. However, given the right information, it should be possible to establish at least the following metrics based on the architecture,

- Average time needed to respond to a given event.
- Maximum number of copies of a given process on a system.
- Maximum sustainable input and output rate for given events and responses.
2.1.5 Wrapup

Publish/subscribe is a paradigm which requires some investment: it needs specialized infrastructure, places tough constraints on the (network) environment, and needs specialized applications to provide useful services. When running, however, publish/subscribe based systems are remarkably powerful and reliable, using the reliability and redundancy of underlying layers to its advantage. These characteristics make it very well suited for mission-critical systems, without the possibility for including absolute time constraints.
3 Agenda

De planning hieronder is voorlopig. Het is mogelijk dat tijdens de komende weken hierin wijzigingen worden aangebracht, en zelfs tijdens de sessie kan besloten worden om de timing wat aan te passen; deze agenda is dus puur indicatief.

<table>
<thead>
<tr>
<th>Tijd</th>
<th>Bijeenkomst</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:00 – 13:30</td>
<td>Ontvangst. Voor wie interesse heeft, kan altijd een lunch geregeld worden.</td>
</tr>
<tr>
<td>13:30 – 14:00</td>
<td>Inleiding</td>
</tr>
<tr>
<td>14:00 – 14:30</td>
<td>Eerste discussieronde: vergelijking van twee architectuurstijlen.</td>
</tr>
<tr>
<td>14:30 – 14:45</td>
<td>Pauze</td>
</tr>
<tr>
<td>15:15 – 15:45</td>
<td>Derde discussieronde: een praktisch probleem</td>
</tr>
<tr>
<td>15:45 – 16:00</td>
<td>Pauze</td>
</tr>
<tr>
<td>16:00 – 16:30</td>
<td>Vierde discussieronde: een praktisch probleem</td>
</tr>
</tbody>
</table>

Na afloop is er de mogelijkheid om nog even over luchtiger zaken te praten, in combinatie met de luminis-vrijdag-middag-borrel.

References


Appendix C

Results discussion 1
Adaptive Architectures
Resultaten discussiemiddag

A. F. M. van der Sijpt

December 2006

Abstract

Dit document geeft een overzicht van de voornaamste resultaten en conclusies van de in de inleiding beschreven discussiemiddag.

De meest in het oog springende conclusies zijn (1) software architectuur 'gebruikt' meer context van het probleem dan verwacht, (2) er zijn minder kwaliteitsfactoren dan verwacht, en (3) de grote context maar software architectuur tot een soort 'kunstvorm'.

Aan deze conclusies wordt gevolg gegeven door (1) een strikter definitie van software architectuur en stijl aan te nemen, die lijkt op de definitie van Fielding, (2) een bestaand set van kwaliteitsattributen te gebruiken (b.v. ISO 9126), in plaats van het 'zelf opsporen' van deze attributen, en (3) het beperken van de vrijheid in toekomstige sessie(s).

1 Inleiding

De discussiemiddag in het kader van het project Adaptive Architectures, als deels beschreven in [vdS06b], met een beschrijving van het volledige project in [vdS06a], heeft resultaten opgeleverd die van belang zijn voor de sturing van het project. Om deze informatie niet te laten 'verdrinken' in de domeinstudie, is er voor gekozen deze resultaten apart op te tekenen.

Dit document is niet bedoeld als integraal verslag van de sessie. Het probeert eerder de voornaamste en meest opvallende zaken uit de sessie weer te geven.

2 Resultaten

2.1 Algemeen

In het algemeen valt op te merken dat de middag een zinvolle discussie heeft opgeleverd. Hoewel de originele doelstelling, het benoemen van kwaliteitsfactoren voor architectuurstijlen, niet bereikt is, heeft de discussie waardevolle informatie opgeleverd voor de sturing van het project.

2.2 Evaluatie elementen

Zoals beschreven in [vdS06b], bestond de middag uit vier min of meer onafhankelijke onderdelen. Hieronder een evaluatie van elk van deze elementen.

2.2.1 Opfrissing voorbereidend materiaal

Deze inleiding is, net als het voorbereidend materiaal, als helder ervaren. Het is leuk om een overzicht te hebben van de voortgang van het project, en om misverstanden over definities te voorkomen is het nuttig om mijn kijk op de zaken over te brengen.
2.2.2 Stellingen
Het voorleggen van stelling om discussie uit te lokken blijkt doeltreffend te zijn. In het geplande halfuur zijn slechts vier van de acht stellingen besproken. Daarnaast is deze discussie een mooie opwarming voor de rest van de middag.

2.2.3 Stijlvergelijking
Door het naast elkaar leggen van een aantal architectuurstijlen, hoopte ik kernwoorden te vinden die de stijlen verbonden of onderscheiden. Helaas heb ik, achteraf gezien, naar de verkeerde zaken gevraagd: interactiestijl en structuur zijn niet de kwaliteitsdimensies die ik graag wilde weten (zie ook paragraaf 3). Doordat ik heb aangestuurd op de verkeerde eigenschappen, heeft dit element niet de verwachte informatie opgeleverd. Gelukkig maar.

2.2.4 Probleemstellingen
Het aanbieden van een omschrijving van een systeem, het vervolgens identificeren van belangrijke kwaliteiten, en het schetsen van een ruwe architectuur zou de nodige kwaliteitsfactoren op kunnen leveren. Hierbij heb ik helaas de nodige informatie verkeerd ingeschat: terwijl ik uitging van ‘eerste ingevingen’ op basis van een probleemstelling, blijkt de nadruk erg te liggen op het volledig willen begrijpen van het probleem1. Typisch gebruik, piek-belasting en de toegevoegde waarde voor de klant bleken hierbij belangrijke drijfveren; kort gezegd wordt er veel meer context bij het probleem betrokken dan ik verwachtte.

Uiteindelijk heeft dit onderdeel geen direct bruikbare kwaliteitsfactoren opgeleverd. Het heeft me wel de kans gegeven om een deel van het proces van dichtbij te zien. Dit heeft me wat inzicht gegeven in de manier van werken en de ‘ongeschreven’ zaken.

2.3 Opvallende zaken
Zoals eerder vermeld, is het niet eenvoudig om direct kwaliteitsfactoren aan problemen of stijlen te verbinden. Het blijkt wel mogelijk zijn om, bij het zien van een probleem, op ‘gevoel’ de lastige zaken aan te wijzen; de zaken die als ‘load-bearing’ worden ervaring, punten die in orde moeten zijn voordat een oplossing zijn beloften kan inlossen. Dit hoeven niet noodzakelijk complexe zaken te zijn, maar eerder ‘non-lokale’ invloeden. Daarnaast valt op te merken dat de functionele eigenschappen van een systeem voor kennisgeving worden aangenomen (“dat wordt wel opgelost”), terwijl de nadruk ligt op niet-functionele eigenschappen en eigenschappen ‘buiten het systeem’, zoals invloeden van de organisatie.

Hoewel ik er geen bewijs voor kan formuleren, durf ik te stellen dat het effectief opsponen van de pijnpunten dicht bij de kern van software architectuur ligt. Deze kunde is niet goed te beschrijven of over te dragen, omdat hij op ervaring en gevoel berust. Het maakt software architectuur tot een soort kunstvorm.

3 Conclusies
Gepoogd architectuur los van context te zien  Om te beginnen is duidelijk dat ik dezelfde fout heb gemaakt die pogingen tot formalisatie van software architectuur ook gemaakt hebben: ik heb geprobeerd om architectuur, en met name stijl, los te weken uit zijn context. Daarnaast heb ik naar de ‘verkeerde zaken’ gezocht. Dimensies als interactiestijl en structuur hebben niks met kwaliteit te maken. Deze zijn noodzakelijk voor een eenduidige beschrijving van architectuurstijl, maar leven op een verkeerd abstractieniveau.

Hierbij moet opgemerkt worden dat het ‘anticiperen’ op veranderende wensen van de klant, en het plaatsen voorzieningen hiervoor in de architectuur, ook een taak van architecten is. Onvolledige informatie is vaak ‘part of the game’.

Adaptive Architectures – Resultaten discussiemiddag 2

---

1Hierbij moet opgemerkt worden dat het ‘anticiperen’ op veranderende wensen van de klant, en het plaatsen voorzieningen hiervoor in de architectuur, ook een taak van architecten is. Onvolledige informatie is vaak ‘part of the game’.
**Beperkte set kwaliteiten** Het lijkt aannemelijk dat het wegvallen van de ‘zichtbare’ eigenschappen van stijl een zeer beperkte set kwaliteitsfactoren overlaat. Aangezien dit weinig ‘uitvinding’ vereist, lijkt het verstandig om een bestaande standaard aan te nemen; meer hierover in paragraaf 4.

**Quality without a name** Hoewel de term enkel in de introductie met zoveel woorden genoemd is, is toch duidelijk geworden dat architecturen streven naar de quality without a name, die in dit verband ook wel “harmonie” genoemd is. Directe aanwijzingen hiervoor zijn te vinden in het pogen te verkrijgen van inzicht in de ‘krachten’ die invloed hebben op de architectuur, alvorens daar een stijl of structuur aan te koppelen.

Eén van de redenen die in de domeinstudie worden aangehaald om het gebrek aan publicaties over stijlkeuze voor gegeven problemen te verklaren, is het feit dat software architectuur meer ‘onder de oppervlakte’ leeft. (Dit in tegenstelling tot architectuur in de bouwkunst, waarbij stijl juist een zichtbaar gegeven is.) Dit wordt veroorzaakt door het feit dat software architectuur zich voornamelijk bezighoudt met niet-functionele eigenschappen, die nou eenmaal minder zichtbaar zijn. Daarnaast zijn deze in het product soms niet meer te herkennen, zoals onderhoudbaarheid.

**4 Follow up**

Zoals in paragraaf 1 is vermeld, heeft de discussiemiddag andere informatie opgeleverd dan waarop gerekend was. Dit heeft een zeker invloed op het verdere verloop van mijn master project.

**4.1 Voor het project**

Eén van de problemen in het project is de omvang van het vakgebied: er zijn vele verschillende interpretaties van software architectuur en stijl, met evenzovele definities, die soms overlappend, en soms wederzijds uitsluitend zijn. Ook is het lange tijd niet geheel duidelijk geweest waar ik ‘naar op zoek was’. Het is nu in elk geval duidelijk dat ik niet op zoek ben naar

- een methode om de intuïtie over de ‘load-bearing’ zaken van architecten te beschrijven, of
- een stelsel van ‘zichtbare’ eigenschappen van stijlen.

Ook het gebied waarin het project zich afspelt wordt steeds kleiner: door de aanwijzingen in de volgende paragraaf is een eenduidige definitie van ‘stijl’ vastgelegd.

**4.2 Voor de domeinstudie**

**Bestaand systeem kwaliteitsfactoren** Nu het aantal mogelijke kwaliteitsfactoren is teruggebracht tot voornamelijk niet-functionele eigenschappen, kan een bestaande set van kwaliteitsattributen worden aangenoemd. Deze sets worden normaal gezien gebruikt om systemen of architecturen te evalueren, maar kunnen met de nodige aanpassingen, en de juiste definitie van stijl, gebruikt worden om stijlen te evalueren.

De kwaliteitsfactoren zijn nu onder te brengen in vier groepen,

- [*Run-time eigenschappen*] Eigenschappen van een draaiend systeem, zoals performance of betrouwbaarheid,
- [*Statische eigenschappen*] Eigenschappen die betrekking hebben op het systeem ‘wanneer het stilstaat’, zoals onderhoudbaarheid,
- [*Architectuur eigenschappen*] Eigenschappen die het gebruik van een stijl beïnvloeden, zoals de vereiste ervaring van het ontwikkelteam en de ‘begrijpbaarheid’ van de stijl,
[Domein-specifieke eigenschappen] In tegenstelling tot de eerste drie groepen zal deze groep geen vaste attributen hebben. Bepaalde stijlen kunnen eigenschappen hebben die ze geschikt maken voor een gegeven domein.

Uitbreiden stijlbeschrijvingen met factoren Op het moment van schrijven zijn 11 stijlen beschreven in de domeinstudie. Deze stijlen zullen worden aangevuld met een evaluatie op basis van de kwaliteitsfactoren.

4.3 Voor een volgende sessie

Stijlvergelijkingen a.h.v. gekozen kwaliteitsfactoren Nu een raamwerk is gekozen voor de kwaliteitsfactoren, kan in een volgende sessie op een efficiëntere manier met de stijlvergelijkingen worden omgegaan. Hierbij valt te denken aan het aanbieden van de kwaliteitsfactoren, met daarbij de opdracht ‘punten’ te verdelen voor de verschillende eigenschappen in gegeven stijlen.

Probleemstellingen met meer context Nu is gebleken dat de nadruk in de eerste fase van een architectuurproces voornamelijk ligt op ‘wat betekent het systeem voor de organisatie’, kan deze kennis gebruikt worden om gerichter probleemstellingen aan te bieden. Het is bijvoorbeeld mogelijk om een uitgewerkte productvisie aan te bieden, waarna gezocht kan worden naar de voornaamste kwaliteiten die hierin worden verwoordt. Aan de hand van deze kwaliteiten kunnen vervolgens een aantal mogelijke stijlen geëvalueerd worden.

References


\[2\] Een leuk voorbeeld in dit verband is de ‘Iowa-bility’ [BCK98, p.281]. Hierin is een overweging dat een project wel ‘hip’ genoeg moet zijn om voldoende personeel aan te trekken in een bepaalde Centraal-Amerikaanse staat.
Appendix D

Results discussion 2
Adaptive Architectures
Results discussion 2
A. F. M. van der Sijpt
March 2007

Abstract
The second session of discussions for project Abstract Architectures dealt with two subjects. The discussion on software architecture confirms the current view that software architecture is a ‘tough’ problem. The discussion on adaptivity did not create directly usable results, since the chosen approach was too ‘philosophical’; instead, this discussion has given some inspiration for a different direction of research into adaptivity.

1 Introduction
A second session of discussions concerning project Adaptive Architectures, held March 9 2007, has yielded some noteworthy results. This document is not a transcription of the afternoon, but summarizes the most prominent observations. Interpretations of these, and their impact, appear only in the concluding sections.

2 On architecture
Accompanied by a short overview of the status of the project, the discussion based on statements regarding the nature of software architecture worked out pretty fine.

2.1 Key points

An engineer is not a carpenter A common misconception is viewing software engineers as the ‘makers’ of what others have designed, like a carpenter who builds a piece of furniture according to a design. This is not true: it is in the nature of software that designs are made at every level, potentially having an impact on seemingly unrelated parts of the system.

Transferring the architecture Just as important as creating the ‘right’ architecture, is transferring its essence to the developers, to get commitment for the ‘direction’ it dictates. The ‘noise’ originating from other levels in the project, possibly having a large impact on the system’s qualities, can be countered by creating this commitment; alignedness of the engineers’ views on the system is a trait of a good architecture.

The ‘good ways’ of software engineering There seem to be practices that every project can benefit from, which have been used since before supporting mechanisms existed; during the discussion, the example of object oriented design has been used.

To stick with this example, OO creates a way of thinking about a problem with powerful mechanisms. Even without articulation of these concepts, good designers have used them for a long time, but they are hard to convey without it.
On tools There is a general opinion that formal tools are necessary for (1) conveying knowledge and (2) abstracting away from details, allowing a focus on the core of the problem. Still, it is acknowledged that the formalisms at hand must be selected carefully.

Is software engineering really engineering? It is acknowledged that software engineering is not a mature engineering discipline. Two conflicting views try to explain this.

- The image of engineering in the ‘other’ fields is idealized to a vision of repeatable and predictable processes. However, engineering is not about repeatability: the word ‘engineer’ shares its etymological roots with ‘ingenious’; all engineering disciplines have the same problems with ‘new’ inventions.
- Software engineering is simply too young: when taking the ENIAC as the starting point of computer programming, it has been around for some 70 years. Other engineering disciplines are much older, so software engineering just needs time to mature.

Should an architect be able to code? Four views on this can be discerned, most of them refuting this claim.

- Being an architect is about so much more, that coding is actually not that important.
- It isn’t the inside of the product that matters, it’s the outside; that’s were the architect’s responsibilities lie, so coding is not that important.
- Civil architects are not required to know how to be a carpenter or a mason, so why should a software architect know how to code?¹
- Since we have no definite model of the way code ‘behaves’ (like those for e.g. concrete or steel), the only way to know the material is to use it. Therefore, an architect must have experience in coding and understand that code is a main product of a software engineering project.

2.2 General

Portraying my views In retrospect, the introduction has been too brief. I chose to drop information about some four months of work (i.e., the time it took to arrive at the current model of software qualities) in favor of a quick start and a ‘clean slate’ for the discussion. This caused an incomplete picture of ‘what it is that I am looking for’, which was eventually filled in after some discussion.

Main observation Two concepts keep popping up throughout the discussion, together indicating the ‘toughness’ of software architecture.

- [Mental matter] Unlike many other sciences, computer science is based on mental constructs alone. This makes it one of the most constructivist of all sciences: whatever can be imagined, can be constructed, since there is no ‘material’ imposing rules. From this, great freedom in constructing models arises, but there is no way of assessing the usefulness or qualities of these models; anything goes.

    The lack of experimental evidence of the correctness of these models creates an incomplete understanding of the nature of computer science.

¹Although in some parts of the world, architecture students are required to work on a building site during summer, doing manual labor.
• [A young discipline borrows terminology] Although the history of computation goes back thousands of years, computer programming can be traced back for at most 70 years.Originating from mechanical and electrical engineering on the one side, and mathematics on the other, it has ‘borrowed’ much terminology and paradigms from related fields. Apart from the aforementioned fields, another influence is civil engineering, carrying over terms as ‘building’ and ‘architect’. For some reason, making sense of software engineering requires analogies, but herein lies a problem: the fields supplying the borrowed elements have a long history, giving each term a lot of context and connotation, which may be well applicable to computer science.

The observation that, even among software architects, analogies to building pyramids and creating pharmacological compounds have been used, is a sign on the wall that we lack understanding of software architecture; as far as I can find out, making these analogies is rare in other disciplines.

The concepts above create a circle: because we lack understanding of the nature of computer science, we resort to analogies to other fields, carrying over terms with incorrect connotations, in turn leading to a cracked understanding of computer science, etc.

2.3 Conclusions

In general, my ‘direction of thought’ seems correct,

• there is a general ‘trait’ to architects which leads to working out the ‘right’ solutions, but it is not quite known where this trait comes from,
• architecture does matter, and can ‘set the stage’, but in the end, “great designs come from great designers”\(^2\),
• software architecture is hard: architecture strives for the ‘quality without a name’, and the field suffers from the characteristics of wicked problems,
• formalisms are necessary for conveying knowledge, but can also lead to fixing a designer in a single view of a problem.

Followup No radical changes have to be made to the planning of the project. The domain study can be finished using this information, ending up with an analysis of the ‘toughness’ of software architecture, instead of a neat and well-tuned model of architecture styles and qualities.

3 On adaptivity

The discussion on adaptivity has created some enjoyable discussion, and has yielded some extra insight for me, but in the end, no clear ‘direction of thought’ has been established.

In general, we can state that the discussion has been focused too much on the question ‘what is intelligence’, which is more of a philosophical statement. This lead to a discussion far away from the participants daily activities.

3.1 Key points

Even though the discussion yielded no very specific results, it is nice to note the subjects discussed.

• What’s in a name: intelligent, adaptive, flexible, robust. Where does one term end, and the other start?

\(^2\)Frederick P. Brooks Jr.: No Silver Bullet – Essence and Accidents of Software Engineering
• Two approaches in creating intelligent systems can be used: top-down, describing the high-level activities that people do (e.g. expert systems), or bottom-up, building from the fundamental blocks of nature (e.g. neural networks).

• What are the ‘edges’ of intelligence, where does intelligence end and rules start? Except for ‘intelligence is that which we don’t understand’, there is no real understanding of what makes a system intelligent.

• Is seeming intelligent the same as being intelligent? The general feeling is ‘no’, but this has been marked as an open issue.

• Sorting algorithms started out as a part of the field of AI, being a typical example of the top-down approach.

• Is generating random code, selected by fitness functions, a useful way to go?

3.2 Conclusion

There has been an enjoyable discussion, though starting from a wrong approach. It would have been better to start from practical examples, and generalize from there.

Followup  It seems I have been ‘pulling on the wrong end’ of adaptivity. It is indeed useful to have an understanding of the philosophical elements of adaptivity, but that is not were advancements can currently be made. An approach for ‘getting some grips’ on the subject can be to start working from the non-adaptive end of the spectrum, represented by monolithic systems, followed by ‘adaptable’ systems, etc. This can result in a ‘timeline’ of \( n^{th} \) order adaptive systems, similar to the ones we know for connectedness of systems.
Appendix E

An engineer is not a carpenter
An engineer is not a carpenter

A. F. M. van der Sijpt

June 2007

Abstract

We show that simple observations made about software architecture point to a set of symptoms and causes of the ‘hardness’ of software engineering, and software architecture in particular. We present a view on software architecture inspired by these challenges, and give recommendations for both averting them and using them to an advantage.

1 Introduction

The world in which software lives is becoming ever more dynamic. Software becomes more complex, devices that were previously driven by electronics are now controlled by software, and more software is required to ‘understand’ that the world is becoming networked. Handling this change from an architecture point of view was the focus of project Adaptive Architectures, as carried out at Luminis in 2006–2007.

We required a good understanding of what software architecture, and in particular style, is about. To get some feel for the matters at hand, a start-up exercise has been defined to create a system of software architecture styles, linking them to their quality factors (see Section 4 for a definition of quality factor).

Starting with an incomplete understanding due to lack of information in literature, this project ended up with just that, an understanding of (a) what is software architecture, and (b) why is it so hard to do it actually right.

This document will show some observations made during the project in Section 2. Taking together the observations, we arrive at a set of symptoms of the toughness of software architecture in Section 3 with their basic causes. Section 4 shows the resulting view on software architecture, after which Section 5 provides some recommendations for not only averting typical problems that stem from the observations made in this document, but to actually use them to an advantage.

2 Observations

48% of projects succeed [Whi99] shows that, for companies of varying sizes, between 15% and 30% of IT projects overran their budgets by more than 50%, and between 25% and 50% overran their schedules by more than 50%. Longer projects tend to overrun more: of projects planned to take less than a year, some 25% doubled their running time, and over 30% of those longer than a year did.

In a more recent publication, [ict07], 48% of IT projects succeed. However, of the remaining projects, 4% fail completely, with 48% having difficulties. Most often, the projects did not deliver what was expected (28%), overran their running time (23%) or budget (11%).

This shows that while more project end correctly, this still represents less than half of the endeavors. The projects that do not deliver as promised are likely to be an artifact of better management techniques: cutting down on functionality, instead of overrunning time or budget.
No consensus, no feuds  Although many fields of science will have at least some problems defining ‘what they are about’, this seems to be especially true in software architecture. This shows in a number of peculiar observations, which will not be a surprise to many practitioners.

It is not really clear what architecture is; a multitude of definitions has been defined by academics and governing bodies, but none of these is really accepted. The same applies to the job an architect does: although it seems logical that an architect is the one who creates the architecture, there is often much more to it, including gathering requirements, coaching engineers and management functions.

The definitions of architecture and the job of an architect have another notable element. The terminology has been borrowed from neighboring fields since the fifties, most notably hardware and civil engineering. Even among practitioners, analogies to neighboring fields are the main way to express views.

Furthermore, there is a ‘lack of sides’: although there are incompatible ideas out there (e.g., reductionism vs. holism), these do not have ranks of dedicated followers, that try to prove the other side is wrong (like, for instance, intuitionist logic). Instead, it seems that many practitioners do not really think about what side they’re on, and why.

The ‘ring’  With borrowing terminology, the professional connotation of term like architecture, construction and engineering get carried over. However, with the lack of accepted definitions, these titles are free to claim, and so happens.

Tangibility  The reductionist view is currently most popular, most likely due to the tangibility and direct applicability of this approach. Reductionism is about chopping up a problem into chunks, working together with defined interfaces and interactions. This is embodied in the box-and-line drawings present in software architecture documents. When focussing on structure, description methods have been devised, known as architecture description languages.

Other movements do exist. [SC97] shows a method for classifying architecture styles, acknowledging their potential as ‘helpers’ in architecting software. However, the same authors have to note in [SC06] that there have been no followers, and the same activities can still bring progress.

Potentially influential was [Fie00], in which the author is dissatisfied with contemporary views of software architecture, and develops a novel vision of software architecture as basis for his own research, breaking with the traditions of Perry & Wolf [PW92] and Parnas [Par72]. The main outcome of Fielding’s research, REST, has an avid following of ‘RESTafarians’; his founding work is all but ignored.

Further more, [BR98] shows our current ways of thinking about software architecture are insufficient, and not much more than structural and behavioral descriptions. Both Baragry’s doctoral thesis and a summary of that [BR01] have been largely ignored.

What do we do?  During the project I asked administrators of open source projects rather specific questions about the architectural choices made in their projects. I ended up with general good advice instead of the basic decisions I expected,

• “Decoupling is extremely important in open source projects to ensure minimal commit conflicts and also demarks areas of responsibility.”

• “Reliability and efficiency can be derived from [maintainability], clear code can be made more reliable and more efficient.”

These men represent some of the most successful open source projects, so there must be a reason for making these statements. After some more correspondence, I decided to ask some of them about the impact of software architecture. This time, I ended up with very different statements.

• “Anyone worth their salt was using object-oriented approaches long before the term object-orientation was coined, or explicit language support existed, for example”
• “qualities are inherent from the strategy [architecture], it is inside, but tactic [engineering]
is here to make it seen by others”

There definitely is something these people do right given the success of their projects. Yet, they have great trouble pinpointing what that is. I do not believe they simply do not want to tell me; I have spoken with a number of software architects from the commercial field and find the same feeling there.

3 Causes

The observations above show a pattern of ‘where things go wrong’. There seems to be a ‘cloud’ of symptoms that explain this phenomenon, all revolving around a ‘nucleus’ (more on that later on). The symptoms are not separate entities, they all influence each other.

Quality without a name In *The Timeless Way of Building*, Christopher Alexander [Ale79] coins the term *Quality without a name* in civil construction, but immediately shows its applicability in other fields that ‘have to do with people’.

The quality ‘names’ the properties that a building can have, which makes its user ‘feel alive’. This is very noticeable, but the actual properties are not identifiable. The quality is not directly engineerable. Alexander states that the quality is

“...generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from a seed.”

The generalized form of this quality applies to all fields of engineering. Software architectures strive for a state of ‘least conflict’ between opposing interests. During the discussion sessions, this has been articulated as ‘harmony’; ‘elegance’ comes close. Like in construction, it is not possible to identify elements of an architecture that are responsible for this elegance, but it shows throughout the product. This quality makes the difference between a solution that *works* and one that *fits the situation*.

Wickedness When looking at it from various angles, the essence of software architecture seems to shift shapes continuously. A problem keeps changing, and the proposed solution influences the shape of the problem. This might just be a wicked problem, as named by Rittel & Webber [RW73] in social planning. [vdS07] shows a full analysis of software architecture as a wicked subject; I will not go into too much detail here.

Three overarching properties of wicked problems can be identified, which can well be shown to apply to software architecture.

1. Describing the problem *is* the problem.
2. There is no stopping rule, it is never clear when the problem ‘has been solved’.
3. Project are constrained in time and money, leaving no opportunity to learn from mistakes.

Formalisms can blind Since the first days of science, formalisms and abstractions have helped practitioners focus on interesting matters, ignoring details. This has allowed tackling ever more complex problems, with essentially the same amount of intellect available.

With the complexity of large software engineering projects, formalisms can prove a problem of their own. Formalisms help abstract away from details, but it cannot be known whether these details are, indeed, of minor importance. This can lead to deliberate denial of potentially crucial factors, because they do not ‘fit’ the formalism.

However, with the right precautions [BH95], many formalisms can prove a very useful element of the architect’s toolbox. So, the problem is not the existence of formalisms as such, it is the possibility that formalisms get misused to cope with the increasing complexity of the problems we face.
Great designers In No Silver Bullet – Essence and Accidents of Software Engineering, Frederick Brooks [Bro86] states

“The central question in how to improve the software art centers, as it always has, on people. (...) Great designs come from great designers. Software construction is a creative process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge.”

Section 3.1 shows in a more explicit way that the human mind is a prerequisite input to the process. No matter how good the methodology is, how good the procedures are and how great the tools are, it requires great designers to end up with great designs.

Unconscious talents During the course of this project, I spoke many practitioners. All of them do something ‘right’ in the choices they make, given the position they are in today. However, when trying to find out what that is, I am not able to identify reasons for that. In Blink – The Power of Thinking Without Thinking, Malcolm Gladwell [Gla05] shows the process of ‘thin slicing’, taking split-second decisions with incomplete information with remarkably good results. It is about relying on the processing power of the subconscious to make decisions that will take a lot of time to rationalize, if they can be rationalized at all.

From Blink (emphasis mine),

“Braden has had a similar experience in his work with professional athletes. Over the years, he has made a point of talking to as many of the world’s top tennis players as possible, asking them questions about why and how they play the way they do, and invariably he comes away disappointed. “Out of all the research that we’ve done with top players, we haven’t found a single player who is consistent in knowing and explaining exactly what he does,” Braden says. “They give different answers at different times, or they have answers that simply are not meaningful.” One of the things he does, for instance, is videotape top tennis players and then digitize their movements, breaking them down frame by frame on a computer so that he knows, say, precisely how many degrees Pete Sampras rotates his shoulder on a cross-court backhand.”

Maybe what we call architecture is based on a wish to explain what happens in an architect’s head. Some professionals can be very good at what they do, but have no way of explaining, or even knowing, what they do. Our society wants to have explanations for everything, everything has to be traceable, and all men are created equal, so we want a rationalization of what people do.

This may explain the phenomenon encountered in contacting open source software administrators. They give general ‘good advice’, while not getting to the key issues. These are not dumb people; they represent the best of the open source software movement. Perhaps they just do not know what it is that makes them choose the right thing, they just do.

3.1 Core

It’s all in your head Plato stated that reality as we perceive it is not ‘raw’, but interpreted by our minds. We create an internal model that strives to explain (and predict) the things we see. New experiences that fit the model are accepted, whereas contradicting experiences either mean something is happening in our perception, or our model is not correct; M.C. Escher’s drawings are an example of this mechanism.

Science, too, creates models to explain observations, a “useful model of reality”. Consider physics as an example. Through the ages, theories have explained the effects objects have on one another. In the 17th century, Newton’s model of physics explained the way an apple falls down, and how a billiard ball transfers its energy to another when hitting it.

Newtonian physics worked well for several centuries. However, when studying small scale or high speed phenomena, it starts to break down. This is where quantum mechanics come in. This,
too, is of a constructivist nature: it explains observations, but we can never know whether the model actually represents reality.

In this respect, Einstein used the metaphor of a man faced with a clock he cannot open. He can construct a model explaining the movement of the hands and the ticking he hears, predicting every observable aspect of the clock with great accuracy. Still, he will never know what really happens inside [EB38].

Scientific models can be verified by experiments, as above, or by deriving new properties from accepted axioms.

In creating software, we model parts of reality in our artifacts. However, the artifact itself is no more than a collection of electrical currents; even the attribution of ones and zeroes to states of a transistor is an abstraction. All the things these bits represent only exist as a mental model; we take blobs of bits to mean numbers, objects, customers, processes.

The model can represent any reality, everything that can be imagined can be attributed to a formal construct, and we can create constructs that are arguably representations of all we can imagine. No experiments can verify the suitability of the model, nor can it be established that it is free of contradictions. On the other hand, no basic axioms exist that allow deriving a correct model.

In the end, in software, we can model any reality we can imagine, but we cannot verify its consistency; we simply have no way of exporting them from our mind in all their glory.

The semantic gap

The difference between two non-automatically translatable representations of an object is known as the semantic gap. In computer science, it usually means the difference between the problem’s description, and its code-counterpart. A typical example is known as the Halting Problem: this can readily be expressed in natural language, but a formal representation can be proven not to return the wanted output on all allowable inputs.

The Von Neumann-machine, which is the basis of all modern computers, is expressionally equivalent to a Turing machine. Elements of this architecture may be aggregated into higher level constructs, such as programming language statements or objects. These help in narrowing the semantic gap. However, these constructs can be computationally translated to machine code, meaning that high level languages are still Turing complete.

The semantic gap will always exist, no matter how high the level of abstraction in constructs becomes.

The circle

The elements above create a circle: in software, any reality can be created and used, but its correctness cannot be verified. This means the mental gap is bridged by non-verifiable constructs, keeping the gap alive since it would only be really closed once a Turing machine can bridge the gap, which cannot be done since the mental constructs are necessary; these cannot be formalized because of the semantic gap, etc.

The circle described above is unlikely to be broken anytime soon. Both elements are caused by the fact that we do not understand how the human mind works (“The mind is a strange and wonderful thing, I’m not sure it’ll ever be able to figure itself out. Everything else, maybe, from the atom to the universe, everything except itself.”). The only way of breaking the circle is finding this understanding and modeling (the relevant aspects of) the human mind in a Turing machine. Note that some believe that human-like intelligence will arise once computer processors get powerful enough.

Until that moment, there is only one tool that can effectively bridge the semantic gap: a human mind able to use sufficiently complex mental constructs, so it has no need to resort to over-abstraction.

---

1 [http://en.wikipedia.org/wiki/Halting_problem](http://en.wikipedia.org/wiki/Halting_problem), “Given a description of a program and a finite input, decide whether the program finishes running or will run forever, given that input.”

2 From *Invasion of the Body Snatchers*.

3 For instance, Arthur Koestler’s *Ghost in the machine*, which argues the human brain is machine, building upon primitive structures, and ‘awakening’ when its complexity is sufficiently high.
Figure 1: Symptoms and core problems.

3.2 Relations

This document has shown a number of observations, the five symptoms they ‘break’ on, and two core elements causes this. Figure 1 shows the symptoms and their core.

Figure 1 can be seen as a cross-section of a sphere, in which the sphere represents the ‘hard elements’ of software architecture. This particular cross-section represent the approach of “using quality factors to make sense of software architecture”. Other approaches will create a different cross-section of the sphere, possibly encountering other symptoms. I believe the core is common for all parts of software engineering.

4 Now, what is software architecture?

4.1 Introduction

Software architecture uses borrowed terminology, placing the architect in a role where he is the one that envisions the entire system, and is occupied with creating its high-level structure. Like this is not entirely true in construction, I believe it is not in software either.

Architecture is about making a system fit its context. This applies to the system as a whole; I do not believe architecture is primarily about structure and ‘chopping up’ a problem into manageable pieces [Par72]. Rather, it is about cohesion and alignment.

In construction, blueprints and maquettes are manifestations of the architecture. The actual choices the architect makes are about the ambiance of a building; decisions like ‘I don’t want any corridors ending in three doors’. These embody unshakeable constraints on the architecture, stating what is required to make the product fit its context. Note that this can include, but is not limited to, structure.

While the architectural choices above are very articulate, it is possible that they are more ‘hidden’. They will manifest themselves in the resulting artifacts, but cannot be articulated as such. Still, they are part of the architecture, and can find their ways into elements like rationale.

These views have been inspired mainly by two authors. First, the idea that software is ‘something completely different’ came from experience during the project, and turns out to have been developed by Jason Baragry [BR98, BR01]. The views on architecture and the resulting artifact have been inspired by Fielding [Fie00].

4.2 A model

An architecture is a set of concepts, guidelines for creating the resulting artifacts. This idea puts styles on the same level of abstraction as architectures, merely a collection of concepts, brought together with a convenient name.

Both styles and architectures can support certain quality factors, given that the implementation does not break them. A system can only exhibit desirable non-functional properties when both
the architectural support and engineered implementation are in place. The artifacts that embody
the architecture give rise to the system. This system will exhibit measurable, non-functional
properties. This cohesion is shown in Figure 2.

In this figure, dashed lines represent the named relations of the previous paragraph. Cardi-
nalities are apparent, and have been left out. Two relations are marked with a question mark.
The link between concepts and quality factors is what this project tried to uncover. The second
questionmarked connection is between non-functional properties and quality factors. This projects
has assumed them to be the same. I believe they actually are the same, or at least one-on-one
mappings can be formulated.

4.3 The components

**Quality factor** “Quality factors are the (non-)functional properties that an architecture can
support, given an upholding of the architecture’s concepts in all relevant artifacts.”

**Architecture** “An architecture is a set of concepts, brought together to make the product fit
its context.”

**Style** Following Perry & Wolf’s view, a style is a less specific architecture.

“A style is a collection of concepts to be called by a convenient name, possibly accom-
panied by a rationale for this combination of concepts.”

**Concept** The most intangible element of this model is the concept. These embody the choices
made by the architect, the leading principles of an architecture. The introduction has shown that
concepts in construction are decisions that could be applied to all buildings, but have been chosen
to be included in just this situation.

“A concept is a fundamental choice—articulated in a pattern, guideline or even some
examples—making the product fit its context, and should be recognizable throughout
the product.”

This principle means that the key concepts making up the architecture should fit on no more
than a few pages. However, rationale and elaboration of the concepts can take up quite some more
paper, but are not considered part of the architecture.
Artifact  Artifacts are the embodiment of the architecture, tangible ‘stuff’ that shows the concepts the architecture is made of. This can include documents, but also code. In all of these, the influence of the architecture has to be held up.

The element that is most commonly seen as ‘the’ architecture, the system’s structure with modules and interfaces, requires some extra attention. When the context requires a fixed structure, this can very well be a concept; however, when other concepts dictate a some structure, it will only be part of the artifacts. Furthermore, in systems with a dynamic structure, it will not even be part of any document, but will be dictated at runtime by governing concepts.

System  The system is the physical manifestation described in the artifacts, i.e., the system behavior.

Non-functional  The non-functional properties of a system indicate ‘how well’ a system does its job.

5  Conclusion

With the problems shown in Section 2, it seems a miracle that so many project actually end well. Granted, not all is bad. We work in a field that attracts bright minds, and often they get their say. However, [Whi99] shows that larger projects have a greater chance of failing. It seems our approaches that work reasonably well, have problems scaling up.

5.1  An engineer is not a carpenter

There are some important aspects that sets an engineer apart from a carpenter; these properties apply to all who have to do with creating software, including architects.

Software is not wood  The tools a software engineer has to work with are largely mental (Section 3.1), though aides have been created to help the engineer visualize his ideas.

Laws of gravity  Artifacts that are created in real life can be touched and handled. When a carpenter builds something, it will soon be apparent when something is crooked. Due to the mental aspects of software, we do not have this luxury.

A module is not a wall  When a contractor orders a mason to build a wall, he has fairly good idea what it will look like when finished. When an IT architect hands over responsibility for a module to an engineer, he has no idea what the engineer will come up with; perhaps, there are some ideas, but the same problems can likely be solved in many ways.

This last difference brings us to a core observation: in construction, one man designs a solution, and the other builds it. We consider the man designing the solution, the architect, to be the creative man, and the man who builds it, the mason, to be the laborer. In software, designing the solution is the solution; the computer is the laborer.

5.2  Recommendations

Be aware  [vdS07] shows that software architecture can reasonably be classified as a wicked problem. So, the same recommendation from [RW73] can be used here: be aware. Be aware of the typical pitfalls that are situated in the field of software architecture. Be aware of the difficulties that are described in Section 3, and learn to recognize them.

An engineer is not a carpenter
Trust good people  And finally, be aware that creating software is a creative, human activity. Only the human mind can bridge the semantic gap. Tools can help the mind in this, but they might also obscure the gap, leading us to believe it has disappeared. No single methodology works for all systems; specific methodologies might prove useful for very specific domains, but even then there is a semantic gap that needs to be closed. A good understanding of ‘what we are doing’ is indispensable.

It has been quoted earlier, but [Bro86]

great designs come from great designers

Creating ‘good’ stuff requires a keen mind that is aware of the problems it might face. This means that sets of competencies do not make people replaceable. The study has shown that good architects have some traits in common; unfortunately, these seem to be hidden in the subconscious, making it impossible to directly screen people for having ‘great designer’-genes.

Still, what is it that makes these great designers? Based on what I’ve seen and heard in the past months, I don’t believe a typical background can be pointed out. Skills taught at universities are indeed useful, if only in training the mind in understanding complex things; however, controlled case work usually delineates the world of the problem very strict, while exactly this framing is a skill of a great designer. The only binding factors for great designers now seem to be a keen mind, a great amount of ‘understanding’. From that point on, as Brooks states, the designers must be ‘grown’.

References


Appendix F

Software architecture as a wicked problem
Software architecture as a wicked problem

A. F. M. van der Sijpt

February 2007

1 Introduction

During my Master’s project, the feeling arose that software architecture is a typical ‘wicked problem’. This was not the goal of the project, though it could make a nice conclusion for the thesis. This document contains a full analysis of the wickedness of the activity of software architecture.

Section 2 explains the nature of wicked problems, Section 3 gives a characterization of software architecture, and Section 4 maps these two field. Section 5 shows some conclusions and suggestions.

2 On wicked problems

In 1973, Rittel and Webber published ‘Dilemmas in a General Theory of Planning’ [RW73], introducing the notion of wicked problems in social planning. The core of the argument is the lack of optimal solutions for policy issues in social areas such as education, public health care and urban planning.

The social problems intended by the authors have a number of defining characteristics, some of which are inherent to today’s complex society, in which there is no undisputable public good.

A good example is the ‘fairness’ of health insurance contributions. Is it fair to charge an equal sum from everyone? Should the contribution be a fixed percentage of one’s income? Should people with a healthy lifestyle pay less, or those with chronic illnesses pay more? Clearly, no single solution, not even a balanced mix of ingredients, will make everyone happy.

‘Not having a definite solution’ is one of the characteristics that makes a wicked problem. This distinguishes wicked problem from hard (or complicated) ones: creating an algorithm for deciding primality of a number in polynomial time (the AKS primality test) is a complicated task, but its goal is very clear. In the health-insurance example, not even the problem itself can be defined in a definite way: how does one define ‘fair’?

Besides, the problem might not be self-contained: changing people’s social security situation might have implications on their lifestyle, influencing the number of illnesses in the population. Even more indirect effects may happen. It is not unimaginable that charging more for health insurance will bring people into financial difficulties, resulting in a slowdown of the economy and an increase of stress, leading to increased mortality rates younger age due to heart-related diseases, followed by a decline in demand for geriatric services. This indicates that resolving a problem in a given way may have unrelated effects, since the problem itself is a symptom of another one.

Note the use of resolution instead of solution. The authors use this term to indicate that proposed actions do not necessarily solve the problem at hand, but seem to be adequate given the current limitations and insights.

2.1 Criticism

The term ‘wickedness’ has been adopted in very diverse fields, often without justification beyond ‘this problem has a social context’. Although I do reject some statements from [RW73], such as the classification of ‘provide housing for the majority of the population’ as a tame problem or the
notion that problems should be defined as a ‘desired condition’, I do believe that the characteristics
give a good representation of what makes certain problems ‘look different every time you look at
them’. Following the characteristics, I believe that some problems cannot be solved in a formal
manner, but only ‘pretty good’ or ‘sufficiently correct’ resolutions can be sketched.

Apart from many followers, there has been some criticism in the past. One notable example has
been published in the same journal as [RW73].

Archie Bahm accuses Rittel and Webber of ‘hiding’ (their) incompetence behind the notion of
wicked problems [Bah75]. His main claim is that it is ‘easier’ to blame the problem than to blame
the practitioner. He believes the incompetence is caused by lack of a philosophical basis of science
and social issues present in our society.

As such, he may be right. There indeed is a lack of philosophical understanding of the fabric of
science, both in our society, and in the professional and academic realms. However, his claim that
wicked problems are a phantom created by incompetence overshoots its goal. Rittel and Webber
do not claim that classifying a problem as wicked is a justification for giving up, as is Bahm’s
interpretation. Rather, they notify the reader of the existence of a class of problems which are
susceptible to over-simplification.

As stated in Section 2, ten characteristics are provided in [RW73]. Bahm strengthens his claim
by refuting six of them. He chooses to provide counter-examples: problems classified as wicked by
Rittel and Webber, for which he shows that a given claim does not hold.

However, he uses different examples for all claims. Thus, while refuting a claim for some prob-
lem, he refuses to make any statements about the other claims, which just might hold for that
problem. By including the observation that not all wicked problems should have all character-
istics\(^1\), it can safely be stated he refuted some claims for some problems, but certainly not all
claims for all problems.

3 On software architecture

3.1 What is architecture, anyway?

The term *architecture* has been used in construction for millennia\(^2\), referring to the science and
art of designing structures for human use. Over the ages, architecture evolved from a method of
joining needs and means (e.g., need for housing and the availability of sticks and leaves) to the
current profession of designing structures in relation to their environment.

In a more general setting, architecture refers to *translating a vision, based on needs and con-
straints, into artifacts which can be used as a basis for further development*; this gives architecture
a significant subjective component.

Ever since programming became a discipline of increasing scope and complexity, terms as ‘struc-
ture’, ‘design’ and ‘architecture’ have been used to describe a high-level overview of the system at
hand. John Zachman is one of the first to give a solid analysis of the similarities between archi-
tecture in construction and architecture in software [Zac87]. He presents two cases of architecture
(building a house and constructing an airplane), followed by a mapping to the field of information
systems.

Software engineering holds a great promise of reuse. Even at the level of architecture, designs
can be reused for similar problems, e.g. an online store will have different characteristics each time,
but after developing some, patterns start to emerge. From this point on, a standard architecture
can be reused and tuned for each application. The architecture process Zachmann considers, only
applies to the first iteration of such a process, although the line between the two is fuzzy.

\(^1\)It can be agreed that a problem statement in the form “that this nation should commit itself to achieving the
goal, before this decade is out, of landing a man on the moon and returning him safely to earth” has some wicked
elements, but it certainly has a stopping rule.

\(^2\)First in ancient Greece, meaning ‘a master builder’.
Software architecture is hardly the only architecture out there. All fields dealing with creating complex structures rely on a form of architecture. Information systems, enterprises, buildings, tools, ... can all have a high-level view which allows relating the artifact to its environment.

### 3.2 Software architecture in a social context

In a sense, the (software) architect is an interpreter, working between the users (stakeholders) and the builders of a system. Just as an interpreter in a courtroom needs to have knowledge of both languages, the architect needs knowledge of both fields, each having its own ‘languages’. He must understand the needs (and potential future needs) of the client, and the possibilities (and limitations) of the builders. Combine this with stakeholders wanting the best solution for the lowest cost in time and money, and we end up with a complex system of inherently contradictory interests.

However, not all interests are directly related to the problem the solution is intended to solve. Many social factors enter the stage, like company politics. It is the architect’s job to find an optimal balance of these interests, even though most of which cannot be described by some formal method; many interests are not even outspoken or ‘visible’.

### 3.3 What problem?

Many design methodologies start from a well-defined problem, and from that point on, work toward a solution in well-defined steps; this can be automated pretty well. However, the complexity described above shows that *describing the problem is the problem*. It is also hard to ‘understand’ the problem; it may well be possible to pinpoint some ‘load-bearing’ factors of the problem (which is an art in itself), it is not possible to exhaustively sum up all the influences a solution should try to balance.

Instead of trying a formal approach, an architect relies on experience and intuition (which, arguably, may be the same thing, just on a different level of consciousness). This, still, does not lead to a full understanding of the problem: rather, it leads to a mental model of a *solution* to the problem at hand.

### 3.4 What solution?

As stated before, it is impossible to word an architecture problem in all its richness. The reverse applies to a solution, however: since a solution is turned into a tangible artifact (likely, code), all details have been sorted out. However, there is no definite method to prove that a given solution is actually the *best* solution under the given circumstances, or even that it does actually resolve the problem.

Methods for evaluating a given set of solutions do exist; still, these only include *given* solutions. They will return the solution *from the set* that is likely to achieve the best balance among the *named* interests.

### 4 Software architecture as a wicked problem?

Rittel and Webber provide ten characteristics for wicked problems. Upon closer inspection, a number of these overlap. They will be grouped into three broad categories, which will be shown to apply to problems of software architecture.

This grouping is inspired by the observation that many characteristics are caused by the same source.

---

3The ‘creation’ is a key element: many other fields of science deal with complex structures, but feel no need of describing an architecture. Examples include astronomy and medicine.
Not all wicked problems have all characteristics; I believe any problem that has one or more characteristics from each category can be safely classified as a wicked problem, and should be dealt with accordingly.

4.1 On formulation and interpretation

Two characteristics have a notion of ‘problematic formulation’, being (1) “There is no definitive solution to a wicked problem.” and (9) “The existence of a discrepancy representing a wicked problem can be explained in numerous ways. The choice of explanation determines the nature of the problem’s resolution.”.

Software architecture problems, as described in Section 3.3, are usually open for many different interpretations. A courtroom interpreter may translate (intentional) ambiguities from one natural language to another. An architect translates from a natural language domain, including unspoken elements, to a formal domain, usually resulting in software. Hence, the ambiguities must be settled.

Certain ambiguities may not be caught by the architect, and end up as an unresolved issue in the specification of the problem. To prevent this, formalisms are used that describe the problem without any ambiguities. These abstract away from some details of the problem, overcoming part of the problems’ complexity, thus allowing the architect to concentrate on the left-over core of the problem. However, herein lies a more grave situation: formalisms may both “guide and blind”. The ‘guide’ part is clear: formalisms provide a certain way of thinking about a problem, proposing an interpretation, that can ‘incidently’ be expressed in that formalism. This guidance also ‘blinds’: once a certain avenue of thought has been chosen, it is hard to deviate from that and consider alternative interpretations. This chosen avenue brands certain issues as ‘not important’. They are likely to be forgotten about altogether, since formalisms have no way of tracking the elements they intend to abstract! They are being ignored, possibly without knowing their significance or impact.

With a fully understood problem, these problems are likely not to occur; a good designer will spot these pitfalls, and choose appropriate formalisms. With the class of problems considered here, however, the problem is only fully understood when a formalized version has been created, or, in essence, an avenue for solving the problem has been chosen. The interpretation can then be heavily influenced by the formalism. Hence, describing the problem is the problem.

4.2 On completion and correctness

Knowing when to stop solving a problem, the assessment of a solution’s correctness, and the ‘borders’ of problems are handled by no less than 5 characteristics; (2) “Wicked problems have no stopping rule.”, (3) “Solution to wicked problems are not true-or-false, but good-or-bad.”, (4) “There is no immediate and no ultimate test of a solution to a wicked problem.”, (6) “Wicked problems do not have an enumerable (or an exhaustively describable) set of potential solutions, nor is there a well-described set of permissible operations that may be incorporated into the plan” and (8) “Every wicked problem can be considered to be a symptom of another problem.”.

Software engineering projects are rarely set up for their own sake. Software is a means of doing, or at least supporting, business. On the other hand, software has its own influence on the way business is done: an automated system can track purchasing behavior of customers, or make more complicated reward-programs possible. More directly, e-business, the delivery of services and goods using the internet, has its influences on the requirements for doing business.

This means that software is a player in the field of forces that influence each other, together making up an organization. Consequences of this are requirements that may change regularly, a group of stakeholders with very diverse or even contradictory needs, and a theater of operation with changing bounds and rules.

Combining this with the inherent intellectual freedom a software architect has leads to a highly dynamic context. With equal material and personnel costs, many different tools can be used.
for resolving the problem at hand, including both tangible tools, such as software development environments, and mental tools, such as description techniques or software paradigms. It cannot be predicted which tools will yield the best results. Furthermore, no tools exist which can assess the adequacy of a given resolution.

4.3 One shot endeavors

Another group of characteristics deals with constraints and the size of implication of a potential solution, comprising (5) “Every solution to a wicked problem is a ‘one-shot operation’; because there is no opportunity to learn by trial-and-error, every attempt counts significantly.”, (7) “Every wicked problem is essentially unique.” and (10) “The planner has no right to be wrong.”.

Given enough time and resources, it should be possible to devise an optimal architecture, being most acceptable to all stakeholders. However, innovative software projects have in common that they are on a tight time- and money budget, and have a potentially large impact on the organization(s) involved. So, possibilities for ‘trying out’ solutions are few; potentially beneficial, but risky, propositions cannot be explored.

While finding ‘load-bearing influences’ is one of the main traits of an architect, the uniqueness of innovative projects creates opportunities for important factors to stay hidden for a very long time. Choosing a formalism or direction of investigation early on in the project creates even more, as does the tendency of finding analogies to earlier projects.

5 Conclusion

From the statements above, I believe it safe to classify software architecture, or at least ‘first iteration’ architecture, as a wicked problem. Both ignoring this and taking it overly serious can have adverse effects: the consequences of ignoring have been shown above, but overestimating its impact can lead to a ‘formalizing does no good, let’s only trust our gut feel’ attitude. A combination of sensible use of formalisms and the awareness that some details might get lost, and should therefore be checked regularly, seems to be a good way of ‘resolving’ these problems. This is nothing new; for example, [BH95] pleads the use of formal mechanisms, but warns the reader not to be blinded by them.

No matter the great tools used, software architecture is still a human activity. Instead of trying to factor this out, making good use of this is much more sensible. We should not try to develop techniques that makes human intellect obsolete. Instead, we should make use of the powerful possibilities the human mind has, and augment this with tools that allow a designer to focus on certain essential elements of the problem at hand.

References


Appendix G

Typing report

The typing report at the end of this appendix is intended to record selected properties of a software system or software architecture style. These properties are about both the physical, ‘engineerable’ properties of the system, and the qualities it exhibits in both operation and maintenance. Section G.1 defines the terms used in the report, Section G.2 provides some guidelines for usage.

G.1 Quality factors

The report uses terminology borrowed from a number of theories, which may not be generally known. Below, the terms used are explained, starting at the bottom layer.

G.1.1 Physical properties

Physical properties characterize the conceptual shape and dynamics of the system.

- [Constituents]
  - [Components] By what ‘name’ do we call the active components of this style? Examples can be clients, filters, etc.
  - [Connectors] By what mechanisms do the components communicate? Examples can be shared data, remote method invocation, data streams, etc.

- [Control issues] The subject of control flow handles the way components work together, i.e., request others’ assistance, delegate work units, etc.
  - [Topology] Which elements invoke functionality in others, and what ‘form’ does it take when this is drawn? Examples are sequential programs with subroutines (creating a hierarchical topology) or client-server systems (having a star-shaped topology).
  - [Synchronicity] In what way do elements synchronize their actions? The following types are proposed,
    * sequential lockstep, every other components’ state can be derived from a single component’s state,
* parallel lockstep, like synchronous lockstep but with more potential threads of control; still, all elements have a rough idea what others are doing,
* synchronous, synchronizations between elements happen regularly, but not necessarily always with the same partner,
* asynchronous, synchronization based on message passing,
* opportunistic, communications only when necessary, not to reach some synchronization between components; typical for agent-based systems.

– [Binding time] At what time is the ‘other side’ of a cooperation known?

• [Data issues] How does data move around the system?
  – [Topology] The shape of data flow, similar to the flow of control above.
  – [Continuity] Does data flow through the system continually, or in bursts?
  – [Mode] What mechanism is used to allow data flow through the system? The following types are proposed,
    * passed, message passing,
    * shared, using a shared store,
    * copy-out-copy-in, shared, but with discrete modifications,
    * broadcast,
    * multicast.
  – [Binding time] When is the partner found for a data transfer?

• [Control/data interaction] How do data– and control-flow influence each other?
  – [Shape] Are the shapes of the data– and the control-flow similar?
  – [Directionality] Does data flow in the same direction as control does?

G.1.2 External factors

External factors are measurable in a running system. These can be ‘interesting for the user’.

• [Reliability] “A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.”
  – [Maturity] “The capability of the software product to avoid failures as a result in the software.”
  – [Fault tolerance] “The capability of the software product to maintain a specified level of performance in cases of software faults or infringement of its specified interface.”
  – [Recoverability] “The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure.”

• [Efficiency] “A set of attributes that bear on the relationship between the software’s performance and the amount of resources used under stated conditions.”
  – [Time behavior] “The capability of the software product to provide appropriate response and processing time and throughput rates when performing its function.”
  – [Resource behavior] “The attributes of software related with measuring the amount of resources required to perform its function.”
G.1.3 Internal factors

Internal factors can be recognized in the design of a system, factors which are interesting for the developer/maintainer.

- **[Maintainability]** “A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adaptations) of software to environmental changes and changes in the requirements and functional specifications.”
  - **[Analyzeability]** “The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or the parts to be modified to be identified.”
  - **[Changeability]** “The capability of the software product to enable a specified modification to be implemented.”
  - **[Stability]** “The capability of the software product to avoid unexpected effects from modifications of the software.”
  - **[Testability]** “The capability of the software product to enable modified software to be validated.”

- **[Portability]** “A set of attributes that bear on the ability of software to be transferred from one environment to another (this includes the organizational, hardware or software environment).”
  - **[Adaptability]** “The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for [by] the software considered.”
  - **[Installability]** “The capability of the software product to be installed in specified environment.”
  - **[Replaceability]** “The capability of the software product to be used in place of another specified product for the same purpose in the same environment.”

G.2 Usage

The report can be used for typing either a system or a style. For both, it is best to fill in the model starting from the bottom, ticking zero or more boxes per question. Some factors can be of no interest to a system or style, and can be left open. It is typical for styles to support multiple answers in the Physical factors.

- First, fill out the Physical factors, but stick to the concepts used in the system. The actual manner of implementation is not that interesting, e.g., using packet-based communication, a conceptual stream (continuous) can be created, and a streaming channel can be used to deliver bursts of information (sporadic). This is easier for styles.
  
  For each question, a number of common types are proposed; it is always possible to use one that is not mentioned here.

- The two layers above, External and Internal factors relate to the qualities the system exhibits, ‘how well does it do its job’.
• When typing a system, the top layer *Architectural factors* is of no interest; this only applies to styles. Anything that sets a style apart from others, and are incentives for (not) using a given style, are entered here. This may include the style’s complexity, or endorsement by a standards organization.

The empty space in the left column can be used for so-called *domain specific* factors. This is reserved for properties that are not mentioned in their respective layer, but are of interest to the system or style, even though they could be a non-issue for others.

Note that internal and external properties can ‘leak’ to the other layer: for instance, maintainability is usually an internal property, but it may be an external one in a system that has been explicitly engineered to allow updates.
| Style |
| Architectural factors |
| … | Desirable | Undesirable |
| … | Desirable | Undesirable |
| … | Desirable | Undesirable |
| … | Desirable | Undesirable |
| Internal factors |
| Maintainability |
| Analyzability | Desirable | Undesirable |
| Changeability | Desirable | Undesirable |
| Testability | Desirable | Undesirable |
| Portability |
| Adaptability | Desirable | Undesirable |
| Installability | Desirable | Undesirable |
| Replaceability | Desirable | Undesirable |
| External factors |
| Reliability |
| Maturity | Desirable | Undesirable |
| Fault tolerance | Desirable | Undesirable |
| Recoverability | Desirable | Undesirable |
| Efficiency |
| Time behavior | Desirable | Undesirable |
| Resource behavior | Desirable | Undesirable |
| Physical factors |
| Components | Connectors |
| o | o |
| o | o |
| o | o |
| Control issues |
| Topology | Synchronicity | Binding time |
| Linear | Seq. Lockstep | Write |
| Acyclic | Par. Lockstep | Compile |
| Hierarchic | Synchron. | Invocation |
| Star | Asynchron. | Run-time |
| Arbitrary | Opportunis. | |
| Data issues |
| Topology | Continuity | Mode | Binding time |
| Linear | Continuous | Passed | Write |
| Acyclic | Sporadic | Shared | Compile |
| Hierarchic | --- And --- | Cp-in-cp-out | Invocation |
| Star | High-volume | Broadcast | Run-time |
| Arbitrary | Low-volume | Multicast | |
| Control/data interaction |
| Same shape? | Same direction? |
| Yes | Yes |
| No | No |